In diesem Kapitel:
- *Design Tätigkeiten*
  - *Architektur Design*
  - *Detail Design*
    - *Packages*
    - *Klassen*
    - *Interfaces*
    - *GUI*
  - *Implementation*
  - *Test und Verteilung*
  - *Ausblick*

18
ROP / RUP Design Case Study Bibliothek

Zur Illustration und besseren Vorbereitung auf das Vordiplom wollen wir ein Projekt von Anfang bis zu Ende mit der Methode (ROP / RUP) behandeln.

18.1. Anwendungsbeispiel : UML und (Rational) Unified Process

Nachdem wir die Analyse angeschlossen haben, wenden wir uns der technischen Seite zu und beschreiben Design und Implementation (Konstruktion) für die Bibliotheksanwendung.

Das Vorgehen ist das Gleiche wie in der Analyse; nur die Inhalte sind verschieden.

18.2. Design Tätigkeiten

Designtätigkeiten beschreiben das *WIE.* Im Design müssen wir die einzelnen Klassen Im Detail festhalten. Es muss eine Vorlage erstellt werden, auf deren Basis das System implementiert werden kann. Die in der Analyse gefundenen Klassen werden verfeinert, nicht neu gesucht.

In der Regel, speziell bei umfangreichen Projekten, unterteilt man Design in zwei Segmente:
- *Architektur Design*
  dabei geht es um eine "high level" Darstellung in der Regel auf Package Ebene (Subsysteme). Es werden Abhängigkeiten zwischen den Paketen festgehalten und die Kommunikationsmechanismen beschrieben..
  Das Ziel ist es, eine einfache, klare Architektur zu finden, die auch noch erweiterbar ist. Kopplung und Kohärenz sind weitere Merkmale, die erstrebenswert sind.
  Gute Architekturen lassen sich nur schwer finden. Deswegen hat man angefangen, gelungene Architekturen zu katalogisieren: die Architektur Patterns.
- *Detail Design*
  Die Pakete werden verfeinert und die Details werden beschrieben. Die verschiedenen Darstellungen (Sequenzdiagramm, Zustandsdiagramm, Klassendiagramm) visualisieren die Details der Abläufe.

18.3. Architektur Design

Eine gute Architektur ist die Basis für ein erweiterbares System. Es hat sich gezeigt, dass in der Regel die zwei Betrachtungen "Benutzerseite, Applikationsdomäne" und die "Technische Ebene" strikt zu trennen sind.

Beide Seiten des Systems müssen relativ leicht und unabhängig voneinander geändert werden können.

Auf der technischen Seite muss man versuchen Standard Pakete finden, um möglichst effizient Systeme bauen zu können.

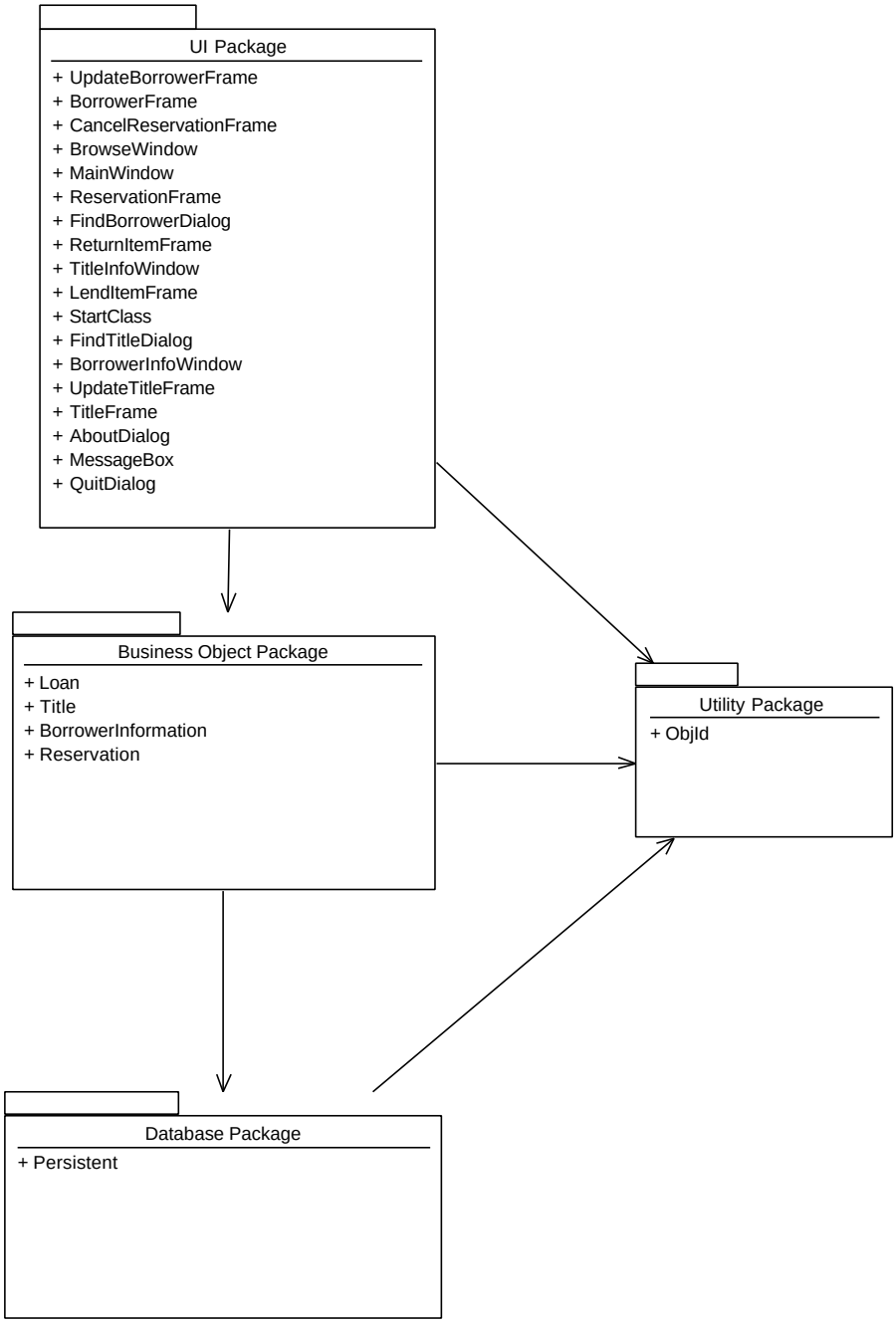
In unserem Bibliothekssystem haben wir folgende Pakete oder Subsysteme:

- *User Interface Package*
  Dieses umfasst alle Benutzerschnittstellen, also alle Masken, alles was dem Benutzer sichtbar ist.
  Diese Klassen basieren auf den AWT Java Klassen. Es werden also keine Swing Komponenten eingesetzt. Das System sieht deswegen auch entsprechend einfach aus, vom GUI her.
  Die Masken müssen es dem Benutzer gestatten, die Objekte zu erfassen, zu speichern und zu mutieren.
- *Business Objekt Systeme*
  In diesem Paket werden die Domänen Klassen zusammen gefasst, zum Beispiel "Title" (wegen der Implementation geschieht ein "Sprachwechsel" von deutsch auf englisch, allerdings wurde die gesamte Dokumentation des Systems in deutsch erstellt und so belassen; die Klassen, Actors sollten eigentlich alle in englisch sein, aber es gibt auch dort ein paar deutsche "Überreste").
  Das BO Paket arbeitet eng mit dem Datenbank Paket zusammen (DB Package), welches für die Speicherung der Daten (schreiben und lesen) verantwortlich ist.
- *Database Package*
  Dieses Paket stellt den andern Paketen Dienste zur Verfügung, damit der Objektzustand bzw.. die Zustände aller Objekte abgespeichert und wieder aktiviert werden können.
  Die aktuelle Version speichert die Objekte mit Hilfe von Dateien ab. Zuständig dafür ist die Persistent Klasse.
- *Utility Package*
  Das Utility Paket enthält alle Klassen, die irgendwelche Dienste für andere Klassen erbringen. Aktuell ist das Paket fast leer: die einzige Klasse ist ObjId, die Klasse, welche die Beziehung zu den persistenten Objekten darstellt, mit Hilfe der Objekt Id. Die Klasse wird von Klassen in den Paketen "User Interface", "Business Objekt" und dem Datenbank Paket eingesetzt.

Die Aufteilung in diese Pakete ist nicht eindeutig. Es sind unterschiedliche Aufteilungen möglich. In unserem Beispiel ist die Definition einzelner Pakete eher künstlich. In der Regel sollte kein Paket definiert werden, welches lediglich eine Klasse enthält.

Aber funktional, von den Aufgaben und den Verantwortungen her gesehen, macht die Aufteilung Sinn.

Und so sieht das Design der Pakete aus:



18.4. Detail Design

Das Ziel des Detail Designs ist es, technische Klassen zu finden und zu definieren, Klassen in den Paketen UI und DB.

Die Spezifikation dieser Klassen muss so detailliert sein, und mit Hilfe der Sprachkonstrukte, dass sie auch implementiert werden können.

Als Hilfsmittel dienen die Klassendiagramme, Sequenzdiagramme und Zustandsdiagramme.

Wir setzen also die gleichen Hilfsmittel wie in der Analyse ein! Aber hier im Design beschreiben wir mehr Details. Die Beschreibung ist auch technisch korrekter, präziser.

Die Anwendungsfälle (Use Cases) aus der Analyse dienen der Verifikation, also der Prüfung auf Vollständigkeit und Korrektheit des Designs. In unserem Beispiel wurden Sie einfach aus der Analyse ins Design Modell übernommen, bzw.. das Analyse Modell wurde unter neuem Namen abgespeichert und als Design Modell weiter verfeinert und modifiziert.

Die Sequenzdiagramme zeigen auf, wie die Pakete technisch realisiert werden sollen.

18.4.1. Das Datenbank Paket

Unsere Beispielapplikation muss die Daten in irgend einer Form speichern können, sonst ist sie nicht brauchbar.

Deswegen haben wir einen Datenbank-Layer hinzugefügt.

Aber statt eine echte Datenbank einzusetzen (das wäre in der Praxis wohl nötig), und weil wir unsere Applikation portabel gestalten und realisieren möchten, verzichten wir darauf, eine bestimmte Datenbank zu verwenden.

Die Lösung besteht darin, eine Datenspeicherungsklasse im Datenbank Paket zu definieren. Details der Klasse sind den Klassen, Objekten und Paketen verborgen. Sie rufen lediglich Methoden der Datenbank Klassen auf. Wie diese implementiert sind, bleibt verborgen, Ihnen auch!

Die gängigen Operationen (Methoden) wie "insert", "update", "delete" stehen zur Verfügung. Die Klasse Persistent ist dafür zuständig.

Und so sieht der "Header" dieser Klasse aus:

```
/
//  Persistent.java: Oberklasse für die Serialisierung aller Objekte
//

package db;
import util.*;
import java.io.*;
import java.net.*;

public abstract class Persistent
```

Die Klasse muss also jeweils noch implementiert werden. *Abstract* besagt bekanntlich, dass **alle** Methoden in den Unterklassen realisiert werden müssen.

Die Methoden, die zwingend implementiert werden müssen, sind *read* und *write.* Weitere Methoden können natürlich auch definiert werden, zum Beispiel eine allgemeine Suchmethode.

Im Zusammenhang mit der Objektspeicherung spielt die Klasse ObjId eine wichtige Rolle: jedes Objekt wird in unserer Beispielapplikation mit Hilfe einer Id identifiziert.

Ein Objekt kann dann mit Hilfe der Methode *getObject* der Klasse *Persistent* ein- und ausgelagert werden, inklusive Typenkonversionen, Typenprüfungen,...

Da die ObjId Klasse genereller einsetzbar ist, haben wir sie in ein separates Package ausgelagert. Sie wird zum Beispiel auch von Klassen im UI Package eingesetzt. Wenn wir die Klasse ObjId im Paket DB speichern würden, dann müsste also das User Interface die Datenbank bemühen. Das ist zwar denkbar, auch bei grossen Applikationen, aber wenig flexibel.

Mutation unserer Objekte sind in unserem Beispiel ineffizient implementiert:
bei Mutationen wird gleich eine neuer Record geschrieben, der alte Datensatz wird wird einfach als gelöscht gekennzeichnet (keine Mutation der Inhalte der Dateien). Der neue Datensatz wird am Ende der Datei eingefügt.

Der Grund liegt in den Tiefen der Systemarchitektur der Datenbanken:
beim Updaten kann sich die Länge des Datensatzes ändern; der neue Datensatz hätte also unter Umständen keinen Platz mehr!

Das Persistent Klassen Interface (die Methoden) sind so allgemein gehalten, dass das spätere Einbinden einer Datenbanklösung möglich sein müsste.

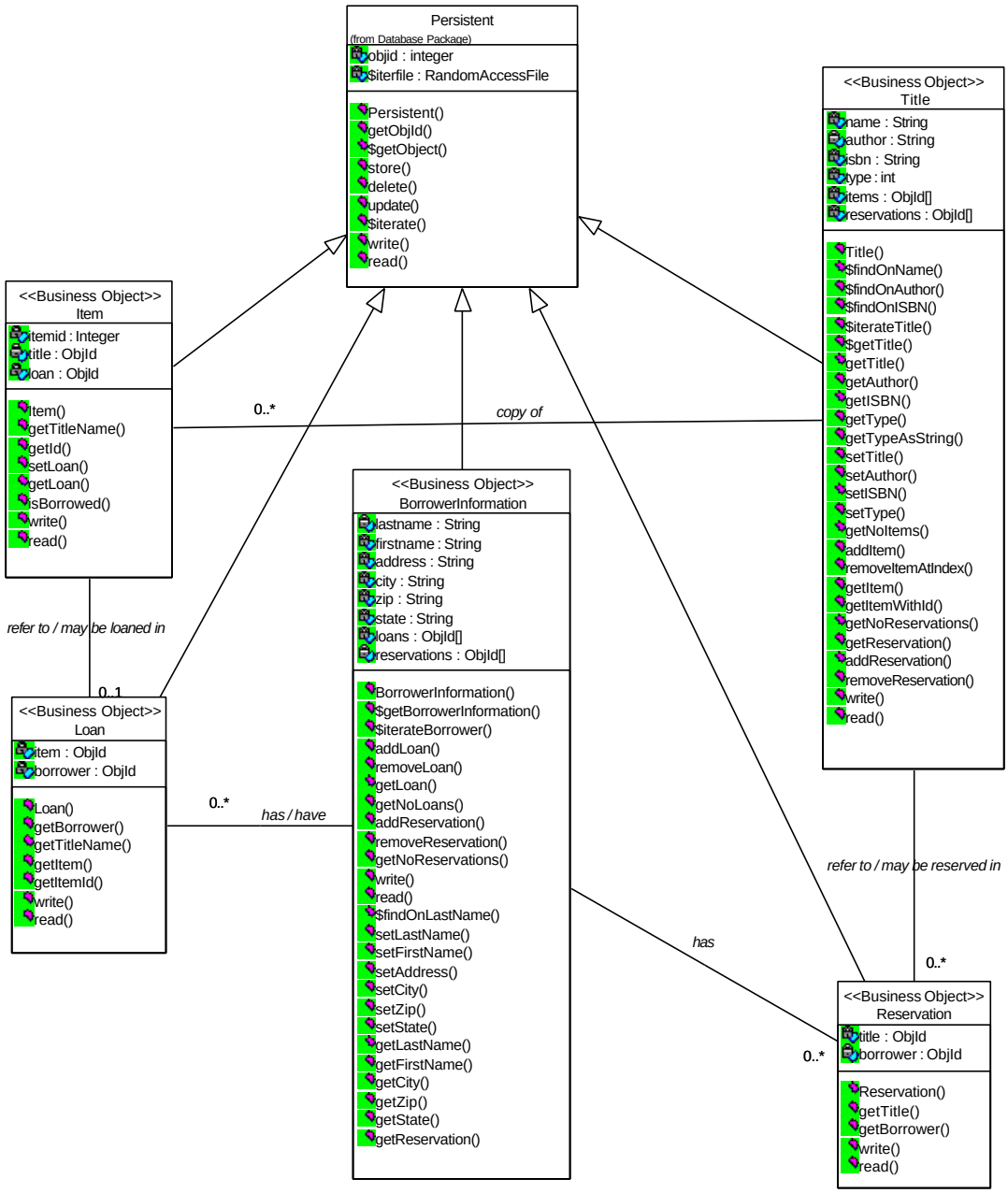
Das einzige was geändert werden müsste, wäre die Persistent Klasse.

18.4.2. Business Objekt Paket

Das BO Package entspricht den Domänen Klassen des Analyse Modells. Alle Klassen und deren Beziehungen werden aus der Analyse übernommen und weiter verfeinert und mit Java Terminologie angereichert.

Eine Finesse ergibt sich aus der Konstruktion des DB Packages :
die Klassen des BO Packages implementieren die Persistent Klasse.

Das folgende Klassendiagramm zeigt dementsprechend wesentlich mehr Details und Datentypen als das Analyse Diagramm:

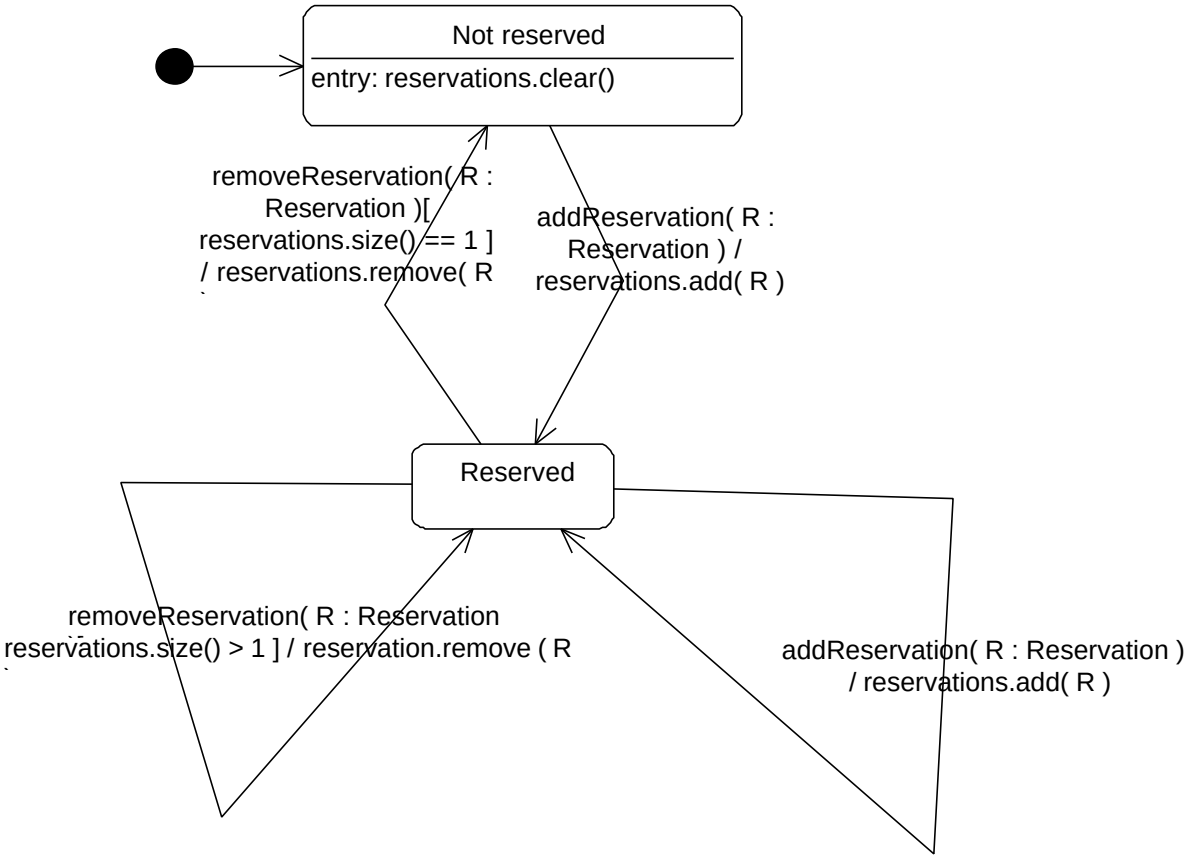

Persistent
(from Database Package)
objid : integer
$iterfile : RandomAccessFile
Persistent()
getObjId()
$getObject()
store()
delete()
update()
$iterate()
write()
read()
<<Business Object>>
Item
itemid : Integer
title : ObjId
loan : ObjId
Item()
getTitleName()
getId()
setLoan()
getLoan()
isBorrowed()
write()
read()
<<Business Object>>
Title
name : String
author : String
isbn : String
type : int
items : ObjId[]
reservations : ObjId[]
Title()
$findOnName()
$findOnAuthor()
$findOnISBN()
$iterateTitle()
$getTitle()
getTitle()
getAuthor()
getISBN()
getType()
getTypeAsString()
setTitle()
setAuthor()
setISBN()
setType()
getNoItems()
addItem()
removeItemAtIndex()
getItem()
getItemWithId()
getNoReservations()
getReservation()
addReservation()
removeReservation()
write()
read()
0..*
copy of
refer to / may be loaned in
0..1
<<Business Object>>
Loan
item : ObjId
borrower : ObjId
Loan()
getBorrower()
getTitleName()
getItem()
getItemId()
write()
read()
0..*
has / have
<<Business Object>>
BorrowerInformation
lastname : String
firstname : String
address : String
city : String
zip : String
state : String
loans : ObjId[]
reservations : ObjId[]
BorrowerInformation()
$getBorrowerInformation()
$iterateBorrower()
addLoan()
removeLoan()
getLoan()
getNoLoans()
addReservation()
removeReservation()
getNoReservations()
write()
read()
$findOnLastName()
setLastName()
setFirstName()
setAddress()
setCity()
setZip()
setState()
getLastName()
getFirstName()
getCity()
getZip()
getState()
getReservation()
has
refer to / may be reserved in
0..*
0..*
<<Business Object>>
Reservation
title : ObjId
borrower : ObjId
Reservation()
getTitle()
getBorrower()
write()
read()

Einige Schwächen der jetzigen Version:

- die Rückgabefristen werden nicht überprüft:
  die dreissig Tage Ausleihfrist werden zwar festgehalten; aber bei einer Überschreitung geschieht nichts.
- Magazine und Bücher (Gegenstände) werden gleich behandelt.
  Die Subklassen der Titel Klasse aus der Analyse Klasse werden deswegen nicht implementiert.

Die Zustandsdiagramme aus der Analyse wurden ebenfalls weiter verfeinert und ins Englische übersetzt, wenigstens teilweise.

Betrachten wir zum Beispiel das Zustandsdiagramm für die Titel (Title) Klasse:



Reservationen werden mit Hilfe eines Vektors implementiert. Alle Reservationen werden als Elemente der Vector Datenstruktur abgespeichert. Die Vektorlänge ist = 0 falls keine Reservationen vorliegen.

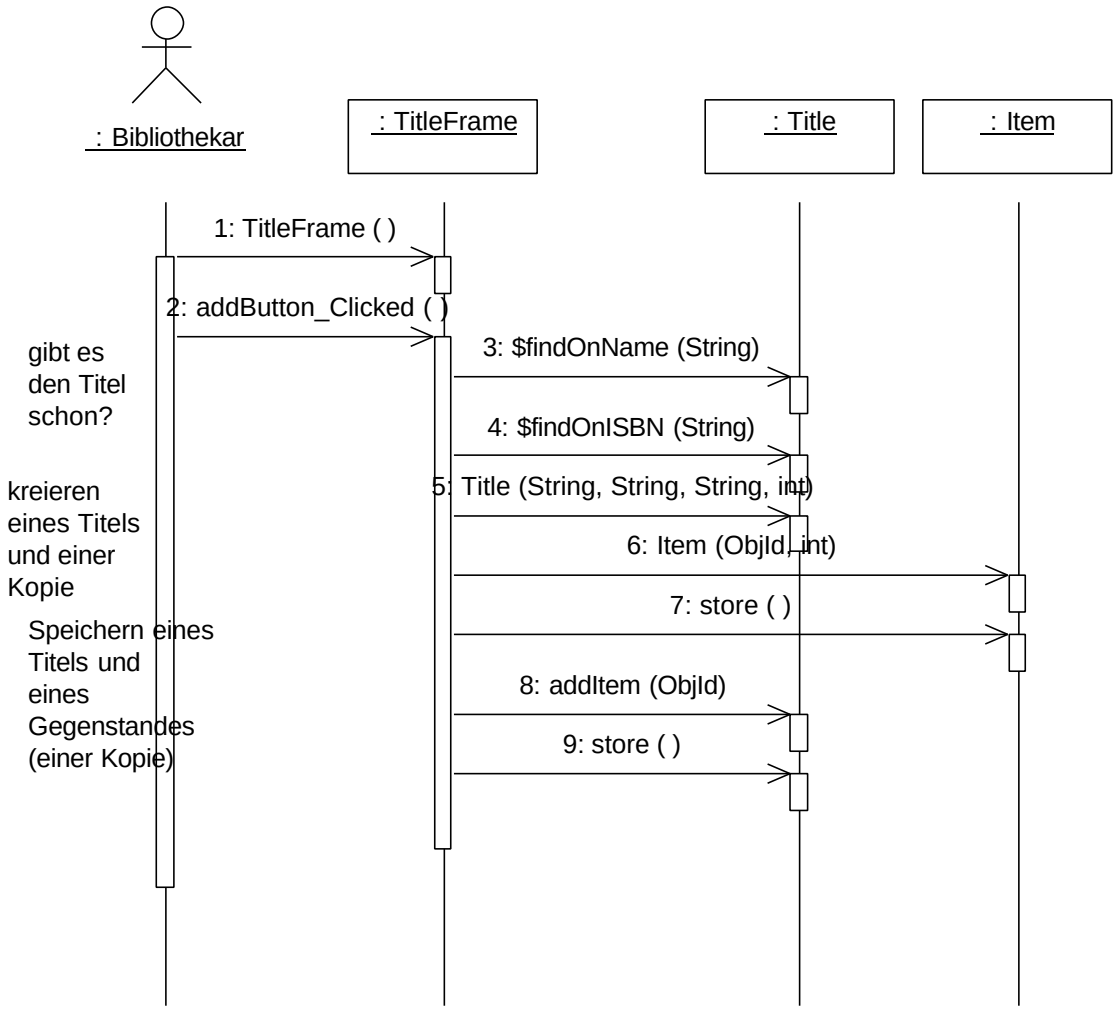
18.4.3. Das User Interface Package

Das UI Package stellt den obersten Layer unser Applikation dar (unsere Architektur verwendet das "Layer Architecture" Pattern). Das UI Package basiert auf AWT und ist somit universeller einsetzbar als ein Swing Package.

Bei Benutzerschnittstellen darf man sich zu sehr gleich am Anfang mit den Klassen und Objekten auf einem sehr tiefen Level beschäftigen; sonst verliert man die Übersicht! Es ist zwar wichtig zu wissen, dass eine Oberfläche sich aus verschiedenen Objekten zusammensetzt; aber die Details sind eher für den GUI Package Designer wichtig!

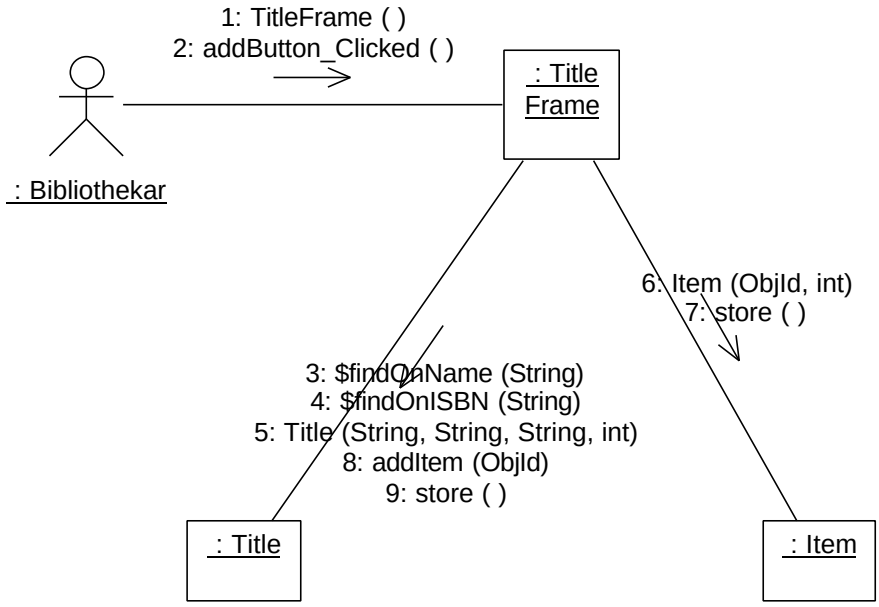
Das dynamische Modell wird mit Hilfe von Sequenzdiagrammen beschrieben. Die Kollaborationsdiagramme sind auch hier wenig aussagekräftig, aber besser geeignet, um sich mit dem Benutzer über das Design zu unterhalten!

In diesem Teil ist die Mischung von Englisch und Deutsch besonders ausgeprägt, sorry!
Alle Beschreibungen sind in Deutsch, die Diagramme haben deutsche Namen, die Inhalte sind aber bereits (weitestgehend) übersetzt.



Die Sequenzdiagramme werden auch laufend (im Verlaufe des Projektes) nachgeführt. Zum Teil werden Verbindungen und Methodenaufrufe erst in der Implementierung sichtbar.

Hier das gleiche Diagramm noch als Kollaborationsdiagramm:



Im Kollaborationsdiagramm werden die "Methoden" einfach durch nummeriert. Der zeitliche Ablauf lässt sich somit auf Grund der Nummern nach vollziehen.

Komplexe Sequenzdiagramme sollte man nicht in Kollaborationsdiagramme umwandeln; die Übersicht ist kaum mehr gegeben.

18.4.3.1. User Interface Design

Die Kriterien, die man anwenden muss beim UI Design werden wir hier nicht besprechen. Auf dem Java Site von Sun steht ein ganzes Buch (Alpha Version) "Java Look and Feel Design Guidelines" gratis zum runter laden.

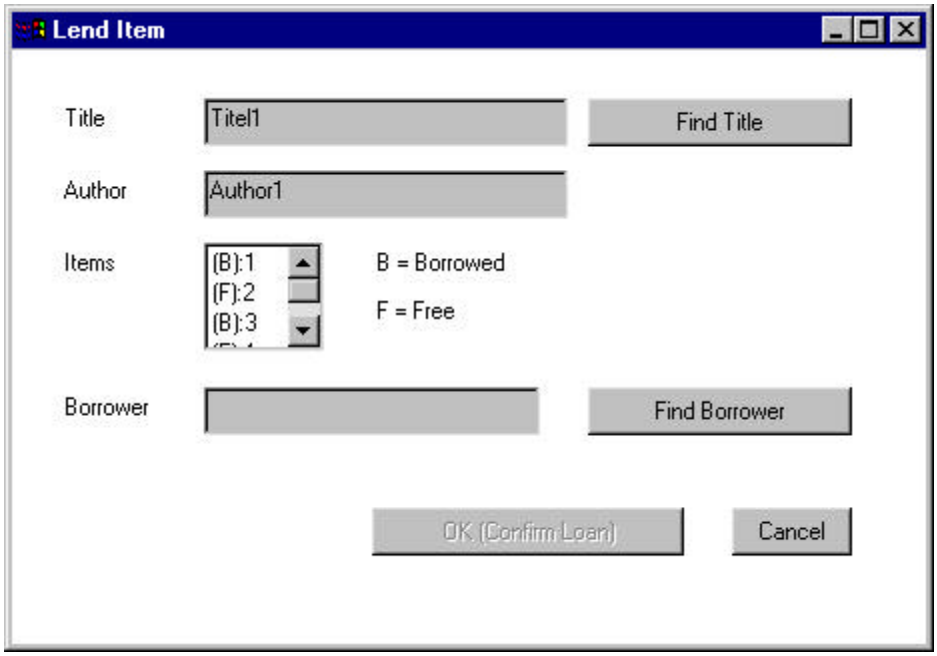
Unser UI basiert auf den Use Cases und wurde wie folgt aufgeteilt:

- *Funktionen:*
  Fenster zum Ausführen der Hauptaufgaben wie Ausleihe, Rückgabe, Reservationen
- *Informationen:*
  Fenster zur Anzeige von Informationen, also Titel- und Ausleiher-Übersichten
- *Wartung / Unterhalt:*
  fenster für die Datenpflege, wie erfassen von Titeln, Ausleihern, Wartung der Stammdaten

Alle Fenster wurden in Symantec Visual Cafe entwickelt, beziehungsweise gezeichnet, da diese Entwicklungsumgebung ein einfaches drag and drop Interface besitzt.

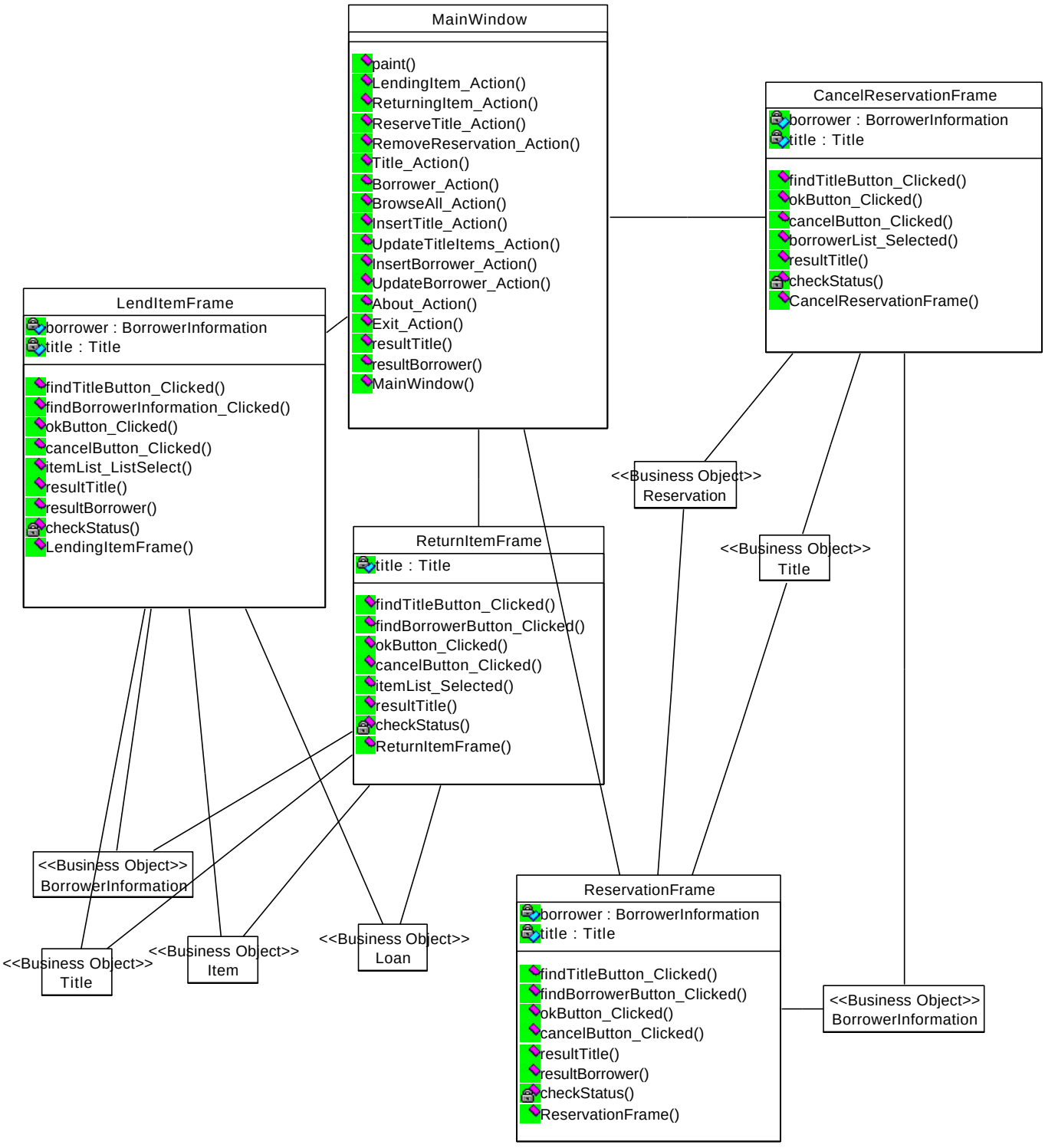
Da ich in der Zwischenzeit die Entwicklungsumgebung zweimal geändert habe (DevStudio von Microsoft, dann JBuilder von Borland), werde ich diese Teile wohl demnächst anpassen. Und hier ein Screen Snapshot des obersten, des Start Fensters:

und eines Applikationsfensters:



(die Programmteile sind 98% in Englisch).

Das UI Klassen der Implementierung für den Funktionsteil sieht wie folgt aus:



18.5. Implementation

Die Implementation der Applikation geschieht primär in der Konstruktionsphase.

Verschiedene Teile der Applikation haben Sie bereits im Abschnitt UI Design gesehen (Look and Feel).

In UML werden bei der Implementation die sogenannten *Component Diagrams* eingesetzt. Diese sehen Sie in unserem Modell in der "**Component View**".

Die logischen Pakete, die Pakete die wir im Design, bei der Spezifikation der Architektur definiert haben, werden mehr oder weniger 1:1 auf Komponenten Pakete abgebildet.

Basis für die Komponenten sind

- *Klassen Spezifikationen*
  diese stammen aus dem Design, möglichst detailliert
- *Klassen Diagramm*
  dieses stammt ebenfalls aus dem Design. Es zeigt die grundlegenden Beziehungen der Klassen (sematisch) auf. Unter Umständen haben bestimmte semantische Konstrukte bei der Implementierung kaum mehr Bedeutung; aber sie sind wichtig für das Verständnis.
- *Zustandsdiagramme*
  Diese zeigen die oft eher grobe Dynamik der Klassen. Es lohnt sich in der Regel nicht die Ablöufe bis in jedes Detail zu modellieren. Das lässt sich auf der Code Ebene leichter ausdrücken.
- *Dynamische Diagramme (Sequenz, Kollaboration)*
  Diese zeigen die Abfolgen der Methodenaufrufe, sind also für die Implementation sehr wesentlich!
- *Use Case(Anwendungsfall) Diagramme und Spezifikationen*
  Diese zeigen eine Gesamtsicht des Systems aus Anwendersicht, können also zur Orientierung vom Entwickler heran gezogen werden.

Wichtig ist folgendes:

- Programmcode ist immer die exakte Dokumentation eines Software Systems; alles andere muss eine Verdichtung sein, also ungenauer, mit weniger Details; dafür vielleicht verständlicher!

Da bei der Programmierung oft Designänderungen vorgenommen werden, wäre ein Roundtrip Engineering (Reverse Engineering, Forward Engineering kombiniert) sehr praktisch. Das wird auch angestrebt. Rose kann offiziell Java, C++, ... Code reverse Engineering. Aber in der Praxis müssen schon einige spezielle Rahmenbedingungen erfüllt sein, damit dies auch klappt.

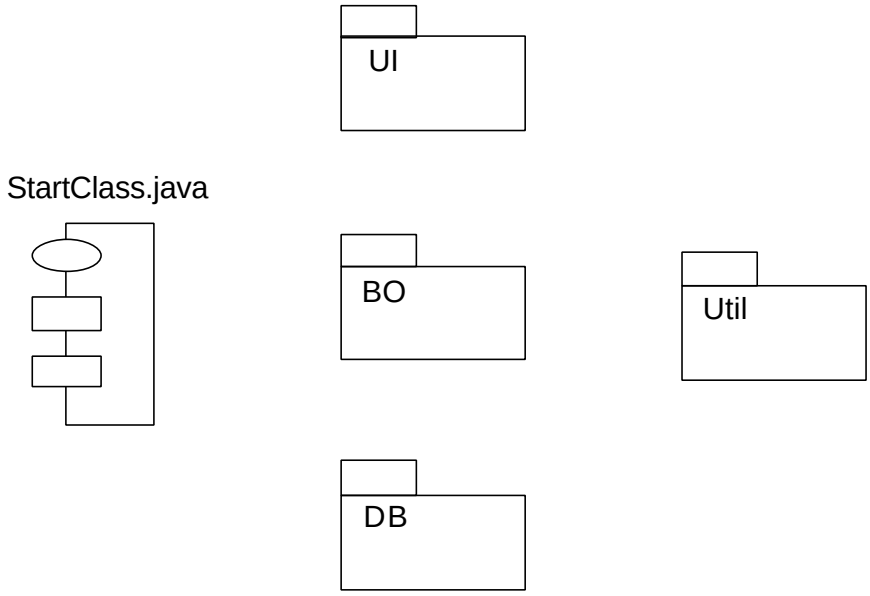
Aufgabe:

- generieren Sie für zwei zusammenhängende Klassen den Java Code (es werden zwei Dateien generiert)
- versuchen Sie daraus wieder ein Diagramm zu generieren. Das Resultat müsste dann ja gleich aussehen, wie das Ausgangsdiagramm!

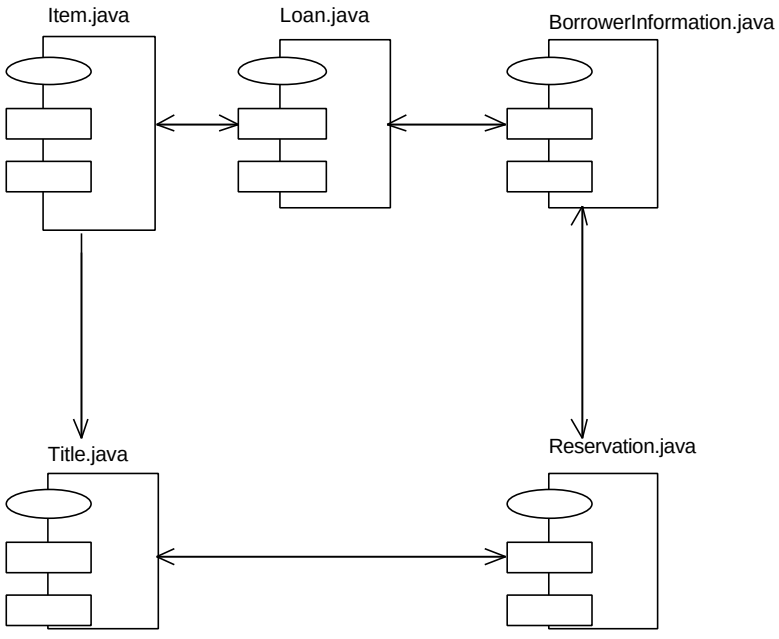
Auf den Source Code der Applikation gehe ich nicht näher ein. Er lässt sich sicher verbessern!

Die einzelnen Komponenten werden im Komponenten Diagramm dargestellt. Dieses leitet sich aus dem Klassendiagramm her:

Als Gesamtübersicht, mit der speziellen Rolle der Starter Klasse:



oder detaillierter am Falle des BO Packages (Business Objects), welches wir durch Doppelklick auf das entsprechende Packages im obigen Diagramm sichtbar machen können:



Jeder Komponente ist ihr Java Source Code als File angehängt. Durch Klicken auf das Icon wird (JBuilder) die Java Entwicklungsumgebung gestartet.

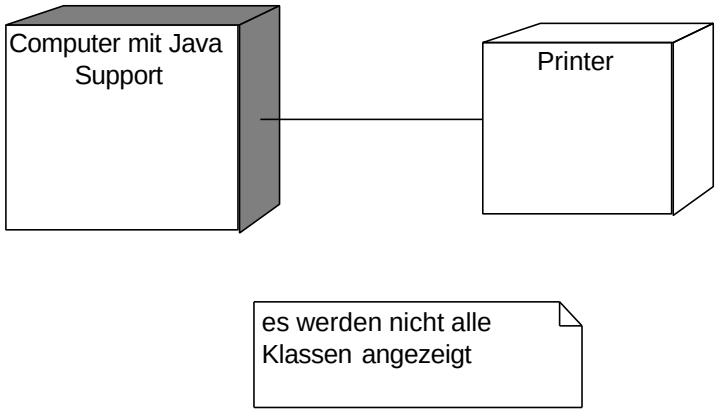
18.6. Test und Verteilung

Das Testen der Applikation geschieht an Hand der Anwendungsfälle, der Use Cases.

Die Applikation kann entweder mit Hilfe der White Box (Code Inspektion) oder Black Box (Funktionen testen) geprüft werden; für beide Methoden liefern die Use Cases die Basis.

Die Verteilung der Applikation ist in unserem Falle eher nicht möglich, entfällt also!

Das Distributionsdiagramm (mit den Knoten) ist entsprechend einfach bis sinnlos:



18.7. Ausblick

Damit sind wir am Ende des Anwendungsbeispiele angelangt. Sie finden in allen modernen Informatik Fach-Zeitschriften laufend Artikel über die UML Notation, Komponenten oder die Rational Unified Method.

Beispiele sind das Objekt Forum (eine "Kiosk" Fachzeitschrift : erhältlich am Kiosk), oder die IEEE Zeitschriften (Software, Internet, Computer.... : alle sind in englisch!).