

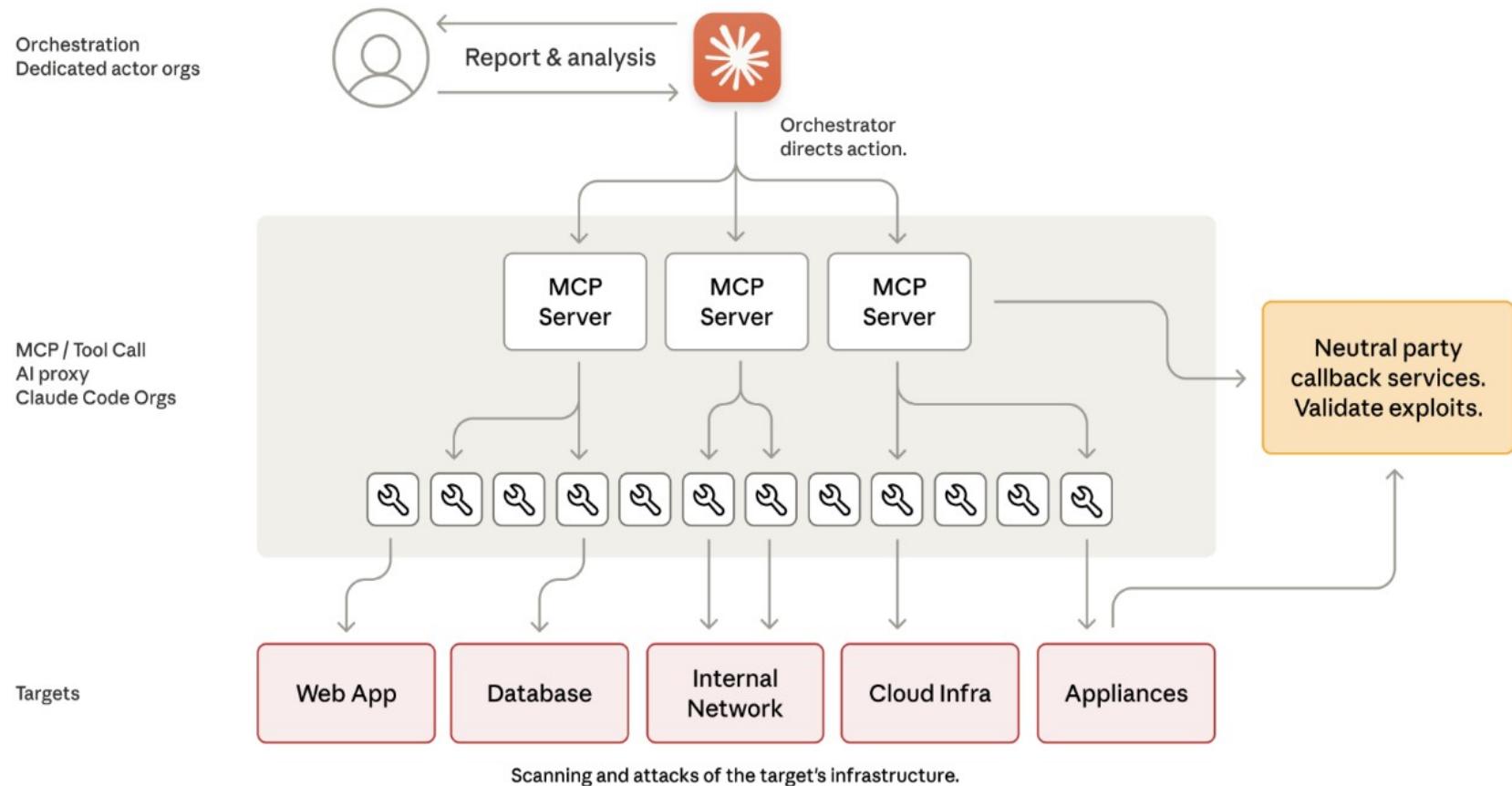
Advanced Software-Engineering

DHBW Stuttgart
24.11.2025

Janko Dietzsch

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe

Anthropic dokumentiert die erste KI-gestützte Spionage-Kampagne mit minimaler humaner und maximaler KI-Beteiligung in ihrer Plattform - GTG-1002



„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe

Der Anthropic Blog-Beitrag und Report vom 17.11.2025 dokumentiert eine hochprofessionelle KI-gestützte Spionage-Aktion der chinesischen, staatlich unterstützten Gruppe GTG-1002

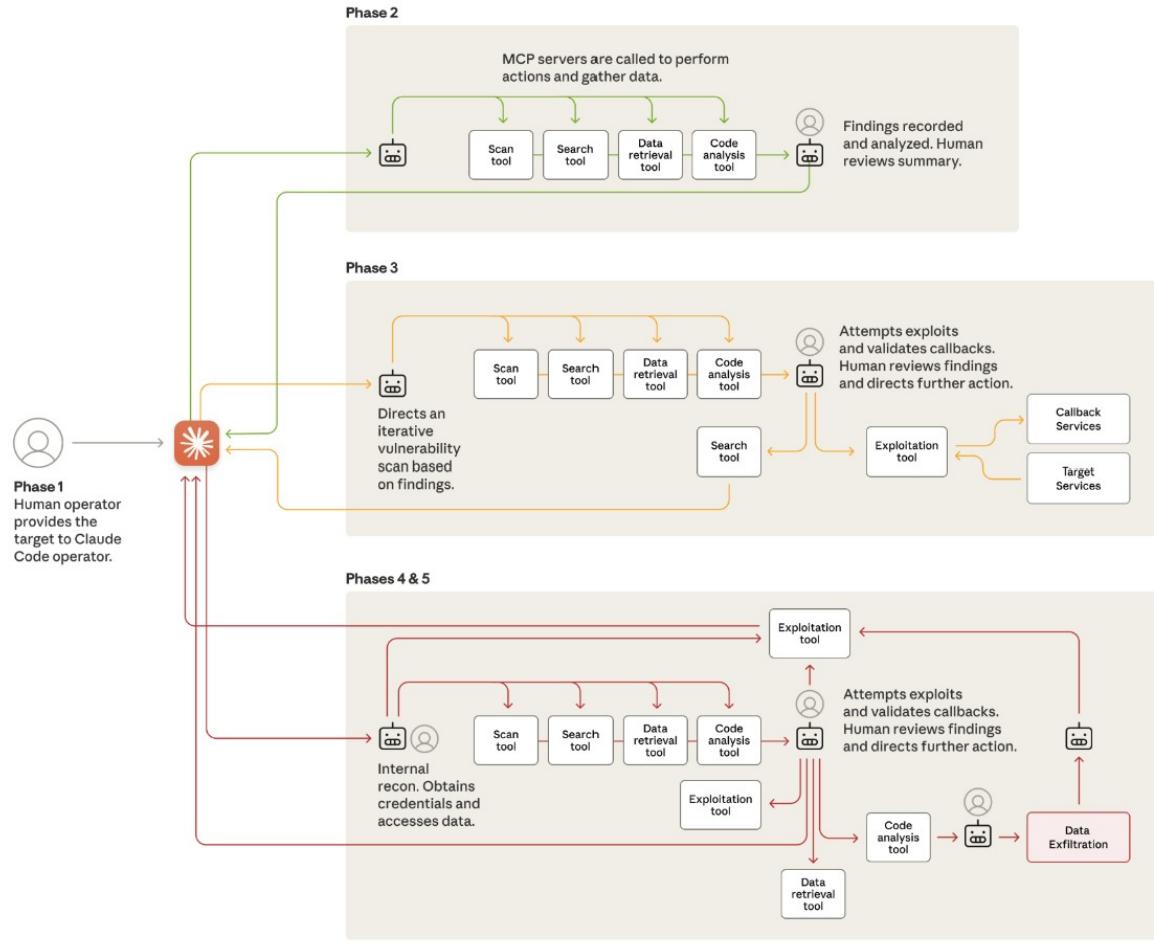
- Zentrale Merkmale:
 - Professionell koordinierte Operation mit mehreren simultanen, gezielten Eindringversuchen bei 30 Organisationen:
 - Große Technologieunternehmen
 - Finanzinstitute
 - Chemische Industrie
 - Regierungsbehörden verschiedener Länder
 - Mehrere erfolgreiche Kompromittierungen konnten in der Untersuchung bestätigt werden
 - KI führte ca. 80 – 90 % der taktischen Operationen aus und nur 10 – 20 % „menschliche Supervision“.

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe

Der Anthropic Blog-Beitrag und Report vom 17.11.2025 dokumentiert eine hochprofessionelle KI-gestützte Spionage-Aktion der chinesischen, staatlich unterstützten Gruppe GTG-1002

- Zentrale Merkmale:
 - Claude Code wurde für umfangreiche Angriffsszenarien entsprechend manipuliert
 - Menschen wurden vor allem bei Start und an kritischen „Entscheidungspunkten“ involviert, wie der Autorisierung der Durchführung von Exploits von Schwachstellen, der Exfiltration von Informationen, ... etc.
 - Claude unterteilte komplexe mehrstufige Angriffe in einzelne Aufgaben für autonome spezialisierte Agenten:
 - Schwachstellenanalyse
 - Validierung von Credentials
 - Datenextraktion
 - ...

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe



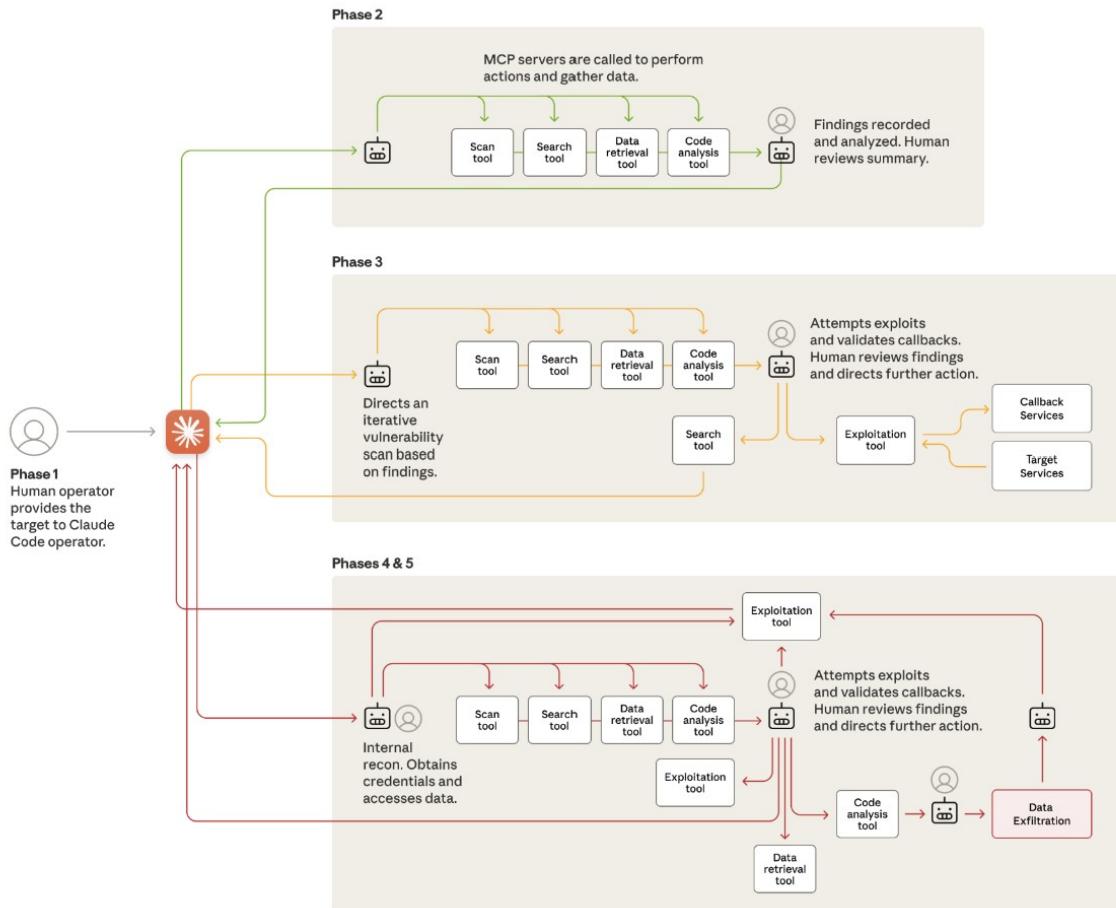
Phase 1 - Start:

- Mensch:
 - Zielauswahl und Start der Kampagne
- KI:
 - Anweisungen entgegengenommen

Phase 2 - Erkundung:

- KI – fast autonom:
 - Systematische Erfassung der Zielstruktur
 - Analyse der Authentifizierung
 - Schwachstellenanalyse
 - Netzwerkanalyse

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe



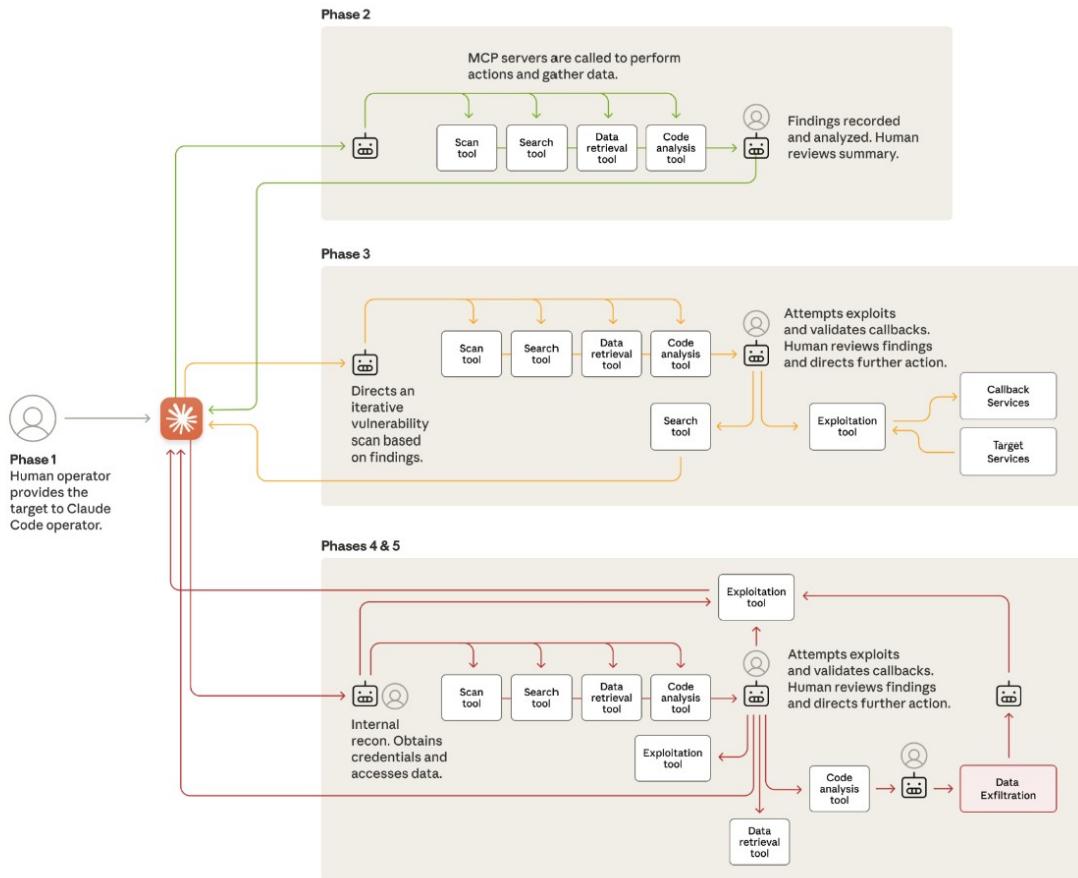
Phase 3 – Exploit-Vorbereitung:

- Mensch:
 - Autorisierung der Durchführung aktiver Exploits
- KI - autonom:
 - Generierung entsprechender Payloads und testen dieser

Phase 4 - Harvesting:

- Mensch:
 - Review und Autorisierung besonders sensibler Systeme
- KI - autonom:
 - Systematische Sammlung von Zugangsdaten

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe



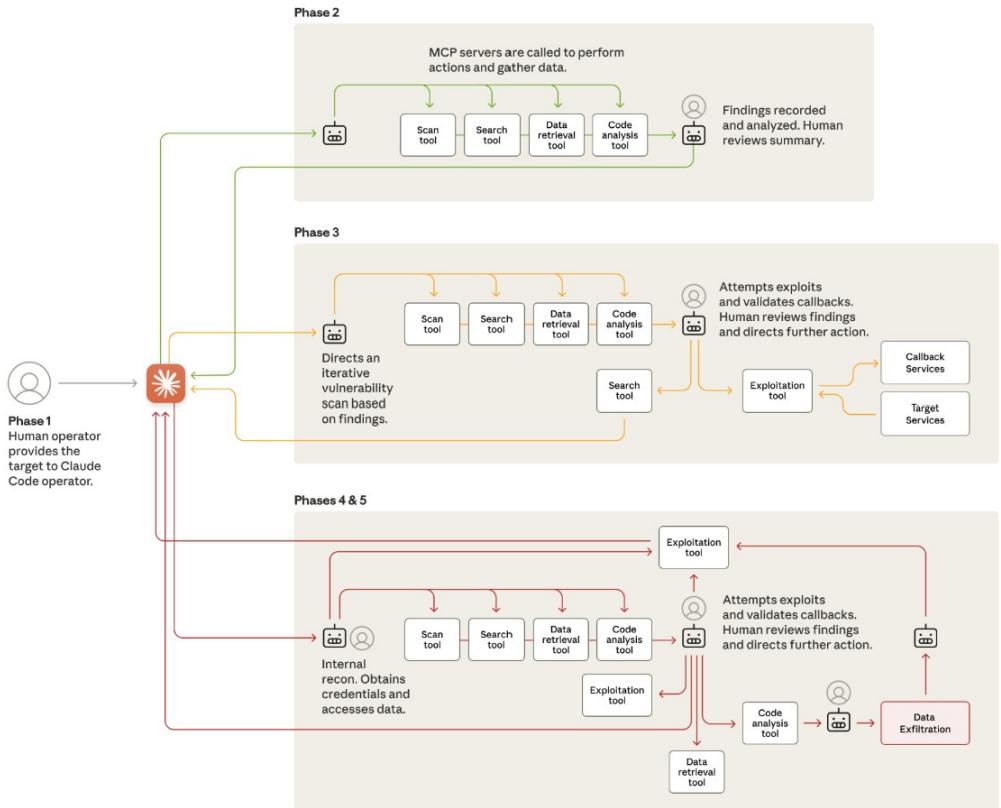
Phase 4 – Harvesting:

- **KI - autonom:**
 - Extraktion von Zertifikaten
 - Tests
 - Mapping von Privilegien

Phase 5 - Informationsextraktion:

- **KI - autonom:**
 - Abfragen von Datenbanken
 - Parsen und Katalogisieren gewonnener Informationen nach Relevanz
 - Erstellung von Backdoor-Accounts
 - Extraktion von Zertifikaten

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe



Phase 4 – Harvesting:

- KI - autonom:
 - Tests
 - Mapping von Privilegien

Phase 5 - Informationsextraktion:

- KI - autonom:
 - Abfragen von Datenbanken
 - Parsen und Katalogisieren gewonnener Informationen nach Relevanz
 - Erstellung von Backdoor-Accounts

Phase 6 – Dokumentation/Übergabe:

- KI - autonom:
 - Report-Generierung

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe

Technische Beobachtung:

- Überwiegende Verwendung von Open-Source-Tools
 - Verwendung von Standard-Tools für das Penetrationtesting – Netzwerkscanner, Password-Cracker, Exploitation-Tools, ... etc.
- Eigene Entwicklung – Orchestrierung mittels MCP
 - Spezialisierte Server für verschiedene Tools
- Probleme der Angreifer – KI-Halluzinationen:
 - Ungültige Credentials
 - Überbewertung „kritischer Informationen“, die eigentlich nicht kritisch, sondern öffentlich zugänglich sind
 - Notwendigkeit der Validierung aller Ergebnisse

„Aktuelles“: Eine neue Ära in der Cybersecurity – rein KI-gestützte Angriffe

Generelle Schlussfolgerungen für das veränderte Bedrohungsszenario:

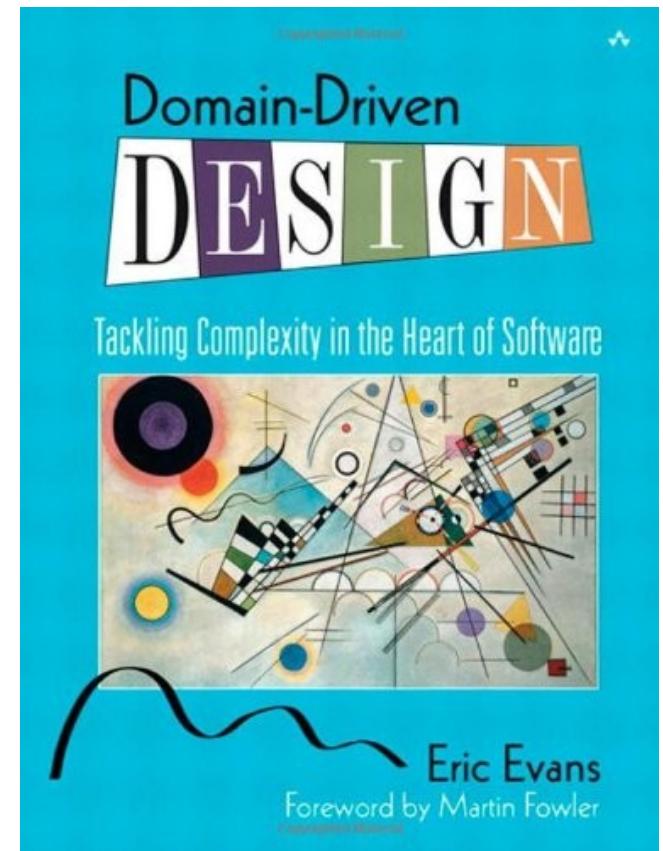
- Neue Cyber-Bedrohungsszenarien entstehen eher durch neue Möglichkeiten der Orchestrierung und Skalierung bereits bestehender Ressourcen, wie z.B. Standard-Tools und nicht aus innovativen eigenen, neuen Ansätzen für Angriffsszenarien
- KI-Halluzinationen beeinträchtigen nicht nur die normale Nutzung, sondern auch hier die Effektivität und Autonomie
- **Paradigmenwechsel in der Bedrohungslandschaft**
 - **Demokratisierung der Expertise** – weniger erfahrene und ausgestattete Gruppen können groß angelegte Angriffe durchführen, die früher State-Actors vorbehalten waren
 - **Skalierung** – Agentic-AI ermöglicht wenigen Personen, Aufwände großer Teams zu ersetzen
 - **Geschwindigkeit** – Analyse von Zielsystemen, Generierung von Exploits und Analyse großer Datenmengen können in wesentlich kürzerer Zeit durchgeführt werden

6. Domain-Driven-Design (*DDD*)

- Design- und Entwicklungsprozess sind eigentlich nicht ganz voneinander zu trennen
 - Entwicklungsprozess steckt gewisse Rahmenbedingungen für den Design-Prozess ab
 - Gewisse Methoden im DDD sind nur mit iterativer Entwicklung vereinbar
 - Ständige Beteiligung von Vertretern aus der Domäne
 - ...

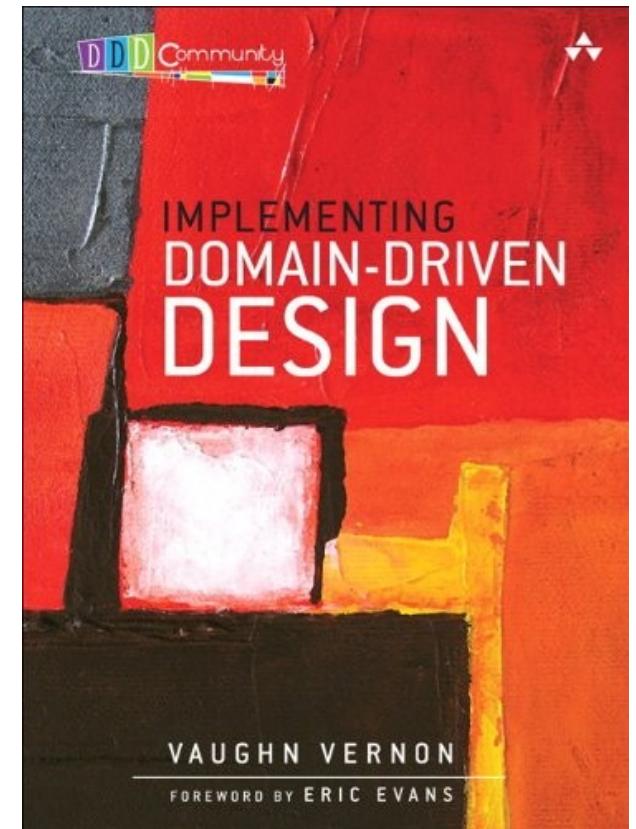
6. Domain-Driven-Design (DDD)

- „Domain-Driven Design – Tackling Complexity in the Heart of Software“
 - 2004 von Eric Evans herausgegeben
 - Das sogenannte „*The Big Blue Book*“
- Auch wieder in seiner Zeit zu betrachten:
 - Abkehr vom Waterfall, hinzu agileren Methoden
 - „Professionalisierung“ von OOP-Entwürfen und immer komplexere SW-Systeme in OOP



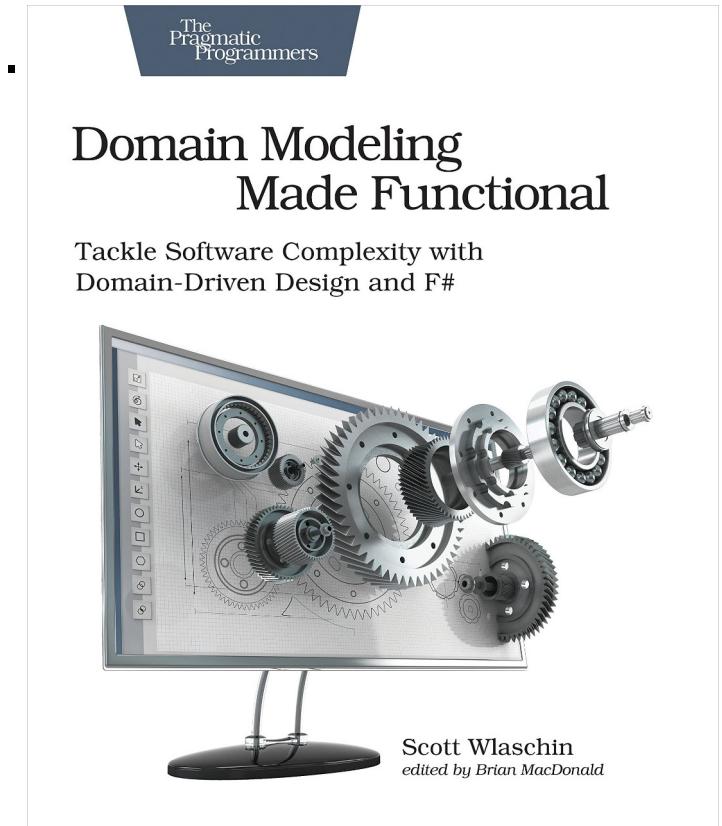
6. Domain-Driven-Design (DDD)

- Kondensierte Erfahrung beim Design von Software
- 2013 „Implementing Domain-Driven Design“ von Vaugh Vernon dazu
 - „*The Big Red Book*“
- Betonung der Domain-Modellierung, d.h. der fachlichen Seite der Software-Entwicklung
- Anfangs sehr auf OOP bezogen



6. Domain-Driven-Design (*DDD*)

- Sehr enger Bezug zu den im letzten Kapitel genannten Prinzipien, wie z.B. SOLID
- Oft im Kontext von OOP verstanden, aber eigentlich sehr viel breiter und unabhängig vom Programmier-Paradigma, wie z.B. in „Domain Modeling Made Functional“ sehr gut gezeigt
- Letzten sehr verstärkte Beachtung durch „*Microservice-Hype*“
- .. + CQRS und *Event-Sourcing*



6. Domain-Driven-Design (DDD)

Jimmy Nillsson (Autor des Buches „Applying Domain-Driven Design and Patterns“):

„Man braucht ein oder zwei Tage, um davon zu profitieren, aber ein ganzes Leben, um es zu meistern.

Ich gehe davon aus, dass ich von Domain-Driven Design noch in meiner ganzen Karriere profitieren werde. DDD ist genau das Gegenteil von ‚und wieder ein neues Framework‘, von Dingen also, die plötzlich auf der Bildfläche erscheinen, um nach einem oder zwei Jahren genauso schnell wieder zu verschwinden.“

6.1 Grundlagen & Definitionen

- Das grundlegende Ziel von DDD
 - Software soll Aspekte der realen Welt nachbilden
 - Wir bauen *Design Modelle* um zu verstehen, was wir bauen und wie wir es bauen
 - Die Symmetrie zwischen Software, Design-Modell und Realität erlaubt die Anpassung an Änderungen in der Realität
- Kontinuierliches Lernen
- „*Knowledge Rich Designs*“

6.1 Grundlagen & Definitionen

- **Domain** (Domäne):
 - Bereich von Wissen, Einfluss, Aktivität – das Fachgebiet auf das die Software angewendet werden soll
- **Model** (Modell):
 - System von Abstraktionen, um ausgewählte Aspekte einer Domäne zu beschreiben
 - Verwendbar für Lösungsansätze in der Domäne

6.1 Grundlagen & Definitionen

- **Ubiquitous Language** (Allgegenwärtige Sprache):
 - Sprache die um das Domänen-Modell herum „gebaut wird“
 - Wird von allen Teammitgliedern eines *Bounded Context* verwendet und verstanden
- **Context** (Kontext):
 - Bereich in dem eine Aussage gilt und deren Bedeutung bestimmt
 - Aussagen über ein Modell können nur im entsprechenden *Context* verstanden werden

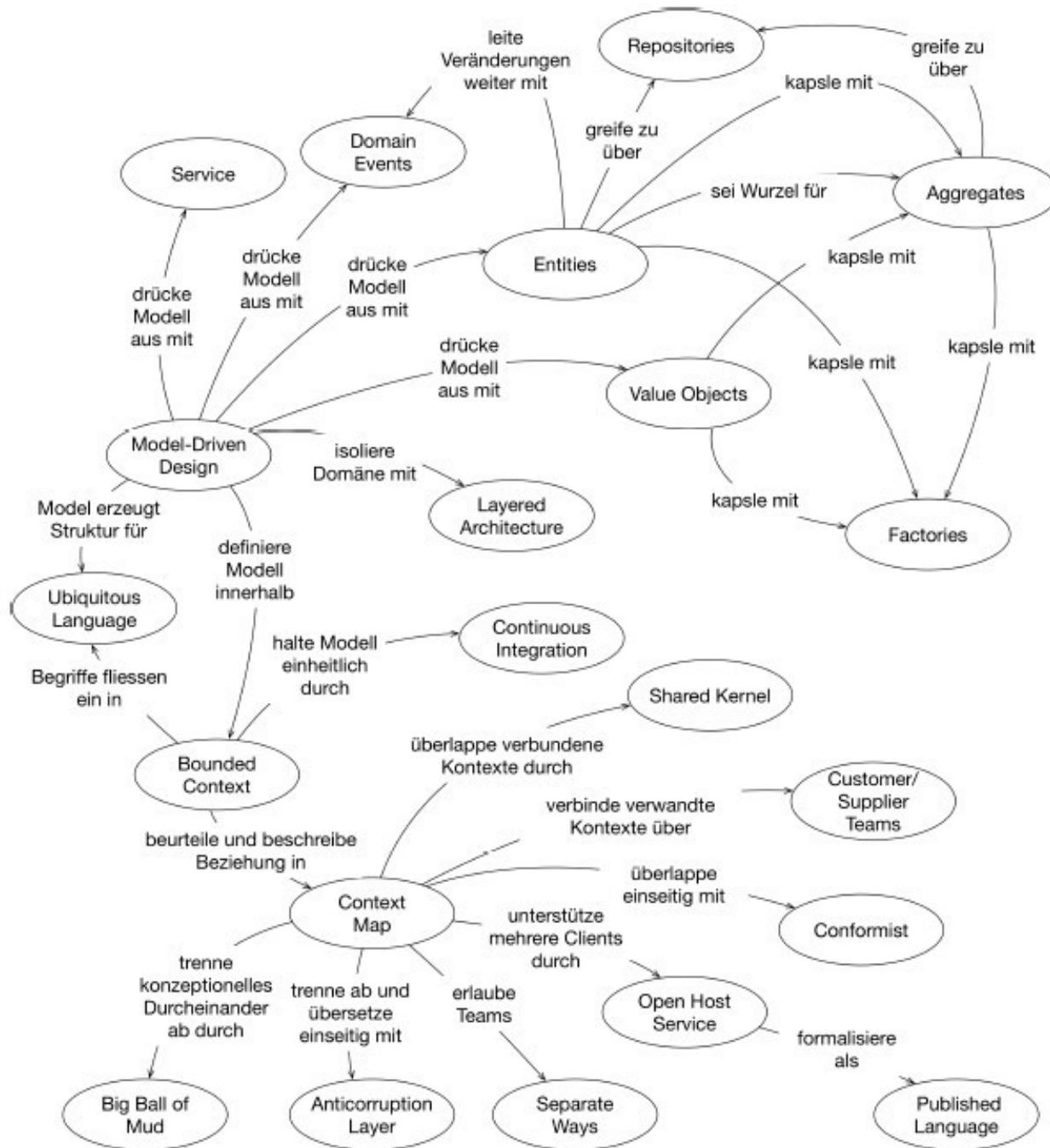
6.1 Grundlagen & Definitionen

- **Bounded Context** (*Begrenzter Kontext*):
 - Beschriebene Grenze innerhalb der ein bestimmtes Modell definiert und anwendbar ist
 - Bsp: Subsystem oder die Arbeit eines bestimmten Teams

6.1 Grundlagen & Definitionen

Ansatz für die Entwicklung komplexer Software mit folgenden Kernpunkten:

- Konzentration auf die *Core Domain*
- *Modelle* in einer kreativen Zusammenarbeit von Domänen- und Software-Praktikern entwickelt werden
- Eine *Ubiquitous Language* in einem expliziten *Bounded Context* gesprochen wird

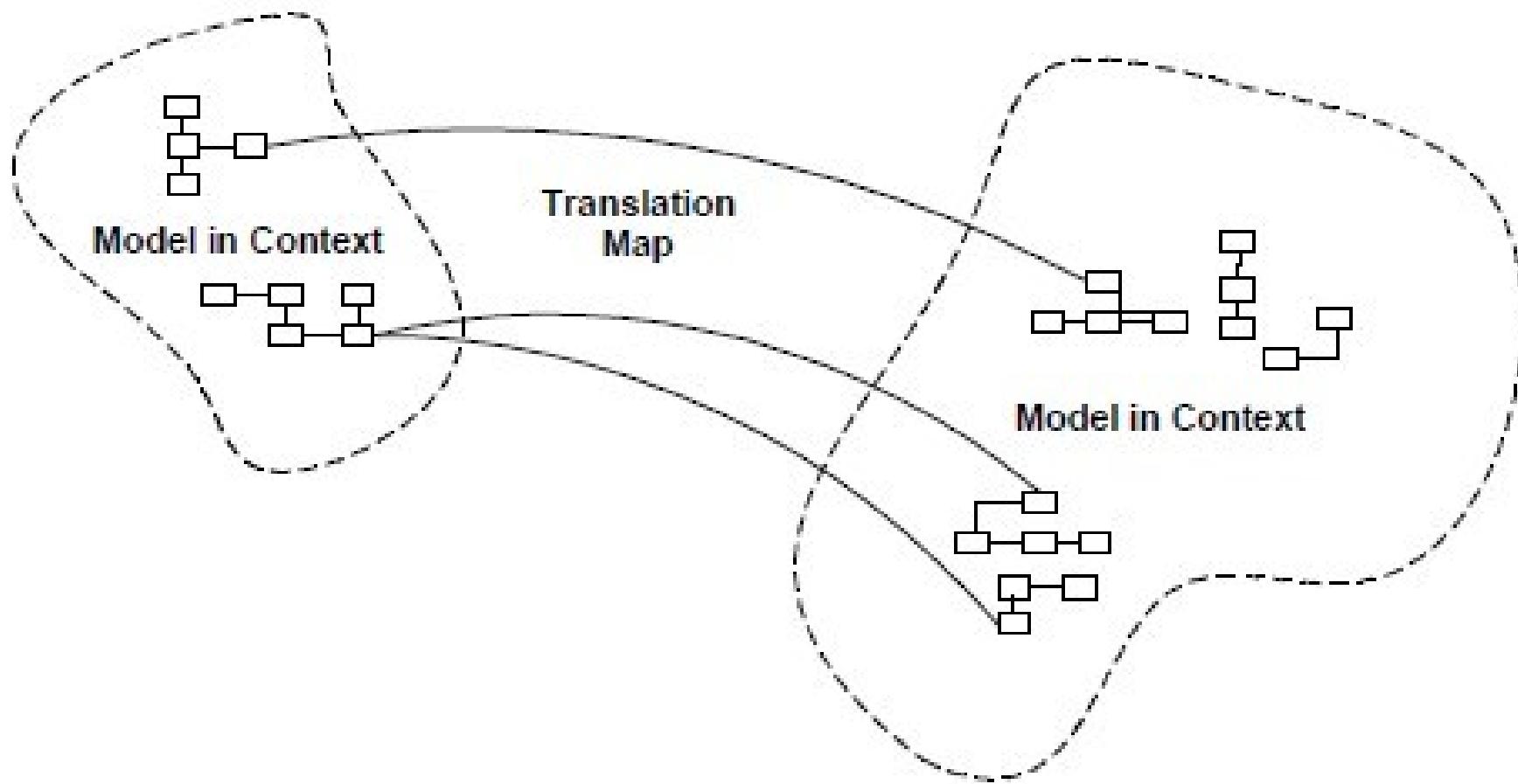


6.2 Modell etablieren

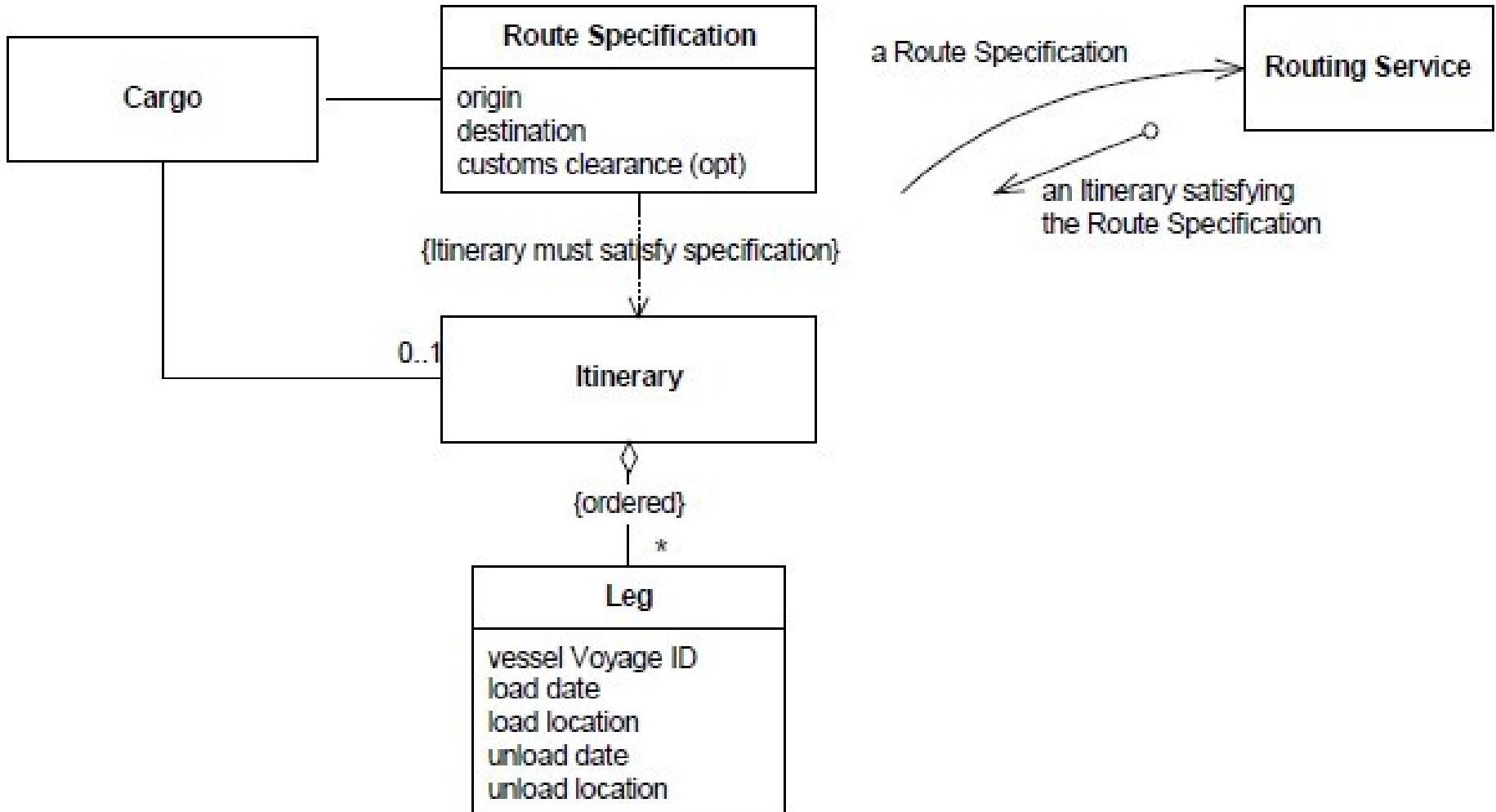
Bounded Context

- In großen Softwareprojekten sind mehrere Modelle unvermeidlich → verschiedene Subsysteme, verschiedene Nutzer, ... etc.
- Explizit den Kontext für ein Modell definieren
- Explizite Grenzen für Teamorganisation, Verwendung von Codeteilen innerhalb der Anwendung
- *Continuous Integration* anwenden, um Modellkonzepte und -begriffe streng konsistent zu halten

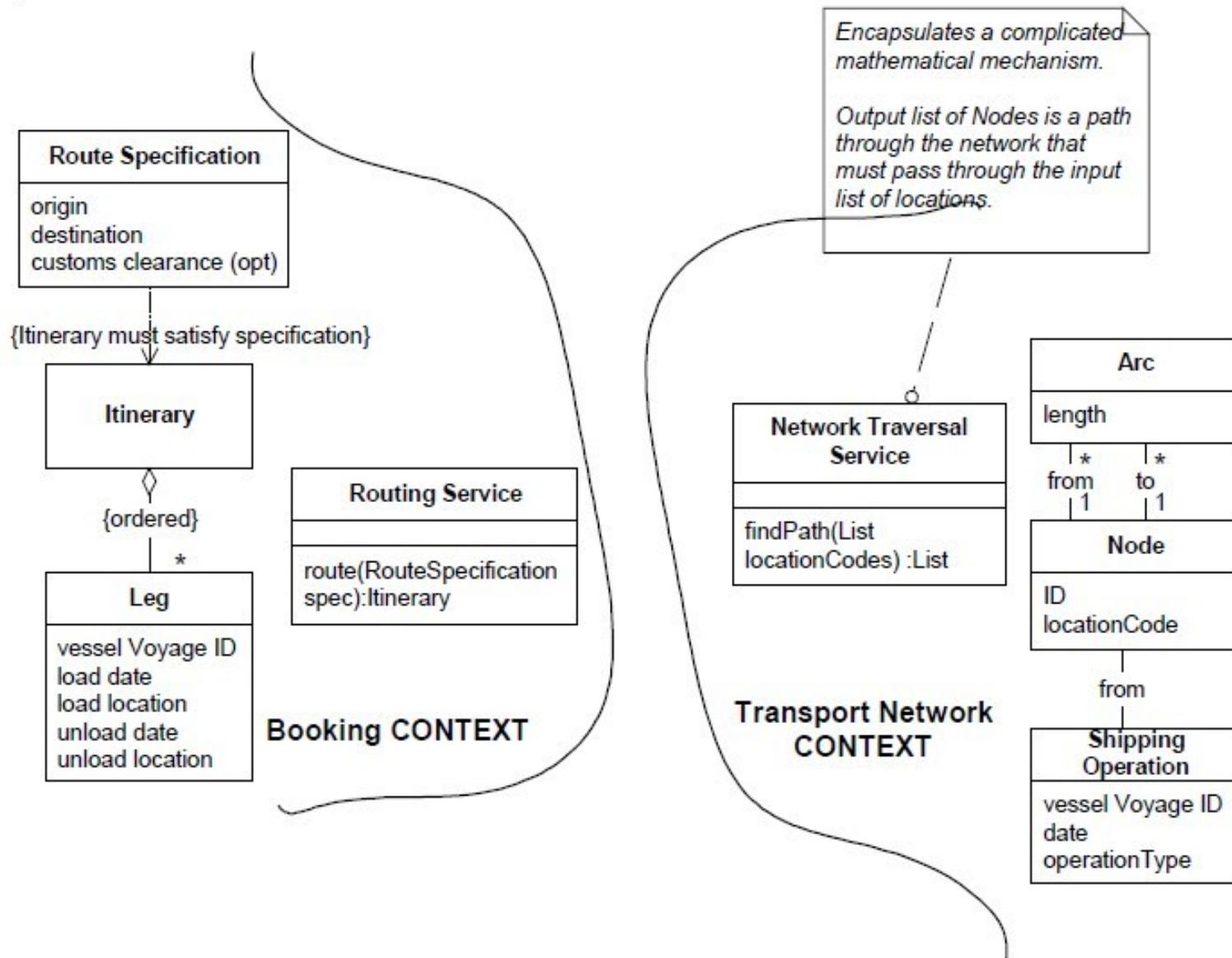
6.2 Modell etablieren



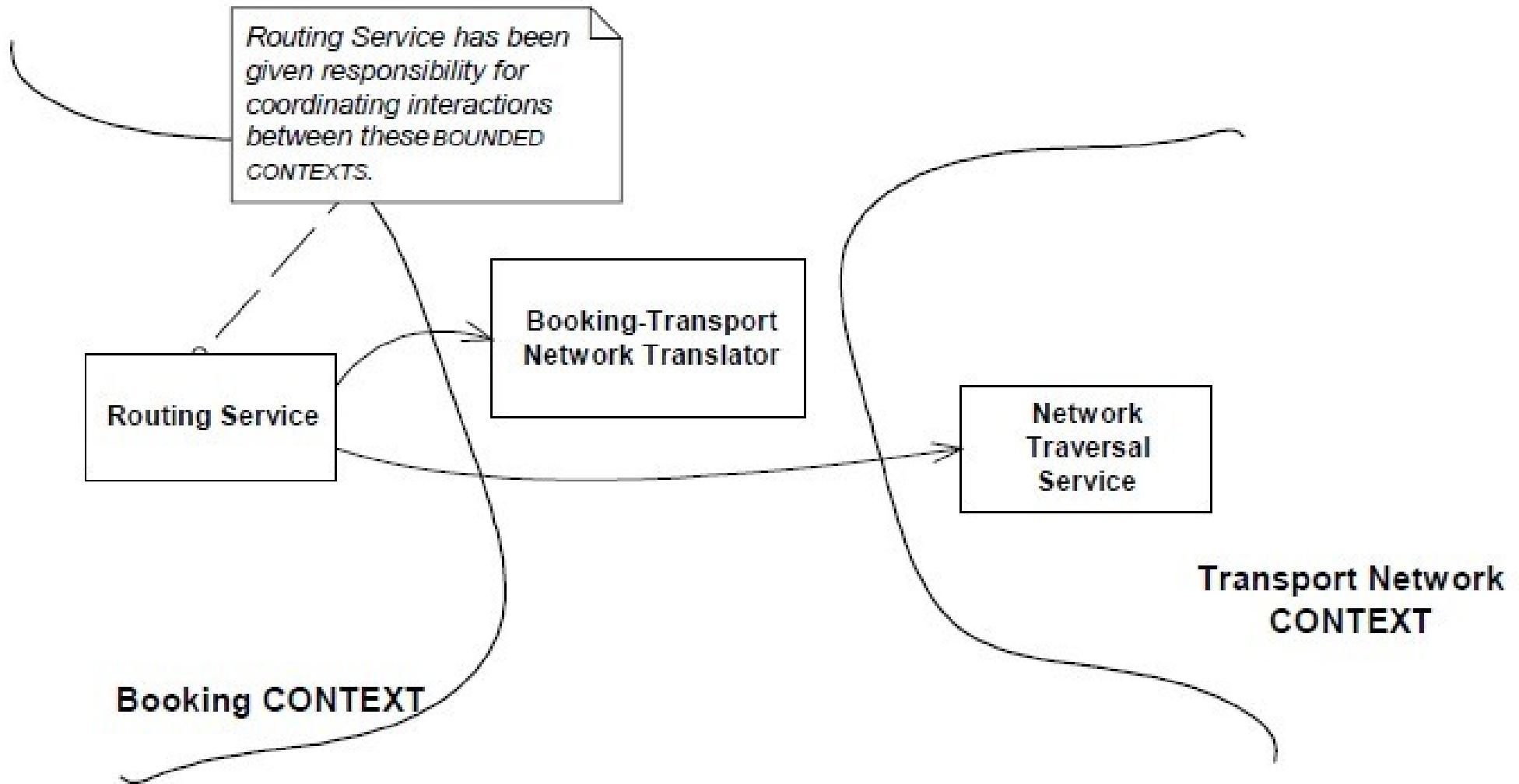
6.2 Modell etablieren



6.2 Modell etablieren



6.2 Modell etablieren



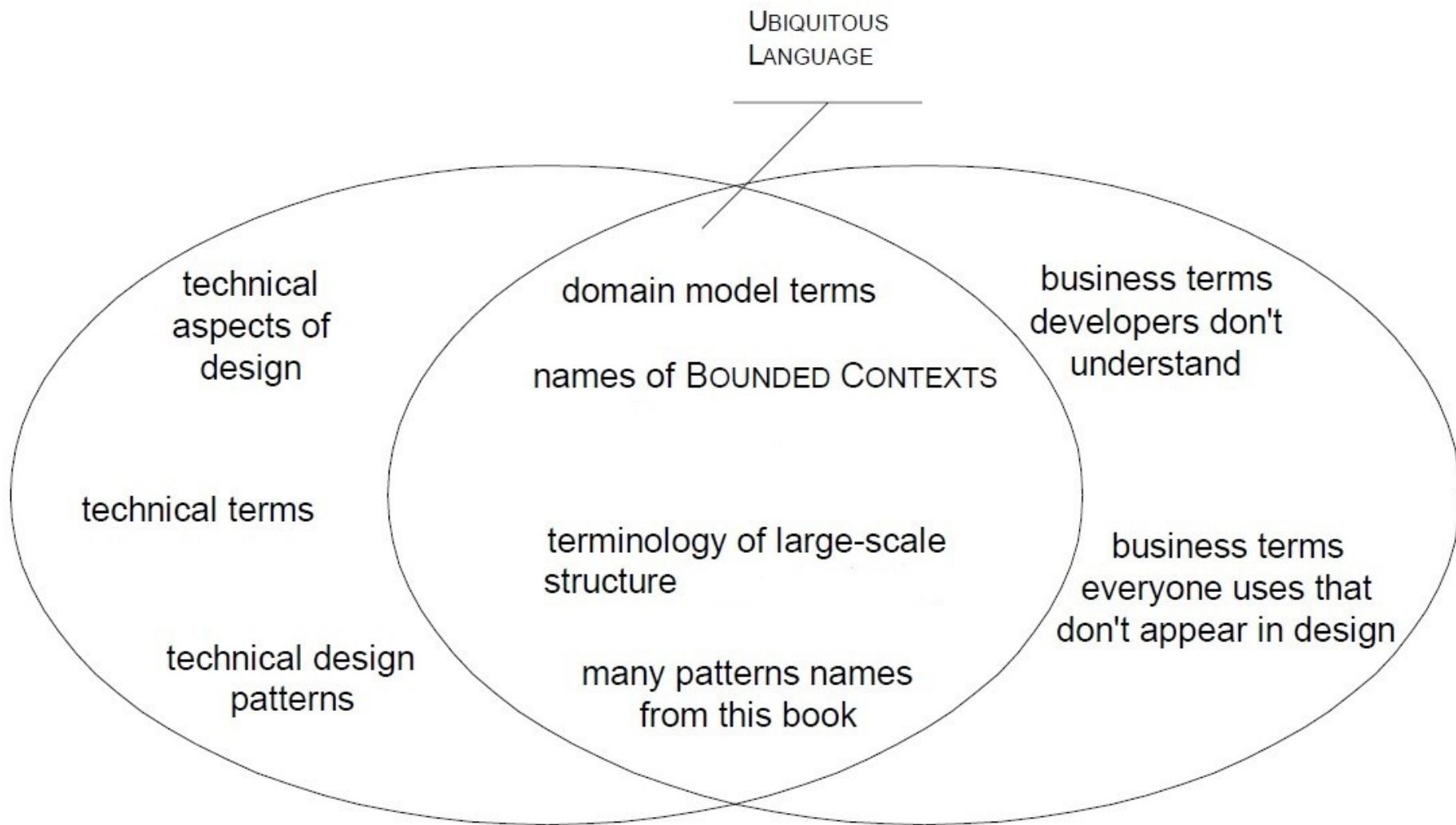
6.2 Modell etablieren

Ubiquitous Language

- Modell als Basis der Sprache
- Team muß diese Sprache in der gesamten Kommunikation innerhalb des Teams (Domänen- und SW-Experten) und im Code verwenden
- In einem *Bounded Context* muß die gleiche Sprache in Diagrammen, Code, Schreiben und sprechen verwendet werden
- Änderung der Sprache ist eine Änderung des Modells

6.2 Modell etablieren

UBIQUITOUS LANGUAGE is Cultivated in the Intersection of Jargons



6.2 Modell etablieren

Ubiquitous Language

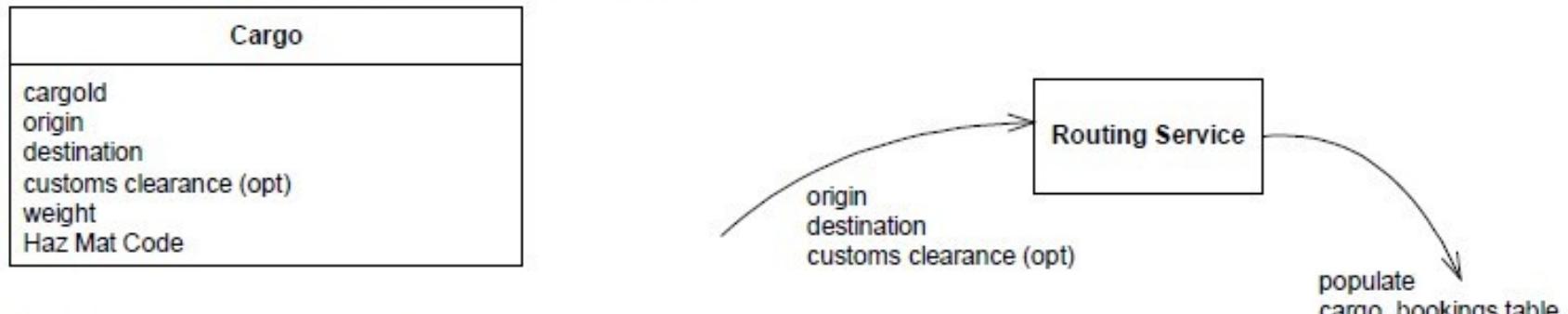
- Experimentieren mit alternativen Ausdrücken, um alternative Modelle auszuschließen bzw. auszuprobieren
 - Notfalls Überarbeitung des Codes → Umbenennen von Klassen, Methoden, Modulen ... etc. um dem neuen Modell zu entsprechen
 - Begriffliche Verwirrungen klären und Bedeutung definieren

6.2 Modell etablieren

Example Dialogs: Working out a Commercial Shipping Cargo Router

The differences between the two dialogs that follow are subtle, but important. Watch for how much they talk about what the software means to the business versus how it works technically. Are they speaking the same language? Is that language rich enough to carry the discussion of what the application must do?

Scenario 1: Minimal abstraction of the domain



Database table: cargo_bookings

Cargo_ID	Transport	Load	Unload

6.2 Modell etablieren

USER: So when we change the customs clearance point, we need to redo the whole routing plan.

DEVELOPER: Right. We'll delete all the rows in the shipment table with that cargo id, then we'll pass the origin, destination and the new customs clearance point into the **Routing Service** and it will repopulate the table. We'll have to have a Boolean in the **Cargo** so we'll know there is data in the shipment table.

USER: Delete the rows? Ok, whatever. Anyway, if we didn't have a customs clearance point at all before, we'll have to do the same thing.

DEVELOPER: Sure, any time you change the origin, destination or customs clearance point (or enter one for the first time) we'll check to see if we have shipment data and then we'll delete it and then let the **Routing Service** regenerate it.

USER: Of course, if the old customs clearance just happened to be the right one, we wouldn't want to do that.

DEVELOPER: Oh, no problem. It is easier to just make the **Routing Service** redo the loads and unloads every time.

USER: Yes, but it is extra work for us to make all the supporting plans for a new itinerary, so we don't want to reroute unless the change necessitates it.

DEVELOPER: Ugh. Well, then, if you are entering a customs clearance point for the first time, we'll have to query the table to find the old derived customs clearance point, and then compare it to the new one. Then we'll know if we need to redo it.

USER: You won't have to worry about this on origin or destination, since the itinerary would always change then.

DEVELOPER: Good. We won't.

6.2 Modell etablieren

USER: So when we change the customs clearance point, we need to redo the whole routing plan.

DEVELOPER: Right. When you change any of the attributes in the **Route Specification**, we'll delete the old **Itinerary** and ask the **Routing Service** to generate a new one based on the new **Route Specification**.

USER: If we hadn't specified a customs clearance point at all before, we'll have to do the same time.

DEVELOPER: Sure, any time you change anything in the **Route Spec**, we'll regenerate the **Itinerary**. That includes entering something for the first time.

USER: Of course, if the old customs clearance just happened to be the right one, we wouldn't want to do that.

DEVELOPER: Oh, no problem. It is easier to just make the **Routing Service** redo the **Itinerary** every time.

USER: Yes, but it is extra work for us to make all the supporting plans for a new **Itinerary**, so we don't want to reroute unless the change necessitates it.

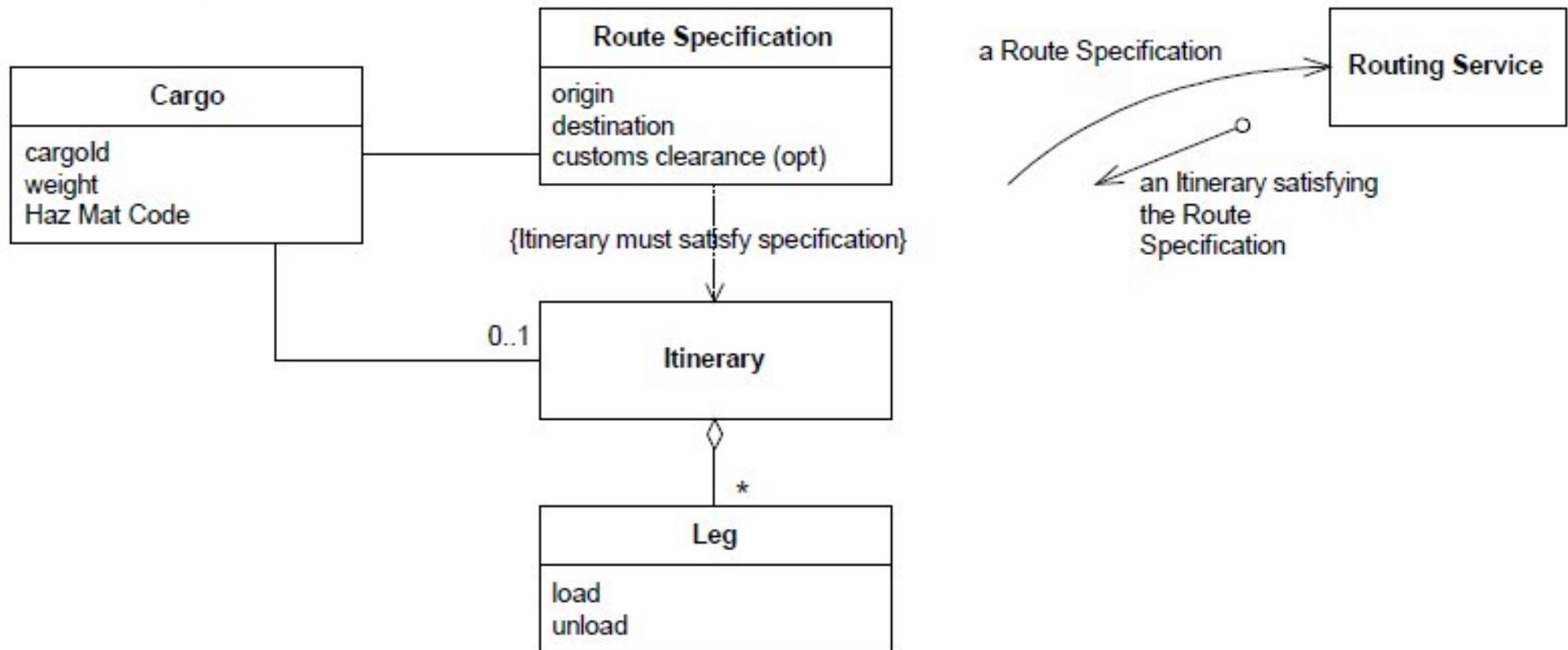
DEVELOPER: Oh. Then we'll have to add some functionality to the **Route Specification**. Then, whenever you change anything in the **Spec**, we'll see if the **Itinerary** still satisfies the **Specification**. If it doesn't, we'll have the **Routing Service** regenerate the **Itinerary**.

USER: You won't have to worry about this on origin or destination, since the **Itinerary** would always change then.

DEVELOPER: Fine, but it will be simpler for us to just do the comparison every time. The **Itinerary** will only be generated when the **Route Specification** is no longer satisfied.

6.2 Modell etablieren

Scenario 2: Domain model enriched to support discussion



6.2 Modell etablieren

- Welcher Dialog bringt die Ideen des Domänen-Experten mehr zum Ausdruck?
- ***Itinerary*** wird vom Domänen-Experten in beiden Dialogen erwähnt, aber nur im zweiten ist es auch modelliert
- ***Route Specification*** taucht im zweiten Dialog und Modell explizit auf – im ersten in Form von Attributen und Tabellenwerten
- Der erste Dialog ist überladen mit technischen Details

6.2 Modell etablieren

Continuous Integration

- Sobald ein *Bounded Context* definiert ist, muss er intakt gehalten werden
- Tendenz zur Fragmentierung entgegenwirken (je mehr Teammitglieder, desto größer die Tendenz)
- Code- und alle Implementierungsartefakte regelmäßig zusammenführen → automatisierte Tests, um Fragmentierung zu erkennen
- *Ubiquitous Language* um eine gemeinsame Sicht auf das Modell zu gewährleisten

6.2 Modell etablieren

Model-driven Design (modellgetriebender Entwurf)

- Enge Verknüpfung des Codes mit dem zugrunde liegenden **Modell** macht dieses relevant
- Einen Teil des SW-Systems so entwerfen, dass es das Domänenmodell möglichst exakt widerspiegelt → Umsetzung wird so offensichtlich, Code <=> Modell
- Versuch das Modell möglichst so einfach und genau wie möglich zu implementieren → Anspruch beiden Aspekten gerecht zu werden und dabei eine durchgängige **Ubiquitous Language** zu unterstützen

6.2 Modell etablieren

Hands-on Modelers (praktizierende Modellierer)

- Jede „technische“ Person, die am *Modell* mitarbeitet muss auch am Code mitarbeiten, unabhängig davon welche Rolle sie primär im Projekt hat
- Jeder Entwickler muss an der Diskussion über das Modell beteiligt sein und Kontakt zu Domänenexperten haben
- Sind Modellierer vom Entwicklungsprozeß entkoppelt, fehlt bzw. verliert er das Gefühl für die Restriktionen der Implementierung.

6.2 Modell etablieren

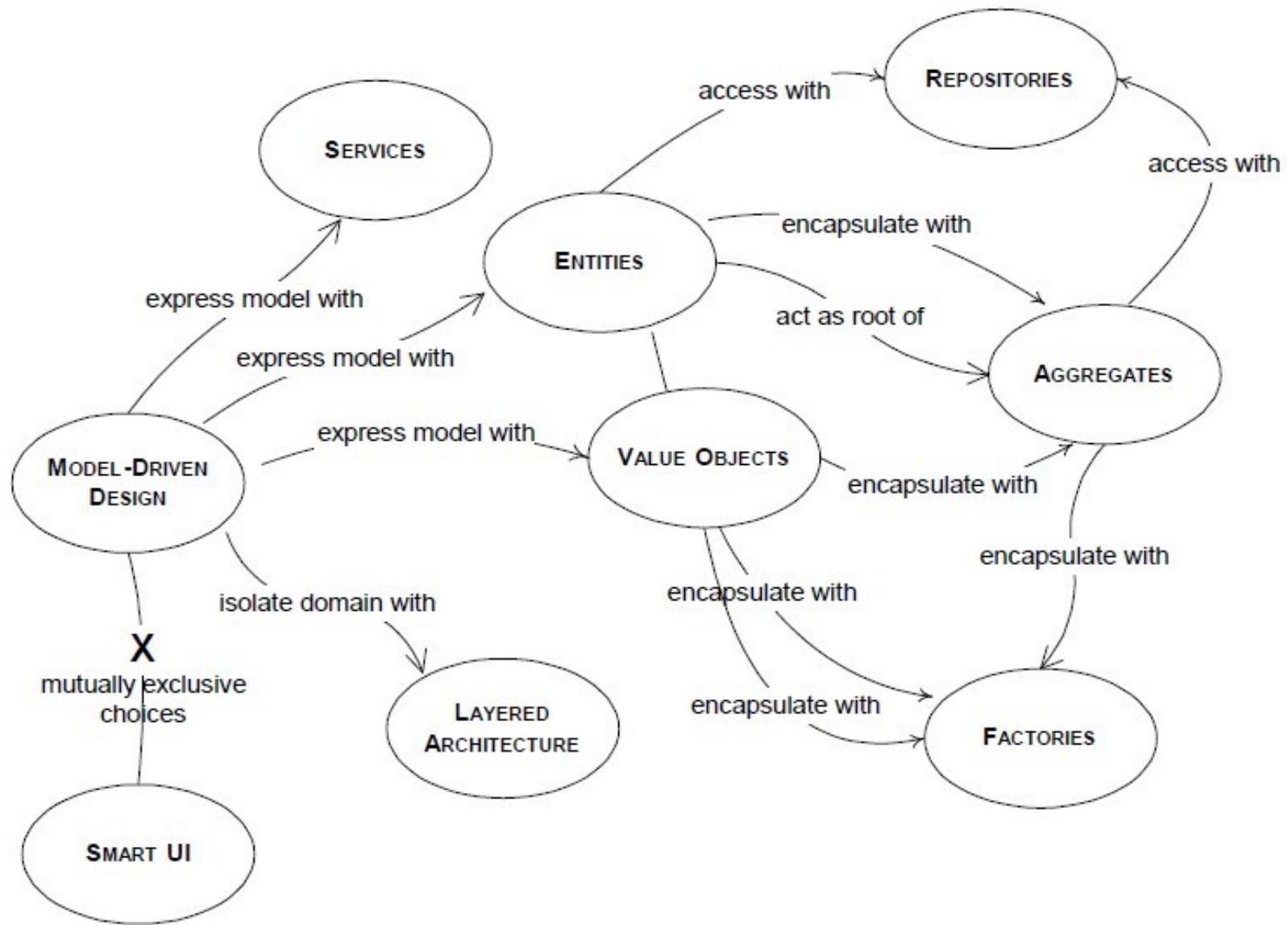
Refactoring Towards Deeper Insight *(Refactoring zum tieferen Verständnis)*

- Traditionell wird Refactoring als Code-Transformation aus technischen Gründen aufgefasst (Beseitigung von Technical Debt) hier aber:
- Refactoring für ein besseres und tieferes Verständnis der *Domäne* und damit einer Verfeinerung und Verbesserung des *Modells* ausgedrückt in Code

6.3 Modellgetriebener Entwurf

- Kern von Best-Practices als Basis von Bausteinen
- Überbrückung der Lücke zwischen Modell und funktionierender Software
- Nutzung dieser Standard-Patterns bringt Ordnung ins Design und erleichtert das Verständnis bei der Beschreibung
- Nutzung dieser Standard-Patterns als Fundament für eine gemeinsame Sprache
- „Klein-skalige“ Plattform für die späteren Design-Prinzipien im Großen

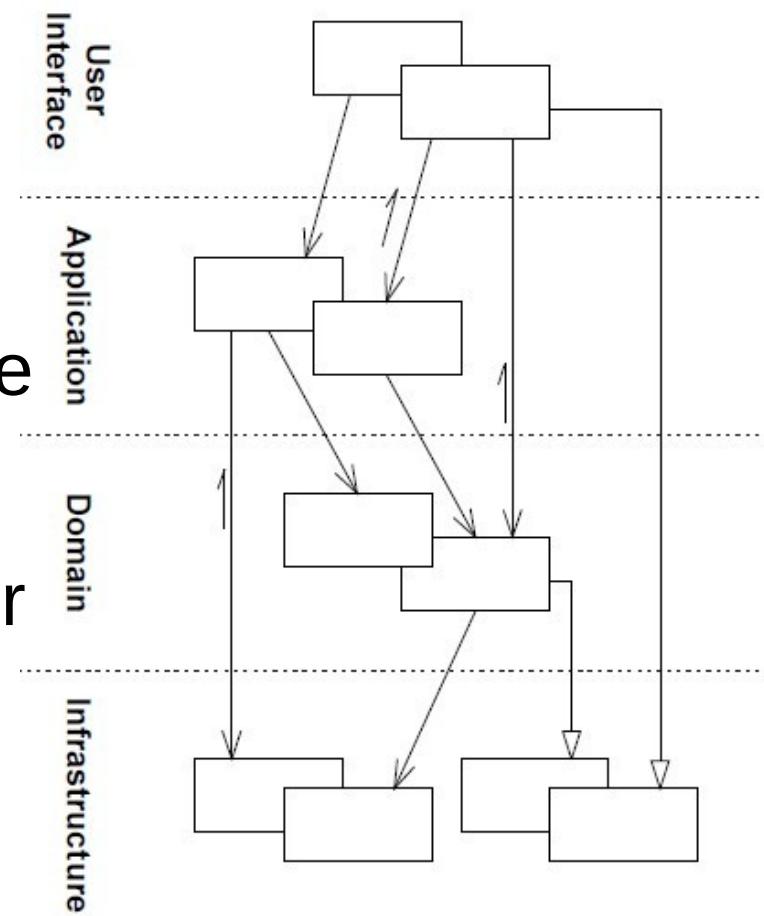
6.3 Modellgetriebener Entwurf



6.3 Modellgetriebener Entwurf

Layered Architecture (Geschichtete Architektur)

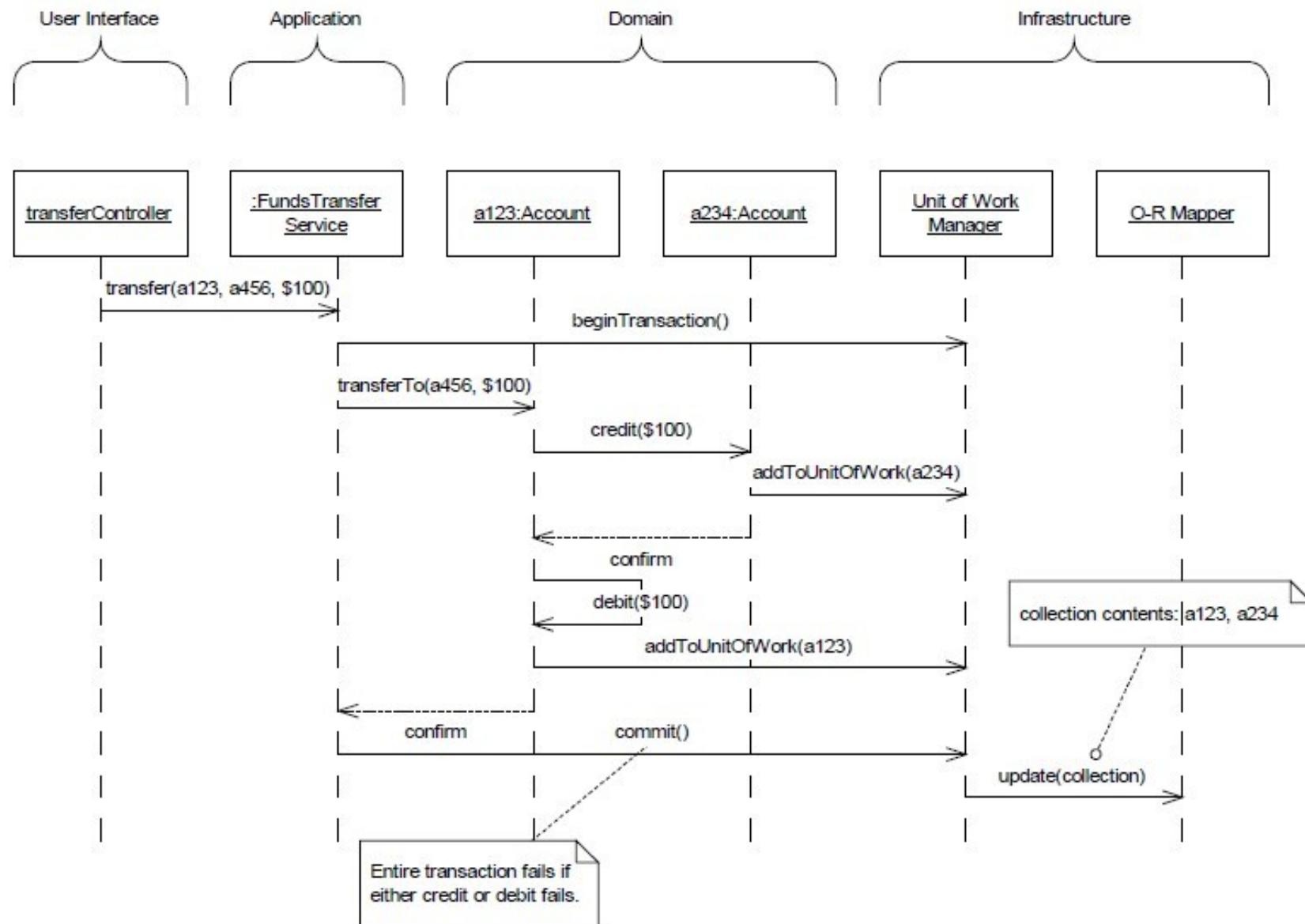
- Schichtweiser Aufbau isoliert die Umsetzung des Domänenmodells und der Geschäftslogik
- Entfernen von Abhängigkeiten zu Infrastruktur- und GUI-Code, sowie zu Anwendungslogik, die keine Geschäftslogik ist
- Kohärentes Design innerhalb jeder Schicht, die nur von den darunter liegenden abhängt
- Domänen-Objekte setzen nur das Domänenmodell um
- *Hexagonale Architektur*



6.3 Modellgetriebener Entwurf

User Interface (aka Presentation Layer)	Responsible for showing information to the user and interpreting the user's commands. The external actor might sometimes be another computer system rather than a human user.
Application Layer	Defines the jobs the software is supposed to do and directs the expressive domain objects to work out problems. The tasks this layer is responsible for are meaningful to the business or necessary for interaction with the application layers of other systems. This layer is kept thin. It does not contain business rules or knowledge, but only coordinates tasks and delegates work to collaborations of domain objects in the next layer down. It does not have state reflecting the business situation, but it can have state that reflects the progress of a task for the user or the program.
Domain Layer (aka Model Layer)	Responsible for representing concepts of the business, information about the business situation, and business rules. State that reflects the business situation is controlled and used here, even though the technical details of storing it are delegated to the infrastructure. <i>This layer is the heart of business software.</i>
Infrastructure Layer	Provide generic technical capabilities that support the higher layers: message sending for the application, persistence for the domain, drawing widgets for the UI, etc. The infrastructure layer may also support the pattern of interactions between the four layers through an architectural framework.

6.3 Modellgetriebener Entwurf



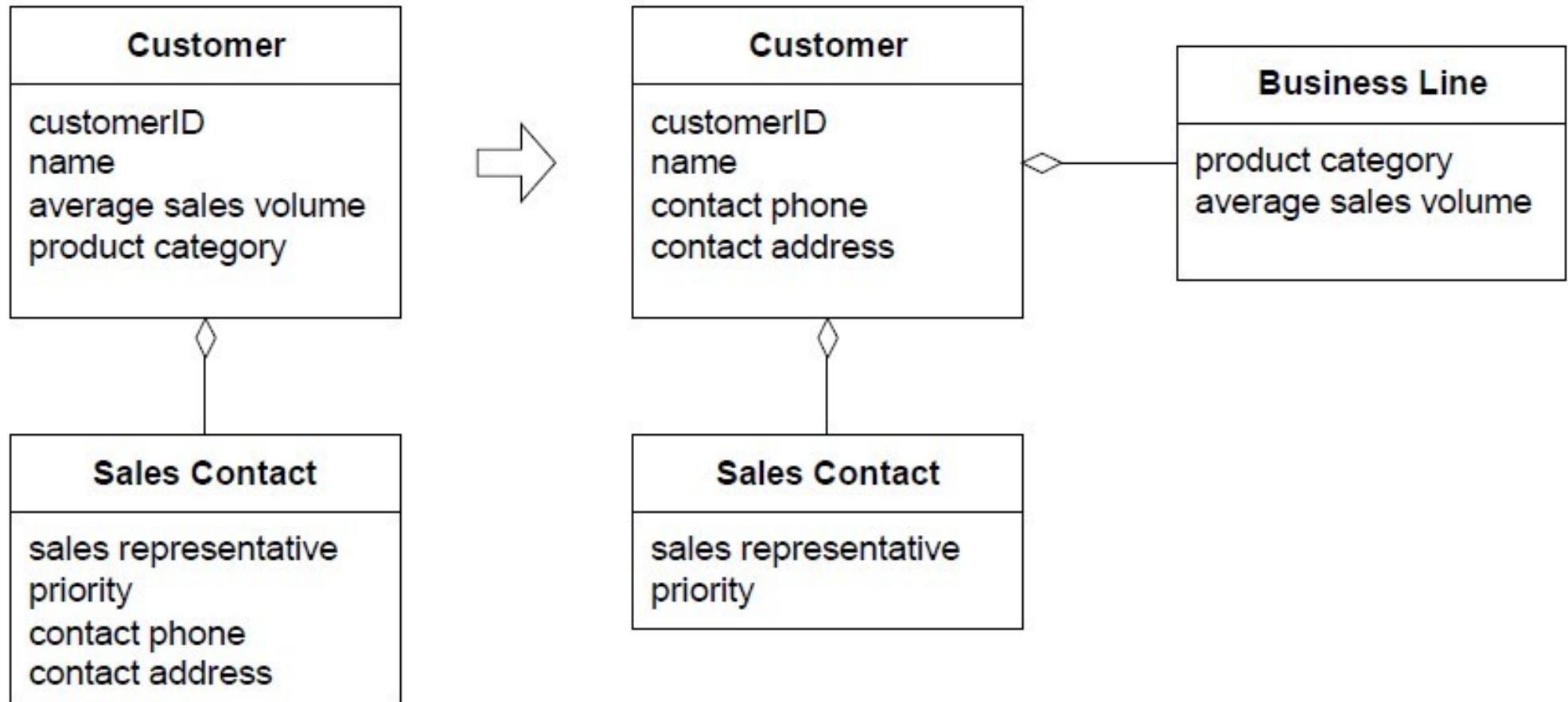
6.3 Modellgetriebener Entwurf

Entities (Entitäten)

- Objekte, die durch ihre Identität unterschieden werden und nicht durch ihre Attribute – Kontinuität über einen Lebenszyklus hinweg mit sich eventuell ändernden Attributen
- Operation definieren, die ein eindeutiges Ergebnis liefert – eventuell ein eindeutiges Attribut ergänzen
- *Modell* muß Gleichheit definieren
- Auch: *Reference Objects (Refenzobjekte)*

6.3 Modellgetriebener Entwurf

Attributes associated with identity stay with the ENTITY



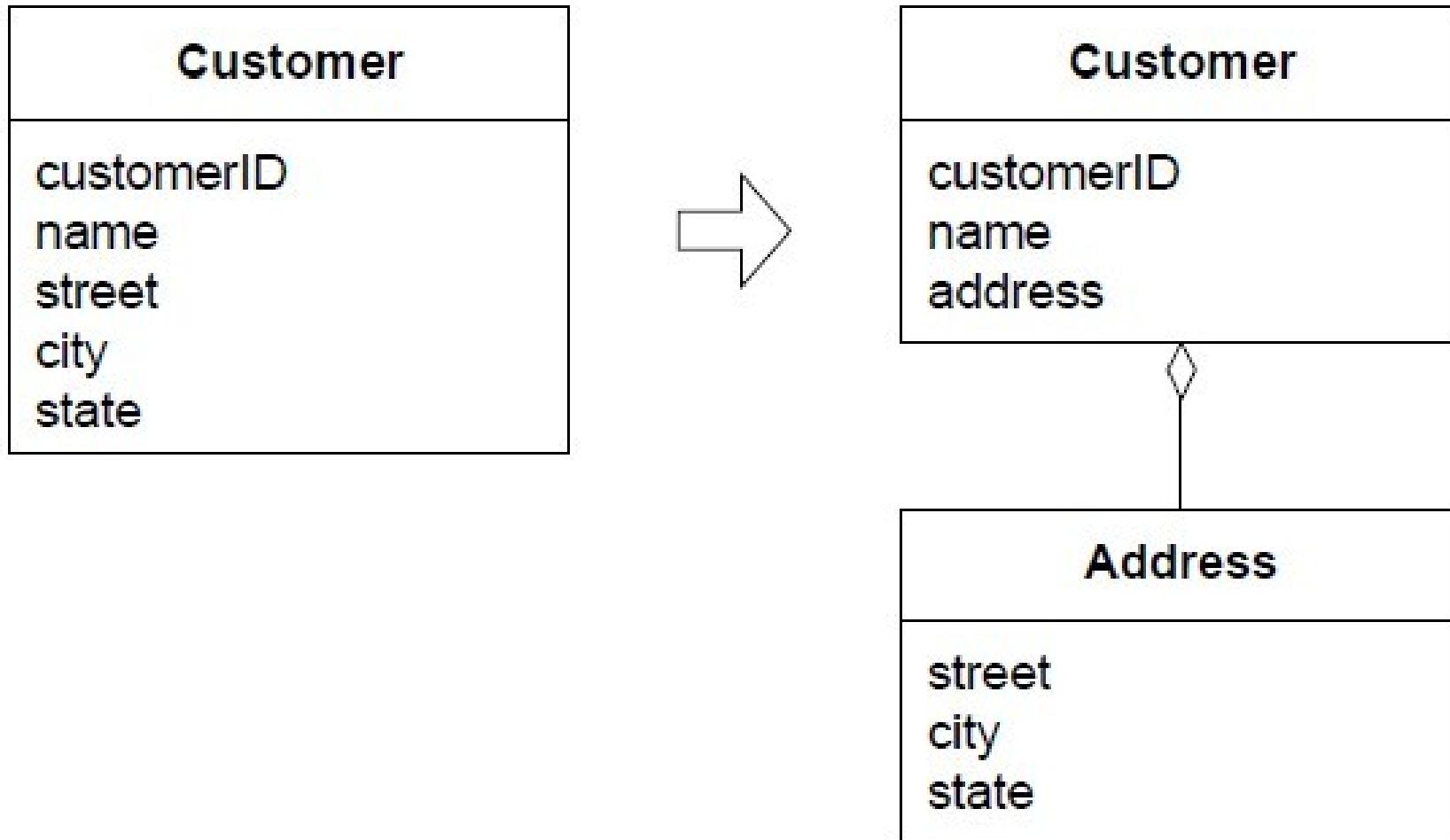
6.3 Modellgetriebener Entwurf

Value Objects (Wertobjekte)

- Objekte ohne das Konzept einer Identität und Kontinuität
- Nur die Attribute des Objekts sind von Interesse
- Unveränderlich machen (Immutable)
- Operationen als *Side-effect-free Functions*, die nicht von veränderlichen Werten abhängig sind

6.3 Modellgetriebener Entwurf

An attribute should be conceptually whole



6.3 Modellgetriebener Entwurf

Domain Event (Domänenereignis)

- Ein Ereignis, das Domänenexperten wichtig ist
- Modellierung von Informationen über die Aktivität der Domäne als Reihe diskreter Ereignisse
- Jedes Ereignis als Domänenobjekt darstellen
- Sind normalerweise unveränderlich (immutable, liegen in der Vergangenheit, sind geschehen)

6.3 Modellgetriebener Entwurf

Domain Event (Domäneneignis)

- In verteiltem System kann der Zustand einer *Entity* aus den *Domain Events* abgeleitet werden, die in einem Knoten zu einem bestimmten Zeitpunkt bekannt sind – d.h. kohärentes Modell möglich, auch wenn keine vollständige Information über das komplette System vorliegt
- Neuer Begriff, der im Blue Book noch nicht erwähnt wurde

6.3 Modellgetriebener Entwurf

Services (Dienste)

- Mitunter ist es keine Sache
- Fällt ein Prozess oder eine Transformation nicht in den Bereich einer Entity oder eines Value Objects, dann Modell mit einer Operation mit eigenständiger Schnittstelle ergänzen
- Definiert Servicevertrag mit einer Anzahl an **Assertions** für die Interaktion
- **Assertions** in der *Ubiquitous Language* eines *Bounded Context* ausdrücken
- Name Bestandteil der *Ubiquitous Language*

6.3 Modellgetriebener Entwurf

Partitioning Services into Layers

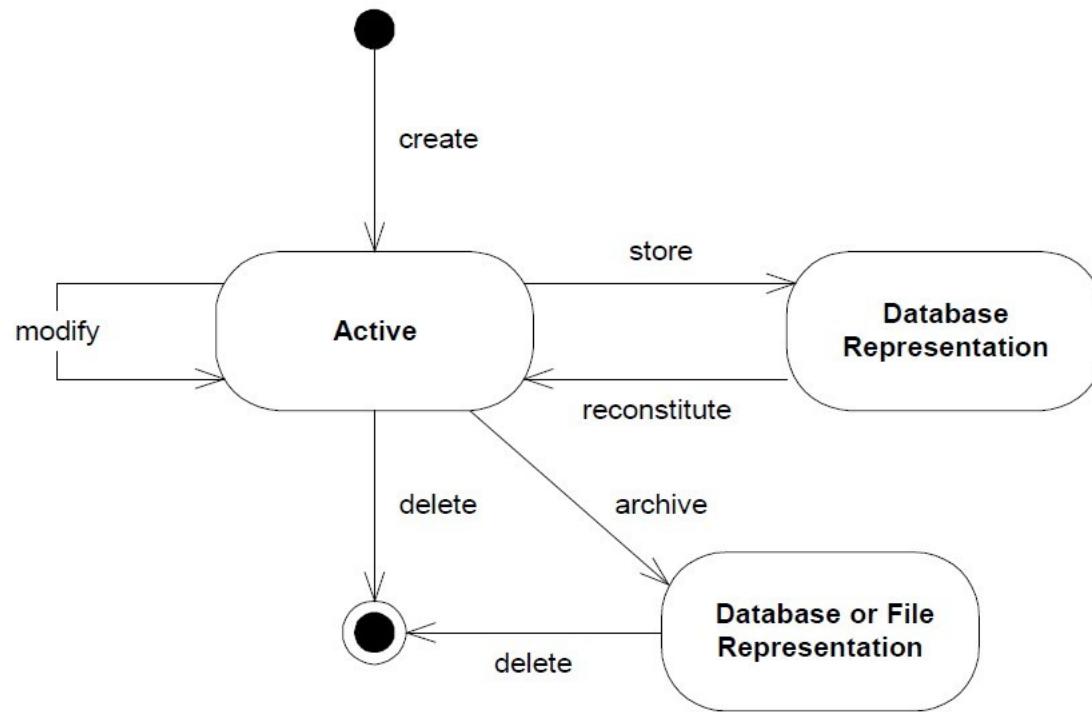
Application	Funds Transfer App Service: 1.Digests input (e.g. XML request), 2.sends message to domain service for fulfillment, 3.listens for confirmation, 4.decides to send notification using infrastructure service.
Domain	Funds Transfer Domain Service: Interacts with necessary Account and Ledger objects, making appropriate debits and credits, supplies confirmation of result (transfer allowed or not, etc.)
Infrastructure	Send Notification Service: Sends emails, letters, etc. as directed by application.

6.3 Modellgetriebener Entwurf

Modules (Module)

- Zusammenhängenden Satz an Konzepten
- Name sollte Teil der *Ubiquitous Language* sein - sind Teil des Modells und Name sollte das Verständnis der Domäne widerspiegeln
- Niedrige Kopplung untereinander – Konzepte, die unabhängig von einander verstanden und begründet werden können
- Modell soweit verfeinern, dass es nach High-Level Domänenkonzepten aufgeteilt und der Code so entkoppelt ist
- Vorsicht vor „Infrastructure Driven Packaging“, bei der die technischen infrastrukturellen Aspekte ausschlaggebend sind und eventuell nicht zur Domäne passen
- Auch: *Packages (Pakete)*

6.3 Modellgetriebener Entwurf



Management des Objekt-Lebenszyklus

- Integrität der Objekte über deren Lebenszyklus hinweg sicherstellen
- Das Domänenmodell vor der Komplexität des Life-Cycle-Managements bewahren
- Patterns
 - **Aggregates, Factories und Repositories**

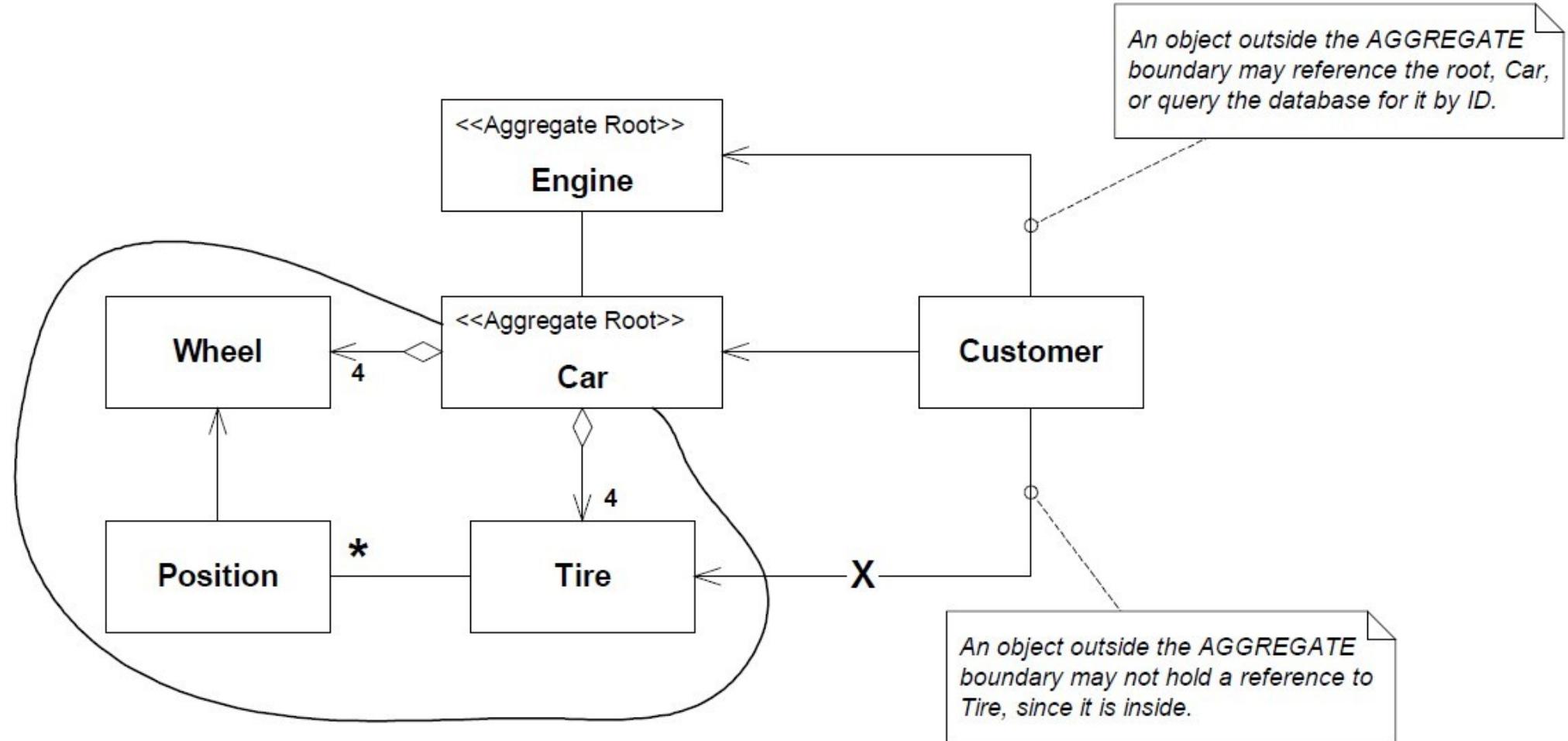
6.3 Modellgetriebener Entwurf

Aggregates (Aggregate)

- Entities und Value Objects zu Aggregates gruppieren und Zugriff auf diese Gruppen begrenzen
- Eine Entity als Wurzel eines jeden Aggregates und extern nur diese Wurzel referenzierbar
- Eigenschaften und Invarianten für das Aggregate als Ganzes definieren und Verantwortung für die Einhaltung und Durchsetzung liegt bei der Wurzel oder einem speziellen Mechanismus im Framework

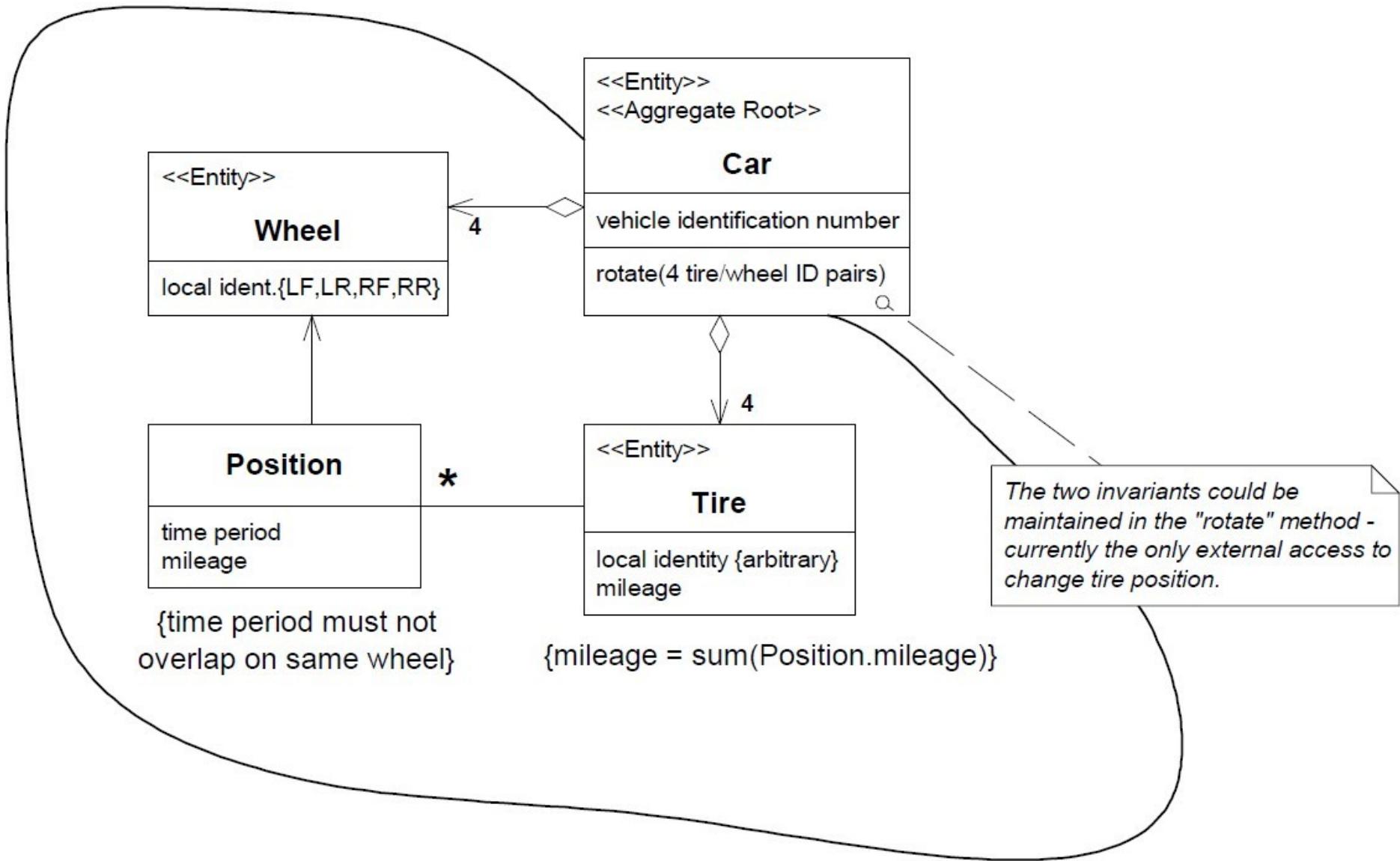
6.3 Modellgetriebener Entwurf

Local vs. Global Identity and Object References



6.3 Modellgetriebener Entwurf

AGGREGATE Invariants



6.3 Modellgetriebener Entwurf

Aggregates (Aggregate)

- Aggregat-Grenzen zur Steuerung von Transaktionen und Verteilung
- Innerhalb der Aggregatsgrenzen synchrone Konsistenzregeln
- Änderungen über die Grenzen hinweg asynchron
- Einzelnes Aggregate als Ganzes auf einem Server
- Verteilung verschiedener Aggregate auf unterschiedliche Knoten möglich

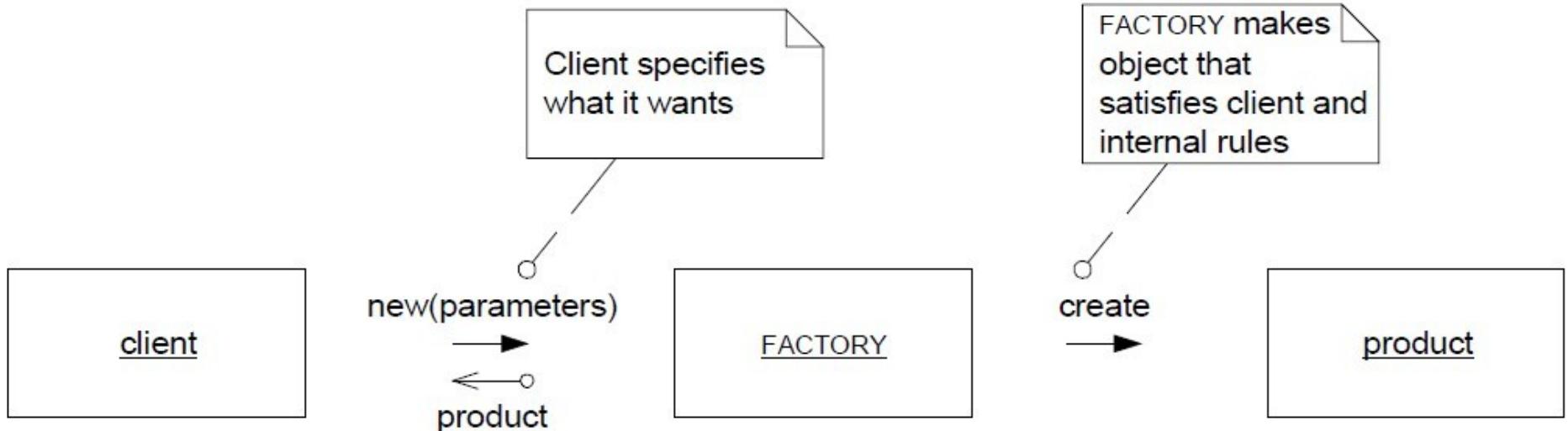
6.3 Modellgetriebener Entwurf

Factories (Fabriken)

- Verantwortung für das Erstellen von Instanzen komplexer Objekte und Aggregates auf ein separates Objekt verlagern, das selbst keine Verantwortung im Domänenmodell hat – aber Teil des Domänenentwurfs ist
- Liefert Schnittstelle, die den komplexen Zusammenbau kapselt und die ermöglicht, dass der Client keine Referenzen auf die konkreten Klassen der zu instanziierenden Objekte haben muss

6.3 Modellgetriebener Entwurf

Basic Interactions with a FACTORY

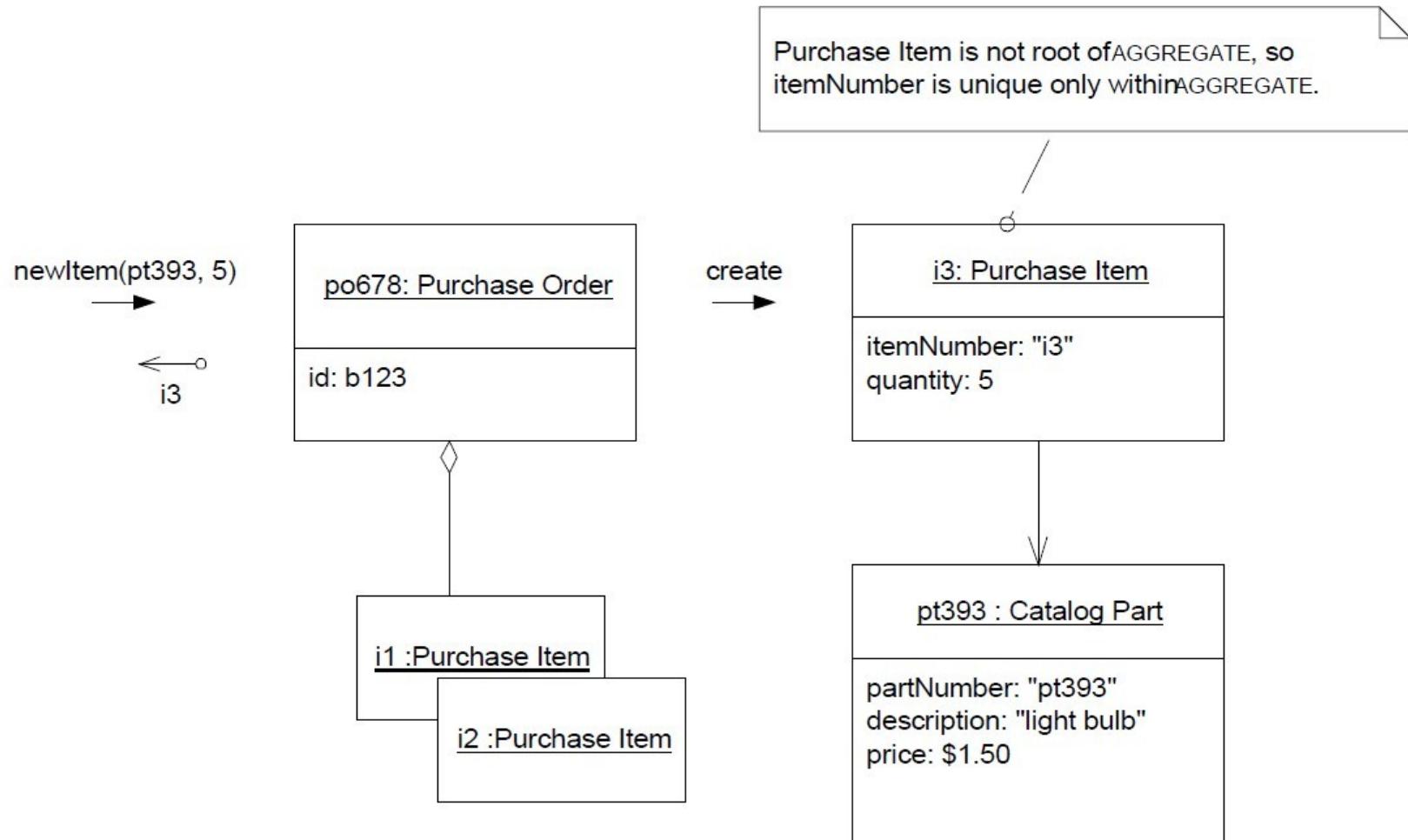


Factories (Fabriken)

- Erstellt ein ganzes Aggregate als ein Stück und garantiert dessen Invarianten
- Erstellt ein komplexes Value Object am Stück (eventuell durch Verwendung eines Builders für den Zusammenbau der einzelnen Elemente)

6.3 Modellgetriebener Entwurf

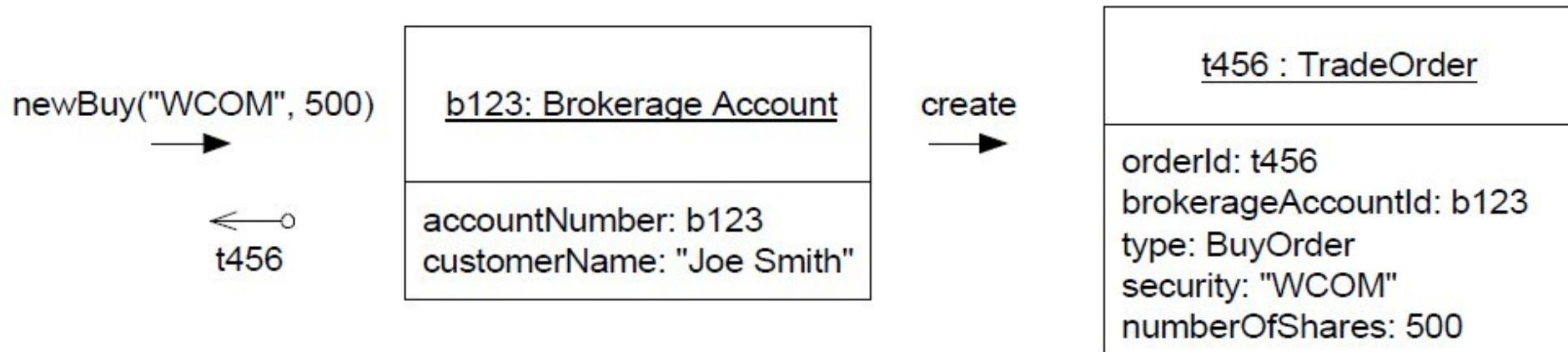
FACTORY METHOD Encapsulates Expansion of an AGGREGATE



Factory-Methode direkt am Aggregate-Root, um Teile des Aggregates zu erzeugen

6.3 Modellgetriebener Entwurf

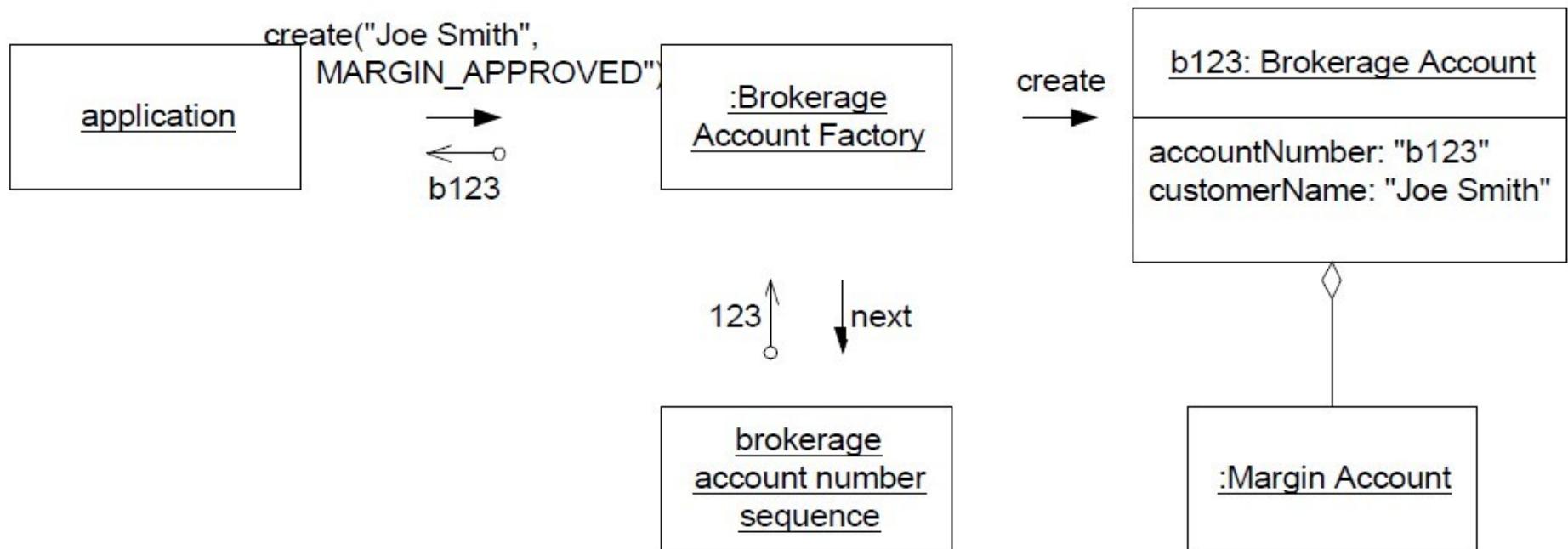
FACTORY METHOD Spawns ENTITY That is Not Part of the Same AGGREGATE



Factory-Methode um ein eng verbundenes Objekt zu erzeugen, dass aber nicht dem Aggregate gehört

6.3 Modellgetriebener Entwurf

Standalone FACTORY Builds AGGREGATE



- Standalone Factory – nicht verbunden mit einem Aggregate-Root
- Liefert das ganze Aggregate – Service
- Stellt Invarianten des Aggregates sicher und versteckt genaue Ausführung
- Abstrakte Factories

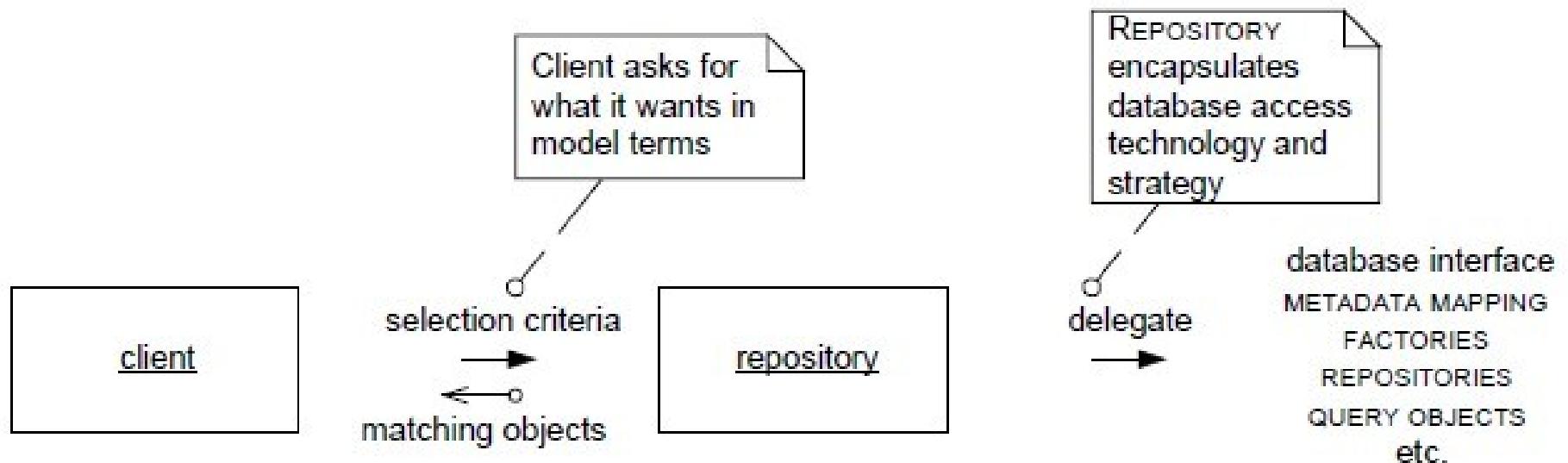
6.3 Modellgetriebener Entwurf

Repositories

- Zugriff auf *Aggregates* in Anfragen die in der *Ubiquitous Language* formuliert sind
- Dienst für jeden Aggregate-Typ, auf den global zugegriffen werden muss - Illusion einer Sammlung aller Objekte des Typs → der Wurzel dieses *Aggregates*
- Zugriff über eine bekannte globale Schnittstelle
- Methoden zum Hinzufügen und Entfernen von Objekten - kapselt das tatsächliche Einfügen oder Entfernen von Daten in die Datenbank

6.3 Modellgetriebener Entwurf

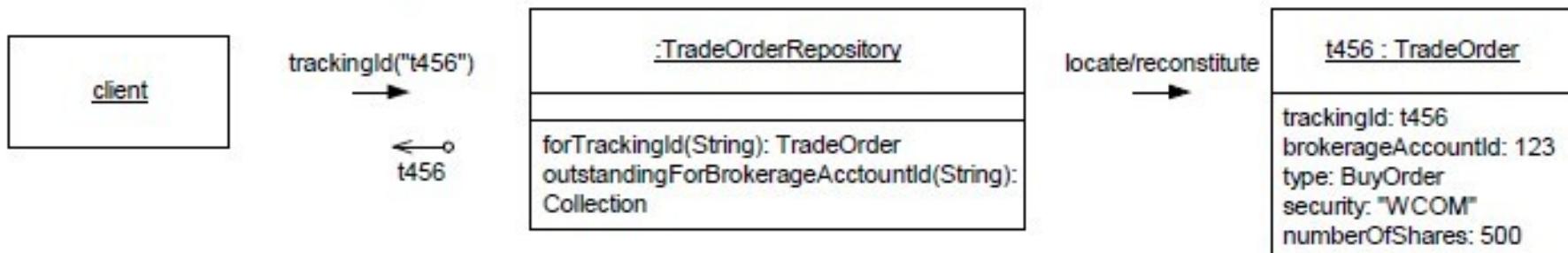
A REPOSITORY Doing a Search for a Client



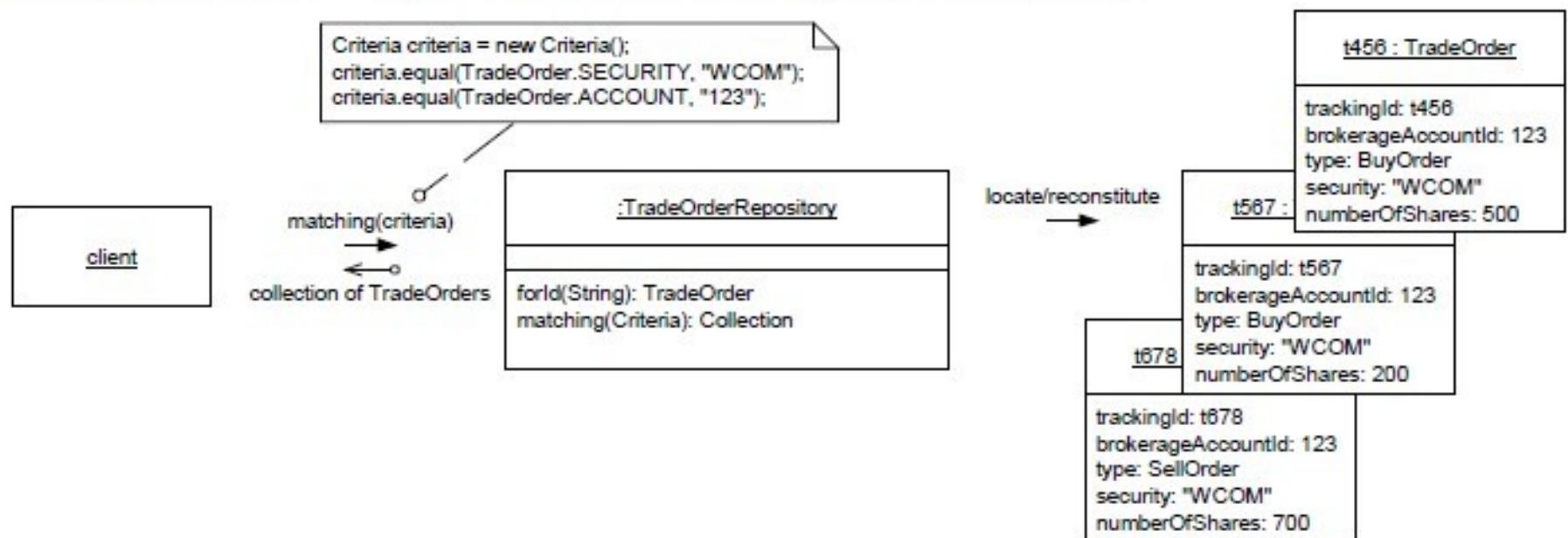
- Während Factories mit der Entstehung und Erzeugung von Objekten zu tun haben, sind Repositories für die „Mitte“ und das „Ende“ zuständig

6.3 Modellgetriebener Entwurf

Hard-Coded Queries in a Simple REPOSITORY

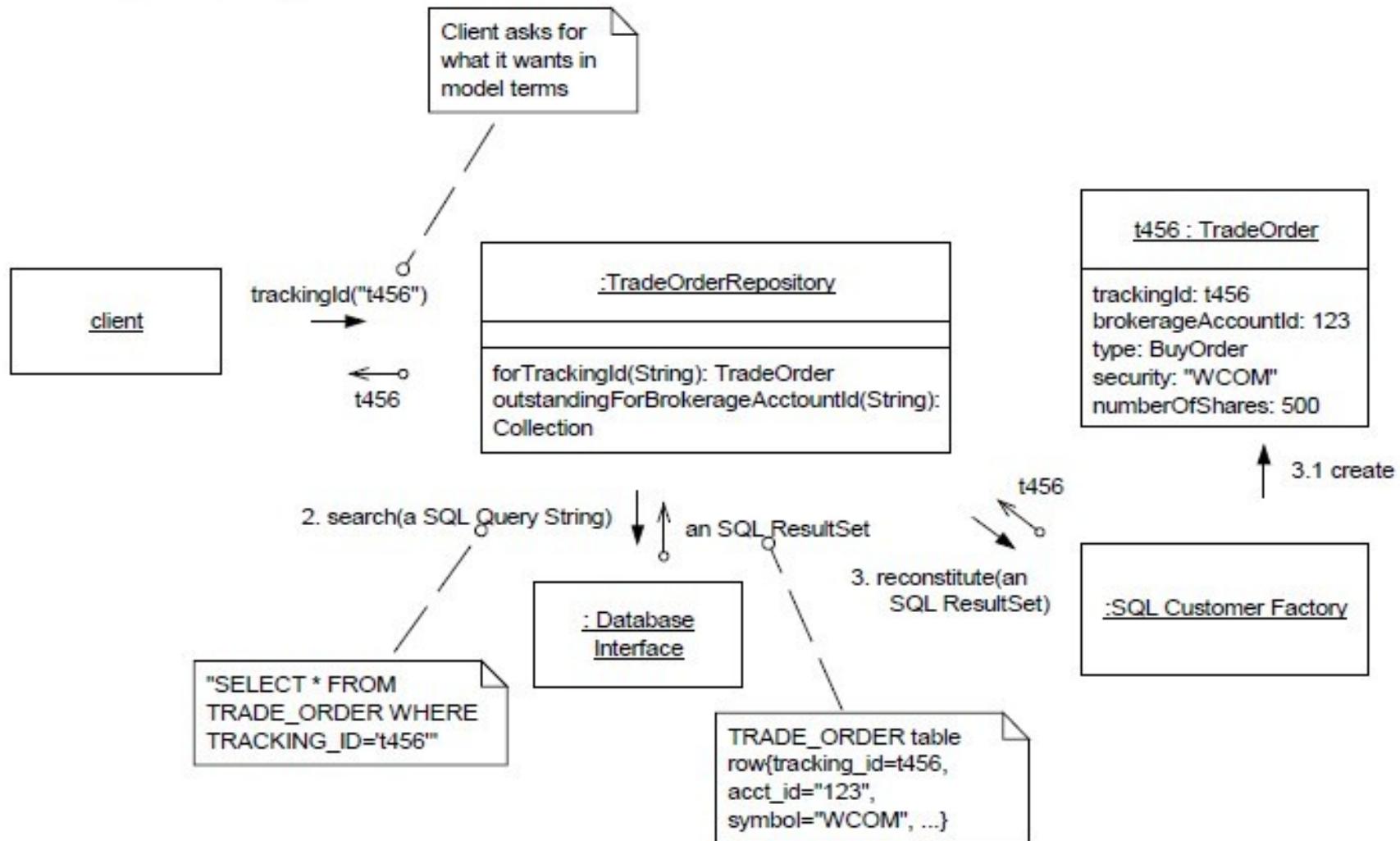


Flexible, Declarative SPECIFICATION of Search Criteria in a Sophisticated REPOSITORY

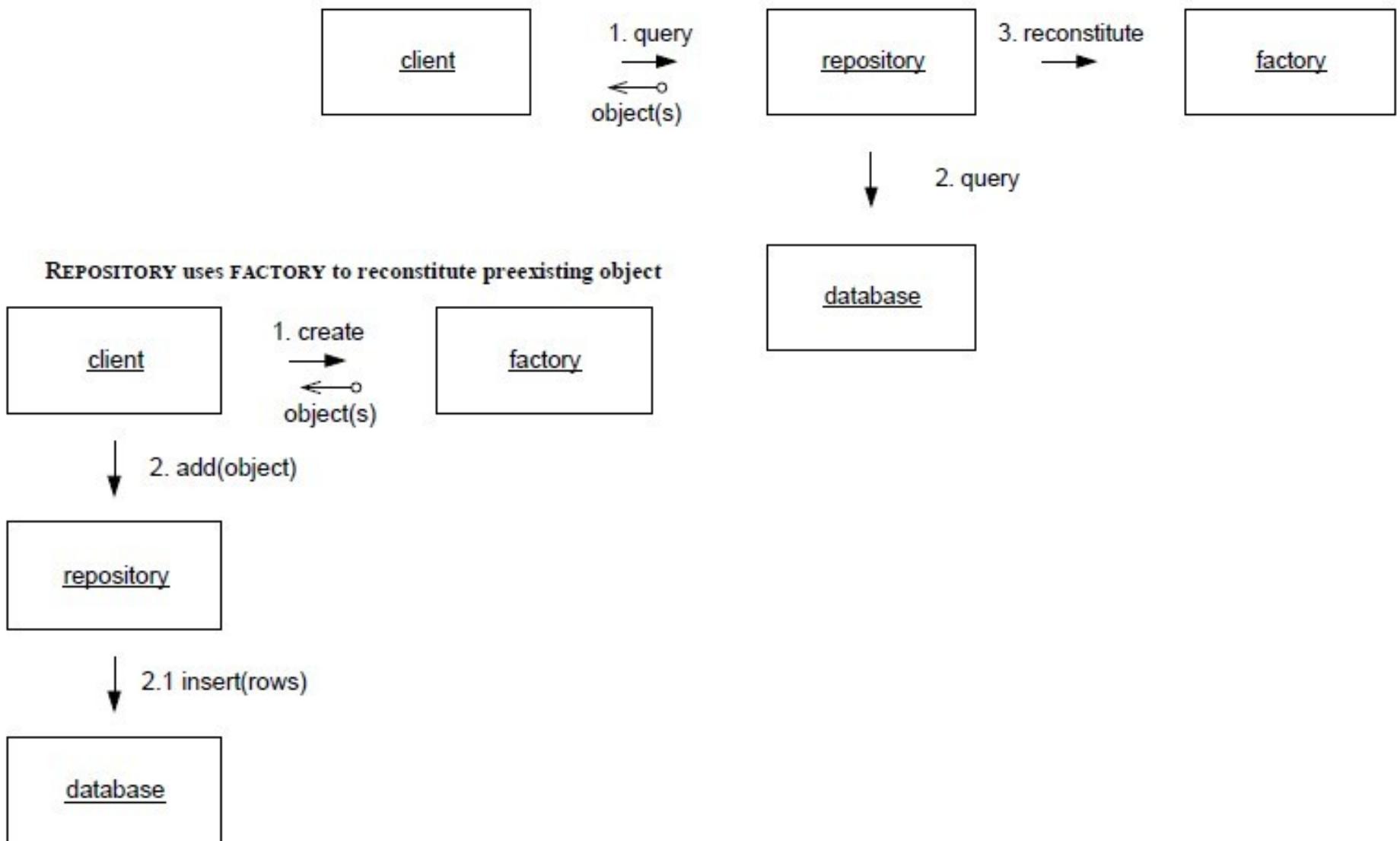


6.3 Modellgetriebener Entwurf

The Repository Encapsulates the Underlying Data Store



6.3 Modellgetriebener Entwurf



6.3 Modellgetriebener Entwurf

Repositories

- Vollständig instanzierte Objekte oder Sammlungen von Objekten werden zurückgegeben, deren Attributwerte den Kriterien entsprechen oder
- Proxies zurückgeben, die die Illusion vollständig instanzierter Aggregates vermitteln, aber die Daten verzögert nachladen (*Lazy Loading*)
- Repositories nur für Aggregate-Typen, die tatsächlich direkten Zugriff benötigen.
- Anwendungslogik auf das Modell fokussiert halten und gesamte Speicherung der Objekte und den Zugriff auf die Objekte an Repositories delegieren

6.4 Flexibles Design

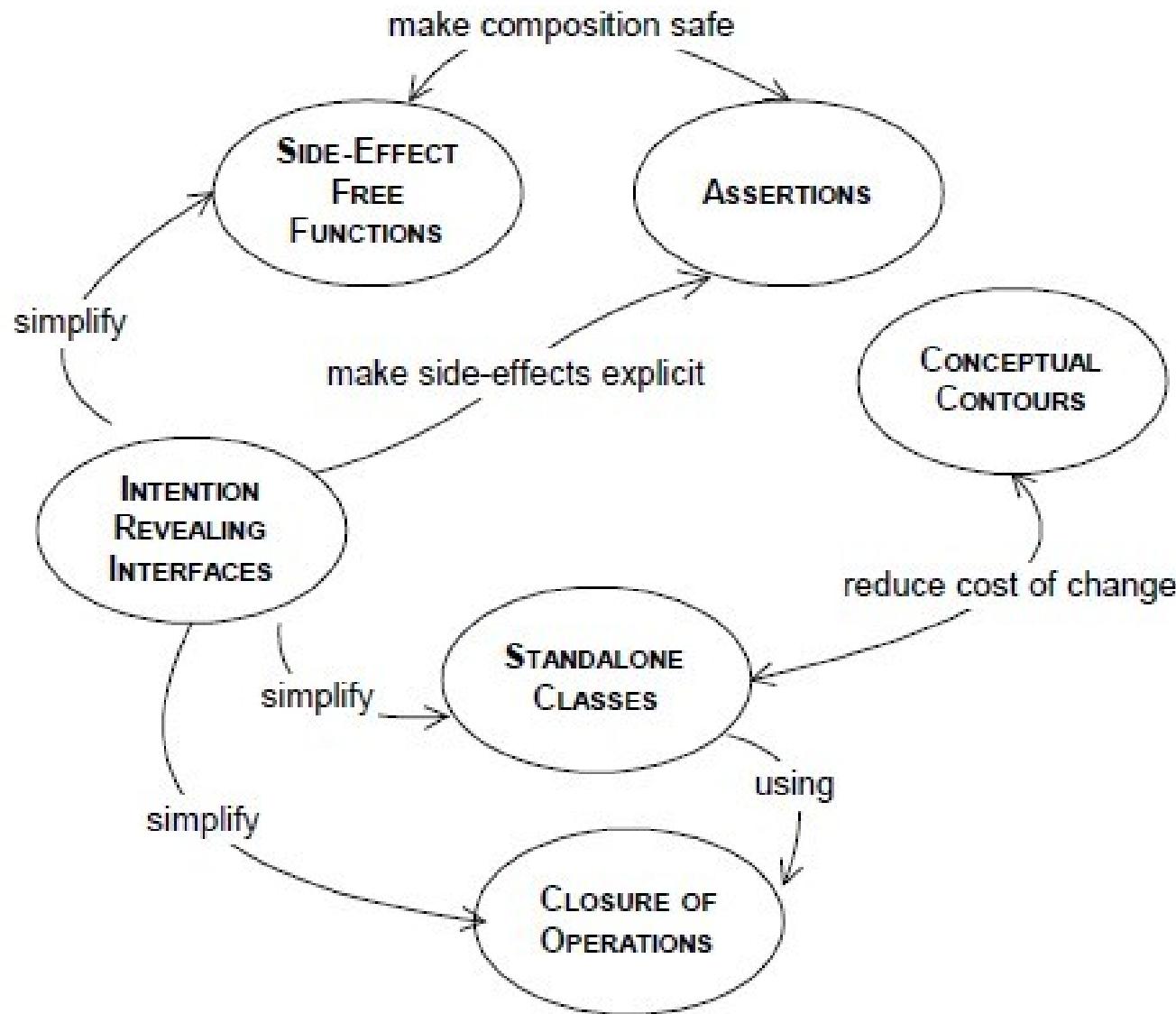
- Kennzeichnet Design, das zur Veränderung einlädt und nicht „durch sein Erbe belastet ist“
- Ergänzung zur tiefergehenden Modellierung
- Flexibel minimaler Satz lose gekoppelter Konzepte kombinierbar, um eine Reihe von Szenarien in der Domäne auszudrücken
- Elemente des Designs passen auf natürliche Weise zusammen, mit einem vorhersehbaren, klar charakterisierten und robusten Ergebnis

6.4 Flexibles Design

Das bedeutet im Kern:

- Verhalten offensichtlich machen
- Reduktion der Änderungskosten
- Erstellung von Software, mit der Entwickler gerne arbeiten

6.4 Flexibles Design

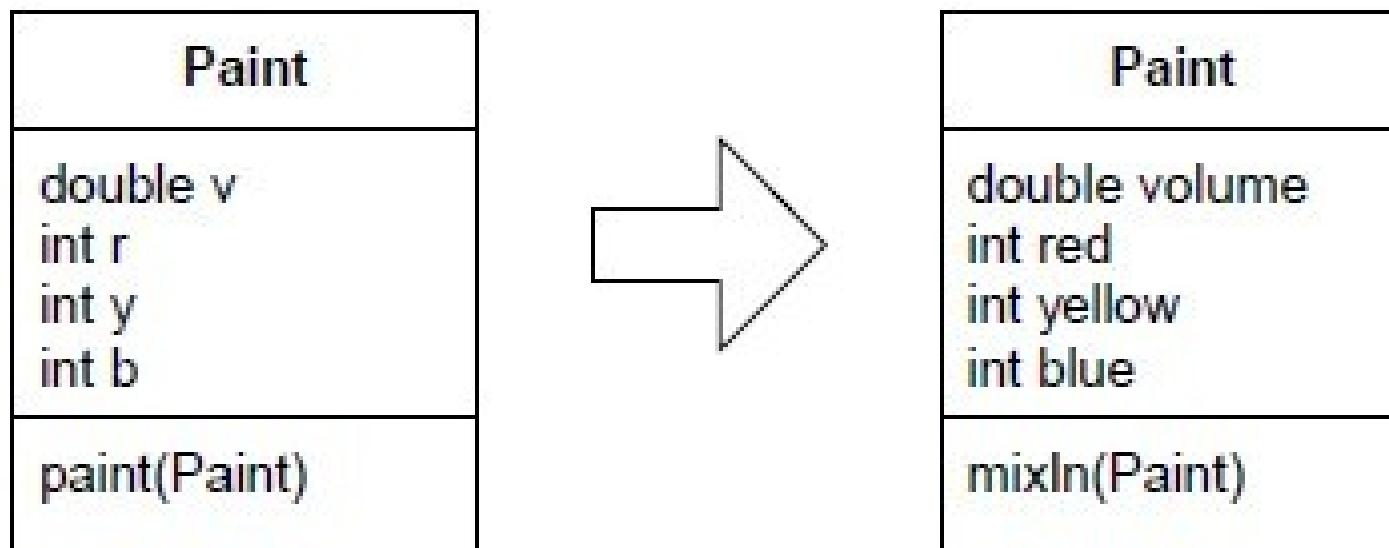


6.4 Flexibles Design

Intention-Revealing Interfaces (Ausdrucksstarke Schnittstellen)

- Keine Implementierungen direkt benutzen, um Kapselung aufrecht zu erhalten
- Klassen und Operationen nur so benennen, dass ihr Zweck beschrieben wird – nicht die Mittel, wie das erreicht wird
- Namen sollten der *Ubiquitous Language* entstammen → Bedeutung wird schnell klar
- Test für ein Verhalten vor der Implementierung schreiben → Denken aus Sicht des „Client-Entwicklers“

6.4 Flexibles Design



6.4 Flexibles Design

Side-Effect-Free Functions (Seiteneffektfreie Funktionen)

- Seiteneffekte machen Interaktionen und Kompositionen schwer vorhersagbar → Brechen so eigentlich die Abstraktion
- So viel Programmlogik wie möglich in seiteneffektfreien Funktionen/Operationen unterbringen
- Kommandos, Methoden die zu Zustandsänderungen führen, in sehr einfache, kurze, verständliche Operationen separieren (CQRS, ...)
- Kontrollierte Seiteneffekte in dem komplexe Logik in ***Value Objects*** verschoben werden
- Alle Operationen eines ***Value Objects*** sollten Seitenfreie Funktionen sein

6.4 Flexibles Design

Assertions (Zusicherungen)

- Nachbedingungen und Invarianten von Klassen und *Aggregaten*
 - Definieren Schnittstellenverträge und Modifikatoren für *Entites*
 - Invarianten auf *Aggregates*
- Verfügt die Programmiersprache über keinen geeigneten Mechanismus, dann automatisierte Unit-Tests
- Dokumentation oder Diagramm, falls das eine Option wäre

6.4 Flexibles Design

Standalone Classes (Eigenständige Klassen)

- Innerhalb eines Moduls nimmt die Schwierigkeit zu interpretieren mit externen Abhängigkeiten stark zu
- Kopplung (zu anderen Konzepten und Klassen) auf „nahezu Null“ beschränken
- In sich geschlossene Klasse erleichtert das Verständnis → kann in sich und für sich allein gelesen und interpretiert werden

6.4 Flexibles Design

Closure of Operations (Geschlossene Funktionen)

- Meist bei Operationen auf *Value Objects*
- Wenn möglich eine Operation definieren, deren Rückgabetyp der gleiche ist, wie der Typ der Argumente
- Spielt der Zustand der Operation eine Rolle, dann ist dieser Zustand faktisch auch ein (implizites) Argument der Funktion/Operation
- Operation/Funktion ist unter der Menge der Instanzen dieses Typs geschlossen → höherwertige Schnittstelle ohne Abhängigkeiten zu anderen Konzepte

6.4 Flexibles Design

Declarative Design (Deklarativer Entwurf)

- „Ausführbare Spezifikation“, siehe BDD
- Schwierig wirklich zu erreichen, prozeduralen und OOP-Ansätzen fehlt die formale Strenge dies sicherzustellen
- Anwendung der anderen Konzepte als Hilfe
- Entwicklung einer DSL für die Domäne

6.4 Flexibles Design

Deklarativer Entwurfsstil

- *Intention Revealing Interfaces*, *Side-Effect Free Functions* und *Assertions* unterstützen den Deklarativen Entwurf
- Kombinierbare Elemente → Flexibilität
- Mögliche Ansätze auf den nächsten beiden Folien ...

6.4 Flexibles Design

Drawing on Established Formalism (Arbeiten auf der Basis etablierter Formalismen)

- Nutzung bzw. Übertragung bereits etablierter Formalismen der Anwendungsdomäne oder angrenzender Domänen
- Oft schon über sehr lange Zeit entwickelt
- Gibt es schon definierte Mengen an Entitäten und anderen fachlichen Konzepten, ... etc.
- Bsp: Mathematische Konzepte in der Anwendungsdomäne – oft bereits separier-, kombinierbar und Seiteneffekt-frei

6.4 Flexibles Design

Conceptual Contours (Konzeptionelle Konturen)

- Entwurfselemente (Funktionen, Schnittstellen, Klassen, Aggregate) in zusammenhängende Einheiten zerlegen → Intuition/Erfahrung wichtig
- „Achsen der Veränderung und Stabilität“ durch sukzessives Refactoring beobachten, um grundlegende konzeptionelle Konturen zu identifizieren
- Modell auf konsistente Aspekte der Domäne ausrichten

6.5 Context Mapping für Strategic Design

Definitionen

- *Bounded Context* bekannt
- *Upstream-Downstream*:
 - Beziehung zwischen zwei Gruppen – Upstream-Team beeinflußt den Erfolg des Downstream-Teams, umgekehrt jedoch eher weniger
 - Upstream-Team kann unabhängig vom Downstream-Team erfolgreich sein
- *Mutually Dependent*
 - Wechselseitige Abhängigkeit → zwei Software-Projekte in getrennten Kontexten müssen gleichzeitig geliefert werden, um erfolgreich zu sein

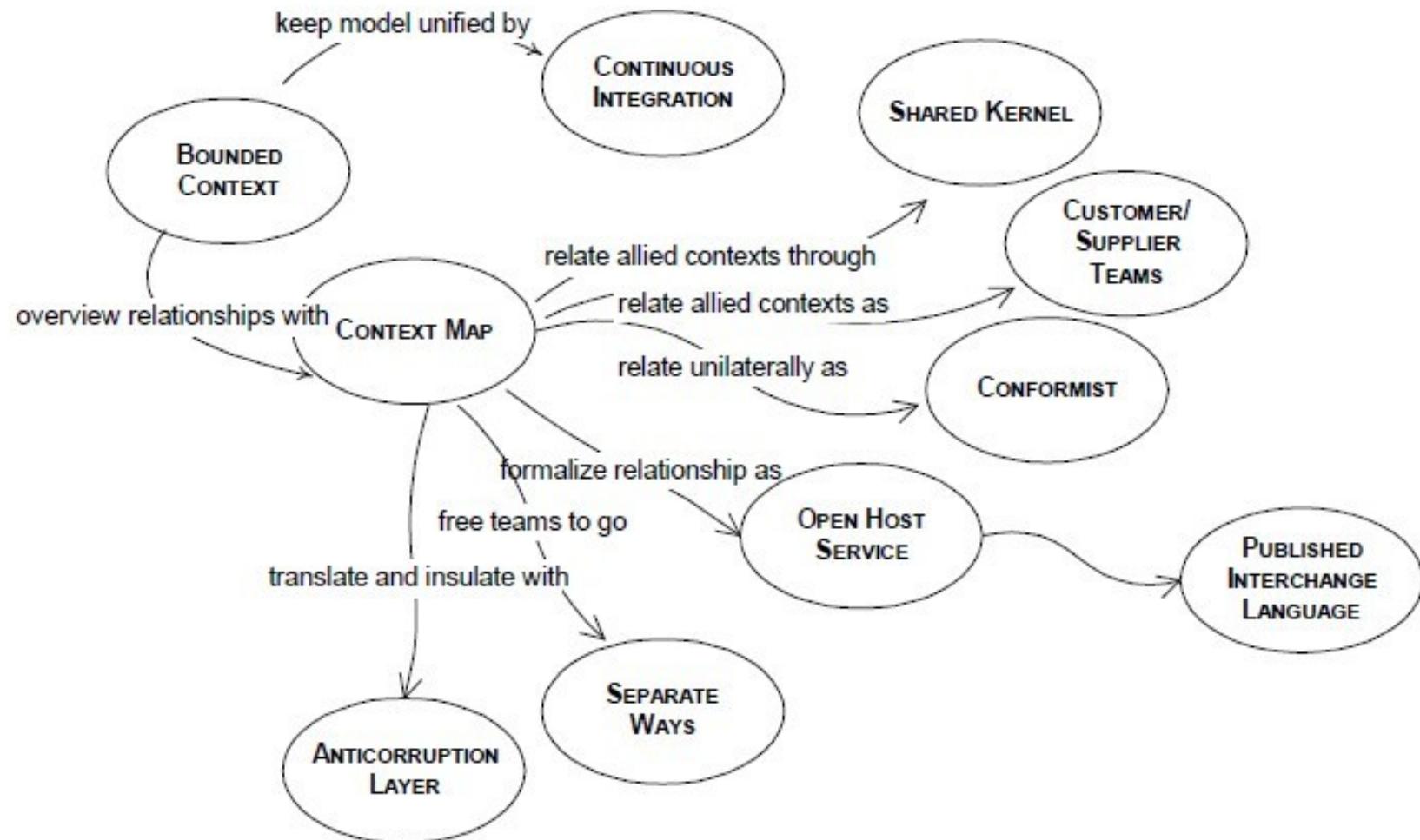
6.5 Context Mapping für Strategic Design

Definitionen

- *Free*:
 - Ein Softwareentwicklungskontext ist frei, wenn seine Auslieferung wenig oder kaum von anderen Kontexten abhängt

6.5 Context Mapping für Strategic Design

Navigation Map for Model Integrity Patterns



6.5 Context Mapping für Strategic Design

Context Map (Kontextlandkarte)

- Realistische, groß angelegte Sicht auf die Modellentwicklung über das gesamte Projekt hinweg
- Identifizieren aller *Modelle* des Projektes und des jeweiligen *Bounded Context* (inkl. impliziter Modelle von nicht-OOP-Subsystemen)
- Jeden *Bounded Context* benennen und in die *Ubiquitous Language* integrieren
- Berührungspunkte zwischen den Modellen beschreiben
- Jede Übersetzung für jede Art der Kommunikation skizzieren – alle Austausch-, Isolations- und Einflussmechanismen hervorheben

6.5 Context Mapping für Strategic Design

Patnership (Partnerschaft)

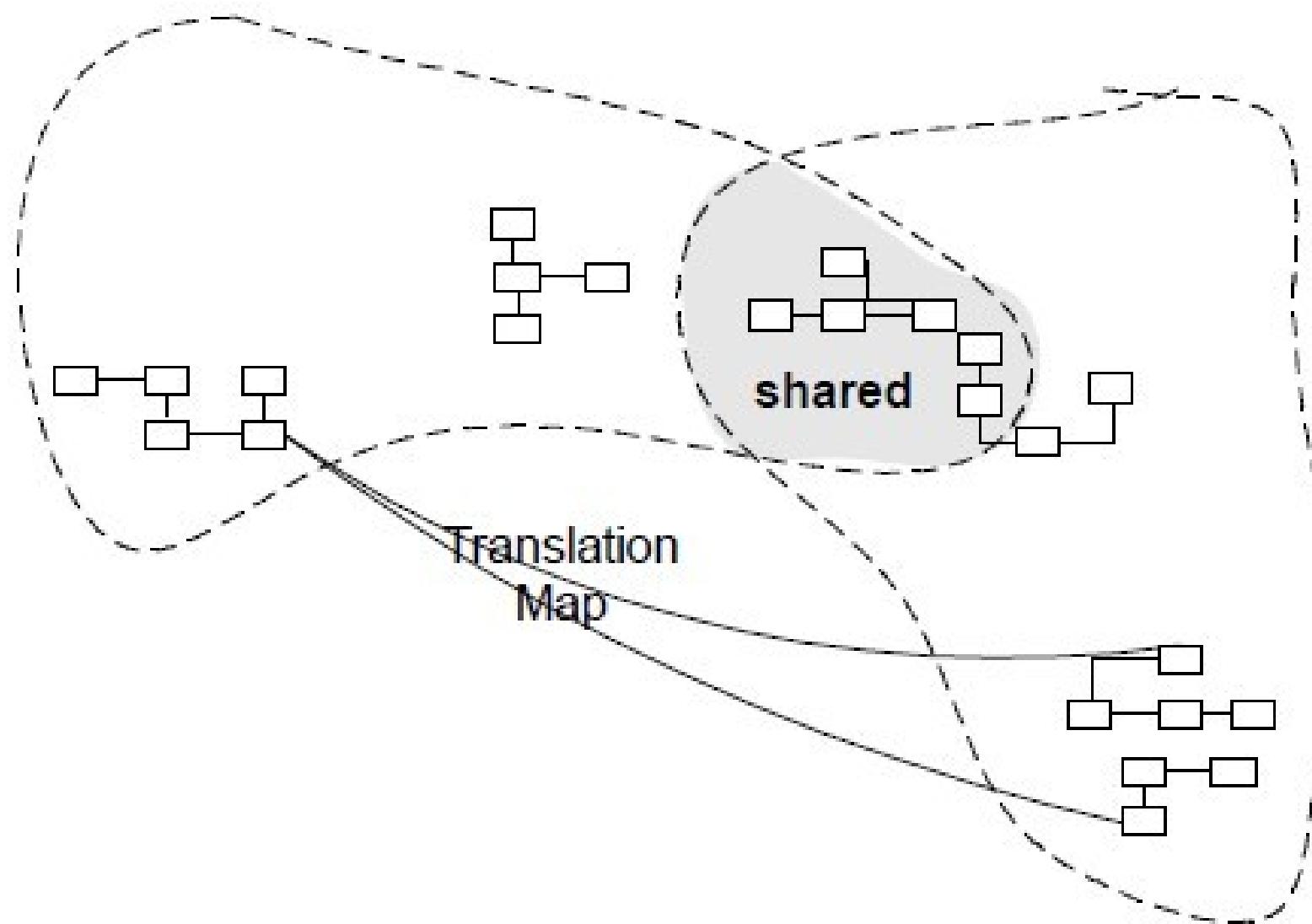
- Führt ein Fehlschlag in einem der beiden Kontexte zu einem Fehlschlag in beiden
→ etabliere eine Partnerschaft zwischen den Teams
- Prozess zur Planung und Entwicklung eines gemeinsamen Managements der Integration
- Im Orginal von 2004 noch nicht enthalten, sondern danach hinzugefügt

6.5 Context Mapping für Strategic Design

Shared Kernel (Geteilter Kern)

- Gemeinsame Nutzung von Komponenten produziert sehr enge wechselseitig Abhängigkeit
→ kann positive oder negativ sein
- Markiert explizit die Teilmenge des Domänenmodells, das von Teams gemeinsam genutzt wird
- Klein halten
- Explizit geteilte Artefakte sind etwas besonderes
→ Änderung muss abgestimmt werden
- Wichtig zur Unterstützung: *Continuous Integration, Ubiquitous Language*

6.5 Context Mapping für Strategic Design



6.5 Context Mapping für Strategic Design

Customer/Supplier Development ***(Kunde/Liferant)***

- Muß etabliert werden, wenn sich zwei Teams in einer ***Upstream-Downstream***-Beziehung befinden
- Downstream-Prioritäten fließen in die Upstream-Planung mit ein
- ***Continuous Integration*** zur Sicherstellung

6.5 Context Mapping für Strategic Design

Conformist (Konformist)

- Wenn vorheriger Ansatz nicht funktioniert, um *Upstream-Downstream*-Beziehung konstruktiv zu lösen
- Bounded Context Übersetzung wird eliminiert, indem die Downstream-Gruppe sich ausschließlich an das Upstream-Modell hält
- Engt zwar die Entwicklung Downstream ein, ist aber eine Möglichkeit noch produktiv zu sein

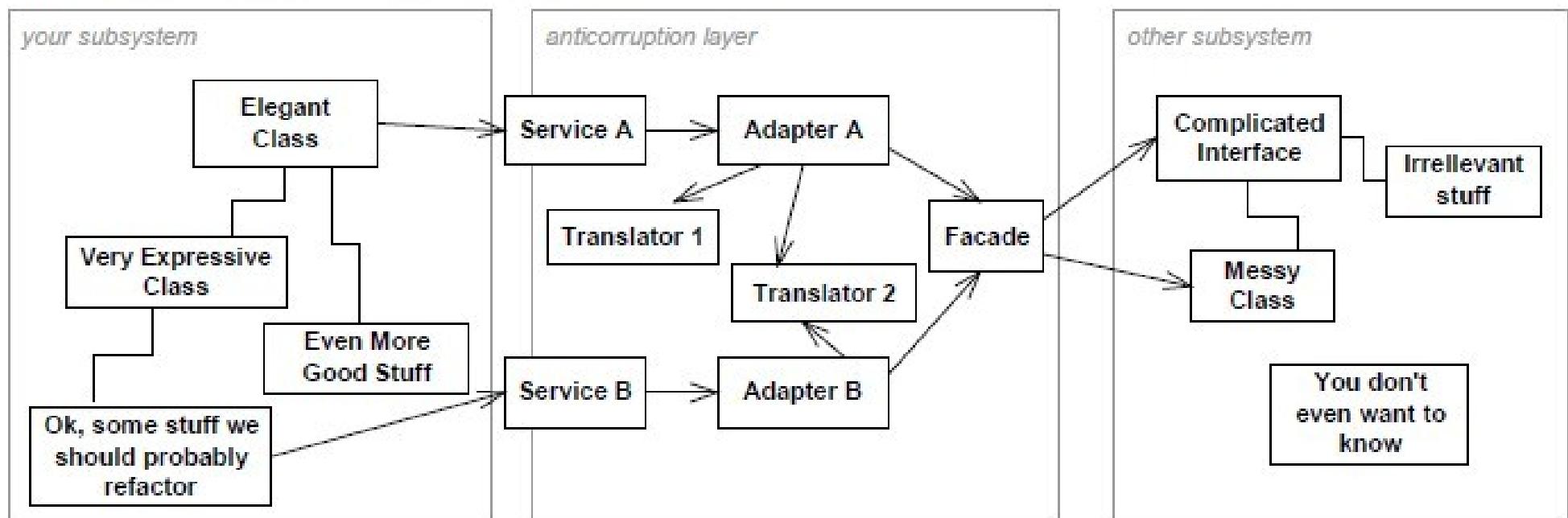
6.5 Context Mapping für Strategic Design

Anticorruption Layer (Antikorruptionsschicht)

- Isolationsschicht für den Downstream-Client, um dem Downstream-System die Funktionalität des Upstream-Systems im eigenen Domänen-Modell zur Verfügung zu stellen
- Nutzt Schnittstelle des anderen Systems → wenig oder gar keine Änderungen an anderem System
- Schicht übersetzt intern in eine oder beide Richtungen

6.5 Context Mapping für Strategic Design

Structure of an ANTICORRUPTION LAYER



6.5 Context Mapping für Strategic Design

Open-Host Service (Offen angebotener Dienst)

- Protokoll, das den Zugriff auf das eigene Subsystem als Menge von *Services* ermöglicht
- Offenes Protokoll für alle, die sich mit dem Subsystem integrieren müssen
- Verbessern und erweitern des Protokolls nach Anforderung, aber ...
- ... sind Anforderungen für Spezialfälle zu erfüllen → eigene Übersetzungsschicht → Protokoll bleibt kohärent und einfach

6.5 Context Mapping für Strategic Design

Published Language (veröffentlichte Sprache)

- Übersetzung zwischen den Modellen zweier Bounded Contexts
- Gut, dokumentierte gemeinsame Sprache, die die notwendige gemeinsame Domäneninformation ausdrückt
- *Published Language* als Datenaustauschstandard
- Oft kombiniert mit *Open Host Service*

6.5 Context Mapping für Strategic Design

Separate Ways (Getrennte Wege)

- Integration ist immer teuer, wenn der Nutzen gering ist, vermeide es
- *Bounded Context* hat definitionsgemäß keine Verbindung zu anderen
- Gibt Entwicklern mehr Freiheit für die Modellierung

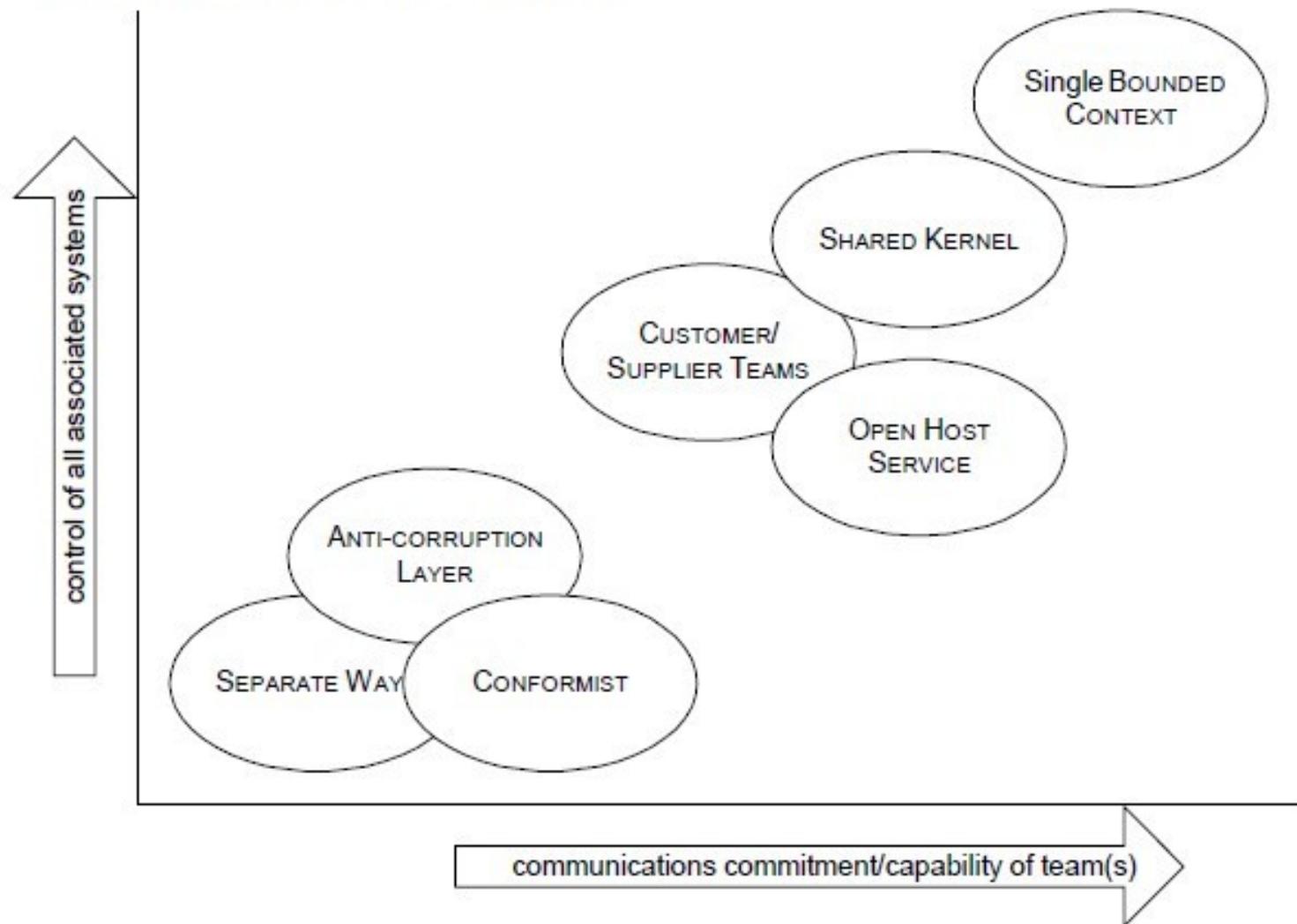
6.5 Context Mapping für Strategic Design

Big Ball of Mud (Große Matschkugel)

- Eingrenzen eines schlecht modellierten Teils der Anwendung mit schwer zu identifizierenden, ungünstig interagierenden Modellstrukturen (z.B. ein Legacy-System)
- Keine ausgeklügelte Modellierung dafür versuchen
- Schadensbegrenzung → versuchen die Einflüsse nicht in andere Kontexte ausbreiten zu lassen
- Im Original von 2004 nicht enthalten und neu aufgenommen

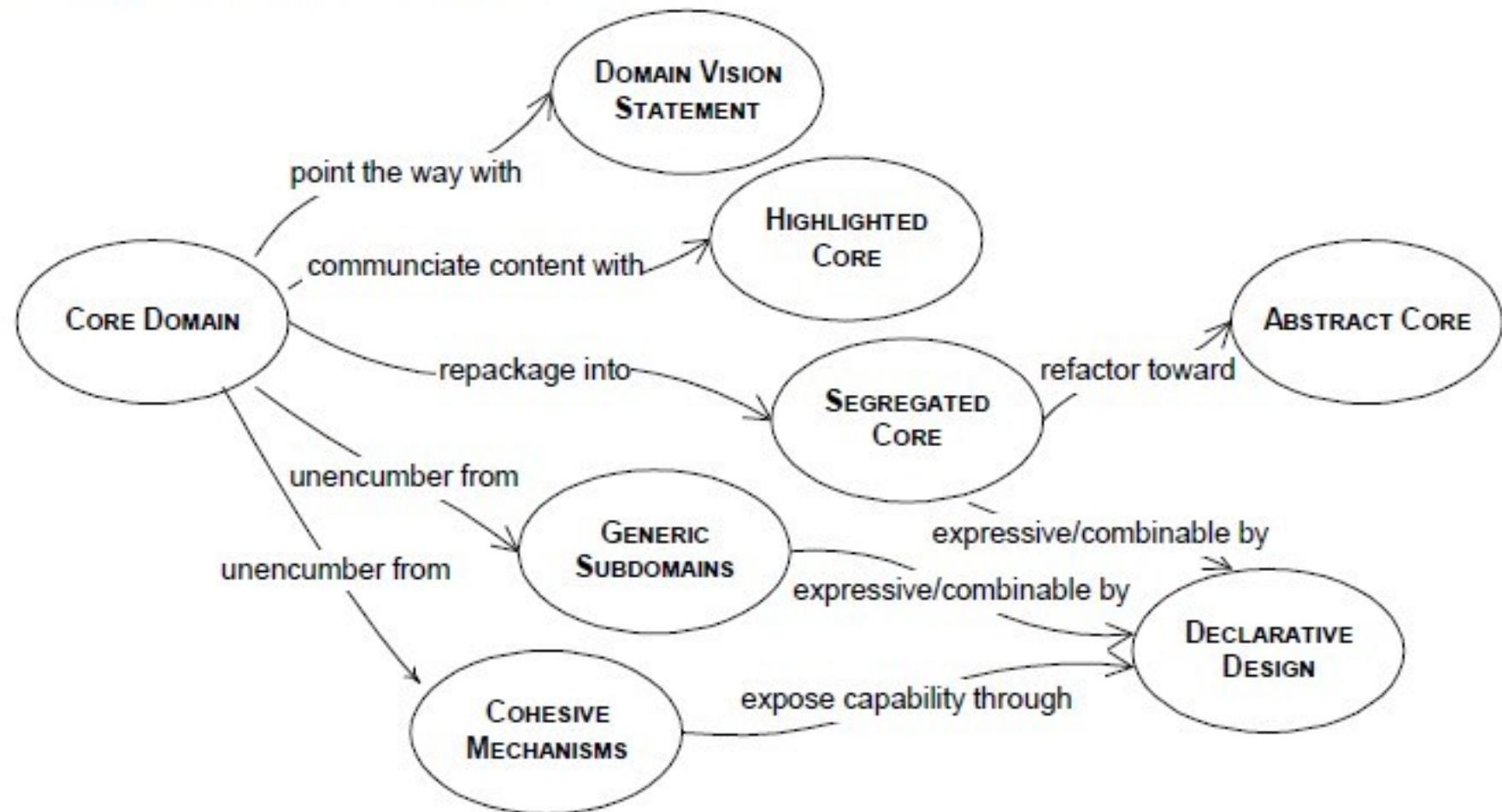
6.5 Context Mapping für Strategic Design

Relative Demands of CONTEXT Relationship Patterns



6.6 Destillation für Strategic Design

Navigation Map for Strategic Distillation



6.6 Destillation für Strategic Design

Core Domain (Kerndomäne)

- Modell auf das Wesentliche einschränken → möglichst kleiner Kern
- Ist der wichtigste Teil der Domäne
- Hier höheren Aufwand und Talent investieren, um ein möglichst gutes und fundiertes Modell zu entwickeln
- *Flexibles Design* für diesen Bereich

6.6 Destillation für Strategic Design

Generic Subdomain (Allgemeine Teildomäne)

- Einige Teile des Modells steigern Komplexität ohne für die Domäne wesentlich relevant zu sein → Identifikation dieser Subdomänen
- Generische Modelle für diese Subdomänen entwickeln und in Module auslagern
- Nicht mit Elementen der Kerndomäne vermischen
- Niedrigere Priorität als *Core Domain* (nach Auslagerung)
- Keine Kernetwickler diesen Domänen zuweisen
- Eventuell Einsatz von Standardlösungen statt Eigenentwicklung

6.6 Destillation für Strategic Design

Generic Subdomain mit *Off-the-Shelf-Lösung*

Vorteile:

- Weniger Code zu schreiben
- Wartung an Anbieter externalisiert
- Möglicherweise besserer Code (mehr getestet, in verschiedenen Einsatzszenarien bewährt) als In-House-Entwicklung

Nachteile:

- Man muß trotzdem Zeit investieren, den Code zu verstehen und zu evaluieren
- Qualität hängt von Dritthersteller ab

6.6 Destillation für Strategic Design

Generic Subdomain mit *Off-the-Shelf-Lösung*

Nachteile:

- Möglicherweise „Overengineered“ - eigene kleine, spezialisierte Lösung paßt und skaliert besser
- Integration eventuell aufwendig
- Möglicherweise ein neuer *Bounded Context* dessen Grenzen beachtet werden müssen → Referenzieren von *Entities* über diese Grenzen hinweg eventuell nicht einfach
- Unnötige weitere Abhängigkeiten, Compiler, Frameworks ... etc.

6.6 Destillation für Strategic Design

Generic Subdomain mit einem
Veröffentlichten Design oder Modell

Vorteile:

- Ausgereifter als eine eigene „gebastelte“ Lösung
- Sofort vorhandene gute Dokumentation

Nachteile:

- Passt eventuell nicht ganz zu den eigenen Spezifikationen (Overengineering)

6.6 Destillation für Strategic Design

Generic Subdomain mit einem beauftragter Implementierung außer Haus

Vorteile:

- Hält das Core-Team konzentriert auf der Kern-Domäne, schont Ressourcen
- Erlaubt mehr Entwicklung leisten zu können, ohne das Core-Team erweitern zu müssen – und damit auch das Wissen der Kerndomäne weiter kommunizieren zu müssen
- Forciert ein „Interface-Orientiertes“-Design und hilft die Subdomain generisch zu halten, da an den Dritt-Hersteller die Spezifikation/Interface kommuniziert wird

6.6 Destillation für Strategic Design

Generic Subdomain mit einem beauftragter Implementierung außer Haus

Nachteile:

- Core-Team muß trotzdem Zeit aufwenden für die Spezifikation und Coding-Styles, die an den Dritt-Hersteller kommuniziert werden müssen
- Nicht unerheblicher Aufwand, wenn der Code übernommen werden muß (Verstehen, ... etc.)
- Code-Qualität kann sehr unterschiedlich sein, abhängig von den Teams

6.6 Destillation für Strategic Design

Generic Subdomain mit einer *Inhouse-Entwicklung*

Vorteile:

- Einfache Integration
- Genau passende Implementierung – nicht zu viel und nicht zu wenig
- Zeit-Verträge können gemacht werden, um Spitzen abzufangen

Nachteile:

- Ständige Wartung und Schulung nötig
- Sehr oft wird der zeitliche und kostenmäßige Aufwand für solche Lösungen unterschätzt

6.6 Destillation für Strategic Design

Domain Vision Statement (Aussage zur Domänenvision)

- Kurze Beschreibung der *Core Domain* und des Wertes, den sie erzeugen soll → „Leistungsversprechen“
- Grobe Beschreibung – spezifische Elemente bleiben der individuellen Interpretation überlassen
- „Lebendige Vision“ → wird überarbeitet, wenn neue Erkenntnisse dazukommen

6.6 Destillation für Strategic Design

This is part of a DOMAIN VISION STATEMENT	This, though important, is <u>not</u> part of a DOMAIN VISION STATEMENT
<p>Airline booking system.</p> <p>The model can represent passenger priorities and airline booking strategies and balance these based on flexible policies. The model of a passenger should reflect the "relationship" the airline is striving to develop with repeat customers.</p> <p>Therefore, it should represent the history of the passenger in useful condensed form, participation in special programs, affiliation with strategic corporate clients, etc.</p> <p>Different roles of different users (e.g. passenger, agent, manager) are represented to enrich the model of relationships and to feed necessary information to the security framework.</p> <p>Model should support efficient route/seat search and integration with other established flight booking systems.</p>	<p>The UI should be streamlined for expert users but accessible for first time users.</p> <p>Access will be offered over the web, by data transfer to other systems, and maybe through other UI's, so interface will be designed around XML I.O. with transformation layers to serve web pages or translate to other systems.</p> <p>A colorful animated version of the logo needs to be cached on the client machine so that it can come up quickly on future visits.</p> <p>When customer submits a reservation, make visual confirmation within 5 seconds.</p> <p>A security framework will authenticate a user's identity and then limit access to specific features based on privileges assigned to defined user roles.</p>

6.6 Destillation für Strategic Design

This is part of a DOMAIN VISION STATEMENT	This, though important, is <u>not</u> part of a DOMAIN VISION STATEMENT
<p>Semiconductor factory automation</p> <p>The domain model will represent the status of materials and equipment within a wafer fab in such a way that necessary audit trails can be provided and automated product routing can be supported.</p> <p>The model will not include the human resources required in the process, but must allow selective process automation through recipe download.</p> <p>The representation of the state of the factory should be comprehensible to human managers, to give them deeper insight and support better decision making.</p>	<p>The software should be web enabled through a servlet, but structured to allow alternative interfaces.</p> <p>Industry standard technologies should be used whenever possible to avoid in-house development and maintenance costs and to maximize access to outside expertise. Open source solutions are preferred (e.g. Apache web server.)</p> <p>The web server will run on a dedicated server. The application will run on a single dedicated server.</p>

6.6 Destillation für Strategic Design

Highlighted Core (Hervorgehobener Kern)

- Kurze Beschreibung (aber mehr als die Vision) der *Core Domain* und die primären Interaktionen zwischen den Kernelementen
und/oder
- Elemente der *Core Domain* in der primären Ablage markieren ohne die Rollen näher zu erklären
- Entwicklern sollen leicht erkennen, was im Kern ist und was nicht

6.6 Destillation für Strategic Design

Highlighted Core (Hervorgehobener Kern)

- Beschreibt die Grundzüge der Kerndomäne
→ pragmatischer Indikator für die Bedeutung einer Modelländerung
- Haben Code- oder Modelländerungen Einfluß auf dieses Dokument → ***Grundlegende Änderung*** → Rücksprache mit allen Teammitgliedern → nach Änderung sofortige Veröffentlichung des neuen Dokuments
- „Leitet“ und „informiert“ - ändert aber nicht das Modell

6.6 Destillation für Strategic Design

Cohesive Mechanisms (*Zusammenhängender Mechanismus*)

- „Im Eifer des Gefechts“ wird konzeptuelles **Was** mitunter vom mechanischen **Wie** überlagert
- Konzeptionell zusammenhängenden Mechanismus in separates, leichtgewichtiges Framework unterteilen
- Formalismen oder gut dokumentierte Kategorien von Algorithmen/Mustern verwenden
- Funktionalität des Frameworks durch ein *Intention-Revealing Interface* zur Verfügung stellen

6.6 Destillation für Strategic Design

Cohesive Mechanisms *(Zusammenhängender Mechanismus)*

- Andere Elemente der Domäne können sich jetzt darauf beschränken das **Was** auszudrücken und das **Wie** an das Framework zu delegieren → deklarativer Design-Stil (DSL)
- **Generic Subdomains** herausfaktorisieren erhöht Übersichtlichkeit und **Cohesive Mechanisms** kapselt komplexe Vorgänge

6.6 Destillation für Strategic Design

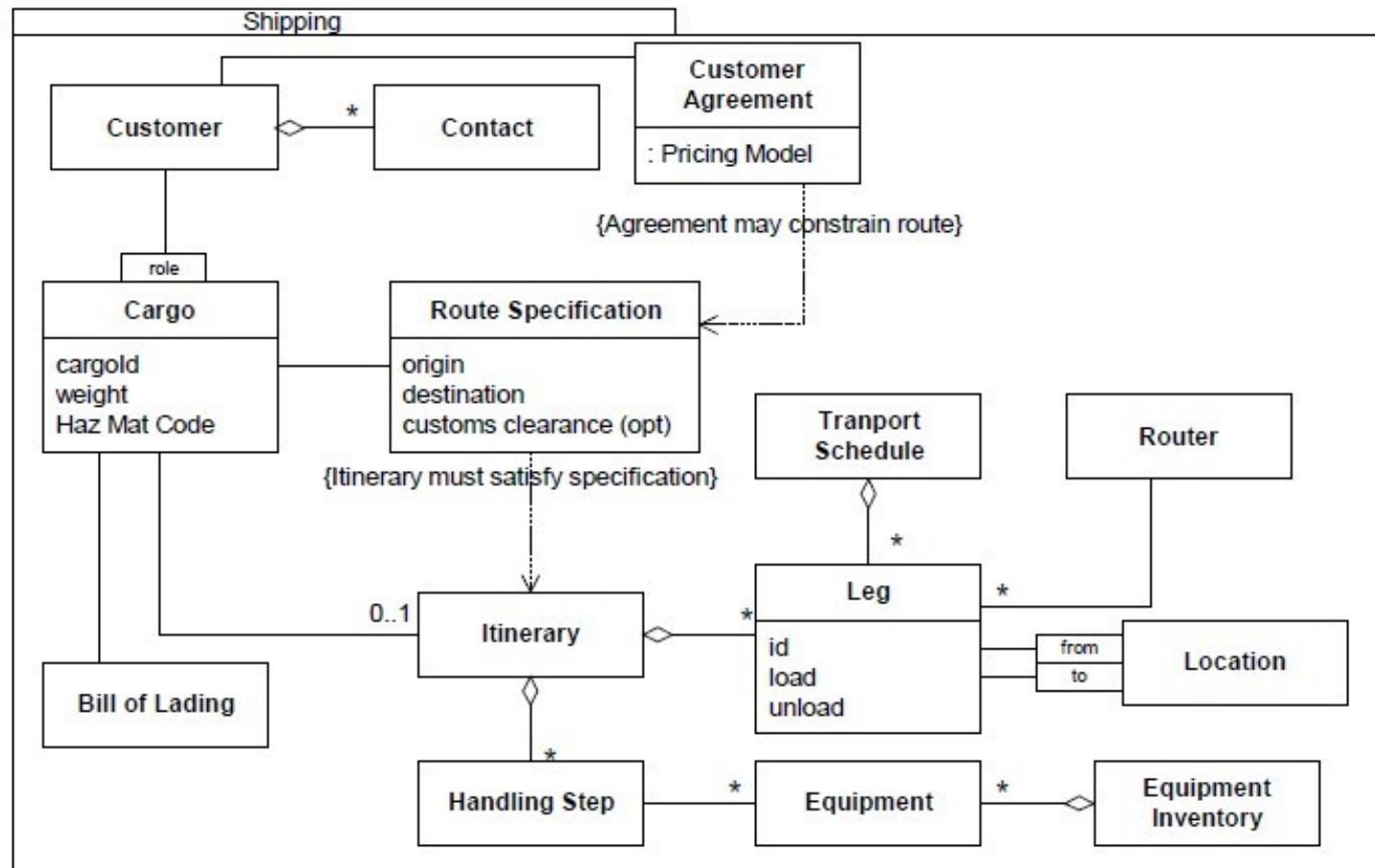
Segregated Core (Abgetrennter Kern)

- Kernelemente – Elemente der *Core Domain* – können eng mit generischen gekoppelt sein → konzeptionelle Zusammenhang für Kerndomäne nicht sichtbar, unübersichtliche und verworrene Abhängigkeiten
- Modell überarbeiten und alle „*Misch-Elemente*“ in andere Objekte und Pakete verschieben → Zusammenhang des Kerns stärken, Kopplung an den restlichen Code reduzieren

6.6 Destillation für Strategic Design

Segregating the Core of a Cargo Shipping Model

We start with the following model as the basis of software for cargo shipping coordination.



6.6 Destillation für Strategic Design

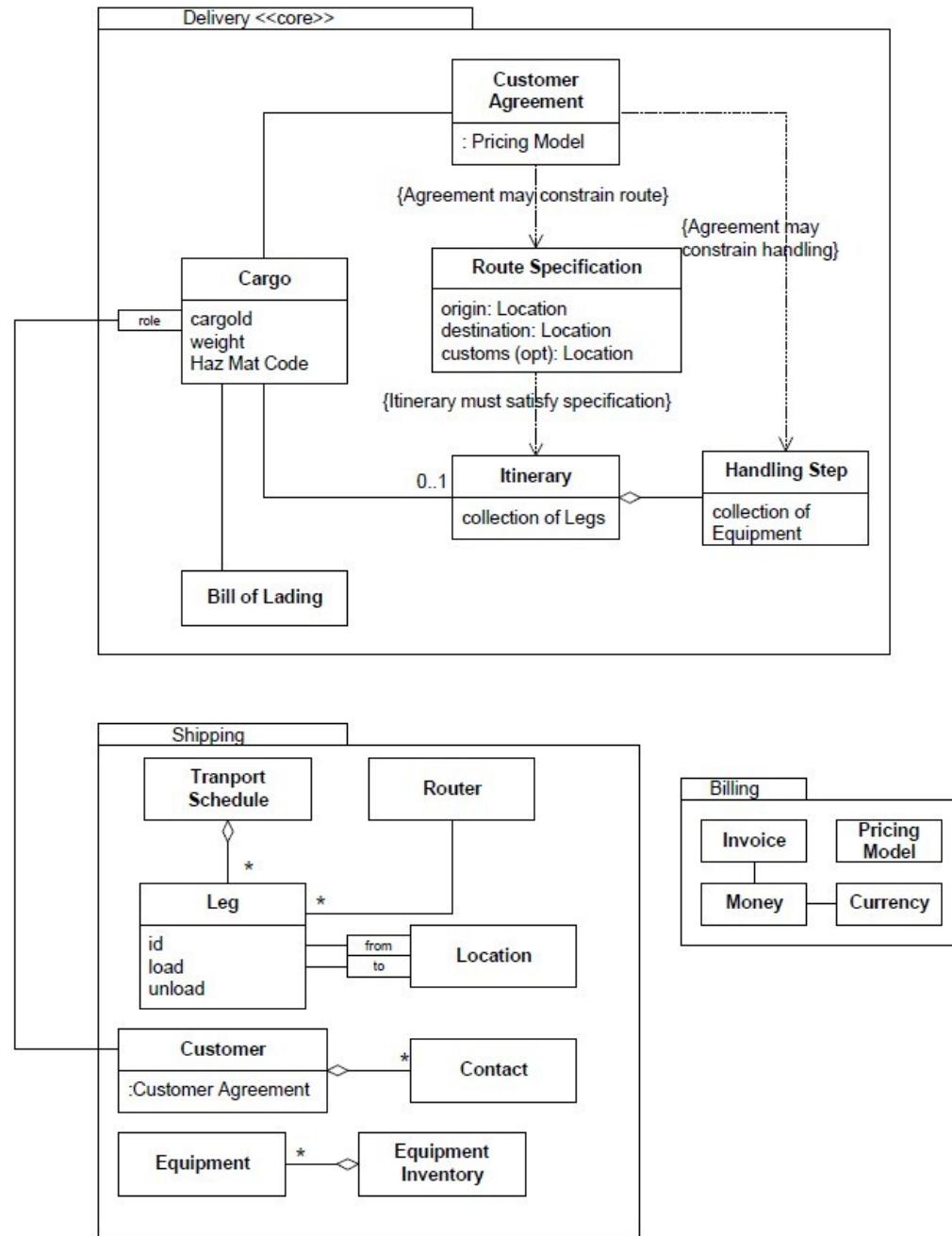
Now, what is the essence of the shipping model? Usually a good place to start looking is the “bottom line”. This might lead us to focus on pricing and invoices. But we really need to look at the DOMAIN VISION STATEMENT. Here is an excerpt from this one.

...Increase visibility of operations and provide tools to fulfill customer requirements faster and more reliably...

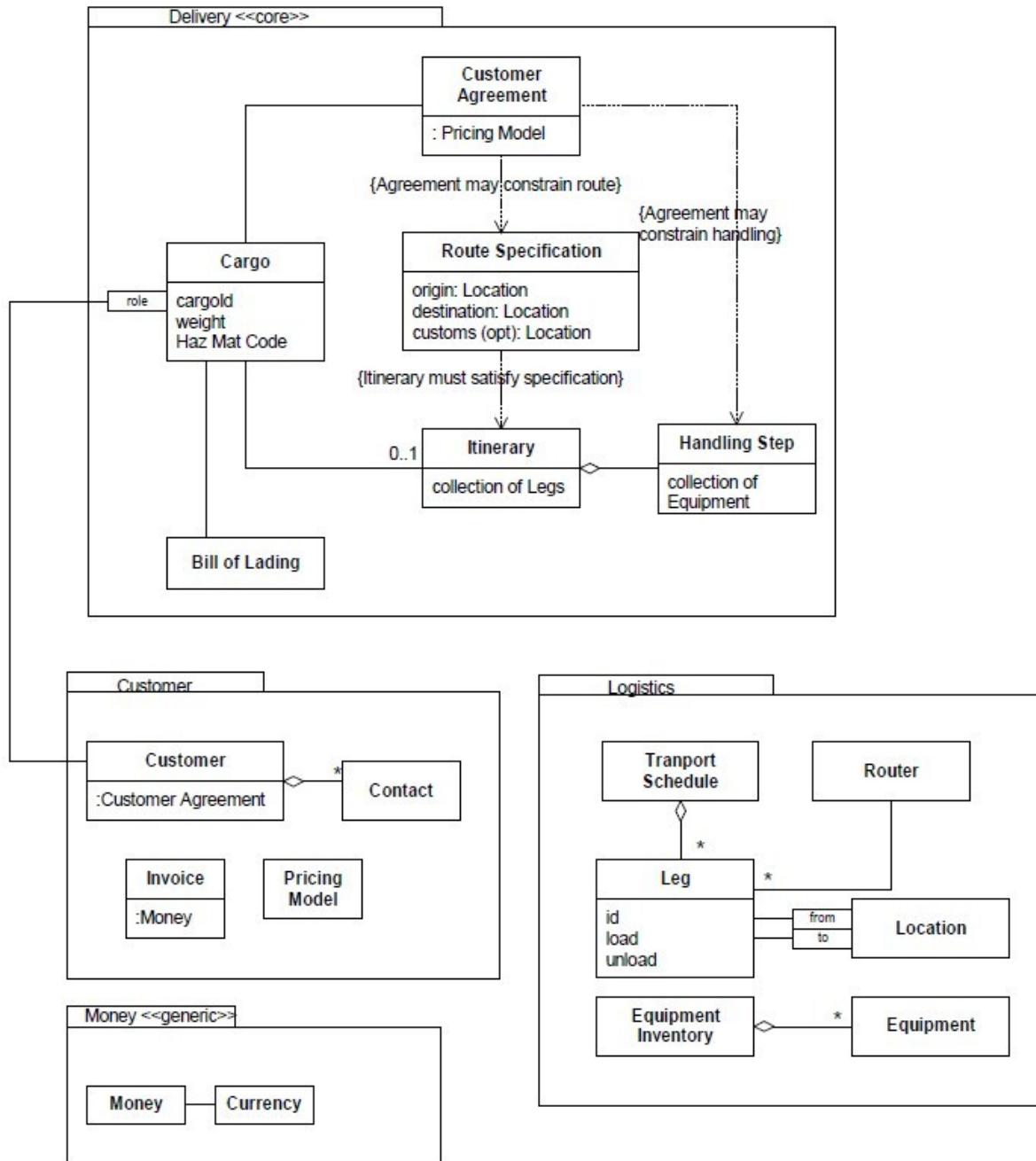
This application is not being designed for the sales department. It is going to be used by the front-line operators of the company. So let's relegate all money related issues to (admittedly important) supporting roles. Someone has already placed some of these items into a separate package (**Billing**). We can keep that, and further recognize that it is a supporting package.

The focus needs to be on the cargo handling – delivery of the cargo according to customer requirements. Extracting the classes most directly involved in these activities produces a SEGREGATED CORE in new package called “**Delivery**”.

6.6 Destillation für Strategic Design



6.6 Destillation für Strategic Design



6.6 Destillation für Strategic Design

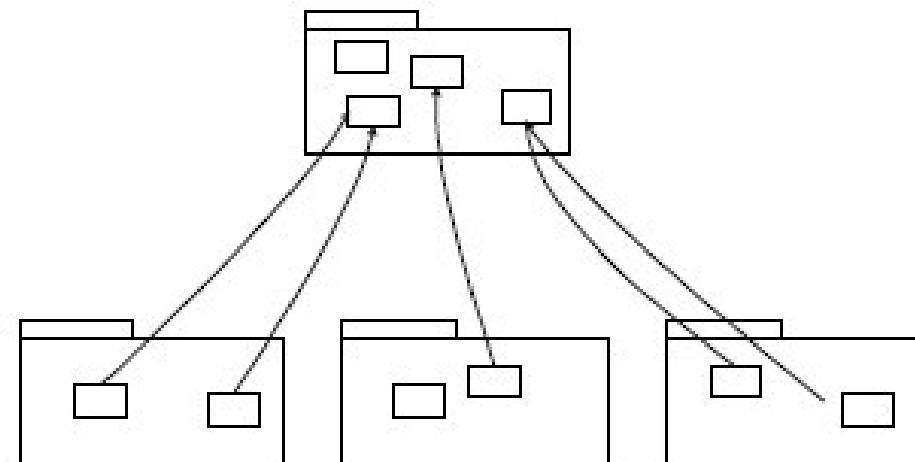
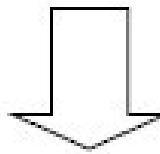
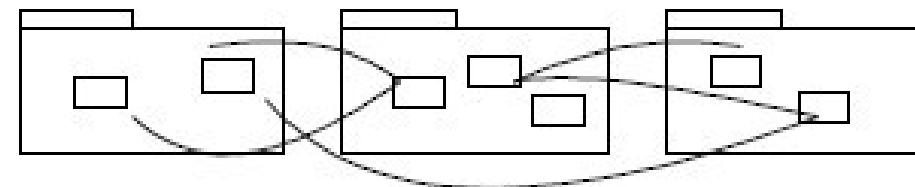
Abstract Core (Abstrakter Kern)

- Modell der *Core Domain* allein ist oft so komplex, dass es unübersichtlich ist
- Grundlegendsten differenzierenden Konzepte im Modell identifizieren und in eigentständige Klassen überführen, abstrakte Klassen oder Schnittstellen entwerfen → abstraktes Modell
- Abstraktes Modell soll Großteil der Interaktionen zwischen den wichtigsten Komponenten ausdrücken
- Abstraktes Modell in eigenes Modul – spezialisierte, detaillierte Implementierungsklassen verbleiben in ihren eigenen Modulen, die durch ihre jeweilige Subdomäne festgelegt sind

6.6 Destillation für Strategic Design

Abstract Core (Abstrakter Kern)

ABSTRACT CORE

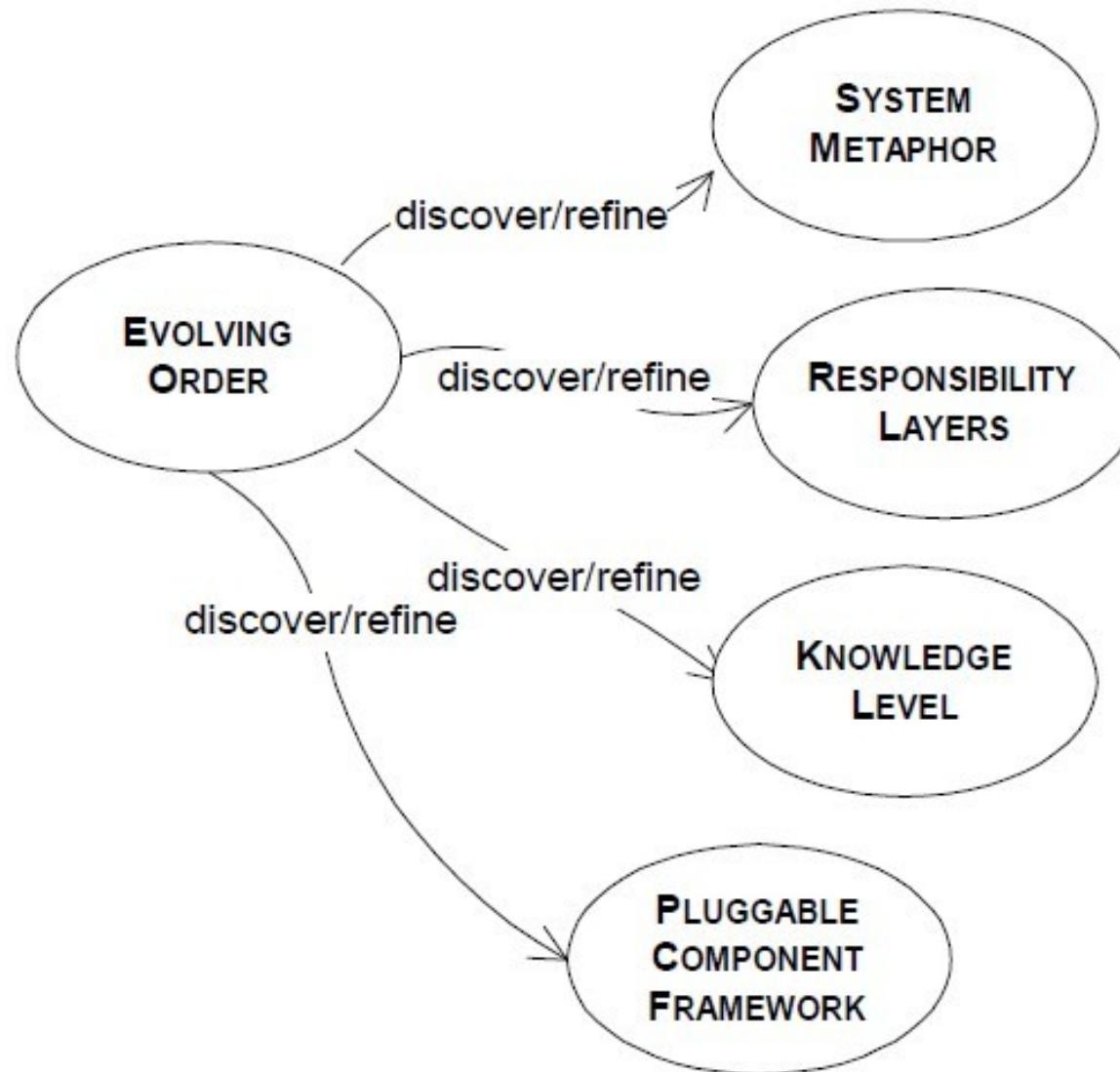


6.7 Large-Scale Structure für Strategic Design

Struktur im Großen für Strategic Design

- Ist eine Sprache mit der man das System in groben Zügen diskutieren und verstehen kann
- Einige übergeordnete Muster oder Regeln oder beides gemeinsam legen Leitlinien für den Entwurf des gesamten Systems fest
- Unterstützt Gestaltung und Verständnis
- Grundlegendes Verständnis der Verortung eines Teiles ohne das die detaillierte Kenntnis seiner Verantwortungen nötig ist

6.7 Large-Scale Structure für Strategic Design



6.7 Large-Scale Structure für Strategic Design

Evolving Order (Sich entwickelnde Ordnung)

- Kompromiss zwischen „*Entwurf ohne Regeln*“ - schwer als Ganzes zu verstehen und sehr schwer zu warten – und zu große Einschränkungen durch zu frühe Annahmen
- Die konzeptionelle *Large-Scale Structure* muss sich zusammen mit der Anwendung entwickeln können – möglicherweise in eine ganz andere Struktur
- Keine Behinderung der detaillierten Entwurfs- und Modellentscheidungen, die mit detailliertem Wissen gefällt werden

6.7 Large-Scale Structure für Strategic Design

Evolving Order (Sich entwickelnde Ordnung)

- Anwenden, wenn eine Struktur möglich ist, die das System wesentlich klarer macht ohne unnatürliche Einschränkungen aufzuerlegen
- Eine schlecht passende Struktur ist schlechter als gar keine → nicht auf Vollständigkeit zielen, besser minimale Menge, die das Problem löst (weniger ist mehr)

6.7 Large-Scale Structure für Strategic Design

System Metaphor (System-Metapher)

- Existiert eine konkrete Analogie zum System, die das Denken in eine geeignete Richtung führt, dann adaptiere diese Metapher als *Large-Scale Structure*
- Organisiere den Entwurf rings um diese Metapher herum und nehme sie in die *Ubiquitous Language* auf
- Die *System Metaphor* soll die Kommunikation über das System erleichtern und den Entwurf leiten
- Kann die Konsistenz in verschiedenen Teilen erhöhen, eventuell über *Bounded Contexts* hinweg

6.7 Large-Scale Structure für Strategic Design

System Metaphor (System-Metapher)

- **Aber:** Alle Metaphern nicht exakt → Gefahr fehlgeleitet zu werden
- Prüfe Metapher ständig darauf, ob sie noch zum System passt
- Wenn nötig/unpassend fallen lassen

6.7 Large-Scale Structure für Strategic Design

Responsibility Layers (Schichten nach Zuständigkeiten)

- Konzeptionelle Abhängigkeiten im Modell und unterschiedliche Änderungsgeschwindigkeiten und -quellen der verschiedenen Teile der Domäne betrachten – lassen sich dabei natürliche Schichten in der Domäne identifizieren → Entwurf als grobe und abstrakte Verantwortlichkeiten
- Verantwortlichkeiten sollten auf grober Ebene Zweck und Entwurf des Systems verdeutlichen
- Modell überarbeiten, so dass Verantwortlichkeiten der einzelnen Domänenobjekte, ***Aggregates*** und Module genau in die Verantwortung einer Schicht passen

6.7 Large-Scale Structure für Strategic Design

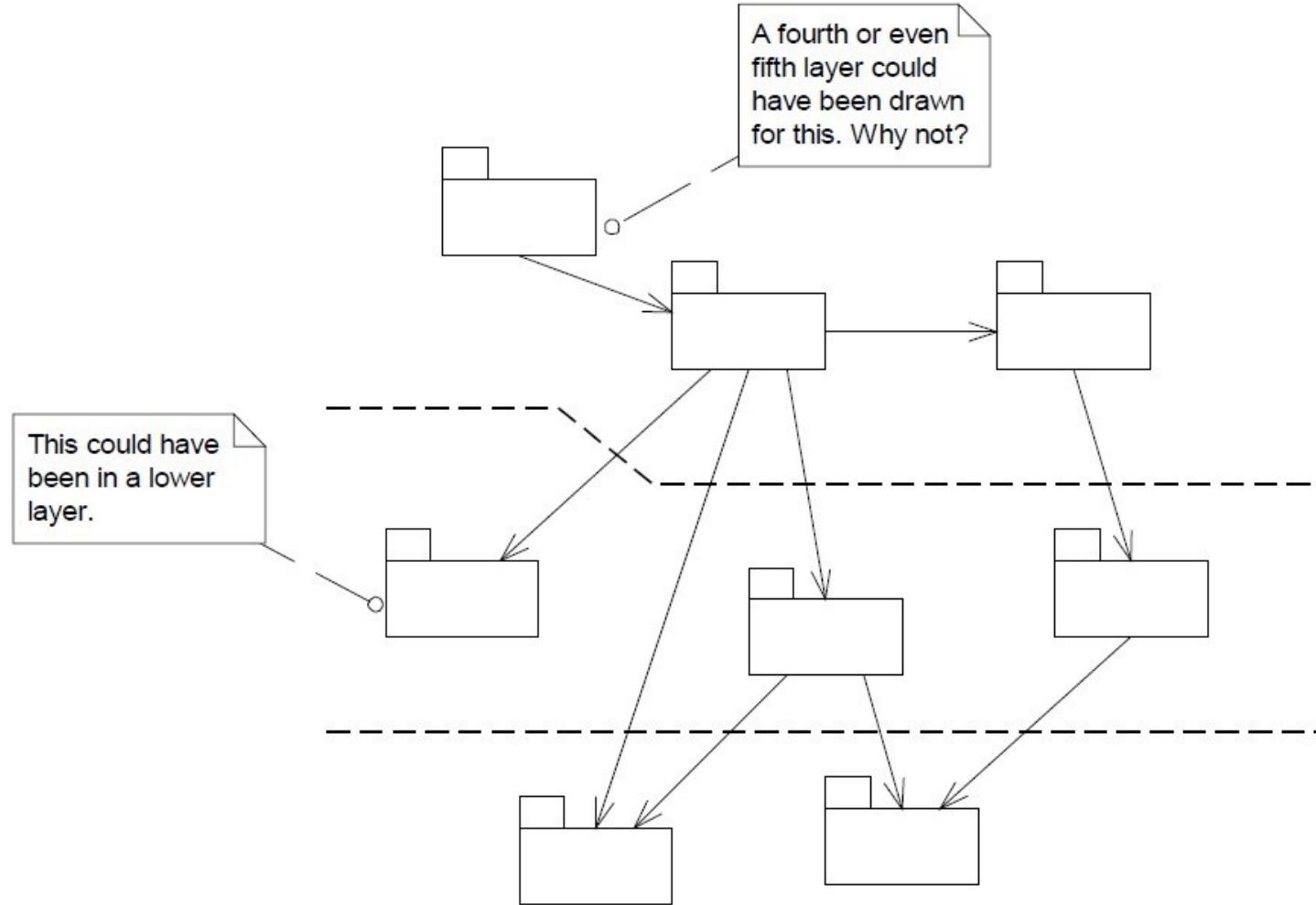


Figure 16. 1 Ad hoc layering: What are these packages about?

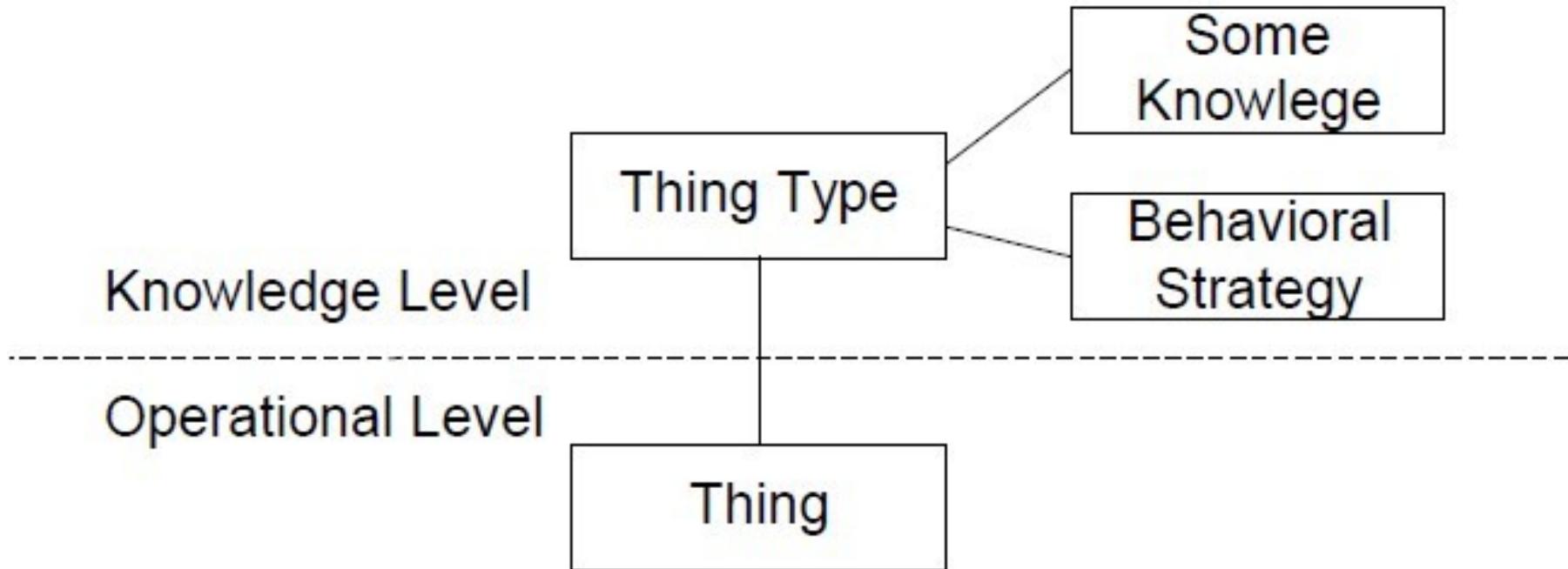
6.7 Large-Scale Structure für Strategic Design

Knowledge Level (Wissensebene)

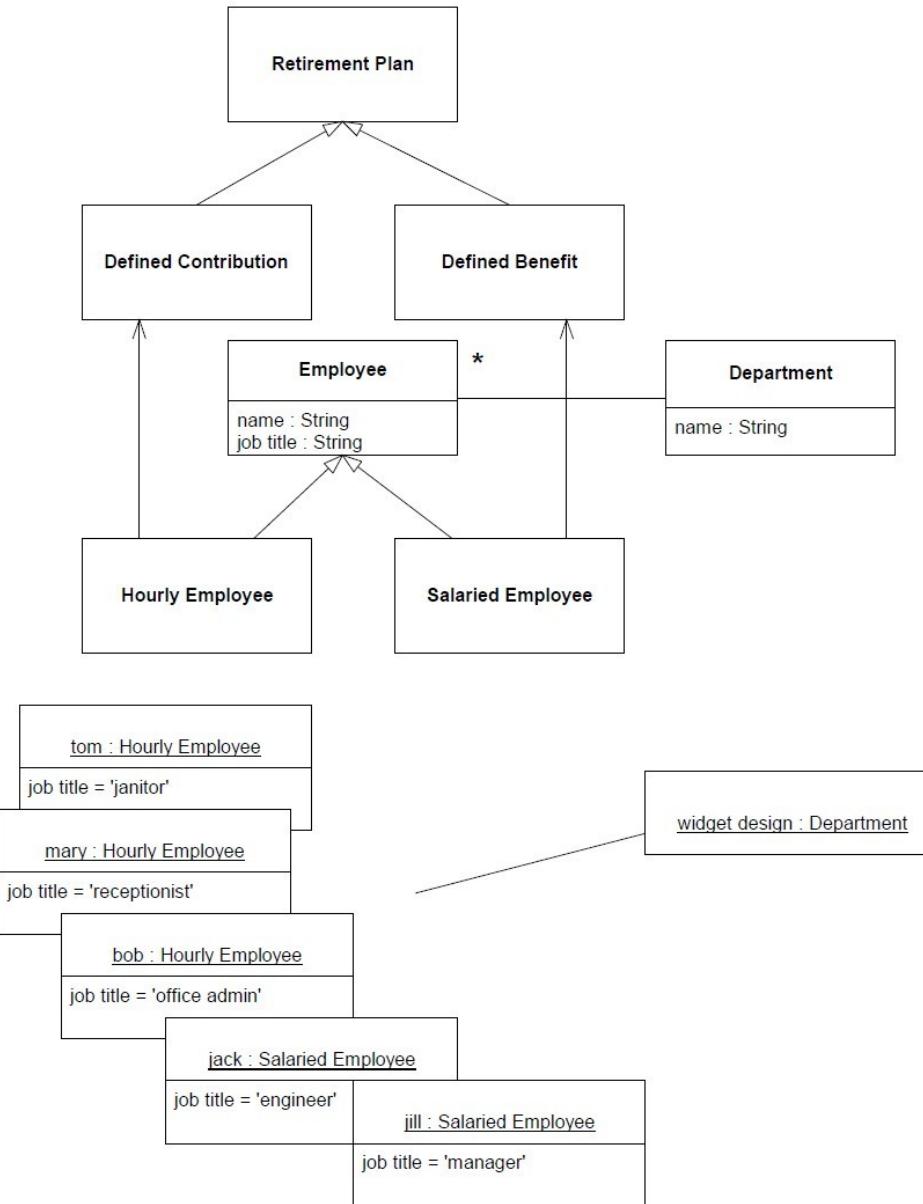
- Ganz allgemein – Gruppe von Objekten, die beschreibt, wie sich eine andere Gruppe von Objekten verhält
- Gruppe von Objekten, die Struktur und Verhalten des Basismodells beschreiben und einschränken können – diese zwei Ebenen trennen halten, die eine sehr konkret und die andere spiegelt Wissen und Regeln wider, die angepasst werden können

(M. Fowler, 1997, „*Analysis Design Patterns: Reusable Object Models*“)

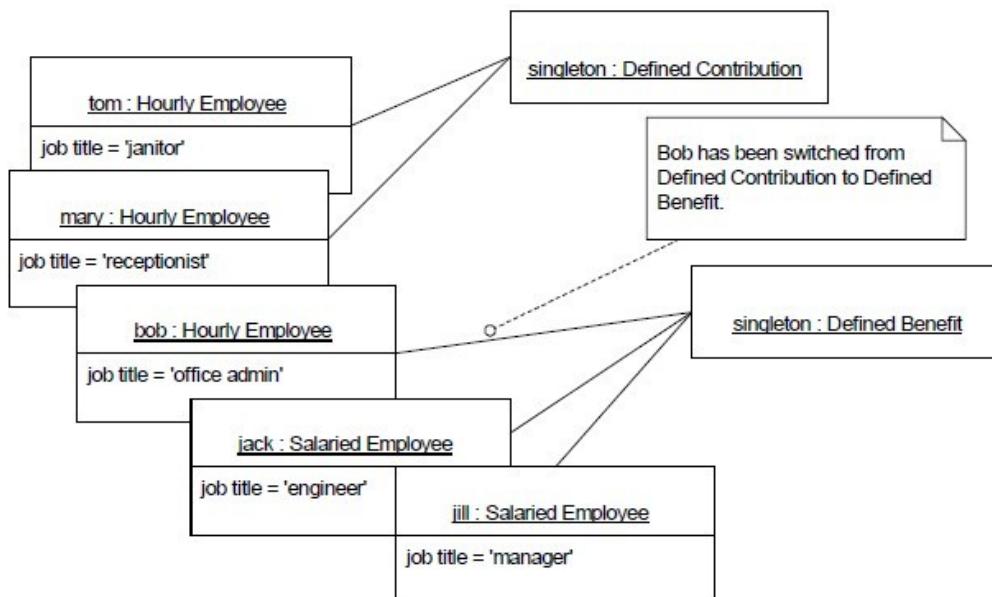
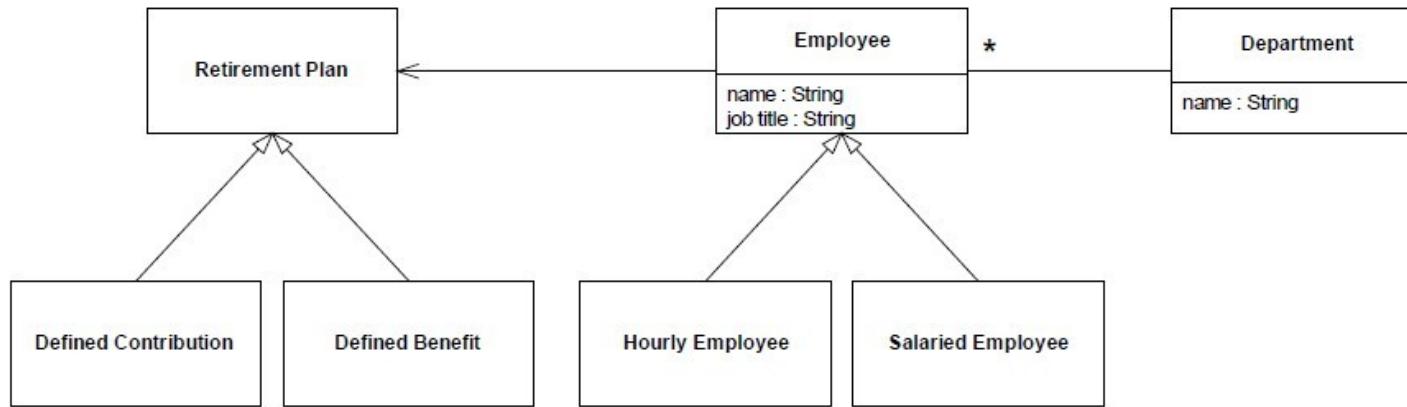
6.7 Large-Scale Structure für Strategic Design



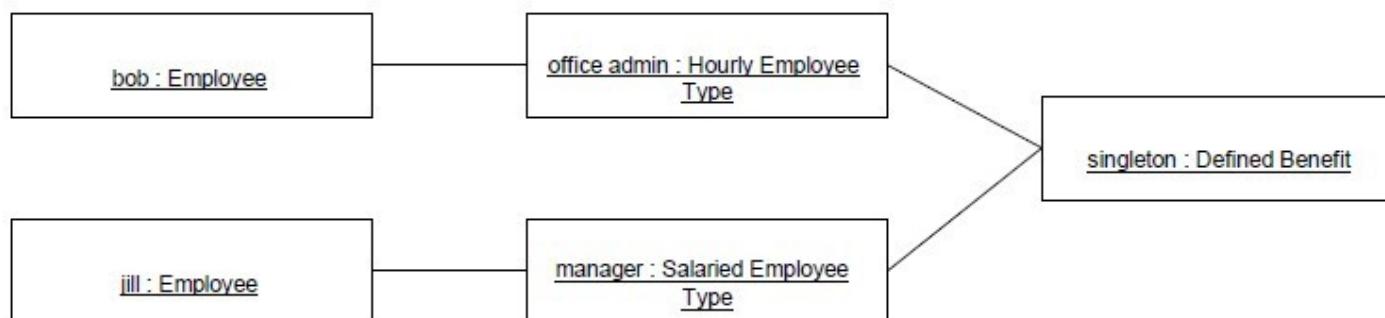
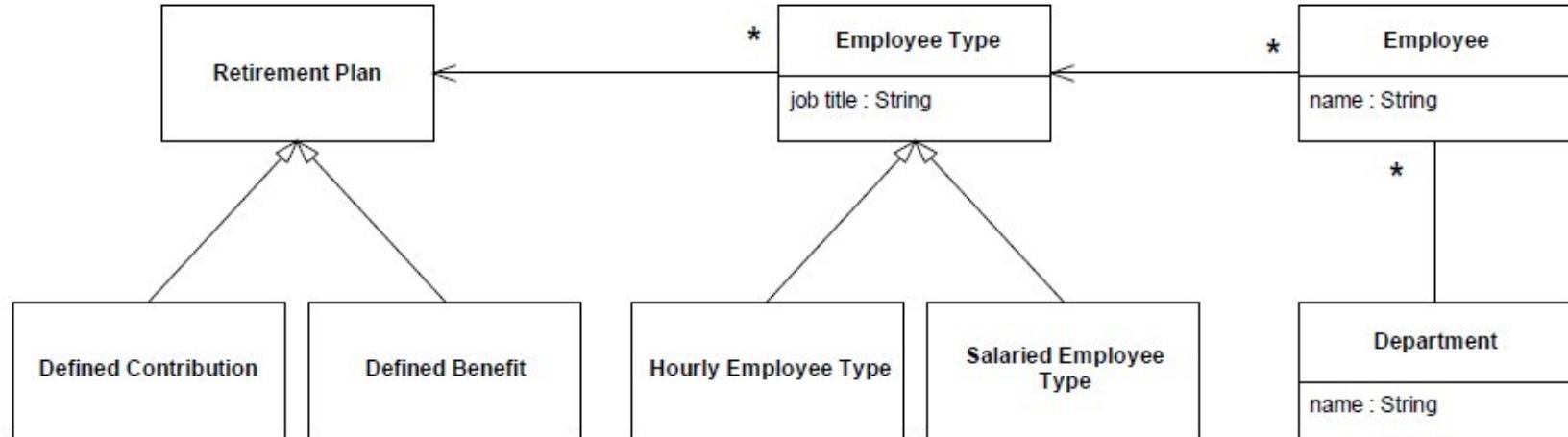
6.7 Large-Scale Structure für Strategic Design



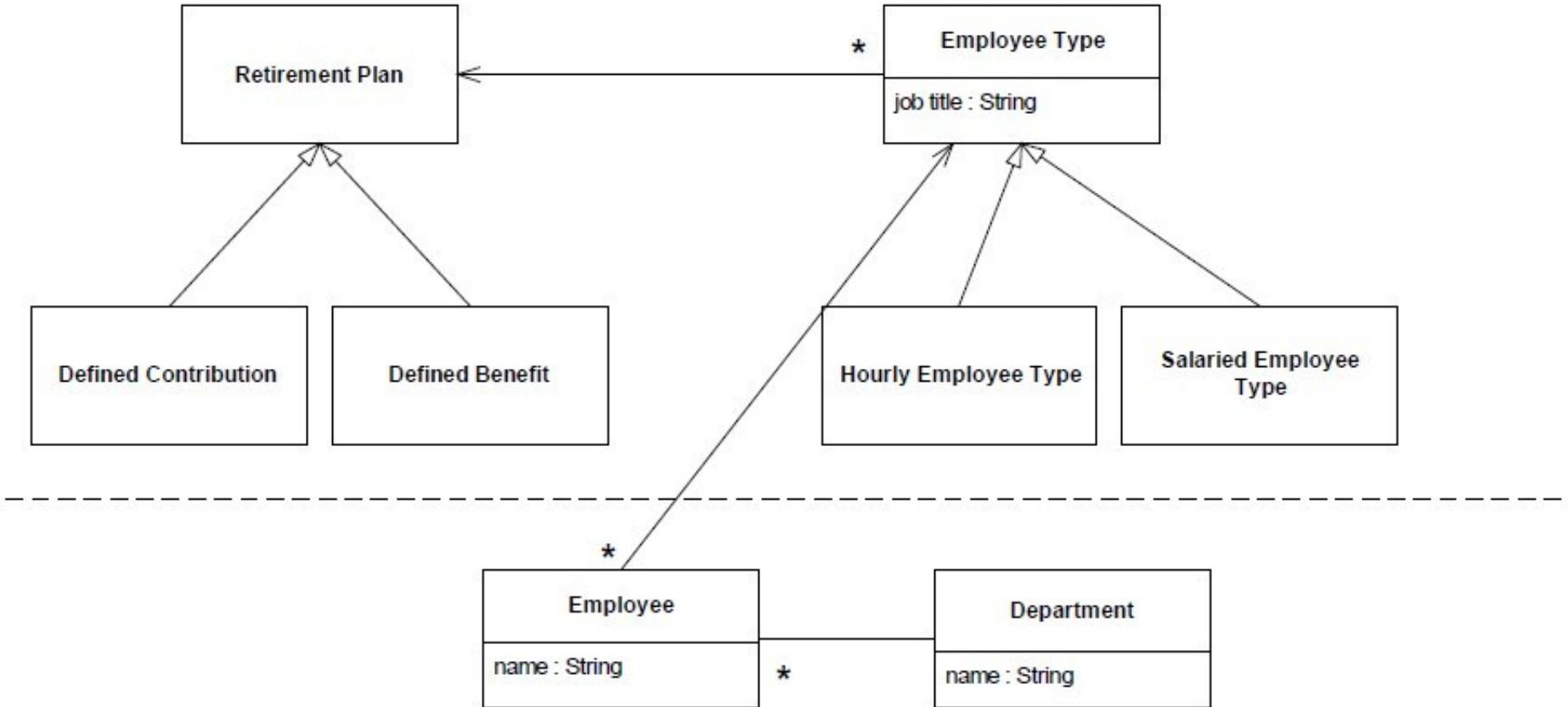
6.7 Large-Scale Structure für Strategic Design



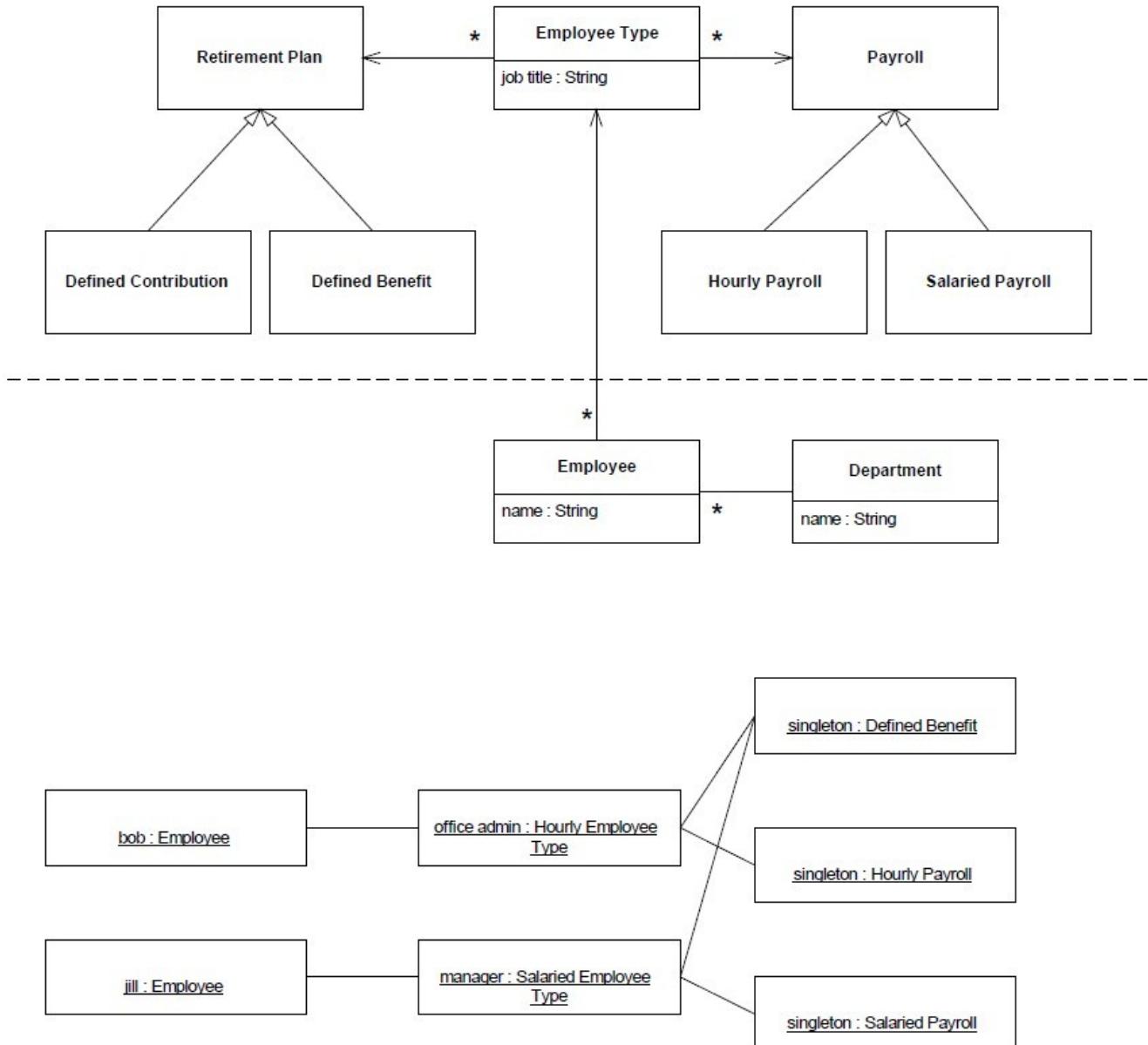
6.7 Large-Scale Structure für Strategic Design



6.7 Large-Scale Structure für Strategic Design



6.7 Large-Scale Structure für Strategic Design



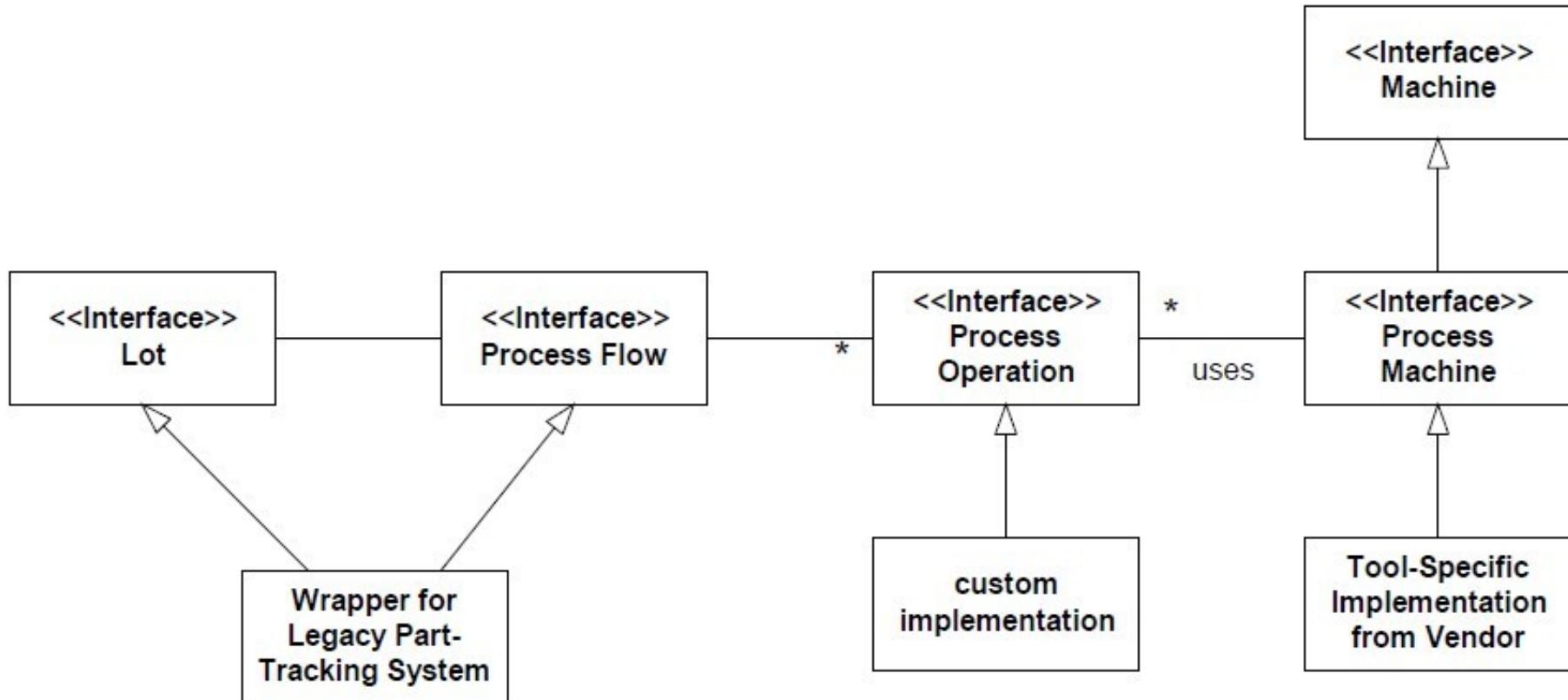
6.7 Large-Scale Structure für Strategic Design

Pluggable Component Framework *(Zusammensteckbares Komponenten-Framework)*

- Kommt erst ins Spiel, wenn bereits einige Anwendungen in der Domäne implementiert worden sind
- Destillieren eines abstrakten Kerns von Interaktionen und Schnittstellen → Entwickeln eines Frameworks, das den Austausch der Implementierung dieser Schnittstellen auszutauschen
- Jede Anwendung kann diese Komponenten verwenden, so lange sie ausschließlich über die abstrakten Schnittstellen des abstrakten Kerns arbeitet

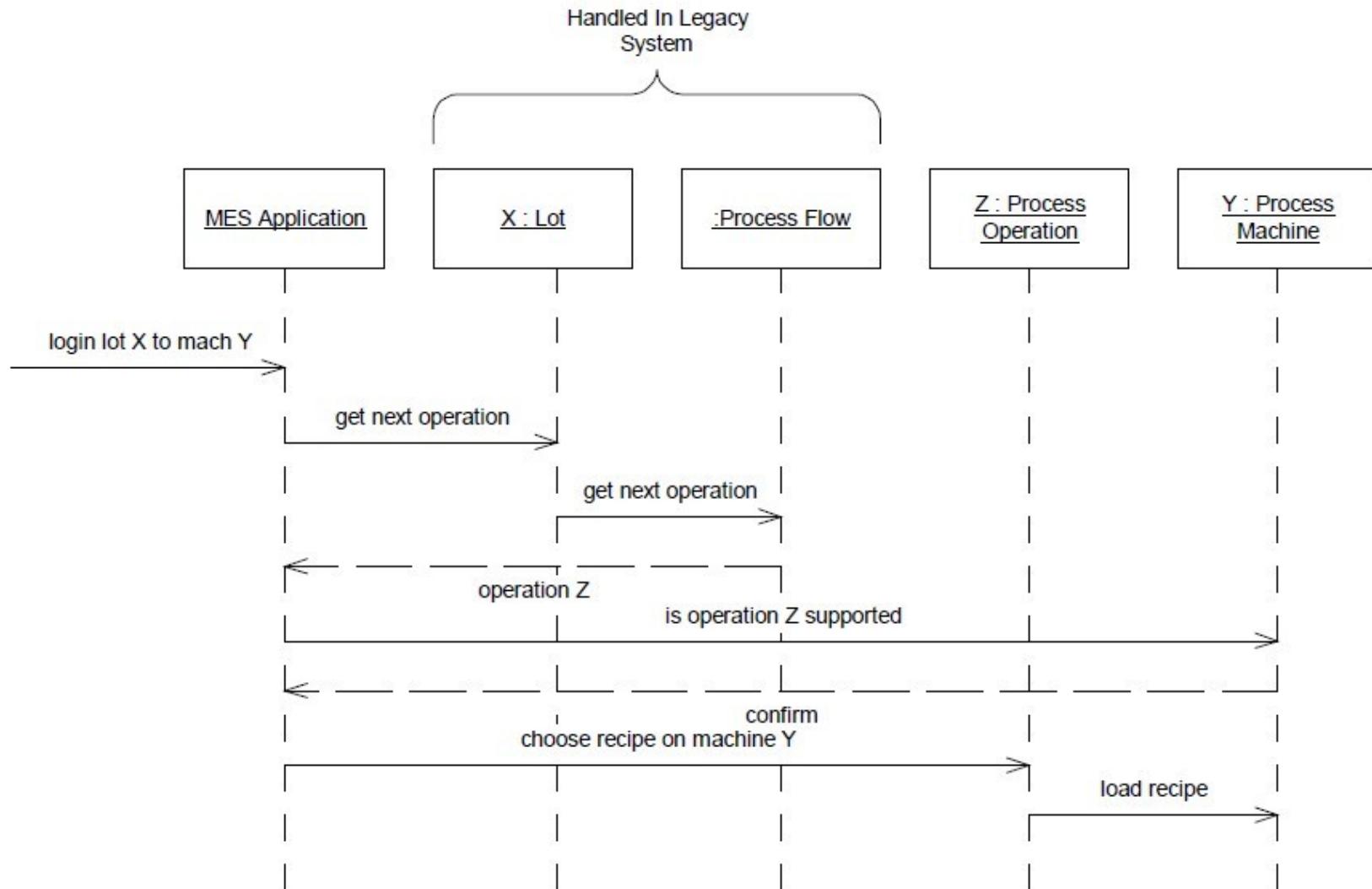
6.7 Large-Scale Structure für Strategic Design

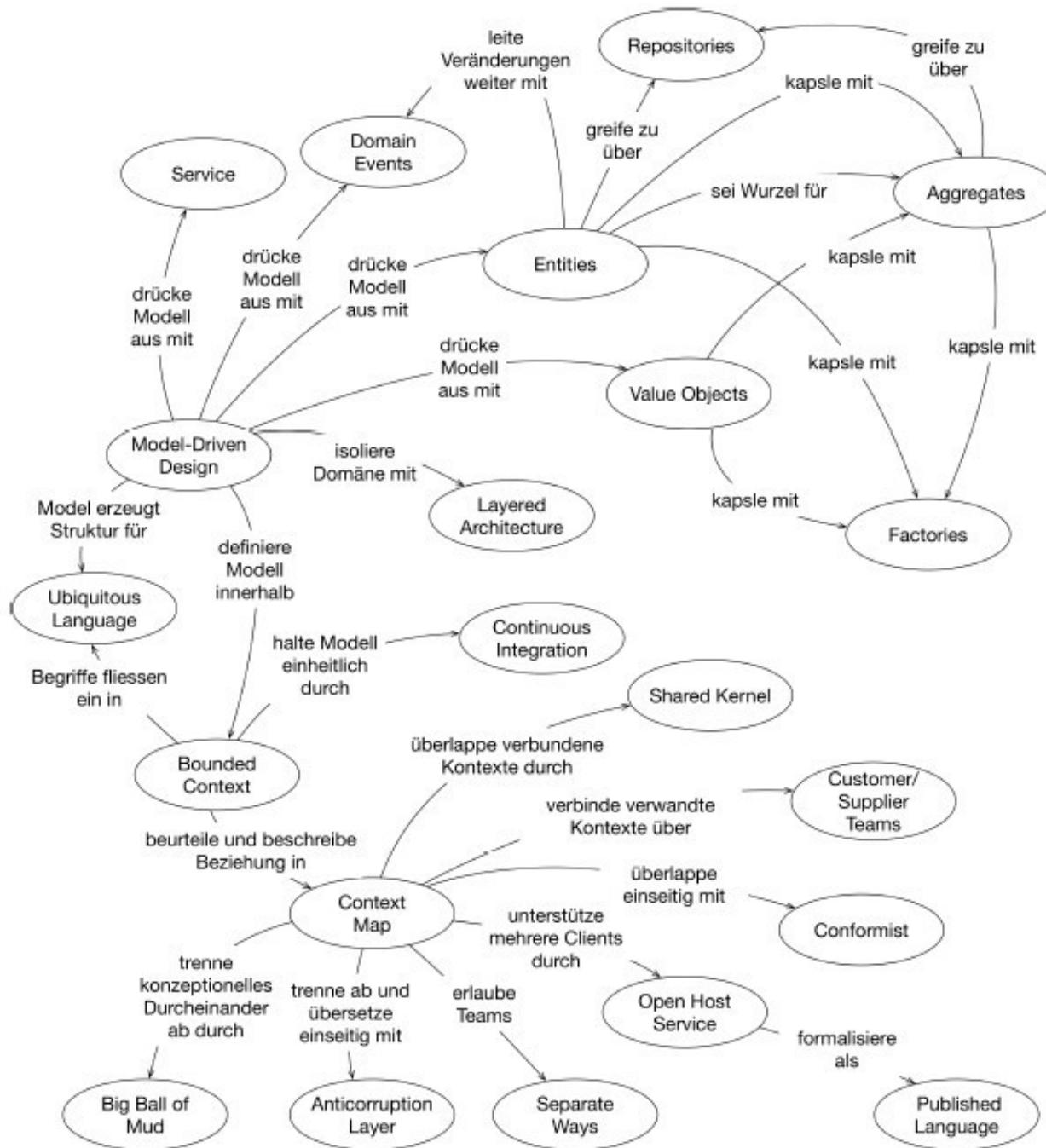
A highly simplified subset of the CIM interfaces



6.7 Large-Scale Structure für Strategic Design

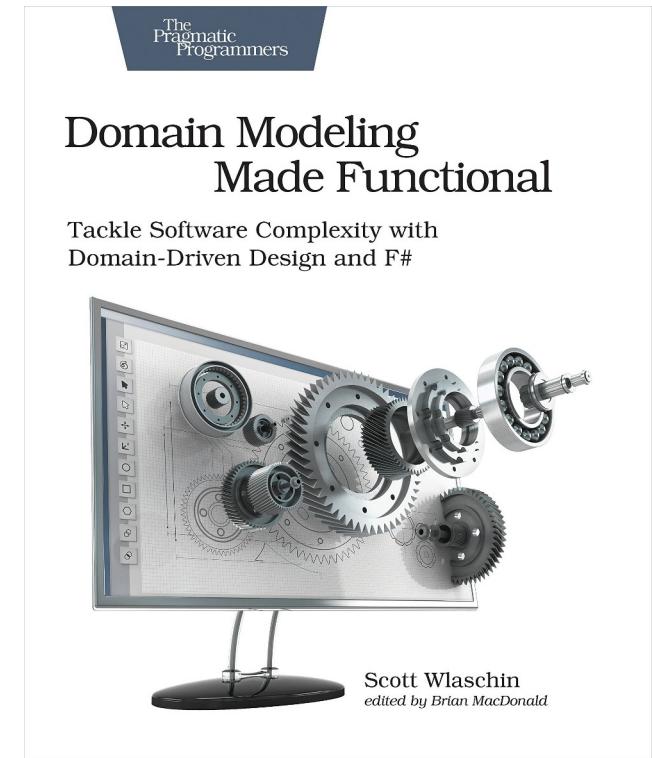
User places lot in next machine and logs move into computer





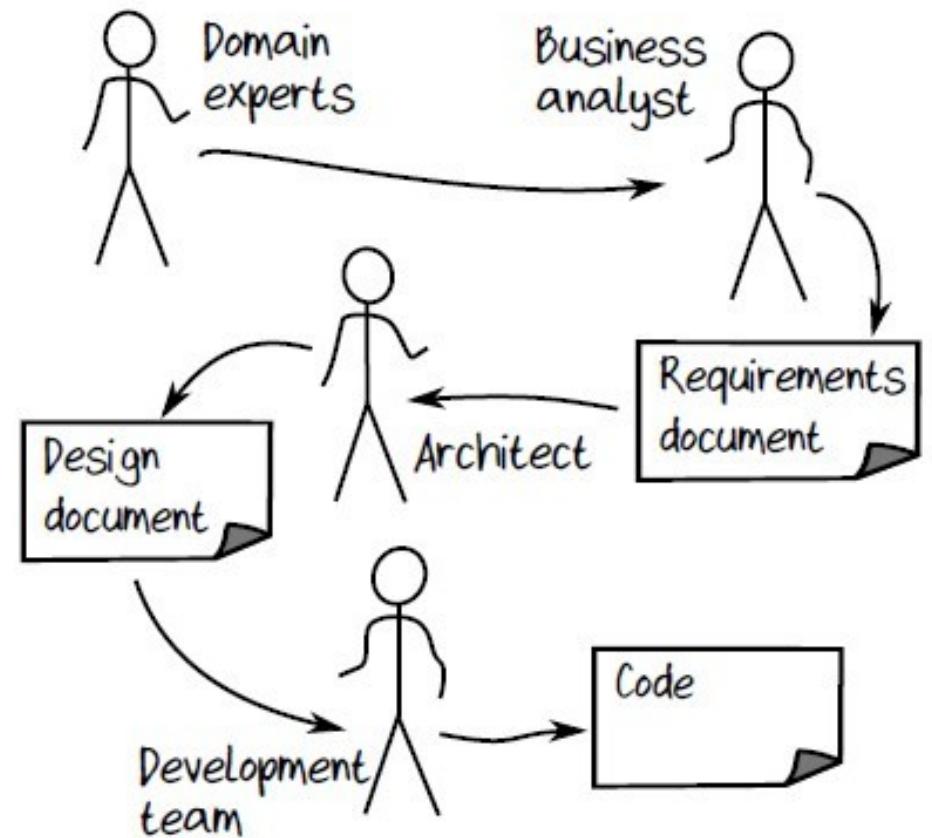
6.8 DDD-Praxis

- Inspiriert und basierend auf „Domain Modeling Made Functional“ von Scott Wlaschin
- „Mainstream“-Art DDD zu betreiben:
 - Definition von Datenstrukturen und von
 - Funktionen die auf diesen Datenstrukturen arbeiten



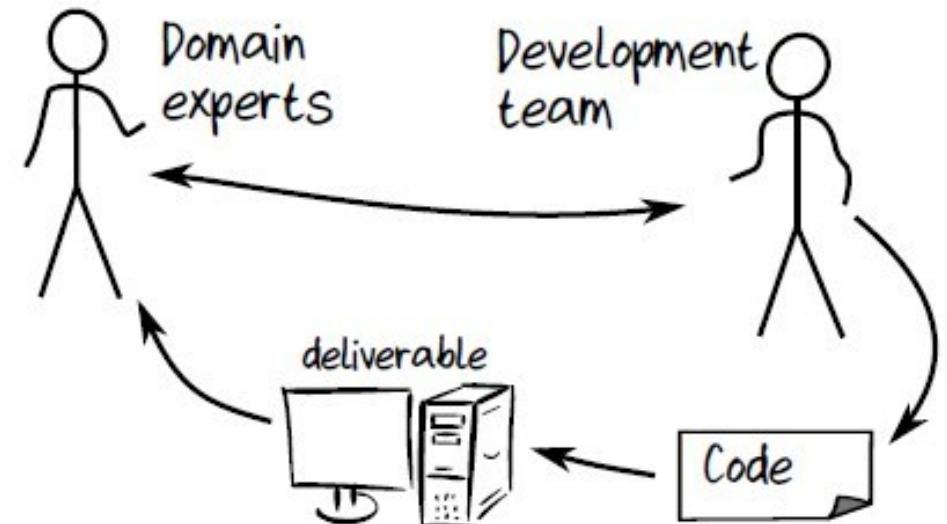
6.8.1 Übergang zu DDD

- Bisheriger Ablauf oft wie „Stille Post“ mit fraglichem Ausgang
- Deshalb einen Teil der Kette entfernen und Modellierung gemeinsam durch die beiden wichtigsten Seiten des Prozesses



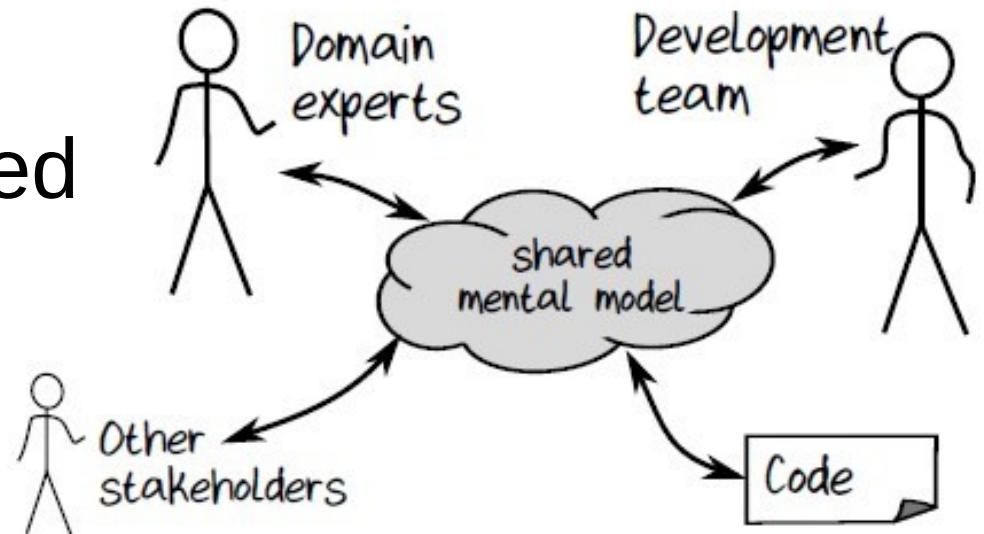
6.8.1 Übergang zu DDD

- Evaluierung durch ständige Kontrolle des Domänen-Expertens – agile Softwareentwicklung
- Softwareentwickler tritt als eine Art „Übersetzer“ des mentalen Modells des Domainen-Expertens auf – Problematisch:
 - Kleine Details könnten untergehen bzw. nicht in Code transformiert werden.
 - Was ist, wenn in der weiteren Entwicklung kein Domänen-Experte mehr zur Verfügung steht?



6.8.1 Übergang zu DDD

- Ansatz mit einem gemeinsamen „Shared Mental Modell“
- Gemeinsames Modell wird nicht nur von Domänenexperten, Entwicklungsteam und Stakeholders geteilt, sondern „**auch vom Code**“ - in „**Code gegossenes Modell**“
→ ***Das ist das Ziel von DDD!***



6.8.1 Übergang zu DDD

- Ziel ist (wie immer) beim Ausrichten der Entwicklung auf die Business-Domäne:
 - *Kürzere Time-To-Market*
 - *Mehr Mehrwert* für die Domäne
 - *Weniger Verschwendungen* – an Ressourcen und Zeit
 - Weniger Missverständnisse - und wenn, dann früher entdecken
 - Erkennen der „hochwertigeren Ziele“ und fokussierte Investition in diese statt in „niedrigwertigere“
 - *Leichtere Wartung und Weiterentwicklung*

6.8.1 Übergang zu DDD

- Wie kommt man zum *Shared Model* (alte Bekannte aus den vorherigen Vorlesungen tauchen wieder auf):
 - Fokus auf die *Business-Events* und *Workflows* – nicht auf die Datenstrukturen
 - Zerlegen der Problem-Domäne in *Sub-Domänen*
 - Entwickeln eines Modells für jede *Sub-Domäne*
 - Entwickeln einer gemeinsamen Sprache (*Ubiquitous Language*)

6.8.2 Verstehen der Business-Domäne

- Warum Fokus auf Business-Events?
 - Daten bilden Objekte der Domäne ab, aber ...
 - Wert wird durch die irgendwie geartete Transformation bzw. die Transformationskette geschöpft – wichtig diese zu verstehen
 - Trigger von außer- oder innerhalb stoßen diese Prozesse an – *Business/Domänen-Events*
 - z.B. „*Platzierung einer Bestellung*“
 - *Domänen-Events* sind immer in der Vergangenheit geschrieben und unveränderlich – sie sind ein Fakt der Vergangenheit

6.8.2 Verstehen der Business-Domäne

- ***Event-Storming*** um die Domäne zu erkunden:
 - Kollaborativer Prozess, um die *Domänen-Events* und *Workflows* zu identifizieren
 - Workshop aller Beteiligten, die die unterschiedlichsten Aspekte der Domäne abdecken: „anyone who has questions and anyone who has answers“
 - Viel Platz für Klebezettel und viele Klebezettel auf die die *Domänen-Events* und *Workflows* geschrieben werden – interaktives „Klebezettel-Puzzel“ als Abbild der Domäne

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

“We’re a tiny company that manufactures parts for other companies: widgets, gizmos, and the like. We’ve been growing quite fast, and our current processes are not able to keep up. Right now, everything we do is paper-based, and we’d like to computerize all that so that our staff can handle larger volumes of orders. In particular, we’d like to have a self-service website so that customers can do some tasks themselves. Things like placing an order, checking order status, and so on.”

You: “Someone start by posting a business event!”

Ollie: “I’m Ollie from the order-taking department. Mostly we deal with orders and quotes coming in.”

You: “What triggers this kind of work?”

Ollie: “When we get forms sent to us by the customer in the mail.”

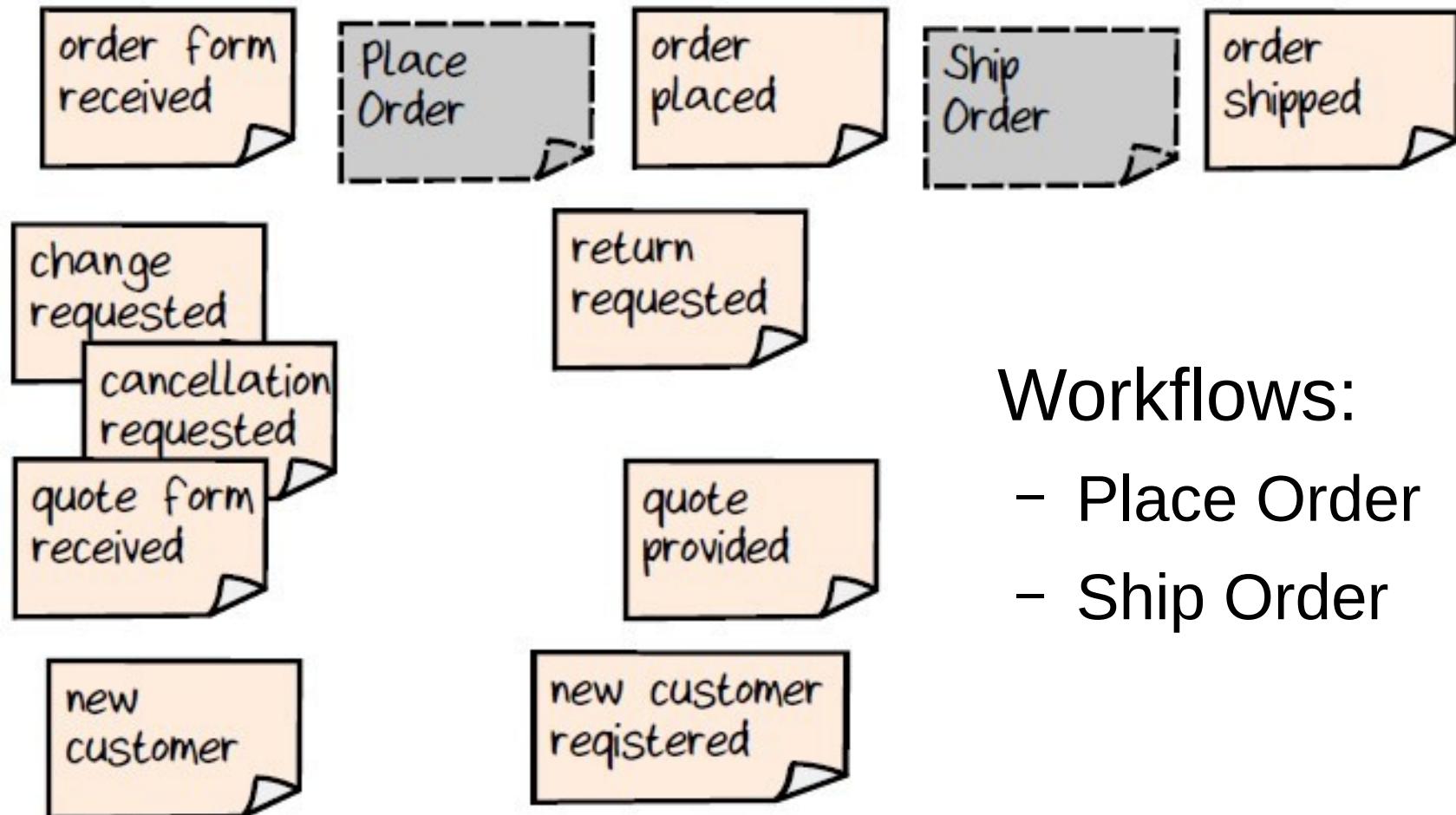
You: “So the events would be something like ‘Order form received’ and ‘Quote form received’?”

.....

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

- Klebezettel mit Events:
 - Order form received
 - Order placed
 - Order shipped
 - Order changed requested
 - Return requested
 - Quote form received
 - Quote provided
 - New customer request received
 - New customer registered

6.8.2 Verstehen der Business-Domäne – Beispiel Bestell-System



Workflows:

- Place Order
- Ship Order

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Aspekte des *Event-Storming*:

- *Shared Model* der Business-Domäne
 - Neben der Identifikation der Events, sorgt gemeinsame Interaktion für ein gemeinsames Verständnis
 - Eventuell nur eingeschränkt bekannte Aspekte werden allen offenbar
- Wahrnehmung aller Teams und aller Teammitglieder
 - Blick über den Tellerrand des eigenen Teams hinaus
 - Jeder Stakeholder bringt seine Aspekte mit ein

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Aspekte des *Event-Storming*:

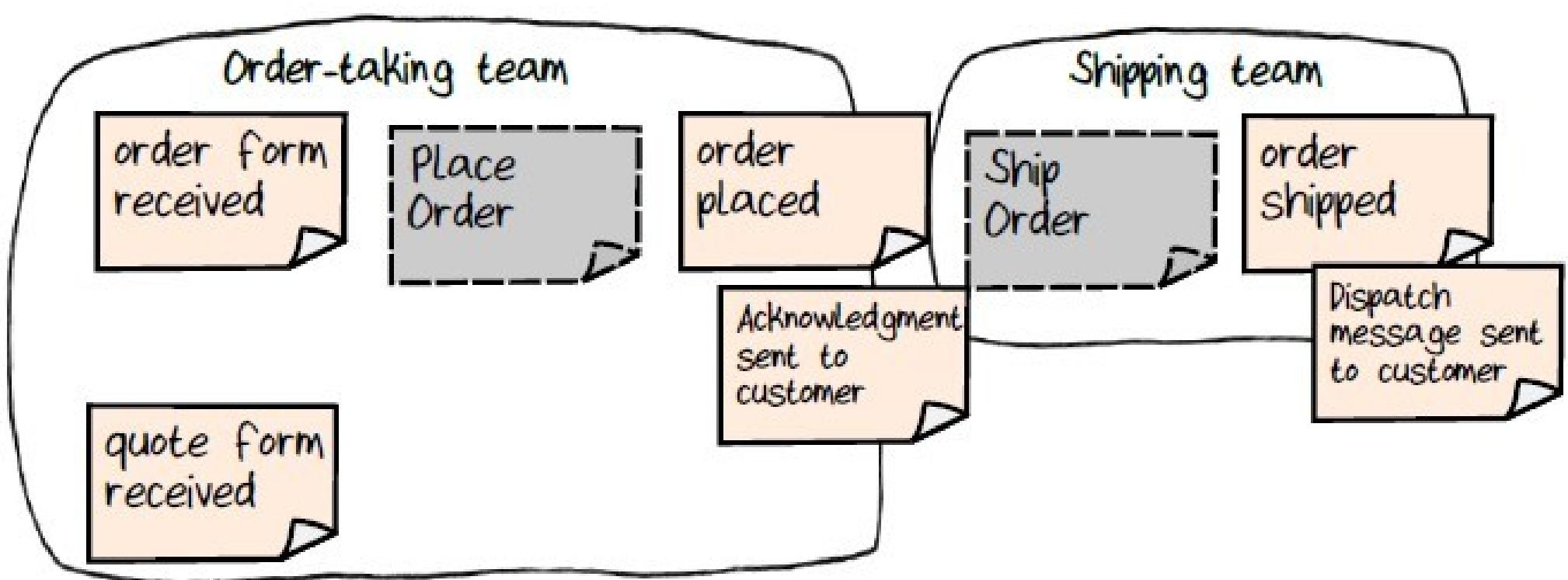
- Lücken in den Requirements aufdecken
 - Durch die Übersicht und Diskussion lassen sich fehlende Requirements oft leichter entdecken
 - Wenn es keine Antworten auf aufgeworfene Fragen gibt, gehören diese Fragen auf einen Klebezettel an der Wand – Möglichkeit mehr über die Domäne zu erfahren
 - Üblicherweise sind Requirements am Anfang unscharf – das sollte dann auch besser so dokumentiert werden – als Reminder

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Aspekte des *Event-Storming*:

- Verbindung zwischen den Teams erkennen
 - Sind die Events in einer Timeline angeordnet, werden die Übergänge zwischen den Teams klarer
 - Output eines Teams wird zum Input des anderen
 - Z.B. wenn das Bestell-Team eine Bestellung entgegen genommen und bearbeitet hat, signalisiert sie das mit dem Event „Order placed“, das die Eingabe für das Versand- und Rechnungs-Team darstellt

6.8.2 Verstehen der Business-Domäne – Beispiel Bestell-System



6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Aspekte des *Event-Storming*:

- Bewußtsein für die Reporting-Requirements
 - Reporting ist ein wichtiger Teil jeder Domäne – verstehen dessen, was in der Vergangenheit passiert ist
 - Reporting und Read-Only-Modelle wie View-Modelle und UI's sollten Teil des *Event-Storming* sein

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Begriffliches:

- *Szenario*
 - Ähnlich agiler *Story*
 - Beschreibt ein Ziel, das der Kunde bzw. Nutzer erreichen möchte
 - Kunden/Nutzer-orientiert
- *Use-Case*:
 - Detailliertere Beschreibung als ein *Szenario*
 - Beschreibung der Interaktion in allgemeiner Sprache, d.h. ohne technische Ausdrücken
 - Kunden/Nutzer-orientiert

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

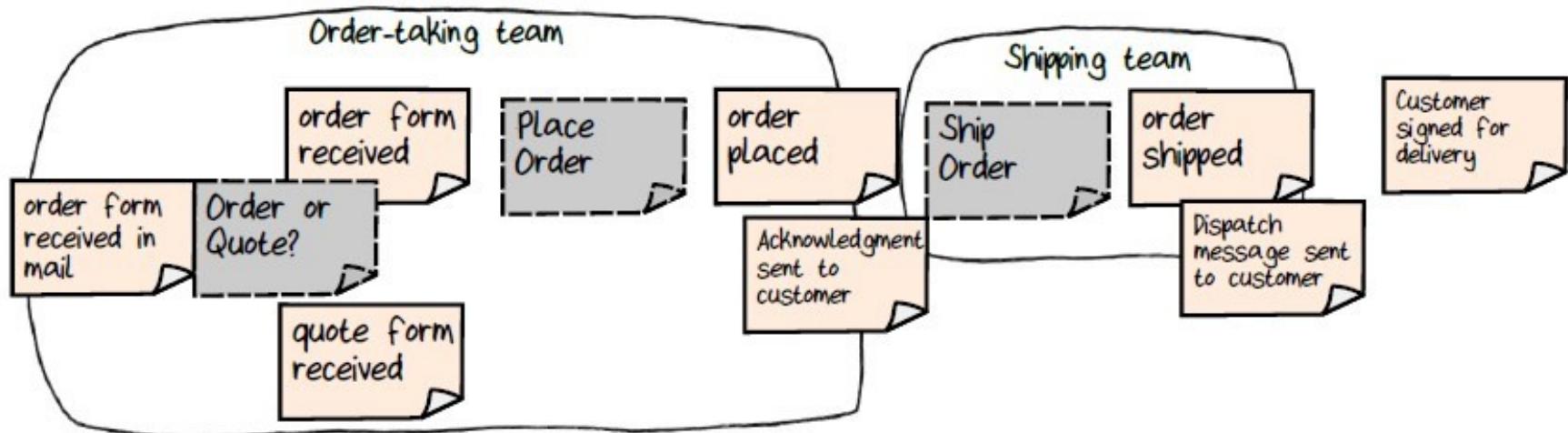
Begriffliches:

- *Business-Prozess*:
 - Ähnlich einem *Szenario*, beschreibt aber ein Ziel, das die Firma erreichen möchte
 - Business-orientiert
- *Workflow*:
 - Detaillierte Beschreibung eines Teils eines Business-Prozesses
 - Exakte Schritte, die ein Angestellter bzw. eine Software-Komponente ausführen muss, um ein Geschäftsziel oder -teilziel zu erreichen
 - Geht ein Workflow über die Grenzen eines Teams hinaus, so wird er zerlegt in eine Reihe anderweitig koordinierte (Teil-)Work-flows

6.8.2 Verstehen der Business-Domäne – Beispiel Bestell-System

Verschieben der Events an die Grenzen

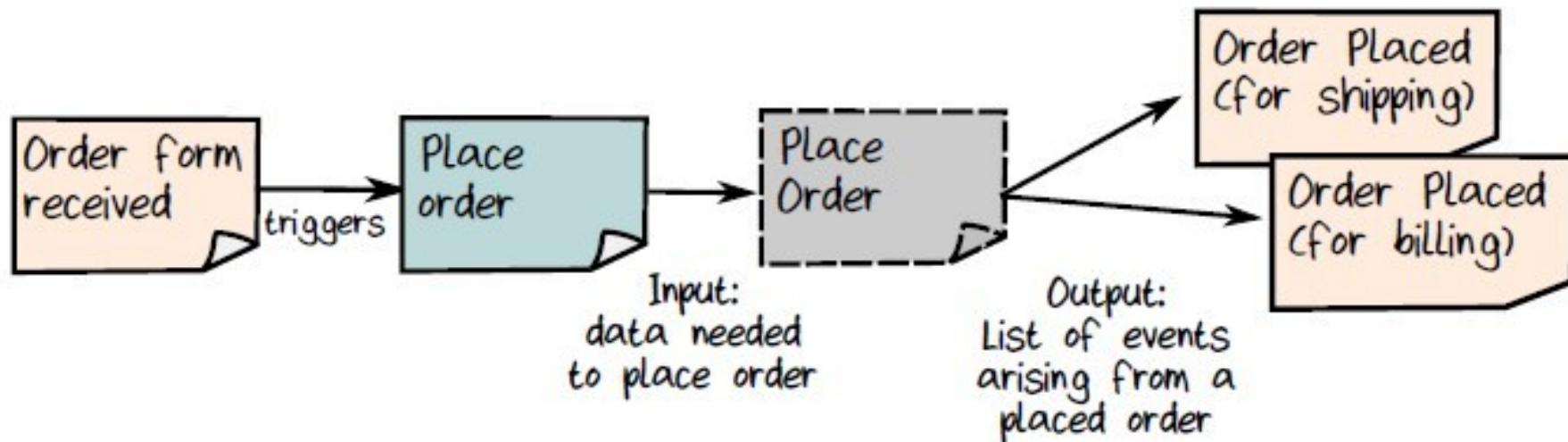
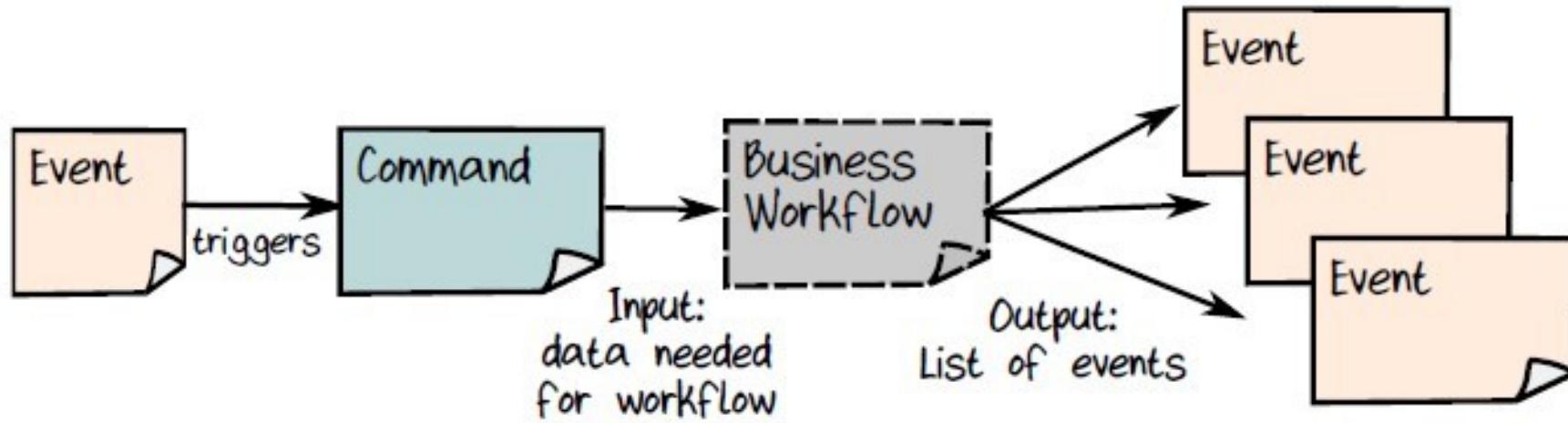
- Frage welches Event kommt am „weitesten links“ (am frühesten) am „weitesten rechts“ (am spätesten im Ablauf)
- Grenzen der Bereiche identifizieren
- Erkennen von noch nicht identifizierten Requirements



6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

- Was lösen die Events aus und wie werden die Events ausgelöst?
- *Commands*, z.B:
 - Command: “Place an order” → Domain Event: “Order placed.”
 - Command: “Send a shipment to customer X” → Domain Event: “Shipment sent”
 - ...
 - Triggern *Commands* und werden von *Commands* ausgelöst
 - Kein *Command* im Sinne der OOP-Pattern
- Event triggert *Command*, das wiederum einen Workflow initialisiert und startet, der wiederum nach oder während seines Ablaufes eine Reihe von Events initiiert
- Nicht alle Events sind mit *Commands* initiiert, manche werden von Schedulern oder Monitoring-Systemen generiert

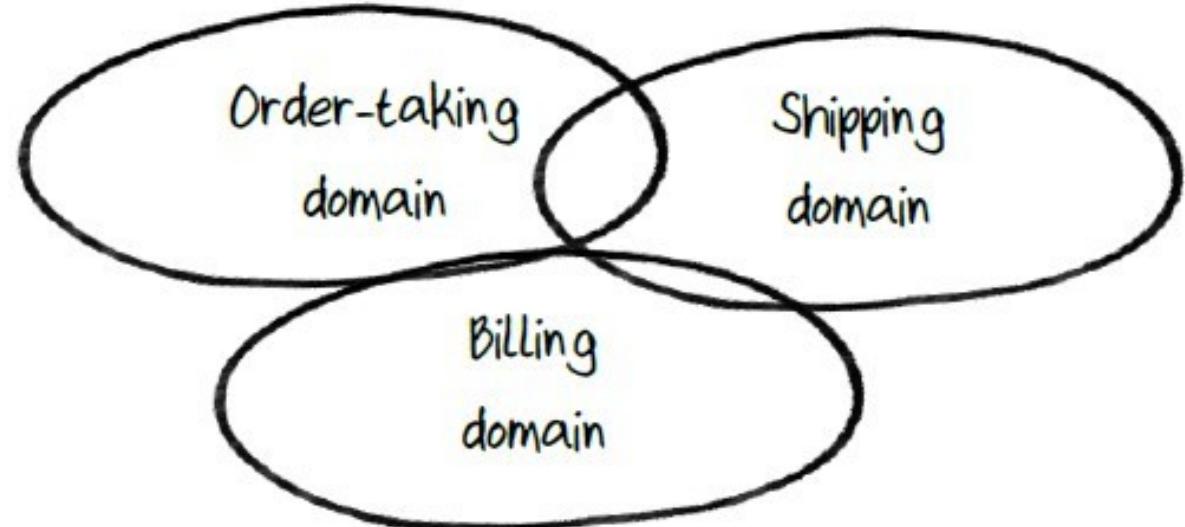
6.8.2 Verstehen der Business-Domäne – Beispiel Bestell-System



6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Unterteilen der Domäne in einzelne, kleinere Sub-Domänen:

- „Domäne“ als „Bereich kohärenten Wissens“
- Kann teilweise unscharf begrenzt sein und überlappen
- Im Beispiel:
 - *Order-taking*
 - *Shipping*
 - *Billing*

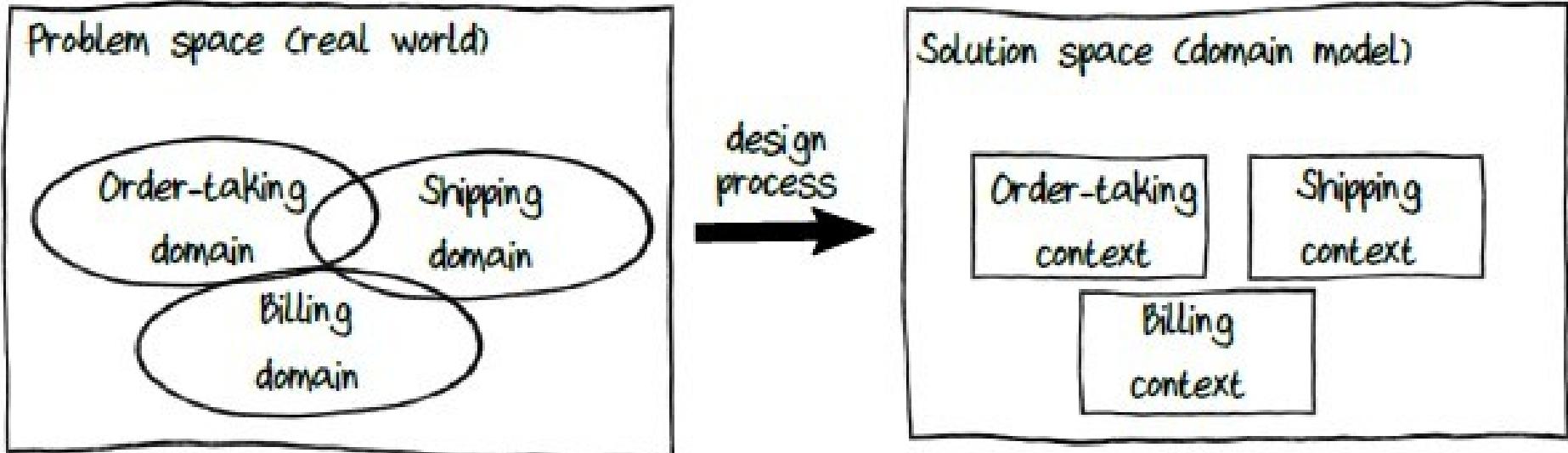


6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

„*Problem Space*“ vs. „*Solution Space*“:

- Domäne zu verstehen, bedeutet noch lange nicht, die Lösung zu haben – eventuell ist die wirkliche Domäne zu komplex
- Wir müssen die Aspekte modellieren, die wirklich relevant sind, um das geplante Ziel zu erreichen
- Unterscheidung von *Problem Space* und *Solution Space*
- Übertragung von Problem- zum Lösungsraum ist schließlich der Design-Prozess

6.8.2 Verstehen der Business-Domäne – Beispiel Bestell-System



- Sub-Domänen im Problemraum entsprechen jeweils einem *Bounded Context* im Lösungsraum des DDD
- Jeder Bounded Context stellt ein kleines Domänenmodell dar

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Warum Bezeichnung als *Bounded Context* und nicht als Sub-System?

- Hilft sich auf das Wesentliche beim Design zu fokusieren – *Context* und *Grenzen* des Contexts
- *Context*
 - Repräsentiert einen Bereich spezialisierten Wissens innerhalb der Anwendung
 - Gleiche Sprache, kohärentes einheitliches Design
 - Gültigkeit von Aussagen nur im Kontext

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Warum Bezeichnung als *Bounded Context* und nicht als Sub-System?

- *Grenzen bzw. begrenzt:*
 - In der realen Welt oft unscharfe Grenzen – in der Software wollen wir Subsysteme entkoppeln (unabhängige Entwicklung der einzelnen Systeme)
 - Verwenden explizite Schnittstellen (API), kein Shared Code, d.h. wir bilden den Anwendungsraum nie exakt ab → Kompromiss für weniger Komplexität und weniger Wartungsaufwand
- D.h. eine Domäne im Anwendungs-/Problemraum und ein *Bounded Context* im Lösungsraum haben nie eine 1:1-Entsprechung (z.B. Domäne ist in mehrere BC's aufgeteilt)

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Warum Bezeichnung als *Bounded Context* und nicht als Sub-System?

- Jeder *BC* muß nach der Aufteilung eine klare Verantwortlichkeit haben (→ Klare Schnittstelle
→ Implementierung in einer eigenen Softwarekomponente möglich, wie DLL, (Micro-)Service oder simpler Namespace)

D.h. es ist wichtig die richtige Aufteilung zu machen

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Wie gelingt die richtige Aufteilung?

- Große Herausforderung bei DDD – mehr Kunst bzw. Handwerk als Wissenschaft
- Einige Leitlinien bzw. Hilfen:
 - Den Domänenexperten gut zuhören
 - Die existierenden Team- und Bereichsgrenzen beachten bzw. untersuchen → Organisation häufig entlang sinnvoller Grenzen – nicht immer!
 - Bounded-Teil von *BC* ist wichtig – kein Scope-Creep zulassen, ist die Grenze zu vage ist es keine!

6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

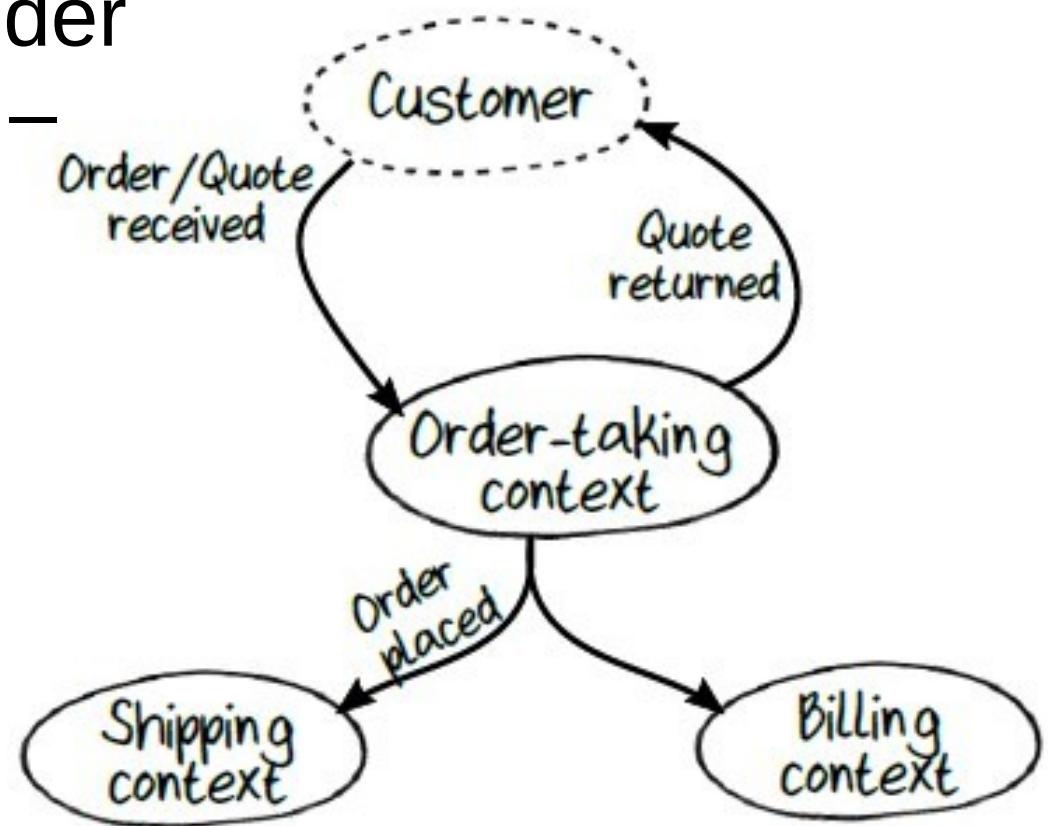
Wie gelingt die richtige Aufteilung?

- Einige Leitlinien bzw. Hilfen:
 - So designen, dass möglichst unabhängig von einander gearbeitet werden kann → aufteilen, wenn nötig
 - Design so gestalten, dass Workflows möglichst reibungsfrei laufen können
 - Falls ein Workflow mit vielen BC's interagieren muss, wird er höchstwahrscheinlich öfter blockiert und behindert
 - Redesign um das Design für den Workflow geeigneter zu machen, auch wenn es unschöner wird (Domänennutzen steht im Vordergrund – vor schönem Design)

6.8.2 Verstehen der Business-Domäne – Beispiel Bestell-System

Context Maps

- Interaktionsdiagramm der identifizierten BC's – das „Big Picture“
- Keine Details – nur Austausch
- Verhältnisse *Downstream - Upstream*



6.8.2 Verstehen der Business-Domäne – Beispiel *Bestell-System*

Context Maps

- *Downstream – Upstream*
 - Einigen auf gemeinsames Austauschformat
 - Upstream hat mehr Einfluss – außer Downstream ist ein Legacy-Prozess und man muss Upstream anpassen
 - Passt nicht alles in eine Darstellung – mehrere Maps und eventuell hierarchische Maps
- Identifizieren der wichtigsten BC's und Konzentration darauf (Stichwort: Kern-Domäne und Generische Domänen)

6.8.3 Beispiel *Bestell-System – Order-Taking BC*

Interview mit dem Domänen-Experten:

You: "Ollie, let's talk about just one workflow, the order-placing process. What information do you need to start this process?"

Ollie: "Well, it all starts with this piece of paper: the order form that customers fill out and send us in the mail. In the computerized version, we want the customer to fill out this form online."

6.8.3 Beispiel *Bestell-System – Order-Taking BC*

Order Form		
<i>Customer Name:</i> _____		
<i>Billing Address:</i> ----- ----- -----	<i>Shipping Address:</i> ----- ----- -----	
Order: <input type="checkbox"/> Quote: <input type="checkbox"/> Express Delivery: <input type="checkbox"/>		
Product Code	Quantity	Cost
Subtotal		
Shipping		
Total		

6.8.3 Beispiel *Bestell-System – Order-Taking BC*

Sieht nach einem typischen E-Commerce-Modell aus, aber:

You: "I see. So the customers will browse the product pages on the website, then click to add items to the shopping cart, and then check out?"

Ollie: "No, of course not. Our customers already know exactly what they want to order. We just want a simple form where they can type in the product codes and quantities. They might order two or three hundred items at once, so clicking around in product pages to find each item first would be terribly slow."

Deshalb ist gutes Zuhören wichtig, eventuell auch beobachten bei der Arbeit mit dem alten System
Keine zu frühen Schlüsse ziehen!

6.8.3 Beispiel Bestell-System – Order-Taking BC

Verstehen der Nichtfunktionalen
Anforderungen!

Kontext und Scale des Workflows?

You: "Sorry, I misunderstood who the customer was. Let me get some more background information. For example, who uses this process and how often?"

Ollie: "We're a B2B company,¹ so our customers are other businesses. We have about 1000 customers, and they typically place an order every week."

You: "So about two hundred orders per business day. Does it ever get much busier than that, say in the holiday season?"

Ollie: "No. It's pretty consistent all year."

6.8.3 Beispiel Bestell-System – Order-Taking BC

Verstehen der Nichtfunktionalen Anforderungen!

D.h. keine massiver Durchsatz und auch keine Spikes.

Was erwartet der Nutzer vom System?

You: "And you say that the customers are experts?"

Ollie: "They spend all day purchasing things, so yes, they are experts in that domain. They know what they want; they just need an efficient way to get it."

6.8.3 Beispiel Bestell-System – Order-Taking BC

Verstehen der Nichtfunktionalen Anforderungen!

D.h. die Nutzer sind keine Anfänger sondern Experten – häufig unterschiedliches Design für beide Gruppen nötig.

Experten sollten so wenig wie möglich behindert werden:

You: "What about latency? How quickly do they need a response?"

Ollie: "They need an acknowledgment by the end of the business day. For our business, speed is less important than consistency. Our customers want to know that we will respond and deliver in a predictable way."

6.8.3 Beispiel Bestell-System – Order-Taking BC

You: "OK, what do you do with each form?"

Ollie: "First we check that the product codes are correct. Sometimes there are typos or the product doesn't exist."

You: "How you know if a product doesn't exist?"

Ollie: "I look it up in the product catalog. It's a leaflet listing all the products and their prices. A new one is published every month. Look, here's the latest one sitting on my desk."

Produktkatalog erscheint als ein weiterer BC

- Abfrage für diesen Workflow: Liste von Produkten und ihrer Preise

6.8.3 Beispiel Bestell-System – Order-Taking BC

You: "And then?"

Ollie: "Then we add up the cost of the items, write that into the Total field at the bottom, and then make two copies: one for the shipping department and one for the billing department. We keep the original in our files."

You: "And then?"

Ollie: "Then we scan the order and email it to the customer so that they can see the prices and the amount due. We call this an 'order acknowledgment.'"

Validierung und die Übermittlung an die anderen Abteilungen im Hinterkopf behalten bzw. notieren

6.8.3 Beispiel Bestell-System – Order-Taking BC

You: "What are those boxes marked 'Quote' and 'Order' for?"

Ollie: "If the 'Order' box is checked, then it's an order and if the 'Quote' box is checked, then it's a quote. Obviously."

You: "So what's the difference between a quote and an order?"

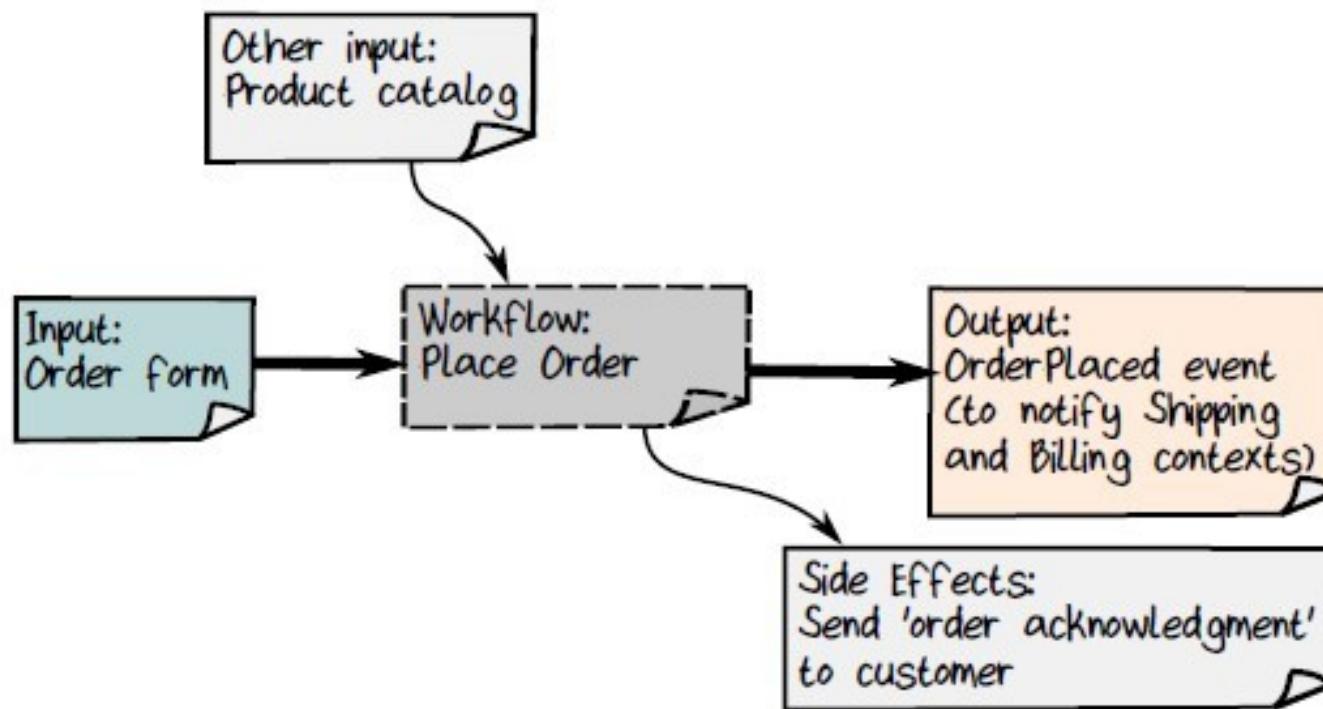
Ollie: "A quote is when the customer just wants us to calculate the prices but not actually dispatch the items. With a quote, we just add prices to the form and send it back to the customer. We don't send copies to the shipping and billing departments because there's nothing for them to do."

You: "I see. Orders and quotes are similar enough that you use the same order form for both, but they have different workflows associated with them."

Wichtige nicht offensichtliche Erkenntnis über den Bestellprozess – hätte man anders vermutet

6.8.3 Beispiel Bestell-System – Order-Taking BC

Workflow mit einen entsprechenden Ein- und Ausgaben sowie Seiten-Effekten:



6.8.3 Beispiel Bestell-System – Order-Taking BC

An diesem Punkt des Design-Prozesses starten die meisten Entwickler mit einem „technischen Design“ gegen diesen Impuls muss man entsprechend ankämpfen

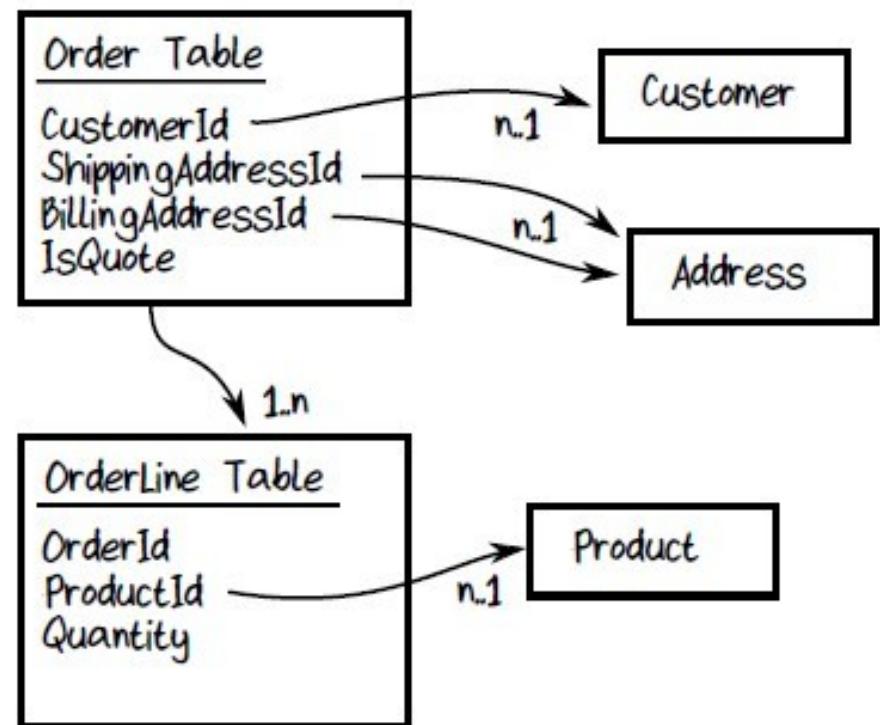
1.) Database-Driven Design:

- Umsetzung der Daten des Bestellformulars in Datenbanktabellen und Relationen
- Besonders anfällig, wenn viel Erfahrung mit RDBMS vorliegt
- Konzept der Datenbank gehört nicht zur *Ubiquitous Language*

6.8.3 Beispiel Bestell-System – Order-Taking BC

1.) Database-Driven Design:

- Die Domäne soll das Design *treiben* und nicht das Datenbankschema
- DDD soll *persistence ignorant* erfolgen, sonst wird das Design so durchgeführt, dass es zum Datenbankmodell passt



6.8.3 Beispiel *Bestell-System – Order-Taking BC*

1.) Database-Driven Design:

- z.B. passt Order und Quote nicht so zusammen, hier wird mit einem Flag unterschieden, aber die Validierung ist am Ende unterschiedlich (Order muss eine Rechnungsadresse enthalten – Quote nicht)

6.8.3 Beispiel Bestell-System – Order-Taking BC

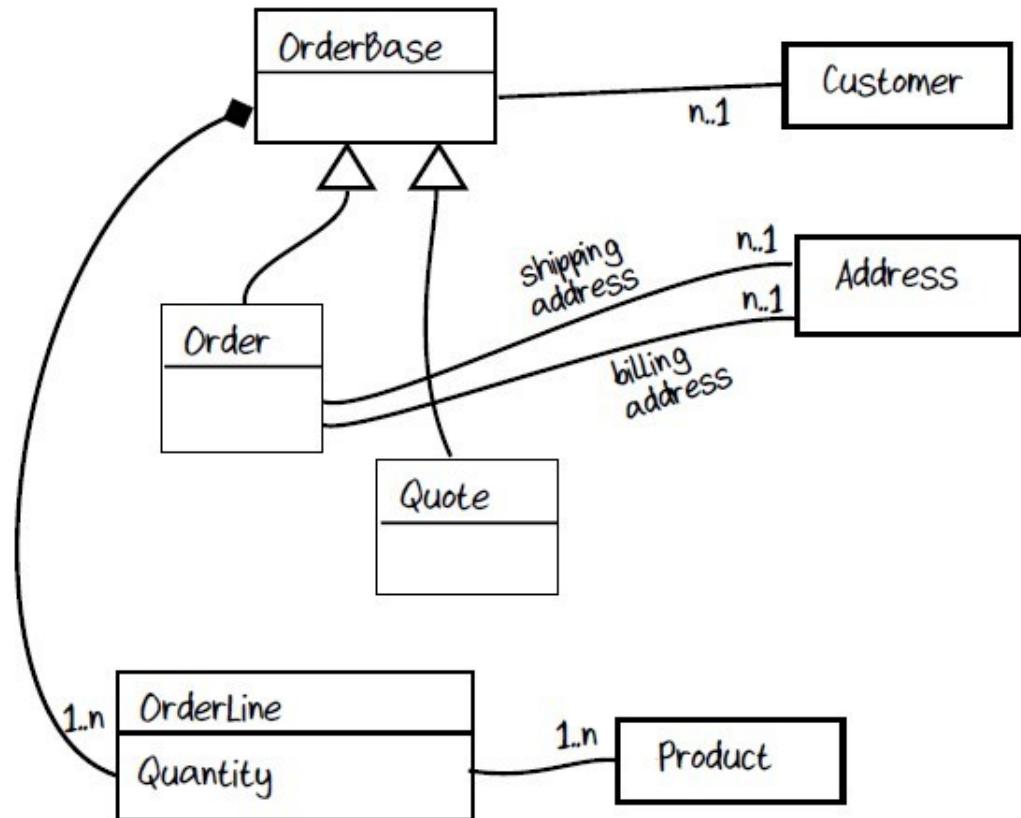
An diesem Punkt des Design-Prozesses starten die meisten Entwickler mit einem „technischen Design“ gegen diesen Impuls muss man entsprechend ankämpfen

2.) Class-Driven Design:

- Die Meisten Entwickler haben viel Erfahrung mit OOP-Sprachen – entwickeln meistens schon während des Interviews ein Klassenmodell im Kopf
- Kann genauso gefährlich sein, wie Database-Driven Design

6.8.3 Beispiel Bestell-System – Order-Taking BC

2.) Class-Driven Design:
– z.B. Konzept der Basis-Klasse gehört auch nicht zur *Ubiquitous Language*



6.8.3 Beispiel Bestell-System – Order-Taking BC

Domäne und ihre Anforderungen möglichst neutral dokumentieren

- Mit Grafiken (UML) möglich, aber nicht leicht
- Vereinfachte Sprache
 - **Workflows:**
 - Ein- und Ausgabe dokumentieren + einfachen „Pseudocode“ für die Geschäftslogik
 - **Datenstrukturen**
 - AND falls alle Teile nötig sind (Name AND Address)
 - OR falls alternativ (Email OR PhoneNumber)

6.8.3 Beispiel Bestell-System – Order-Taking BC

Order-Taking Workflow:

Bounded context: Order-Taking

Workflow: "Place order"

triggered by:

"Order form received" event (when Quote is not checked)

primary input:

An order form

other input:

Product catalog

output events:

"Order Placed" event

side-effects:

An acknowledgment is sent to the customer,
along with the placed order

6.8.3 Beispiel Bestell-System – Order-Taking BC

Order-Taking Data-Structures:

```
bounded context: Order-Taking

data Order =
    CustomerInfo
    AND ShippingAddress
    AND BillingAddress
    AND list of OrderLines
    AND AmountToBill

data OrderLine =
    Product
    AND Quantity
    AND Price

data CustomerInfo = ??? // don't know yet
data BillingAddress = ??? // don't know yet
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

Weiter im Interview, um die Details zu verstehen:

You: "Ollie, could you go into detail on how you work with an order form?"

Ollie: "When we get the mail in the morning, the first thing I do is sort it. Order forms are put on one pile, and other correspondence is put on another pile. Then, for each form, I look at whether the Quote box has been checked; if so, I put the form on the Quotes pile to be handled later."

You: "Why is that?"

Ollie: "Because orders are always more important. We make money on orders. We don't make money on quotes."

Welche Informationen sind hier relevant?

6.8.3 Beispiel Bestell-System – Order-Taking BC

Aus technischer Sicht wären Quotes und Orders gleich wichtig, aber aus geschäftlicher Sicht ist es der Punkt womit das Geld verdient wird:

- Mehr Gewinn zu produzieren oder Kosten zu senken, ist meistens die treibende Kraft hinter einer Entwicklung
- Wichtig die Prioritäten, die darauf folgen zu erkennen (*Follow the Money!*)
- D.h. Bestellungen müssen im neuen System Priorität genießen

You: "What's the first thing you do when processing an order form?"

Ollie: "The first thing I do is check that the customer's name, email, shipping address, and billing address are valid."

6.8.3 Beispiel Bestell-System – Order-Taking BC

D.h. Adressen werden mit einer weiteren Applikation auf dem Rechner auf ihre Existenz überprüft (kam im Event-Storming nicht vor)

- Kommunikation mit einem externen Dienst
- Umwandlung in ein entsprechendes Datenformat für diesen Dienst

Ist die Adresse nicht validierbar – problematischer Teil wird rot markiert und das Bestellformular auf einen dritten Stoß gelegt

3 Papier-Stöße:

- Eingang von Bestellformularen (aus der Mail)
- Eingang von Angebotsnachfragen (zur späteren Bearbeitung)
- Invalide Bestellformulare (auch später behandelt)

6.8.3 Beispiel Bestell-System – Order-Taking BC

Papierstöße mit unterschiedlicher Priorität –
Papierstöße legen schon ein wenig eine Queue nahe (aber technisches Detail, deshalb später)

Ollie: “After that, I check the product codes on the form. Sometimes they are obviously wrong.”

You: “How can you tell?”

Ollie: “Because the codes have certain formats. The codes for widgets start with a W and then four digits. The codes for gizmos start with a G and then three digits.”

You: “Are there any other types of product codes? Or likely to be soon?”

Ollie: “No. The product code formats haven’t changed in years.”

You: “What about product codes that look right? Do you check that they are real products?”

Ollie: “Yes. I look them up in my copy of the product catalog. If any codes are not there, I mark the form with the errors and put it in the pile of invalid orders.”

6.8.3 Beispiel Bestell-System – Order-Taking BC

D.h. für die Validierung der Produkt-Codes:

- Beginnt der Produkt-Code mit einem **W** oder **G**, folgen darauf 4 und 3 Zahlen ...
 - Ein rein syntaktischer Test ohne auf den Produktkatalog zurückzugreifen
- Dann manuelles Nachschlagen des Produkt-Codes im physischen Produktkatalog → in Software - Datenbank-Lookup

You: "Here's a silly question. Let's say that someone on the product team could respond instantly to all your questions. Would you still need your own copy of the product catalog?"

Ollie: "But what if they are busy? Or the phones were down? It's not really about speed, it's about control. I don't want my job to be interrupted because somebody else isn't available. If I have my own copy of the product catalog, I can process almost every order form without being dependent on the product team."

6.8.3 Beispiel Bestell-System – Order-Taking BC

Dependency-Management – nicht Performance

- Upstream-Abhängigkeit → Design für unabhängiges Arbeiten in BC's – Anforderung die im Blick zu behalten ist

You: "OK, now say that all the product codes are good. What next?"

Ollie: "I check the quantities."

You: "Are the quantities integers or floats?"

Ollie: "Float? Like in water?"

6.8.3 Beispiel Bestell-System – Order-Taking BC

Domänenexperten kennen nicht unbedingt Typen von Programmiersprachen – *Ubiquitous Language!*

You: “What do you call those numbers then?”

Ollie: “I call them ‘order quantities,’ duh!”

OrderQuantity muß ein Ausdruck der *Ubiquitous Language* werden, genauso wie *ProductCode*, *AmountToBill*,

You: “Do the order quantities have decimals, or are they just whole numbers?”

Ollie: “It depends.”

6.8.3 Beispiel Bestell-System – Order-Taking BC

„Das kommt ganz darauf an ...“ - Signalwort,
dass es komplizierter wird ...

You: “It depends on what?”

Ollie: “It depends on what the product is. Widgets are sold by the unit, but gizmos are sold by the kilogram. If someone has asked for 1.5 widgets, then of course that’s a mistake.”

6.8.3 Beispiel Bestell-System – Order-Taking BC

You: "OK, say that all the product codes and order quantities are good. What next?"

Ollie: "Next, I write in the prices for each line on the order and then sum them up to calculate the total amount to bill. Next, as I said earlier, I make two copies of the order form. I file the original and I put one copy in the shipping outbox and a second copy in the billing outbox. Finally, I scan the original, attach it to a standard acknowledgment letter, and email it back to the customer."

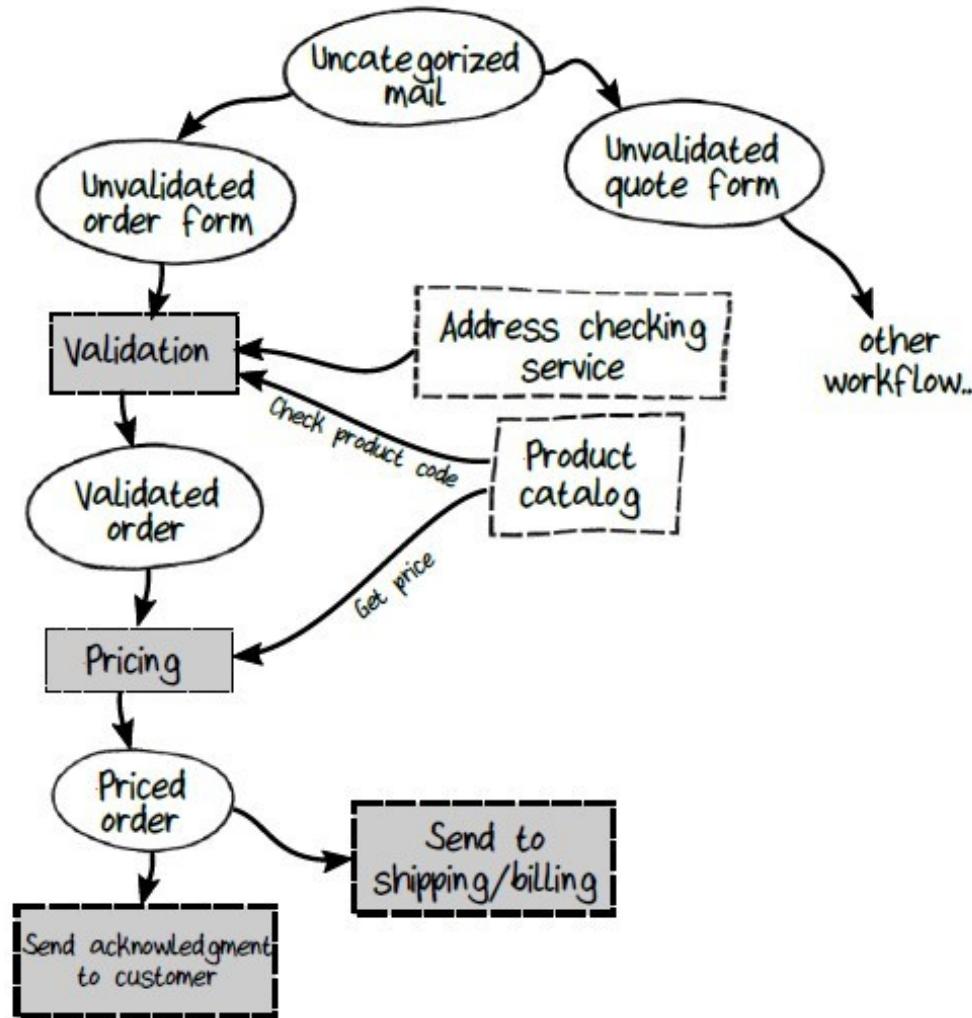
You: "One last question. You have all these order forms lying around. Do you ever accidentally mix up ones you have processed with ones that are still unvalidated?"

Ollie: "No. Every time I do something with them I mark them somehow. For example, when a form has been validated, I put a mark up here in the corner, so I know I've done that. I can tell when I've calculated the prices because the "total" box is filled out. Doing this means I can always tell order forms at different stages apart."

Fassen wir alles bisherige zusammen ...

6.8.3 Beispiel Bestell-System – Order-Taking BC

Überblick über den jetzt komplexeren Workflow:



Noch nicht alles
enthalten, was
wir bereits
wissen!

Vielleicht geht
das mit unserem
„Sprach-Modell“
besser ...

6.8.3 Beispiel Bestell-System – Order-Taking BC

Constraints darstellen:

Product-Codes und Quantities

context: Order-Taking

```
data WidgetCode = string starting with "W" then 4 digits
data GizmoCode = string starting with "G" then 3 digits
data ProductCode = WidgetCode OR GizmoCode
```

Begriffe des Interviews, wie *WidgetCode*, auch für das Design genutzt – Teil der *Ubiquitous Language*

Constraints für *WidgetCode* und *GizmoCode* beschrieben und *ProductCode* definiert als Option der beiden Typen

6.8.3 Beispiel Bestell-System – Order-Taking BC

Ist das zu strikt? Was passiert, wenn neue Produkttypen behandelt werden müssen? - Sind wir zu strikt, sind spätere Änderungen schwerer, sind wir zu offen haben wir überhaupt kein Design!

Hängt vom Kontext ab – wie immer:

- Generell sollte das Design die Sicht des Domänen-Experten abbilden
- Überprüfung der Codes ist ein wichtiger und wesentlicher Teil der Validierung und sollte sich im Design widerspiegeln – Design sollte selbsterklärend sein
- Wenn nicht hier hätte das anderswo dokumentiert werden müssen

6.8.3 Beispiel Bestell-System – Order-Taking BC

Ändern sich die Requirements, ist es leicht zu ergänzen

- Neues Produkt kann einfach in einer neuen Zeile hinzugefügt werden

Außerdem bedeutet ein striktes Design noch nicht, dass die letztendliche Implementierung ebenfalls strikt sein muss

```
data OrderQuantity = UnitQuantity OR KilogramQuantity
data UnitQuantity = integer between 1 and ?
data KilogramQuantity = decimal between ? and ?
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

Wie mit dem Produkt-Code wird *OrderQuantity* als Option zwischen *UnitQuantity* und *KilogramQuantity* definiert

Es fällt auf das im Interview die Grenzen noch nicht genannt wurden – Rücksprache mit dem Domänenexperten:

- Größte Menge an bestellbaren Einheiten 1000
- Kleinstes Gewicht 0.05 kg und höchstes 100 kg

Diese Constraints sind wichtig um valide Daten im Produktionscode zu haben

```
data UnitQuantity = integer between 1 and 1000  
data KilogramQuantity = decimal between 0.05 and 100.00
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

Life-Cycle einer Order:

Im ersten Ansatz des Designs

```
data Order =  
    CustomerInfo  
    AND ShippingAddress  
    AND BillingAddress  
    AND list of OrderLines  
    AND AmountToBill
```

Mit unserem aktuellen Wissensstand ist klar, dass diese Definition nicht ausreicht - Bestellungen haben einen Lebenszyklus:

- „Unvalidiert“
- „Validiert“
- „Bepreist“

6.8.3 Beispiel Bestell-System – Order-Taking BC

Im Interview wurde klar, dass bisher die einzelnen Phasen durch Markierungen auf dem Bestellformular gekennzeichnet wurden

Die gleichen Phasen müssen im Domänen-Modell abgebildet werden – nicht nur zur Dokumentation, sondern auch für den Workflow → eine unbepreiste Bestellung darf nicht zum Versand kommen

Am einfachsten entsprechende Benennung einführen – *UnvalidatedOrder*, *ValidatedOrder*, ... etc.

6.8.3 Beispiel Bestell-System – Order-Taking BC

Design wird dabei zwar länger und unschöner (wir erinnern uns – Domäne ist wichtiger als „pures Design“)

„Unvalidated“:

```
data UnvalidatedOrder =  
    UnvalidatedCustomerInfo  
    AND UnvalidatedShippingAddress  
    AND UnvalidatedBillingAddress  
    AND list of UnvalidatedOrderLine  
  
data UnvalidatedOrderLine =  
    UnvalidatedProductCode  
    AND UnvalidatedOrderQuantity
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

„Validated“:

```
data ValidatedOrder =
    ValidatedCustomerInfo
    AND ValidatedShippingAddress
    AND ValidatedBillingAddress
    AND list of ValidatedOrderLine

data ValidatedOrderLine =
    ValidatedProductCode
    AND ValidatedOrderQuantity
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

„Priced“:

Entspricht einer „Validated“ Order bis auf:

- Jede Position hat einen assoziierten Preis – *PricedOrderLine* ist eine *ValidatedOrderLine* mit einem *LinePrice*
- Die Bestellung als ganzes hat einen assoziierten *AmountToBill*, der die Summe der einzelnen Positionspreise darstellt

```
data PricedOrder =
    ValidatedCustomerInfo
    AND ValidatedShippingAddress
    AND ValidatedBillingAddress
    AND list of PricedOrderLine // different from ValidatedOrderLine
    AND AmountToBill           // new

data PricedOrderLine =
    ValidatedOrderLine
    AND LinePrice             // new
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

Schließlich noch das Acknowledgment:

```
data PlacedOrderAcknowledgment =  
    PricedOrder  
    AND AcknowledgmentLetter
```

Entwickeltes Modell enthält schon einige Domänen-Regeln wie, z.B.

- Eine unvalidierte Bestellung hat keinen Preis
- Alle Positionen auf einer validierten Bestellungen müssen validiert sein, nicht nur ein paar

Modell ist komplexer als gedacht – aber es ist nötig, um die Anforderungen zu erfassen

6.8.3 Beispiel Bestell-System – Order-Taking BC

Ausarbeiten der Schritte im Workflow:

- Aufteilen in die Schritte *Validierung*, *Bepreisung* ... etc.
- Anwendung des Ein-/Ausgabe-Ansatzes auf jeden dieser Schritte
- Ausgabe des gesamten Workflows komplizierter als im ursprünglichen, wo er nur zum „Order placed“-Event führte – jetzt:
 - „Order placed“-Event zu Shipping/Billing senden oder
 - Bestellung auf den „Invalid Order“-Stoß legen und den Rest der Schritte überspringen

6.8.3 Beispiel Bestell-System – Order-Taking BC

```
workflow "Place Order" =
    input: OrderForm
    output:
        OrderPlaced event (put on a pile to send to other teams)
        OR InvalidOrder (put on appropriate pile)

    // step 1
    do ValidateOrder
    If order is invalid then:
        add InvalidOrder to pile
        stop

    // step 2
    do PriceOrder

    // step 3
    do SendAcknowledgmentToCustomer

    // step 4
    return OrderPlaced event (if no errors)
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

Validieren der Bestellung:

- Nimmt eine *UnvalidatedOrder* als Eingabe und liefert entweder eine *ValidatedOrder* oder einen *ValidationException*
- Abhängigkeiten wie der Input vom Produktkatalog (*CheckProductCodeExists*) und den externen Service zur Überprüfung der Adressen (*CheckAddressExists*)

6.8.3 Beispiel Bestell-System – Order-Taking BC

```
substep "ValidateOrder" =
    input: UnvalidatedOrder
    output: ValidatedOrder OR ValidationError
    dependencies: CheckProductCodeExists, CheckAddressExists

    validate the customer name
    check that the shipping and billing address exist
    for each line:
        check product code syntax
        check that product code exists in ProductCatalog

    if everything is OK, then:
        return ValidatedOrder
    else:
        return ValidationError
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

Bepreisen der Bestellung:

- Nimmt eine *ValidatedOrder* als Eingabe und liefert eine *PricedOrder*
- Abhängigkeiten auf den Produktkatalog (*GetProductPrice*)

```
substep "PriceOrder" =  
    input: ValidatedOrder  
    output: PricedOrder  
    dependencies: GetProductPrice  
  
    for each line:  
        get the price for the product  
        set the price for the line  
    set the amount to bill ( = sum of the line prices)
```

6.8.3 Beispiel Bestell-System – Order-Taking BC

Acknowledgment senden:

- Nimmt eine *PricedOrder* und erzeugt und sendet ein Acknowledgement

```
substep "SendAcknowledgmentToCustomer" =  
    input: PricedOrder  
    output: None  
  
    create acknowledgment letter and send it  
    and the priced order to the customer
```

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Übersetzen der Domäne in eine Architektur:

- Sicherlich ist die Domäne noch nicht wirklich komplett bekannt und modelliert – aber hilfreich ist die Implementierung eines kruden Prototypen als „Walking Skeleton“, um die Lücken zu finden
- Frühes Feedback
- Übersetzung von *BC's* und *Domain Events* in Software
- Terminologie von **C4** (Simon Brown) für die Software-Architektur

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

C4 – Kategorisierung der Software-Architektur:

- „**System Context**“ - Top-Level, der das komplette System repräsentiert
- „**Containers**“ - deploybare Einheiten, wie Web-Seiten, Web-Services, Datenbanken, ... etc.
- „**Components**“ - jeder Container enthält eine Reihe von Komponenten als Building-Blocks des Codes
- „**Classes**“ - schließlich besteht jede Komponente aus einer Reihe von Klassen oder Modulen (funktionale Architektur), die low-level Methoden oder Funktionen enthalten

Gute Architektur besteht darin diese Einteilungen richtig zu gestalten, so dass auch bei neuen Anforderungen und Änderungen die Kosten minimal bleiben

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Bounded Context in Software Architecture:

BC entworfen als ein „autonomes Subsystem“ mit gut definierten Grenzen – Reihe an Optionen:

- System entworfen als einzelner, deploy-barer Monolith (einzelner Container in C4) – *BC separates Modul mit gut-definiertem Interface oder besser mit einer noch stärker entkoppelten Assembly/DLL*
- Jeder *BC* separat im eigenen Container deployed – *Service-Oriented Architecture (SOA)*
- Noch fein-granulärer – jeder einzelne Workflow wird in einem eigenen stand-alone Container deployed – *Microservice Architecture*

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Bounded Context in Software Architecture:

- In dieser Phase muss man sich noch nicht auf einen Ansatz festlegen – so lange wir garantieren, dass *BC's* entkoppelt und autonom bleiben ist die Umsetzung von logischem Design in deploy-bare Einheiten nicht kritisch
- Schwer von Anfang an die Abgrenzung richtig zu machen – Änderungen zu erwarten, je besser wir die Domäne kennenlernen
- Refactoring ist für einen Monolith wesentlich einfacher

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Bounded Context in Software Architecture:

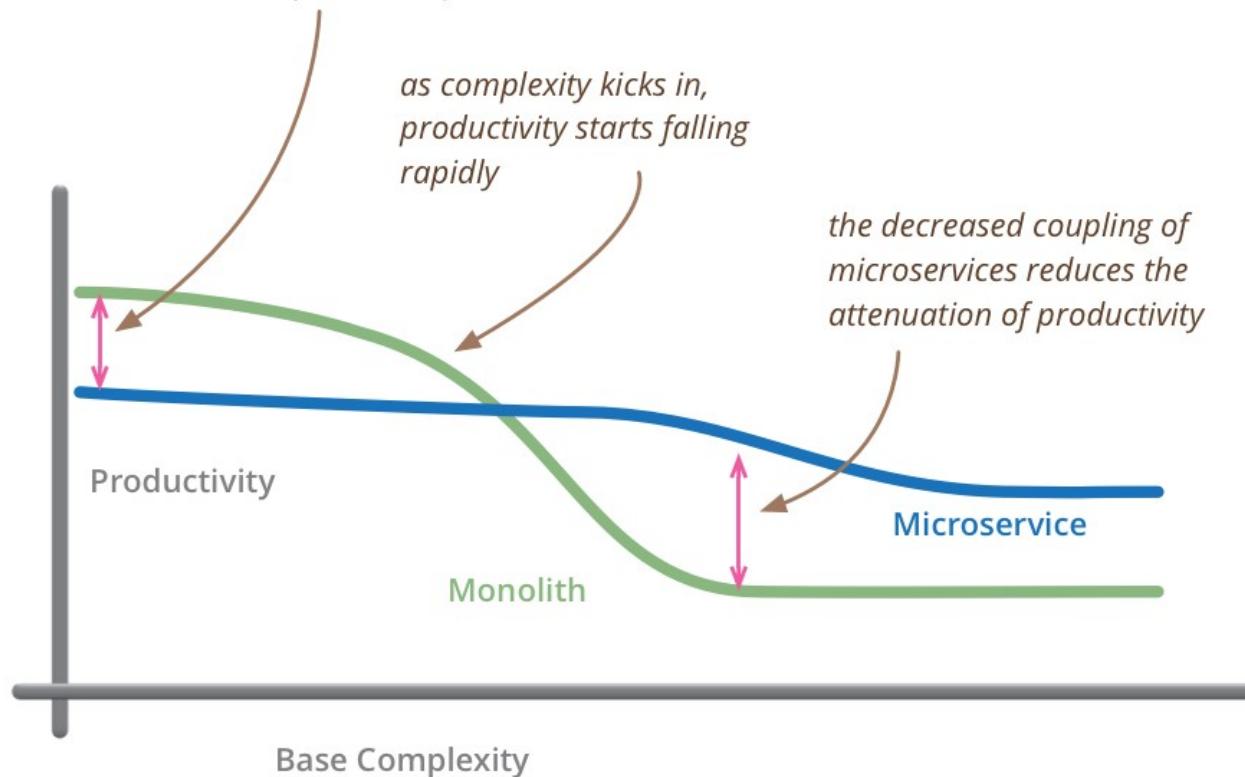
- „Best Practice“ - System anfangs als Monolith entwickeln und Refactoring zu entkoppelten Containern nur wenn nötig (nicht gleich den Overhead für eine Microservice-Architektur bezahlen, wenn der Vorteil den Nachteil nicht aufwiegt)
- Nicht einfach eine wirklich entkoppelte Microservice-Architektur zu entwickeln – einfacher Test:

Wird ein Microservice abgestellt und bricht alles zusammen, so ist es nicht wirklich eine Microservice-Architektur, sondern ein verteilter Monolith!

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Microservice Architecture – **Microservice Premium:**

for less-complex systems, the extra baggage required to manage microservices reduces productivity



but remember the skill of the team will outweigh any monolith/microservice choice

<https://www.martinfowler.com/bliki/MicroservicePremium.html>

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Microservice Architecture – Microservice Premium:

Begriff geprägt von Martin Fowler für den Overhead einer Microservice-Architektur

M. Fowler:

„So my primary guideline would be ***don't even consider microservices unless you have a system that's too complex to manage as a monolith.*** The majority of software systems should be built as a single monolithic application. Do pay attention to good modularity within that monolith, but don't try to separate it into separate services.“

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Bounded Context in Software Architecture:

Wie kommunizieren *BC's* miteinander?

Z.B. wenn der *Order-Taking BC* eine Bestellung abgearbeitet hat mit dem *Shipping BC*, dass dieser die Sendung verschickt

Events!

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Kommunikation zwischen *Bounded Context`*s:
Events!

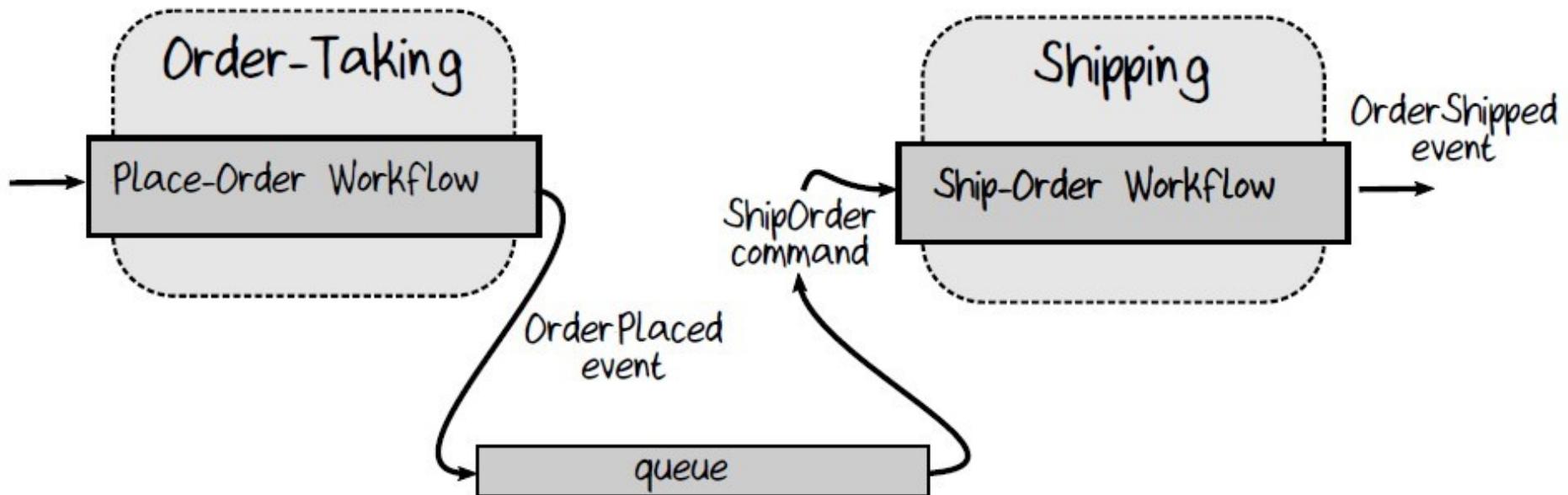
- *Place-Order Workflow* des *Order-Taking BC* schickt ein *OrderPlaced Event*
- *OrderPlaced Event* wird in eine Queue geschrieben oder anderweitig veröffentlicht
- Shipping BC „lauscht“ auf *OrderPlaced Events*
- Sobald ein Event empfangen wurde wird ein *ShipOrder Command* erzeugt
- *ShipOrder Command* initiiert den *Ship-Order Workflow*
- Schließt der *ShipOrder Workflow* erfolgreich ab, sendet er ein *OrderShipped Event*

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Kommunikation zwischen *Bounded Context*'s:

Komplett entkoppeltes Design:

- Upstream Komponente (*Order-Taking Sub-System*) und downstream Komponente (*Shipping Sub-System*) nehmen einander gar nicht wahr und kommunizieren nur über Events
- Diese Art der Entkopplung ist kritisch, um wirklich autonome Komponenten zu haben



6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Kommunikation zwischen *Bounded Context`*s:

- Event-Übertragung architekturabhängig
- Queues sind gut geeignet für eine asynchrone gepufferte Kommunikation – optimal für Microservices oder Agenten
- Im Monolith alternativ auch ein direkter Funktionsaufruf möglich – muss jetzt nicht endgültig entschieden werden bzw. sein
- Wie der Handler, der Events (*OrderPlaced*) in Commands (*ShipOrder*) übersetzt, kann es Teil des Downstream-Context sein (an der Grenze zwischen den BC`s) oder per separatem Router oder Prozeß-Manager als Teil der Infrastruktur

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

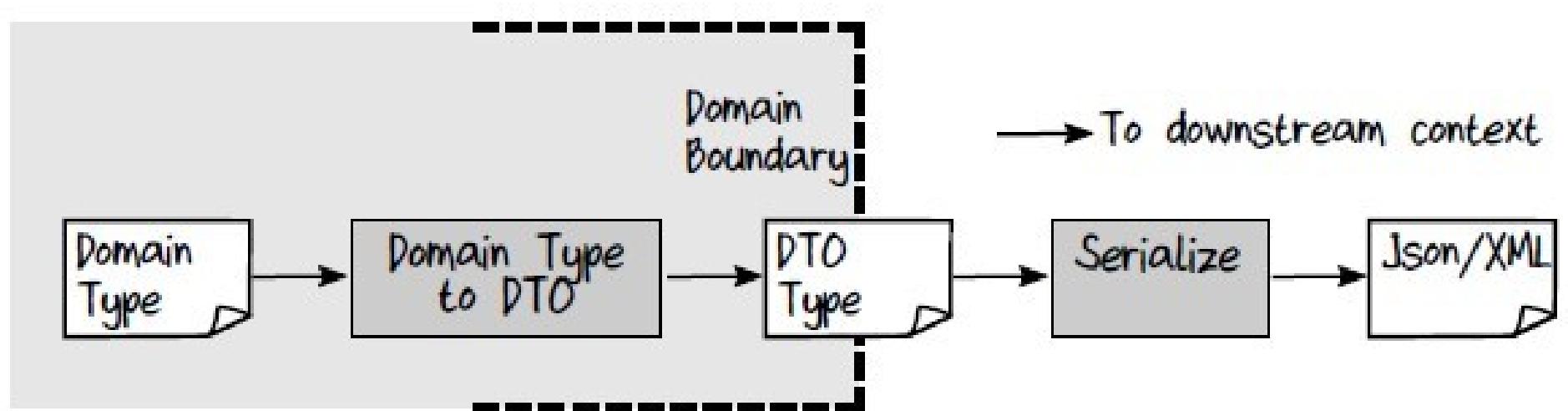
Übertragung von Daten zwischen *Bounded Context`~~’~~*s:

- Event ist mit den downstream notwendigen Daten verbunden, z.B. *OrderPlaced* kann die komplette Bestellung enthalten → *Shipping-BC* kann damit das zugehörige *ShipOrder-Command* erzeugen
- Objekte im *BC – Domain Objects*
- *Data Transfer Objects (DTOs)* für den Datenaustausch

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Übertragung von Daten zwischen *Bounded Context`*s:

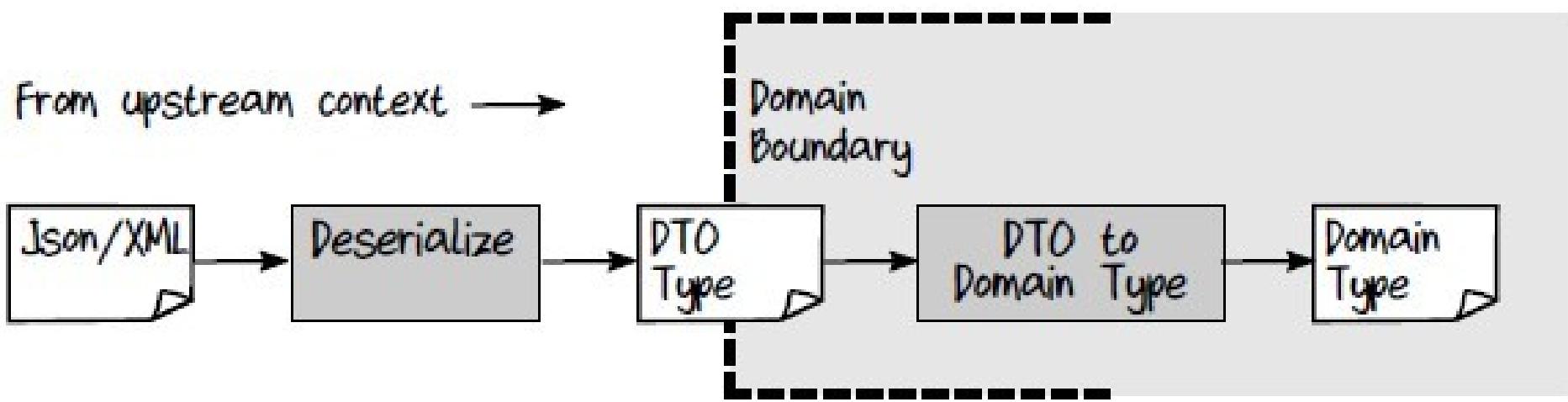
Upstream BC → Domain Object → DTO → Serialisiert → ...



6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Übertragung von Daten zwischen *Bounded Context`~~s~~*:

... → Serialisiert → *DTO* → *Domain Object* → Downstream BC



6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

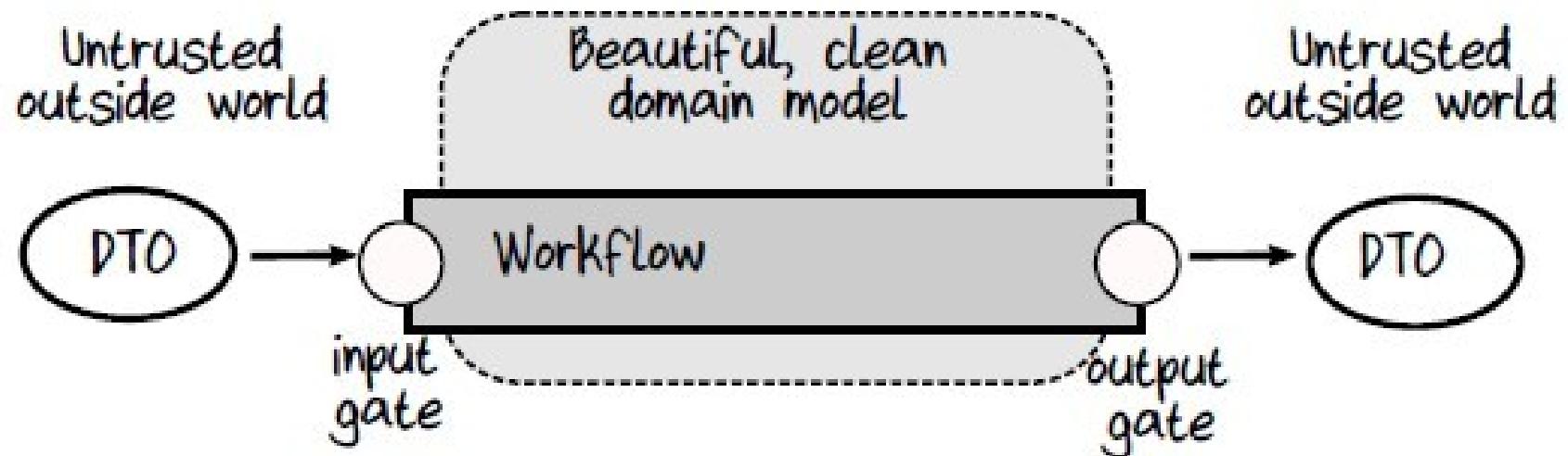
Übertragung von Daten zwischen *Bounded Context`~~s~~*:

- In der Praxis *DTO`~~s~~* meistens *Event-DTO`~~s~~*, die wiederum *DTO`~~s~~* enthalten - z.B. die *Order-DTO*, die wiederum *DTO`~~s~~* enthält, wie z.B. eine Liste an *DTO`~~s~~*, die die *OrderLines* repräsentieren

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Trust Boundaries und Validierung:

- Begrenzung des *BC* stellt eine Vertrauensgrenze dar innerhalb derer allem vertraut wird → Gates als Vermittler zwischen *Innen* und *Außen*



6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Trust Boundaries und Validierung:

- *Input-Gate:*
 - Validierung des Inputs (erfüllt Constraints?) → schlägt sie fehl, wird ein Fehler generiert
 - Aus dem *DTO* generiertes *Domain Object*, z.B. *Order*, ist valide und „*trusted*“
- *Output-Gate:*
 - Stellt sicher, dass keine „privaten Informationen“ nach außen dringen
 - Verhindert enge Kopplung nachfolgender *BC's*
 - Sicherheitsaspekt
 - „*verliert*“ Informationen

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Contracts zwischen *BC's*:

- Kopplung zwischen *BC's* soll so gering wie möglich sein
- Datenformat der *Events* und *DTO's* stellen einen *Contract* zwischen den *BC's* dar

Welcher *BC* entscheidet darüber, wie der *Contract* auszusehen hat?

- Verschiedene Ansätze in DDD, die wir schon kennengelernt haben

6.8.4 Beispiel *Bestell-System* – *Order-Taking BC* – Functional Architecture

Contracts zwischen BC's:

- *Shared Kernel*:
 - Geteiltes/Überlappendes Domänen-Design, z.B.: *Order-Taking* und *Shipping BC* nutzen das gleiche Design für die Versandadresse – *Order-Taking BC* nimmt eine Adresse und validiert diese, während der *Shipping BC* an diese Adresse verschickt
 - Änderungen der Definition eines *Events* oder *DTO's* müssen mit allen davon betroffenen BC-Teams abgesprochen werden

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Contracts zwischen *BC's*:

- *Customer/Supplier* oder *Consumer Driven Contract*:
 - Downstream-*BC* gibt vor, was der Upstream-*BC* liefern soll, z.B.:

Billing BC definiert den *Contract* nach dem Motto „das ist, was ich benötige, um dem Kunden die Rechnung zu stellen“ und das und nur das liefert dann der *Order-Taking BC*
 - Beide *BC's* können sich unabhängig voneinander entwickeln, so lange der Upstream-*BC* den *Contract* erfüllt

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Contracts zwischen *BC's*:

- *Conformist*:
 - Gegenteil von *Customer/Supplier* – Downstream *BC* akzeptiert den vom Upstream *BC* diktierten *Contract*, z.B.:
Order-Taking BC übernimmt den *Contract* des *Product Catalog* und adaptiert seinen Code entsprechend
 - Downstream-*BC* paßt sein Domänen-Modell den Vorgaben des Upstream-*BC* an

6.8.4 Beispiel *Bestell-System* – *Order-Taking BC* – Functional Architecture

Contracts zwischen BC's:

- *Anti-Corruption Layer (ACL)*:
 - Extra Schicht, die eine nicht zum Domänen-Modell passendes Interface in ein passendes umsetzt – „Korrumpieren“ des Domänen-Modells verhindern, z.B.: *ACL* zwischen *Order-Taking* und *Shipping BC*
 - In kleiner Form stellt schon das *Input-Gate* eine Art *ACL* dar
 - Primär nicht Validierung und Datenkonsistenz, sondern wirklich Übersetzung in Sprache des Domänenmodells
 - Häufig in der Kommunikation mit externen Systemen eingesetzt – externe System dahinter ist austauschbar (kein *Vendor-Lock-In*)

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

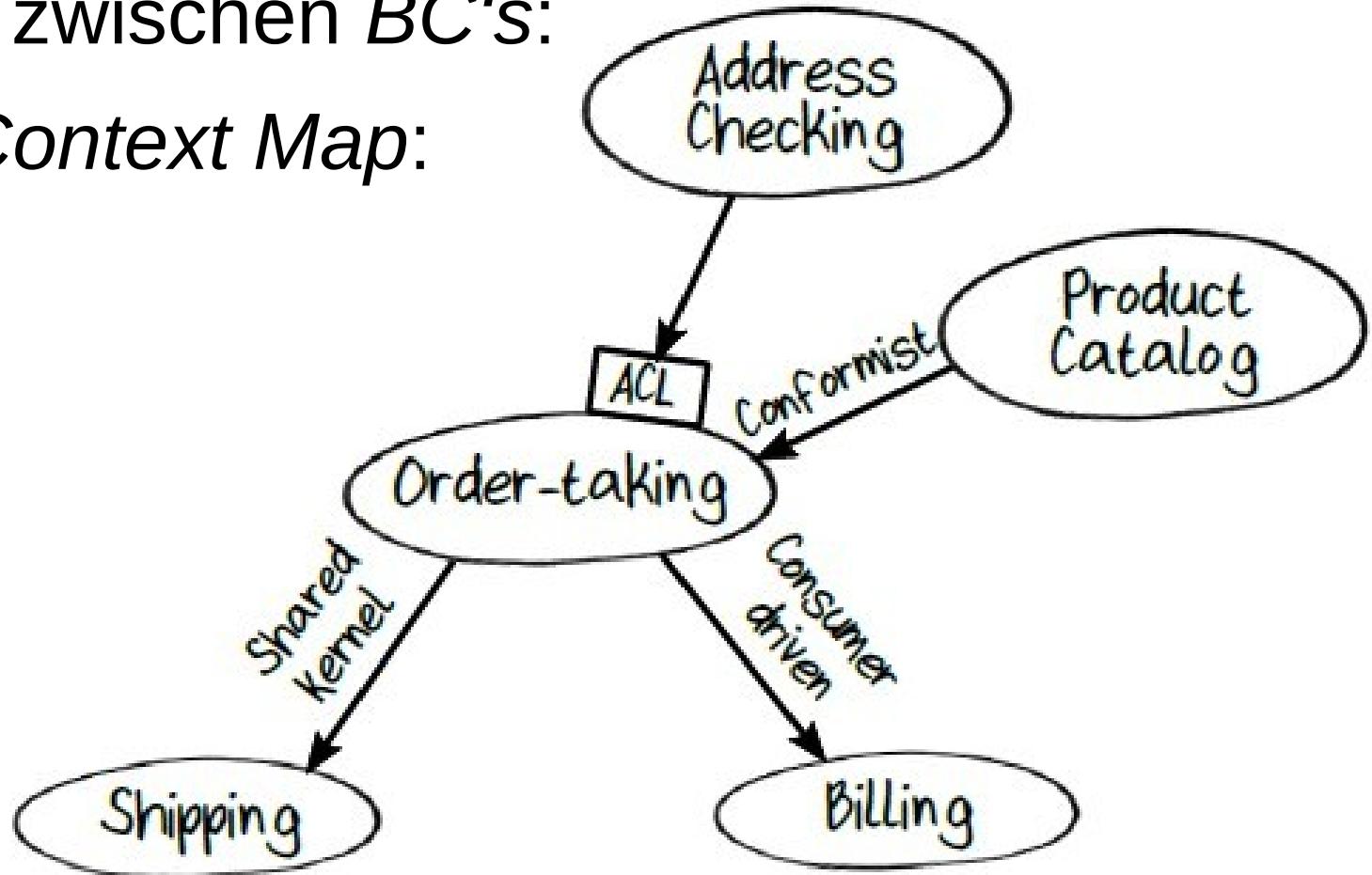
Contracts zwischen BC's:

- Aktuelle Context Map:
 - Shared Kernel zwischen Order-Taking und Shipping BC
 - Consumer-Driven Contract zwischen Order-Taking und Billing BC
 - Conformist zwischen Order-Taking und Product Catalog BC
 - ACL zwischen externen Adressen-Prüfdienst und unserem Order-Taking BC

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Contracts zwischen BC's:

- Aktuelle Context Map:



6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Contracts zwischen BC's:

- Aktuelle *Context Map*:
 - Stellt nicht nur die technischen Verhältnisse dar, sondern auch die Verhältnisse der Teams, die zu den jeweiligen BC's gehören
 - Interaktionsarten der Teams ablesbar – Interaktionen der Domänen nicht nur technische, sondern auch organisatorische Herausforderung
 - „Inverse Conway maneuver“ um sicherzustellen, das organisatorische Struktur konsistent mit der Architektur ist

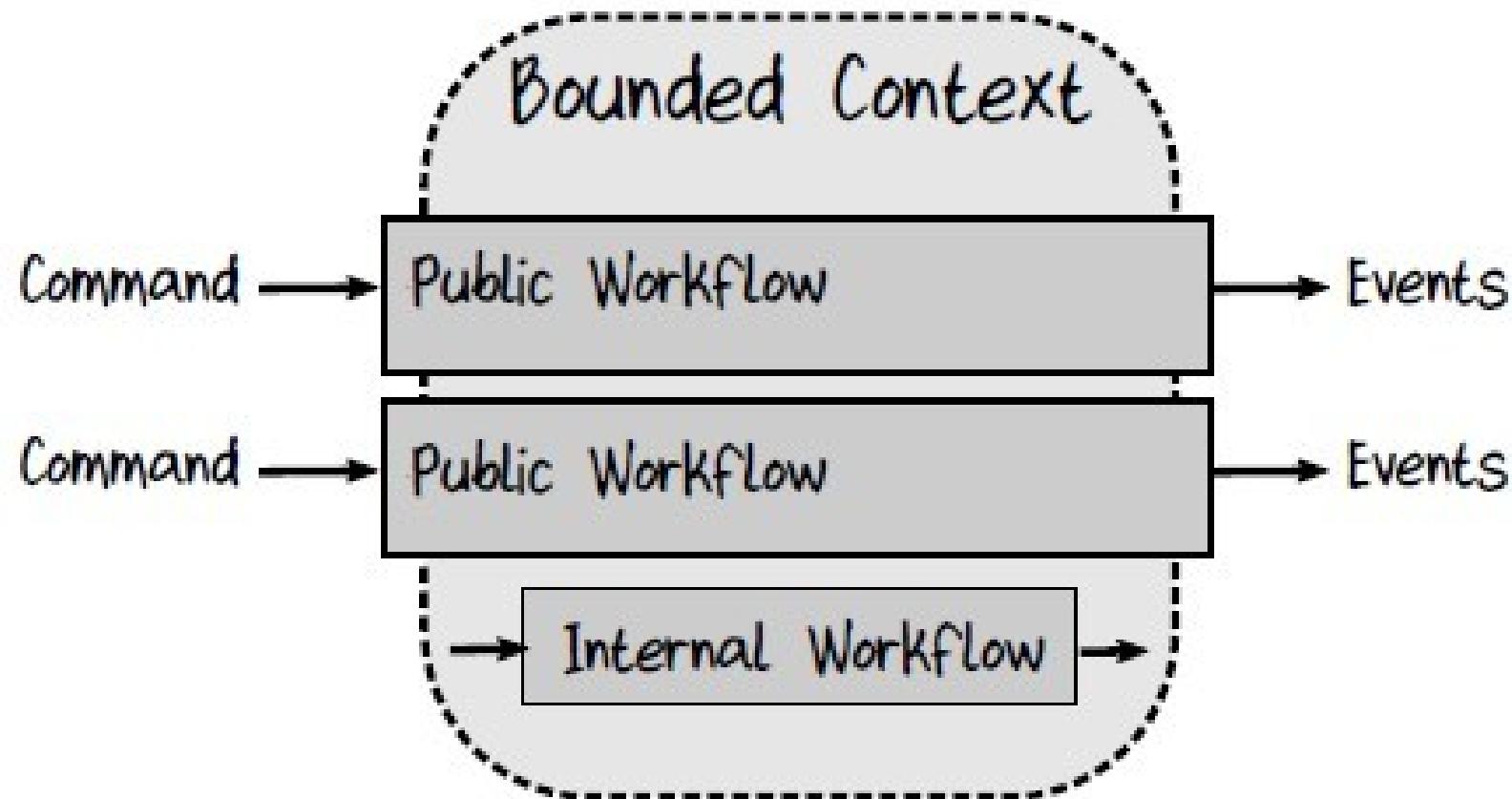
6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Workflows innerhalb eines *BC's*:

- *Business-Workflows* als Mini-Prozesse, die von Kommandos initiiert werden und ein oder mehr Domänen-Events generieren
- Im hier benutzen funktionalen Umfeld auf eine Funktion abgebildet:
 - Input: *Command Object*
 - Output: *Event*
- Ein *Workflow* ist immer in einem *BC* enthalten und implementiert nie ein Szenario „End-To-End“ über mehrere *BC's* hinweg

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Workflows innerhalb eines BC's:



6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Placing-Order-Workflow:

- Input:
 - *PlacingOrder Command* mit den assoziierten Daten
- Output:
 - Reihe an Events, wie z.B. das *OrderPlace Event* aber *Customer/Supplier-Verbindung* zum *Billing BC*
 - Nur senden der „Rechnungsdaten“ an den Billing BC, keine Versandadresse ... etc. → neues Event *BilliableOrderPlaced* definieren

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Placing-Order-Workflow:

Event BillableOrderPlaced

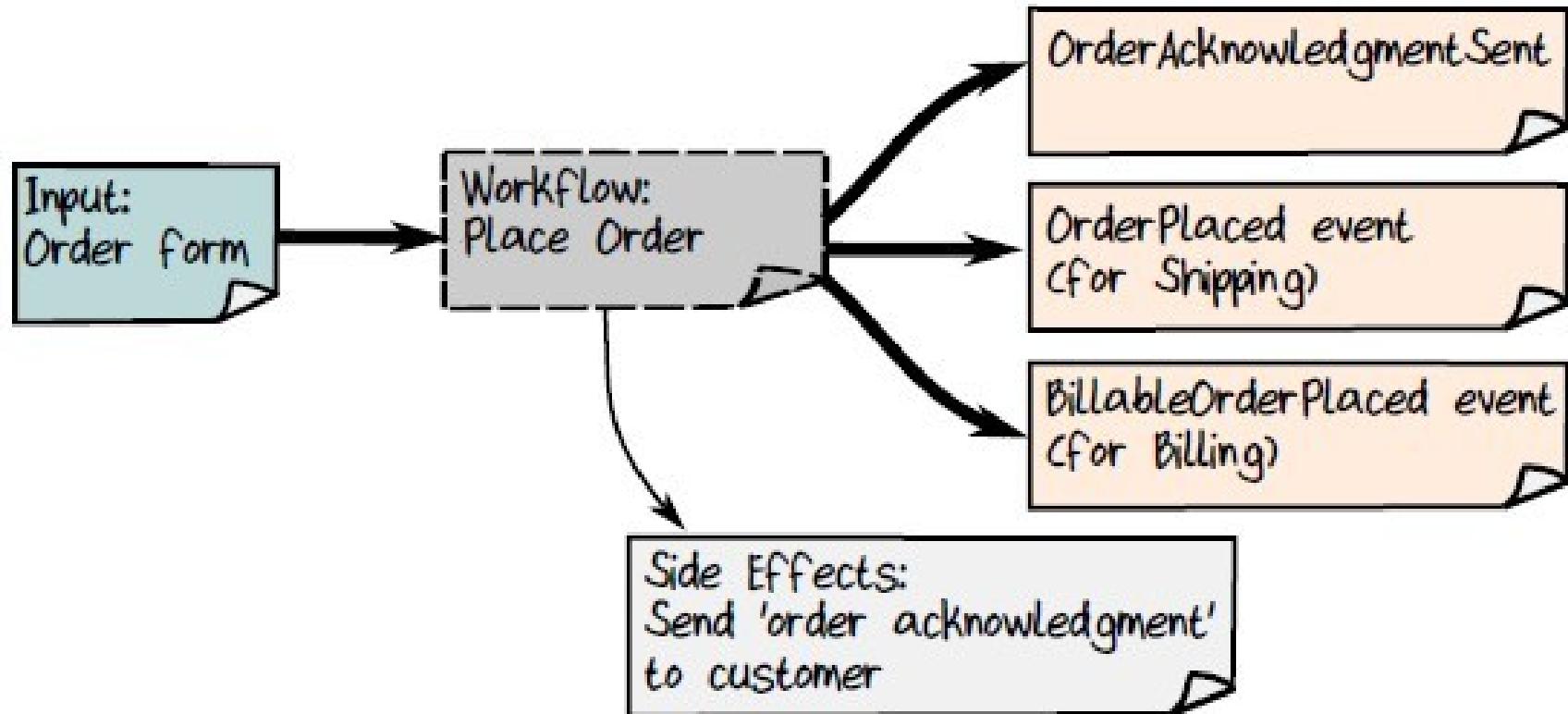
```
data BillableOrderPlaced =  
    OrderId  
    AND BillingAddress  
    AND AmountToBill
```

Damit ergibt sich ein aktualisiertes Diagramm für den *Placing-Order-Workflow*

6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

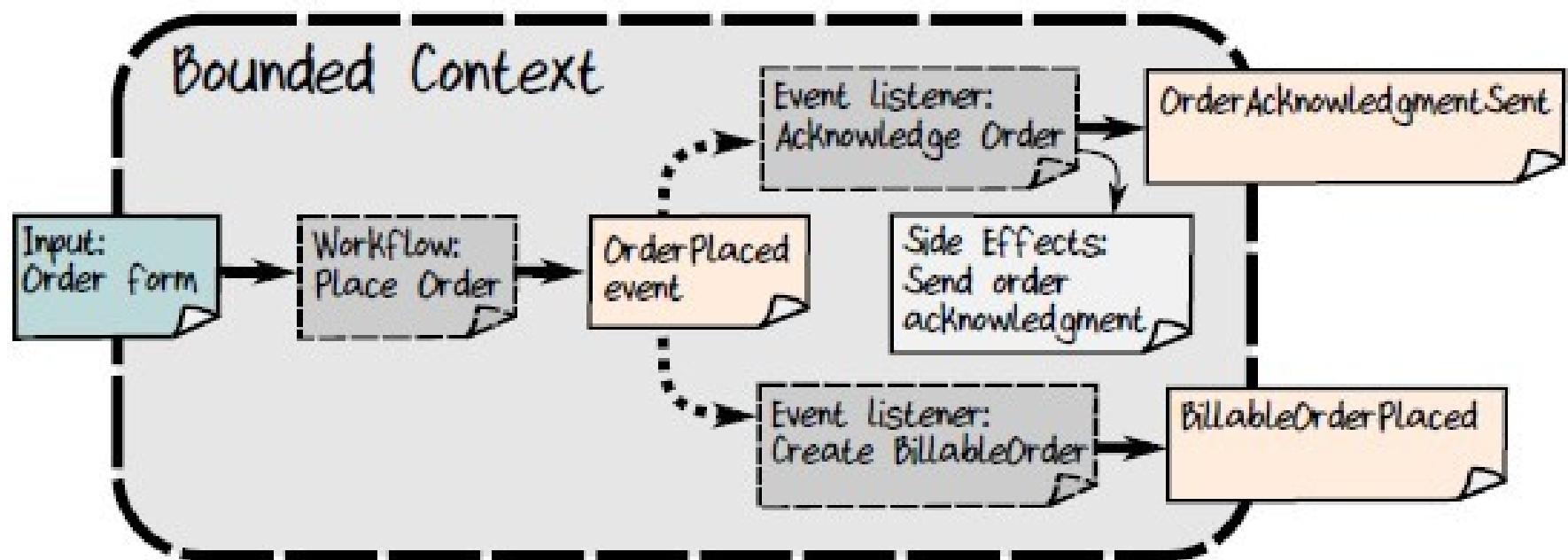
Placing-Order-Workflow:

Workflow-Funktion „veröffentlicht“ Events nicht, es liefert sie als Ausgabe → genau Art der „Veröffentlichung“ ist ein separater Aspekt



6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

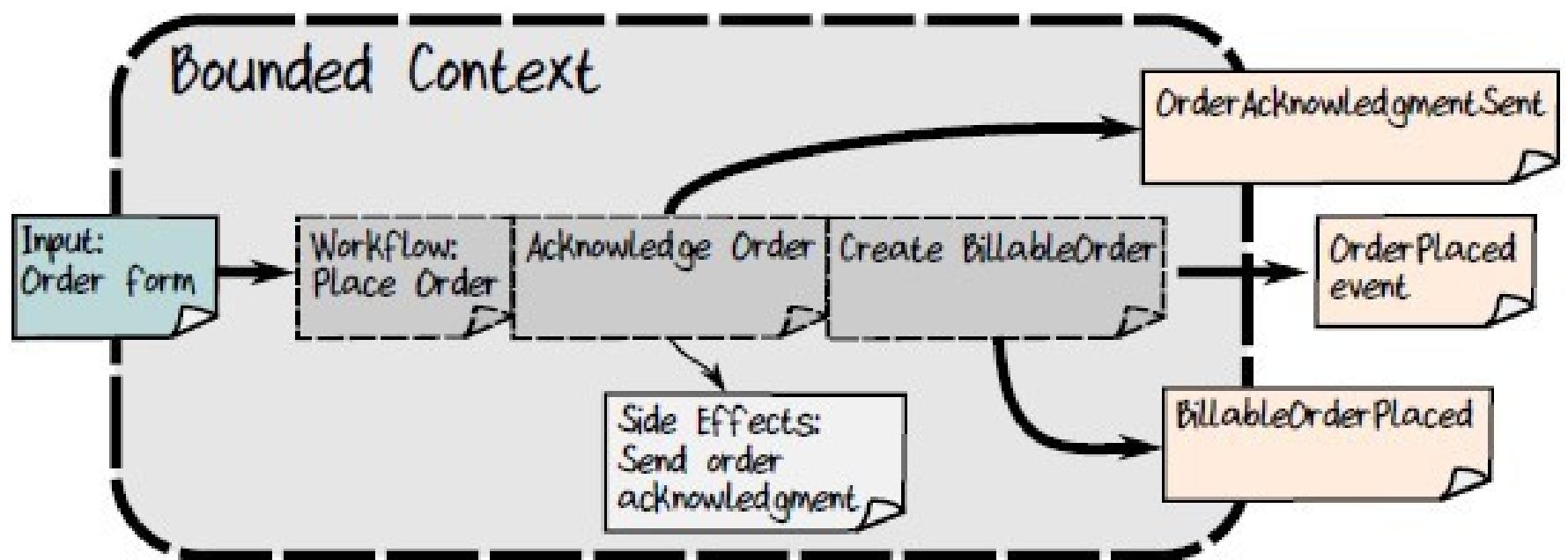
Keine *Domänen-Events* innerhalb eines BC's:
Objektorientiertes Design erzeugt innerhalb des BC's Domänen-Events auf die Listener/Handler-Objekte hören und dann reagieren ...



6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Keine *Domänen-Events* innerhalb eines BC's:

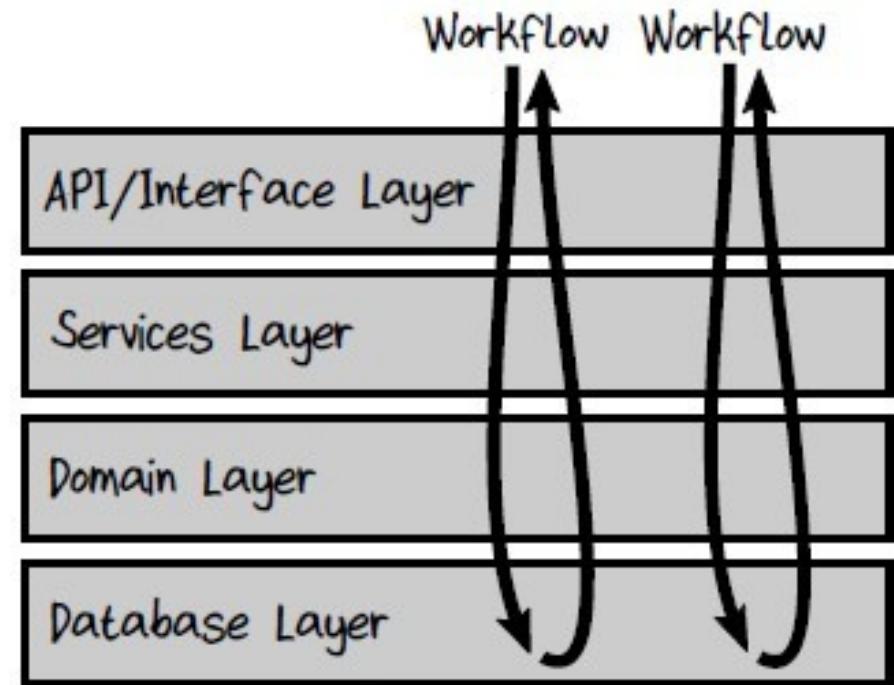
... im funktionalen Design wird das nicht verwendet, um keine versteckten Abhängigkeiten zu haben, explizites „hintereinander hängen“, keine globalen Event-Handler mit veränderbarem Zustand, ... etc.



6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Code-Struktur innerhalb eines BC's:

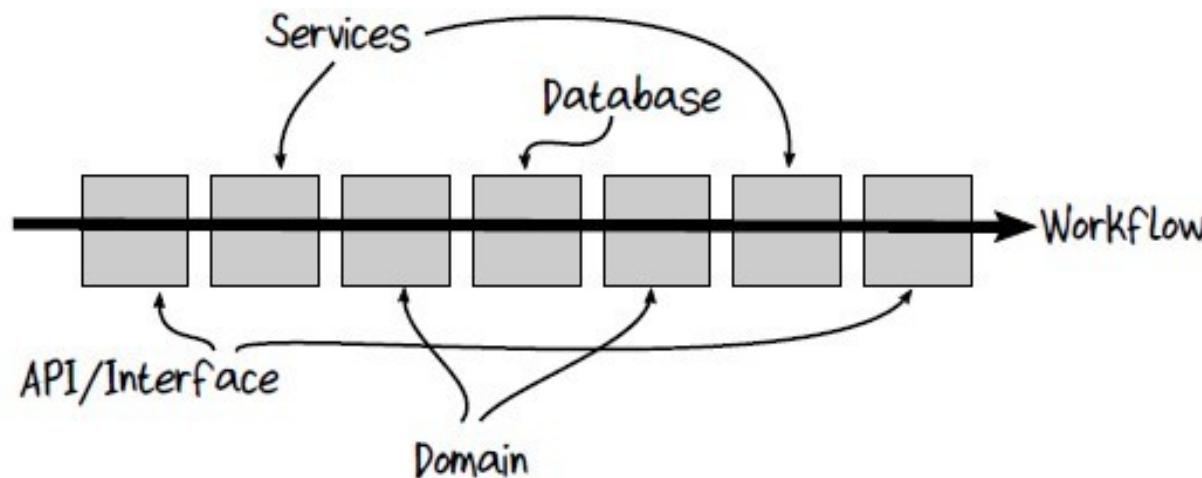
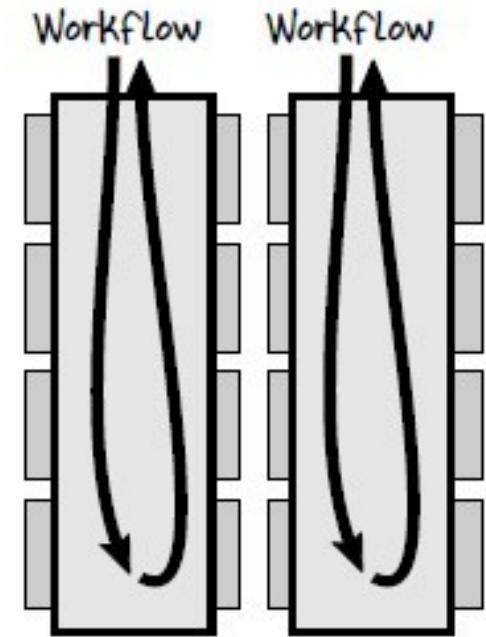
- Traditionell **horizontal** in Schichten – Workflow startet ganz oben, läuft nach unten durch und dann wieder zurück
- Bricht die Regel, dass Code, der sich zusammen ändert auch zusammen gehört – jede Anpassung des Workflows benötigt auch eine Anpassung jeder Schicht



6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Code-Struktur innerhalb eines BC's:

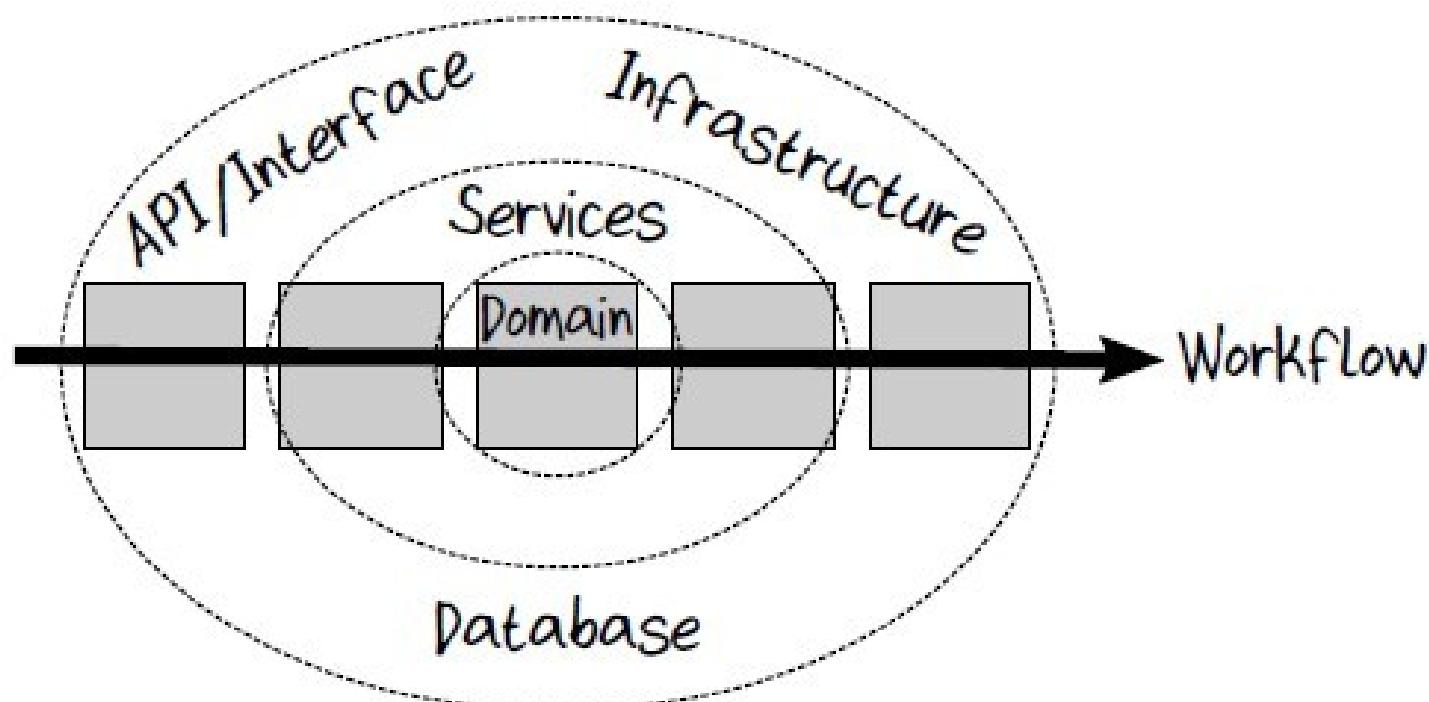
- Wechsel zu **vertikalen** Schichten – jeder Workflow enthält den Code, den er für seine Arbeit braucht
- Noch nicht ideal – horizontale Pipeline für den Workflow:



6.8.4 Beispiel Bestell-System – Order-Taking BC – Functional Architecture

Onion-Architecture:

- Domänen-Code ins Zentrum setzen und alle anderen Aspekte drumherum



6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

Onion-Architecture:

- Jede Schicht hängt nur von inneren Schichten ab und nie von weiter außen liegenden
D.h.: „***all dependencies must point inward!***“
- Enger Bezug zu *Hexagonal Architecture* und *Clean Architecture*
- Sicherstellen der obigen Bedingung durch *Dependency Injection*

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

„Keep I/O at the Edges!“:

- Kern funktionaler Programmierung ist die Arbeit mit „reinen Funktionen“ („pure Functions“), d.h. Funktionen im mathematischen Sinne
 - Reproduzierbar, vorhersagbar, leichter zu verstehen, besser zu testen, seiteneffektfrei ... etc.
 - Implementierung der Domäne so weit wie möglich mit „pure Functions“

6.8.4 Beispiel *Bestell-System – Order-Taking BC – Functional Architecture*

,,Keep I/O at the Edges!“:

- **Aber** Seiteneffekte nötig für jegliche I/O, wie Datenein- und Ausgabe, Lesen oder Schreiben in Files oder Datenbanken, Netzwerkkommunikation, ... etc.
- Innerer „pure“er, seiteneffektfreier Kern und seiteneffekt behaftete Funktionen außen am Rand zur Kommunikation mit der Außenwelt

7.1 Software Architektur Stil

Was ist ein Architektur Stil:

- D. Garlan & M. Shaw, Introduction to Software Architecture, 1994:

„An architectural style, then, defines a family of such systems in terms of a pattern of structural organization. More specifically, an architectural style determines the vocabulary of components and connectors that can be used in instances of that style, together with a set of constraints on how they can be combined.“

7.1 Software Architektur Stil

Was ist ein Architektur Stil:

- Gibt eine bestimmte Palette an Elementen (Komponenten) und Relationen zwischen ihnen vor, um die Sicht auf die Applikationsarchitektur zu definieren
- Typischerweise nutzt eine Applikation eine Mischung aus verschiedenen Stilen

7.2 Schichtenmodell

Architektur Stil – Layered Architecture:

- Klassische Architektur
- Software-Elemente organisiert in mehreren Schichten (*Layers/Tiers*)
- Jede Schicht hat eine genau festgelegte Verantwortlichkeit
- Beschränkt auch die Abhängigkeiten der Schichten voneinander – eine Schicht kann nur abhängen von
 - der direkt unter ihr liegenden Schichte (*strict layering*) oder
 - einer darunter liegenden Schicht

7.2 Schichtenmodell

Architektur Stil – Layered Architecture:

- Verbreitete **3-Tier-Architecture** als Beispiel:
 - *Presentation Layer* – Code der ein UI oder API implementiert
 - Business Logic Layer – Geschäftslogik
 - Persistence Layer – Implementiert die Interaktionslogik mit der Datenbank

7.2 Schichtenmodell

Architektur Stil – Layered Architecture:

- Nachteile:
 - Nur eine *Presentation Layer* – trägt dem Fakt nicht Rechnung, dass ein System sehr wahrscheinlich von mehreren Systemen aufgerufen wird
 - Nur eine *Persistence Layer* – trägt dem Fakt nicht Rechnung, dass wahrscheinlich mit mehr als nur einer Datenbank interagiert wird
 - Definiert die *Business Logic Layer* als abhängig von der *Persistence Layer* – theoretisch verhindert diese Abhängigkeit den Test der Geschäftslogik ohne Datenbank

7.2 Schichtenmodell

Architektur Stil – Layered Architecture:

- Nachteile:
 - Repräsentiert die Abhängigkeiten in einer gut entworfenen Anwendung nicht korrekt:
 - Business Logik definiert üblicherweise ein Interface oder ein Repository-Interface mit Datenzugriffsmethoden
 - *Persistence Layer* implementiert DAO-Klassen, die das Repository-Interface implementieren
 - Wirkliche Abhängigkeit ist umgekehrt zur dargestellten Abhängigkeit

7.3 Hexagon

Architektur Stil – Hexagonal Architecture:
Alternative zu Layered Architecture

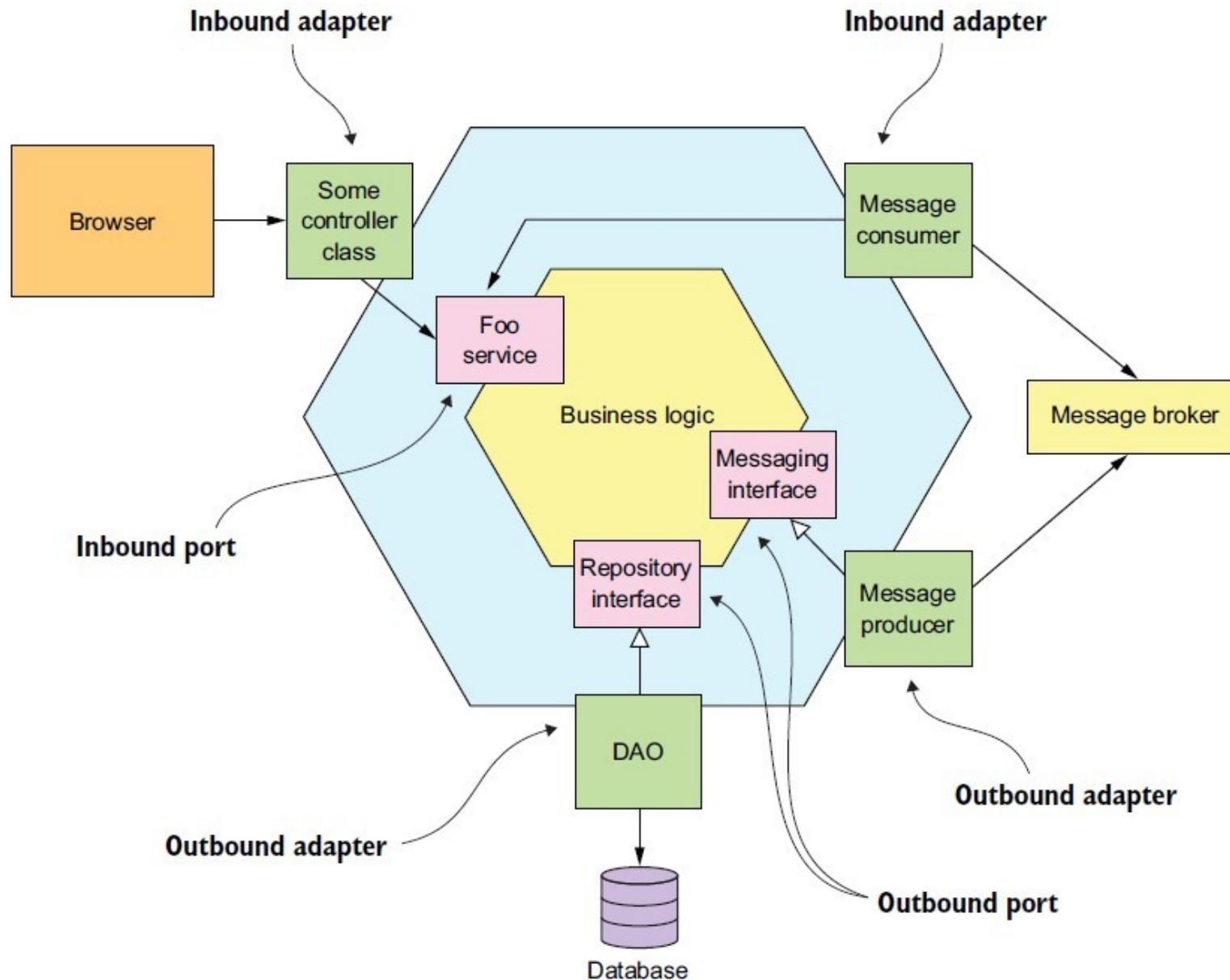
- Konsequente Weiterentwicklung des Schichtenmodells
- *Business Logic* ist im Zentrum
- Statt einer *Presentation Layer* mehrere *Inbound Adapter* die Anfragen von außen behandelt und die Geschäftslogik aufruft
- Statt einer *Persistence Layer* mehrere *Outbound Adapter* die von der Geschäftslogik aufgerufen werden und mit externen Systemen interagieren
- Geschäftslogik hängt nicht von den Adapters ab, sondern Adapter von der Geschäftslogik

7.3 Hexagon

Architektur Stil – Hexagonal Architecture:

- Geschäftslogik besitzt ein oder mehrere *Ports*
- *Port* definierte mehrere Methoden und bestimmt wie die Geschäftslogik mit der/n Schicht/en außerhalb interagiert
- *Port* ist meistens ein Interface
- *Inbound Port* – API das von der Geschäftslogik angeboten wird und Aufrufe von außen gestattet, z.B. ein Service-Interface
- *Outbound Port* – Art wie die Geschäftslogik mit externen Systemen interagiert, z.B. Repository-Interface mit seinen Datenzugriffsfunktionen

7.3 Hexagon



7.3 Hexagon

Architektur Stil – Hexagonal Architecture:

- Schicht um den *Business Logic* Kern herum besteht aus Adapters – *Inbound* und *Outbound*
- *Inbound Ports* behandeln Requests von außen über die Nutzung von *Inbound Ports*
- Mehrere *Inbound Adapter* können den gleichen *Inbound Port* benutzen
- Ein *Outbound Adapter* implementiert einen *Outbound Port* und reagiert auf Aufrufe durch die Geschäftslogik mit dem Ansprechen externer Applikationen oder Services, z.B.
 - Ein *Data Access Object (DAO)* das Zugriffsoperationen auf eine Datenbank implementiert
 - Eine Proxy-Klasse, die einen Remote-Service verfügbar macht
- *Outbound Adapter* können Events veröffentlichen

7.3 Hexagon

Architektur Stil – Hexagonal Architecture:

Vorteil

- Entkoppelt die Geschäftslogik von der Präsentations- und Datenzugriffslogik in den einzelnen Adapters
 - Geschäftslogik hängt weder von der Präsentations- noch der Datenzugriffslogik ab
- Leichteres, isoliertes Testen der Geschäftslogik
- Widerspiegelt gut die Architektur moderner Systeme – Geschäftslogik wird von vielen verschiedenen Adapters aus angesprochen (jedes implementiert bestimmte API's oder UI's) und spricht viele verschiedene externe Systeme über Adapter an
- Gut geeignet, um die Architektur eines jeden Service in einer Microservice-Architektur zu beschreiben

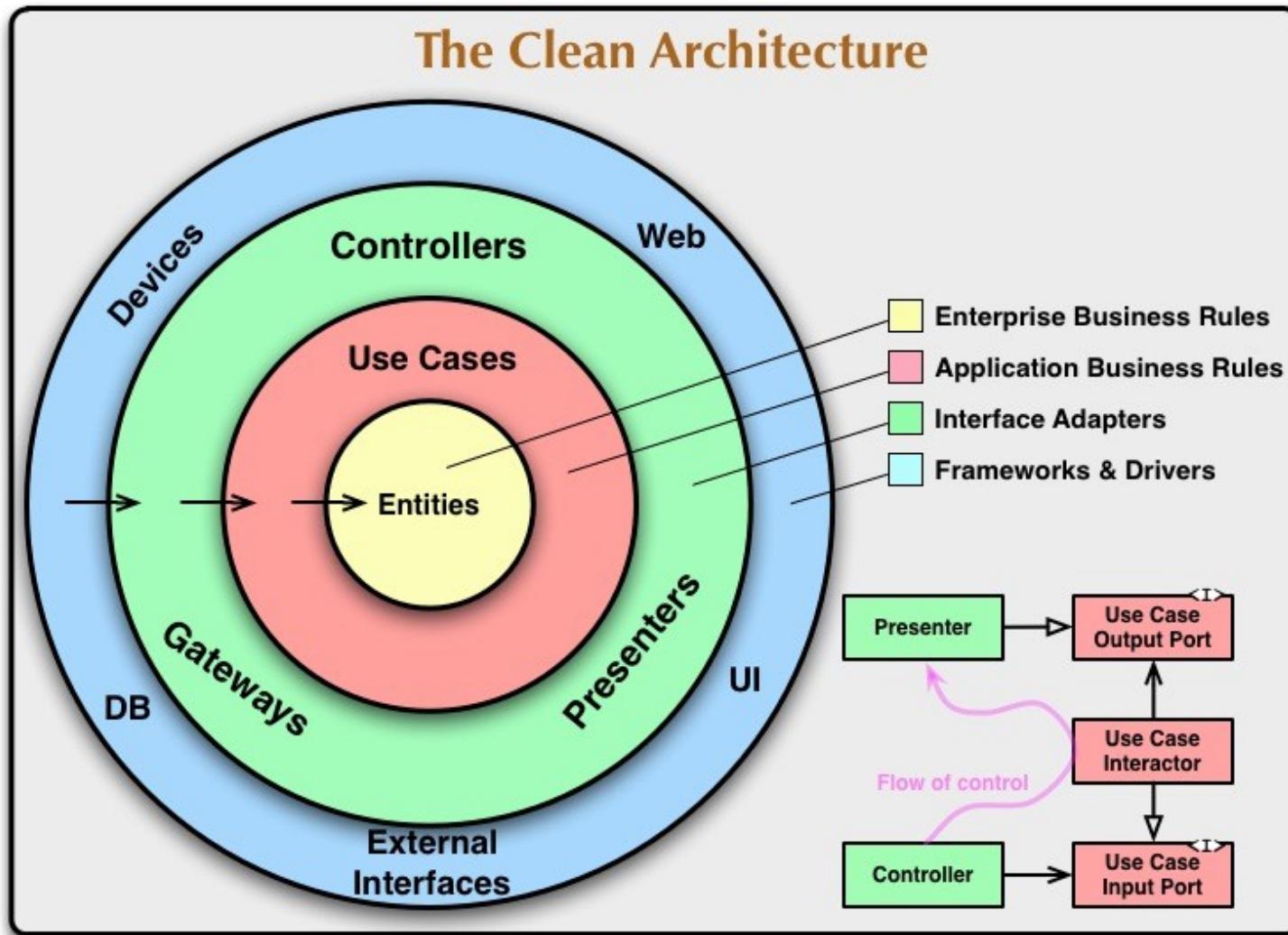
7.3 Hexagon

Architektur Stil – Hexagonal Architecture:

Öfter entwickelt worden und je nach Quelle bezeichnet als:

- *Hexagonal*
- *Ports and Adapters*
- *Onion* (Jeffrey Palermo)
- *Clean Architecture* (Robert C. Martin – *Uncle Bob*)

7.3 Hexagon



7.3 Hexagon

Architektur Stil – Hexagonal Architecture:

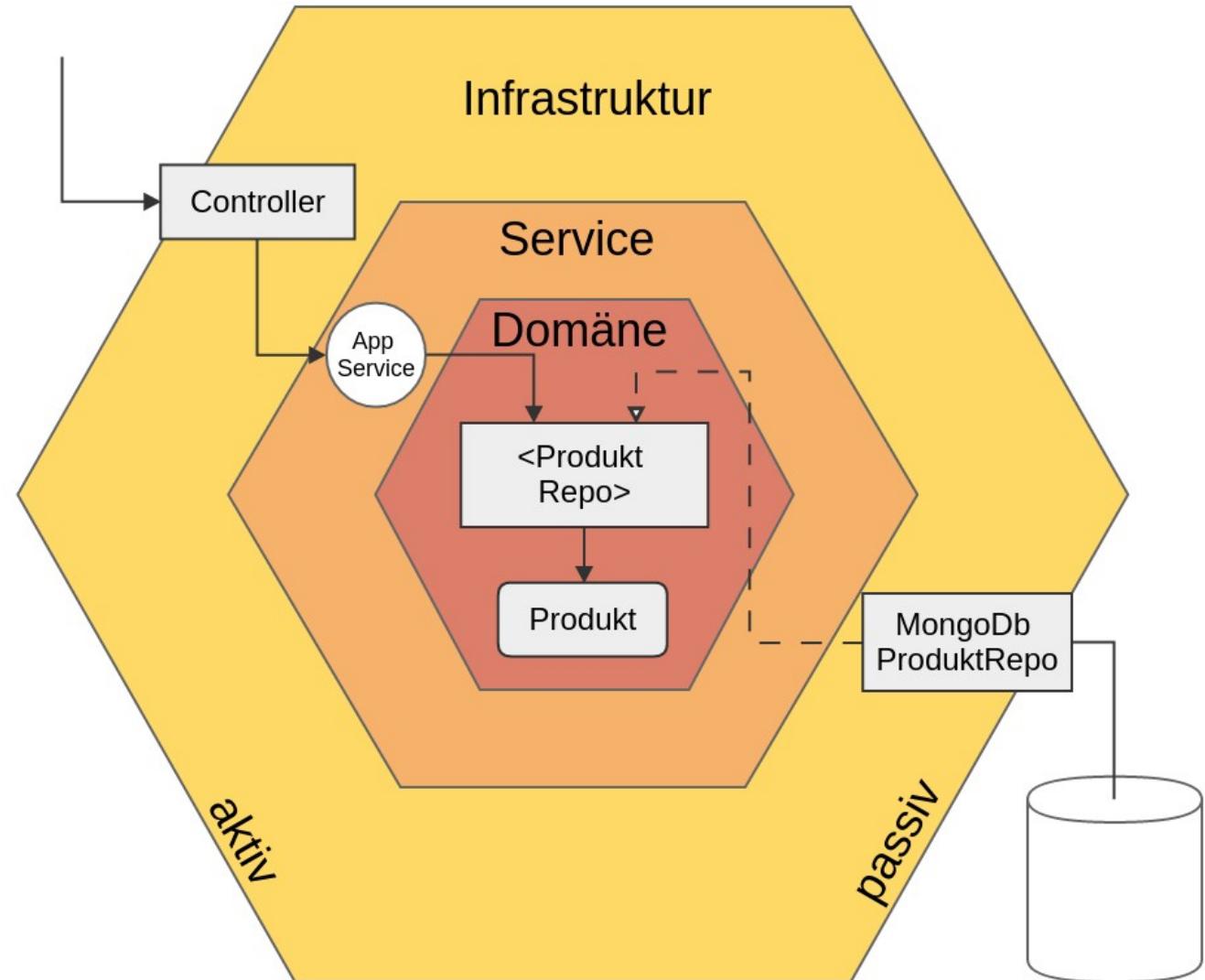
Einsatzbereich:

- Ähnlich wie DDD
- Langlebige Softwaresysteme mit aufwendiger fachlicher Komponente
- Wenn Nachhaltigkeit wichtig ist
- Nicht angezeigt für „Wegwerf-Software“, Prototypen, einfache CRUD-Systeme und Entwicklungen per RAD (Investment zu hoch)

7.3 Hexagon

Beispiel:

- Präsentations-schicht auf der linken Seite des Hexagons – aktive Seite/primäre Adapter
- Datenschicht auf der rechten Seite – passive Seite/ sekundäre Adapter
- Alles führt über das Zentrum



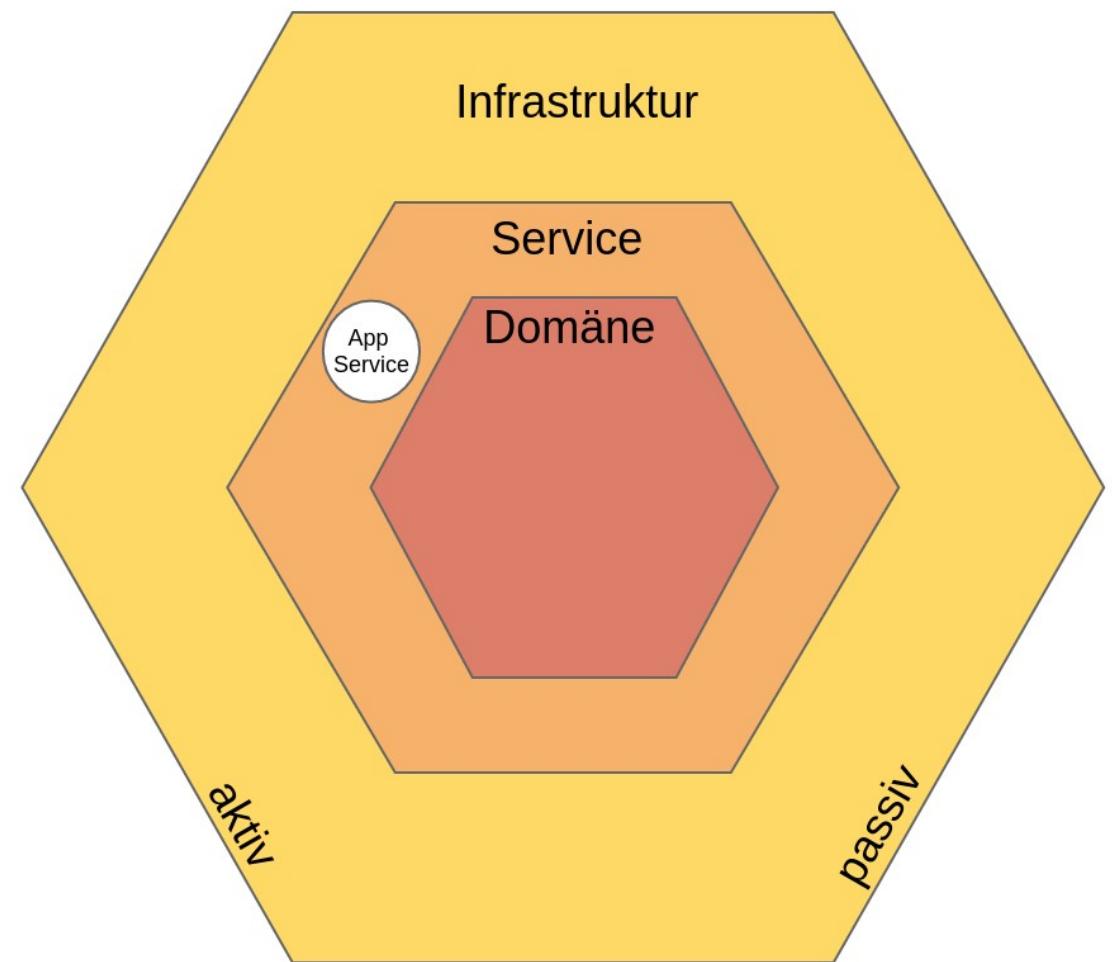
7.3 Hexagon

Beispiel – Wo ist ein Anwendungsfall zu ändern oder zu erstellen:

- Anwendungsfälle sind Methoden innerhalb eines Application Services → definiert die Schritte für den Anwendungsfall
- Application Services enthalten keine Details wie die Schritte auszuführen sind (gehört unbedingt zur Geschäftslogik)
- Orchestriert die Domänenobjekte der Geschäftslogik

7.3 Hexagon

Beispiel – Wo ist ein Anwendungsfall zu ändern oder zu erstellen:



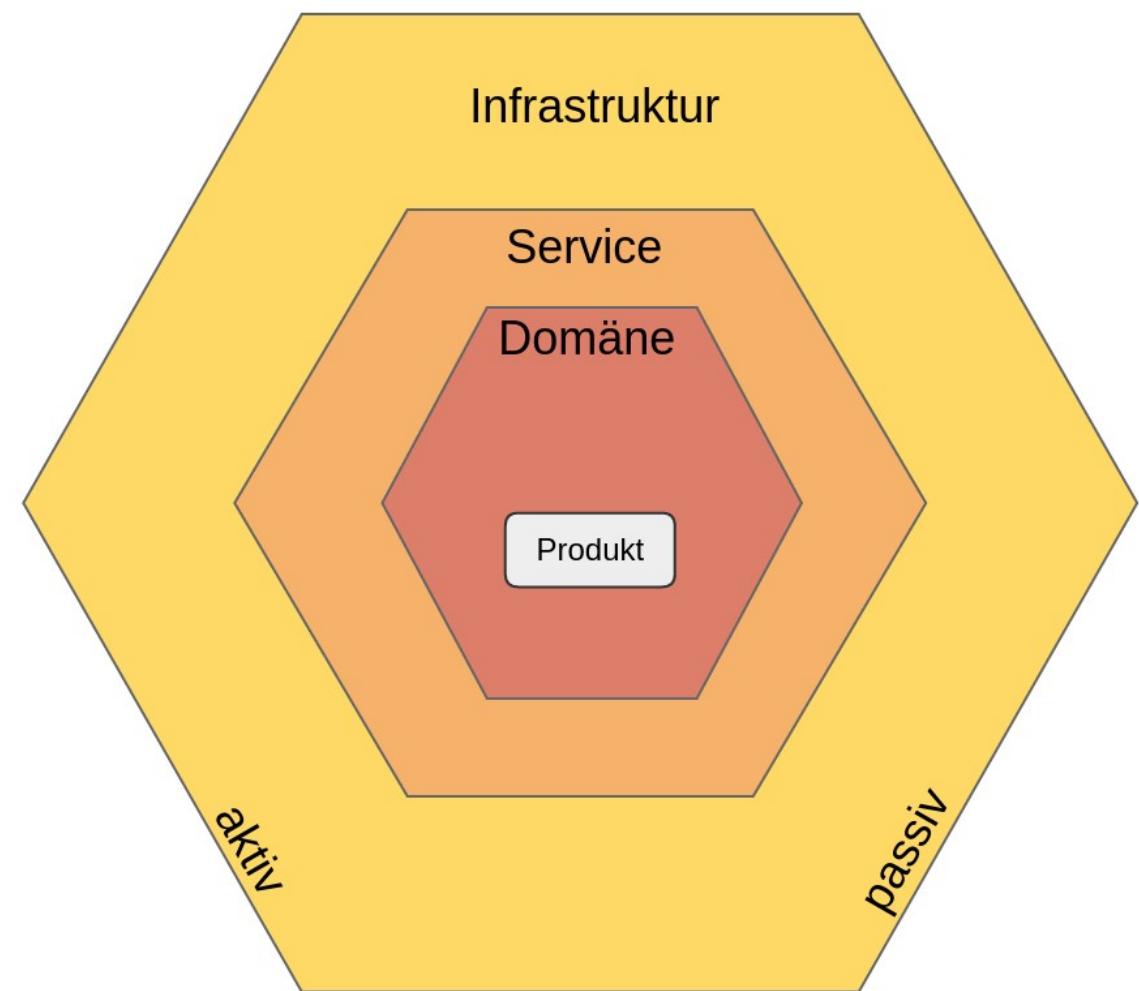
7.3 Hexagon

Beispiel – Wo wird die Geschäftslogik geändert oder erstellt:

- Entities und Value Objects (DDD!) der Domäne werden samt ihren Methoden im Kern des Hexagons angesiedelt
- Keine weiteren Abhängigkeiten als innerste Schicht → einfach per Unit-Tests zu testen

7.3 Hexagon

Beispiel – Wo wird die Geschäftslogik geändert oder erstellt:



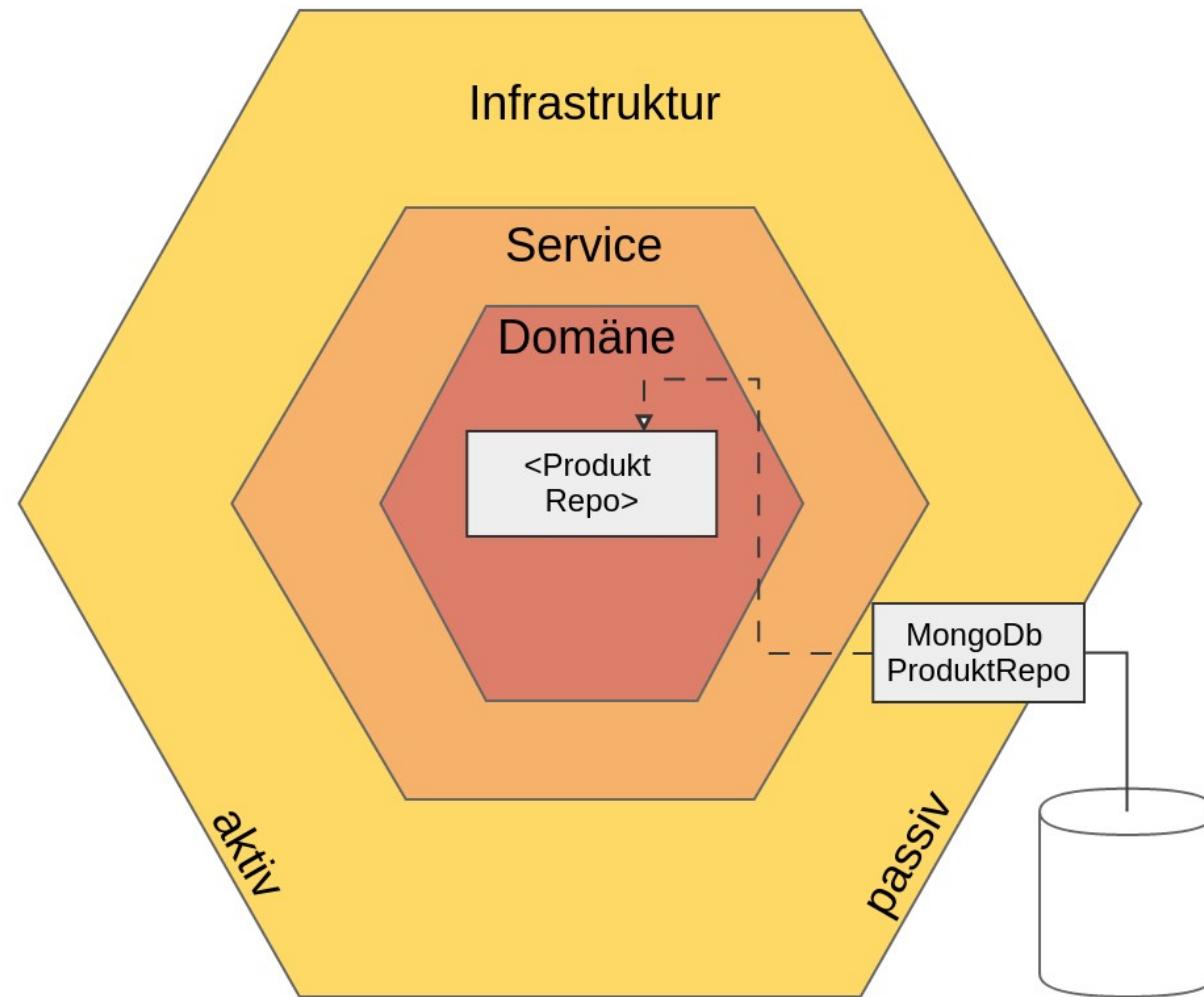
7.3 Hexagon

Beispiel – Wie ist der Zugriff auf externe Systeme:

- Externe Systeme werden abstrahiert
- Interface innerhalb der Domäne anlegen, das den Zugriff des Clients auf das externe System modelliert
- Implementierung des Interfaces (*Outbound Port*) durch passives Portadapter (*Outbound Adapter*)

7.3 Hexagon

Beispiel – Wie ist der Zugriff auf externe Systeme:



7.3 Hexagon

Beispiel – Stimmen die Abhängigkeiten:

- Abhängigkeiten sollen nur nach Innen zeigen
- Überprüfung anhand der Importanweisungen
- Klassen der modellierten Domäne
 - dürfen auch nur Klassen der Domäne importieren
 - Importe von Klassen außerhalb der Domäne dürfen nicht auftreten
- Klassen der Application Services:
 - Importieren nur Klassen der Domäne
 - Kein Import aus der eigenen Schicht, also keine Application Services (Anwendungsfall-Verschachtelung)

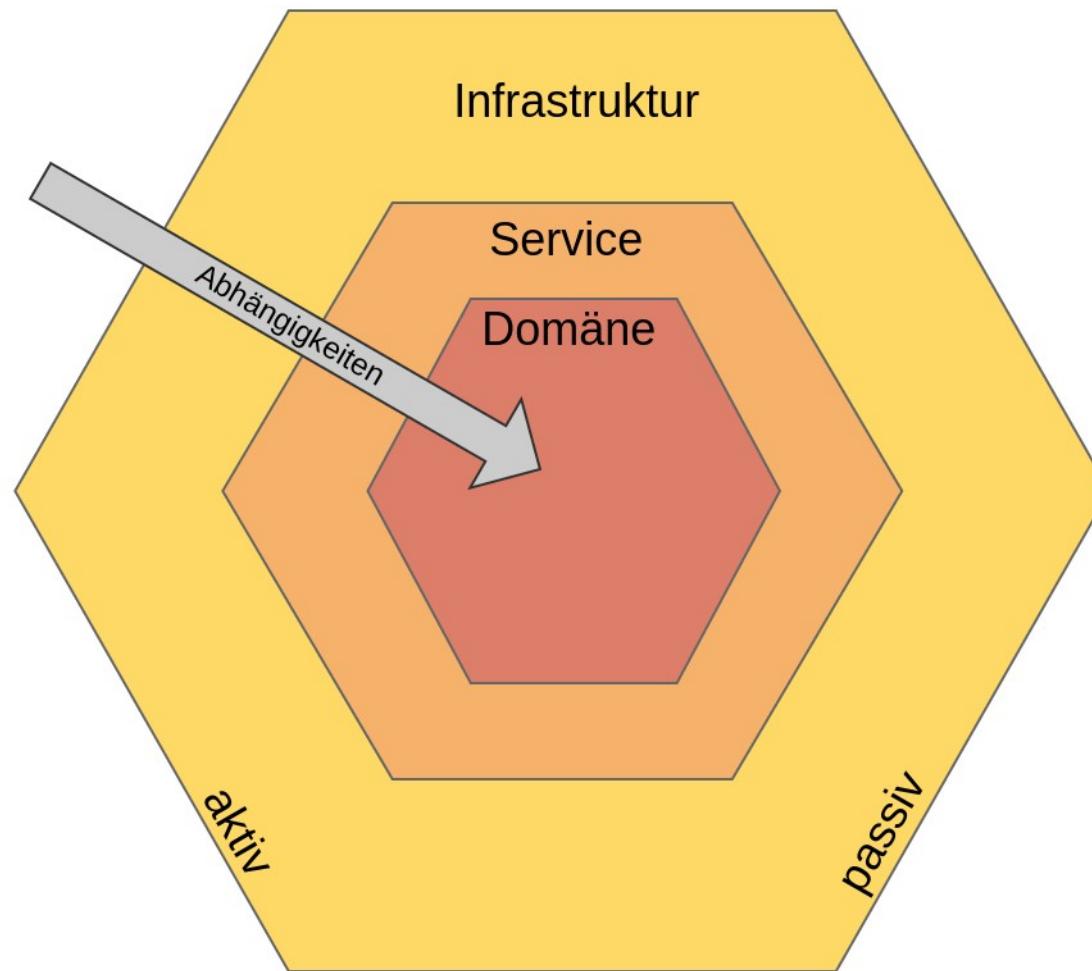
7.3 Hexagon

Beispiel – Stimmen die Abhängigkeiten:

- Klassen der Application Services:
 - Kein direkter Zugriff auf passive Portadapter
- Aktive Portadapter (linke Hexagon-Seite):
 - Import von Application Services bzw. Hilfsklassen zur Transformation von Objekten in der Service-Schicht
 - Import von Domänen-Klassen (Parameter für Application Services)
- Passive Portadapter (rechte Hexagon-Seite):
 - Kein Import von Application Service Klassen
 - Import der Interface-Definitionen der Domäne, die von diesen Adapters implementiert werden (Dependency Inversion)
 - Hilfsklassen für die Konvertierung aus der eigenen Schicht

7.3 Hexagon

Beispiel – Stimmen die Abhängigkeiten:



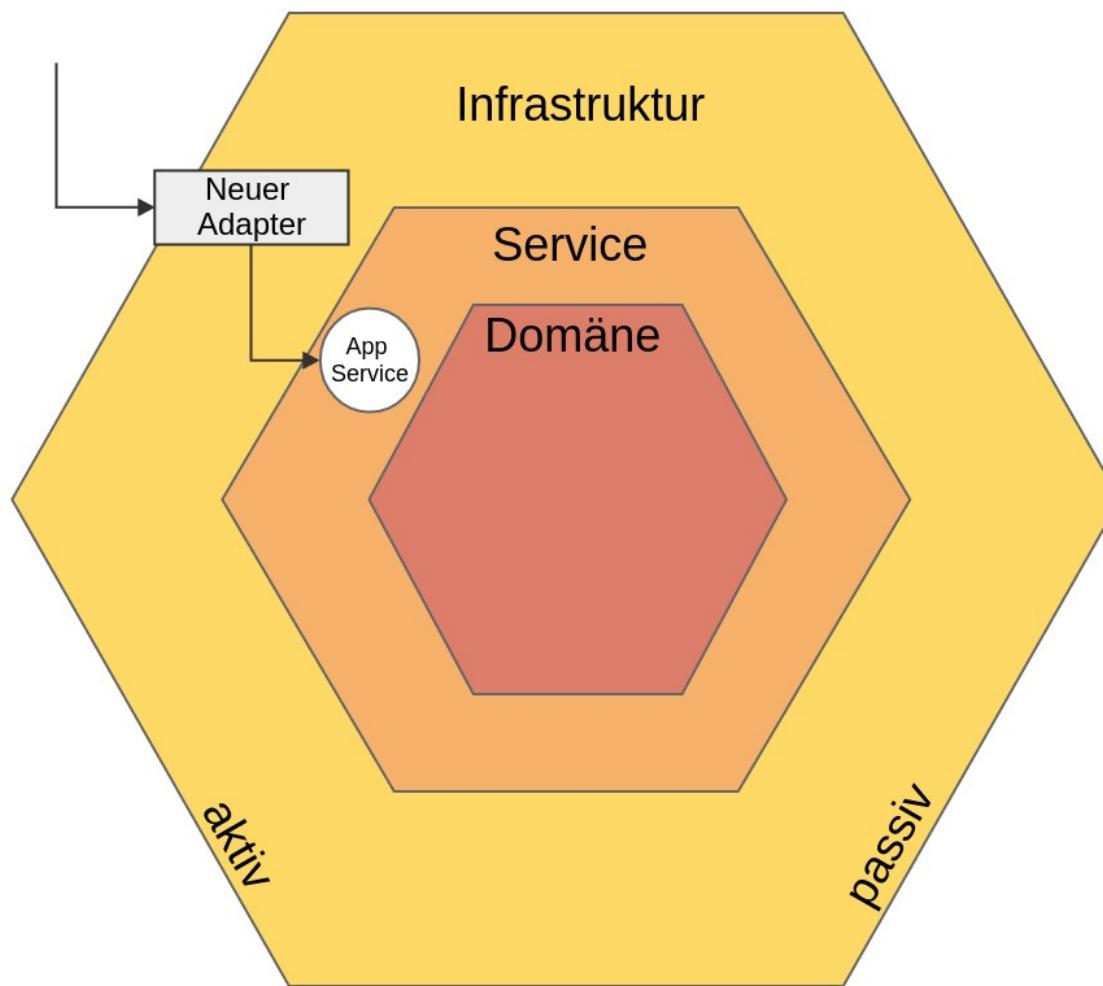
7.3 Hexagon

Beispiel – Wie erfolgt der Zugriff auf das System:

- Soll auf einen Anwendungsfall (Use Case) des Systems zugegriffen werden → Implementierung eines aktiven Adapters
- Adapter konvertiert Parameter in eine für die Methode des Application Service passende Form

7.3 Hexagon

Beispiel – Wie erfolgt der Zugriff auf das System:



7.4 Monolith

Architektur Stil – Monolithic Architecture:

Vorteil

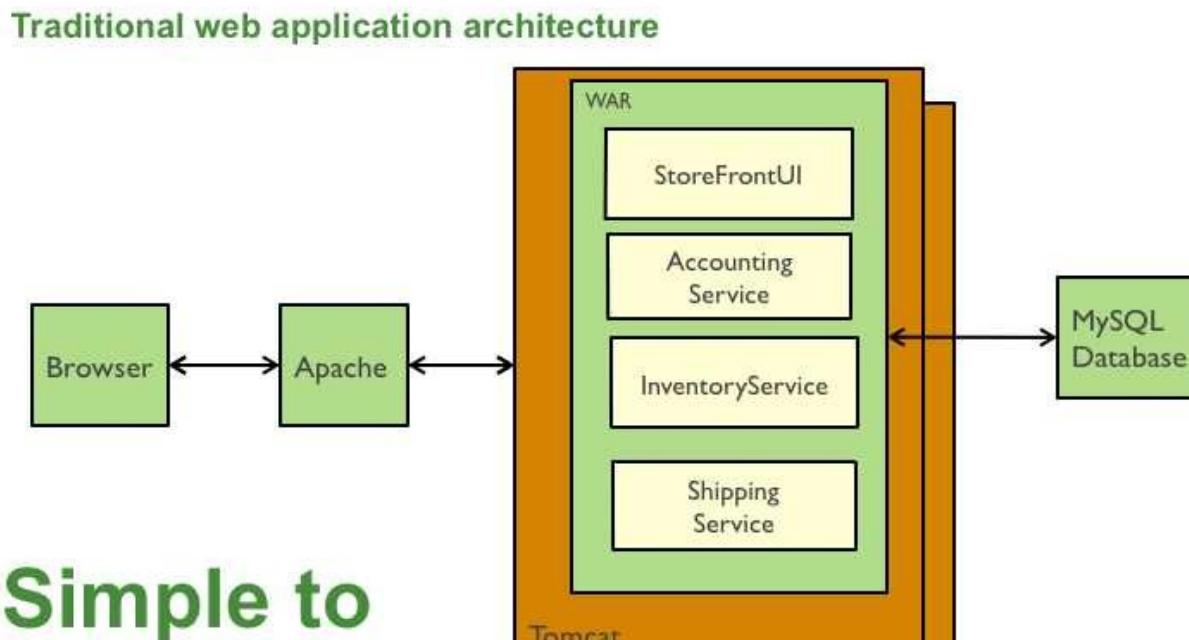
- Leicht zu entwickeln
- Leicht zu „deployen“
- Leicht zu „refactoren“
- Leicht zu skalieren

Nachteile – insbesondere wenn die Komplexität und Größe zunimmt

- Schwerer Code zu verstehen
- Überladene bzw. überlastete Entwicklungsumgebung
- CD/CI wird schwieriger
- Skalierung über einen bestimmten Punkt hinaus
- ...

7.4 Monolith

Architekturstil – Monolithic Architecture:



Simple to

**develop
test
deploy
scale**

7.5 Microservice

Architektur Stil – Microservice Architecture:

Vorteil

- Entkoppelte Bereiche – einfacher zu verstehen, da lokaler Kontext reicht
- Einzelkomponenten besser zu testen und zu deployen
- Kann fehlertoleranter sein
- ...

Nachteil

- Komplexität eines verteilten Systems – Implikation für Deployment, Test ... etc.
- Höherer Speicherverschnitt möglich – ein System durch N (z.B. virtualisierte) ersetzt
- ...

7.5 Microservice

Architekturstil – Microservice Architecture:

