Standard Design Pattern

A Standard Design Pattern

A.1 Übersicht

A.1.1 Typische technologische Beschränkungen

Wie bereits beschrieben, ist beim momentanen Stand der Netzwerktechnologien ein naives Modellieren einer Anwendung ohne Berücksichtigung technologischer Aspekte kaum sinnvoll möglich. Fordert die fachliche Vorgabe beispielsweise das Iterieren über eine Ergebnismenge mit Tausenden von Treffern, so muss sich die Ergebnismenge lokal auf dem gleichen physikalischen Rechner befinden wie der Iterator: RMI Methodenaufrufe sind im Vergleich zu lokalen Aufrufen drastisch[1] langsamer, so dass es einfach zu berechnen ist, wie groß die Ergebnismenge sein muss, bis „die Anwendung steht“.

Eine ähnliche Argumentation gilt für den ausfallsicheren Clusterbetrieb: Eine fachliche Vorgabe, die das Halten von Megabyte-Großen Objektbäumen auf Serverseite fordert (so genannte „Sessions“), kann im Cluster nicht einfach so funktionieren: Die Ausfallsicherheit fordert ja, dass Session-Informationen zwischen den Rechnern des Clusters repliziert wird, und dafür sind aus technischen Gründen recht rigide Größenbeschränkungen (etwa 50 KByte) gegeben.

Einer der wichtigsten Einflüsse ist jedoch die Call by Value-Semantik bei entfernten Methodenaufrufen. Diese Semantik ist ja, wie bereits erwähnt, dadurch begründet, dass das Datenmodell auf Grund beschränkter Netzwerke nicht in der Lage ist, alle registrierten Views aktiv über Zustandsänderungen zu benachrichtigen.

Hinweis: Eine weitere Begründung sind Transaktionsgrenzen: Ein Client, der mehrere Schreibvorgänge hintereinander innerhalb einer Transaktion gruppieren möchte, muss gegebenenfalls den Datensatz exklusiv sperren. Dies kann nicht Aufgabe des Clients sein, da das Risiko von lang andauernden Transaktionen (Gesperrte Datensätze müssen ja wieder durch eine Client-initiierte Aktion aktiv entsperrt werden) nicht ausgeschlossen werden kann.

A.1.2 Einfluss auf das Design der Anwendung

Es stellt sich nun natürlich die Frage, ob und in welchem Maße diese technologischen Einflüsse auf Auswirkungen auf das Design der An-

[1] Bei guten Netzwerken liegt ein RMI-Aufruf im unteren Millisekundenbereich, lokale Aufrufe sind mehrere Größenordnungen schneller.

wendung haben werden. Die Antwort darauf ist nicht eindeutig, es kommt natürlich stark auf die Ebene der Abstraktion an. Im Allgemeinen sind jedoch für eine realitätsnahe Abbildung, die effizient in eine Anwendung umgesetzt werden kann, zumindest ein Satz Beschränkungen zu beachten. Diese können durch einen Katalog von Design Patterns beschrieben werden, weil die momentan verwendeten Plattformen zur Abbildung von e-Business-Anwendungen im Wesentlichen auf den gleichen Technologien basieren und demzufolge auch den gleichen Nebenbedingungen genügen.

A.1.3 Übersicht

Problemstellungen während des Software-Entwicklungsprozesses zeigen Regelmäßigkeiten. Damit sind auch ähnliche Lösungsansätze möglich. Diese können sprach- und plattformunabhängig formuliert werden

Hinweis: Ist dies nicht möglich, spricht man von sprach-abhängigen „Idiomen“

- Ein Design Pattern besteht aus
- Einer allgemeine Problemstellung
- Einem allgemeiner Lösungsansatz
- Allgemeinen Konsequenzen bei der Verwendung
- Menge von Umsetzungen auf verschiedene Plattformen mit verschiedenen Sprachen, den „Mappings“

Design Patterns verbessern die Wartbarkeit und Wiederverwendung erstellter Komponenten und erhöhen damit die Effizienz der Programmierung durch Einsatz bewährter Strategien. Weiterhin ermöglichen sie eine Klassenmodellierung auf einer höheren Ebene. Dadurch wird die Konsistenz der erstellten Anwendungen garantiert.

Es gibt eine ganze Reihe von Gruppen von Design-Patterns-Katalogen. Diese unterscheiden sich voneinander im Wesentlichen durch die Allgemeinheit ihrer Problemstellung:

- Die 23 GoF-Pattern („Gang of Four“ mit ihrem „Lead-Designer“ Erich Gamma) sind Sprach- und Anwendungsunabhängig. Sie sind ihrerseits unterteilt in Erzeugungsmuster, Strukturmuster und Verhaltensmuster
- Analyse-Pattern enthalten einen Satz gängiger fachlicher Objekte (Kunde, Rechnung…) und ihre gegenseitigen Abhängigkeiten.
- Pattern für verteilte Anwendungen sind ebenfalls Sprach- und Anwendungsunabhängige Pattern, die häufig bei e-Business Anwendungen anzutreffen sind[2]. Damit sind in ihrer Problemstellung die oben erwähnten technologischen Einflüsse erkennbar. Diese Patterns sind zwar erst in der Definitionsphase, es gibt aber einen de Facto Katalog von Martin Fowler
- Technologie-Pattern, z.B. „J2EE-Patterns“ zur Erstellung einer Anwendung im Rahmen des J2EE-Programmierungsmodells
- Anwendungs-Pattern. Jede Anwendung entwickelt in der Regel weitere spezielle Patterns und Idiome, die jedoch nur schwer allgemein formuliert werden können.

In den folgenden Abschnitten wird näher auf die GoF- und die e-Business-Pattern eingegangen. Die J2EE Pattern werden nach der Einführung der J2EE Plattform behandelt. Analyse-Pattern und die Anwendungs-Pattern sind nur im Kontext einer konkreten großen Anwendungsentwicklung relevant und werden deshalb in diesem Zusammenhang nicht detailliert erörtert.

[2] Damit wird dieser Pattern-Katalog auch mit dem Synonym „e-Business Pattern“ belegt.

A.2 Die GoF-Pattern

Auch im Rahmen von e-Business-Anwendungen können die klassischen Patterns erkannt und deren Lösung nutzbringend angewendet werden. Die Kombination der Patterns wird eine drastische Verbesserung der Architektur bringen.

Hinweis: „Pattern-orientiertes Programmieren“ wird als neue Stufe bzw. als Ersatz zum Objektorientierten Ansatz aufgefasst.

Die konkrete Implementierung innerhalb einer e-Business-Plattform erfordert jedoch teilweise große Änderungen im Vergleich zu einer lokalen Anwendung. So steht bei Verteilten Anwendungen kein gemeinsamer Rechnerspeicher zur Verfügung.

In den folgenden Abschnitten wird jeweils die typische Problemstellung, das Klassendiagramm und Hinweise für die Verwendung des Patterns in verteilten Anwendungen gegeben.

A.2.1 Erzeugungsmuster

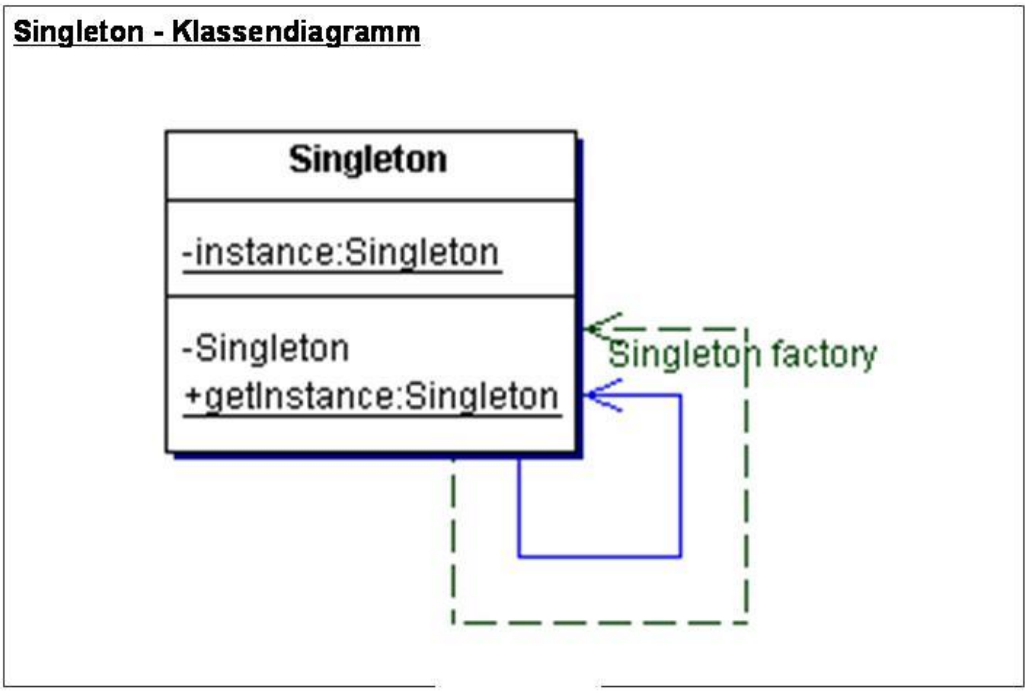
Erzeugungsmuster kapseln die Instanzierung von Objekten und Erweitern somit das Konstruktor-Konzept. Die folgenden Muster sind geläufig:

- Singleton
- Factory
- Builder
- Prototype

A.2.2 Singleton

„Sichere ab, dass eine Klasse genau ein Exemplar besitzt, und stelle einen globalen Zugriffspunkt darauf bereit.“

Die Implementierung eines Singletons in einer Objektorientierten Sprache erfolgt meist durch ein statisches Attribut, eine statische Zugriffsmethode darauf und einen privaten Konstruktor:



Applikationsdienste sind typische Beispiele für die Verwendung eines Singletons

- Authentifizierung
- Zentrale Datenhaltung

Das besondere Verhalten in Mehrschichten-Architekturen führt jedoch dazu, dass die klassische Singleton-Implementierung häufig problematisch ist: Gilt „eine Instanz“ pro Anwendung, pro Schicht oder insgesamt? Die klassische Singleton-Implementierung in Java garantiert ausschließlich eine Instanz pro Klassenlader pro Virtueller Maschine und ist damit spätestens im Cluster-Betrieb mit mehreren Rechnern und Prozessoren sicherlich kein echtes Singleton mehr.

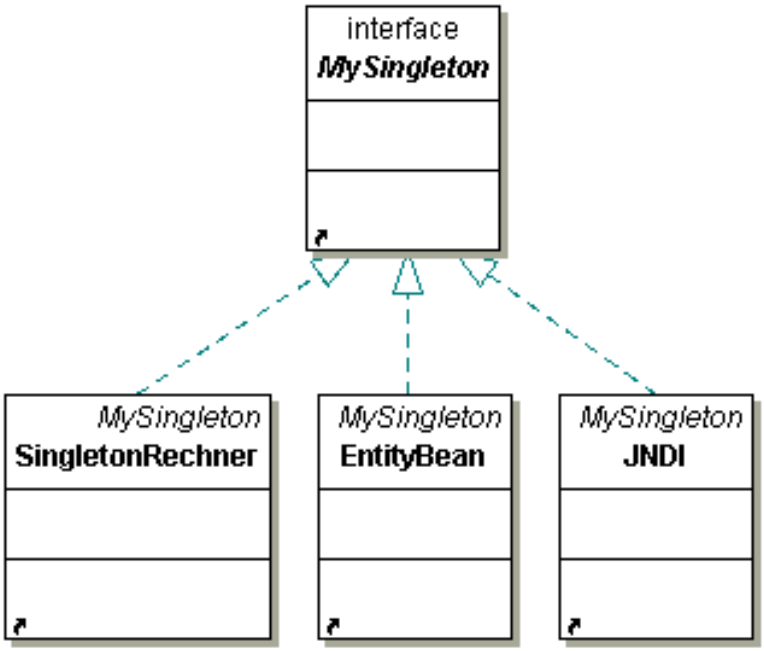
Das Singleton in verteilten Anwendungen ist somit fehleranfällig und sollte durch alternative Ansätze (z.B. eine zentrale Registry) ersetzt werden. Klassische Singletons dürfen in J2EE-Anwendungen niemals als Zwischenspeicher für Informationen missbraucht werden; sie sind

aus der Sicht des Aufrufers somit zustandslos. Ein weiteres potenzielles Problem tritt bei Multithreaded-Anwendungen auf.

Lösungsansätze:

- Datenhaltung des Singletons erfolgt in einer Datenbank (Zugriff mit JDBC oder EntityBeans). Datenhaltung erfolgt mit Hilfe eines JNDI-Kontextes (JNDI-Lookups). Dies ist sinnvoll für Dienste wie Print-Queues etc.
- Datenhaltung wird auf einen Cluster-Rechner („Singleton-Rechner“) ausgelagert. (Netzwerkprotokoll, z.B. RMI, selbstdefinierte TCP/IP-Protokolle). Falls 100% Ausfall-Sicherheit notwendig ist, ist dieser Ansatz nicht sinnvoll.

Das folgende Klassendiagramm zeigt, wie ein Singleton als Schnittstelle definiert und in unterschiedlicher Form implementiert werden kann:



Ein Singleton in einer verteilten Anwendung kann somit durchaus aus mehreren Objekten bestehen! Wichtig ist nur, dass die Gesamtheit der Objekte als Singleton auftritt.

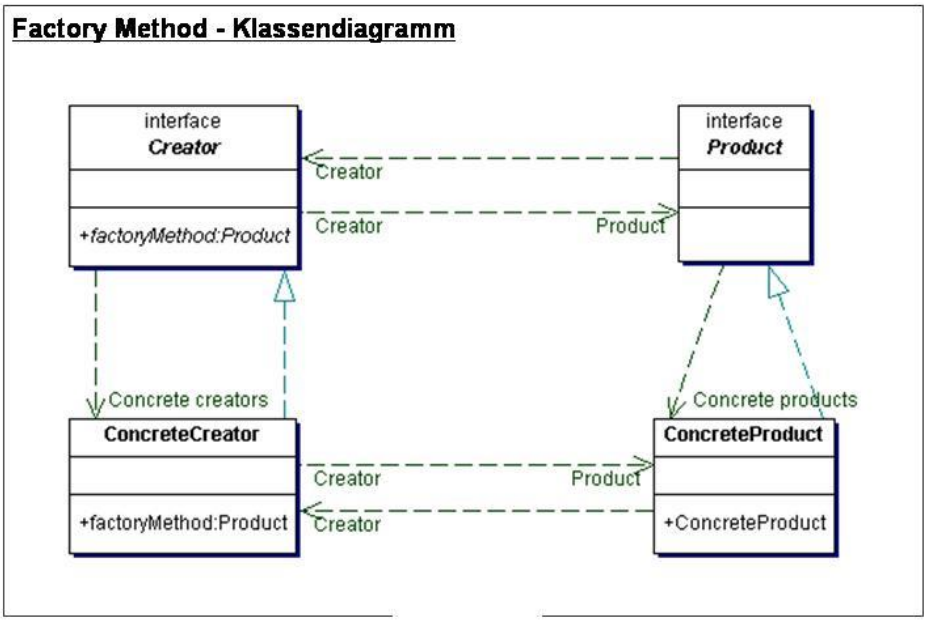
A.2.3 Factory

„Definiere eine Klassenschnittstelle mit Operationen zum Erzeugen eines Objekts, aber lasse Unterklassen entscheiden, von welcher Klasse das zu erzeugende Objekt ist. Fabrikmethoden ermöglichen es einer Klasse, die Erzeugung von Objekten an Unterklassen zu delegieren.“

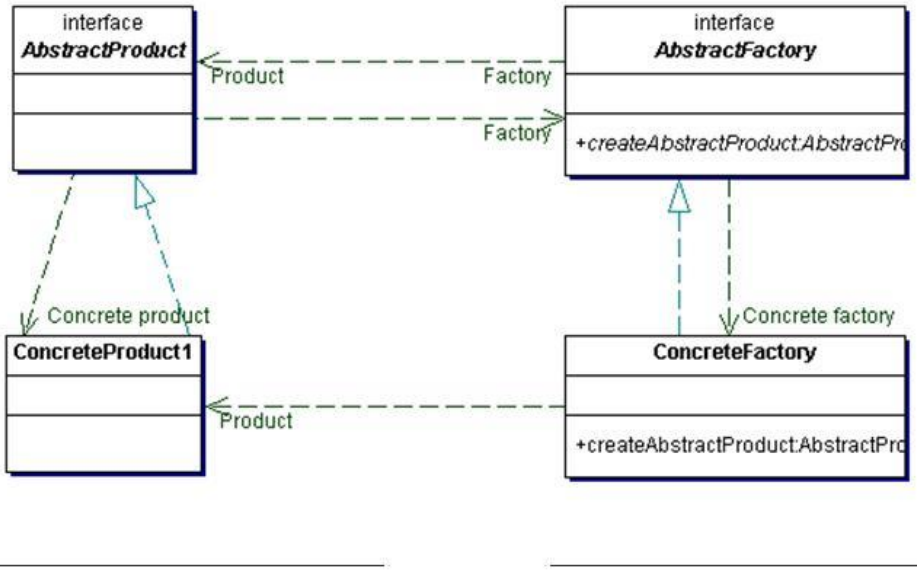
Von der Factory gibt es unterschiedliche Ausprägungen:

- Abstract Factory
  - Die Objekterzeugung wird von einer Subklasse übernommen
- Factory Method
  - Überschreiben/Implementieren einer Objekterzeugenden Methode
- Builder
  - Eine Factory delegiert die eigentliche Objekterzeugung nochmals an eine weitere Hilfsklasse weiter

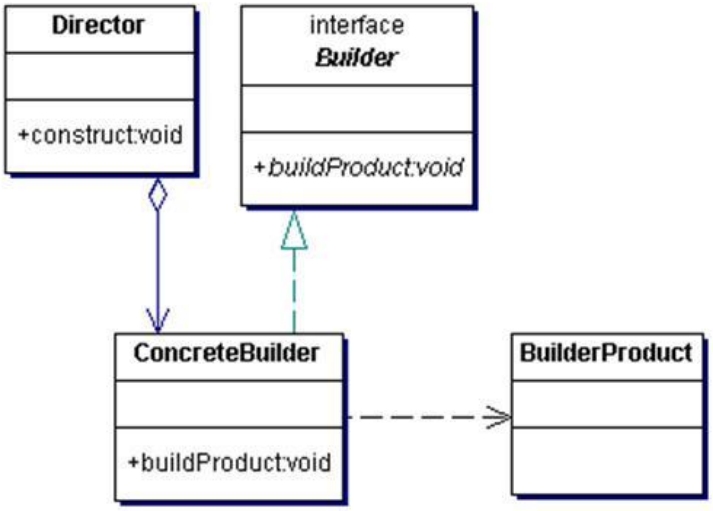
Durch den Java-Reflection-Mechanismus können sehrt schön generische konfigurierbare Factory-Implementierungen verwendet werden, so dass der Unterschied der ersten beiden Patterns verschwindet.



Abstract Factory - Klassendiagramm



Builder - Klassendiagramm



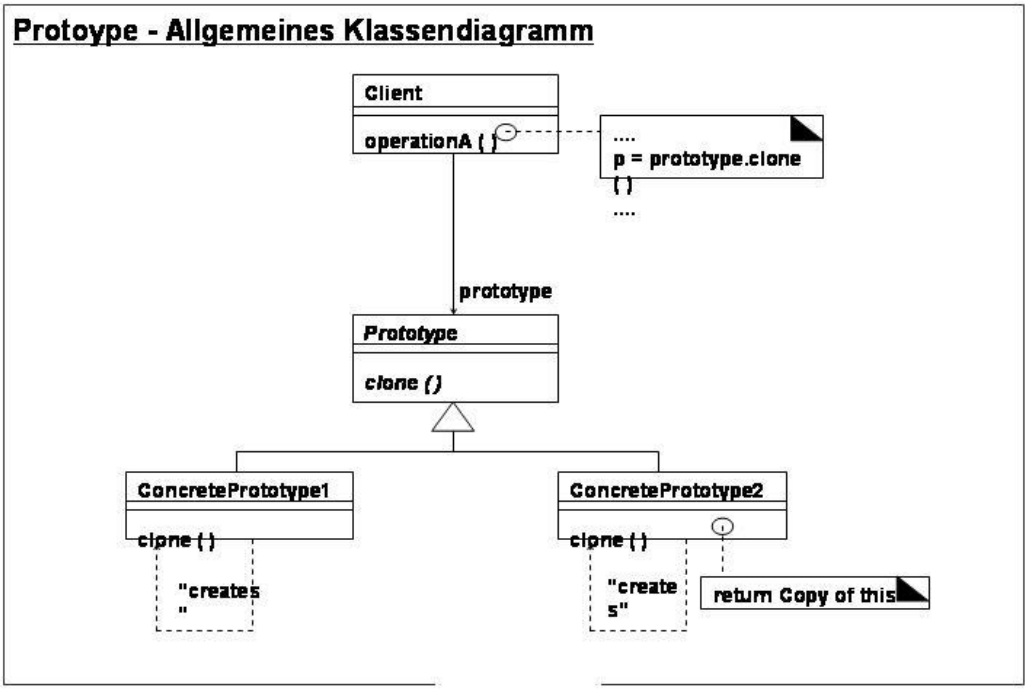
Factories sind die zentralen Stellen zur Entkopplung von Abhängigkeiten und sind somit in e-Business-Anwendungen weit verbreitet:

- Kapselung der Implementierungsdetails für Geschäftsobjekte
- Änderungen und Erweiterungen der Geschäftsobjekte können durch Subklassen durchgeführt werden
- Transparenz und Austausch der verwendeten Schichten
- Austausch von Kommunikationsprotokollen
- Austausch von verwendeten Backend-Technologien
- Factories sind zentraler Bestandteil komplexerer e-Business-Patterns wie Plug In oder Service

Die Implementierung einer Factory kann häufig risikolos als klassisches Singleton erfolgen, da sie in der Regel zustandslos konzipiert werden kann.

A.2.4 Prototype

„Bestimme die Arten zu erzeugender Objekte durch die Verwendung eines prototypischen Exemplars und erzeuge neue Objekte durch Kopieren dieses Prototypen.“



Prototyping ist durch die „Call by Value“-Semantik implizit Bestandteil einer verteilten Anwendung. Eine Datenhaltende Klasse kann als Prototyp aufgefasst werden, der beim Zugriff des Clients die Rückgabewerte liefert.

Bei zentralen Prototyping-Strategien ist im Cluster-Betrieb auf die Synchronisierung der Prototyp-Instanzen zu achten, also das typische Problem der Cache-Synchronisation.

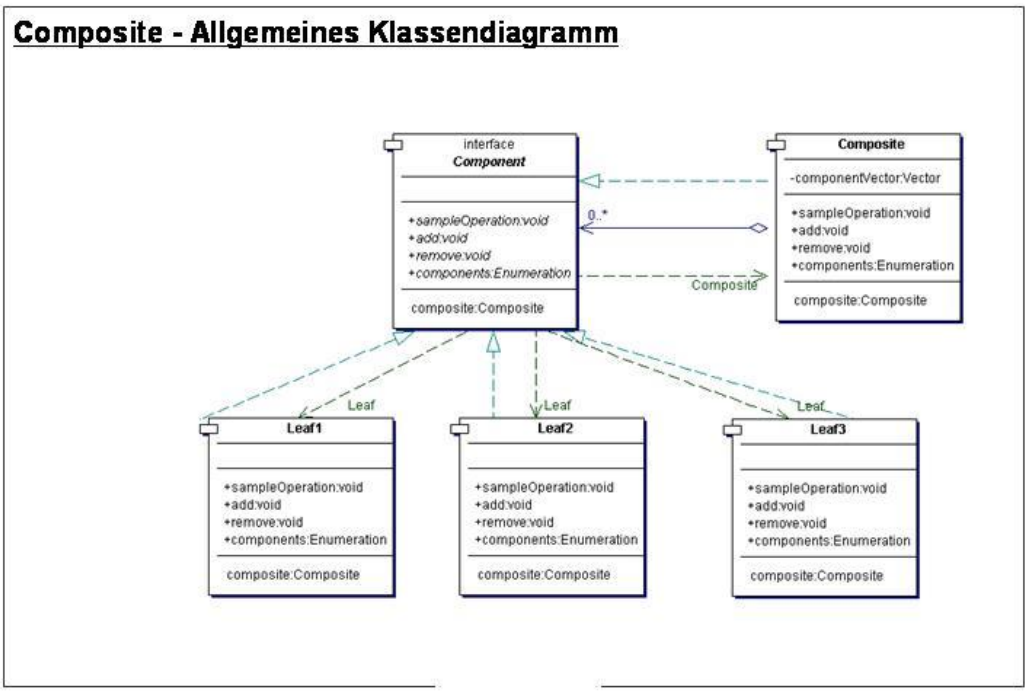
A.2.5 Strukturmuster

Durch die Strukturmuster wird die Abhängigkeit von Objekten untereinander beschrieben. Die folgenden Patterns sind vorhanden:

- Composite
- Adapter
- BridgeDecorator
- Facade
- Flyweight
- Proxy

A.2.6 Composite

„Füge Objekte zu Baumstrukturen zusammen, um Teil-Ganzes-Hierarchien zu repräsentieren. Das Kompositionsmuster ermöglicht es Klienten, sowohl einzelne Objekte als auch Kompositionen von Objekten einheitlich zu behandeln.“



Die komplexen Datenstrukturen einer e-Business-Anwendung werden häufig als Composite definiert:

- XML
- Objekt-Aggregate, die Daten enthalten
- Dokumente
- Verschachtelte Container zum Aufbau von Oberflächen

A.2.7 Hüllklassen

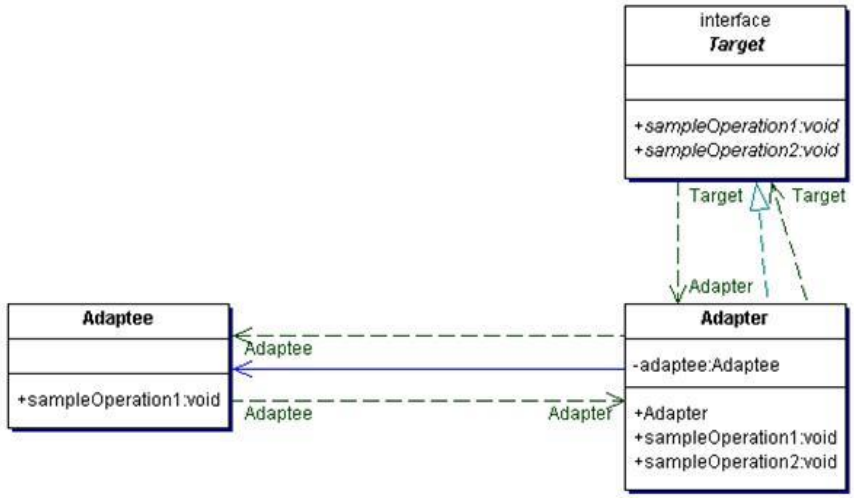
„Anpassen von Signaturen und Trennung der Abstraktion von der konkreten Implementierung“

Die konsequente Verwendung von Hüllklassen hat als Ergebnis modulare und damit sehr wieder verwendbare Anwendungen. Statt alle Funktionalität in einige wenige Klassen zu zwängen, die damit nur auf eine ganz bestimmte Anwendungssituation passen, können die einzelnen Hüllen oft überraschen flexibel kombiniert werden.

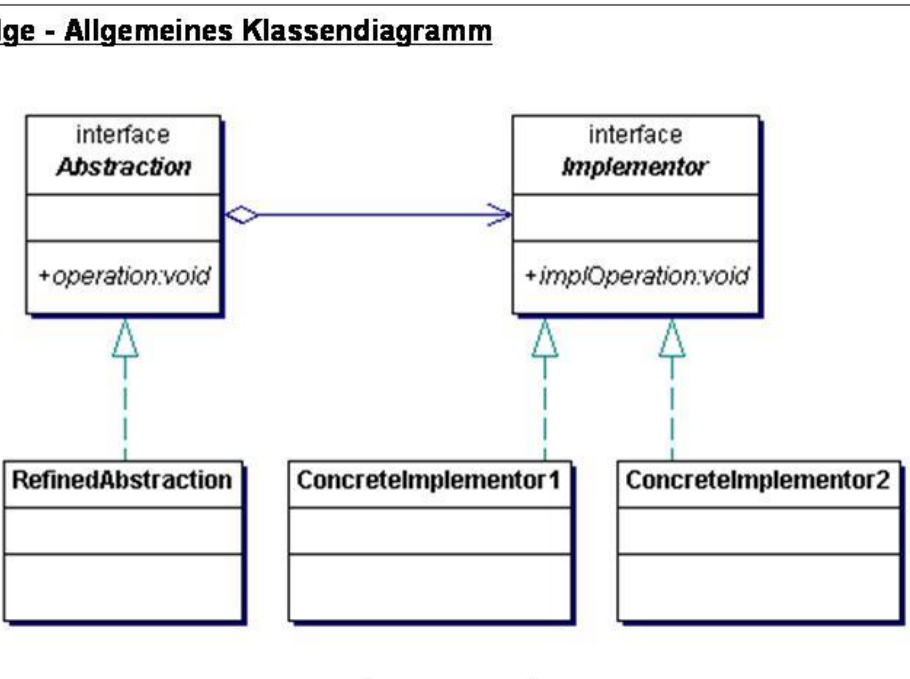
Ausprägungen sind:

- Adapter
- Bridge
- Decorator
- Proxy

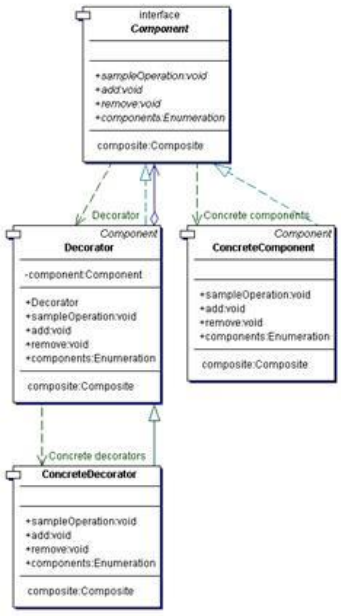
Adapter - Allgemeines Klassendiagramm



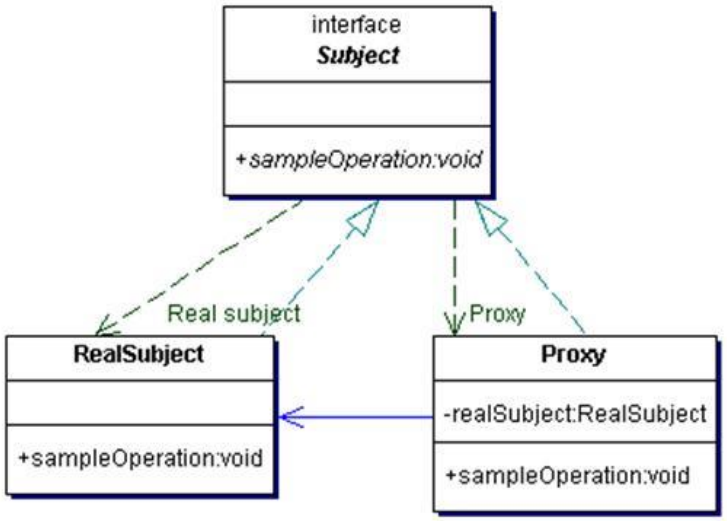
Bridge - Allgemeines Klassendiagramm



Decorator - Allgemeines Klassendiagramm



Proxy - Allgemeines Klassendiagramm

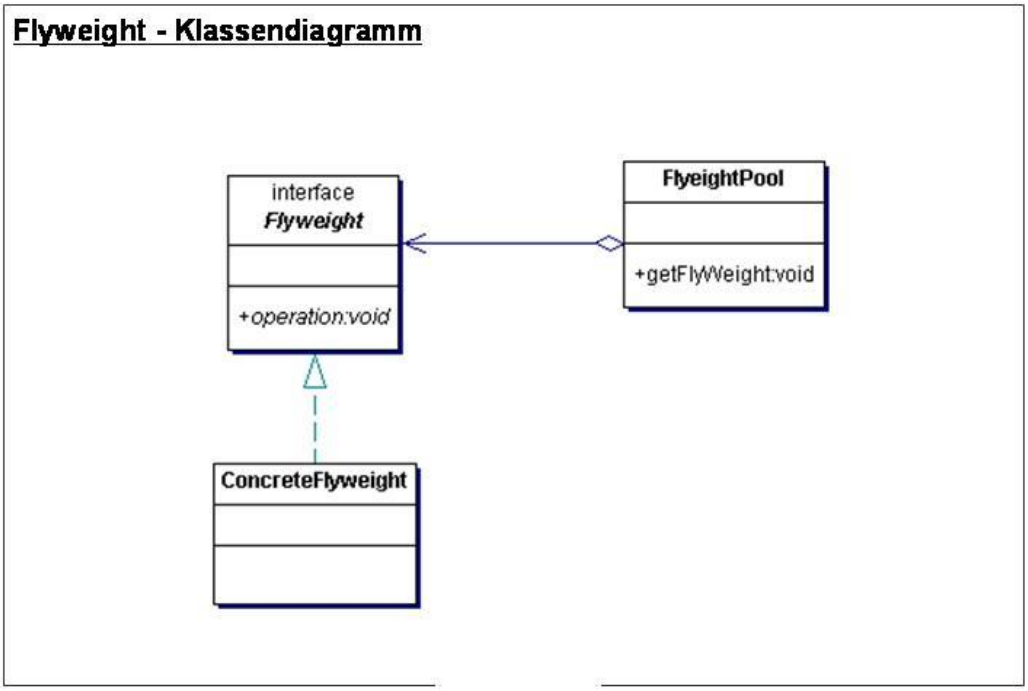


In verteilten Anwendungen kommen insbesondere Adapter und Proxy häufig zum Einsatz:

- Integration von Subsystemen
- Altanwendungen (Legacy)
- Anpassen von Signaturen direkt nicht-kompatibler Systeme
- Erweiterung fachlicher Abläufe durch technologische Nebenbedingungen
- Proxies als Stubs für verteilte Anwendungen
- Dynamische Zuschalten weiterer fachlicher Anforderungen
- Autorisierung, Session Management

A.2.8 Flyweight

„Nutze Objekte kleinster Granularität gemeinsam, um große Mengen von ihnen effizient verwenden zu können."

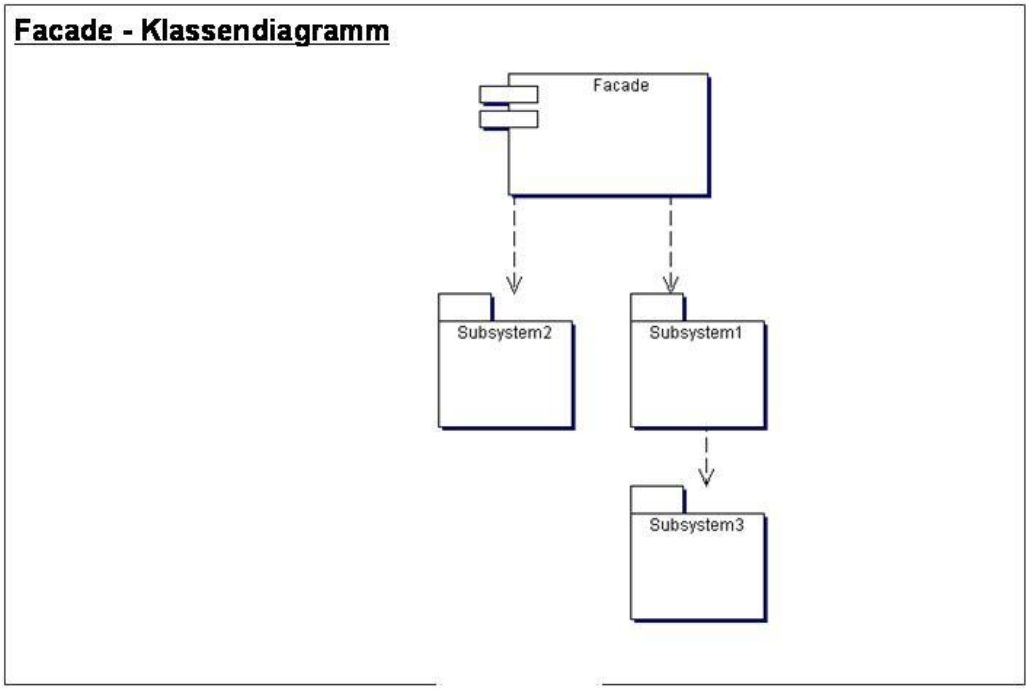


Mit Hilfe des Flyweight Patterns können effizient Caches und Pools aufgebaut werden. Damit ist es ein Mittel zur transparenten Optimierung von Ausführungsgeschwindigkeit und Speicherbedarf. Flyweight dient als Grundlage für die Implementierung eines Containers für e-Business-Komponenten. Zu beachten ist jedoch, dass die vom Fly-

weight verwalteten Objekte zumeist unveränderbar (immutable) sein müssen.

A.2.9 Facade

„Schaffe einen einfachen Einstieg in ein komplexes Subsystem.“



Die Client-Sicht auf eine e-Business-Anwendung ist stets eine Fassade. Zwischen den (fachlichen oder technologischen) Schichten der Anwendung werden ebenfalls Fassaden eingesetzt, um die Abhängigkeiten zwischen Subsystemen zu minimieren.

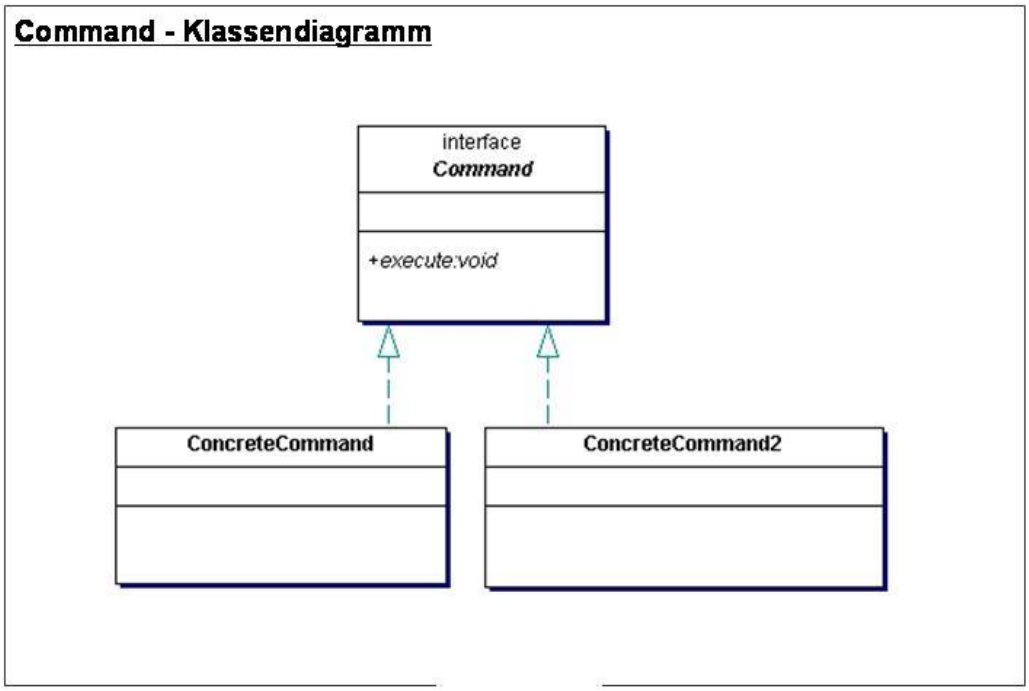
A.2.10 Verhaltensmuster

Erzeugungs- und Strukturmuster decken den eher statischen Teil einer Anwendung ab und sind deshalb auch häufig alleine durch Klassendiagramme zu beschreiben. Verhaltensmuster definieren den dynamischen Ablauf der Anwendung; zur Beschreibung sind deshalb häufig Sequenzdiagramme hilfreich.

- Command
- Chain of Command
- Template Method
- Mediator
- Observer
- State
- Strategy
- Visitor
- Iterator
- Memento
- Interpreter

A.2.11 Command

„Kapsle einen Befehl als Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Queue zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen."

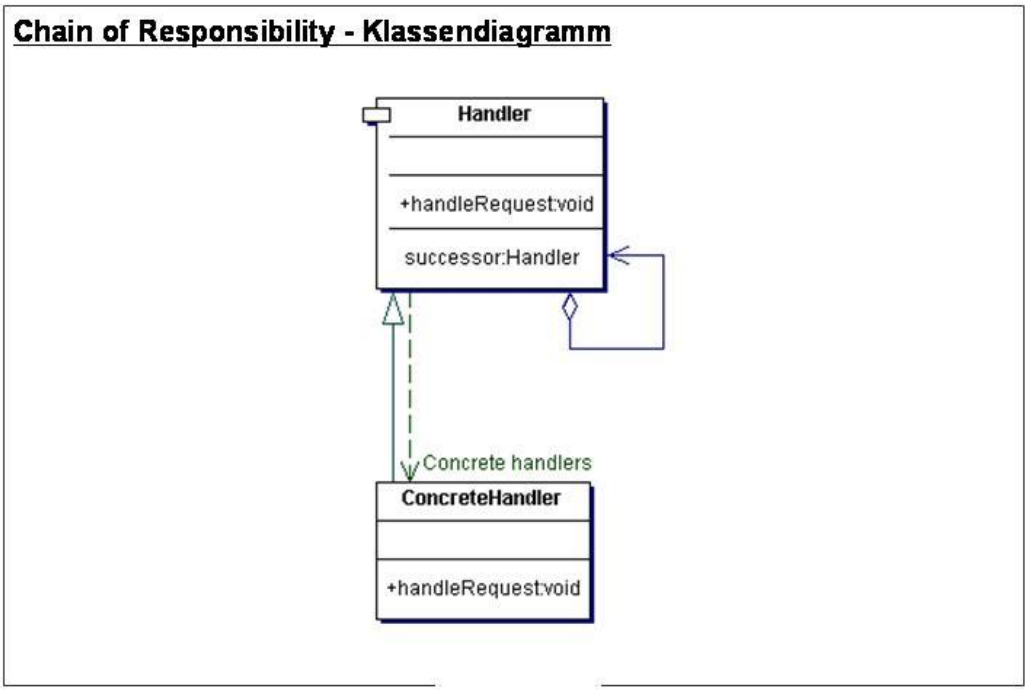


Controller des MVC-Patterns verwenden zur Interpretation der Aktionskommandos Command-Objekte. Der Controller erkennt das Aktionskommando und holt sich von einer Factory die zugehörige Command-Implementierung.

In einer Message Oriented Middleware (MOM) dienen Messages zur Definition asynchroner ausgeführter Commands: Eine Queue empfängt zentral Messages und verteilt sie an einzelne Empfänger. Diese Interpretieren dann den Inhalt der Message und führen diese dann aus.

A.2.12 Chain of Responsibility

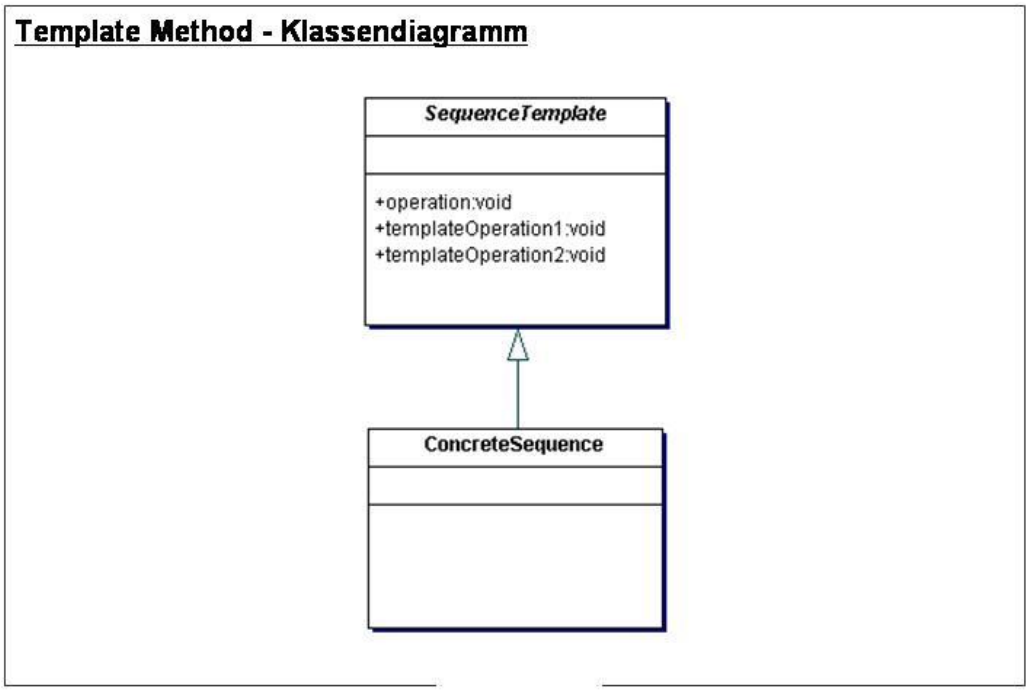
„Vermeide die Kopplung des Auslösers einer Anfrage mit seinem Empfänger, indem mehr als ein Objekt die Möglichkeit erhält, die Aufgabe zu erledigen. Verkette die empfangenden Objekte und leite die Anfrage an der Kette entlang, bis ein Objekt sie erledigt.“



Komplexere Abläufe, wie sie bei e-Business-Anwendungen die Regel sind, können durch verkettete Controller modular realisiert werden („Hierarchisches MVC“). Durch Registrieren/Abmelden von Sub-Controllern kann eine dynamische Ablaufsteuerung realisiert werden.

A.2.13 Template Method

„Definiere das Skelett eines Algorithmus in einer Operation und delegiere einzelne Schritte an Unterklassen. Die Verwendung einer Schablonenmethode ermöglicht es Unterklassen, bestimmte Schritte eines Algorithmus zu überschreiben, ohne seine Struktur zu verändern.“

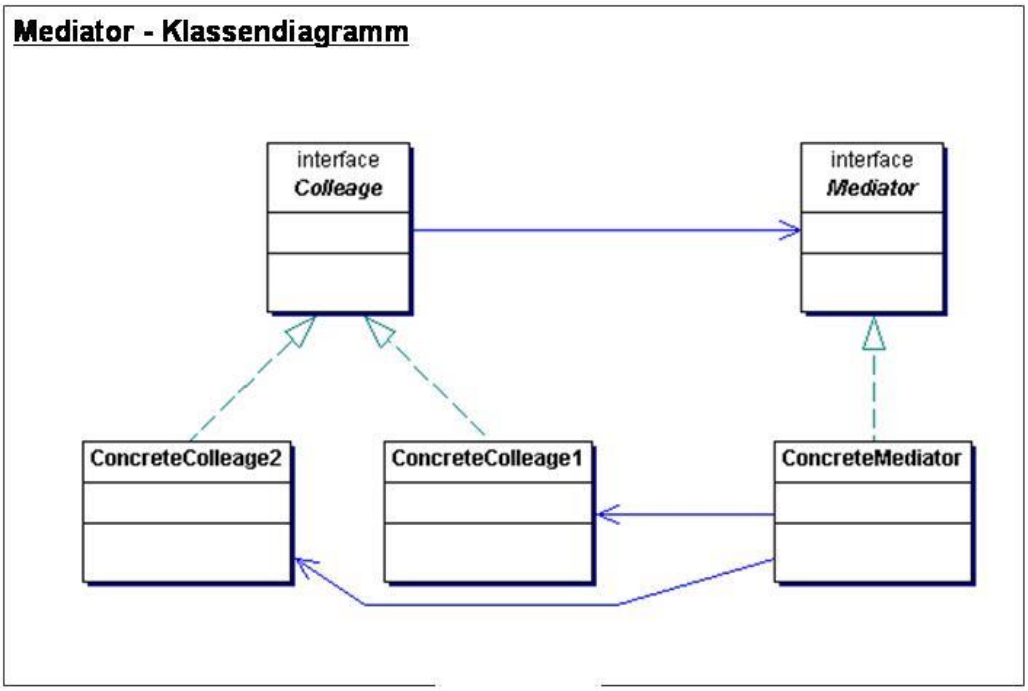


Die Architektur einer Anwendung kann durch Method Templates auf einer hohen Abstraktionsebene definiert werden. Das Abbilden von abhängigen Use Cases durch Method TemplatesCode Generatoren erzeugt Rümpfe, die den Container-Component-Contract des Komponentenmodells erfüllen, der Anwendungsprogrammierer implementiert nur noch abstrakte Business-Methoden.

So kann beispielsweise eine Authentifizierungs- Autorisierungssequenz innerhalb eines Method Templates vorgegeben werden, so dass die eigentliche Programmlogik davon unbeeinflusst programmiert werden kann.

A.2.14 Mediator

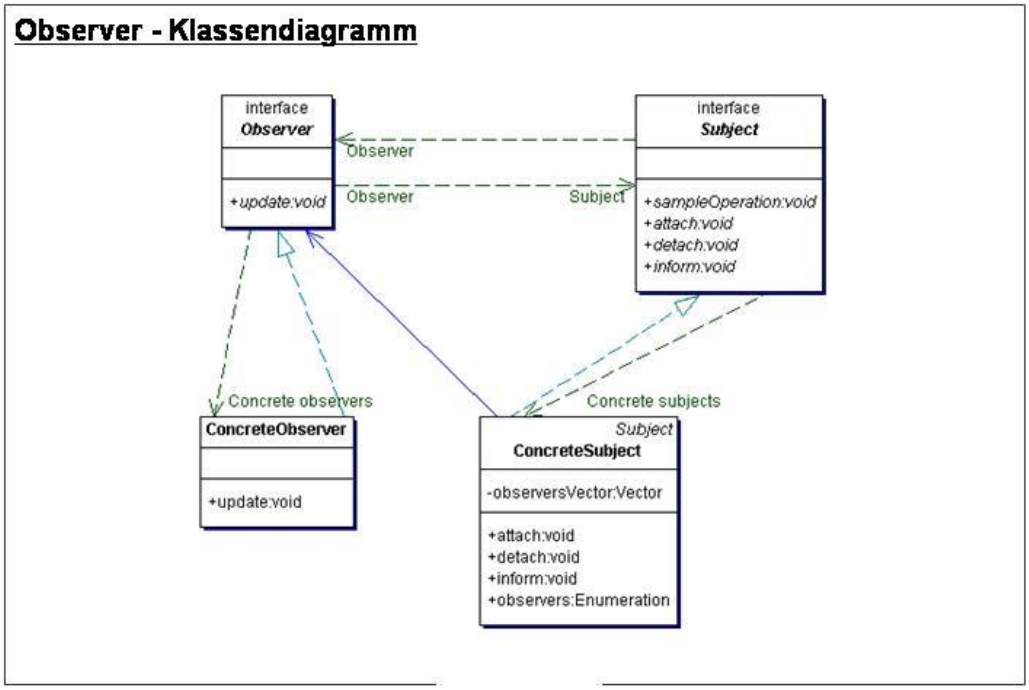
„Definiere ein Objekt, das ein komplexes Zusammenspiel verschiedener anderer Objekte regelt, ohne dass diese sich direkt kennen müssen."



Komplexe Oberflächen zur Eingabe und Darstellung von Geschäftsdaten werden durch Plausibilitätsregeln und daraus resultierende Pflichteingaben und weitere Validierungen gesteuert. Der zulässige Wertebereich eines Eingabefeldes wird beispielsweise durch eine Auswahl aus einer Liste heraus definiert.

A.2.15 Observer

„Definiere eine 1-zu-n Abhängigkeit zwischen Objekten, so dass die Änderung des Zustands eines Objekts dazu führt, dass alle abhängigen Objekte benachrichtigt und automatisch aktualisiert werden.“

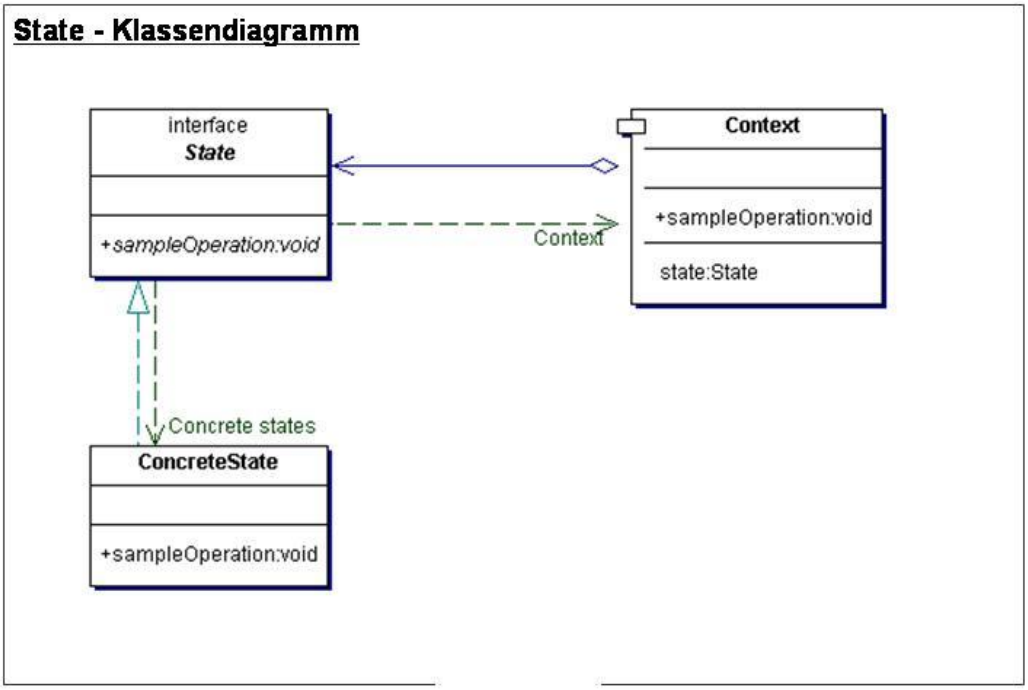


Observer dienen in e-Business-Anwendungen beispielsweise zu:

- Lose Kopplung zwischen verarbeitenden Prozessen
- Dynamisches Hinzufügen und Entfernen von Verarbeitungsschritten

A.2.16 State

„Ermögliche es einem Objekt, sein Verhalten zu ändern, wenn sein interner Zustand sich ändert. Es wird so aussehen, als ob das Objekt seine Klasse gewechselt hätte.“

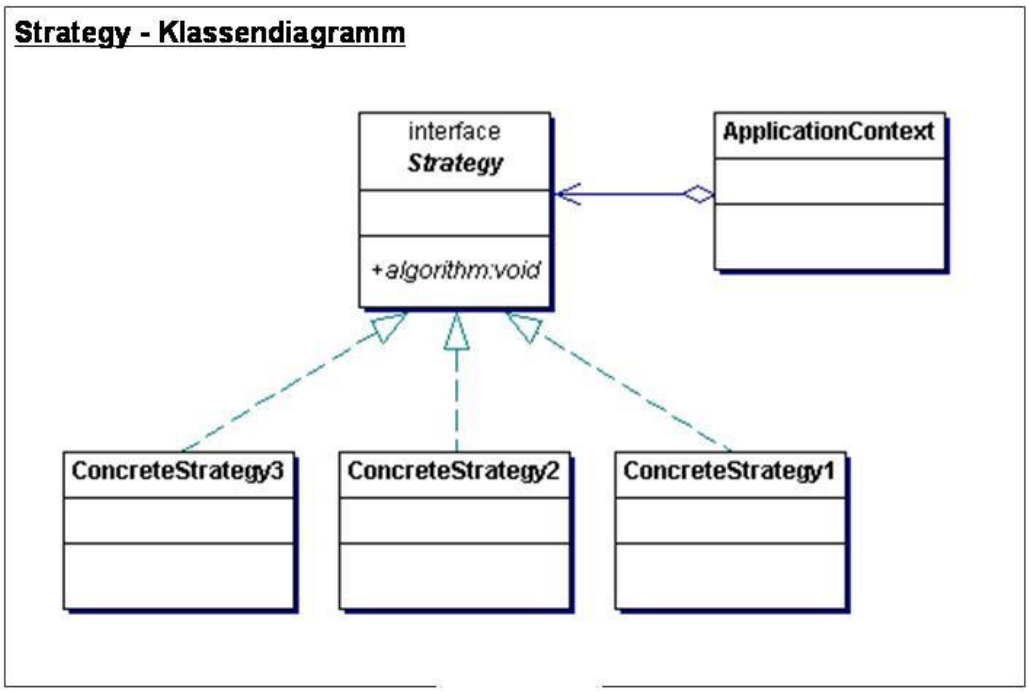


State ermöglicht die Änderung des Verhaltens eines Geschäftsprozesses in Abhängigkeit vom internen Zustand eines Objektes und damit

- eine andere Verarbeitung
- das Hinzufügen weiterer Abläufe

A.2.17 Strategy

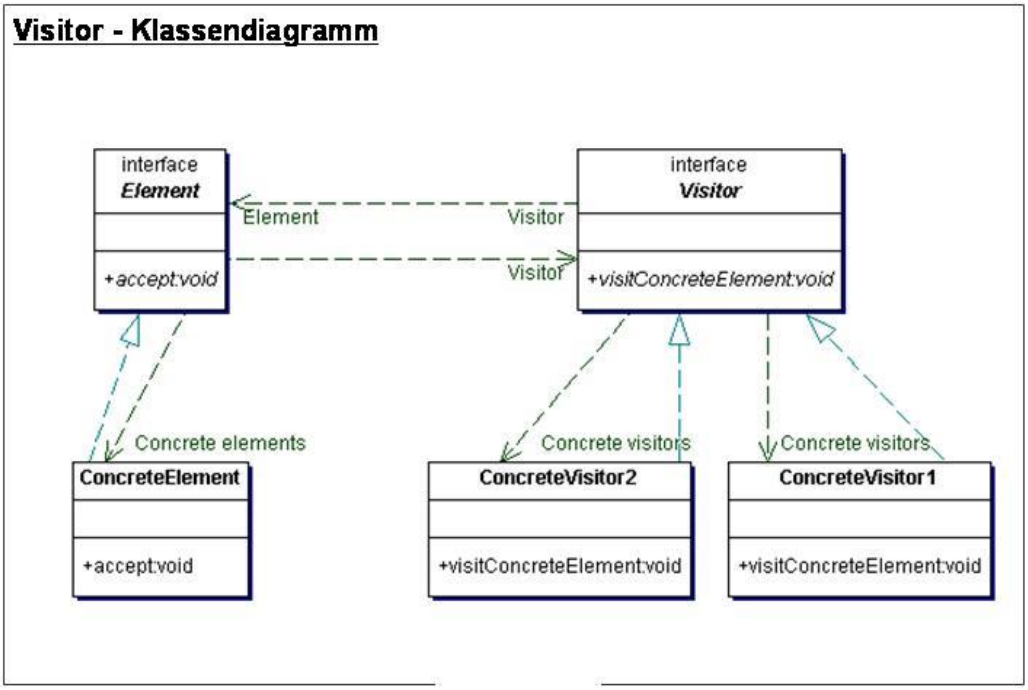
„Definiere eine Familie von Algorithmen, kapsele jeden einzelnen und mache sie austauschbar. Das Strategiemuster ermöglicht es, den Algorithmus unabhängig von ihn nutzenden Klienten zu variieren."



Strategien definieren austauschbare Business-Regeln und Verarbeitungsschritte. In letzter Konsequenz realisieren alle Patterns als Gesamtbild eine Strategy.

A.2.18 Visitor

„Definiere eine Operation, die auf einem Satz von Elementen ausgeführt werden kann. Ein Visitor kann neue Logik enthalten, ohne dass die verarbeitenden Klassen geändert werden müssen.“



Ergebnisse eines Geschäftsprozesses müssen häufig von verschiedenen nach geschalteten Prozessen verändert werden, um ein bestimmtes Dokumentenformat oder zusätzliche Inhalte zu erreichen:

- XML-Transformationen
- Aufbereitung von Datenformaten (PDF)
- Personalisierte Web-Seiten

A.2.19 Interpreter, Iterator und Memento

Diese Pattern werden in der Regel selten direkt in eigenen Anwendungen realisiert, sondern stehen zur Verfügung:

- Iteratoren als Bestandteil von Collections
- Memento als Objektserialisierung
- Interpreter als Grundlage für reguläre Ausdrücke

A.3 Patterns für verteilte Anwendungen

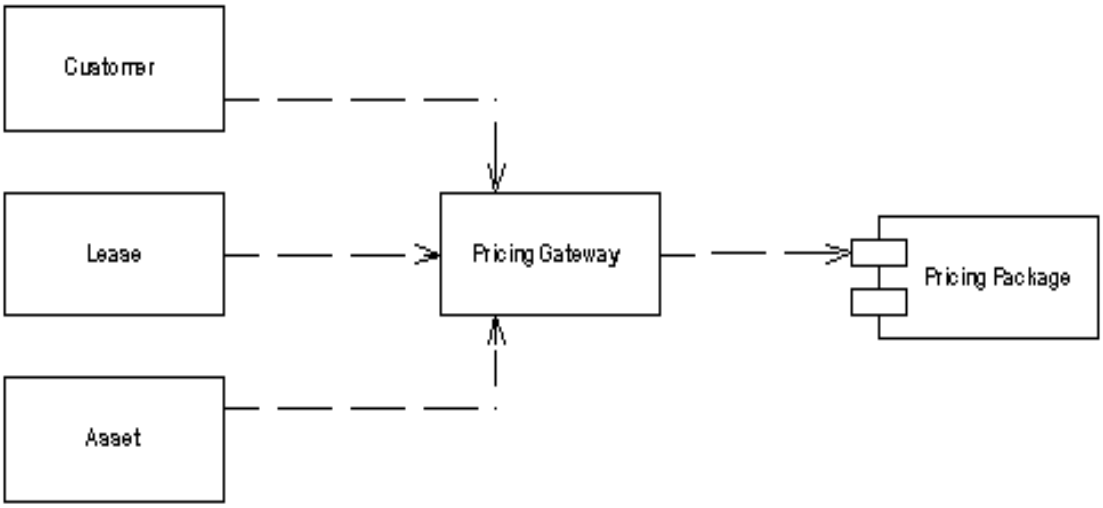
Diese Patterns sind in ihrem Problemkreis nicht mehr so allgemein, sondern betreffen typische Problemstellungen bei verteilten Anwendungen. Schwerpunkte sind hier beispielsweise die Probleme der Datenübertragung über ein Netzwerk, Definition von Transaktionen sowie Elemente des O/R-Mappings.

Eine hervorragende Übersicht dieser Patterns liefert Martin Fowler:

http://www.martinfowler.com/eaaCatalog

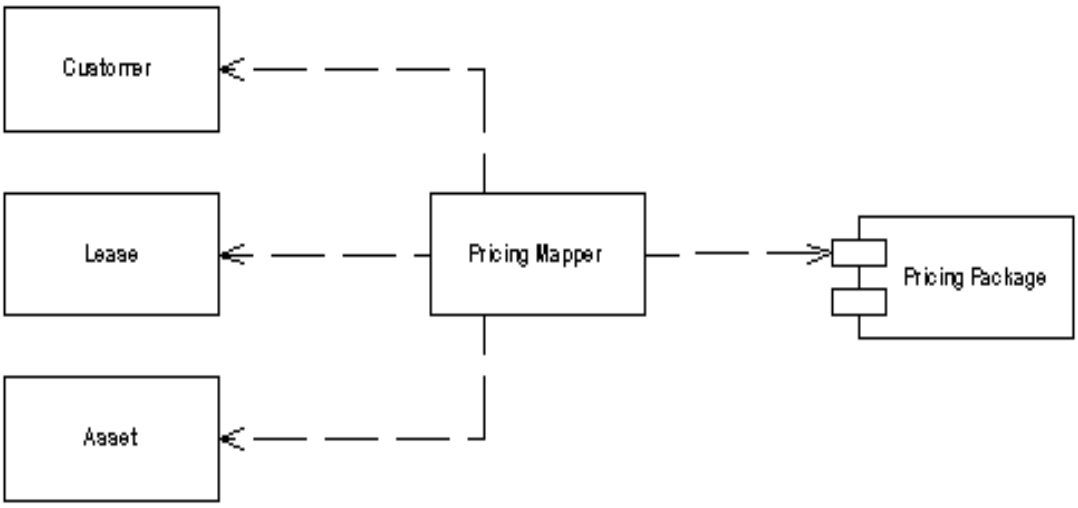
A.3.1 Gateway

Ein Gateway kapselt den Zugriff auf ein externes System Subsystem und bietet eine wohl definierte Schnittstelle auf dieses. Dessen konkrete Implementierung bleibt damit austauschbar.



A.3.2 Mapper

Realisiert die Kommunikation zwischen zwei unabhängigen Systemen. Im Gegensatz zum Gateway erlaubt der Mapper auch Callbacks aus dem Subsystem in den Aufrufer.



A.3.3 Layer Supertype

Gemeinsame Superklasse für alle Komponenten einer Schicht. In der J2EE bieten sich dafür natürlich an:

- Servlets
- Servlet-Filter
- Enterprise JavaBeans
- …

Darin können zentrale Aufgaben abgebildet werden

- Spezielle Unternehmens-spezifische Aufgaben
- Logging, Monitoring, Auditing…

Hinweis: Layer Supertype liefert damit auch die Möglichkeit, dass die Superklasse beispielsweise Dienste zur Verfügung stellt bzw. beispielsweise über Method Templates Sequenzen festlegt. Dieses Prinzip wird häufig auch als „Inversion of control“ bezeichnet.

A.3.4 Registry

Objekt, das zentral andere Objekte oder Dienste zur Verfügung stellt. Dies ist die e-Business-Formulierung eines Singletons:

Strategien zur Implementierung einer Registry sind beim GoF-Pattern Singleton diskutiert.

A.3.5 Value Object

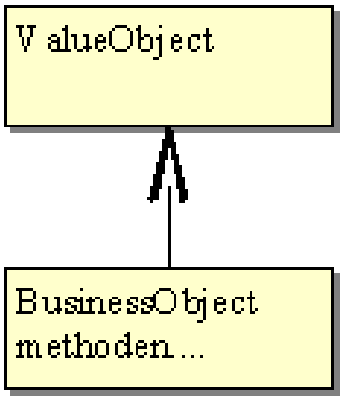
Ein „dummes Objekt, das nur zum Austausch von Daten „per value“ verwendet wird. Die Gleichheit eines Value Objects beruht auf gleichem Inhalt, nicht auf der Objekt-Identität. Ein Value Object verhält sich somit ähnlich einem einfachen Datentypen oder einer Struktur; ein Value Object enthält in der Regel keinerlei Geschäftslogik.

Die Java-Implementierung eines Value Objects ist eine einfache JavaBean ausschließlich mit Getter- und Setter-Methoden. Zusätzlich sind `equals` und `hashCode` zu implementieren.

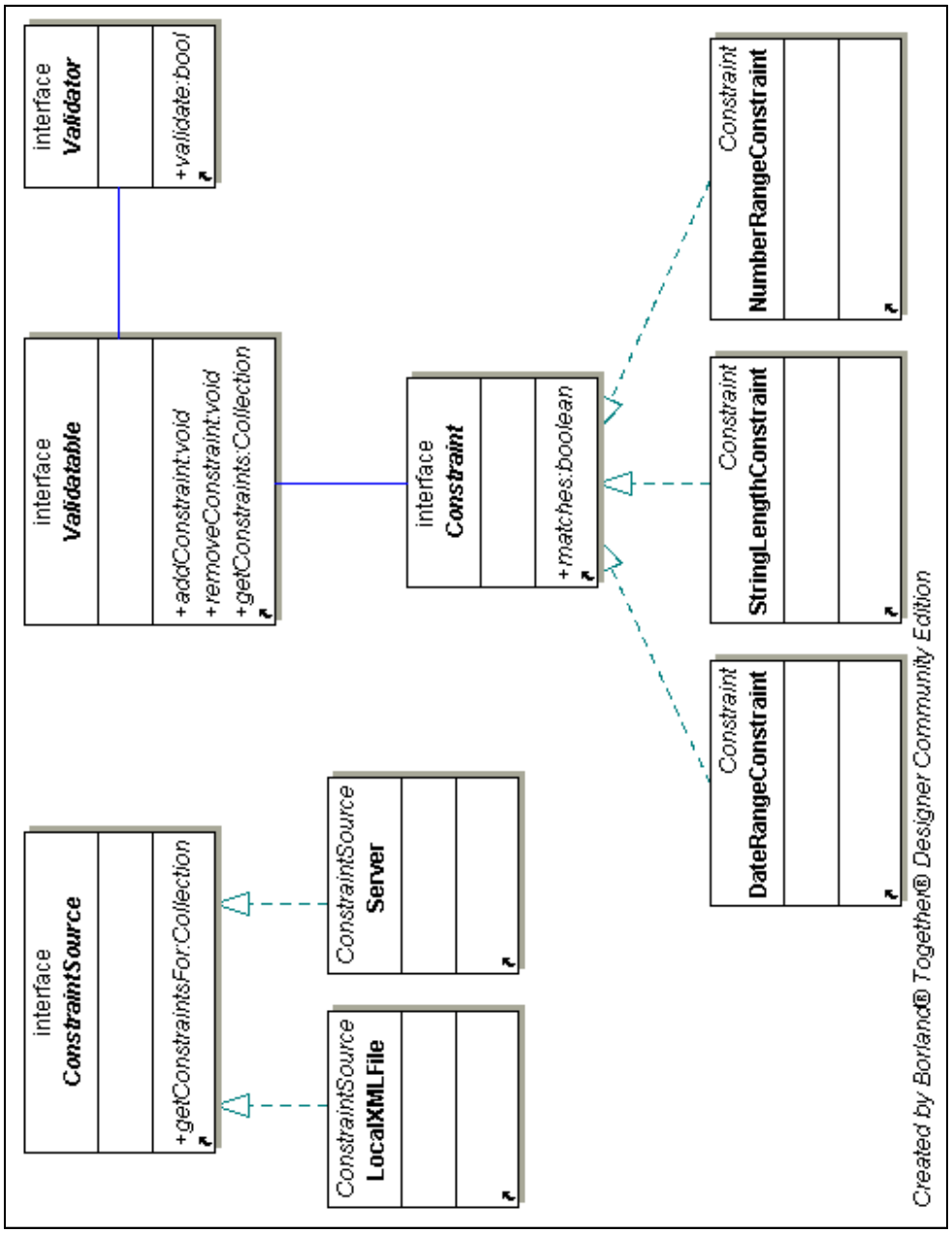
Hinweis: Die Diskussion „Wie viel Intelligenz dürfen Value Objects haben?“ ist nicht eindeutig lösbar, siehe das Paradebeispiel „Validierung“. Als Grundlage des Designs sollte aber die obige Definition erhalten bleiben.

Ein „intelligentes“ Value Object wird auch als „Business Object“ bezeichnet. Validierung etc. sind gut in diesem untergebracht:

„business object“ = Value-Object + beliebige Logik

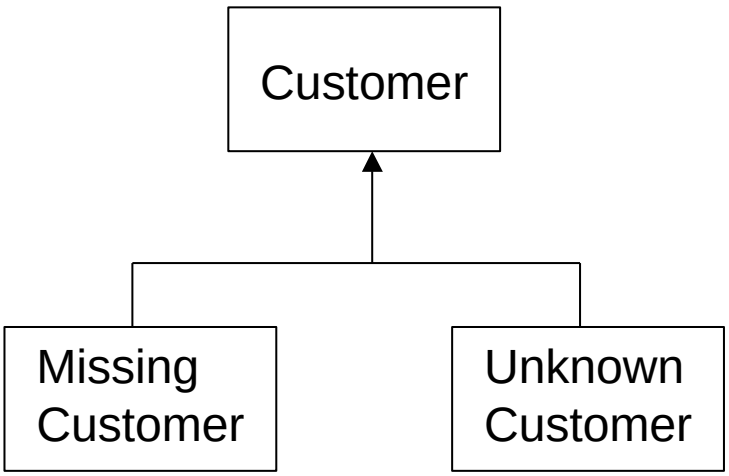


Beispiel: Validierung:



A.3.6 Special Case

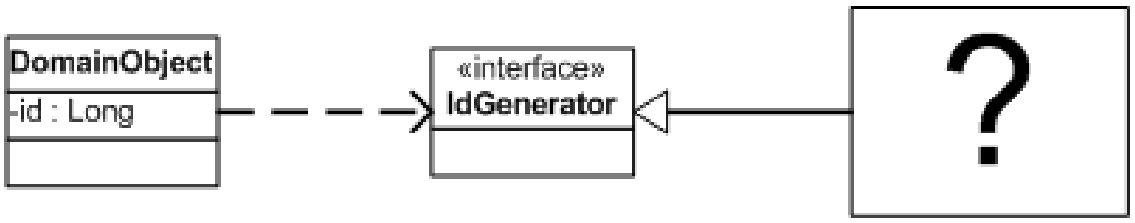
Subklassen, die einen Spezialfall abbilden, Beispiel: Ergebnis „Null-Referenz“ entspricht einer Instanz der Klasse „Missing Customer“:



Dieser Ansatz erleichtert den Umgang mit Daten im Objekt-orientierten Umfeld: Statt mehr oder weniger komplexen und unhandlichen Abfragen kann Polymorphie verwendet werden.

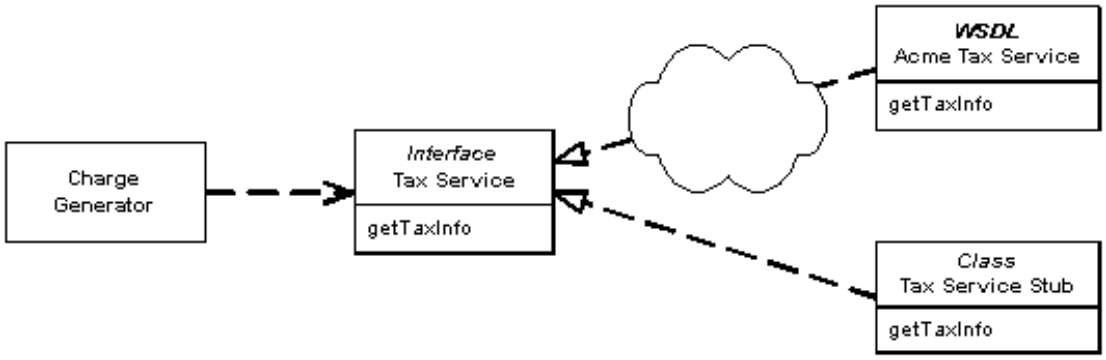
A.3.7 Plug In

Ein über eine separierte Schnittstelle definierter Dienst wird über eine konfigurierbare Implementierung zur Verfügung gestellt. Das Erzeugen der Implementierung übernimmt in der Regel eine konfigurierbare Factory:



A.3.8 Service Stub

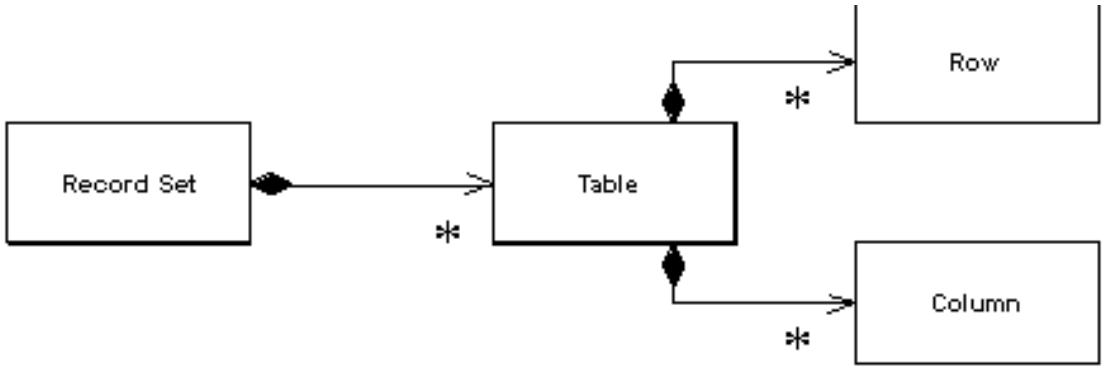
Der Zugriff auf Dienste wird wiederum durch eine Schnittstelle entkoppelt. Zu Testzwecken kann eine einfache Implementierung verwendet werden.



Dies entspricht der Idee der „Mock-Objekte“ bzw. von Testtreibern.

A.3.9 Record Set

Repräsentierung des Ergebnisses einer SQL-Abfrage im Speicher. Das Objekt wird von anderen Systemen zur Verfügung gestellt bzw. manipuliert:

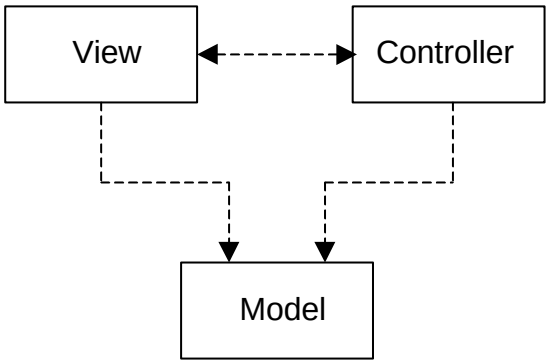


In Java ist sind dies die Implementierungen von `java.sql.ResultSet`.

A.3.10 Model View Controller

Die klassische Aufteilung in die drei Rollen

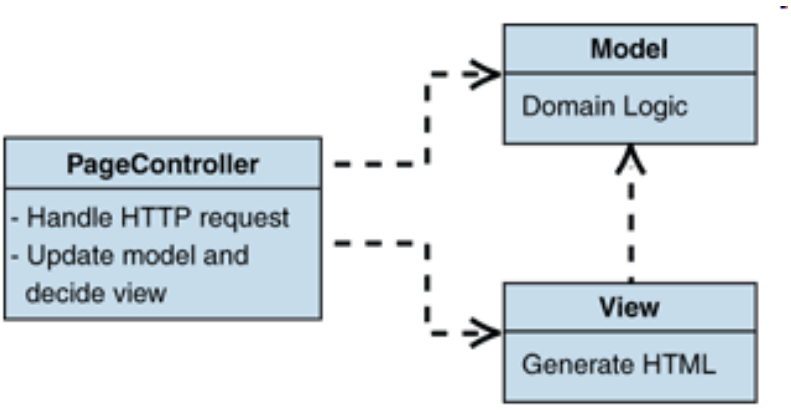
- Model enthält Domänenlogik
- View mit Präsentationslogik
- Controller



Dieses Pattern ist ein „Mega-Pattern“, das in praktisch allen Anwendungen mehrfach auftritt. Die einzelnen Bestandteile können selber wiederum nach MVC strukturiert sein. So kann z.B. das Datenmodell für eine Web-Anwendung selber eine View für den eigentlichen Datenbankzugriff darstellen.

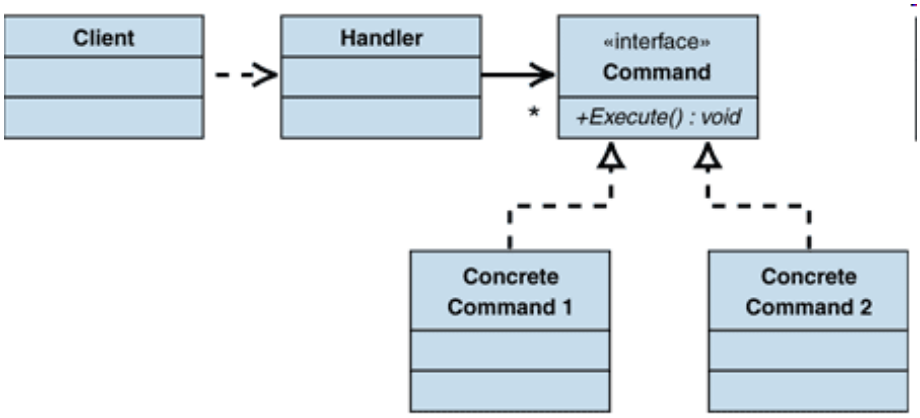
A.3.11 Page Controller

Dieses Pattern gilt für den typischen Ablauf einer Web-Anwendung, ist aber auch für dicke Clients (Beispielsweise Swing oder SWT) gültig. Pro Web Seite bzw. pro Dialog kontrolliert ein eigener Page Controller den Ablauf.



A.3.12 Front Controller

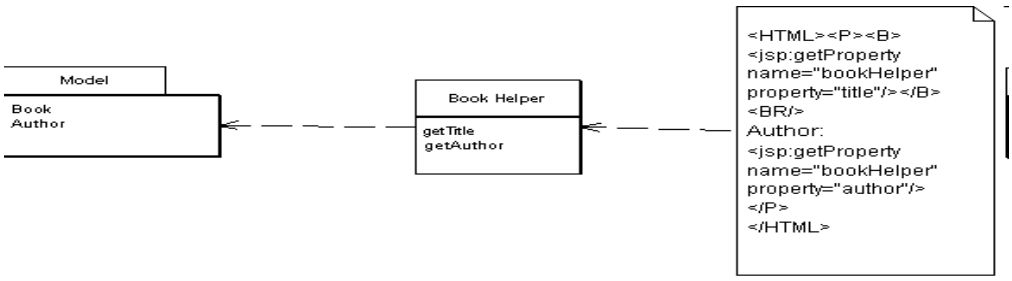
Ein zentraler Controller für die gesamte Web-Applikation:



Dieser zentrale Kontroller empfängt alle Requests und verteilt sie an die eigentliche Ausführungsschicht weiter. Dazu wird sinnvoller weise, wie in obigem Diagramm, das Command-Pattern eingesetzt.

A.3.13 Template View

Hinzufügen dynamischer Informationen zu einer HTML-Seite:

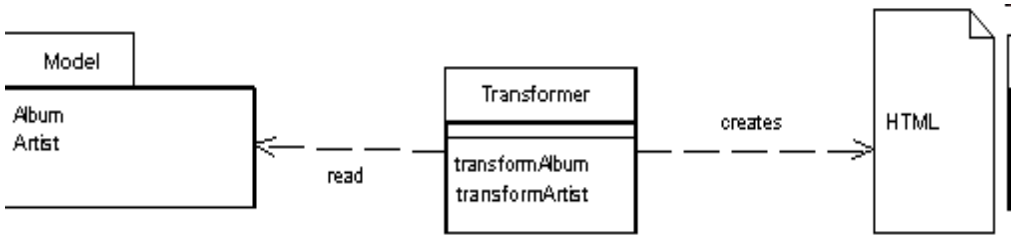


Im Java-Umfeld entspricht dies häufig den JavaServer Pages der J2EE.

Hinweis: Es gibt aber auch eine ganze Menge weiterer Template Engines, so z.B. Apache Velocity.

A.3.14 Transform View

Transformation von Datenformaten in andere Datenformate:

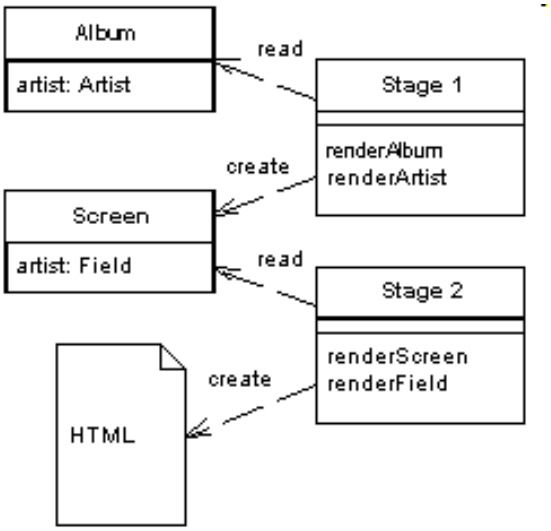


- Domänen-Objekt in ein HTML-Dokument
- XML-Transformationen

A.3.15 Two step View

Aufbereitung der präsentierten Seite in zwei Schritten:

- Aufbau des logisches Dokument
- Weitere „Verschönerung“, Personalisierung, Werbung…

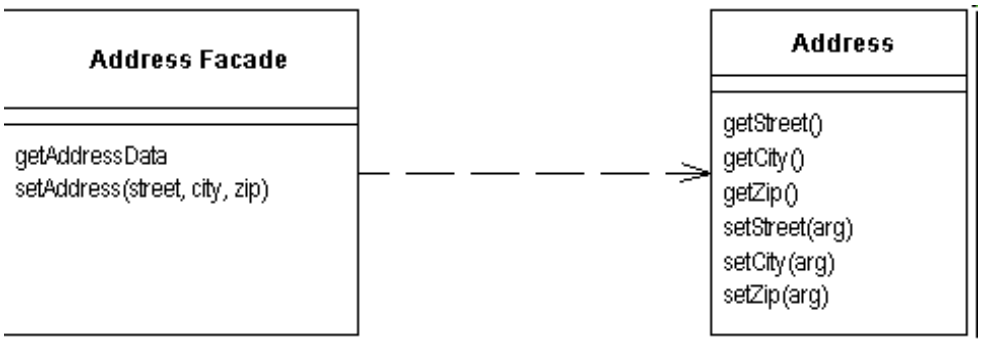


A.3.16 Application Controller

Ein zentraler Controller, der den Ablauf der Anwendung und die Abfolge der dargestellten Seiten steuert.

A.3.17 Remote Facade

Die Remote Facade bietet einen grob-granulierten Einstiegspunkt für die e-Business-Anwendung mit Rückgabe von Value Objekten:



Die Remote Facade entspricht somit in wesentlichen Elementen der klassischen Facade.

A.3.18 Data Transfer Object

Container, um Daten zwischen Prozessen und logischen Schichten austauschen zu können

Data Transfer Objekte werden in der Regel vom Server erzeugt, beispielsweise von der Remote Facade. Diese Funktionalität wird meistens von einer Hilfsklasse, dem Assembler übernommen.

Hinweis: Dies ist nicht gleichbedeutend mit einem Value Object! In Java können Transfer Objekte zwar in erster Näherung durch serialisierbare Value Objects realisiert werden. Trotzdem wäre es fatal, beides gleich zu setzen.

A.3.19 Client Session State

Halten von Session Informationen auf dem Client. Damit wird der Server entlastet, da keine oder weniger Session Informationen gehalten werden müssen.

Zur Realisierung kann der Aufrufer alle notwendigen Informationen als Parameter übergeben, inklusive aller Session Informationen mit senden.

A.3.20 Server Session State

Halten von Session-Informationen auf dem Server. Dies ist eine sehr einfache Möglichkeit der Session Verwaltung, da keinerlei Client-Abhängigkeiten zu berücksichtigen sind: Der Client braucht nur noch eine „Session ID“ zu halten oder über eine feste Verbindung mit dem Server zu kommunizieren. Dafür treten aber eventuell Probleme durch Migrieren großer Sessions auf, was bei ausfallsicheren Systemen durchgeführt werden muss.

A.3.21 Database Session State

Auslagern der Session Information in eine Datenbank.

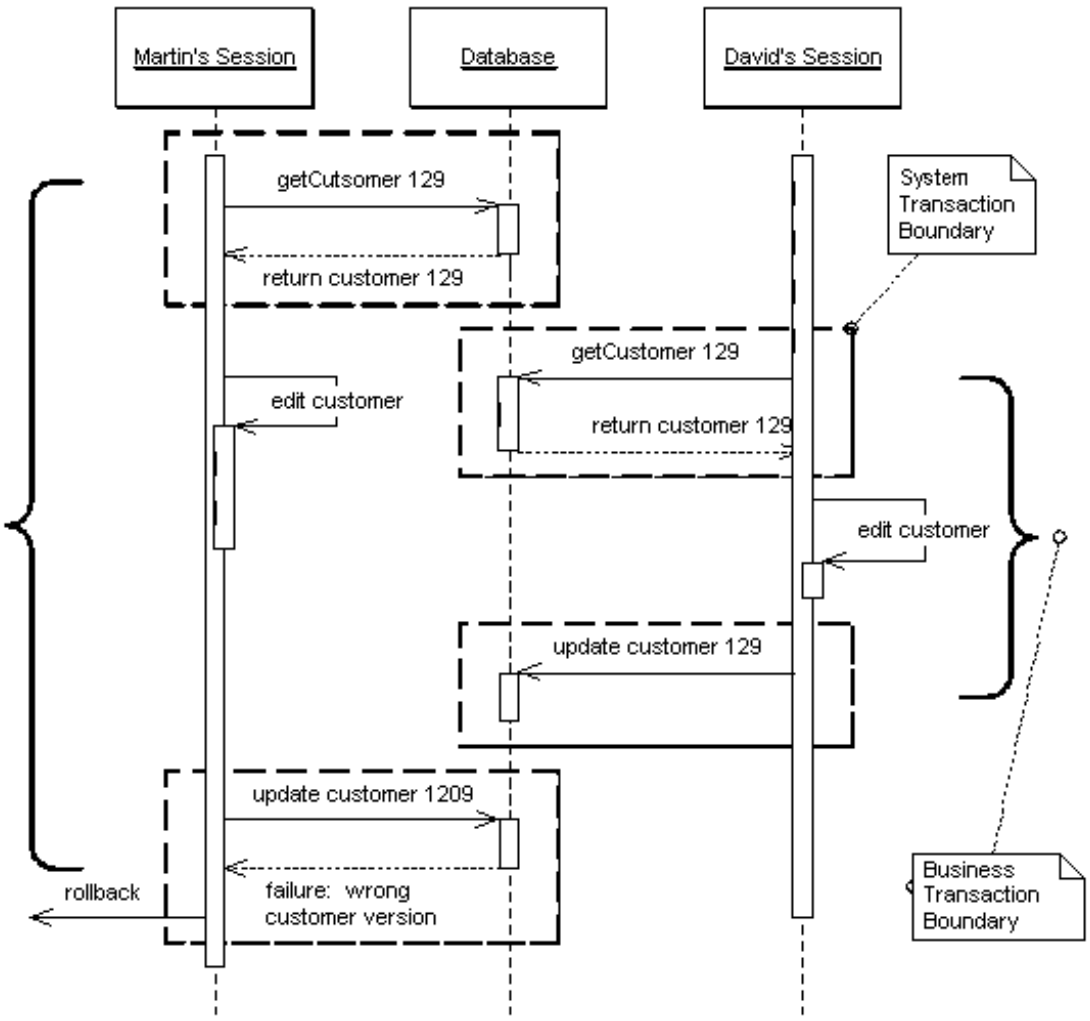
Hinweis: Dies kann bei Applikationsservern häufig durch Konfiguration erreicht werden

Auch hier sendet der Client nur eine Session ID, die nun jedoch nur als eindeutiger Schlüssel für einen Datenbankzugriff verwendet wird. Damit wird der Speicher des Servers entlastet.

Hinweis: Auch Datei-basierte Session States werden häufig eingesetzt.

A.3.22 Optimistic Offline Lock

Eine eventuelle Dateninkonsistenz wird durch Prüfung einer Versionsnummer bzw. einer eindeutigen ID der letzten Änderung geprüft.



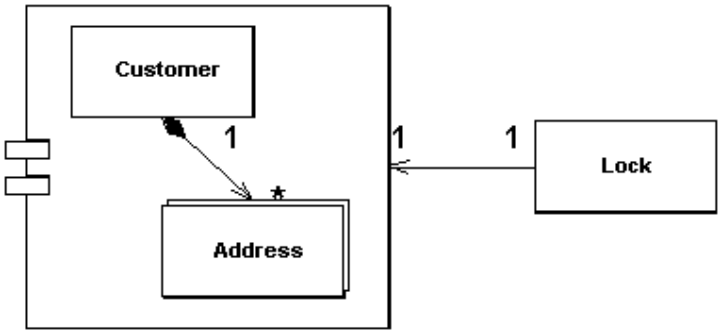
Fehler durch inkonsistente Daten werden erst beim Versuch des Speicherns registriert. Dafür können beispielsweise rein lesende Zugriffe einfach parallel ablaufen.

A.3.23 Pessimistic offline lock

Hier wird das Sperren des Datensatzes durch das Sperren eines Datenobjektes realisiert, Dateninkonsistenzen werden durch das Serialisieren der Zugriffe sicher vermieden.

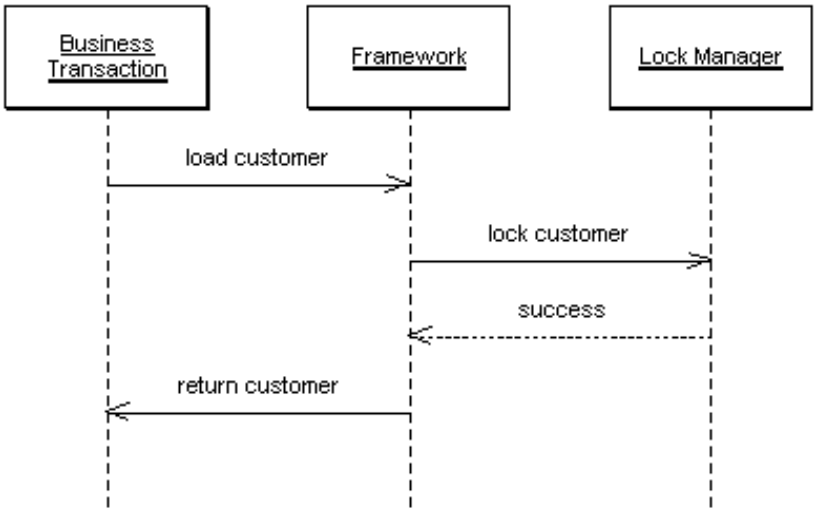
A.3.24 Coarse Grained Lock

Das Locken einer Menge von Objekten mit einem einzigen Lock-Objekt:



A.3.25 Implicit Lock

Hier wird das Sperren von einer Superklasse oder dem Framework übernommen:



A.3.26 Transaktionsskript

Ausführen der Geschäftslogik als einfache Folge einzelner Datenbankbefehle innerhalb einer Prozedur. Damit wird auf die Erzeugung von eventuell komplexen Objektbäumen verzichtet, die in bestimmten Anwendungs-Situationen überflüssig sein können:

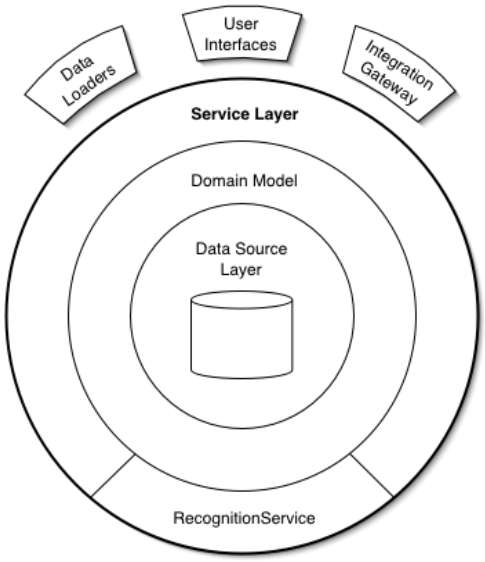
- Schreiben eines Datensatzes (Rechnung), ohne diesen sofort bearbeiten zu müssen
- Darstellung einer Tabellen-View in einer Client-Tabelle.

A.3.27 Domain Model

Ein Objektmodell der Domänenlogik wird bei komplexen Abläufen verwendet. Hier werden im Gegensatz zum Transaktionsskript Objekte benutzt, deren Erzeugung in der Regel automatisch von einem Modul übernommen wird.

A.3.28 Service Layer

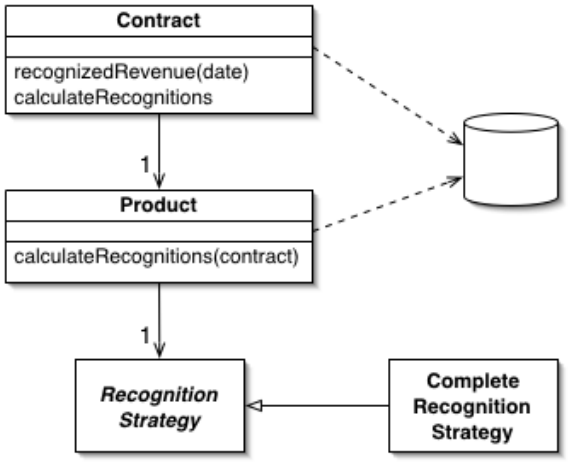
Mehrere Domänenmodelle und/oder Transaktionsskripte werden zu einem Service-Layer zusammengefasst:



Wesentliches Kennzeichen eines Services ist seine exakte Definition. Wie diese Spezifikation erfolgt, ist Sprach- bzw. Architekturabhängig. Im Java-Umfeld sind üblich Java-Schnittstellen, IDL-Schnittstellen für CORBA oder aber WSDL-Dokumente für Web Services.

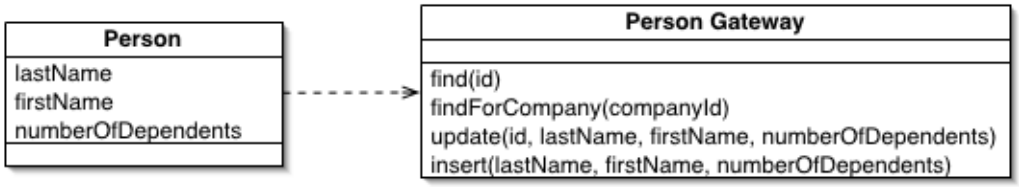
A.3.29 Table Modul

Ein einzelnes Objekt, das den Zugriff auf eine Tabelle komplett übernimmt:



A.3.30 Table Data Gateway

Ein Objekt, das als Gateway für eine Datenbanktabelle dient:

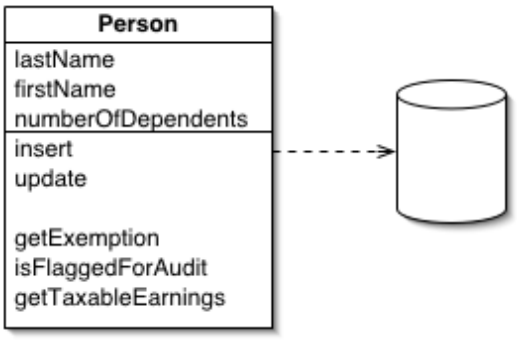


A.3.31 Row Data Gateway

Objekt, das als Gateway für eine Tabellenzeile dient.

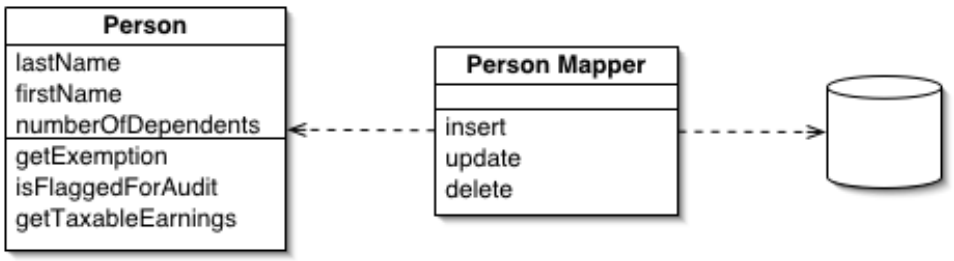
A.3.32 Active Record

Objekt, das einer Tabellenzeile entspricht und zusätzliche Domänenlogik enthält, um den Datenbestand des Objektes mit der Datenbank zu synchronisieren.



A.3.33 Data Mapper

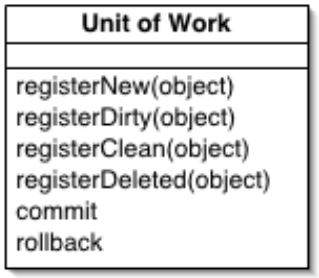
Eine Schicht von Mapper-Objekten, die die Synchronisierung zwischen den Objekten und der Datenbank übernimmt und damit für eine Entkopplung von Objekt- und Datenmodell sorgt:



A.3.34 Unit of Work

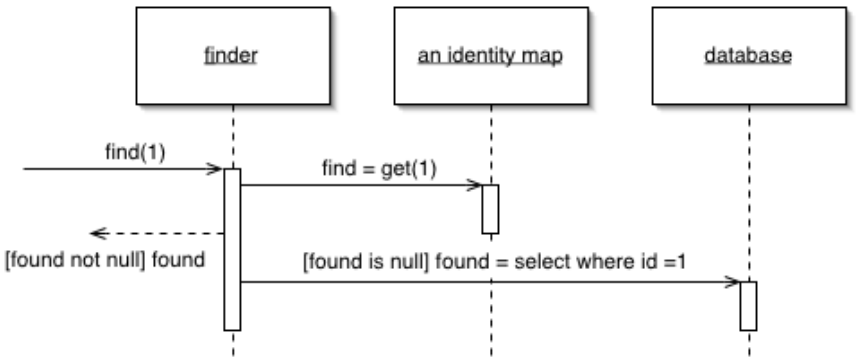
Ein Objekt, das eine Liste von Objekten hält, die im Rahmen einer Transaktion geändert wurden:

- Schreiben der Daten
- Lösen eventuell vorhandener Probleme durch gleichzeitigen Zugriff



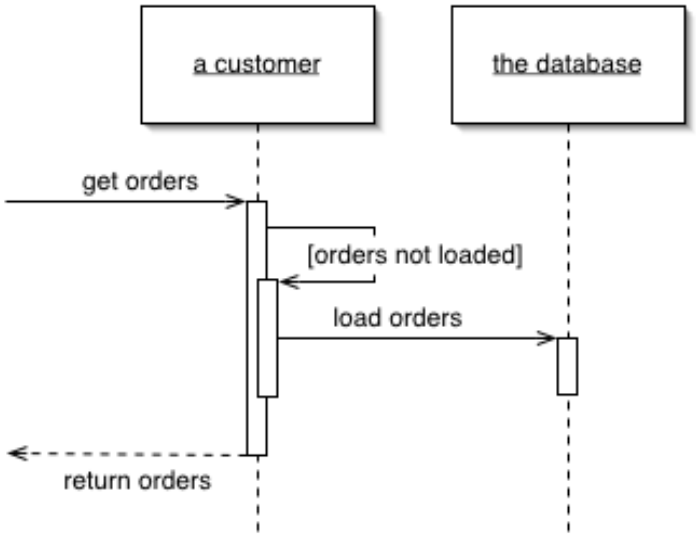
A.3.35 Identity Map

Eine Map, die ein mehrfaches Laden von Daten verhindert:



A.3.36 Lazy Load

Ein Objekt, das die Daten nicht vollständig hält, aber „weiß“, wie sie geladen werden können:



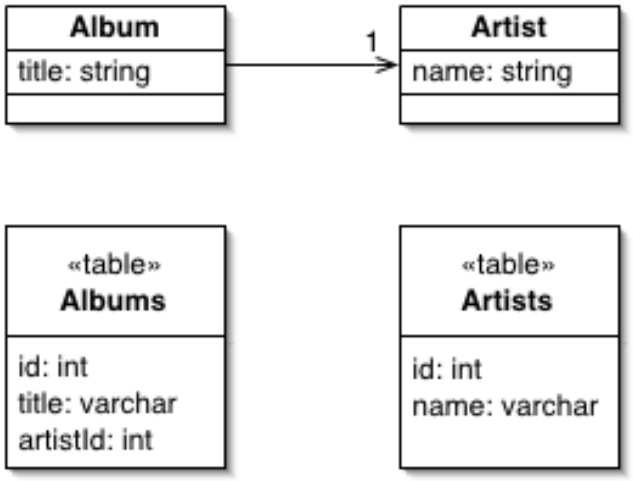
Dies entspricht einem Proxy für Daten.

A.3.37 Identity Field

Der Primärschlüssel eines Datenbestandes wird als Attribut innerhalb eines Objekts abgelegt.

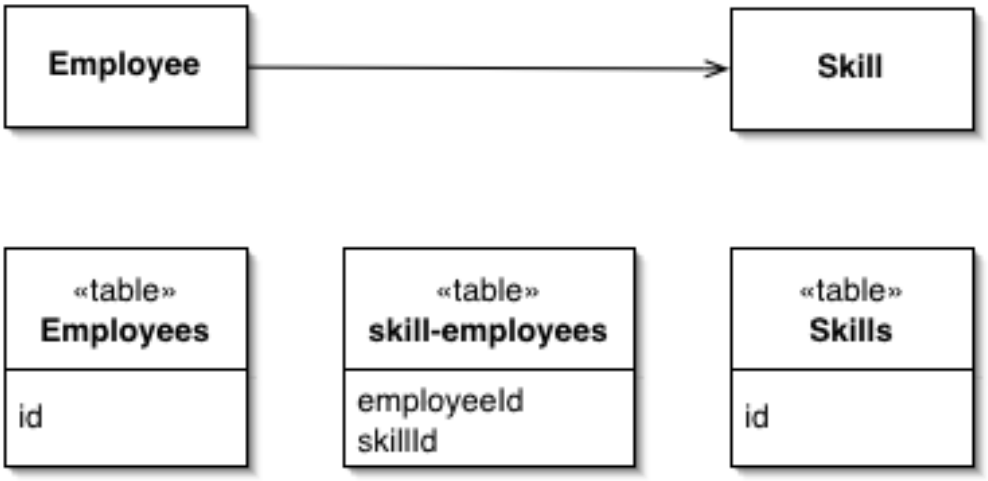
A.3.38 Foreign Key Mapping

Abbilden von Objekt-Assoziationen und Datenbankrelationen:



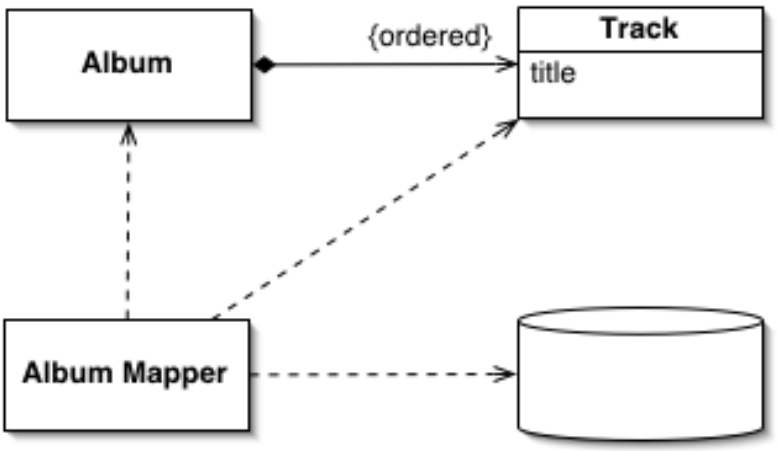
A.3.39 Association Table Mapping

Die Assoziation zwischen Objekten wird durch eine Verknüpfungstabelle realisiert:



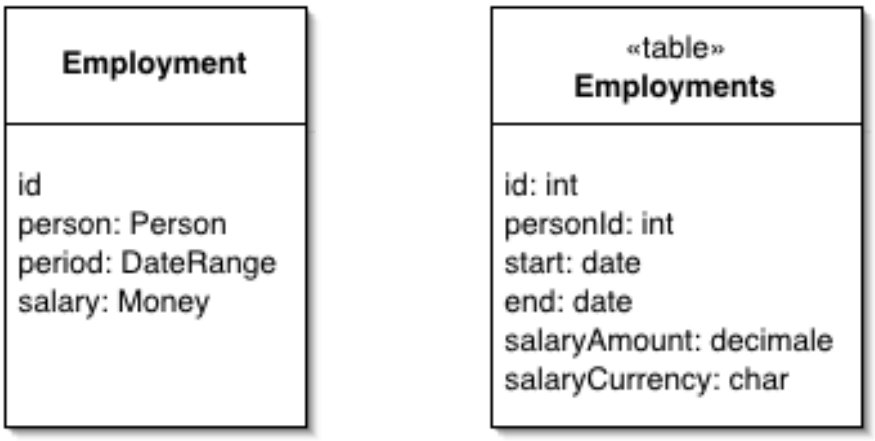
A.3.40 Dependent Mapping

Ein einzelner Mapper liest abhängige Daten:



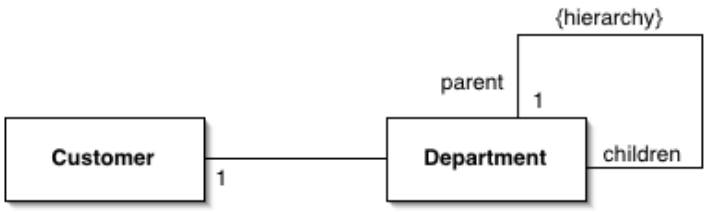
A.3.41 Embedded Value

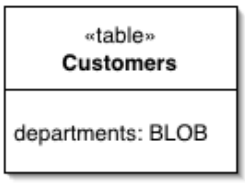
Ein Objekt hält ein anderes Objekt, das aber keine Entsprechung als eigene Datenbanktabelle hat:



A.3.42 Serialized Large Object

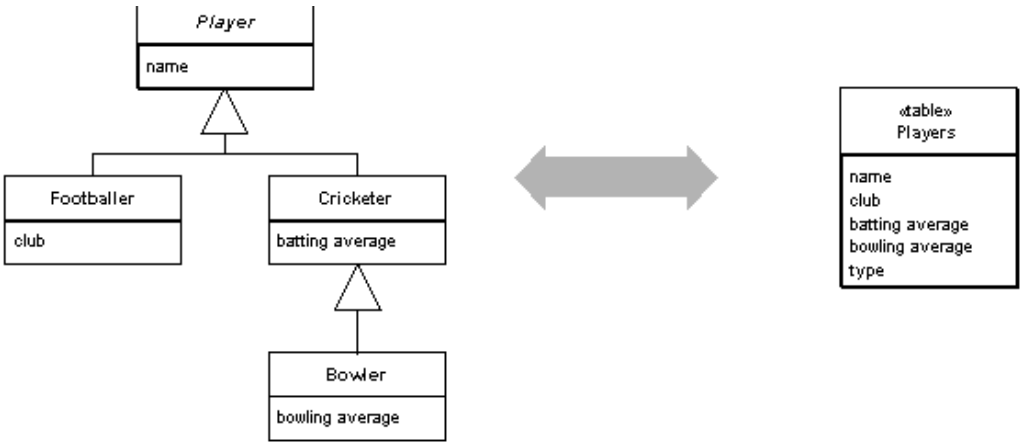
Mächtige abhängige Objekte werden als serialisiertes Objekt abgelegt. Damit werden jedoch die Suche und weitere Relationen erheblich erschwert:





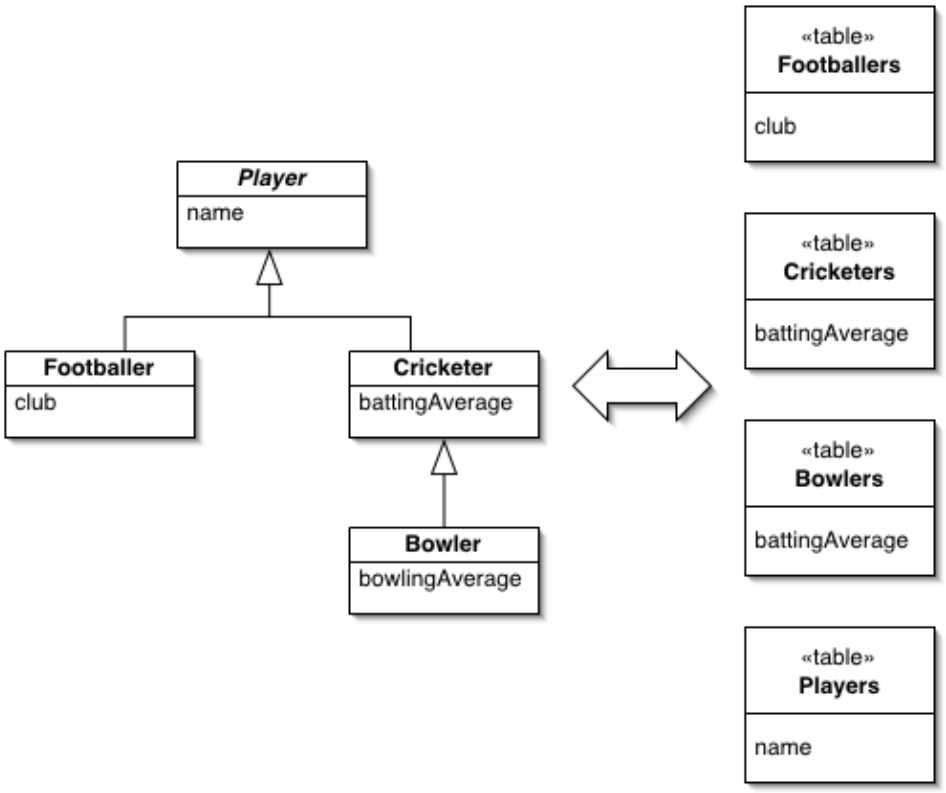
A.3.43 Single Table Inheritance

Eine Klassenhierarchie, die durch den Inhalt von Tabellenspalten gebildet wird:



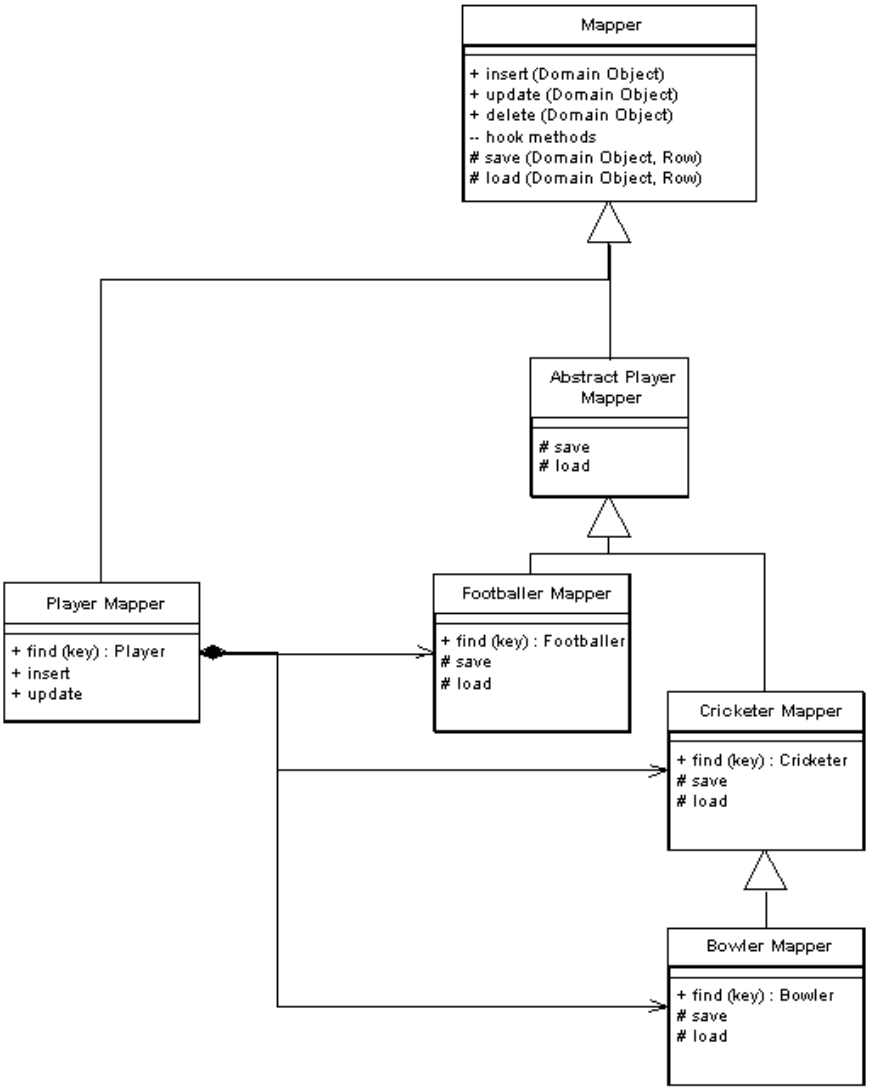
A.3.44 Class Table Inheritance

Eine Klassenhierarchie, die durch mehrere Tabellen definiert wird:



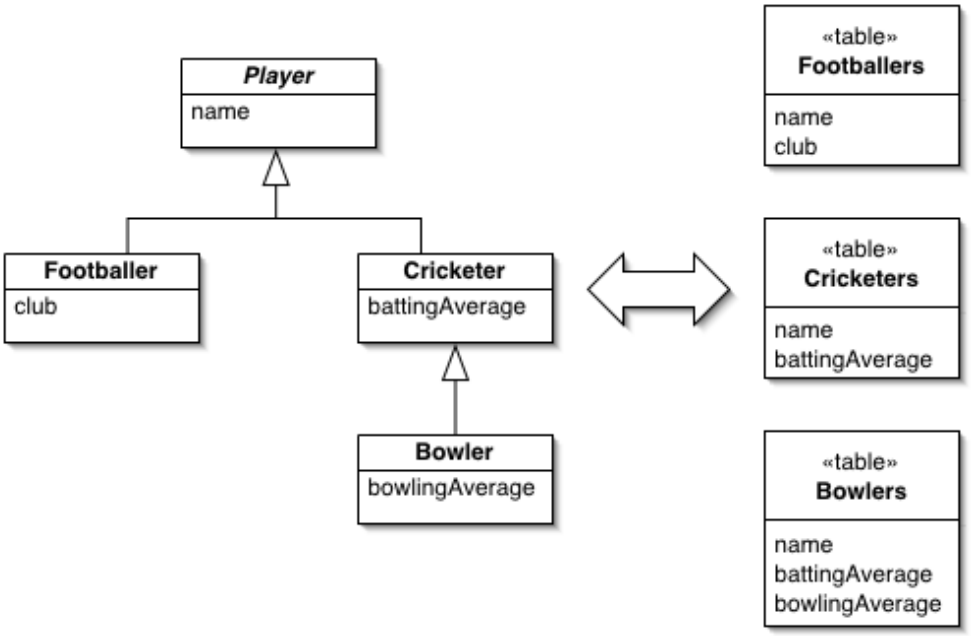
A.3.45 Inheritance Mappers

Hierarchie von Mappers, die die Vererbungshierarchie aufbauen:



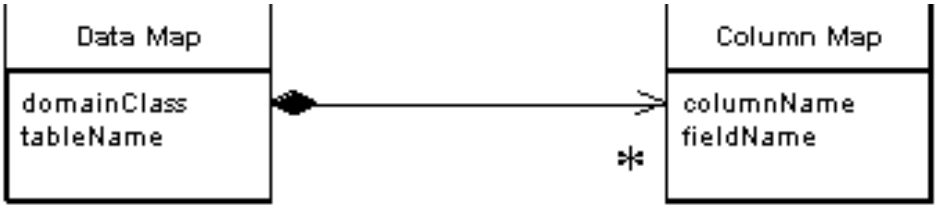
A.3.46 Concrete Table Inheritance

Wie Class Table Inheritance, jetzt existieren aber nur Tabellen für die nicht-abstrakten Klassen



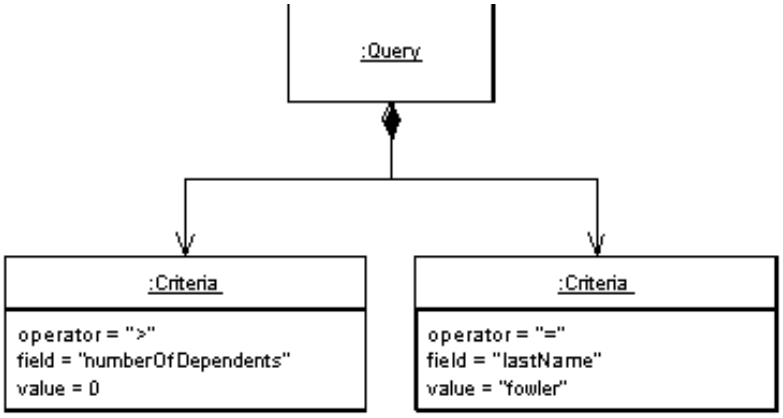
A.3.47 Metadata Mapping

Objekt, das die Metadaten des Datenmodells hält



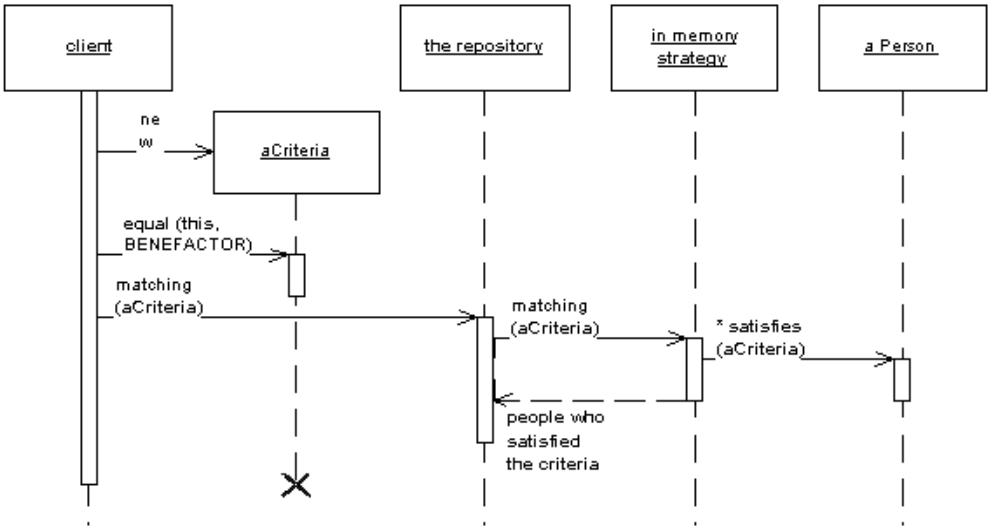
A.3.48 Query Objekt

Ein Objekt, das eine Datenbankabfrage hält:



A.3.49 Repository

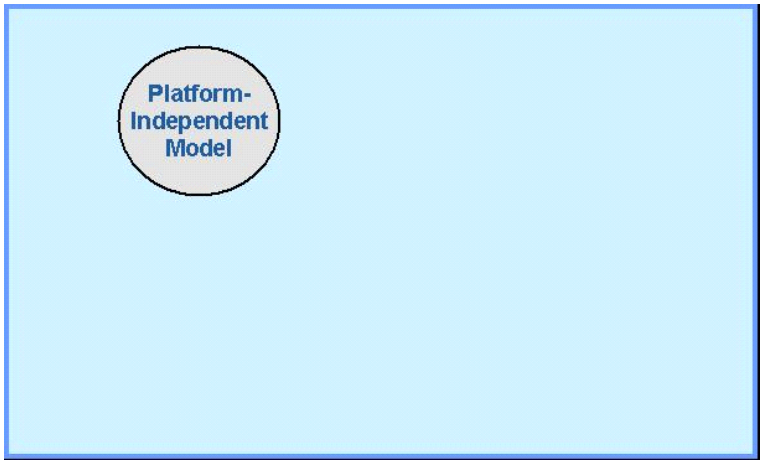
Der Zugriff auf die Domänenobjekte wird durch eine weitere Schicht gekapselt:



A.4 Model Driven Architecture

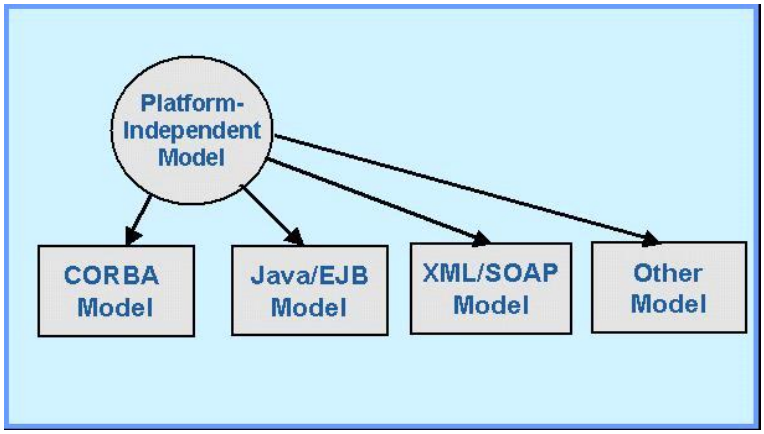
Mit Hilfe der oben angeführten Patterns kann die Anwendung in den wesentlichen Abläufen und mit den notwendigen Sequenzen definiert werden. Konkretere Annahmen über eine konkrete Plattform (J2EE, .NET oder CORBA) waren bisher nicht notwendig. Weitere technisch begründete Komponenten und Klassen verlangen nun aber natürlich genau diese zusätzliche Information. Andererseits sind diese neuen Bestandteile der Anwendung in der Standard-Implementierung ohne weiteren Designprozess aus dem bereits definierten fachlichen Modell ableitbar. Dies ist die Grundlage der Model Driven Architecture, ein seit Anfang des Jahres 2000 von der Object Management Group spezifizierte und kontrollierte Ansatz.

Der Startpunkt ist ein Plattform-unabhängiges fachliches Modell, das mit Hilfe der UML 2.0 definiert wird[3]. Darin sind beispielsweise auch Constraints mit Hilfe der „Objects Constrained Language“ abgelegt, semantische Abläufe mit der „Action Language“

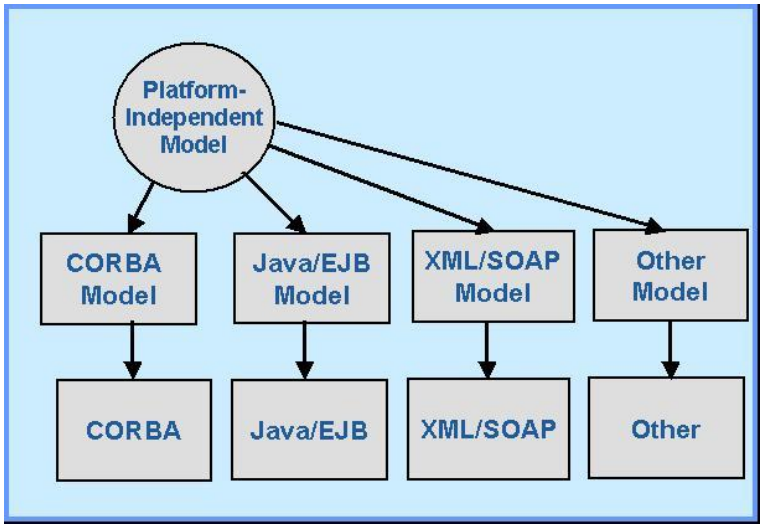


[3] Die Idee, in der UML alle Informationen inklusive Ausführungslogik (Constraints etc) abzulegen und alle daraus resultierenden Quellcodes in beliebigen Sprachen umsetzen zu können führt zu dem Begriff des „Executable UML“. Allerdings ist weder dir Werkzeugunterstützung noch die Akzeptanz der Programmierer bisher zufrieden stellend.

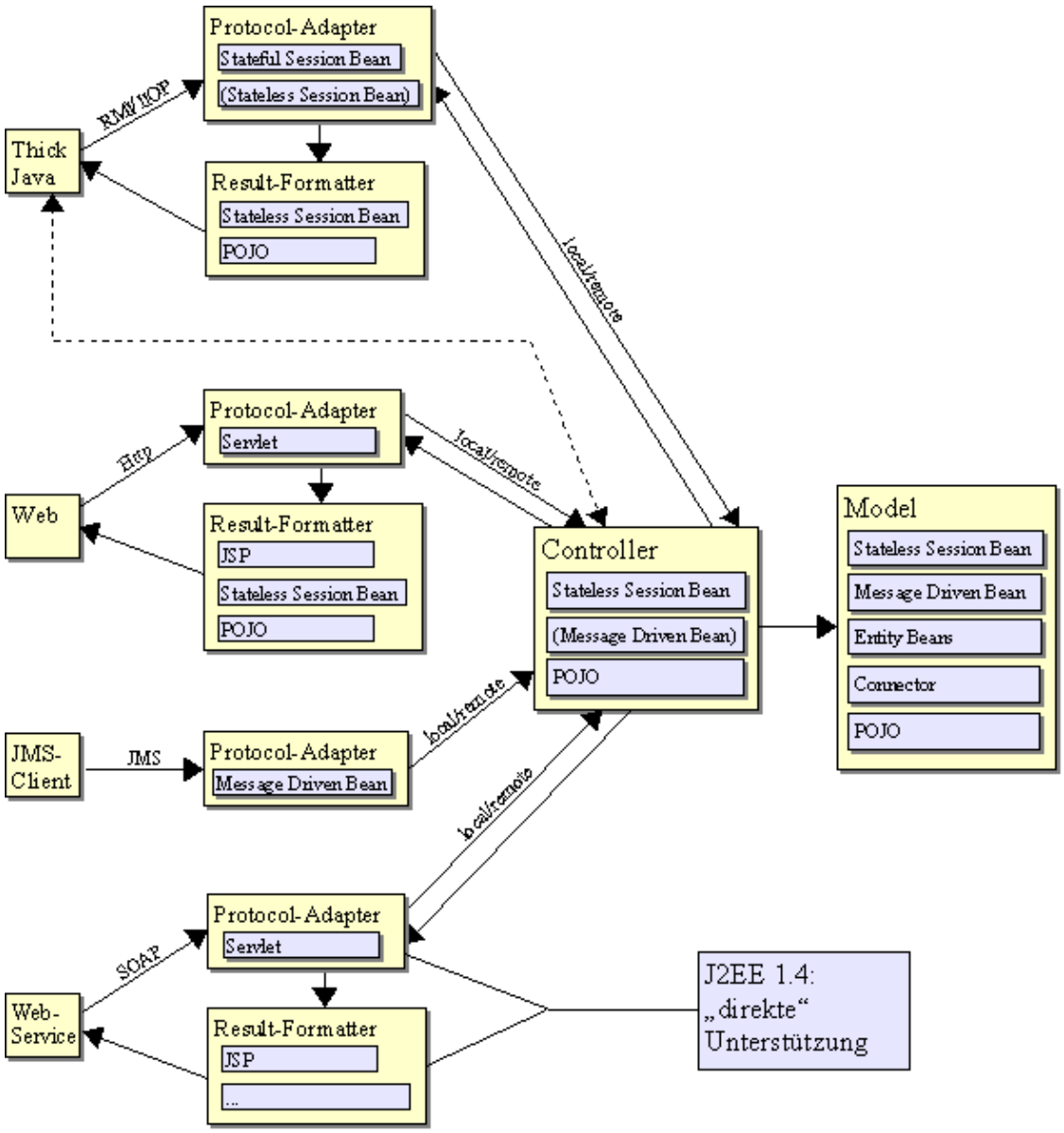
Im nächsten Schritt wird das Plattform-abhängige Modell abgeleitet. Dieses kann dann selber wiederum aus bestimmten Patterns und Idiomen bestehen:



Nun erst erfolgt in einem letzen Schritt die Erzeugung der Plattform-abhängigen Programme:



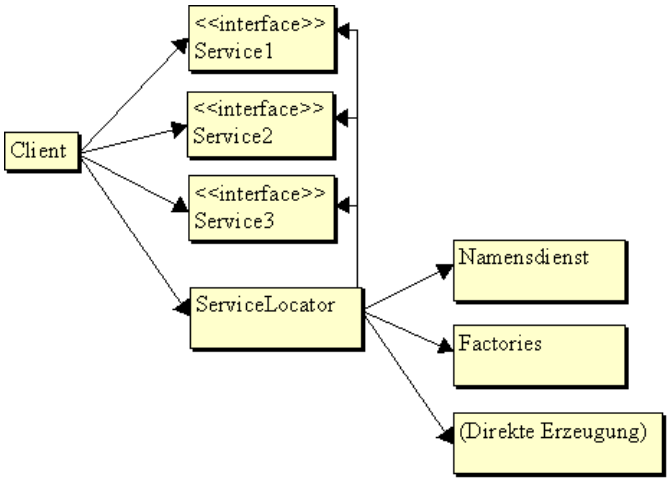
A.5 Das Basis-Design mit JEE-Komponenten



A.6 JEE Patterns

A.6.1 Service Locator

Der Service Locator kapselt den Zugriff auf Dienste des Applikationsservers und entspricht damit im Wesentlichen dem Pattern „Registry“:

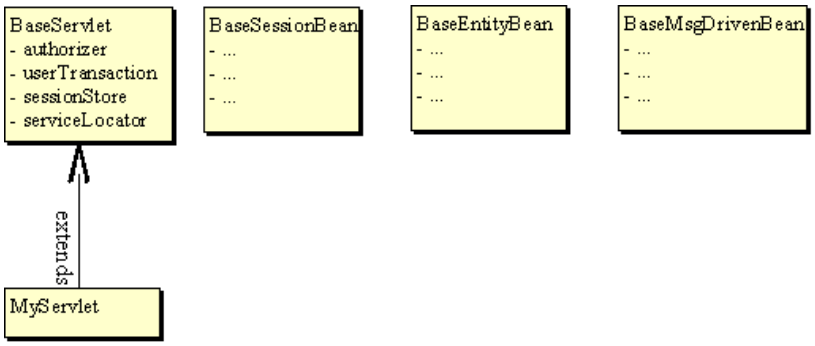


Service-Locator werden als Caches für die Dienste benutzt.

Vorteil: Performance-Steigerung, der JNDI-Lookup benötigt etwa 10-20 msec

Nachteil: Problematisch ist ein Ausfall/Restart von Cluster-Bestandteilen

A.6.2 Layer Supertype für JEE

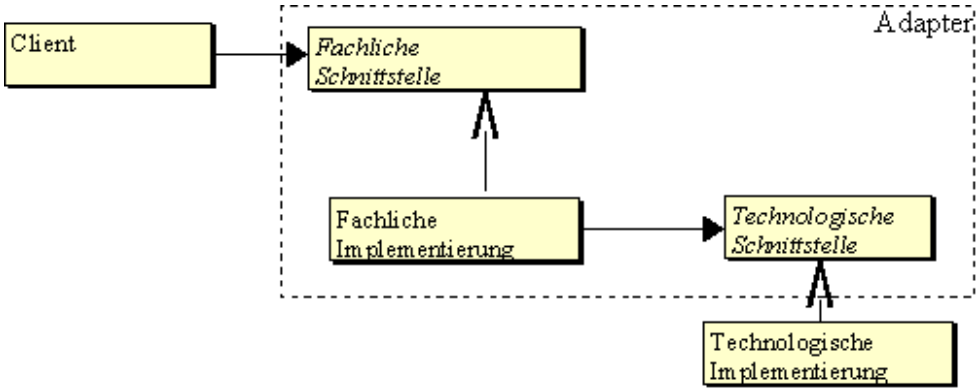


Hinweis: Die EJB 3.0-Spezifikation enthält Elemente des Layer Supertype. So wird der Zugriff auf Environment, andere EJBs, DataSources etc. automatisch jeder Bean als Getter-Methode zur Verfügung gestellt.

Hinweis: Die Vererbung ist hier nicht kritisch, da es sich bei den Vater-Klassen um rein technische Hüll-Klassen handelt.

A.6.3 Business Delegate und Session Facade

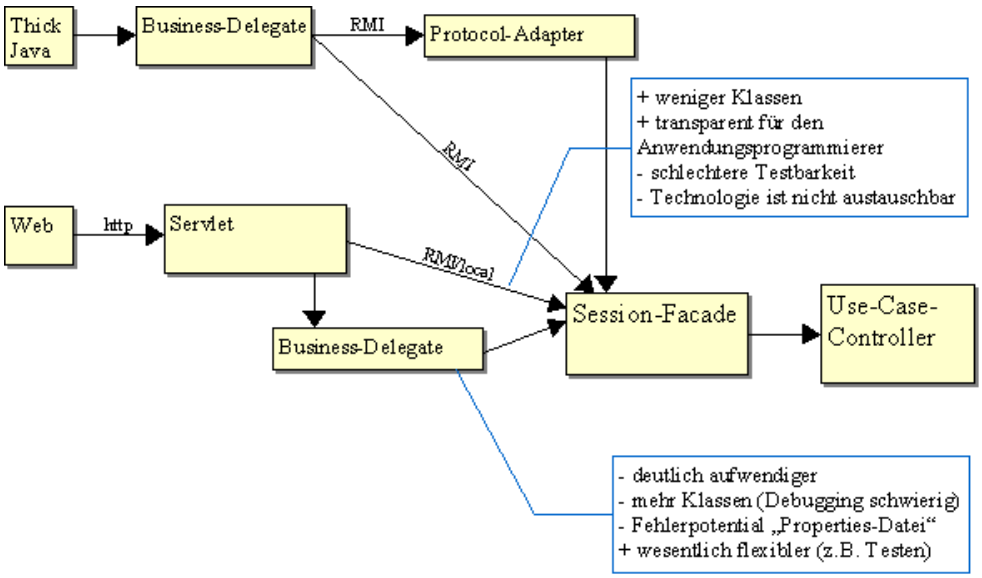
Zwischen der fachlichen Schnittstelle und der SessionBean wird ein „Technologie-Adapter“ geschaltet. Dieser bekommt den Namen „Business Delegate“, sein Pendant auf Serverseite ist die Session Facade:



Die Kombination von „Business Delegate“ und „Session Facade“ ermöglicht die getrennte Optimierung der technologischen Implementierung.

Diese beiden Patterns haben eine direkte Entsprechung zu zwei Patterns für verteilte Anwendungen: Service Stub und Remote Facade.

Der Einsatz dieser beiden Patterns definiert bereits ein wesentliches Basis-Design für das JEE-Modell:



A.6.4 Value List Handler

Ein Beispiel für die Verwendung einer Stateful SessionBean ist ein Value List Handler. Die Problemstellung dieses Patterns lautet: „Wie kann das RMI-Protokoll für Collections eine Fetch-Size bekommen?“. Der Begriff „Fetch-Size“ ist dem Protokoll für Datenbankzugriffe entnommen und bedeutet, dass beispielsweise eine Ergebnismenge von 1000 Treffern in einstellbaren Blöcken von z.B. 100 Datensätzen übertragen wird.

Diese Aufgabe ist eine Erweiterung des RMI-Protokolls, die somit von einem Protokoll-Adapter, eben dem Value List Handler, übernommen wird. Dieser ist somit nichts anderes als ein Iterator, der auf Serverseite die gesamte Ergebnismenge hält und auf Client-Anforderung jeweils den nächsten Block liefert.

Zur Verwendung eines Value List Handlers gibt es zwei Ausprägungen:

- Die Iterator-Logik ist Bestandteil des Use Cases. Wenn bei der Definition der Controller-Schnittstelle bereits ein seitenweises Anzeigen der Daten vorgesehen ist (Paradebeispiel: Web Anwendung), so kann die Iterator-Signatur bereits dort vorgesehen werden.
  - Nachträgliches Einführen des Iterierens ohne die Möglichkeit, die Controller-Signatur zu ändern. Dies ist eine typische Optimierungsmaßnahme und erfordert keine (!) Änderung der Clients:
  - Die Einführung der Stateful SessionBean mit Iterator-Signatur wird vom Business Delegate vollständig gekapselt.
- Der Client bekommt eine spezielle Value List Handler-Collection-Implementierung, die aktiv die noch nicht geladenen Daten nachladen kann. Diese Collection ist somit ein Proxy, der über eine Lazy Loading-Strategie die Daten nachladen kann.

A.6.5 Value Object und Transfer Object

Das notwendige „Request/Response“ Verhalten mit „call-by-value“-Semantik beeinflusst, wie bereits mehrfach erwähnt, das Design des Use-Case-Controllers:

call by reference	call by value

`Book book = bookManager.findBook("...");`
`book.setPrice(42);`
	`Book book = bookManager.findBook("...");`
	`book.setPrice(42);`
	`bookManager.updateBook(book);`
	`//oder`
	`bookManager.updateBook("...", 2);`

Das Pattern „Value Object“ ist auch als JEE Pattern anzutreffen, wobei hier der Fokus mehr auf dem Transfer von Daten über Schichtengrenzen hinweg gemeint ist. Es ist deshalb besser, von einem „Data Transfer Object“ oder kurz „Transfer Object“ zu sprechen:

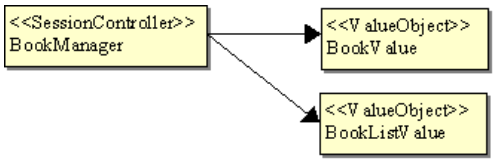
Zur Übertragung zwischen Client und Server muss ein Objekt serialisierbar sein.

Mögliche Data Transfer Object-Implementierungen sind:

- Value-Object mit „implements Serializable“
  - Daten-Container mit key-value-Paaren, also im Wesentlichen eine Map
  - JMS-Messages
  - RowSet, ein serialisierbares ResultSet
- XML, dieses Transfer-Objekt ist aber nur aufwändig zu erzeugen und sollte deshalb nur eingesetzt werden, wenn Daten bereits als XML vorliegen. Eine Erzeugung aus einem Java-Objekt ist sonst nur durch notwendige Interoperabilität zu begründen.

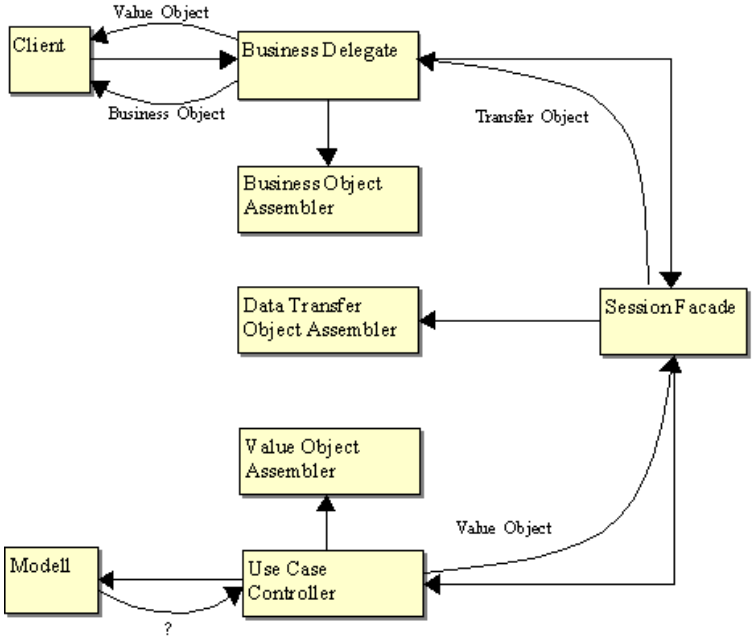
A.6.6 Value Object Assembler, Data Transfer Object Factory

Die Erzeugung der Value/Transfer Objects übernimmt entweder der fachliche Controller oder die Session Facade.



In beiden fällen übernimmt diese Erzeugung meistens eine Hilfsklasse, der Value Object Assembler bzw. die Data Transfer Object Factory, beides ebenfalls JEE Patterns.

In Zusammenhang mit dem Begriff des Business Objects ergibt sich hieraus ein mögliches Design wie folgt:



Vorschlag für eine geeignete Paketstruktur:

- de.integrata.<use-case-diagramm-name>
- Use-Case-Controller-Schnittstelle
- Factory für Use-Case-Controller
  - fachliche Exception-Klassen
  - de.integrata.<use-case-diagramm-name>.values
- Value-Objects
  - Value-Object Assembler
  - de.integrata.<use-case-diagramm-name>.implementation
- de.integrata.<use-case-diagramm-name>.implementation.mock
  - Mock Use-Case-Controller
- de.integrata.<use-case-diagramm-name>.implementation.ejb
  - Session Facade (Implementierung + 2 Schnittstellen)
  - Ejb Business Delegate

Dies sind mehr als 10 Schnittstellen/Klassen für einen Use-Case. Durch den Model Driven-Ansatz sind aber die allermeisten dieser Klassen trivial generiert.

A.6.7 Data Access Object

A.6.7.1 Definition

„Data-Access-Objects“ (DAOs) sind die Controller für die Datenzugriffe. Das DAO, der „Model-Controller“, kann, falls die Architektur dies erlaubt, lokal angesprochen werden. Damit sind „Live“-Objekte möglich, die via Referenz übergeben werden können. Ist der lokale Zugriff nicht möglich (Beispiel: Servlet-Container und Persistenz-Framework laufen aus Sicherheitsgründen auf unterschiedlichen Maschinen), so muss auch hier mit Value Objects gearbeitet werden. Diese Objekte werden in der Literatur auch als „Data Objects“ bezeichnet.

Das Data-Access-Object selbst fokussiert in der Literatur häufig nur auf die Kapselung der Datenbank-Zugriffe; die Betrachtung als „Model Controller“ erscheint aber sinnvoll.

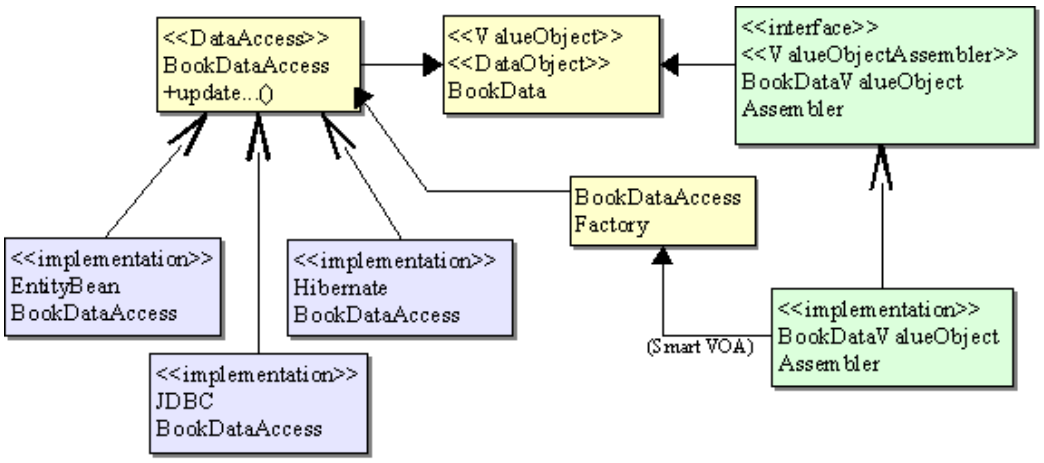
Das DAO ist ein fachlich getriebenes Objekt, ein komplexes Objekt-Aggregat, das einem fachlichen Zustand entspricht.

A.6.7.2 DAO-Varianten

Das Data-Access-Object liefert Data Objects, deren Aufbau im Wesentlichen aus dem Datenmodell abgeleitet werden kann. Data Objects repräsentieren damit in der Regel keine spezielle Client-Sicht auf die Daten[4].

```
Collection<BookData> listBooks()
```

Es ist Aufgabe eines Value Object Assemblers, aus den Data Objects die Daten in das benötigte Value Object zu kopieren.



[4] DataObjects dürfen nur dann bis zum Client durch gereicht werden, wenn der Use-Case-Controller dem Data Access Object entspricht.

Durch die Einführung des Assemblers ergeben sich einige Konsequenzen:

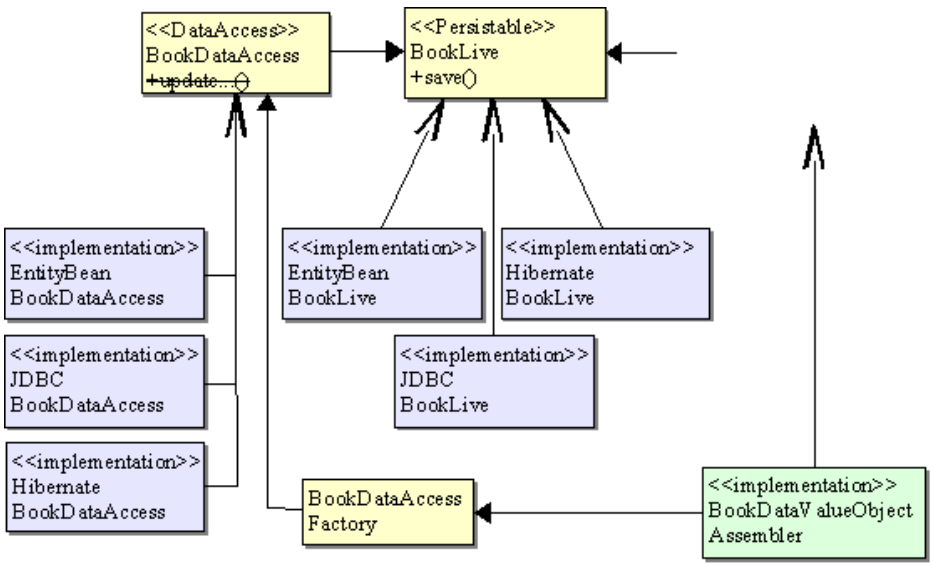
- Vorteil:

  Controller VOA sind unabhängig von der DataAccess-Technologie

  1. Nachteil:
     Es ist zusätzliche Logik für das zurück schreiben notwendig: Der Controller bekommt ein BookValue, der in BookData umgewandelt wird. In der DAO-Implementierung wird das BookData dann ausgelesen und gespeichert. Das heißt, die BookDataAccess-Schnittstelle muss auch Methoden wie update(BookData) oder updateBook(String isbn, String title, ...) implementieren.

Pro Use Case gibt es im Standardfall (nur ein Datenzugriff notwendig) somit mehrere unterschiedlich aufgebaute Value Objects, die alle unterschiedliche Client-Sichten auf ein Data Object liefern. Bei komplexeren Sequenzen werden zum Aufbau der Value Objects mehrere Data Objects benötigt, so dass dann auch der Begriff „Assembler“ seine wirkliche Berechtigung bekommt.

A.6.7.3 Data AccessObject mit Live-Objekten



- Vorteil: Es wird kein „überflüssiges“ Data-Object erzeugt
- keine spezielle Speicher-Logik (außerhalb der Live-Objekte) erforderlich

  1. Nachteil:
     „BookLive“ ist aufwendiger in der technischen Realisierung. Nicht alle Persistenz-Technologien ermöglichen das Übertragen von Live-Objekten über Schicht-Grenzen hinweg (derzeit nur Hibernate; demnächst EJB 3.0 mit der neuen Spezifikation für Entity-Beans).

A.6.7.4 Paket-Struktur mit Datenzugriffs-Klassen

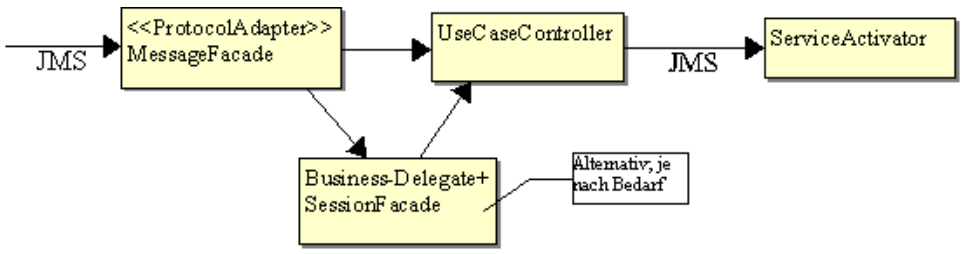
- de.integrata.data.<data-name>
  1. Data-Object
     - Live-Object
       - Data-Access-Object-Schnittstelle
  2. Factory für Data-Access-Objects
- de.integrata.data.<data-name>.implementation.mock
  - Mock-Data-Access-Object
  2. de.integrata.data.<data-name>.implementation.ejb
     - Entity-Bean
  3. de.integrata.data.<data-name>.implementation.jdbc
     - SQL-nahe Zugriffsklasse (JEE-Pattern „Fast Lane Reader“ bzw. „JDBC for Reading“)
- de.integrata.data.<data-name>.implementation.hibernate
  - Hibernate-Klassen
- de.integrata.data.<data-name>.implementation.ldap
  - de.integrata.data.<data-name>.implementation.connector
- de.integrata.data.<data-name>.implementation....

A.6.8 Die Message Facade

Die Message Facade ist ein „inoffizielles“ JEE-Pattern und beschreibt den Einsatz einer Message Driven Bean als Protokoll-Adapter.

A.6.9 Der Service Activator

Sehr ähnlich zur Message Facade ist das JEE Pattern „Service Activator“. Auch hier wird eine MEssageDrivenBean eingesetzt, nur liegt der Schwerpunkt hier auf der Möglichkeit, Logik asynchron ausführen zu lassen.



A.6.10 Aggregate Entity

Die Aggregate Entity als JEE Pattern beschreibt, dass ein Objektaggregat mit Hilfe von EntityBeans unter Verwendung von Container Managed Persistence und Container Managed Relations automatisch in das Datenmodell umgesetzt werden kann.

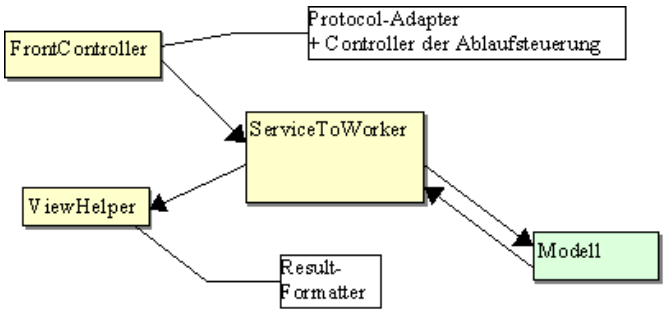
Wird keine Entity verwendet, so kann der Begriff „Aggregate Entity“ natürlich auch problemlos auf Data Access Objects angewendet werden. Man spricht dann von „Aggregate DAOs“.

A.6.11 Der Front Controller

Der Front Controller innerhalb der JEE wird als Servlet implementiert. Dieses JEE Pattern entspricht nicht nur vom Namen her dem „Front Controller“ des Pattern-Katalogs für verteilte Anwendungen.

A.6.12 Service to Worker

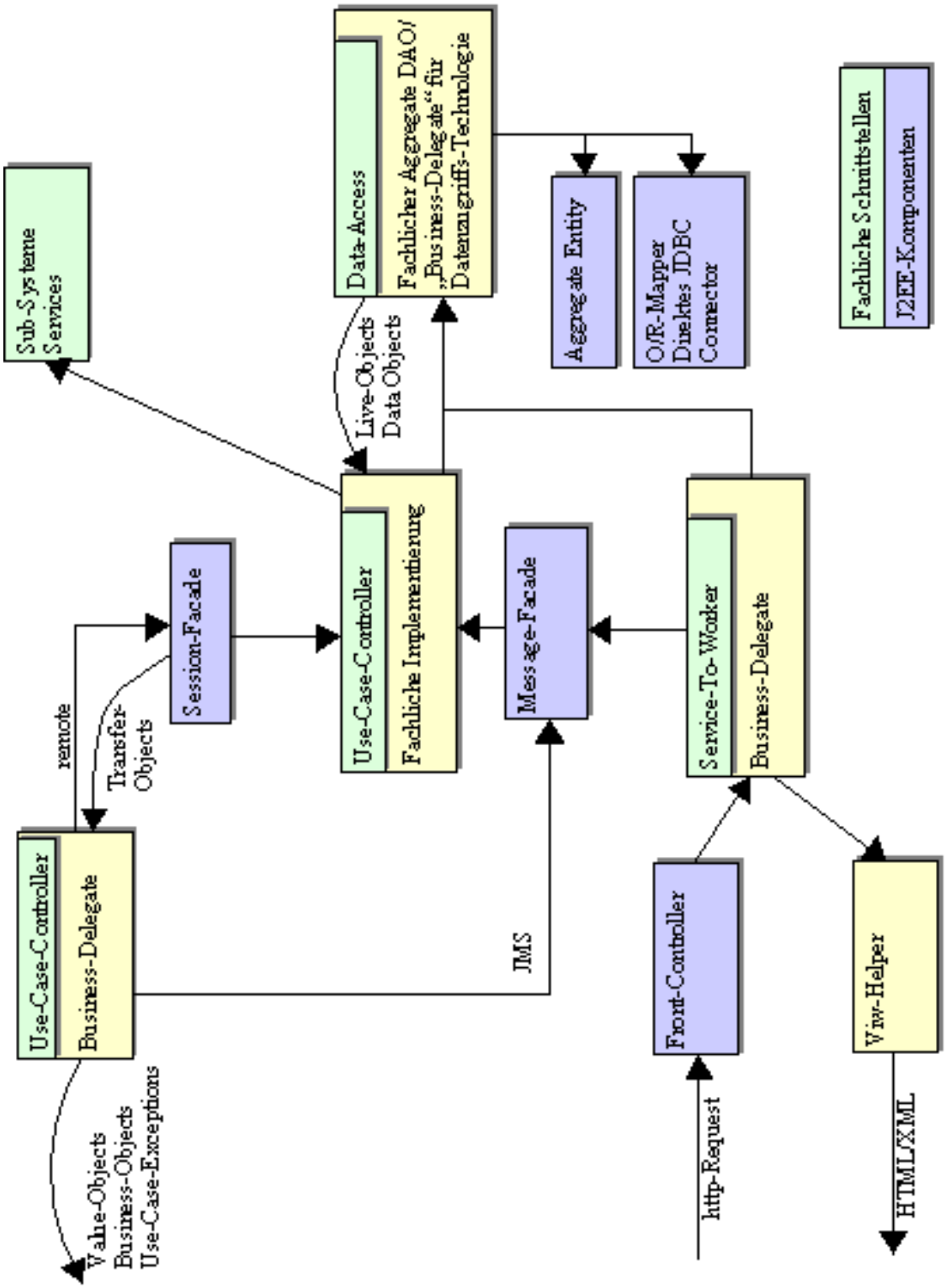
Nachdem der Front Controller die Ablaufsteuerung für die Web-Anwendung enthält, benötigt dieser noch eine Anbindung an den fachlichen Controller. Die Aufgabe, den „richtigen“ Controller zu finden und zu verwenden, übernimmt der Service to Worker:



A.6.13 Weitere Patterns

Der JEE-Patterns-Katalog enthält insbesondere in der Web schicht noch ein paar weitere Patterns und wird auch immer weiter ergänzt. Von den Ideen und Namen her orientieren sich diese Pattern jedoch sinnvoller weise an den bereits erwähnten Patterns für verteilte Anwendungen. Die JEE Patterns sind somit eigentlich nur Idiome.

A.7 Basis-Design unter Verwendung von JEE Patterns



Wesentliches Element dieses Basis-Designs ist die vollständige Plattform-unabhängigkeit für fachliche Implementierungen. Die Verwendung eines anderen Frameworks (z.B. .NET oder ein leichtgewichtiges Java-Framework wie Spring) erfordert nur den Austausch von technologischen Hilfsklassen, die durch den Model Driven-Ansatz größtenteils automatisch erzeugt werden können.