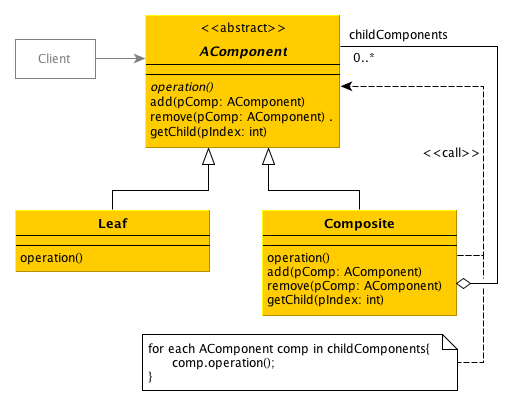
**Entwurfsmuster  
Beschreibung, Variationen, Vor- und Nachteile**

Composite



Beschreibung

Das Composite Entwurfsmuster ermöglicht dem Client eine elegante Handhabung einer verschachtelten (Baum-)Struktur, da diese vereinheitlicht wird.

Component bildet **eine gemeinsame Schnittstelle** mit definierten Methoden für Composite (Elementbehälter) und Leaf (atomares Element), die diese implementieren sollen. Composites delegieren ihre Methodenaufrufe an ihre Components, die aus Leafs oder wiederum Composites zusammengesetzt sind.

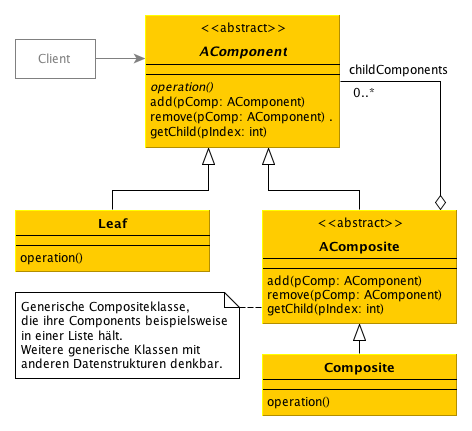
Implementierungshinweise

Transparenz oder Sicherheit sind die zwei grundsätzlichen Strategien bei der Schnittstellendefinition.

Bei der **Transparenz** wird die Componentschittstelle **breit** definiert und enthält die **Summe** aus den Leaf- und Compositemethoden als Defaultimplementierungen, die nichts Zweckmäßiges tun.  
Vorteilhaft für den Client ist, dass ihm **der volle Funktionsumfang** aller möglichen Components ohne Fallunterscheidung für Leafs und Composites zur Verfügung steht und kann die Components somit wirklich **einheitlich behandeln**.  
Der Nachteil hierbei ist, dass der Client **sinnlose Methoden** auf Leafs aufruft und damit rechnen muss, dass ein Aufruf nichts bewirkt oder gar eine Exception bzw. einen Error wirft.

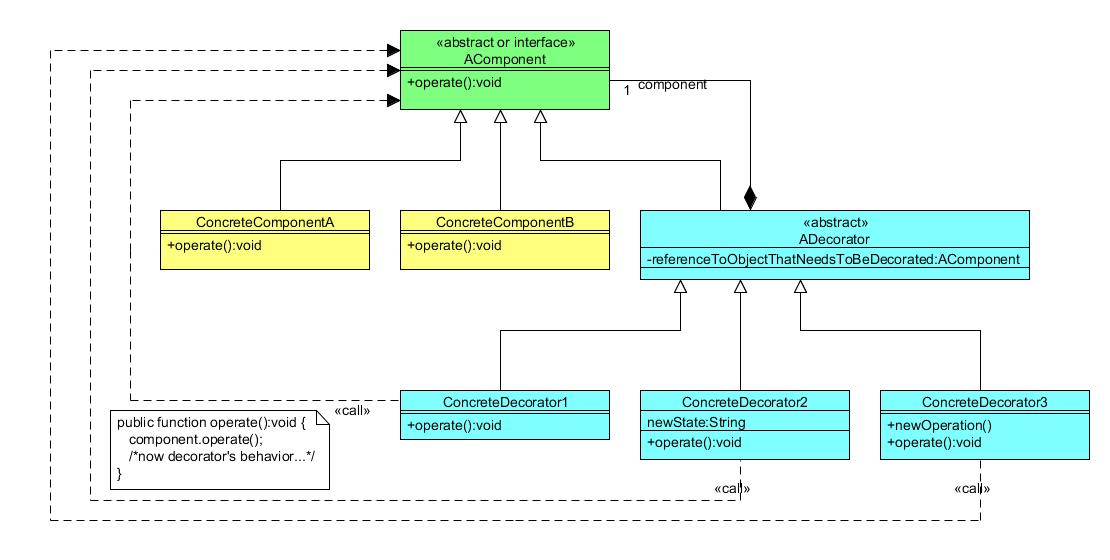
Bei der **Sicherheit** wird die Componentschnittstelle **schmal** definiert und beschreibt den **kleinsten gemeinsamen Nenner** aus den Leaf- und Compositemethoden als Defaultimplementierungen.  
Vorteilhaft für den Client ist, dass er sich beim Methodenaufruf auf Components auf eine **sinnvolle Ausführung** verlassen kann.  
Der Nachteil hierbei ist, dass der Client **Fallunterscheidungen** für einen sicheren **Downcast** machen muss, um an compositespezifische Methoden zu gelangen.

Ergänzend zu den Strategien bei der Schnittstellendefinition, sollte man noch folgende Punkte beachten:

* Jede Component kann zusätzlich eine Referenz auf seine superComponent halten.  
  Vorteil:  
  Man kann die Struktur vereinfacht und beliebig traversieren und verwalten. Das Löschen von Components gestaltet sich beispielsweise einfacher; die subComponent kann seine Löschung in der superComponent veranlassen, da in der Hierarchie nach oben gegangen werden kann.  
  Nachteil:  
  Es besteht die Gefahr von Inkonsistenz, da immer sichergestellt werden muss, dass die Referenzen miteinander übereinstimmen.
* Composites können ihre subComponents je nach Systemanforderung in anderen Datenstrukturen (z.B. Array, Dictionary, Vector, …) halten als andere Composites.
* Code zur Verwaltung von subComponents kann in einer abstrakten Compositeklasse ausgelagert werden, von der jede Compositeklasse erbt, um sich die erneute Implementierung für jeden neuen Compositetypen zu sparen.  
  
* Da das Durchlaufen der (Baum-)Struktur oder die rekursive Berechnung dieser in manchen Fällen sehr performancelastig sein kann, ist das Zwischenspeichern von Berechnungen (Caching) sinnvoll. Bei einer späteren Anfrage wird das zwischengespeicherte Ergebnis ausgeliefert. Falls sich etwas in den enthaltenen Komponenten ändern sollte, kann die subComponent seinen superComponents rekursiv mitteilen, dass sich diese geändert hat.

Allgemeine Vor- und Nachteile des Composite Entwurkfsmusters:

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Repräsentation von verschachtelten Strukturen mit beliebiger Verschachtelungsbreite und –tiefe * Vereinfachter Clientcode durch Auslagerung konkreter Implementierungen * Elegantes Arbeiten mit der (Baum)Struktur durch Rekursion * Flexibilität und Erweiterbarkeit, da neue Elemente durch Implementierung der Componentschnittstelle in das System integriert werden | * Schwierige Definition der Componentschnittstelle * Spätere Einschränkungen der erlaubten Compositeelemente sind schwer realisierbar und laufen auf aufwändige Typprüfungen zur Laufzeit hinaus, da dieses Entwurfsmuster Allgemeingültigkeit anstrebt |

Decorator (Wrapper)

Beschreibung

Das Decorator Entwurfsmuster ist eine flexible Alternative zur statischen Vererbung, die eine dynamische Erweiterung von Klassen um weitere Funktionalitäten auch zur Laufzeit ermöglicht.

Da AComponent **eine gemeinsame Schnittstelle** mit definierten Methoden für Decorator und ConcreteComponent bildet, können das zu erweiternde ConcreteComponent-Objekt mit beliebig vielen Decorator-Objekten (ähnlich wie Wrapper) dekoriert bzw. umhüllt werden. Der Decorator **delegiert Methodenaufrufe** an seine Component, die entweder ein weiterer Decorator oder eine ConcreteComponen ist, weiter und führt sein eigenes spezifisches Verhalten davor oder danach aus.

Implementierungshinweise

1. Abstrakte Klasse ADecorator nicht nötig, aber sinnvoll.
2. AComponent so schmal wie möglich halten.
3. Decorator Entwurfsmuster ist gut geeignet zur Erweiterung der Oberfläche, jedoch schlecht geeignet für inhaltliche Erweiterung sowie umfangreiche Objekte.

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Mehr Flexibilität statt statischer Vererbung durch dynamisches Hinzufügen und Entfernen von zusätzlichen Funktionalitäten * Vermeidung von langen Vererbungshierarchien sowie von einer Überladung von superClasses mit Funktionen. Jeder Decorator repräsentiert genau eine Funktion. * Der Client kann sowohl mit einer dekorieten als auch einer undekorierten ConcreteComponent arbeiten, ohne dass sein Code bricht. * Performancegewinn, da nur benötigte Funktionalitäten initialisiert werden müssen. | * Erschwerte Fehlerfindung durch lange Aufrufketten von dekorierten Objekten * Jedes neu zu dekorierende Objekt bedarf eines neuen Decoratorobjekts. Zwar sind Decoratorobjekte leicht konfigurierbar, aber die hohe Objektanzahl schwer wartbar. *(Zur Kapselung des Erstellungscodes empfiehlt sich eine Factory)* * Keine Objektidentität: ConcreteComponentA != dekorierte ConcreteComponentA ***(Katja fragen, ob richtig verstanden!)*** |

Observer

Beschreibung

Das Observer Pattern ermöglicht Observerobjekte sich bei einem Subjectobjekt zu registrieren und fortan vom Subjectobjekt über Änderungen informiert zu werden.

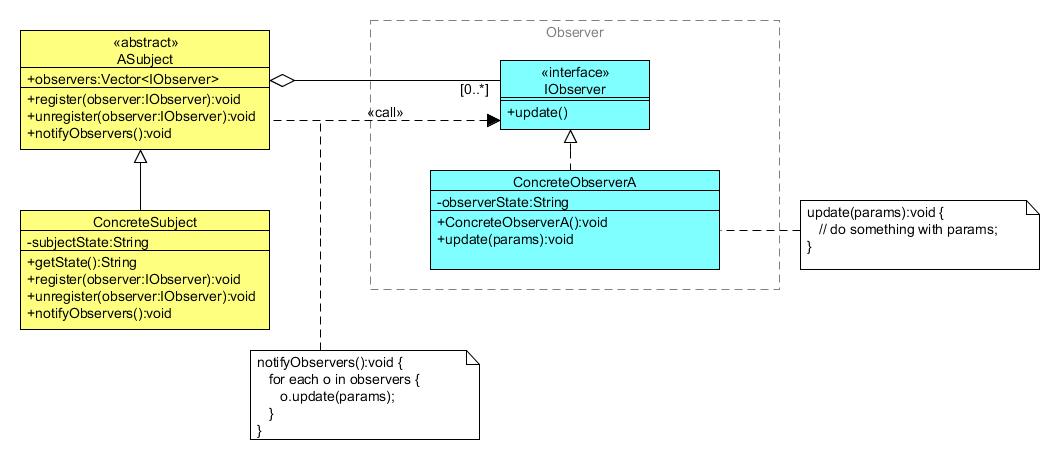
ConcreteObserver implementieren Aktualisierungsmethoden (in den meisten Fällen mit Daten zur Änderung parametrisiert), die ihre einheitliche Schnittstelle definiert hat. So bestimmt jeder ConcreteObserver selbst wie auf Aktualisierungen vom ConcreteSubject reagiert werden soll.

Das ConcreteSubject erbt bzw. implementiert Administrationsmethoden damit sich der ConcreteObserver zur Datenstruktur des ConcreteSubjects hinzufügen bzw. daraus wieder löschen kann. Bei Änderungen am Zustand des ConcreteSubjects werden alle registrierten ConcreteObserver durch iteratives Aufrufen der Aktualisierungsmethoden informiert.

Implementierungshinweise

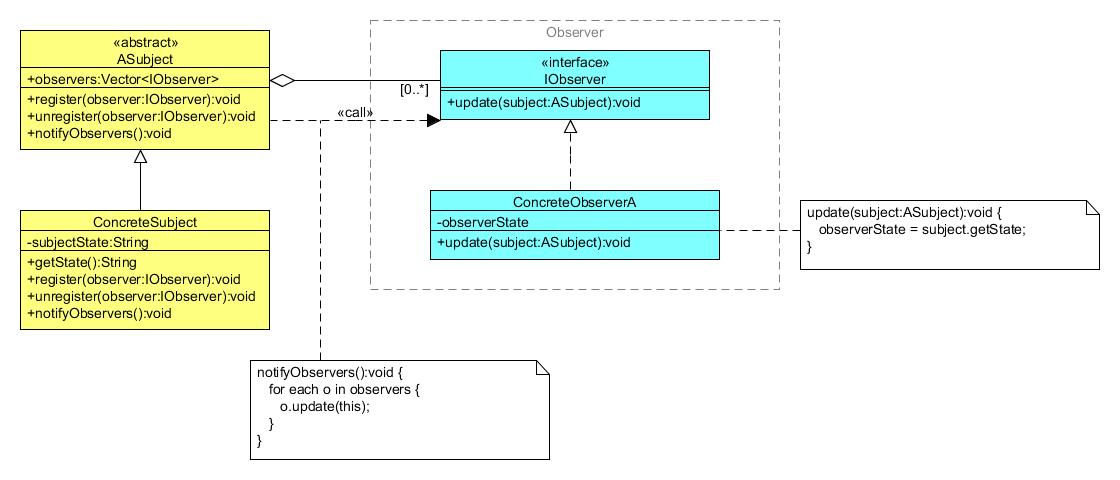
Für die Art und Weise, wie ConcreteObserver die benötigten Informationen erhält, gibt es zwei Varianten.

Beim **Push**-Verfahren übergibt das ConcreteSubject der Aktualisierungsmethode detaillierte Informationen über seine Änderung als Parameter:



|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Stärkere Entkopplung von Observer und Subject, da Observer keine Subjectreferenz benötigt * Nur ein Methodenaufruf zur Übergabe von Informationen | * Informationsüberladung, da nicht jeder Observer zwangsläufig alle Parameter, die übergeben werden, benötigt * Unklarheit darüber, welches ConcreteSubject sich geändert hat, wenn mehrere ConcreteSubjects beobachtet werden * Erweiterbarkeit erschwert, da evtl. alle bereits bestehenden ConcreteObserver angepasst werden müssen * Abhilfe: Eventobjekte kapseln die Änderungsinformation |

Beim **Pull**-Verfahren bekommt der ConcreteObserver eine Referenz zum ConcreteSubject übergeben und holt sich seine benötigten Informationen selbst:

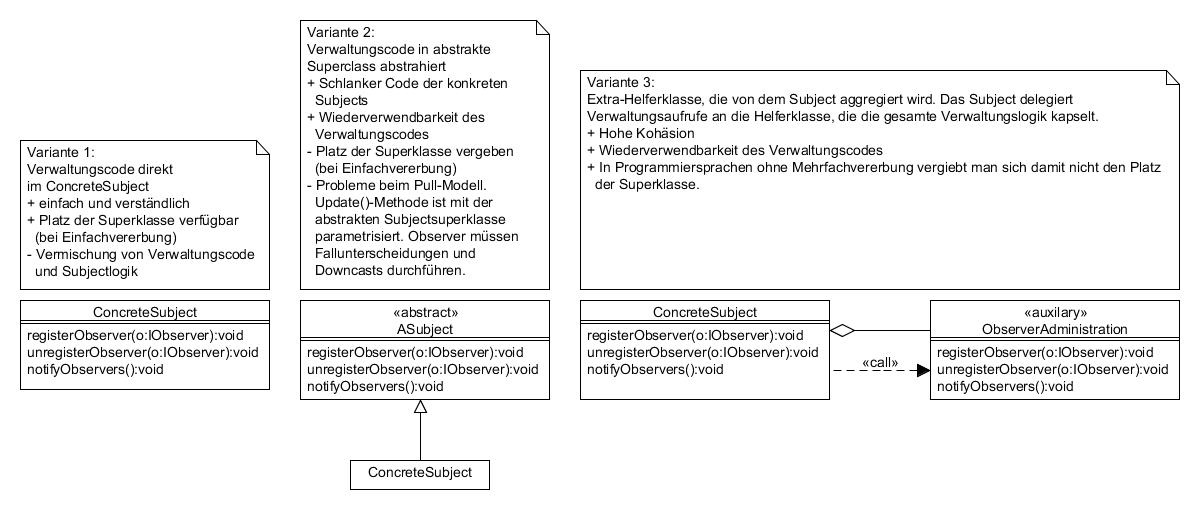


|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Keine Informationsüberladung, da sich jeder ConcreteObserver seine benötigten Informationen selbst holen kann. Lediglich muss das ConcreteSubject getter-Methoden zur Verfügung stellen. | * Ineffizienz, da der ConcreteObserver herausfinden muss, was sich geändert hat. |

Das Push-Verfahren sollte bevorzugt werden, wenn das ConcreteSubject über die Bedürfnisse seiner Observer weiß.

Das Pull-Verfahren hingegen sollte bevorzugt werden, wenn das ConcreteSubject nichts über seine Observer weiß.

Ergänzend zu den beiden Verfahren der Informationsbeschaffung, sollte man noch folgende Punkte beachten:

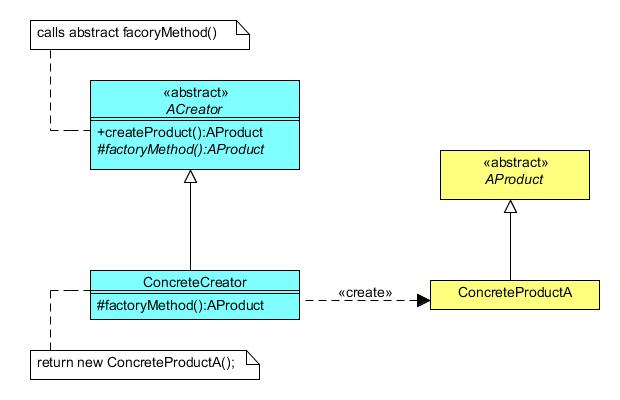
* Wenn kein allgemeiner Administrations- und Aktualisierungscode von ConcreteSubjects abstrahiert werden kann, da sie zu unterschiedlich sind, bietet sich ein Interface als Subjectschnittstelle an.  
  Wenn das Subject nicht ausgetauscht werden muss, kann auch keine Subjectschnittstelle definiert werden.
* Es gibt zudem 3 Möglichkeiten, wo man den Verwaltungscode des Subjects für die Observer hinpacken will:  
  
* Ruft das ConcreteSubject nach jeder Zustandsveränderung selbstständig notifyObservers() (hier protected deklariert) z.B. in Setters auf, so kann es zwar nie vergessen werden, kann aber zu unnötigen Aktualisierungen führen, wenn jedes Mal alle Observer bei mehreren nacheinander folgenden Zustandsänderungen des Subjects benachrichtigt werden.   
  Ruft der Client selbst notifyObservers() (hier public deklariert) auf, kann er dies erst nachdem er alle Subjectzustände modifiziert hat. Die Gefahr ist allerdings groß, dass dies vergessen wird.

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Die Daten im Gesamtsystem bleiben konsistent, da die angemeldeten Observer ihren Zustand automatisch bei Änderung des Subjects anpassen. * Beliebig viele Observer können beliebig viele Subjects beobachten. Klassen können sowohl Subject als auch Observer sein. * Subject und Observer sind lose und abstrakt gekoppelt, können somit in verschiedenen Abstraktionsschichten eines Systems liegen und lassen sich unabhängig voneinander variieren. | * Wenn man ein Observerobjekt beim Subject nicht mehr braucht, kann es passieren, dass man vergisst es abzumelden. Dies kann in Fällen von Mehrfachanmeldung merkwürdige Effekte zur Folge haben und verhindert die automatische Speicherfreisetzung (Garbage Collector). * Bei umfangreichen Systemen mit vielen Subjects und Observer kann eine Änderung eine ganze Änderungskette nach sich ziehen oder im schlimmsten Fall zu sich rekursiv wiederholenden Aufrufen führen. Außerdem steigt bei komplexen Subject-Observer-Interaktionen die Gefahr von **unnötigen Aktualisierungen**. Daraus resultierende Fehler sind nur schwer ausfindig zu machen. |

Factory Method und Abstract Factory Method

Allgemeines

Zwecke einer Factory ist der flexible Austausch von Implementierungen.

Factory Method

Beschreibung

Das Factory Method Entwurfsmuster wird oft zur Trennung von zentraler Objektverarbeitung und individueller Objektherstellung verwendet.

Der Erstellungscode für Products wird in eine abstrakte Creator-Klasse ausgelagert, die wiederum die konkrete Objektinstanziierung an ihre subClass, ConcreteCreator, delegiert. Concrete Creator entscheidet selbst, welches konkrete Produktobjekt erstellt wird, und übergibt es an den Creator. Der Creator kann noch allgemeinen Herstellungscode enthalten, die auf jedes Produktobjekt angewandt werden muss, bevor es an den Client geliefert wird.

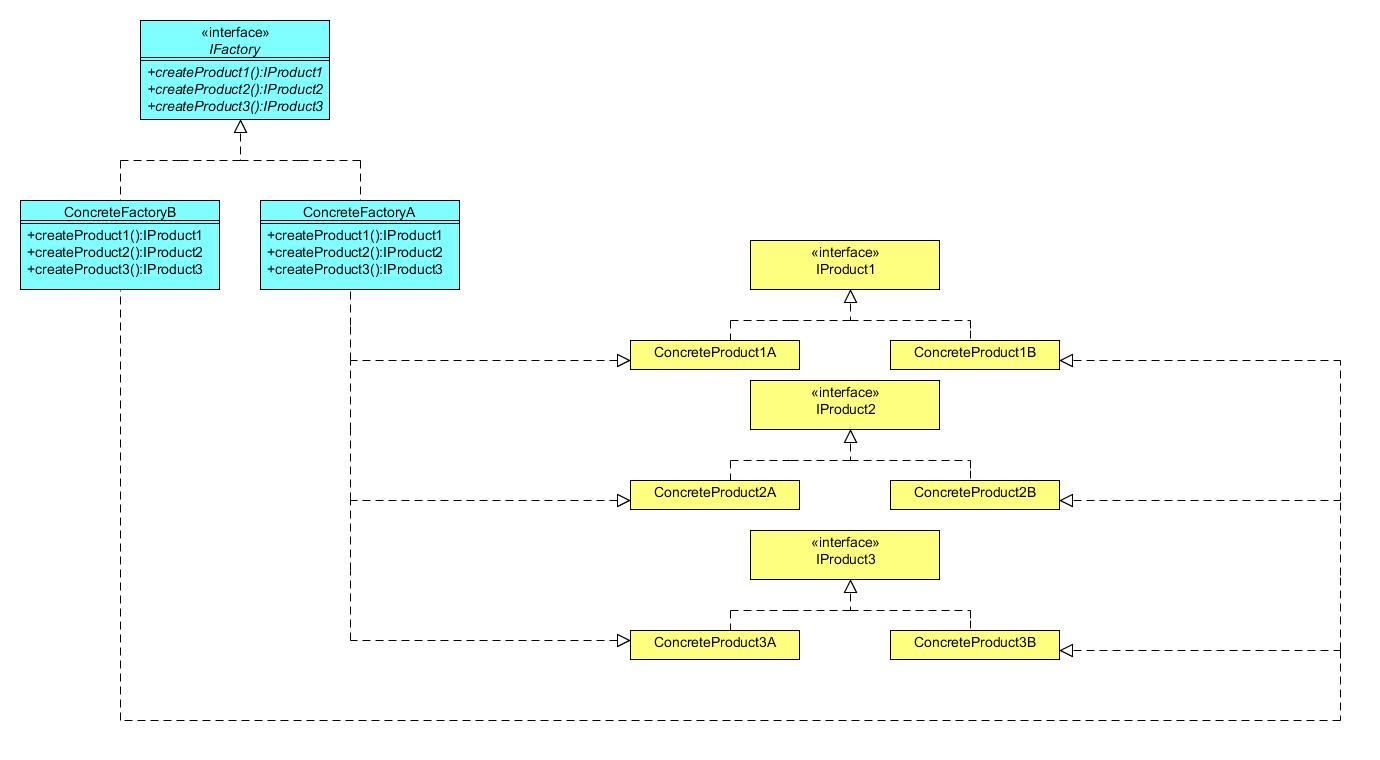
Implementierungshinweise

Eine Unterklassenbildung kann auch optional werden, indem der Creator konkret wird und eine Defaultimplementierung für die Fabrikmethode enthält, die die Unterklassen überschreiben können.

Die Fabrikmethode kann außerdem parametrisiert sein, um Produktinformationen an den Creator mitzugeben. Der Vorteil ist ein schnelles und unkompliziertes Hinzufügen und Austauschen von neuen Produktobjekten.

Bei einer Fabrikmethode ohne Parameter kann jeder ConcreteCreator genau ein Produkt herstellen. Für weitere Produkte müssen neue ConcreteCreator gebildet werden. Wenn eine geringe Anzahl von Produkten hergestellt werden soll, kann diese Variante sinnvoll sein.

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Unterschiedliche Produktimplementierungen können denselben Produktionsvorgang durchlaufen, da Herstellungsprozess von der konkreten Implementierung getrennt ist. * Erweiterbarkeit, Wiederverwendbarkeit und Wartbarkeit durch die Schnittstellen, ACreator und AProduct. | * Enge Kopplung eines ConcreteCreators an ein ConcreteProduct - bei einer parametrisierten Fabrikmethode muss ein ConcreteCreator angepasst werden. |

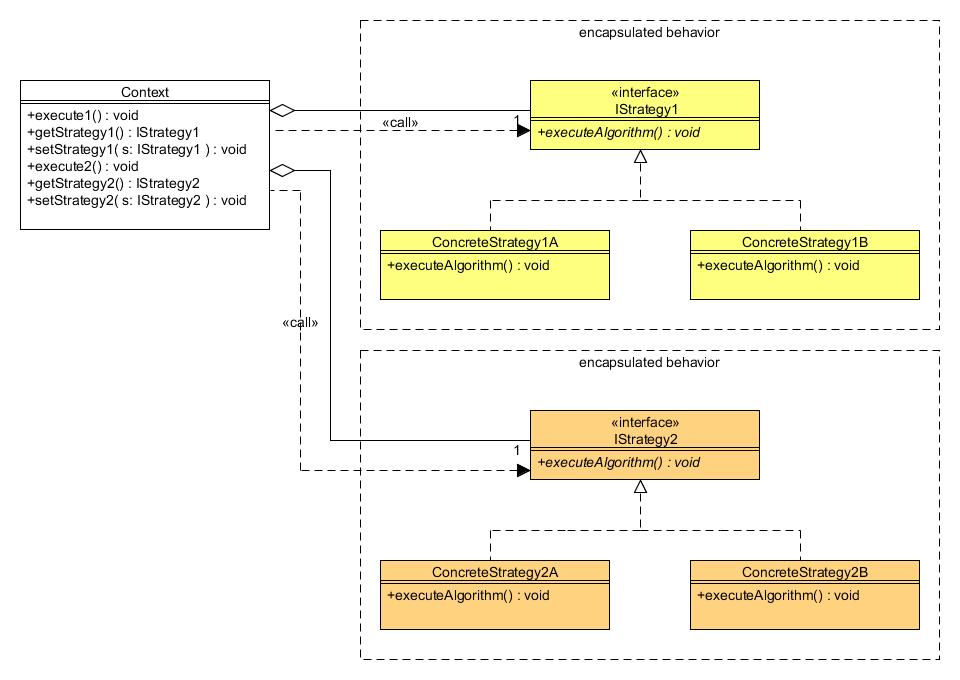
Abstract Factory

Beschreibung

Das Abstract Factory Entwurfsmuster dient der Definition von Produktfamilien und trennt die Objekterstellung von der Objektverarbeitung.

Der Client kennt nur die Schnittstellen.  
Die Factoryschnittstelle definiert für jede Produktfamilie (Bsp. IProduct1, IProduct2, IProduct3) eine Methode, die der Client zur Erhaltung eines Produktobjekts der jeweiligen Produktfamilie aufruft. Da die konkrete Factory die Erstellungsmethoden der Factoryschnittstelle implementiert, wird sichergestellt, dass der Client nur mit zusammen gehörenden Produkten, Produktsets, (Bsp. ConcreteProduct1A, ConcreteProduct2A, ConcreteProduct3A) arbeitet.

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Clientcode ist allgemeingültig und schlank. * Konsistenz: genau ein konkretes Produkt pro Produktfamilie darf im Einsatz sein und nur zusammenhängende Produkte werden an den Client übergeben. * Möchte man ein neues Produktset hinzufügen, muss man lediglich eine konkrete Factory hierfür erstellen. * Konkrete Produkte können zu verschiedenen Sets gehören. | * **Unflexibilität hinsichtlich neuer Produktfamilie: Möchte man eine neue Produktfamilie einführen (Bsp. IProduct4, so ist der Änderungsaufwand groß, da neben der Factoryschnittstelle, alle konkreten Factorys angepasst werden müssen.** |

Strategy

Beschreibung

Das Strategy Entwurfsmuster dient zur Definition von Familien austauschbaren Verhaltens.  
Das Verhalten eines Contextobjekts wird in eine eigene Schnittstelle ausgelagert. Das Contextobjekt ist somit implementierungsunabhängig und kann mit neuem Verhalten ausgestattet werden, ohne dass sein Code verändert werden muss, wobei die neue Strategie die Schnittstelle korrekt implementieren muss. Außerdem können andere Contextklassen die Strategien wiederverwenden.

Implementierungshinweise

**Da der Client die passenden Strategien des Contextobjekts setzt, muss er folglich die Implementierungen kennen. Das Strategy Entwurfsmuster sollte nur genutzt werden, wenn das Variieren des Verhaltens für den Client wichtig ist. Hier kann aber mit einer Factory Abhilfe geschaffen werden, denn damit lässt sich die Erstellungslogik aus dem Client auslagern.**

|  |  |
| --- | --- |
| **Vorteile** | **Nachteile** |
| * Client kann das Verhalten eines Contextobjekts sowohl zur Design- als auch zur Laufzeit mit setter-Methoden **dynamisch** ändern. * Neuartige Contextklassen mit neuem Verhalten entstehen durch Erstellung neuer Verhaltensklassen oder neue Kombination bestehender Verhaltensklassen. -> **Code-Recycling** + **hohe** **Kohäsion** * **Alternativimplementierung: Auch kann dieselbe Funktion durch verschiedene Implementierungen angeboten werden, die sich in nichtfunktionaler Hinsicht unterscheiden, Bsp. Performance, Speicherbedarf.** | * **Unnötige Context-Strategie-Kommunikation möglich, wenn im Interface eine parametrisierte Methode definiert ist, dessen Parameter von konkreten Strategien nicht verarbeitet werden.** |

Meine Idee:

Mix aus Kompositum, Strategie und Beobachter.