Prinzipien

- abstraktion (modellbildung, vernachlässigung von details); 3 Merkmale:
  - Abbildungsmerkmal: Bildet reelles/fiktives ab => Realitätsbezug
  - Verkürzungsmerkmal: hebt Wesentliches hervor, lässt Unwesentliches weg
  - Pragmatisches Merkmal: Kann unter bestimmten Bedingungen original ersetzen
- Strukturierung: komplex => reduzierte Darstellung mit gleichem character und spezifischen Merkmalen
- Dekomposition (Stepwise Refinement, Zerteilung eines problems)
- Hierarchisierung (Erstellen einer Rangordnung => Hierarchieebenen; Spezialfall Strukturierung)
- Wiederverwendung (technische,organisatorische Vorraussetzungen; $\tilde{6}0\%$ Mehraufwand)
- Standardisierung (Festlegung/Einhaltung von Richtlinien, Normen)
- Verbalisierung (Ideen geeignet in Worten ausdrücken, z.B. Namen)
- Modularisierung (Zusammenbau des Systems aus Bausteinen s.U.)

Modularität

- hohe Kohäsion (Abhängigkeiten im Modul,eine Verantwortung)
- lose Kopplung (Abhängigkeiten zwischen modulen)
- Information Hiding (Geheimnisprinzip)
- Schnittstellenspezifikation
- Kontextunabhängigkeit (Analog zu lose Kopplung zur Umgebung
- Lokalitätsprinzip (Alle Bestandteile zu Problemlösung an einer Stelle)

=> Änderungsfreundlichkeit, Wartbarkeit, Standardisierung, Arbeits- Organisation und Planung, Überprüfbarkeit

Modulare Operatoren:

- Splitting: Aufteilen eines Moduls
- Substituting: Austausch von Modul durch Modul gleicher Funktion
- Augmenting: Hinzufügen von Modul für Funktionserweiterung (leicht)
- Excluding: Entfernen von Modul für Funktionsverringerung (schwer)
- Inverting: Herausziehen von redundanter Funktionalität aus Modulen in höheres Modul
- Porting: Verwenden eines Translators für ein Modul in nicht vorhergesehenem Kontext

Architectural Erosion: Verletzungen der Architektur. Änderungen => inkonsistenter Code (entfernen tragender Säulen).

Architectural Drift: Erweiterung ohne Beachtung der bestehenden Architektur (Anbauten).

Gegenmaßnahmen: Dokumentation, Open-Closed-Prinzip, Kapselung, Modularisierung

Strukturparadigmen

Orhogonal zu Programmierparadigmen

- Unstrukturiert (Spagetti-code, goto)
- Strukturiert (Blöcke (z.B. if))
- Modular (sammlung von Typen/Prozeduren, explizite schnittstellen)
- Objektorientierung (Kapselung von Daten/Methoden in Objekten)

CleanCode Prinzipien

Fail-Fast(Exceptions statt Rückgabe),
Command-Query-Separation(nicht Zustandänderung u. -abfrage in ener Methode),
Least Astonishment (tu das was signatur beschreibt) ,
unnötiger Boilerplate-code (Boilerplate = zus. Aktionen vor/nach Funktionsaufruf) ,
defensive Kopien (Kopien bei get/set-Aktionen, incl. Konstruktor)

Objektorientierung

Prinzipien eines wiederverwendbaren OO Designs:

- Program to an Interface, not an Implementation
- Favor delegation over class inheritance

Vererbung:

Arten:

Realisierung: Interfacevererbung => nur Subtyping
Generalisierung/Spezialisierung: Classenvererbung => Subtyping und Subclassing

Facetten:

Subtyping: Vererbung von ähnlichem Verhalten/Contractfragmenten (Signaturen)
Subclassing: Vererbung von Implementierungen

Liskovsches Substitutionsprinzip:

S1: Subtypen haben gleiche Operationen mit kompatibler Signatur (Compilergeprüft)
S2: Subtypen zeigen das gleiche Verhalten (Programmiergeprüft)
=> Design-By-Contract: Subtypen dürfen Precondition (Anfang der Methode/Parameter) schwächen (AND-Terme weglassen), und/oder Postconditions (Ende der Methode/Rückgaben) stärken (AND Verknüpfung) => Obermenge der gültigen Eingaben, Teilmenge der gültigen Ausgaben.

Ko- Kontravariant: Szenario: Unterklasse überschreibt Methode in Oberklasse mit anderen Typen in Ein- Ausgabewerten.

	Parametertyp in Unterklasse	Parametertyp	Rückgabetyp
kovariant	ist Subtyp des Parameters in Oberklasse	unsicher	sicher
kontravariant	ist Obertyp des Parameters in Oberklasse	sicher	unsicher

Delegation: siehe Design Pattern Strategy

Law of Demeter/Feature Envy: Empfehlung: Klassen sollten nur mit direkt gekoppelten Klassen arbeiten, nicht mit indirekt verknüpften
$A \rightarrow B \rightarrow C$ => A ruft methoden von B auf, nicht von C

high Fan-In, lowFan-Out: Fan-in: Anzahl der Module, die das Modul verwenden (tolerabel) Fan-out: Anzahl der verwendeten Module (möglichst klein)

Open-Close Prinzip: Offen für Erweiterung, geschlossen für Veränderungen

Muster Allgemein

Lösungsschablone zur Lösung eines Problems, in der Praxis bewährt

Architekturmuster

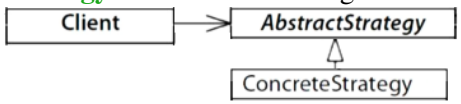
Vorraussetzung: Modularität

Model-View-Controller: Darstellung von variablen Daten in verschiedenen Sichten
Leichte Änderbarkeit der Views => Unabhängig vom Kern implementiert;
Direke Anpassung der Sichten bei Änderungen
Lösung: Modularisierung:
Model: Kernfunktionalität und Datenmodell; registriert und informiert View u. Controller
View: Darstellung der vom Model abgefragten Daten
Controller: Auswertung Benutzereingaben => Aufruf von Model bzw. Viewfunktionen
Bei Implementierung:
Observer Pattern: Observable->Model, Observer->View,Controller
Command Pattern: Aufrufer->View, Befehl->Controller (Interface) Konkreter Befehl->ControllerImpl.

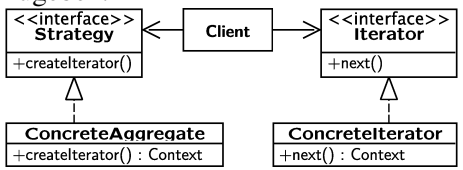
Design Pattern

Vorraussetzung: Objektorientierung

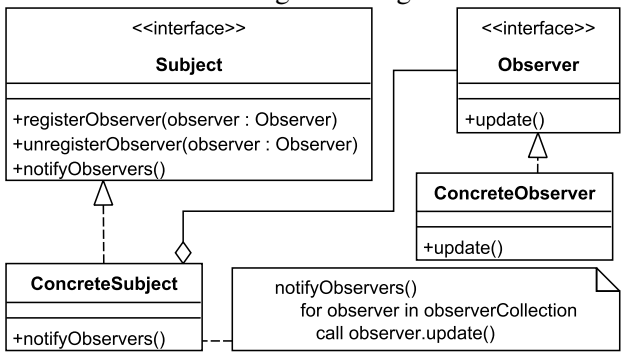
Strategy: Austauschbare Algorithmen



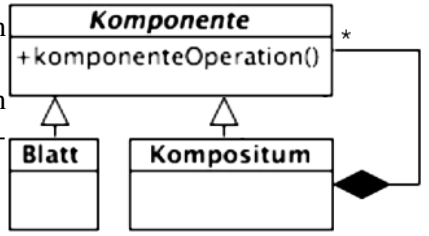
Iterator: Sequentieller Zugriff auf Elemente eines Aggregats ohne interna preiszugeben.



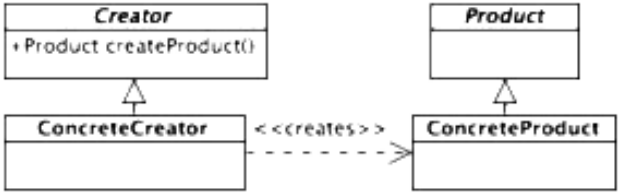
Observer: Informieren von Objekten über Zustandsänderungen eines Objektes Varianten:
Pull: keine information im notify
Push: Zustandsänderung wird mitgeschickt



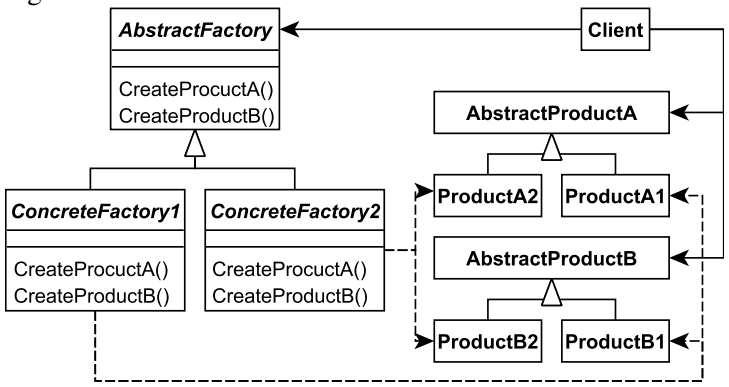
Composite: Hierarchiestruktur,
Verwendung der Teile unabhängig von der Position (Knoten vs Blatt)
Lösung: Schnittstelle mit Operationen für Hierarchieverwaltung und Elementfunktionen
=> Keine Unterschiede für Client



Factory Method:
Problem: Objekt erzeugen. Nur Schnittstelle nicht konkrete Klasse bekannt.
Factory Method: Hook für Subklasse, um Verhalten in der Basisklasse zu konfigurieren. Product und Creator bilden häufig „parallele“ Klassenhierarchien.

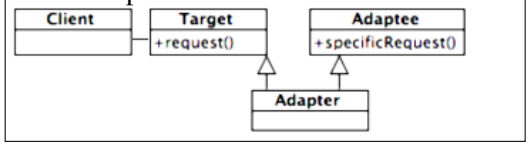


Abstract Factory:
entkopplung Objekterzeugung/-verwendung. Factory Method legt Klasse der Objekte fest. (Delegieren Objekterzeugung)
Factory Instanzen implementieren Factory Methoden für Objekte einer Klassenfamilie
Schnittstelle AbstractFactory sieht eine Factory Methode je Produkt vor. Definiere pro Produktfamilie eine ConcreteFactory Klasse
=> System arbeitet unabhängig von Objekterzeugung. Austausch von Produktfamilien ist einfach. Schwierig, neue Produkte einzuführen, da alle Factoryklassen um eine Methode ergänzt werden müssen.

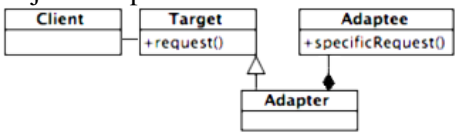


Adapter: Client erwartet andere Schnittstelle als Service bietet (z.B. Legacy)
=> Adapter Zwischenobjekt
Implementierungen:

class Adapter

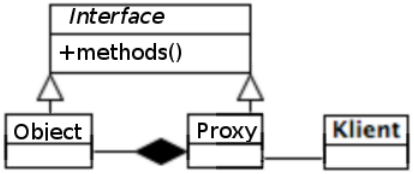


object Adapter

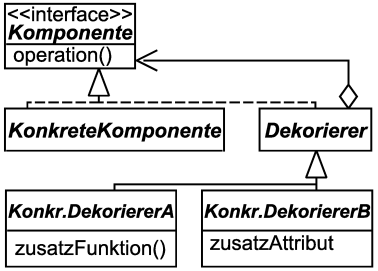


Vergleich: Schnittstelle, welche verschieden vom Service Objekt ist.
Varianten: Two-Way-Adapter: Adapter bietet beide Schnittstellen an
Adapt Interface: Klasse implementiert nicht alle methoden der Schnittstelle => adapter liefert default implementierung.

Proxy: Bei teurem Zugriff auf Service Objekt (z.B nur bei Bedarf erzeugen u. initialisieren)
Entkopplung des Aufrufs durch Zwischenobjekt
Vergleich: Proxy Implement gleiche Schnittstelle wie Service Objekt.

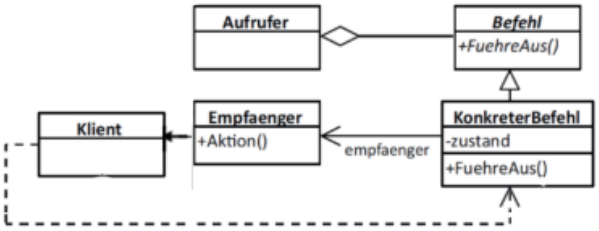


Dekorator: Funktionalität von Instanzen erweitern, ohne Klasse zu verändern
=> Objekt in Dekorator kapseln
Vergleich: Schnittstelle, welche die des Service Objekts erweitert.

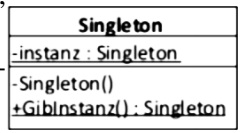


Facade:
Client interagiert mit vielen Schnittstellen eines Subsystems => erhöhte Kompexität und Abhängigkeiten => Zusammenfassen in einem Objekt

Command: kapselung von Operation mit Kontext z.B. für Queueing, entkoppeln auswahl/ausführen (z.B. in anderem Thread), gruppieren von Operationen



Singleton: Nur eine Instanz der Klasse, globaler Zugriffsmechanismus
=> statische Fabrikmethode getInstance; private Konstruktoren;



Programmieridiome

Vorraussetzung: Konkrete Sprache

Mixin: Erweitert bestehende Klasse ohne Vererbung zu verändern
ohne Mehrfachvererbung, mit Mehrfachvererbung