## Methodik des Entwurfs

- Prinzipien des Software-Entwurfs
- Entwurfsmuster

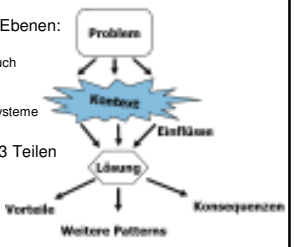
## Entwurfsmuster

- erprobte generische Lösungen für häufig auftretende Entwurfsprobleme
- allgemeine Wissens- und Kommunikationsbasis für Softwarestrukturen
  - "best practices"
  - so wie jeder oo-Entwickler weiß, was eine abstrakte Klasse ist,
  - weiß jeder, was ein Singleton oder eine Factory ist
  - kein Erklärungsbedarf beim Auftreten dieses Musters
- Ein MUSS für jeden OO-Entwickler



## Entwurfsmuster

- Konzept von den Architekten übernommen:
  - C.Alexander, A Pattern Language of Forms 1977
- Entwurfsmuster auf verschiedenen Ebenen:
  - Architekturmuster
  - Allgemeine Entwurfsmuster (und auch Analysemuster)
  - JEE-Entwurfsmuster
  - Entwurfsmuster für nebenläufige Systeme
- Jedes Entwurfsmuster besteht aus 3 Teilen
  - **Problem**
  - **Kontext**
  - **Lösung**





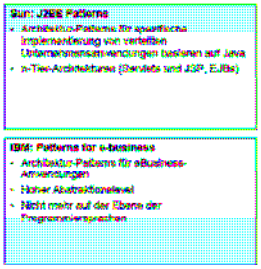

## Design Patterns

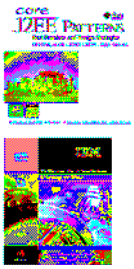
- Bekannteste Variante der Patterns
- Muster für die Umsetzung von Anwendungskomponenten
- Aufbau der Klassen mit Attributen und Methoden
- Kommunikation zwischen den Klassen
- Festlegung des dynamischen Verhaltens von Instanziierung bis Destruktion


- 23 Design-Patterns
- Low - level
- Beschreibung
  - Name
  - Problem
  - Lösung
  - Konsequenzen
- Wichtig für jeden OO-Entwickler



## Architektur-Patterns






## Weitere Patterns

Bekannte Veröffentlichungen

- **Martin Fowler: Analysis Patterns**
  - OO konzeptionelle Modelle
  - Spezielle Domänen
    - Gesundheitswesen,
    - Buchhaltung
    - Handel
    - Planung
  - Modellierung von Geschäftsanforderungen
- **Kent Beck: Coding Patterns**
  - Smalltalk-Patterns
  - Nützlich auch für andere Sprachen

## Vorteile von Design Patterns

- einfache, elegante Lösungen für spezifische Probleme
- vielfach getestet, revidiert, weiter entwickelt
- Flexibilität und Wiederverwendbarkeit praktisch etabliert

- erlernbar
- Wiedererkennung erleichtert Systemverständnis

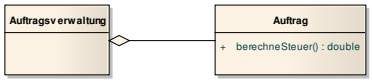
- Werkzeugunterstützung bei der Erstellung
- und beim Refactoring



## Entwurfsmuster - Einführung

Beispiel (nach Shalloway und Trott):

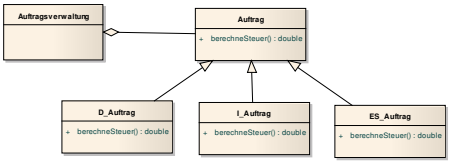
- In einem Verkaufssystem gibt es eine Komponente zur Verwaltung von Aufträgen, sowie Aufträge. Die Berechnung der fälligen Steuer liegt beim einzelnen Auftrag.






## Einführung – Schritt 2

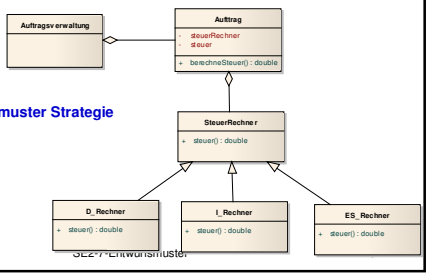
- Durch Ausdehnung der Geschäftsbeziehungen in andere europäische Länder sind verschieden Steuergesetze zu berücksichtigen.
- Problem: Dynamische Zuordnung von unterschiedlichem Verhalten
- Lösung 1: Vererbungshierarchie
  - Oberklasse von Auftrag spezifiziert die Schnittstelle abstrakt
  - Verschiedene Unterklassen konkretisieren sie






## Einführung - Schritt 3

- Die Unterklassen unterscheiden sich nur in einem Aspekt.
- Für jedes neue Land muss eine neue Unterklasse programmiert werden.
- Problem: Dynamische Zuordnung eines Teilverhaltens, erweiterbare Zahl von Teilverhalten




- Lösung: **Entwurfsmuster Strategie**



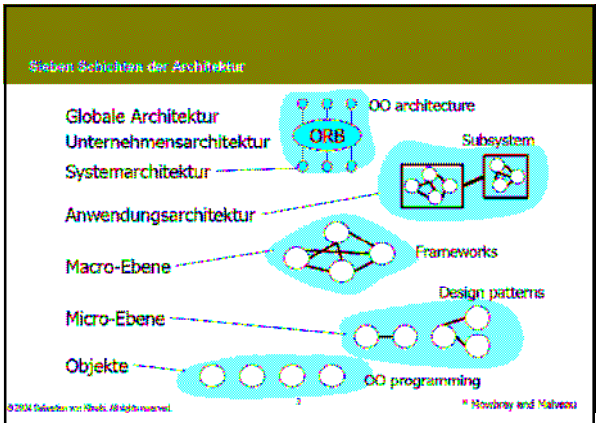
## Entwurfsprinzipien

- Entwurfsmuster helfen dabei, Entwurfsziele umzusetzen.

- Zunächst brauchen wir Ziele und Kriterien
→ Entwurfsprinzipien

- Grundprinzipien auf verschiedenen Ebenen:
  - Prinzipien des Architekturentwurfs
  - Prinzipien des Strukturentwurfs
  - Prinzipien des Klassenentwurfs





## Die Schichten (Niveaus) im einzelnen:

- Systemarchitektur:
  - Integration unterschiedlicher Systeme
  - teils auf unterschiedlichen Plattformen (heterogen)
- Anwendungsarchitektur
  - Zerlegung in / Integration von Subsysteme(n)
  - Bildung funktionaler Komponenten
  - Bildung von Abstraktionsschichten
- Macro-Architektur
  - Standardarchitekturen für Komponenten/Schichten
  - Einsatz von Frameworks
- Micro-Architektur
  - Lokale Standardstrukturen: Entwurfsmuster (Desgn Patterns)
- Objektentwurf
  - Programmiersprache
  - Kapselung, Vererbung, getter/setter, ...



## *Literaturempfehlung:*

- Ludewig, Lichtner: Software Engineering , dpunkt 2007
- Starke, Effektive Software-Architektur, Hanser 2008



## Software-Entwurf

- Software-Entwurf ist Strukturentwurf auf verschiedenen Ebenen
- Schwerpunkt: die drei mittleren Ebenen
  - Anwendungsarchitektur
  - Komponente (Gliederungseinheit des Systems)
  - Modul (Gleiderungseinheit der Programmierung)
- Ziel ist die Beherrschung der Komplexität, also Einfachheit:
  - "Wenn du es nicht in fünf Minuten erklären kannst, hast du es nicht verstanden oder es funktioniert nicht." *Rechtin*
- Grundtechniken sind Zerlegung und Kapselung



## Grundformen der Zerlegung

- Horizontal:
  - "in Scheiben schneiden"
  - Schichtung
  - Abstraktionsebenen
  - jede Schicht baut auf der darunter liegenden auf
- Vertikal:
  - "in Stücke schneiden"
  - jedes Teil übernimmt eine benennbare fachliche oder technische Funktion
- Kapselung
  - in jedem Fall die Teile (Komponenten) als Black Box betrachten
  - kommunizieren mit der Umgebung über klar definierte Schnittstellen



## Hauptprinzip des Architektur-Entwurfs

Eine gute Software-Architektur verfügt über:

- Starke Kohäsion
  - "Unteilbarkeit" der Komponenten
- Schwache Kopplung
  - leichter Austausch von Komponenten

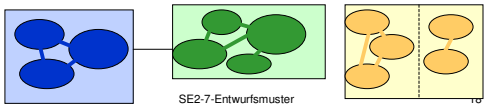
... und ist verstehbar ☺

- Referenzarchitekturen
- Best Practices



## Lose Kopplung und starke Kohäsion

- Optimierung von Abhängigkeiten:
  - Flexibilität (Veränderungen können lokal vorgenommen werden)
  - Stabilität (Änderungen wirken sich nur lokal aus)
- Kopplung
  - Abhängigkeit zwischen Moduln
  - sollte möglichst gering sein: Austauschbarkeit, Veränderbarkeit
- Kohäsion
  - Abhängigkeit innerhalb eines Moduls, innerer (logischer) Zusammenhang
  - sollte stark sein → sonst Modul teilen



## Grundprinzipien des Strukturentwurfs

- Einfachheit vor Allgemeinverwendbarkeit
- Minimale Verwunderung
- DRY – don't repeat yourself
- Getrennte Verantwortung (Separation of Concerns)
- Offen-Geschlossen-Prinzip (OGP):
  Offen für Erweiterungen – geschlossen für Veränderungen

- Zu Prinzipien gewordene Maßnahmen:
  - Umkehrung der Abhängigkeiten
    (Abhängigkeit von Abstraktionen, nicht Implelentierungen)
  - Aufteilung der Schnittstellen
  - Solide Annahmen (oder besser keine!)
  - Gemeinsame Wiederverwendung von Paketen
  - Auflösung zirkulärer Abhängigkeiten
  - Prinzip der stabilen Abhängigkeiten



## Grundprinzipien des Klassenentwurfs

- Kapseln
  - Daten + zugehöriges Verhalten
  - aber: kleine Kapseln (keine allmächtigen Klassen)
  - Veränderliche Aspekte verbergen
  - "schmutzige" Lösungen verbergen
- Schnittstellen
  - Schlanke Schnittstellen, möglichst "privater" Entwurf
  - Benutzer dürfen nur von der öffentlichen Schnittstelle abhängen
  - Klasse muss unabhängig von ihren Benutzern bleiben
  - Klasse darf keine Annahmen über ihren Nutzungskontext machen
  - lange Argumentlisten vermeiden, ggf. Aggregate verwenden
- Vererbung
  - Mehr als eine Instanz der Unterklasse (sonst Objekt, nicht Unterklasse)
  - Strukturelle Ähnlichkeit zur Realwelt anstreben
  - Keine explizite Typabfrage, sondern Polymorphie
  - Liskov's Substitutionsprinzip:
    Funktionalität der Oberklasse weitestgehend erhalten



## Einfachheit

- Zerlegung in 5 +/- 2 Komponenten je Einheit
  - ggf. hierarchische Verfeinerung
- Im Zweifelsfall die weniger komplexe Alternative wählen
- Entwurf nach Verantwortlichkeiten:
  - Trennung von Technik und Fachlichkeit
  - Komplexe Diagramme nach Verantwortlichkeiten in einfachere unterteilen
- Konzentration auf Schnittstellen:
  - Details bleiben gekapselt
- Robuste Komponenten planen:
  - Fehler berücksichtigen
  - Fehlervermeidung durch Verständlichkeit
  - Auswirkungen von Fehlern möglichst lokal halten
  - Robustheit komponentenweise planen



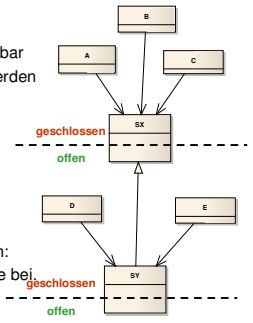
## Kapselung

- Ein Modul gibt nur die Informationen preis, die der Benutzer wirklich benötigt.
- Vorteile:
  - Bessere Kontrolle innerhalb des Moduls
  - Vermeidung von Benutzerfehlern und -angriffen
  - Lokalisierung von Fehlersituationen
  - Leichtere Austauschbarkeit
- Nachteil:
  - Overhead beim Zugriff
  - zusätzliche Komponenten und zusätzlicher Code



## Das Offen-Geschlossen-Prinzip (OGP)

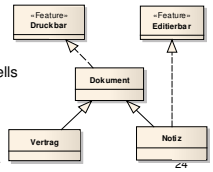
- geschlossenes Modul:
  - Schnittstelle ist stabil, nicht veränderbar
  - kann ohne Anpassung verwendet werden
- offenes Modul:
  - Erweiterungen sind möglich
- ein OGP-Modul
  - ist offen für Erweiterungen
  - aber geschlossen für Veränderungen:
    behält seine bestehende Schnittstelle bei.





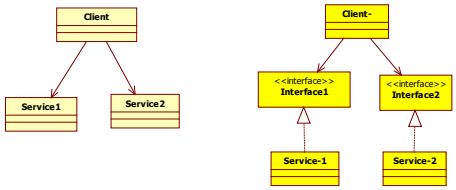

## Trennung von Zuständigkeiten

- Grundprinzip:
  - Jede Komponente sollte nur einen genau umrissenen Aufgabenbereich abdecken
- Auftrennung von Schnittstellen
  - Jede Schnittstelle sollte nur einen Aufgabenbereich abdecken
  - ggf. Implementierung mehrerer Schnittstellen durch dasselbe Modul
  - dadurch entstehen spezifische Abhängigkeiten
- Trennung von Funktion und Interaktion
  - entspricht der Isolation des fachlichen Modells
  - ermöglicht separate Veränderungen
  - MVC ☺



## Umkehrung der Abhängigkeiten:
### Allgemeines Schema

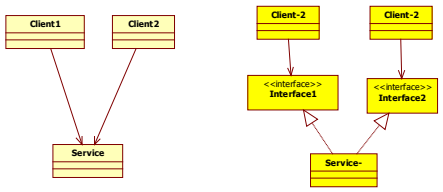
- Abhängigkeit von Abstraktionen, nicht Implementierungen





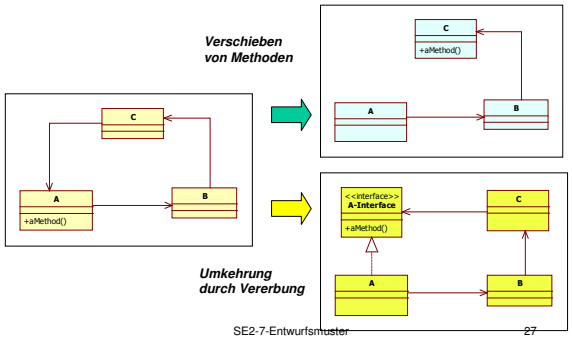

## Aufteilung der Schnittstellen:
### Allgemeines Schema:

- Clients sollen nicht von Diensten abhängen, die sie nicht benötigen.
- Interfaces nach den Client-Erfordernissen entwerfen, nicht nach den Klassen-Angeboten:






## Auflösung zirkulärer Abhängigkeiten:
### Allgemeines Schema






## GoF-Entwurfsmuster

- Die wichtigsten GoF-Entwurfsmuster (Gang of Four)
  - zunächst nach Entwurfsprinzipien geordnet
- Systematik der Entwurfsmuster üblicherweise nach Wirkung:
  - Erzeugungsmuster
  - Strukturmuster
  - Verhaltensmuster



## GoF Design Patterns

- Creational patterns
  - Umgang mit Initialisierung und Konfiguration von Objekten/ Klassen
- Structural patterns
  - Umgang mit Schnittstellen und Kommunikation zwischen Objekten/ Klassen
- Behavioral patterns
  - Umgang mit dynamischen Interaktionen zwischen Gruppen von Objekten/ Klassen

| Creational | Structural | Behavioral |
|---|---|---|
| Factory Method<br>Abstract Factory<br>Builder<br>Prototype<br>Singleton | Adapter<br>Bridge<br>Composite<br>Decorator<br>Facade<br>Flyweight<br>Proxy | Chain of Responsibility<br>Command<br>Iterator<br>Mediator<br>Memento<br>Observer<br>State<br>Strategy<br>Visitor |





## Lesestoff...

Thomas Ziemer hat ein lesenswertes Skript über GoF-Muster verfasst:

http://www.ziemers.de/downloads/tfh/swp/script/swp2/03Entwurfsmuster_(Erster_Teil).pdf

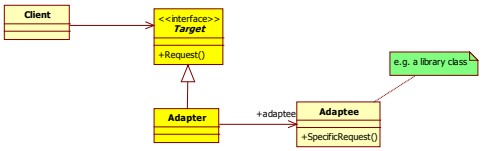
http://www.ziemers.de/downloads/tfh/swp/script/swp2/03aEntwurfsmuster_(Zweiter_Teil).pdf

http://www.ziemers.de/downloads/tfh/swp/script/swp2/04Model-View-Controller-Konzept.pdf

*access se-es*

## Kapselung von Abhängigkeiten: Adapter

- Zweck: ***Strukturmuster***
  - Anpassung einer gelieferten Schnittstelle an eine erwartete
  - Wiederverwendung von Klassen trotz leichter Inkompatibilitäten
  - Verwendung von Bibliotheken
- Kontext:
  - Klasse hat die gewünschten Daten und das gewünschte Verhalten
  - Schnittstelle passt nicht (Namen, Parameter, Ausnahmen...)






## Adapter

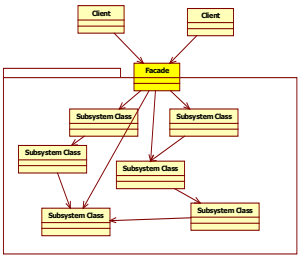
- Vorteile:
  - Wiederverwendung
  - Kapselung von Abhängigkeiten, z.B. Anbindung an spezifische Fremdsoftware
- Nachteile:
  - zusätzlicher Aufwand durch Delegation
  - ggf. performance-relevant bei Übergabe großer Objektstrukturen, falls diese kopiert werden müssen.
- Alternativen:
  - Fassade, Wrapper, Proxy



## Kapselung von Abhängigkeiten: Fassade

***Strukturmuster***

- Zweck:
  - Vereinfachter Zugriff auf en komplexes Subsystem
  - Kapseln der inneren Struktur
- Kontext:
  - Verwendung eines Subsystems mit wichtigen inneren Abhängigkeiten
  - Verwendung eines Subsystems, bei dem die Gesamtfunktionalität sich aus den Schnittstellen mehrerer Klassen zusammensetzt
  - Alternative Subsysteme mit unterschiedlicher Struktur
  - "Querschnitt-Aufgaben" wie Logging, Transaktionen oder Zugangskontrolle





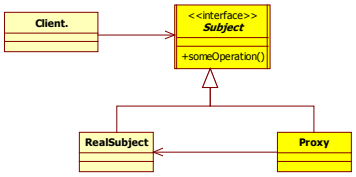

## Fassade

- Vorteile:
  - Clients von den Details des Subsystems entkoppelt
  - Austauschbarkeit von Subsystem-Schnittstellen und –Komponenten
- Nachteile:
  - Clients können Fassade umgehen
  - einige Typen müssen evtl. zusätzlich extern verfügbar bleiben
  - Fassade muss an jede Änderung im Subsystem angepasst werden (erhöhte innere Abhängigkeit)
- Varianten
  - Session Facade, vgl. JSF
  - Message Facade – asynchroner Zugang zu einem Subsystem



## Kapselung von Abhängigkeiten: Proxy

- Zweck: ***Strukturmuster***
  - Möglichkeit, ein Objekt durch ein Stellvertreter-Objekt (zeitweise) zu ersetzen
- Kontext:
  - Zugang zum echten Objekt schwierig oder teuer
  - Direkter Zugang zum echten Objekt aus Sicherheitsgründen unerwünscht





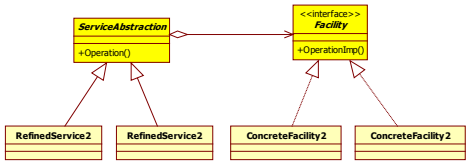

## Proxy

- Vorteile:
  - Implementierung von Aspekten, die mit der Fachlogik nichts zu tun haben:
    - Sicherheitsabfrage, Lazy Loading, Protokollierung, Caching,...
  - Erweiterung des Objektverhaltens in einem bestimmten Kontext
    - Verhaltenserweiterung von Bibliotheksklassen ohne Ableitung
- Nachteile:
  - Methode gleicher Signatur im Proxy – redundanter Code
- Typische Verwendungen:
  - RemoteProxy
  - VirtualProxy
  - ProtectionProxy
- Varianten:
  - Dynamic Proxy
    wird zur Laufzeit durch Reflection erzeugt

## Kapselung von Änderungen: Bridge

*Strukturmuster*

- Zweck:
  - Trennung von funktionaler Abstraktion und Implementierung
  - so dass beide ***unabhängig voneinander verändert*** werden können
- Kontext:
  - Eine Klasse bietet eine bestimmte Leistung (Service) an *Beispiel: Transport*
  - Sie benutzt dazu eine andere Klasse als "Betriebsmittel" (Facility) *Beispiel: Transportmittel*
  - Sowohl Service als auch Facility sollen unabhängig weiterentwickelt werden können





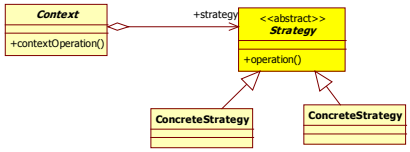

## Bridge

- Vorteile:
  - Beliebige Kombinationen
  - Funktionale Abstraktionen und konkrete Implementierungen beliebig mischbar
  - Für den Client transparent
  - geringer Aufwand für das Pattern
- Nachteile:
  - (keine bekannt)



## Open-Closed-Priciple: Strategy (schon bekannt)

- Zweck:
  - Austauschbarkeit von Algorithmen zur Laufzeit
  - unabhängig von den nutzenden Clients
- Kontext:
  - Ein Dienstmerkmal soll von außen steuerbar ausgetauscht werden können
  - sehr wichtig z.B. für Internationalisierung: länderspezifische Funktionalität






## Strategy

*Verhaltensmuster*

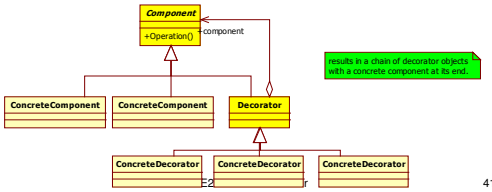
- Vorteile:
  - Client unabhängig von konkreten Implementierungen
- Nachteile:
  - Clients müssen die unterschiedlichen Strategien kennen.
- Varianten:
  - Policy
    - mehr als eine Operation



## Open-Closed-Principle: Decorator

*Strukturmuster*

- Zweck:
  - Dynamisches Hinzufügen von Funktionalität zu einer Komponente
  - Grundfunktionalität wird erhalten und ergänzt
- Kontext:
  - Reversible, konfigurierbare Modifikation des Verhaltens eines existierenden Objekts
  - Direkt angesprochen, hat das Objekt das Standardverhalten (Gegensatz zu Strategie)






## Decorator

- Vorteile:
  - Funktionalitäten zur Laufzeit zu- und abschaltbar
  - Komponenten kennen ihre Dekorierer nicht
- Nachteile:
  - möglicherweise viele ähnlich aussehende Klassen, wenig übersichtlich
- Varianten:
  - Pipe and Filter (s. Analysemuster)
  - Unterschied: Kette von abgeschlossenen Operationen anstelle von Delegationskette.

## Open-Closed-Principle: Umsetzende Entwurfsmuster

- Änderungen nicht durch Codemodifikation,
- sondern durch Hinzufügen weiterer Klassen

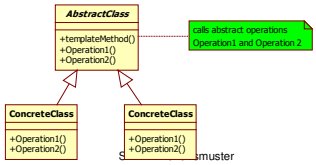
- Strategy
  - Die Aufgabe wird delegiert an ein Objekt, von dem zunächst nur die Schnittstelle bekannt ist.
  - Die Auswahl erfolgt dynamisch
  - Verfügbare Strategien sind zur Compilezeit bekannt (i.d.R.)

- Decorator, Pipe and Filter
  - Änderungen "außen angefügt", Objekt selbst bleibt unverändert.

- PlugIn
  - gewissermaßen Strategy auf Systemebene
  - Software dynamisch ergänzbar durch weitere Komponenten
  - Hinzufügen durch Konfiguration (keine Neukompilation)
  - benötigt Factory Method, Abstract Factory und Registry

## DRY: Template Method

- Zweck: ***Verhaltensmuster***
  - Verwendung der Struktur eines Algorithmus mit verschiedenen Einzeloperationen
  - "Generischer" Algorithmus
- Kontext:
  - Ein Algorithmus besteht aus einer globalen Ablaufstruktur (z.B. Iterieren über eine Liste)
  - und innerhalb dieser Struktur Detailoperationen (z.B. Verdoppeln des Elementwerts)
  - Die Ablaufstruktur ist unabhängig von den Detailoperationen und wird an verschiedenen Stellen benötigt.





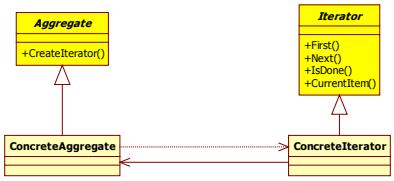

## Template Method

- Vorteile:
  - Vermeidung redundanter Algorithmenstrukturen (DRY)
  - Oberklasse ruft Methoden ihrer Unterklasse
    - Grundmuster für IoC
    - verbessert Wiederverwendbarkeit



## Kapselung von Abhängigkeiten: Iterator

- Zweck: ***Verhaltensmuster***
  - Ursprünglich in Kombination mit Compositum
  - Iteration über die Struktur ohne Kenntnis der Implementierung mit der Möglichkeit, die in der Schnittstelle angegebene (für Knoten und Blätter implementierte) Methode an beliebiger Stelle auszuführen.
  - Verallgemeinert für traversierbare rekursive Strukturen.






## Iterator

- Vorteile:
  - Aggregat hat schlankes Interface
  - Client benötigt keine Kenntnisse über den Aufbau des Aggregats.
  - Verschiedene Traversierungsstrategien sind möglich

- Nachteile:
  - parallele Änderungen der Struktur (Einfügen / Entfernen von Einträgen) sind problemetisch

  → evtl. *zu weit* von der Struktur abgekoppelt!



## Vereinendes und Trennendes ...

- Die heute behandelten Muster dienen alle der strukturellen Entkopplung
  - schaffen kleinerer Einheiten, die separat in ihren Kontext eingebracht werden können
  - "Einfangen" von Abhängigkeiten
  - "Lokalisierung" von Änderungen und Anpassungen
- Ziel ist vor allem
  - Auswirkungen von geänderten Anforderungen oder Entwurfsentscheidungen zu begrenzen
- Erwünschter Zusatzeffekt:
  - getrennte Entwicklung
  - getrennte Qualitätssicherung

**...genug für heute ...**

**Die nächtsen Male:**

- **wichtige Entwurfsmuster im Einzelnen**
- **Kombination von Entwurfsmustern**