M223
Zusammenfassung

Flavio De Roni
Uferweg 27
6014 Littau

27.12.2009

© by Flavio De Roni aka MrF2theD

Inhaltsverzeichnis

UML **5**

KLASSENDIAGRAMM 6
Assoziation *6*
Aggregation *6*
Komposition *6*
Generalisierung/Spezialisierung *6*
SEQUENZDIAGRAMM 7
Lebenslinie (Lifeline) *7*
Nachricht (Message) *7*
Interaktionsrahmen (Interaction Frame) *7*
Kombinierte Fragmente (Combined Fragments) *7*
ANWENDUNGSFALLDIAGRAMM 8
Systemgrenze *8*
Akteur *8*
Anwendungsfall *8*
Assoziation *8*
Generalisierung/Spezialisierung *8*
Include-Beziehung *8*
Extend-Beziehung *8*
AKTIVITÄTSDIAGRAMM 9
Aktion *9*
Kontrollfluss *9*
Aktivitätsbereich *9*
Objektknoten und Objektfluss *9*
Signal-Sendung und Signal-Empfang *9*
Aktivität *9*
Start- und Endknoten *9*
Entscheidungs- und Verbindungsknoten *9*
Gabelung und Vereinigung *9*
Schleifenknoten *9*
Bedingungsknoten *9*
Unterbrechungsbereich *9*
ZUSTANDSDIAGRAMM 10
Zustand *10*
Event und Transition *10*
Startzustand, Endzustand und Terminator *10*
Entscheidung und Kreuzung *10*
Zusammengesetzter Zustand *10*
Region *10*
Rahmen eines Zustandsautomaten *10*
Generalisierung/Spezialisierung *10*

ENTWURFSMUSTER **11**

OOP-PRINZIPIEN 11
STRATEGY 12
Verwendung *12*
Beispiel *12*
Vorteile *12*
Nachteile *12*
Code *12*
OBSERVER 13
Verwendung *13*
Vorteile *13*
Nachteile *13*
Beispiel *14*
Code *14*
DECORATOR 15
Eigenschaften *15*

Verwendung ... *15*
Akteure ... *15*
Vor- und Nachteile ... *15*
Code ... *16*
FACTORY ... 17
Vor- und Nachteile von statischen gegenüber Nichtstatischen Fabriken? ... *17*
FABRIKMETHODE ... 17
Verwendung ... *17*
Akteure ... *17*
Vorteile ... *17*
Nachteile ... *17*
EinfacheFabrik - Code ... *18*
Statische Fabrikmethode ... *18*
Fabrikmethode ... *19*
ABSTRAKTE FABRIK ... 20
Verwendung ... *20*
Akteure ... *20*
Vorteile ... *20*
Nachteile ... *20*
Beispiel ... *20*

ZUSAMMENGESETZTE MUSTER ... 21

MVC ... 21
Model (M) ... *21*
View (V) ... *21*
Controller (C) ... *21*
Entwurfsmuster ... *21*
Ablauf ... *21*

JAVA UND APACHE-TOMCAT ... 22

BEGRIFFE ... 22
Webserver ... *22*
Apache-Tomcat ... *22*
JSP ... *22*
CGI ... *22*
Servlet ... *22*
WEBANWENDUNGEN ... 23
Was ist TomCat? ... *23*
Client → Webserver → Tomcat ... *23*
Servlet-Namen ... *23*
J2EE ... *23*
MVC ... *23*
J2EE-Anwendungsserver ... *23*

MULTITASKING ... 24

ALLGEMEIN ... 24
Synchrone und asynchrone Ausführung von Programmcode (Multithreading) ... *24*
Wann Multithreading verwenden? ... *24*
Grundbegriffe zum Thema Threads ... *24*
Kontextwechsel ... *24*
Thread-Priorität ... *25*
Thread-Zustände ... *25*
MULTITHREADING BEI .NET ... 25
Möglichkeiten des Multithreadings bei .NET ... *25*
SPERREN UND SYNCHRONISIEREN VON THREADS ... 26
Möglichkeiten der Synchronisierung bei .NET ... *26*
lock() ... *26*
Monitor ... *26*
Mutex ... *27*
Semaphor ... *27*

INTERPROZESS-KOMMUNIKATION ... 27

ALLGEMEIN 27
Datenströme *27*
Ereignisse *27*
Funktionsaufrufe *27*
Shared Memory *27*
.NET REMOTING 27
Proxyobjekte *27*
Channel *27*
verteiltes Objekt *28*
GLOSSAR 28
RPC *28*
Corba *28*
DCOM *28*
DCOP *28*
DDE *28*
SOAP *28*
Webservice *28*
SOA *28*
Remoting *28*
RMI *28*
MEHRSCHICHTIGE SOFTWAREARCHITEKTUR 28
ALLGEMEIN 28
3-schichtige Architektur *28*
2-schichtige Architektur *29*
Client-Server *29*
CLIENT-SERVER-ARCHITEKTUR 29
Die 7 Ebenen *29*

UML

Unified Modelling Language

UML definiert Gestaltungsrichtlinien zur Beschreibung (Spezifikation, Visualisierung, Konstruktion und Dokumentation) von Softwaresystemen.

Vorteile von UML:

- Eindeutigkeit:
  Notationselemente besitzen eine präzise Semantik und sind von vielen Experten definiert und geprüft

- Standardisierung & Akzeptanz:
  weltweit ist die UML in der Software-Branche im Einsatz

- Plattform- und Sprachunabhängigkeit:
  Softwaresysteme für jede Plattform modellierbar

- Unabhängigkeit von Vorgehensmodellen:
  den Software-Entwicklern wird diese Entscheidung selbst überlassen

Strukturdiagramme (Structure Diagrams)
modellieren statische, zeitunabhängige Elemente eines Systems.

- Klassendiagramm → Bauplan für Objekte
- Objektdiagramm → Momentaufnahme eines Systems
- Paketdiagramm → organisieren UML-Elemente in Pakete

Verhaltensdiagramme (Behaviour Diagrams)
modellieren dynamische, zeitabhängige Aspekte des Systems

- Anwendungsfalldiagramm → Beziehung zwischen Akteuren und Anwendungsfällen
- Aktivitätsdiagramm → Verhalten einer Klasse oder Komponente
- Zustandsdiagramm → Automaten, Zustandsübergänge im Leben eines Systems
- Sequenzdiagramm → Interaktionen zwischen Objekten (Nachrichtenfluss)

Klassendiagramm

5 grundlegende Phasen:

- Analyse/Definition → was muss SW-System leisten?
- Entwurf/Design → Wie soll SW-System realisiert werden?
- Implementierung
- Test → Klassendiagramme können als Referenz verwendet werden
- Einsatz/Wartung → Übersicht/Doku

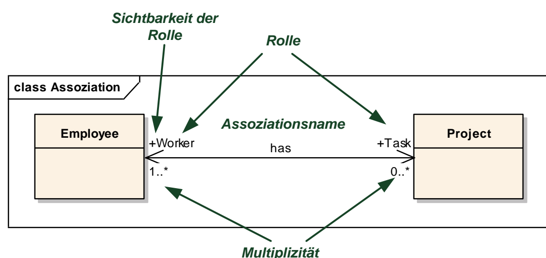
Sichtbarkeit von Variablen:

+	public (öffentlich, für alle sichtbar)
-	private (nur innerhalb Klasse sichtbar)
#	protected (Vererbungshierarchie)
~	package (Klassen im gleichen Paket)

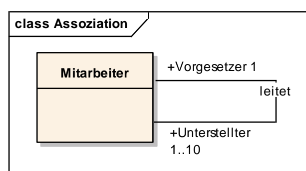
Assoziation

Wird eine Assoziation mit der Angabe einer Navigierbarkeit versehen, so spricht man von einer **gerichteten Assoziation.**

- **Navigierbar**
  Kenntnis des anderen Assoziationspartners vorgeschrieben
- **Nicht navigierbar**
  Verbietet die Kenntnis
- **Unspezifiziert**
  keine zwingende Aussage über die Kenntnisse der Klassen untereinander
- **bidirektionale Navigierbarkeit**
  erlaubt Kenntnis beider Klassen voneinander

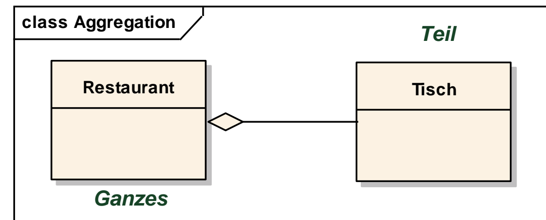


Reflexive Assoziation



Aggregation

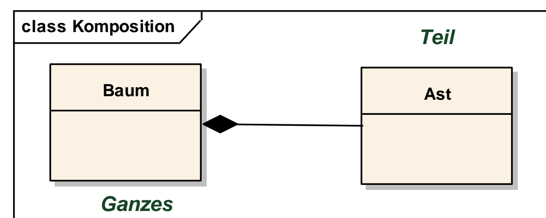
Die **Aggregation** ist eine spezielle Form einer binären Assozation und beschreibt eine **Ganzes-Teile-Beziehung**.



Komposition

Die **Komposition** ist eine starke Form der Aggregation. Die **Verbindung** zwischen den Teilen und dem Ganzen wird jedoch als **untrennbar** definiert.

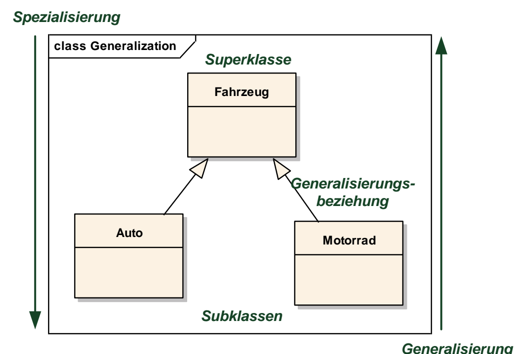
Wird das Ganze zerstört, endet auch die Existenz der Teile.



Generalisierung/Spezialisierung

Die **Generalisierung** modelliert eine **Beziehung zwischen einer spezifischen Subklasse und einer allgemeinen Superklasse** und definiert damit eines der zentralen Konzepte er Objektorientierung.

Eine Subklasse kann alle Attribute und Operationen der Superklasse verwenden. Sie erbt sozusagen die Eigenschaften.
➔ Vererbung



Sequenzdiagramm

Sequenzdiagramme modellieren Interaktionen zwischen Objekten. Sie stellen Abläufe dar und konzentrieren sich auf den Nachrichtenaustausch.

Lebenslinie (Lifeline)

Eine Lebenslinie (engl. Lifeline) repräsentiert **genau einen** individuellen **Teilnehmer einer Interaktion.**

Nachricht (Message)

Eine Nachricht definiert eine **Kommunikation und deren Richtung** zwischen zwei Teilnehmern einer Interaktion.

synchron
der Sender wartet auf die Ausführung der Operation/Behandlung des Signals durch den Empfänger.

asynchron
der Sender führt nach dem Senden der Nachricht ohne zu warten seine Tätigkeit fort.

Interaktionsrahmen (Interaction Frame)

Ein Interaktionsrahmen kapselt eine **Verhaltensdefinition**, deren **Fokus auf** der Darstellung eines **Informationsaustauschs** liegt.

Kombinierte Fragmente (Combined Fragments)

- Alternative Kommunikationspfade
  alt = Alternative

- Optionale Kommunikationspfade
  opt = Option

- Parallele Kommunikationspfade
  par = Parallelität

- Einzuhaltende Sequenzen von Nachrichten (schwach/strikt)
  seq = schwache Sequenz
  strict = strikte Sequenz

- Fehlerhafte Sequenzen von Nachrichten
  neg = Negation

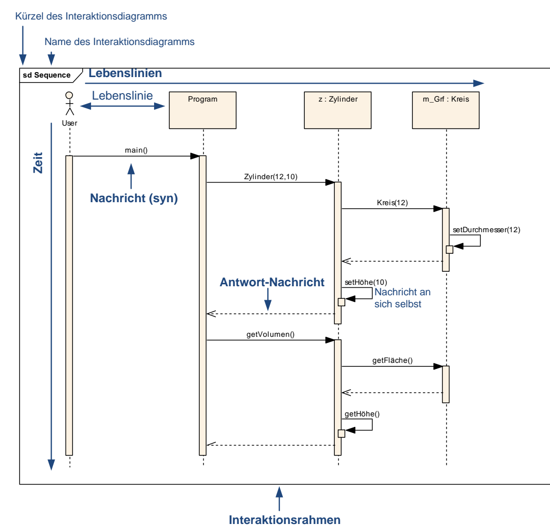
- Kritische, nicht unterbrechbare Bereiche
  critical = kritischer Bereich

- Sicherstellungen von Interaktionsabläufen
  assert = Sicherstellung

- Irrelevante und zu ignorierende Nachrichten
  ignore {Nachricht} = irrelevante Nachricht

- Besonders relevante und bedeutende Nachrichten
  consider {Nachricht} = relevante Nachricht

- Schleifen
  loop (i <= 10) = Schleife



Anwendungsfalldiagramm

Anwendungsfalldiagramme (engl. Use Case Diagrams) modellieren die Funktionalität des Systems auf einem **hohen Abstraktionsniveau** aus der so genannten Black-Box-Sicht des Anwenders. **Es werden nur die Anwendungsfälle definiert, die ein externer Anwender wahrnehmen kann und deren Ausführung ihm einen erkennbaren Nutzen bringt.**

Systemgrenze

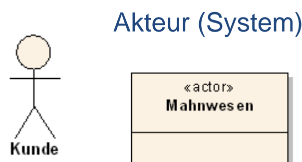
Die **Systemgrenze** umfasst **ein System**, das die benötigten Anwendungsfälle bereitstellt und mit dem die Anwender interagieren.

Akteur

Ein Akteur modelliert **einen Typ oder eine Rolle**, die ein externer Benutzer oder ein externes System während der Interaktion mit einem System einnimmt.

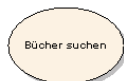
➔ **Akteure werden immer ausserhalb der Systemgrenze modelliert!**

Akteur (System)



Anwendungsfall

Ein Anwendungsfall spezifiziert eine abgeschlossene Menge von **Aktionen**, die von einem **System bereitgestellt** werden und einen **erkennbaren Nutzen** für einen oder mehrere Akteure bringen.



Assoziation

Eine Assoziation modelliert in Anwendungsfalldiagrammen **eine Beziehung zwischen Akteuren und Anwendungsfällen**.

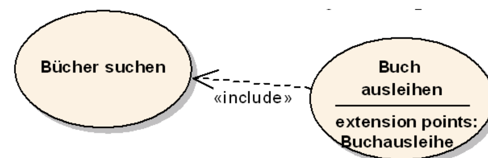
Generalisierung/Spezialisierung

Eine Generalisierung kann in Anwendungsfalldiagrammen zwischen Akteuren oder Anwendungsfällen modelliert werden und definiert eine **Beziehung zwischen einem spezifischen und einem allgemeine Element.**

Include-Beziehung

Eine Include-Beziehung modelliert die **unbedingte Einbindung der Funktionalität eines Anwendungsfalls in einen anderen Anwendungsfall.**

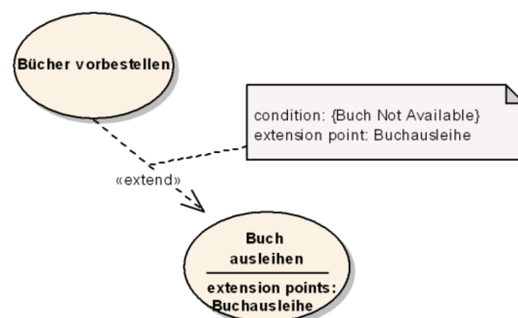
Jedes Mal, wenn der einbindende Anwendungsfall aufgerufen wird, muss auch der eingebundene Anwendungsfall aufgerufen werden.



Extend-Beziehung

Eine Extend-Beziehung modelliert die **bedingte Einbindung der Funktionalität eines Anwendungsfalls in einen weiteren Anwendungsfall.**

Die Funktionalität des erweiternden Anwendungsfalls ***kann*** in die Funktionalität des erweiterten Anwendungsfalls eingefügt werden. **Im Gegensatz zum Include ist der erweiternde Anwendungsfall vom erweiterten unabhängig und kann auch ohne ihn ausgeführt werden.**



Aktivitätsdiagramm

Aktivitätsdiagramme modellieren das Verhalten einer Klasse oder Komponente. Sie können in allen Phasen der Softwareentwicklung eingesetzt werden.

Aktion

Eine **Aktion** stellt die fundamentale **Einheit ausführbarer Funktionalität** dar, die im Modell nicht weiter zerlegt wird und somit **atomar** ist.

Kontrollfluss

Der **Kontrollfluss** ist eine **gerichtete Verbindung zwischen Aktivitätsknoten** und repräsentiert deren **Ausführungsreihenfolge**.

Würde der Kontrollfluss das Diagramm unübersichtlich machen, weil sich beispielsweise mehrere Kontrollflüsse schneiden, erlaubt die UML ihre Auftrennung durch **Konnektoren**.

Aktivitätsbereich

Aktivitätsbereiche gruppieren Aktivitätsknoten zu Organisationseinheiten

Objektknoten und Objektfluss

Objektknoten modellieren die **Übergabe von Objekten** zwischen Aktionen und können als eine Art **Speicher für Objekte der spezifizierten Klasse** betrachtet werden.

Der **Objektfluss** repräsentiert den **Transport von Objekten.**

Signal-Sendung und Signal-Empfang

Eine **Signal-Sendung** ist eine spezielle Art einer Aktion, die asynchron ein **Signal an ein Zielobjekt sendet.**

Ein **Signal-Empfang** ist eine spezielle Art einer Aktion, die auf den **Empfang eines Signals wartet.**

Aktivität

Eine **Aktivität** umfasst eine **geordnete Folge von Aktivitätsknoten.**

Start- und Endknoten

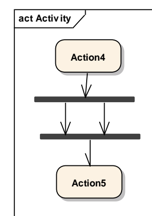
Ein **Startknoten** stellt den **Startpunkt des Kontrollflusses** innerhalb einer Aktivität dar.

Das **Flussende beendet nur den in ihn hineinlaufenden Kontrollfluss**. Er hat keine Auswirkung auf die weiteren Kontrollflüsse der Aktivität.
Das **Aktivitätsende beendet alle Kontrollflüsse** der Aktivität und somit die Aktivität selbst.

Entscheidungs- und Verbindungsknoten

Ein **Entscheidungsknoten stellt eine Verzweigung des Kontrollflusses** dar, an der genau einer der möglichen Kontrollflüsse ausgewählt wird.
Ein **Verbindungsknoten fasst mehrere alternative Kontrollflüsse zusammen**.

Gabelung und Vereinigung



Eine **Gabelung teilt einen Kontrollfluss in mehrere parallele Kontrollflüsse auf**. Eine **Vereinigung fasst mehrere Kontrollflüsse** zu einem einzigen Kontrollfluss **zusammen**.

Schleifenknoten

Ein **Schleifenknoten** ist eine **zusammengesetzte Aktivität**, die eine **Initialisierung** (for-Bereich), einen **Test** (while-Bereich) und einen **Schleifenkörper** (do-Bereich) repräsentiert.

Bedingungsknoten

Ein **Bedingungsknoten** ist eine **zusammengesetzte Aktivität**, die eine exklusive **Wahl zwischen einer oder mehreren Alternativen repräsentiert**.

Unterbrechungsbereich

Ein **Unterbrechungsbereich** umfasst eine **Gruppe von Aktivitätsknoten, deren Ausführung abgebrochen** werden kann.

Zustandsdiagramm

Zustandsdiagramme modellieren wie Aktivitätsdiagramme das dynamische Verhalten eines Systems. Im Gegensatz zu Aktivitätsdiagrammen, die ihren Fokus auf die *Aktionen* eines Systems legen, konzentrieren sich Zustandsdiagramme auf die *Reaktionen* eines Systems.

Zustand

Ein **Zustand** modelliert eine **Situation**, in der gewisse, genau definierte Bedingungen gelten.

Spezielle interne Aktionen:

- **entry**
  wird beim Betreten des Zustands ausgeführt
- **do**
  startet nach entry, wird so lange ausgeführt, bis sie endet oder Zustand verlassen wird
- **exit**
  wird vor dem Verlassen eines Zustands ausgeführt

Event und Transition

Eine **Transition** ist eine gerichtete Beziehung zwischen zwei Zuständen und stellt einen **Zustandsübergang vom Quell- zum Zielzustand** dar.

Es gibt fünf Arten von Events:

- **CallEvent**
  Wird im Quellzustand die Call-Operation aufgerufen, wird in den Zielzustand gewechselt.
- **SignalEvent**
  Empfängt das Objekt im Quellzustand ein SignalEvent, wechselt es in den Zielzustand.
- **ChangeEvent**
  boolescher Ausdruck (when Kontostand < 0) etc.
- **TimeEvent**
  Zum Beispiel „nach der Mittagspause“
- **AnyReceiveEvent**
  wird mit **all** notiert. Gilt für alle Aktionen, für die nichts definiert wurde.

- **[Guard]**
  Eine Transition wird nur ausgeführt, wenn Guard zu true ausgewertet wurde → [Lust zu Arbeiten]
- **/Effekt**
  definiert Aktionen, die bei einer Transition ausgeführt werden

Startzustand, Endzustand und Terminator

Der **Startzustand** stellt den **Startpunkt** des Zustandsautomaten dar.

Die Ausführung einer **Region oder Ebene von Zuständen ist bei Erreichen eines Endzustands** beendet.
die Ausführung eines ganzen **Zustandsautomaten** ist bei Erreichen eines **Terminators** beendet.

Entscheidung und Kreuzung

Eine **Kreuzung modelliert eine Hintereinanderschaltung von Transitionen** (Entscheidung wird bereits vor dem Erreichen der Kreuzung getroffen → Statisch)

Entscheidungen modellieren dynamische Verzweigungen.

Zusammengesetzter Zustand

Zusammengesetzte Zustände modellieren Hierarchien von Zuständen.

Region

Regionen teilen zusammengesetzte Zustände oder ganze Zustandsautomaten **in disjunktive Bestandteile** auf.

Rahmen eines Zustandsautomaten

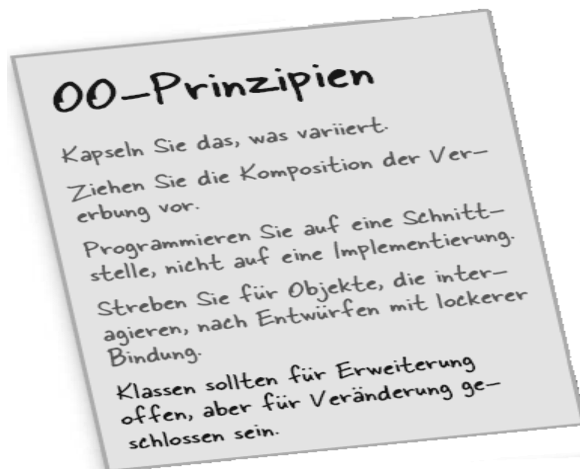
Ein **Zustandsautomat** kann **von einem Rahmen** umfasst und benannt werden. Das Kürzel **sm** steht für **StateMachine**, die englische Bezeichnung für einen Zustandsautomaten.

Generalisierung/Spezialisierung

Zustandsdiagramme können generalisiert und spezialisiert werden.

Entwurfsmuster

OOP-Prinzipien



- Kapseln Sie was variiert (Buch S. 9).
- Ziehen Sie die Komposition der Vererbung vor (Buch S. 23, + Strategie).
- Programmieren Sie auf eine Schnittstelle, nicht auf eine Implementierung (Buch S.11, +Strategie).
- Streben Sie für Objekte, die interagieren nach Entwürfen mit lockerer Bindung (Buch: Observer).
- Klassen sollten für Erweiterung offen, aber für Veränderung geschlossen sein. (Buch: Decorator).

Strategy

Das Strategy-Muster definiert eine Familie von Algorithmen, kapselt sie einzeln und macht sie austauschbar. Das Strategy-Muster ermöglicht es, den Algorithmus unabhängig von den Clients die ihn einsetzen, variieren zu lassen.

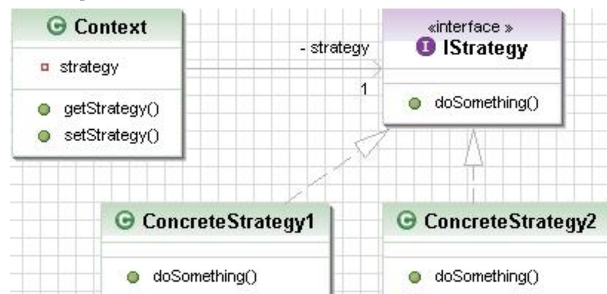
(aus wikipedia)

Verwendung

Die Verwendung von Strategien bietet sich an, wenn:

- viele verwandte Klassen sich nur in ihrem Verhalten unterscheiden.
- unterschiedliche (austauschbare) Varianten eines Algorithmus benötigt werden.
- Daten innerhalb eines Algorithmus vor Klienten verborgen werden sollen
- verschiedene Verhaltensweisen innerhalb einer Klasse fest integriert sind (meist über Mehrfachverzweigungen), aber
    - die verwendeten Algorithmen wiederverwendet werden sollen bzw.
    - die Klasse flexibler gestaltet werden soll

Beispiel



IStrategy dient nur als Schnittstelle für alle unterstützten Algorithmen. Die Implementierung der eigentlichen Algorithmen befinden sich in den Ableitungen **(ConcreteStrategyX).** Der **Context** erhält eine Strategie, welche vom Typ IStrategy abgeleitet werden muss. **Dies hat den Vorteil, dass alle von IStrategy abgeleiteten Klassen übergeben werden können.** Somit wird der Algorithmus über die Schnittstelle verwendet und dies **ermöglicht das Austauschen der Strategie zur Laufzeit.**

Vorteile

- Es wird eine Familie von Algorithmen definiert.
- Es wird die Auswahl aus verschiedenen Implementierungen ermöglicht und dadurch erhöht sich die Flexibilität und Wiederverwendbarkeit.
- Es können Mehrfachverzweigungen vermieden werden und dies erhöht die Übersicht des Codes.
- Strategien bieten eine Alternative zur Unterklassenbildung der Kontexte.
- **AUSTAUSCH DER STRATEGIE ZUR LAUFZEIT!!!**

Nachteile

- Klienten müssen die unterschiedlichen Strategien kennen, um zwischen ihnen auswählen und den Kontext initialisieren zu können.
- Gegenüber der Implementierung der Algorithmen im Kontext entsteht hier ein zusätzlicher Kommunikationsaufwand zwischen Strategie und Kontext.
- Die Anzahl von Objekten wird erhöht.

Code

```
public class Context
{
        private IStrategy m_Strategie;

        public Context(string x)
        {
          m_x = x;
        }

     public void doSomething()
     {
         //m_Strategie.doSomething();
     }
}

public class ConcreteStrategy1 :
IStrategy
{

        public void doSomething()
     {
         //Hier was machen
        }

}

public interface IStrategy
{
        void doSomething();
}
```

Observer

Das Observer-Muster definiert eine Eins-zu-viele-Abhängigkeit zwischen Objekten in der Art, dass alle abhängigen Objekte benachrichtigt werden, wenn sich der Zustand des einen Objektes ändert.

Muster, mit dem einer Gruppe von Objekten auf Basis einer lockeren Bindung Zustände mitgeteilt werden können.

Das, was beim Observer-Muster variiert, ist der Zustand des Objekts und die Typen der Beobachter. Mit diesem Muster können Sie die Objekte variieren, die vom Zustand des Objektes abhängig sind, ohne das Subjekt verändern zu müssen. Das nennt man vorausschauend handeln.

Subjekt und Beobachter nutzen beide Interfaces. Das Subjekt hält Objekte nach, die das Interface Observer implementieren, während die Beobachter sich registrieren und vom Subjekt-Interface benachrichtigt werden. Dies hält die Dinge ordentlich und locker gebunden.

Das Observer-Muster nutzt Komposition, um eine beliebige Anzahl von Beobachtern mit Subjekten zu verbinden. Diese Beziehungen werden nicht durch irgendeine Art von Vererbungshierarchie implementiert. Nein, wie werden zur Laufzeit durch Komposition eingerichtet.

(aus wikipedia)

Verwendung

z.B. Messstation

- Subjekt verwaltet bestimmte Datenmenge
- Beobachter haben ein Abonnement beim Subjekt (sind registriert), um Aktualisierungen zu erhalten, wenn sich die Daten des Subjekts ändern.

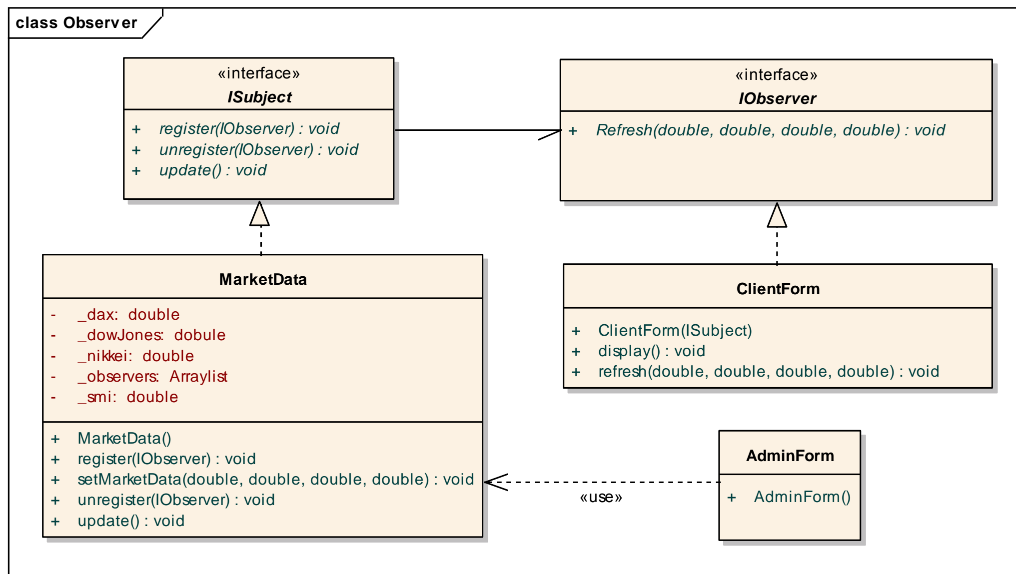
Vorteile

- Subjekte und Observer können unabhängig variiert werden.
- Subjekt und Benutzer sind auf abstrakte und minimale Art lose gekoppelt. Das beobachtete Objekt braucht keine Kenntnis über die Struktur seiner Observer zu besitzen, sondern kenn diese nur über die Observer-Schnittstelle.
- Ein abhängiges Objekt erhält die Änderungen automatisch
- Multicasts werden unterstützt.

Nachteile

- Änderungen am Objekt führen bei grosser Observeranzahl zu hohen Änderungskosten. Einerseits informiert das Subjekt jeden Observer, auch wenn dieser die Änderungsinformation nicht benötigt. Zusätzlich können die Änderungen weitere Änderungen nach sich ziehen und so einen unerwartet hohen Aufwand haben.
- Ruft ein Observer während der Bearbeitung einer gemeldeten Änderung wiederum Änderungsmethoden des Subjektes auf, kann es zu Endlosschleifen kommen.
- Der Mechanismus liefert keine Information darüber, was sich geändert hat. Die daraus resultierende Unabhängigkeit der Komponenten kann sich allerdings auch als Vorteil herausstellen.
- Bei der gerade durchgeführten Observierung eines Objektzustands kann es notwendig sein, einen konsistenten Subjektzustand zu garantieren. Dies kann durch synchrone Aufrufe der Notifizierungsmethode des Observers sichergestellt werden. In einem Multithreading System sind evtl. Lockingmechanismen oder Threads mit queuing zur Observer-Notifizierung erforderlich.

Beispiel



Code

```
public interface IObserver
    {
        void Refresh(double SMI,
double DowJones, double Nikkei, double
DAX);
    }


public partial class ClientForm :
Form, IObserver     {

//Membervariablen…
        private ISubject _subject =
null;

        public ClientForm(ISubject
marketData)
        {
            InitializeComponent();

            _subject = marketData;
            marketData.Register(this);
        }

        public void Refresh(double
smi, double dowJones, double nikkei,
double dax)
        {
             //Werte ändern…
            Display();
        }
        public void Display()
        {
             //Sachen anzeigen…
    }
```

```
public interface ISubject
    {
        void Register(IObserver
observer);
        void Unregister(IObserver
observer);
        void Update();
    }

public class MarketData : ISubject
    {
        //Membervariablen…

        private ArrayList _observers =
null;

        public MarketData()
        {
            _observers = new
ArrayList();
        }

        public void Register(IObserver
observer)
        {
            _observers.Add(observer);
        }
        public void
Unregister(IObserver observer)
        {

_observers.Remove(observer);
        }
        public void Update()
        {
            foreach (IObserver
observer in _observers)
            {
                observer.Refresh(_smi,
_dowJones, _nikkei, _dax);
            }
        }
    }
```

Decorator

Das Decorator-Muster fügt einem Objekt dynamisch zusätzliche Verantwortlichkeiten hinzu. Dekorierer bieten eine flexible Alternative zur Ableitung von Unterklassen zum Zweck der Erweiterung der Funktionalität.

Eigenschaften

1. Dekorierer haben den gleichen Supertyp wie die Objekte, die sie dekorieren

2. Sie können ein oder mehr Objekte verwenden, um ein Objekt einzupacken.

3. Da der Dekorierer den gleichen Supertyp wie das dekorierte Objekt hat, können wir das dekorierte Objekte an Stelle des ursprünglichen (jetzt eingepackten) Objekts herumreichen.

4. *Der Dekorierer fügt sein eigenes Verhalten hinzu, bevor und/oder nachdem der Aufruf an das dekorierte Objekt delegiert wurde, um die Arbeit abzuschliessen.*

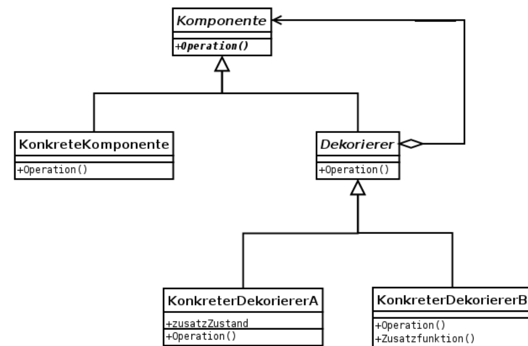
5. Objekte können jederzeit dekoriert werden. Wir können Objekte also zur Laufzeit dynamisch mit so vielen Dekorierern dekorieren, wie es uns gefällt.

Verwendung

Die Instanz eines Dekorierers wird vor die zu dekorierende Klasse geschaltet. Der Dekorierer hat die gleiche Schnittstellt wie die zu dekorierende Klasse. Aufrufe an den Dekorierer werden dann verändert oder unverändert weitergeleitet (Delegation), oder sie werden komplett in Eigenregie verarbeitet. Der Dekorierer ist dabei „unsichtbar“, da der Aufrufende gar nicht mitbekommt, dass ein Dekorierer vorgeschaltet ist.

Beispiele

- Verschönern von GUI-Komponenten, z.B. Textfeld dekorieren
- Erweitern der Funktionalität von GUI-Komponenten.



Akteure

Die abstrakte Komponente definiert die öffentliche Schnittstelle für zu dekorierende Objekte. Die konkrete Komponente definiert Objekte, die dekoriert werden können.

Der abstrakte Dekorierer hält eine Referenz auf eine konkrete Komponente und bietet dieselbe Schnittstelle wie die abstrakte Komponente. Der konkrete Dekorierer definiert und implementiert eine oder mehrere spezielle Dekorationen.

Vor- und Nachteile

Die Vorteile bestehen darin, dass mehrere Dekorierer hintereinandergeschaltet werden können; die Dekorierer können zur Laufzeit und sogar nach der Instantiierung ausgetauscht werden. Die zu dekorierende Klasse ist nicht unbedingt festgelegt (wohl aber deren Schnittstelle). Zudem können lange und unübersichtliche Vererbungshierarchien vermieden werden.

Zu den Nachteilen zählt, dass eventuell viele Methoden implementiert werden müssen, die die Aufrufe lediglich weiterleiten. Weiterhin werden Erweiterungen der zu dekorierenden Klasse unabsichtlich versteckt. Manchmal behilft man sich deshalb mit einem Broadcast-Mechanismus. Schließlich wird auch der Aufbau der Dekorierer-Instanzen komplexer. Als Hilfe kann dazu das Entwurfsmuster des Erbauers dienen

Code

Komponente

```
public abstract class Getraenk
    {
        protected string beschreibung
= "Unbekanntes Getränk";
        protected size _s;

        public virtual string
getBeschreibung()
        {
            return beschreibung+"
("+_s+")";
        }

        public abstract double
preis();

    }
```

Konkrete Komponente

```
public class DunkleRoestung: Getraenk
    {
        public DunkleRoestung()
        {
            beschreibung = "Dunkle
Röstung";
        }
        public override double preis()
        {
            return 2.20;
        }
    }
```

Dekorierer

```
public abstract class ZutatDekorierer:
Getraenk
    {
        public override abstract
string getBeschreibung();
    }
```

Konkreter Dekorierer

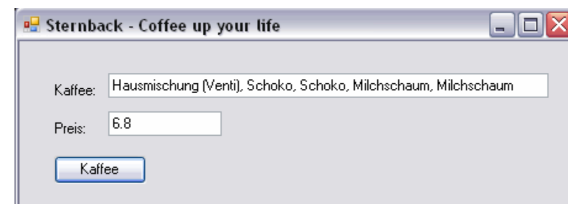
```
public class Schoko:ZutatDekorierer
    {
        private Getraenk _getraenk;

        public Schoko(Getraenk g)
        {
            _getraenk = g;
        }
        public override string
getBeschreibung()
        {
            return
_getraenk.getBeschreibung() + ",
Schoko";
        }
        public override double preis()
        {
            return _getraenk.preis() +
0.50;
        }
    }
```

Program.cs (Dekoration)

```
Getraenk _g = new
Hausmischung(size.Venti);
            _g = new Schoko(_g);
            _g = new Schoko(_g);
            _g = new Milchschaum(_g);
            _g = new Milchschaum(_g);

            txtCoffee.Text =
_g.getBeschreibung();
            txtPrice.Text =
_g.preis().ToString();
```



Factory

Wenn Sie Programmcode haben, der viele konkrete Klassen verwendet, riskieren Sie Probleme. Weil der Code evtl. geändert werden muss, wenn neue konkrete Klassen hinzugefügt, werden müssen! Der Programmcode muss also wieder geöffnet werden, wenn Änderungen oder Erweiterungen anstehen. → Pflege und Aktualisierung
(Objekterstellung kapseln)

Vor- und Nachteile von statischen gegenüber Nichtstatischen Fabriken?

Vorteile:

- Einfacher, da das Erstellen der Objekte wegfällt
- Sinnvoll, wenn nur eine Fabrik gebraucht wird.

Nachteile:

- Wenn man Pizzas aus verschiedenen Fabriken zubereiten will, müssen mehrere Factories erstellt werden. Dies ist mit einer statischen Fabrik nicht möglich. Hier soll mit nichtstatischen Methoden gearbeitet werden
- Flexibilität ist eingeschränkt. Man kann keine Unterklassen bilden.

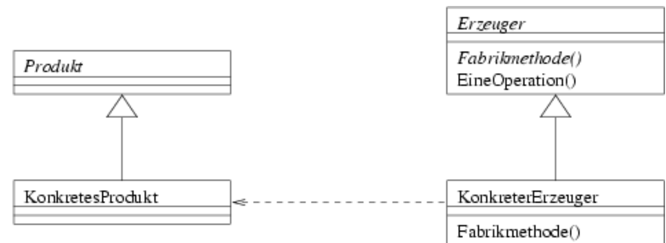
Fabrikmethode

Das Factory Method-Muster definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klassen instantiiert werden. Factory Method ermöglicht einer Klasse, die Instantiierung in Unterklassen zu verlagern.

Verwendung

Die Fabrikmethode (in der formal korrekten Bedeutung) findet Anwendung, wenn:

- eine Klasse die von ihr zu erzeugenden Objekte nicht kennen kann bzw. soll (1), 2)) oder
- Unterklassen bestimmen sollen, welche Objekte erzeugt werden (nur 1)).



Akteure

Produkt

- Basistyp (Klasse oder Schnittstelle) für das zu erzeugende Produkt

Erzeuger

- deklariert die Fabrikmethode, um ein solches Produkt zu erzeugen und kann eine Default-Implementation beinhalten. Mitunter wird für die Fabrikmethode eine Implementierung vorgegeben, die ein „Standard-Produkt" erzeugt.

KonkretesProdukt

- implementiert die Produkt-Schnittstelle (Subtyp von Produkt)

KonkreterErzeuger (nur bei 1)

- überschreibt die Fabrikmethode um konkrete(re) Produkte zu erzeugen, determiniert damit das konkrete Produkt (z. B. indem er den Konstruktor einer konkreten Klasse aufruft)

Vorteile

1. Fabrikmethoden entkoppeln ihre Aufrufer von Implementierungen konkreter Produkt-Klassen. Das ist insbesondere wertvoll, wenn Frameworks sich während der Lebenszeit einer Applikation weiterentwickeln - so können zu einem späteren Zeitpunkt Instanzen anderer Klassen erzeugt werden, ohne dass sich die Applikation ändern muss.
2. In C++, Java und ähnlichen Sprachen kann eine Fabrikmethode im Gegensatz zu einem Konstruktor einen aussagefähigeren Namen haben, z. B. Color.createRGB(...) vs. Color.createHSB(...).

Nachteile

1. Die Verwendung dieses Erzeugungsmusters läuft auf Unterklassenbildung hinaus.
2. Es muss eine eigene Klasse vorhanden sein, die die statische Methode aufnehmen kann (das hat schon zu Forderungen geführt, dass in Java- oder C#-Schnittstellen statische Methoden erlaubt werden sollten).

EinfacheFabrik - Code

Fabrik

```
public class EinfachePizzaFabrik
    {
        public IPizza
erstellePizza(string typ)
        {
            IPizza pizza = null;
            if (typ.Equals("Salami"))
                pizza = new
SalamiPizza();
            else if
(typ.Equals("Spinat"))
                pizza = new
SpinatPizza();
            else if
(typ.Equals("Schinken"))
                pizza = new
SchinkenPizza();
            if (pizza == null)
                return null;
            return pizza;
        }
    }
```

Pizzeria

```
public class Pizzeria
    {
        private string m_PizzeriaName;
        private EinfachePizzaFabrik
_Fabrik;


        public Pizzeria(string name) {
            setPizzeriaName(name);
        }
        public Pizzeria(string name,
EinfachePizzaFabrik fabrik)
        {
            setPizzeriaName(name);
            _Fabrik = fabrik;
        }
        public string
getPizzeriaName()
        {
            return m_PizzeriaName;
        }
        public void
setPizzeriaName(string value)
        {
            m_PizzeriaName = value;
        }

        public IPizza
bestellePizza(string typ)
        {
            IPizza pizza;
            pizza =
_Fabrik.erstellePizza(typ);
            pizza.vorbereiten();
            pizza.backen();
            pizza.schneiden();
            pizza.einpacken();
            return pizza;
        }

    }
```

Program.cs

```
class Program
    {
        static void Main(string[] args)
        {
            List<IPizza> VielePizzen = new
List<IPizza>();

            EinfachePizzaFabrik _Fabrik =
new EinfachePizzaFabrik();
            Pizzeria _Ristorante = new
Pizzeria("Ristorante Pizzeria di BBZW",
_Fabrik);


VielePizzen.Add(_Ristorante.bestellePizza("
Salami"));

VielePizzen.Add(_Ristorante.bestellePizza("
Schinken"));

VielePizzen.Add(_Ristorante.bestellePizza("
Salami"));

VielePizzen.Add(_Ristorante.bestellePizza("
Spinat"));
            foreach (IPizza p in
VielePizzen)
                Console.WriteLine(p);
            Console.ReadLine();
        }
    }
```

Statische Fabrikmethode

Bei statischer Methode sieht die bestellePizza-Methode von Pizzeria wie folgt aus:

```
public IPizza bestellePizza(string
typ)
        {
            IPizza pizza = null;
            pizza =
EinfachePizzaFabrik.erstellePizza(typ)
;
            pizza.vorbereiten();
            pizza.backen();
            pizza.schneiden();
            pizza.einpacken();
            return pizza;
        }
```

Fabrikmethode

Produkt

```
public abstract class Pizza
    {
        public virtual void vorbereiten()
        {
            Console.WriteLine("Make pizza
bereit fur dich");
            Console.WriteLine("Werfe
Teig...");
            Console.WriteLine("Füge Sugo
hinzu.");
            Console.WriteLine("Füge Beläge
hinzu:");
        }
        public virtual void backen()
        {
            Console.WriteLine("Backe Pizza
25 min bei 200 Grad in echte Holzofe");
        }
        public virtual void schneiden()
        {
            Console.WriteLine("Schneide
Pizza diagonal in Stucke");
        }
        public void einpacken()
        {
            Console.WriteLine("Zum
mitnehme? Packe Pizza in die offizielle
Pizzeria-Schachtel");
        }
    }
```

Pizzeria - Fabrik

```
public abstract class Pizzeria
    {
        private string _PizzeriaName;
        private IFabrik _Fabrik;

        public Pizzeria()
        {
            setPizzeriaName("AL FORNO DI
MILANO");
        }
        public Pizzeria(string name)
        {
            setPizzeriaName(name);
        }
        public Pizzeria(string name,
IFabrik fabrik)
        {
            setPizzeriaName(name);
            _Fabrik = fabrik;
        }
        public string getPizzeriaName()
        {
            return _PizzeriaName;
        }
        public void setPizzeriaName(string
value)
        {
            _PizzeriaName = value;
        }

        public Pizza.Pizza bestellen(string
element)
        {
            Pizza.Pizza pizza = null;
            pizza = erstellePizza(element);
            pizza.vorbereiten();
            pizza.backen();
            pizza.schneiden();
            pizza.einpacken();
            return pizza;
        }

        protected abstract Pizza.Pizza
erstellePizza(string typ);
    }
```

Konkreter Erzeuger

```
public class BerlinPizzeria :  Pizzeria
    {
        protected override
_4_MitFabrikMethode.Pizza.Pizza
erstellePizza(string typ)
        {
            Pizza.Pizza pizza = null;
            if (typ.Equals("Salami"))
                pizza = new
Pizza.BerlinSalamiPizza();
            else if (typ.Equals("Spinat"))
                pizza = new
Pizza.SpinatPizza();
            else if
(typ.Equals("Schinken"))
                pizza = new
Pizza.SchinkenPizza();
            if (pizza == null)
                return null;

            return pizza;
        }
    }
```

Program.cs

```
static void Main(string[] args)
        {
            Pizzeria.Pizzeria
berlinerPizzeria = new BerlinPizzeria();
            Pizzeria.Pizzeria
muenchenPizzeria = new
MuenchenPizzeria("Buitoni. Prodotto
d'Italia");

            List<Pizza.Pizza> VielePizzen =
new List<Pizza.Pizza>();

            Console.WriteLine("\n------
ZUBEREITUNG START----------\n");

VielePizzen.Add(berlinerPizzeria.bestellen(
"Salami"));
            Console.WriteLine("\n----------
------");

VielePizzen.Add(muenchenPizzeria.bestellen(
"Salami"));
            Console.WriteLine("\n----------
------");

VielePizzen.Add(berlinerPizzeria.bestellen(
"Spinat"));
            Console.WriteLine("\n-------
ZUBEREITUNG FINITO---------\n\n");

            foreach (Pizza.Pizza p in
VielePizzen)
                Console.WriteLine(p);
            Console.ReadLine();
}
```

Abstrakte Fabrik

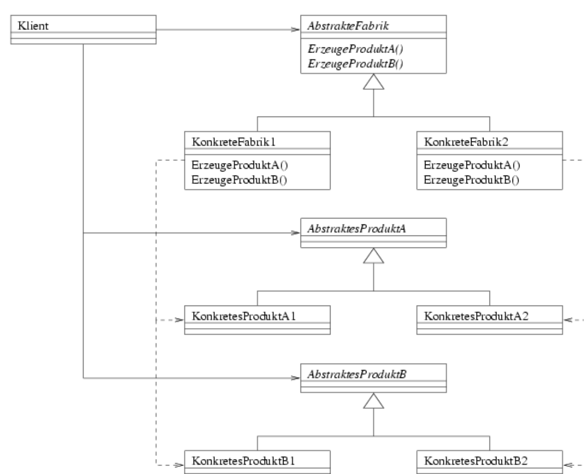
Das Abstract Factory-Muster bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben.

Verwendung

Die Abstrakte Fabrik findet Anwendung, wenn

- ein System unabhängig von der Art der Erzeugung seiner Produkte arbeiten soll,
- ein System mit einer oder mehreren Produktfamilien konfiguriert werden soll,
- eine Gruppe von Produkten erzeugt und gemeinsam genutzt werden soll oder
- wenn in einer Klassenbibliothek die Schnittstellen von Produkten ohne deren Implementierung bereitgestellt werden sollen.
- Eine typische Anwendung ist die Erstellung einer grafischen Benutzeroberfläche mit unterschiedlichen Themes.

Eine Abstrakte Fabrik vereinigt die Verantwortungen "Zusammenfassung der Objektgenerierung an einer Stelle" und "Möglichkeit zu abstrakten Konstruktoren" (siehe auch unten unter "Verwandte Entwurfsmuster").



Akteure

AbstrakteFabrik

- definiert eine Schnittstelle zur Erzeugung abstrakter Produkte einer Produktfamilie

KonkreteFabrik

- erzeugt konkrete Produkte einer Produktfamilie durch Implementierung der Schnittstelle

AbstraktesProdukt

- definiert eine Schnittstelle für eine Produktart

KonkretesProdukt

- definiert ein konkretes Produkt einer Produktart durch Implementierung der Schnittstelle, wird durch die korrespondierende konkrete Fabrik erzeugt

Klient

- verwendet die Schnittstellen der abstrakten Fabrik und der abstrakten Produkte

Vorteile

- Konkrete Klassen werden isoliert.
- Der Austausch von Produktfamilien ist auf einfache Art und Weise möglich.

Nachteile

- Konkrete Klassen werden isoliert.
- Der Austausch von Produktfamilien ist auf einfache Art und Weise möglich.

Beispiel

Es soll eine Spielesammlung per Software entwickelt werden. Die verwendeten Klassen sind dabei

1. *Spielbrett* (erstes abstraktes Produkt), auf das Spielfiguren platziert werden können und das beispielsweise eine Methode besitzt, um sich auf dem Bildschirm anzuzeigen. Konkrete, davon abgeleitete Produkte sind *Schachbrett*, *Mühlebrett*, *Halmabrett* etc.
2. *Spielfigur* (zweites abstraktes Produkt), die auf ein Spielbrett gesetzt werden kann. Konkrete, davon abgeleitete Produkte sind *Hütchen*, *Schachfigur* (der Einfachheit halber soll es hier nur einen Typ an Schachfiguren geben), *Holzsteinchen* etc.
3. *Spielfabrik* (abstrakte Fabrik), die Komponenten (Spielbrett, Spielfiguren) eines Gesellschaftsspiels erstellt. Konkrete, davon abgeleitete Fabriken sind beispielsweise *Mühlefabrik*, *Damefabrik*, *Schachfabrik* etc.

Ein Klient (z.B. eine Instanz einer Spieler- oder Spielleiter-Klasse) kann sich von der abstrakten Fabrik Spielfiguren bzw. ein Spielbrett erstellen lassen. Je nachdem, welches konkrete Spiel gespielt wird, liefert beispielsweise die...

- Schachfabrik ein Schachbrett und Schachfiguren
- Damefabrik ebenfalls ein Schachbrett, aber Holzsteinchen
- Mühlefabrik ein Mühlebrett, aber ebenfalls Holzsteinchen

Zusammengesetzte Muster

MVC

Model View Controller

MVC besteht aus 3 zusammengefügten Mustern (***Observer, Strategy, Composite***).

View- und Model-Objekte werden durch den Aufbau eines Protokolls zur Benachrichtigung entkoppelt. View-Objekt muss sicherstellen, dass seine Darstellung den Zustand des Model-Objekts wiedergibt.
→ ermöglicht es, mehrere Views an ein Model zu binden, um verschiedene Präsentationen anzubieten. Es können auch neue Views entwickelt werden, ohne das Programm umschreiben zu müssen.

Weitere Eigenschaft ist die mögliche Schachtelung von Views. So kann bspw. eine Dialogbog als eine komplexe View implementiert werden, die aus weiteren Views besteht, die einzelne Knöpfe darstellen (Composite-Pattern)

Model (M)

Das Model enthält die gesamte Daten-, Zustands und Anwendungslogik. Es weiss nichts über View und Controller, allerdings bietet es eine Schnittstelle, über die sein Zustand beeinflusst und abgerufen werden kann, und es kann Benachrichtigungen über Zustandsänderungen an Beobachter senden.

View (V)

Liefert eine Präsentation des Models. Die View enthält normalerweise den Zustand und die Daten, die sie benötigt, direkt vom Model.

Controller (C)

Nimmt die Eingabe des Benutzers an und stellt fest, was sie für das Model bedeutet.

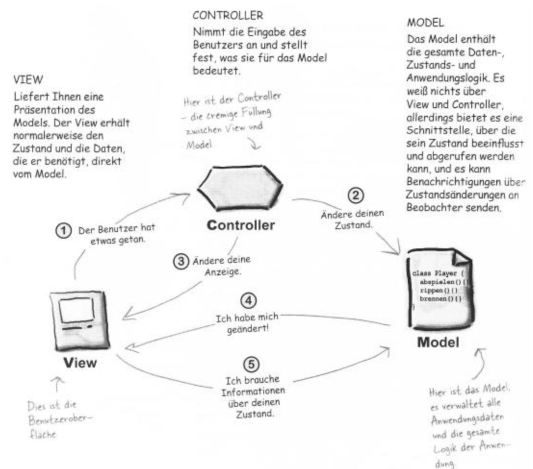
Entwurfsmuster

Model → Observer
Das Model implementiert das Observer-Muster, um interessierte Objekte bei Zustandsänderungen auf dem neuesten Stand zu halten. Das Observer-Muster macht das Model völlig unabhängig von View und Controller. So können wir für das gleiche Model unterschiedliche Views oder sogar mehrere Views auf einmal verwenden.

View → Composite
Die Anzeige besteht aus ineinander verschachtelten Fenstern, Panels, Buttons, Textlabels usw. Jede Anzeigekomponente ist ein Kompositum (z.B. ein Fenster) oder ein Blatt (z.B. ein Button). Wenn der Controller eine Aktualisierung des Views haben möchte, muss er es einfach nur der obersten View-Komponente sagen. Das Composite-Muster besorgt dann den Rest.

Controller → Strategy
Der View und der Controller implementieren das klassische Strategy-Muster: Der View ist ein Objekt, das mit einer Strategie konfiguriert ist; der Controller liefert diese Strategie. Die View ist nur für die visuellen Aspekte der Anwendung zuständig; alle Entscheidungen über das Verhalten der Schnittstelle delegiert er an den Controller. Durch die Verwendung des Strategy-Musters bleibt die View ausserdem entkoppelt vom Model, denn es ist ja der Controller, der bei der Bearbeitung von Benutzeranfragen für die Interaktion mit dem Model zuständig ist. Die View weiss nichts darüber, wie das vor sich geht.

Ablauf



1. **Sie sind der Benutzer – Sie interagieren mit der View**.
   View ist Sicht auf das Model. Sie teilt dem Controller mit, was getan wurde.

2. **Controller fordert Model auf, seinen Zustand zu ändern**
   nimmt Aktionen auf und interpretiert sie

3. **Auch Controller kann View auffordern, seinen Zustand zu ändern**.

4. **Model benachrichtigt View, wenn sich sein Zustand geändert hat.**

5. **View fragt Model nach seinem Zustand.**

Java und Apache-Tomcat

Begriffe

Webserver

Ein Webserver (lat. servus, engl. server „Diener, Dienst") ist ein Computer, der Dokumente an Clients wie z.B. Webbrowser überträgt. Als Webserver bezeichnet man den Computer mit Websoftware oder nur die Websoftware. Webserver werden lokal, in Firmennetzwerken und überwiegend als WWW-Dienst im Internet eingesetzt. Dokumente können somit dem geforderten Zweck lokal, firmenintern und weltweit zur Verfügung gestellt werden.
Als Übertragungsmethoden dienen standardisierte Übertragungsprotokolle (HTTP, HTTPS) und Netzwerkprotokolle (TCP/IP) üblicherweise über den Port 80.

Apache-Tomcat

Apache Tomcat stellt eine Umgebung zur Ausführung von Java-Code auf Webservern bereit. Es handelt sich um einen in Java geschriebenen Servlet-Container, der mithilfe des JSP-Compilers Jasper auch JavaServer Pages in Servlets übersetzen und ausführen kann. Dazu kommt ein kompletter HTTP-Server.

JSP

JavaServer Pages, abgekürzt JSP, ist eine von Sun Microsystems entwickelte auf JHTML basierende Technik, die im Wesentlichen zur einfachen dynamischen Erzeugung von HTML- und XML-Ausgaben eines Webservers dient. Sie erlaubt, Java-Code und spezielle JSP-Aktionen in statischen Inhalt einzubetten. Dies hat den Vorteil, dass die Logik unabhängig vom Design implementiert werden kann.

Die JSP-Syntax ermöglicht mittels spezieller XML-Tags (JSP-Aktionen), vordefinierte Funktionalität einzubinden. Diese JSP-Aktionen werden in so genannten Tag-Bibliotheken (Tag-Library) als Erweiterung der HTML- bzw. XML-Tags definiert.

CGI

Das Common Gateway Interface (CGI) – in etwa Allgemeine Vermittlungsrechner-Schnittstelle – ist ein Standard für den Datenaustausch zwischen einem Webserver und dritter Software, die Anfragen bearbeitet. CGI ist eine schon länger bestehende Variante, Webseiten dynamisch bzw. interaktiv zu machen, deren erste Überlegungen auf das Jahr 1993 zurückgehen.

Ein Webserver, der CGI unterstützt, stellt der externen Software eine Laufzeitumgebung zur Verfügung, die insbesondere aus folgendem besteht:

- Umgebungsvariablen (z. B. „SERVER_NAME"), die dem Programm helfen, sich über die Anfrage, Webserver-Einstellung und -Situation zu informieren. Die Bezeichnungen sowie das Format der Inhalte sind größtenteils standardisiert.
- Bereitstellung von Ein- und Ausgabekanälen. Meist wird der stdout-Kanal mit der Antwort des Webservers verknüpft, stdin mit dem eventuell vorhandenen Request-Body.

Servlet

Als Servlets bezeichnet man Java-Klassen, deren Instanzen innerhalb eines Java-Webserver Anfragen von Clients entgegen nehmen und beantworten. Weiterhin sind sie fester Bestandteil aller Java-EE-Anwendungsserver. Die Servlet-Komponenten müssen immer die Schnittstelle javax.servlet.Servlet oder eine davon abgeleitete implementieren. Normalerweise wird eine Klasse erstellt, die von der Klasse javax.servlet.http.HttpServlet abgeleitet wird, welche wiederum javax.servlet.Servlet implementiert. Der Inhalt der Antworten kann dabei dynamisch, also im Moment der Anfrage, erstellt werden und muss nicht bereits statisch (etwa in Form einer HTML-Seite) für den Webserver verfügbar sein. Servlets stellen somit das Java-Pendant zu CGI-Skripten oder anderen Konzepten, mit denen dynamisch Web-Inhalte erstellt werden können (PHP, Ruby on Rails, Active Server Pages etc.), dar.

Webanwendungen

Was ist TomCat?

TomCat stellt eine Umgebung zur Ausführung von Java-Code auf Webservern bereit. Es handelt sich um einen Servlet-Container, der auch JSP in Servlets übersetzen und ausführen kann.

Client → Webserver → Tomcat

1. Client sendet dem Webserver via get oder post eine Anfrage an ein Servlet.
2. Der Webserver leitet die Anfrage an den Webcontainer (z.B. TomCat) weiter.
3. TomCat erstellt 2 Objekte
   a. HTTPServletRequest
   b. HTTPServletResponse
4. TomCat such Servlet und erstellt Threads für die Anfrage und übergibt die beiden Objekte
5. TomCat ruft die service()-Methode des Servlets auf → doGet() oder doPost() wird aufgerufen
6. doPost() / doGet()-Methode des Servlets erstellt dynamische Seite und packt diese Informationen in ein Antwort-Objekt.
7. Thread endet.
   Container wandelt Antwort-Objekt in HTTP-Antwort um
8. Container übergibt http-Antwort dem Webserver.
9. Webserver gibt die Seite an den Client zurück

Servlet-Namen

Ein Servlet hat drei verschiedene Namen:
1. öffentlicher URL-Name
2. interner vertraulicher Name (Deployment Name)
3. tatsächlicher Name

Deployment Descriptor (web.xml) wird benutzt, um das, was der Client anfordert (URL-Name) auf die eigentliche Servlet-Klassendatei abzubilden.

Warum haben Servlets mehrere Namen?

- um den tatsächlichen Pfad (Verzeichnisstruktur) zu verbergen
- Flexibilität (Servlet kann verschoben werden, ohne dass immer Code angepasst werden muss.

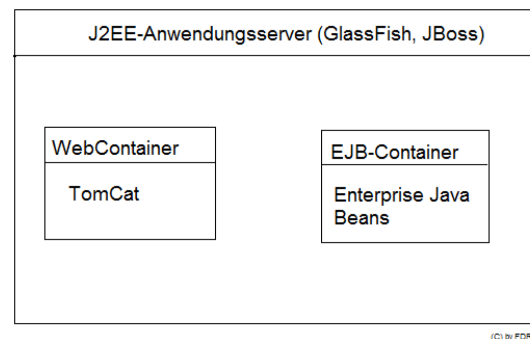
J2EE

Java Platform, Enterprise Edition, abgekürzt Java EE oder früher J2EE, ist die Spezifikation einer Softwarearchitektur für die transaktionsbasierte Ausführung von in Java programmierten Anwendungen und insbesondere Web-Anwendungen.

MVC

- **Model (M)**
  Java-Klasse
- **View (V)**
  JSP
- **Controller (C)**
  Servlet

J2EE-Anwendungsserver



Multitasking

Allgemein

Als Multitasking wird die Fähigkeit eines Betriebssystems bezeichnet, mehrere Prozesse parallel auszuführen. Ein Prozess läuft in einem gegenüber anderen Prozessen geschützten Adressraum im Arbeitsspeicher des Rechners

Programmcode, der in einer Sequenz abgearbeitet wird, nennt man einen Programmfaden, engl. Thread.

Ein Programm, das nur aus einem einzigen Programmfaden besteht, heisst single threaded.-

Ein Programm, das über mehrere Threads verfügt, wird Multithread-Programm genannt.

Synchrone und asynchrone Ausführung von Programmcode (Multithreading)

Der Thread ist die ausführende Instanz des Betriebssystems für unser Programm. Alle Arbeiten, die er ausführt, sind automatisch synchron (hintereinander geschachtelt). Aufgrund der natürlichen Serialisierung unseres Programms kann es vorkommen, dass die Programme nicht mehr flüssig arbeiten. Ein so konstruiertes Programm ist heute nicht mehr akzeptabel

Der asynchrone Methodenaufruf (=nebenläufige Programmausführung) zeichnet sich dadurch aus, dass der Code, der eine Methode asynchron aufruft, nicht wartet, bis die aufgerufene Methode fertig abgearbeitet ist, sondern unmittelbar nach dem Aufruf selbst weiterarbeitet.

Allerdings hat diese Nebenläufigkeit auch ihre Tücken:

- Ein Thread ist eine verwaltende Einheit eines Betriebssystems und braucht entsprechende Ressourcen des Systems.
- Allzu viele Threads bringen keine Geschwindigkeitsvorteile, sogar das Gegenteil kann der Fall sein
- Der Zugriff auf Daten von mehreren Threads muss kontrolliert werden, damit die Informationen permanent und konsistent sind.

Wann Multithreading verwenden?

Immer dann, wenn eine Aufgabe mehr als eine Sekunde in Anspruch nehmen kann, sollten sie darüber nachdenken, ob sich der Programmieraufwand für die Erzeugung eines Threads lohnt.

z.B. Darstellung einer einfachen Liste aller Rechner im Netzwerk (man weiss nicht wie lange das es geht, von NW abhängig) → NW Kommunikation

Niemals ein Multithreading-Programm, wenn Single-threaded ausreicht!

- **Antwortverhalten**
- **Performance**
- **Einfachheit des Programmcodes**

Grundbegriffe zum Thema Threads

Multhithreading bedeutet Parallelität und damit eine Unbestimmtheit, die ein rein sequenzielles Programm nicht hat-

Ein Thread läuft immer im Adressraum seines Prozesses ab, er ist also „Prozess affin". Er verfügt über einen eigenen Stack, aber jeder Thread hat einen uneingeschränkten und gleichen Zugriff auf den Heap. Das bedeutet, dass ein Thread zu jedem Zeitpunkt jedes beliebige Byte auf dem Heap manipulieren kann. Und zwar unabhängig davon, ob andere Threads dadurch beeinflusst werden. Das Betriebssystem übernimmt hier keine Kontroll- und Schutzfunktion.

Beim gleichzeitigen Zugriff auf ein Objekt kann es zu Fehlern kommen, wenn das Objekt parallel Verwendung nicht unterstützt. Ein Objekt, das auf mehreren Threads gleichzeitig benutzt werden darf, ohne dass es deswegen zu Fehlern kommt, wird threadsicher oder threadsafe genannt.

Kontextwechsel

Das Windows-System stellt einem Prozess die Ressource CPU nur für ein kurzes Zeitintervall, dem Quantum, zur Verfügung. Genauer gesagt wird nicht dem Prozess, sondern einem Thread eines Prozesses die CPU zugeteilt. Nach Ablauf des Intervalls erhält ein anderer Thread die CPU – und so weiter, bis der erste Thread wieder an der Reihe ist (Round-Robin-Verfahren).

Dieser Zuteilungswechsel der CPU von einem Thread zu einem anderen wird Kontextwechsel (Context Switch) genannt.

- Thread-Kontextwechsel
- Prozess-Kontextwechsel
  (braucht mehr Zeit)

Die Aufgabe, diesen Wechsel durchzuführen, wird vom sogenannten Scheduler übernommen.

Thread-Priorität

Der Scheduler teilt die aktiven Threads in Gruppen gleicher Priorität ein. Es werden alle Threads einer Gruppe abgearbeitet, bis entweder alle Threads dieser Gruppe beendet sind (oder durch einen Prioritätenwechsel die Gruppe verlassen haben) oder aber eine nicht leere Gruppe mit höherer Priorität existiert.

Sobald ein Thread mit höherer Priorität aktiv wird, wird der laufende Thread unterbrochen und der entsprechenden Gruppe mit höherer Priorität wird die CPU zugewiesen.

Thread-Zustände

Running

Der Thread wird aktuell auf der CPU ausgeführt. Pro CPU kann nur ein Thread diesen Zustand einnehmen.

Ready

Der Thread wartet darauf, eine CPU zugeteilt zu bekommen. Der Thread ist nicht im Zustand Running weil die CPU nicht frei ist. Alle Threads im Zustand Ready sind entsprechend ihrer Priorität gruppiert. Nur Threads aus der obersten nicht leeren Prioritätsgruppe können in den Zustand Running wechseln.

Wait

Der Thread wartet auf die Freigabe einer Ressource oder auf ein Ereignis und steht dem Scheduler dadurch nicht zur Verfügung. Tritt das Ereignis ein, wechselt er den Zustand. Warten mehrere Threads gleicher Priorität auf das Ereignis oder die Ressource, so garantiert das Betriebssystem nicht, dass der Thread, der länger wartet, als erster in den Zustand Running tritt.

Initialized

Der Thread wurde initialisiert, aber noch nicht gestartet.

Standby

Der Thread wird nachfolgend die CPU-Ressoruce erhalten. Nur ein Thread kann in diesem Zustand sein.

Transistion

Der Thread wartet auf eine Ressource, die nicht die CPU ist. Er wartet nicht auf ein Synchronisationsobjekt oder –ereignis, sondern bspw. auf das Nachladen des benötigten Speichers von der Festplatte.

Terminated

Der Thread wurde beendet.

[Diagramm mit Wechsel VISIO]

Multithreading bei .NET

Die Common Language Runtime (CLR) des .Net-Framworrks unterteilt einen Prozess in sogenannte Anwendungsdomänen.
Ein Thread innerhalb einer Anwendungsdomäne ist ein eigenständig ablaufender Programmteil, der im Adressraum der Anwendungsdomäne abläuft und dort einen eigenen Stack besitzt.
Die Threads einer Anwendungsdomäne bekommen, abhängig von ihrer Priorität, Rechenzeit zugeteilt.

Da alle Threads im selben Adressraum arbeiten, können sie auf dieselben Variablen bzw. Objekte zugreifen.

Möglichkeiten des Multithreadings bei .NET

Multitasking

Mehrere gleichartige Aufgaben, die gleichzeitig unterschiedliche Arbeiten bearbeiten und gegenseitig keinen Einfluss nehmen.

Timer

Zyklisch wiederkehrende Aufgaben.
Gleichzeitiger Zugriff auf gemeinsame Daten im Prozessraum ist möglich.

Multithreading

Beliebige wiederkehrende Aufgabenteilung und gleichzeitiger Zugriff auf prozessinterne Daten (Heap) aus verschiedenen Threads möglich.

Asynchrone Verarbeitung

Spezialform des Multithreadings. Die .NET-Bibliothek unterstützt mit vielen Klassen die asynchrone Verarbeitung.

Hintergrundverarbeitung (Backgroundworker)
Spezialform des Multithreadings für vereinfachte Handhabung der Threads.

Client/Server
Bearbeitung der Aufgaben auf zentralen Computern. Funktioniert nur mit Interprozesskommunikation (z.B. Remoting)

Wichtige Eigenschaften und Methoden der Klasse Thread
→ Multitasking Seite 18

Sperren und synchronisieren von Threads

Greifen mehrere Threads auf die gleichen Objekte zu, kann es zu undefinierten Zuständen der Objekte kommen, wenn die Threads nicht synchronisiert werden. Das geschieht bspw. dann, wenn ein Thread unterbrochen wird, der gerade den Zustand eines Objekts verändert. Weitere Beispiele sind:

- Mehrere Threads versuchen gleichzeitig, Daten in eine Datei zu schreiben
- von einem Konto soll ein Betrag abgebucht und auf einem anderen Konto gutgeschrieben werden
- Erzeuger/Verbraucher-Problem: Erzeuger bringen ihre produzierten Waren in ein Lager, und Verbraucher entnehmen die gelagerte Ware.
- „Philosophen-Problem“
- vier Autos an einer Kreuzung mit Rechtsvortritt

Selbst wenn die gemeinsam zu bearbeitende Information ein einfacher Ganzzahl-Wert ist, kann es sein, dass dieser plötzlich nicht das gewünschte Resultat aufweist. Bei einfachen Werten gibt es sogar mehrere Begründungen dafür:

- Wird der Compiler im Prozessorregister gelagert, kann es sein, dass der Wert anstelle einmal (wie programmiert) effektiv n-mal (einmal pro Thread) im laufenden Programm vorhanden ist, da die Prozessorregister pro Thread verwaltet werden.
- Operationen auf einfachen Werten können sich in der Effektiven Verarbeitung über mehrere Prozessoranweisungen hinweg verteilen. Wir ein Thread inmitten einer solchen Verarbeitung unterbrochen, kann der Datenstand korrumpiert werden.

Möglichkeiten der Synchronisierung bei .NET

volatile
Das C#-Schlüsselwort definiert, dass eine Variable nicht in einem Prozessorregister gespeichert wird und dass stets der aktuelle Wert vorliegt.

Monitor
Ein Mittel der .Net-Klassenbibliothek für die Synchronisation der Codeausführung.

lock()
Wie Monitor, aber Sprachmittel von C#

Mutex
Mittel der .Net-Klassenbibliothek für die Mutual-Exclusive-Synchronisation des Zugriffs mehrerer Threads auf Daten

Semaphor
Mittel der .Net-Klassenbibliothek für die Synchronisation des Zugriffs mehrerer Threads auf Daten. Der Semaphor unterscheidet sich vom Mutex dadurch, dass keine Exklusivität, wohl aber eine maximale Menge von gleichzeitigen Zugriffen kontrolliert werden kann.

lock()

Mit Hilfe der Anweisung lock(„Ausdruck“) kann in einem Code ein kritischer Bereich definiert werden. Ein kritischer Bereich wird im Multitasking garantiert nicht unterbrochen. Die Kontrolle über den kritischen Bereich erfolgt über das Objekt, dessen Referenz der Anweisung als Parameter übergeben wird.
Das Sperren von Objekten führt zu einem Leistungsverlust, weil die Threads auf eine Freigabe gesperrter Objekte warten müssen.

Monitor

Die Klasse Monitor ist ein Äquivalent zur Anweisung lock() der .NET-Klassenbibliothek. Es muss beachtet werden, dass eine erstellte Sperre in jedem Fall aufgehoben wird (→ finally der Ausnahmebehandlung). Ein infolge Ausnahme nicht freigegebenes gesperrtes Objekt kann unter Umständen zu einem Deadlock führen.

Mutex

Mit Hilfe eines Mutexobjekts kann der exklusive Zugriff auf eine Information kontrolliert werden. Ein Objekt der Klasse Mutex wird auch einfach als Mutex (Mutual exclusion) bezeichnet. wenn ein Thread einen von ihm angeforderten Mutext werhält, wird jeder weitere Thread, der diesen Mutex abrufen will, so lange angehalten, bis der erste Thread den Mutex freigibt.

Semaphor

Das Semaphor erlaubt den kontrollierten Zugriff einer definierbaren Menge von Threads auf eine Ressource. Die Anzahl der verfügbaren Plätze wird bei der Konstruktion des Semaphors angegeben und anschliessend mit denselben Methoden wie bei der Klasse Mutex kontrolliert.

Ein Semaphor erlaubt eine Zugriffsbeschränkung auf mehrere Zugriffe gleichzeitig.

Interprozess-kommunikation

Allgemein

Unter Interprozesskommunikation (engl. IPC, Interprocesscommunication) versteht man Methoden zum Informationsaustausch (Datenübertragung, von nebenläufigen Prozessen oder Threads).
Im engeren Sinne versteht man unter IPC die Kommunikation auf demselben Computer, deren Speicherbereiche aber strikt voneinander getrennt sind (Speicherschutz).
Im weiteren Sinne bezeichnet IPC aber jeden Datenaustausch in verteilten Systemen, angefangen bei Threads, die sich ein Laufzeitsystem teilen, bis hin zu Programmen, die auf unterschiedlichen Rechnern laufen und über das Netzwerk kommunizieren.
Für die Kommunikation ist dabei eine geeignete Prozesssynchronisation (lock, Monitor, Semaphor, Mutex) notwendig.

Datenströme

Datenströme sind Kommunikationskanäle, die Dateneinheiten (einzelne Bytes oder auch komplexe Objekte) sequentiell von einem Prozess zu einem anderen weiterreichen-
Die am meisten verbreitete Variante dieser Kommunikation sind so genante **Pipes**.
Auch verbindungsorientierte NW-Protokolle (besonders TCP) verhalten sich wie Datenströme und können zur IPC benutzt werden (man spricht auch von Sockets).

Ereignisse

Manche Betriebssysteme erlauben Prozessen auf Ereignisse in anderen Prozessen zu reagieren – solche Ereignisse können auch als Nachrichten verstanden werden, die ein Prozess einem anderen sendet.

Funktionsaufrufe

Die aus Sicht des Programmierers einfachste Art auf einen anderen Prozess zuzugreifen ist, auf einzelne Funktionen des Prozesses Zugriff zu haben, als seien es Funktionen innerhalb des Prozesses – also ein Verfahren, das es für den Programmierer transparent macht, wo sich die Implementation einer anderen Funktion befindet – im eigenen Prozess oder in einem anderen, der sich möglicherweise sogar auf einem anderen Computer befinden kann.

Shared Memory

Hier werden Bereich des Hauptspeichers für den Zugriff durch mehrere Prozesse reserviert, wobei das Lesen und Schreiben durch einen Mutex-Mechanismus geregelt wird.

.Net Remoting

Framework für die Kommunikation zwischen Objekten, die sich in verschiedenen Anwendungsdomänen oder Prozessen bzw. auf unterschiedlichen Computern befinden.
Es ermöglicht sozusagen die Kommunikation zwischen Applikationen, die lokal am selben Computer, auf verschiedenen Computern im gleichen Netzwerk oder sogar auf Computern über mehrere Netzwerke hinweg liegen.

Proxyobjekte

Proxyobjekte werden bei der Aktivierung eines Remoteobjektes durch einen Client erstellt. Das Proxyobjekt verhält sich wie ein Stellvertreter des Remoteobjekts. Es stellt sicher, dass alle für den Proxy gesendeten Aufrufe an die richtige Remoteobjektinstanz weitergeleitet werden.

Channel

Auf entfernte Objekte wird mittels so genannten Channels zugegriffen. Channel transportieren die Nachricht zum Objekt hin und vom Objekt weg. Es gibt zwei Channel, die wichtig sind: TcpChannel und HttpChannel.

verteiltes Objekt

Objekt muss von MarshalByRefObject abgeleitet werden.

Es gibt zwei unterscheidbare Vorgangsweisen, wie entfernte Objekte erstellt werden können:

- Objekterzeugung durch Server
- Objekterzeugung durch Clients
    - Single-call object
    - singleton object

Glossar

RPC

Remote Procedure Call ist eine Technik zur Realisierung von IPC. Sie ermöglicht den Aufruf von Funktionen in anderen Adressräumen.

Corba

CORBA = Common Object Request Broker Architecture

- Spezifikation für eine objektorientierte Middleware
- plattformübergreifende Protokolle und Dienste

DCOM

DCOM = Distribute Component Object Model

- objektorientiertes RPC-System, das auf dem DCE-Standard basiert
- wurde von Microsoft definiert, um die Technologie COM übers Netzwerk kommunizieren zu lassen.

DCOP

DCOP = Desktop Communication Protocol

- Client-Server Protokoll, das das Interprozessing zwischen KDE-Anwendungen unterstützt

DDE

DDE = Dynamic Data Exchange

- Protokoll für den Datenaustausch zwischen verschiedenen Anwendungsprogrammen

SOAP

SOAP = Simple Object Access Protocol

- Netzwerkprotokoll, mit dessen Hilfe Daten zwischen Systemen ausgetauscht und RPCs durchgeführt werden können.

Webservice

Dienste, die für die Kommunikation zwischen Computern konzipiert und über verschiedene Plattformen hinweg interoperabel sind.

SOA

Eine service-orientierte Architektur stellt eine flexible, anpassbare IT-Architektur dar, die die verteilte Datenverarbeitung unterstützt.

Remoting

.NET Remoting ist ein umfassende, erweiterbares Framework für die nahtlose Kommunikation zwischen Objekten, die sich in verschiedenen Anwendungsdomänen oder Prozessen bzw. auf verschiedenen Computern befinden.

RMI

Remote Method Invocation ist ein Aufruf einer Methode eines entfernten Java-Objektes und realisiert die JAVA-eigene Art des RPC.

Mehrschichtige Softwarearchitektur

engl. multi-tier architecture

Allgemein

3-schichtige Architektur

Die 3-schichtige Architektur (three-tier architecture) besteht aus:

1. GUI-Schicht
2. Fachkonzeptschicht und
3. Datenhaltungsschicht

Bei der Drei-Schichten-Architektur sind zwei Ausprägungen möglich:

1. **strenge Drei-Schichten-Architektur:** GUI-Schicht kann nur auf die Fachkonzeptschicht zugreifen, und diese wiederum nur auf die Datenhaltungsschicht.

   Vorteil: GUI-Schicht ist nur von der Fachkonzeptschicht abhängig und nicht von der gewählten Speicherung der Daten.

2. **flexible Drei-Schichten-Architektur:** GUI-Schicht darf zusätzlich auf die Datenhaltungsschicht zugreifen

   Vorteil: grössere Flexibilität, bessere Performance

Nachteil: geringere Wartbarkeit und geringere Portabilität

Die grundlegende Idee der Drei-Schichten-Architektur ist, dass keine andere Schicht direkt auf die Benutzungsoberfläche zugreifen darf.

Dies konsequente Trennung ermöglicht eine getrennte Entwicklung von Benutzungsoberfläche und Fachkonzept und einen leichteren Austausch der Benutzungsoberfläche

Änderungen können auf der Benutzungsoberfläche durch zwei Arten sichtbar gemacht werden:

1. Polling
2. indirekte Kommunikation (notify mittels observer pattern

Polling
Benutzeroberfläche sendet in regelmässigen Intervallen Nachrichten an die Fachkonzeptobjekte, um Änderungen, welche sich auf die Oberfläche auswirken, abzufragen.
➔ ungünstige Auswirkung auf die Performance

indirekte Kommunikation
Kommunikation mittels Observer-Muster. Man spricht von Benachrichtigung (notify). Die Oberfläche holt sich daraufhin selbständig die Daten.

2-schichtige Architektur

Der Nachteil dieser Architektur ist die feste Verzahnung von GUI und Fachkonzept. Änderungen der Benutzungsoberfläche oder gar deren kompletter Austausch sind aufwändig durchzuführen.

Client-Server

→ siehe Block „Mehrschichtige Softwarearchitektur“ Seite 3

Client-Server-Architektur

Die Client-Server Architektur organisiert sich in funktionellen Ebenen, die eine Lösung im Detail implementiert.
Durch die systematische Unterteilung der anstehenden Arbeiten entstehen verschiedene Möglichkeiten des Aufbaus von Client-Server-Architekturen.

Die 7 Ebenen

→ siehe Block „Mehrschichtige Softwarearchitektur“ Seite 4/5