Lambda

Äquivalenzregeln:

Äquivalenzregeln: ungerichtet

$\lambda u.e \equiv \lambda v.[v/u]e \quad (v \notin \mathrm{fv}(e))$		α-Konversion (Umbenennung)
$(\lambda v.f)\, e \equiv [e/v]f$		β-Konversion (Anwendung)
$\lambda v.(e\, v) \equiv e \quad (v \notin \mathrm{fv}(e))$		η-Konversion (Sonderfall)

$e \equiv f$ wenn $e \in E$ durch (wiederholte) Anwendung obiger Regeln in $f \in E$ umwandelbar

Reduktionsregeln

Reduktionsregeln: strikt von links nach rechts

$(\lambda v.f)\, e \to [e/v]f$		β-Reduktion
$\lambda v.(e\, v) \to e \quad (v \notin \mathrm{fv}(e))$		η-Reduktion

$e \in E$ ist in **Normalform** wenn e durch Reduktionsregeln nicht weiter reduzierbar

Berechnung im λ-Kalkül: reduziere Ausdruck e zu Normalform f (es gilt $e \equiv f$)

Modularisierung

Objekt

- existiert zur Laufzeit
- anonym
- Identität
- Zustand und Verhalt unterscheidbar

Klasse

- für Objekterzeugung
- Klassifizierung (Java-Klasse/Interface)
- nicht static

Module

- Übersetzungseinheit
- importiert über Namen
- zyklenfrei Java-Klasse
- static

Komponente

- Übersetzungseinheit
- Import anonym
- Deployment nötig; z. B. Java-EE-Bean

Namensraum

- zur Organisation von Modularisierungseinheiten; z. B. Java-Paket
- Global

Paradigmen

Prozedural

- Programmfluss soll Kontrollfluss entsprechen
- globale Variablen und Aliase erlaubt und nötig, aber unerwünscht
- zur Modularisierung höchstens Module
- Prozeduren/Objekte nicht als Daten
- globale Daten, jeder Wert nur auf eine Weise zugreifbar
- schon kleine Programme wirken komplex
- überschaubare, anfängerfreundliche Menge sprachlicher Ausdrucksmittel
- viel Kontrolle über Details
- spezielle Hardware ansprechbar
- schon kleine Programme wirken komplex
- niedrige Abstraktionsgrade, häufig λ-Abstraktion, manchmal nominale Abstraktion
- Basis für formale Korrektheitsbeweise bei λ-Abstraktion
- Schleifen häufig, Rekursion selten
- entweder nur dynamisch oder weitgehend statisch typisiert (explizite Typspezifikationen)
- unkontrollierbare Kommunikation über Variablen und Aliase ist problematisch
- hardwarenahe Programmierung
- Echtzeitprogrammierung
- Scripting
- flexible Software-Architekturen (z. B. Micro-Services) s
- Hobby-Programmierung und Anwendungsprogrammierung
- Hauptziel: gute Kontrolle
- Wichtigste Daten: Zahlen, Arrays
- Programmieren im Feinen
- Abstraktionsform: λ. nominal
- Abstraktionslevel: niedrig

- Seiteneffekte: wichtig
- Umgang mit Aliasen: problematisch
- Erlernbarkeit: einfach
- Fehleranfälligkeit: hoch
- Typisierung: statisch, dynamisch
- Besonders geeignet für: hardwarenahe Programmierung
- nicht geeignet für: große Projekte

OOP

- Prozeduren zusammen mit Objekten als Daten
- Objekte, Variablen haben nominale abstrakte Datentypen, abstrakt verständlich
- dynamisches Binden erzwingt abstraktes Verständnis (Kontrollfluss zu unklar)
- örtlich eingegrenzte Kommunikation über Variablen (private)
- offensiver Umgang mit Aliasen (Identität versus Gleichheit)
- verschiedene Daten zu Objekt zusammengefasst
- professionelle Werkzeugkette für Entwicklung und Wartung großer, langlebiger Software
- nominale Abstraktionen auf hohem Niveau
- Programmierung im Groben (Faktorisierung, Abstraktionshierchien, . . .) im Mittelpunkt
- Zusammenarbeit professioneller Entwickler_innen notwendig
- komplexes Gefüge an Denkmustern
- sehr teuer, aber auch für sehr komplexe Systeme erfolgversprechend
- erfordert vollen Einsatz und viel Wissen („ein bisschen objektorientiert“ ist sinnlos)
- ungeeignet für kleine Projekte und sehr komplexe Algorithmen
- überfordert unerfahrene Programmierer_innen
- Hauptziel: langfristige Wartung
- Wichtigste Daten: Objekte
- Programmieren Groben
- Abstraktionsform: nominal
- Abstraktionslevel: hoch
- Seiteneffekte: wichtig
- Umgang mit Aliasen: Identität
- Erlernbarkeit: schwer
- Fehleranfälligkeit: mittel
- Typisierung: stark
- Besonders geeignet für: große Projekte
- nicht geeignet für: kleine Programme, komplexe Algorithmen

Funktional

- Funktionen als Daten
- ersetzen Kontrollstrukturen
- Seiteneffekte verboten
- Aliase stören dadurch nicht (referentielle Transparenz)
- Modularisierung wichtig
- λ-Abstraction
- nominale und strukturelle Abstraktion
- große Strukturen
- änderbare Daten referenzieren stabile
- Programmierung im Groben gut unterstützt
- ohne Seiteneffekte keine Kommunikation über gemeinsame Variablen
- Aliase harmlos
- Original und Kopie nicht unterscheidbar (referenzielle Transparenz)
- „sauber“: aufgesammeltes Wissen geht nie verloren
- Funktion höherer Ordnung = funktionale Form, kann jede Kontrollstruktur ersetzen
- bei hohen Abstraktionsgraden eher λ-Abstraktion oder strukturelle Abstraktion
- ausschließlich Rekursion statt Schleifen
- heute meist vollständig statisch typisiert (Typinferenz)
- Lazy-Evaluation einfach
- Hauptziel: Programmiereffizienz
- Wichtigste Daten: Funktionen
- Programmieren Feinene
- Abstraktionsform: strukturell, λ
- Abstraktionslevel: niedrig bis hoch
- Seiteneffekte: verboten
- Umgang mit Aliasen: referentielle Transparenz
- Erlernbarkeit: mittelmäßig
- Fehleranfälligkeit: niedrig
- Typisierung: statisch (Typinferenz)
- Besonders geeignet für: komplexe Algorithmen
- nicht geeignet für: hardwarenahe Programme

Parallel

- kurze Gesamtlaufzeiten auf vielen Prozessoren angestrebt
- Daten meist in Bereiche aufgeteilt, die unabhängig bearbeitbar sind
- Zielerreichung durch Speedup ausgedrückt: Sp = T1/Tp (größer ist besser)
- Speedup abhängig von Details der Aufgabenstellung, Daten und Hardware
- Wissen über diese Details nötig

- Summe der Rechenzeit (p Recheneinheiten) höher als sequentielle Zeit T1 (daher Sp < p)
- Sp > 1 nur wenn Parallelausführung Zusatzaufwand überkompensiert
- hoher Aufwand auf vielen Ebenen (Hardware, Softwareentwicklung, Wartung, . . .)
- zahlt sich nur zur Verarbeitung großer Mengen einheitlich strukturierter Daten aus
- wichtig ist Finden einer Vorgehensweise, die unabhängige Datenblöcke ermöglicht
- fertige Bibliotheken für häufige Einsatzzwecke

Nebenläufig

- effiziente Reaktionsfähigkeit auf Ereignisse angestrebt
- unterschiedliche Handlungsstränge sollen auf unterschiedliche Daten zugreifen
- viele gleichzeitig/überlappt ablaufende Handlungsstränge
- um Programm zu vereinfachen
- Ausführung der Handlungsstränge häufig durch Warten auf Ereignisse unterbrochen
- vielfältige Ziele und Anwendungsgebiete, z. B. Webserver, Telefonanlage, Simulation
- Bewältigung vieler Handlungsstränge (hohe Last) meist wichtiger als kurze Antwortzeiten
- Anzahl der Handlungsstränge an Bedarf
- nicht an Hardwarefähigkeiten ausgerichtet
- in Java Handlungsstränge meist als Threads implementiert
- Zugriffe auf gemeinsame Daten kaum vermeidbar, Synchronisation wichtig
- Vermeidung von Liveness-Problemen notwendig, aber schwierig
- zahlreiche Synchronisationsmechanismen, in Java hauptsächlich Monitore

Applikativ

- Variante der funktionalen Programmierung,
- funktionale Formen mit Hilfsfunktionen parametrisiert und zusammengefügt
- verwendet vorgefertigte Teile mit wenig zusätzlichem Code
- sehr produktiv
- für komplexe Algorithmen geeignet
- kurze, kompakte Programme mit wenigen Funktionen
- Rekursion kaum sichtbar, nur im Hintergrund
- beruht auf überschaubarer Menge zusammenpassender vorgefertigter funktionaler Formen
- kann div. Programmiertechniken einfach unterstützen (Lazy-Evaluation, Parallelität, . . .)
- Programmstruktur kann sehr kreativ sein
- hoher Abstraktionsgrad, strukturelle Abstraktion oder λ-Abstraktion
- meist generisch und statisch typisiert mit Typinferenz

- eher schwer lesbar

Liveness- Probleme

Starvation

- wichtige Programmteile bekommen nicht genug Ressourcen (etwa Rechenzeit)
- Auswirkung: langsamer Programmfortschritt, kann zum Stillstand führen
- Ursache: unwichtige Programmteile binden Ressourcen (schlechte Ressourcenverteilung)
- Erkennen des Problems: ausgiebig Testen
- Problembeseitigung: gezielte Steuerung, z. B. Zwischenschalten passender Puffer

Deadlock

- mehrere Threads warten gegenseitig auf exklusive Objektzugriffe, die sie halten
- Auswirkung: gegenseitige dauerhafte Blockade, kein Programmfortschritt
- Ursache: mehrere Threads halten und brauchen exklusive Zugriffe auf gleiche Objekte
- Erkennen des Problems: ausgiebig Testen, Programmanalyse (statisch oder dynamisch)
- Problembeseitigung: Timeout, exklusiver Zugriff nur in vorbestimmter Reihenfolge

Livelock

- ähnelt Deadlock, statt Warten wird nachgefragt, ob exklusiver Zugriff möglich
- Auswirkung und Ursache wie bei Deadlock
  Erkennen des Problems: ausgiebig Testen, Programmanalyse kaum zielführend
- Problembeseitigung: Timeout, nicht aktiv warten, nicht „schrittweise Anfragen“

Abstraktionshierarchien

Arten von Abstraktionshierarchien:

- Untertypbeziehung
  - B ist Untertyp von A wenn jede Instanz von B verwendet werden
- Vererbungsbeziehung
  - Übernehmen von Programmtexten aus Oberklasse in Unterklasse
- Untertypbeziehung höherer Ordnung
  - wenn B$\langle U \rangle$ Untertyp von A$\langle U \rangle$ für alle U

Varianz von Typen

- Kovarianz

    - A verhält sich zu B wie T zu U
    - Typ von Konstante, Ergebnis, Ausgangsparameter
- Kontravarianz
    - A verhält sich zu B umgekehrt wie T zu U
    - Typ von Eingangsparameter
- Invarianz
    - gleichzeitig Ko- und Kontravarianz
    - Typ von Variable, Durchgangsparameter

Zusicherungen

- Client-Server-Beziehungen
    - Server bietet Dienste an, Client nutzt Dienste
- Vorbedingung (Precondition)
    - Wer: Client
    - Wann: vor Methodenaufruf.
    - Was: hauptsächlich Eigenschaften von Argumenten
- Nachbedingung (Postcondition)
    - Wer: Server
    - Wann: return von Methode
    - Was: Eigenschaften von Methodenergebnissen sowie Änderungen und Eigenschaften des Objektzustands
- Invariante
    - Wer: Server
    - Wann: vor und nach Ausführung von Methoden
    - Was: unveränderliche Eigenschaften von Objekten und Variablen
- Server-kontrollierter History-Constraint
    - Wer: Server
    - Wann: nach Ausführung von Methoden
    - Was: Einschränkungen auf Veränderungen von Variable
- Client-kontrollierter History-Constraint
    - Wer: Client
    - Wann: vor Methodenaufrufen
    - Was: Einschränkungen auf der Reihenfolge von Aufrufen

Zusicherung mit Untertypen

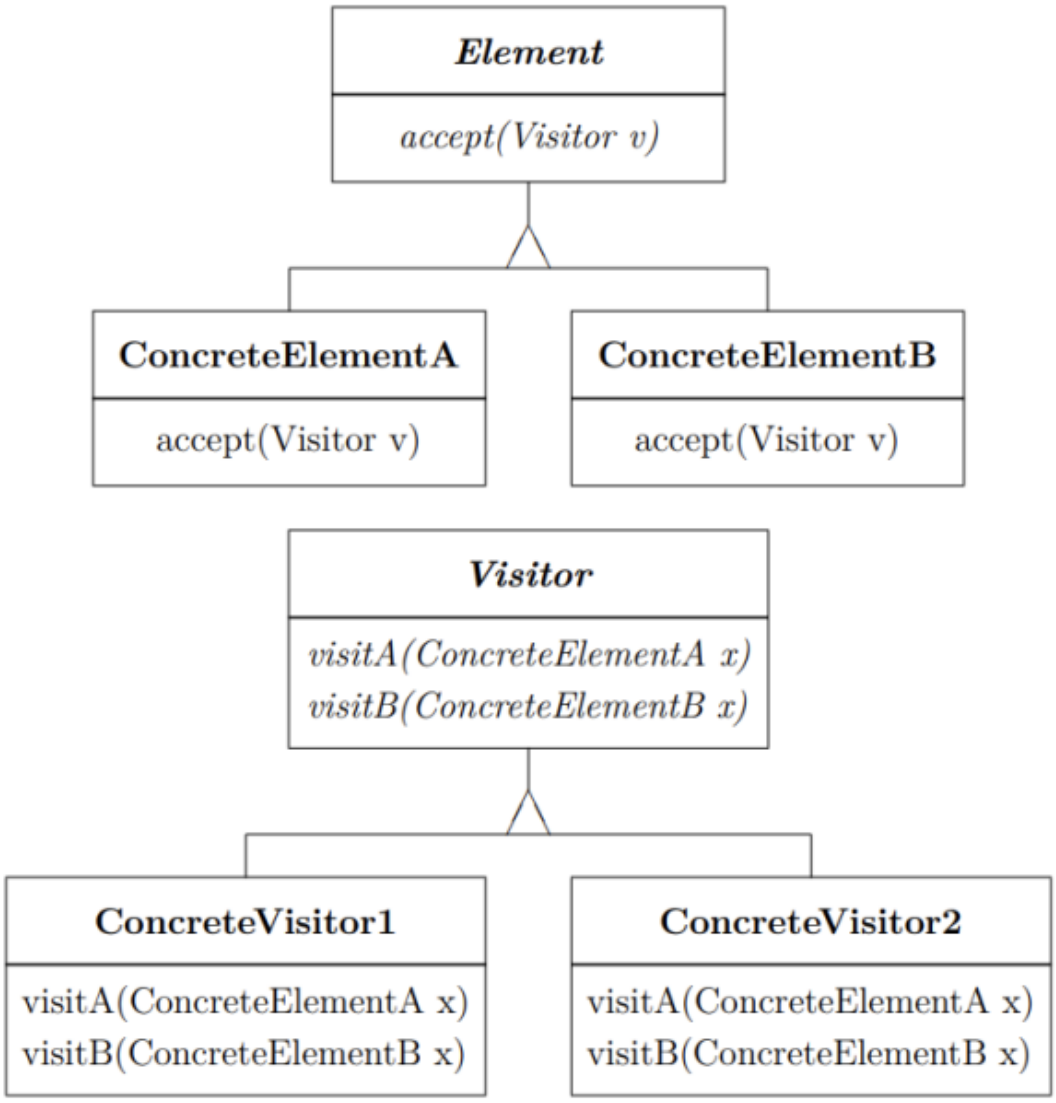
- Vorbedingungen in Untertypen sind schwächer oder gleich
- Nachbedingungen in Untertypen sind stärker oder gleich
- Invarianten in Untertypen sind stärker oder gleich

Entwurfsmuster

Visitor Pattern

Anwendung:

- viele unterschiedliche, nicht verwandte Operationen auf einer Objektstrutkur realisiert werden soll
- sich die Klassen der Objektstruktur nicht ändern
- häufig neue Operationen auf der Objektstruktur integriert werden müssen oder
- ein Algorithmus über die Klassen einer Objektstrutkur verteilt arbeitet, aber zentral verwaltet werden soll.



Eigenschaften:

- Neue Operationen lassen sich leicht durch die Definition neuer Untertypen von Visitor hinzufügen.
- Verwandte Operationen werden im Visitor zentral verwaltet und von Visitor-fremden Operationen getrennt.
- Ein Visitor kann mit Objekten aus voneinander unabhängigen Klassenhierarchien arbeiten.

- Die gute Erweiterungsmöglichkeit der Klassen unterhalb von Visitor muss mit einer schlechten Erweiterbarkeit der Klassen der konkreten Elemente erkauft werden. Müssen neue konkrete Elemente hinzugefügt werden, so führt dies dazu, dass viele Methoden implementiert werden müssen.
- Häufig wird angeführt, dass die visit-Methoden nicht einfach auf konkrete Elemente zugreifen können; oft können dies Parameter der visit-Methoden ausgleichen, wodurch es auf Implementierungsdetails ankommt, inwieweit das relevant ist.

Iterator

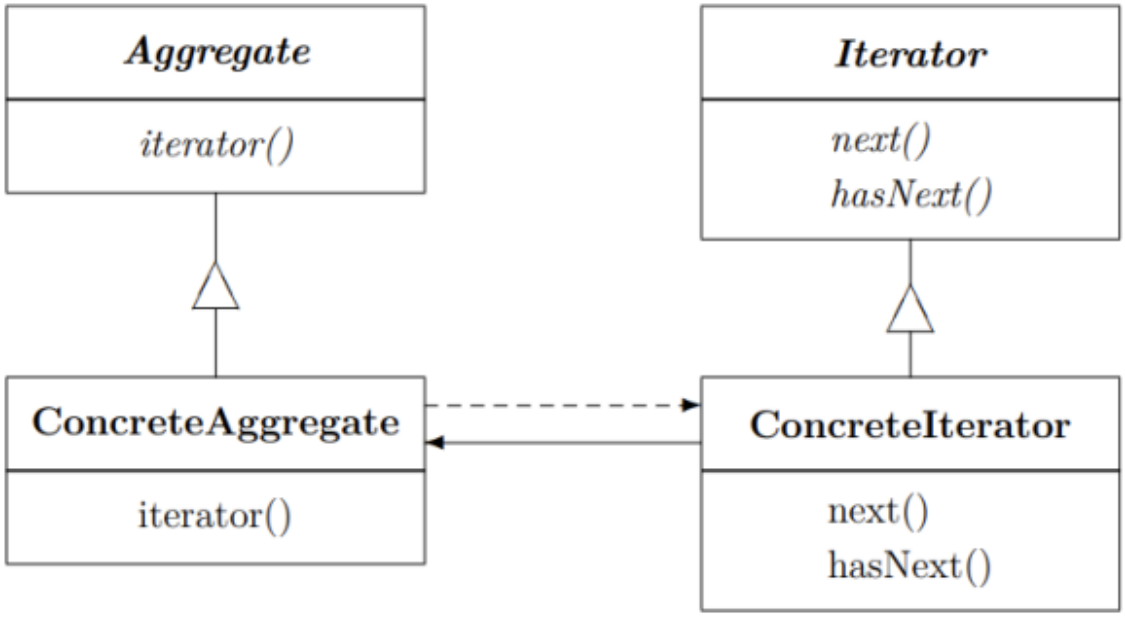
Iterator, auch Cursor, ermöglicht sequentiellen Zugriff auf die Elemente eines Aggregats (Sammlung von Elementen), ohne die innere Darstellung des Aggregats offenzulegen

Anwendungsfälle:

- auf die Inhalte eines Aggregats zugreifen zu können, ohne die innere Darstellung offen legen zu müssen,
- mehrere (gleichzeitig bzw. überlappende) Abarbeitungen der Elemente in einem Aggregat zu ermöglichen,
- eine einheitliche Schnittstelle für die Abarbeitung verschiedener Aggregatstrukturen zu haben, also um polymorphe Iterationen zu unterstützen.

Eigenschaften:

- Unterschiedliche Varianten in der Abarbeitung, z.B. für Baumstrukturen gibt es zahlreiche Möglichkeiten. Mehrere Iteratoren für unterschiedliche Reihenfolgen implementierbar.
- Vereinfachen die Schnittstelle von Aggregate, da Zugriffsmöglichkeiten durch Iteratoren bereitgestellt werden. Weitere Methoden möglich, z.B. in Java in Iterator die Methode remove
- Mehrere Iterator können gleichzeitig auf einem Aggregat arbeiten, da jeder Iterator den aktuellen Abarbeitungszustand verwaltet.

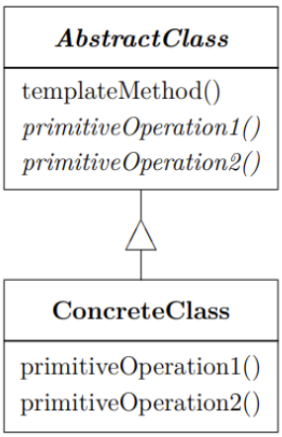


Template Pattern

Definiert das Grundgerüst eines Algorithmus in einer Operation, überlässt die Implementierung einiger Schritte aber einer Unterklasse.

Anwendbar:

- um den unveränderlichen Teil eines Algorithmus nur einmal zu implementieren und es Unterklassen zu überlassen, den veränderbaren Teil des Verhaltens festzulegen,
- wenn gemeinsames Verhalten mehrerer Unterklassen in einer einzigen Klasse lokal zusammengefasst werden soll,
- um mögliche Erweiterungen in Unterklassen zu kontrollieren, beispielsweise durch TemplateMethods, die Hooks als primitive Operationen (siehe Strukturzeichnung) aufrufen und nur das Überschreiben dieser Hooks in Unterklassen ermöglichen.



Eigenschaften

- Fundamentale Technik zur direkten Wiederverwendung von Programmcode.
- Umkehrung der üblichen Kontrollstruktur, die manschmal als Hollywood-Prinzip bezeichnet wird („Don’t call us, we’ll call you.“). Das bedeutet, die Oberklasse ruft

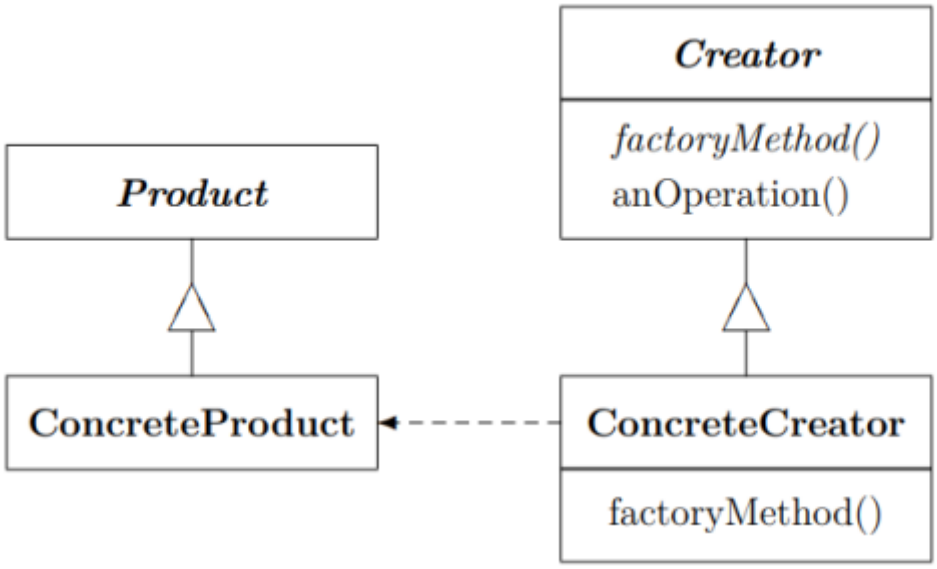
Methoden der Unterklasse auf — nicht wie in den meisten Fällen umgekehrt.

- „templateMethod“ ruft folgende Arten von primitiven Operationen auf
  - konkrete Operationen in AbstractClass, also Operationen, die ganz allgemein auch für Unterklassen sinnvoll sind;
  - abstrakte primitive Operationen, die einzelne Schritte im Algorithmus ausführen und in „ConcreteClass“ implementiert werden;
  - Hooks
  - Factory-Methods, also Methoden, die in Unterklassen neue Objekte erzeugen und zurückgeben und damit die Template-Method auch zu einem anderen Entwurfsmuster werden lassen (siehe Factory-Method).

Factory

Anwendbar

- eine Klasse Objekte erzeugen soll, deren Klasse sie aber nicht kennt
- eine Klasse möchte, dass ihre Unterklassen die Art der Objekte bestimmen, welche die Klasse erzeugt,
- Klassen Verantwortlichkeiten an eine von mehreren Unterklassen delegieren und das Wissen, an welche Unterklasse delegiert wird, lokal gehalten werden soll,
- die Allokation und Freigabe von Objekten zentral in einer Klasse verwaltet werden soll



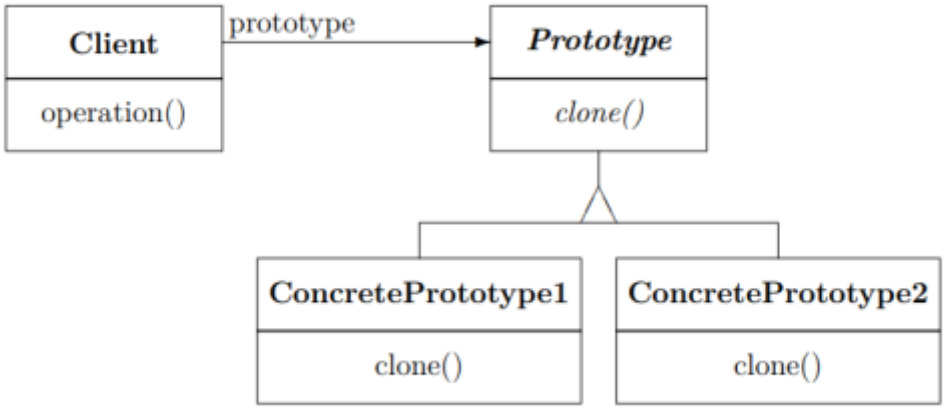
Eigenschaften:

- Bieten Anknüpfungspunkte für Unterklassen, indem sie genau vorgeben, welche Methoden für das Überschreiben vorgesehen sind. Das führt zu einer Umkehrung der Abhängigkeiten: Oberklassen hängen von Unterklassen ab. Die Erzeugung eines neuen Objekts mittels FactoryMethod ist fast immer flexibler als die direkte Objekterzeugung. Vor allem wird die Entwicklung von Unterklassen vereinfacht.
- Verknüpfen parallele Typhierarchien, die Creator-Hierarchie mit der Product-Hierarchie.

Prototype

Dient dazu, die Art eines neu zu erzeugenden Objekts durch ein Prototyp-Objekt zu spezifizieren. Neue Objekte werden durch Kopieren dieses Prototyps erzeugt.
Anwendbar:

- Generell anwendbar, wenn ein System unabhängig davon sein soll, wie seine Produkte erzeugt, zusammengesetzt und dargestellt werden
- die Klassen, von denen Objekte erzeugt werden sollen, erst zur Laufzeit bekannt sind
- wenn Factory-Method Hierarchien vermieden werden sollen
- jedes Objekt einer Klasse nur weniger unterschiedliche Zustände haben kann (einfacher für jeden Zustand einen Prototype erstellen)



Eigenschaften:

- Konkrete Produktklassen werden vor den Anwendern versteckt und die Anzahl der Klassen, die Anwender kennen müssen wird dadurch reduziert. Die Anwender müssen nicht geändert werden, wenn neue Produktklassen dazukommen oder geändert werden.
- Prototypen können auch zur Laufzeit jederzeit dazugegeben und weggenommen werden (Klassenstruktur zur Laufzeit nicht änderbar).
- Sie erlauben die Spezifikation neuer Objekte durch änderbare Werte.
- Sie vermeiden eine übertrieben große Anzahl an Unterklassen. Im Gegensatz zur Factory-Method ist es nicht nötig, parallele Klassenhierarchien zu erzeugen.
- Sie erlauben die dynamische Konfiguration von Programmen.

`clone()` in Java

Singleton

Sichert zu, dass eine Klasse nur eine Instanz hat und erlaubt globalen Zugriff auf dieses Objekt.

Anwendbar:

- es genau ein Objekt einer Klasse geben soll und dieses global zugreifbar sin soll
- die Klasse durch Vererbung erweiterbar sein soll und Anwender ide erweiterte Klasse ohne Änderungen verwenden können sollen

```
public class Singleton {
    private static Singleton singleton = null;
    private Singleton() {} // no object creation from outside

    public static Singleton instance() {
        if (singleton == null){
            singleton = new Singleton();
        }
        return singleton;
    }
}
```

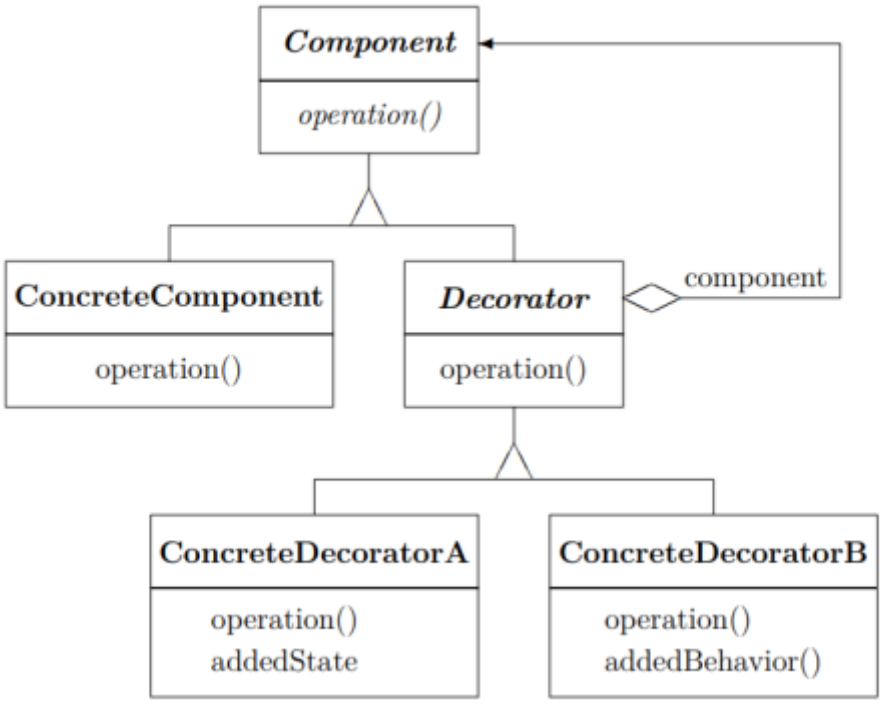
Eigenschaften:

- Kontrollierter Zugriff auf das einzige Objekt.
- Verzichtet auf globale variablen
- Vererbung wird unterstützt
- Unkontrollierte Erzeugung von Instanzen wird verhindert (non public Konstructor)
- Prinzipiell auch mehrere Instanzen erzeugbar. Klasse hat vollständige Kontrolle darüber, wie viele Objekte erzeugt werden
- Auf das `instance` Objekt kann über die Objektmethode durch dynamisches Binden flexibler zugegriffen werden

Decorator

Anwendbar:

- um dynamisch Verantwortlichkeiten zu einzelnen Objekten hinzuzufügen, ohne andere Objekte zu beeinflussen;
- für Verantwortlichkeiten, die wieder entzogen werden können;
- wenn Erweiterungen einer Klasse durch Vererbung unpraktisch sind, z.B. um eine sehr große Zahl an Unterklassen zu vermeiden



Eigenschafen:

- Mehr Flexibilität als statische Vererbung
- Vermeiden Klassen, die bereits weit oben in der Typhierarchie mit Methoden und Variablen überladen sind.
- Objekte von Decorator und die dazugehörenden Objekte von „ConcreteComponent“ sind nicht identisch.
- Führen zu vielen kleinen Objekten

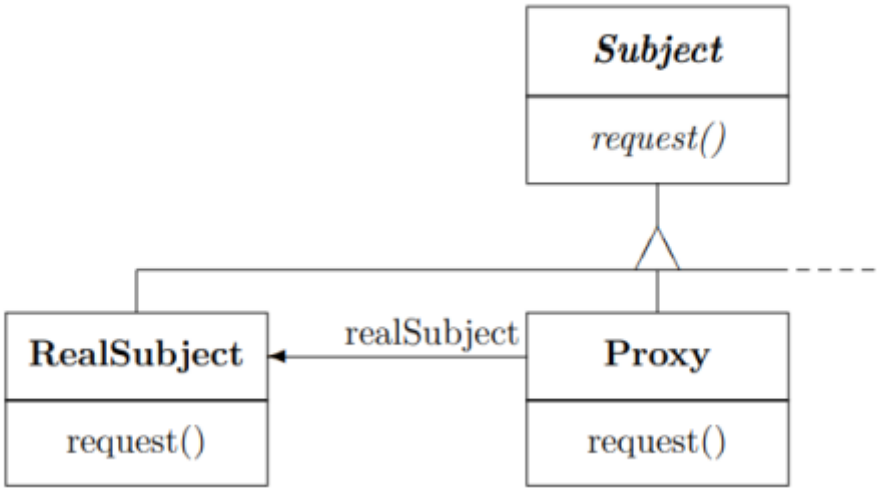
Proxy

Auch Surrogate, stellt einen Platzhalter für ein anderes Objekt dar und kontrolliert Zugriffe darauf.

Anwendbar:

- wenn eine intelligenter Referenz auf ein Objekt als ein simpler Zeiger nötig ist
- Remote-Proxies: Platzhalter für Objekte, die in anderen Namensräumen existieren. Nachrichten an die Objekte werden von den Proxies über komplexere Kommunikationskanäle weitergeleitet.
- Virtual-Proxies: erzeugen Objekte bei Bedarf (Performance Gründe).
- Protection-Proxies: kontrollieren Zugriffe auf Objekte.
- Smart-References: können bei Zugriffen zusätzliche Aktionen ausführen, z.B. Mitzählen der Referenzen, das Laden von Objekten in den Speicher bei erstmaligem Aufrufen, . .

.



AspectJ

- Ziel
  - Ergebnisse von Berechnungen auf eine gewisse Weise abzuändern, ohne den Programmtext oder die Daten zu ändern. Sie bewegt sich in einem Graubereich, was die genaue Zuordnung zu Programm, Daten oder Sprachsemantik betrifft
- Join-Point
  - zur Laufzeit identifizierbare Stelle in einem Programm, z.B. der Aufruf einer Methode oder der Zugriff auf ein Objekt.
- Pointcut
  - syntaktisches Element in einer .aj-Datei, das einen Join-Point (bzw. mehrere gleichartige Join-Points) auswählt und kontextabhängige Information dazu sammelt, z.B. die Argumente eines Methodenaufrufs oder eine Referenz auf das Zielobjekt.
- Advice
  - syntaktisches Element in einer .aj-Datei, das den Programmtext definiert, der an einem Join-Point ausgeführt werden soll. Dabei kann man den Programmtext vor (before()), nach (after()) oder anstatt (around()) dem Join-Point ausführen (wobei bei around() der Join-Point häufig innerhalb des Programmtexts explizit ausgeführt wird).
- Aspect
  - zentrales syntaktisches Element in einer .aj-Datei, das alle Teile zu einer Einheit zusammenführt. Ein Aspekt enthält Deklarationen von Variablen und Definitionen von Methoden (wie eine Java-Klasse) sowie Pointcuts und Advices.

Shell Commands

- `|` - Redirect stdout to stdin (e.g., `ls | grep file.txt`)
- `||` - Logical OR

- `>` - Output redirect (replace if it exists)
- `>>` - Redirect stdout to a file (append if exists)
- `&>` - Redirect both stdout and stderr to a file
- `&>>` - Append both stdout and stderr to a file
- `&&` - Logical AND
- `<` - Input redirection (reads input from a file)
- `&` - Run in the background
- `&|` - it is shorthand for `2>&1`
- `2>&1` - Redirect stderr to stdout