Design Patterns

Christopher Alexander:

> Ein Design Pattern beschreibt ein Problem, das immer wieder in unserer Umgebung anzutreffen ist. Dabei soll die Lösung für dieses Problem so sein, dass Sie diese Lösung millionenfach verwenden können, ohne diese zweimal auf dieselbe Weise zu verwenden.

Design Pattern sind Beweis für bewährte Software in der Praxis. Ein Baustein mit verschiedenen Abstraktionsebenen: Idiom (Coplien, 1991), Design Pattern (Gamma et al., 1995) und Architekturmuster (Buschmann et al., 1996).

Idiome

Idiome sind keine objektorientierte Design Patterns!

Ein Idiom ist ein Muster auf niedriger Ebene, normalerweise auch spezifisch für eine Programmiersprache. Hier zum Beispiel String-Copy in C, wobei s und d Zeichenarrays sind.

```
while (* d ++ = * s ++);
```

Faule Instantiierung von Singletons in Java (Double-checked Locking Idiom):

```
private static Device device = null;
  public static Device instance() {
    if (device == null) {
      synchronized (Device.class) {
        if (device == null) {
          device = new Device();
        }
      }
    }
  return device;
}
```

Template Method

Designziel ist beispielsweise einen Algorithmus so zu implementieren, dass bestimmte Abschnitte später angepasst oder geändert werden können. Definieren Sie hierzu ein Skelett des Algorithmus in einer Operation und lagern Sie dabei einige Schritte auf Unterklassen aus. Diese Herangehensweise wir oft in Frameworks und APIsverwendet.

Verwenden Sie das Template Method Pattern, für getrennte Varianten und Invarianten. Vermeiden Sie dadurch Duplizierung von Code. Das allgemeine Verhalten wird in einer gemeinsamen Klasse zusammengefasst und dirt lokalisiert.

Die Template Method ist die Methode, die den Algorithmus unter Verwendung abstrakter und konkreter Operationen definiert. Außerdem können abstrakte Operationen überschrieben und Hook-Operationen definiert werden.

Motivation hinter Design Pattern

Da das entwickeln von wiederverwendbarer und erweiterbarer Software oft sehr schwer ist und Entwickler, welche zuvor nicht bei der Entwicklung dabei waren meistens überfordert sind, hat man Pattern eingeführt um sich leichter einen Überblick verschaffen zu können. Pattern sind stark an Erfahrungen angelehnt und einige Designlösungen kommen immer wieder vor. Dadurch ist das Verständnis wiederkehrender Lösungen leichter zu verstehen und gibt viel Information wo welche Änderungen vorgenommen werden müssen. Dadurch hat dich der Aufbau auf eine generische Weise etabliert durch die Konsequenzen und Kompromisse leichter verständlich sind.

Architektonische Muster oder Architectural Patterns

Architektonische Muster sind keine Design Pattern!

Architekturmuster helfen lediglich, die grundlegende Struktur eines Softwaresystems oder wichtiger Teile davon zu bestimmen. Architekturmuster haben einen wesentlichen Einfluss auf das Erscheinungsbild konkreter Softwarearchitekturen. Dabei definieren Architekturmuster die globalen Eigenschaften eines Systems, zum Beispiel, wie verteilte Komponenten zusammenarbeiten und Daten austauscht werden oder auch die Grenzen für Subsysteme. Die Auswahl eines Architekturmusters ist eine grundlegende Entwurfsentscheidung.

Bekannte Architekturmuster sind dabei:

- Pipes and Filters
- Broker Pattern
- MVC
- Broker

Oft reicht es aber nicht aus, nur ein architektonisches Muster zu wählen. Stattdessen müssen mehrere architektonische Muster kombiniert werden.

Model-View Controller (MVC)

Das MVC-Muster beschreibt eine grundlegende strukturelle Organisation für interaktive Softwaresysteme dabei enthält das Modell die wichtigsten Funktionen und Daten wobei es unabhängig von Ausgabedarstellungen oder Eingabeverhalten ist.

Die Benutzeroberfläche besteht aus: • der Ansicht / Ansichten, die dem Benutzer Informationen anzeigen. Die Ansicht bezieht die Daten aus dem Modell. • dem Controller, der die Benutzereingaben verarbeitet. Dabei hat jede Ansicht einen eigenen Controller, der die Eingaben empfängt. Die

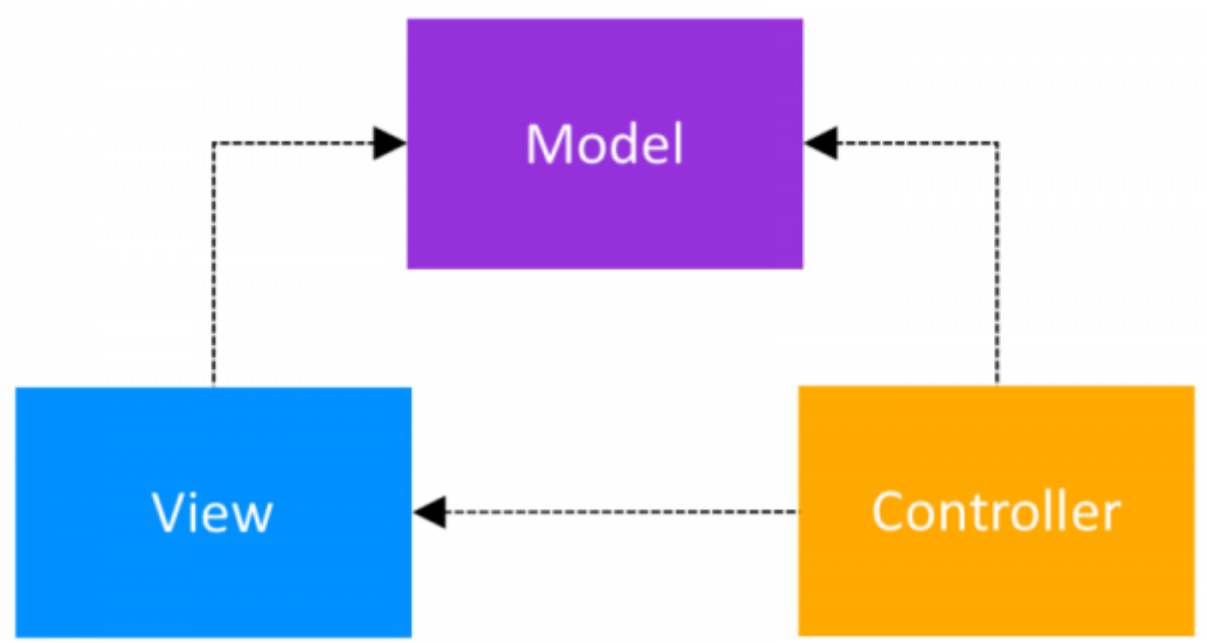
Ereignisse werden dann in Serviceanforderungen für das Modell oder die Ansicht übersetzt. Alle Interaktionen laufen dabei über einen Controller.

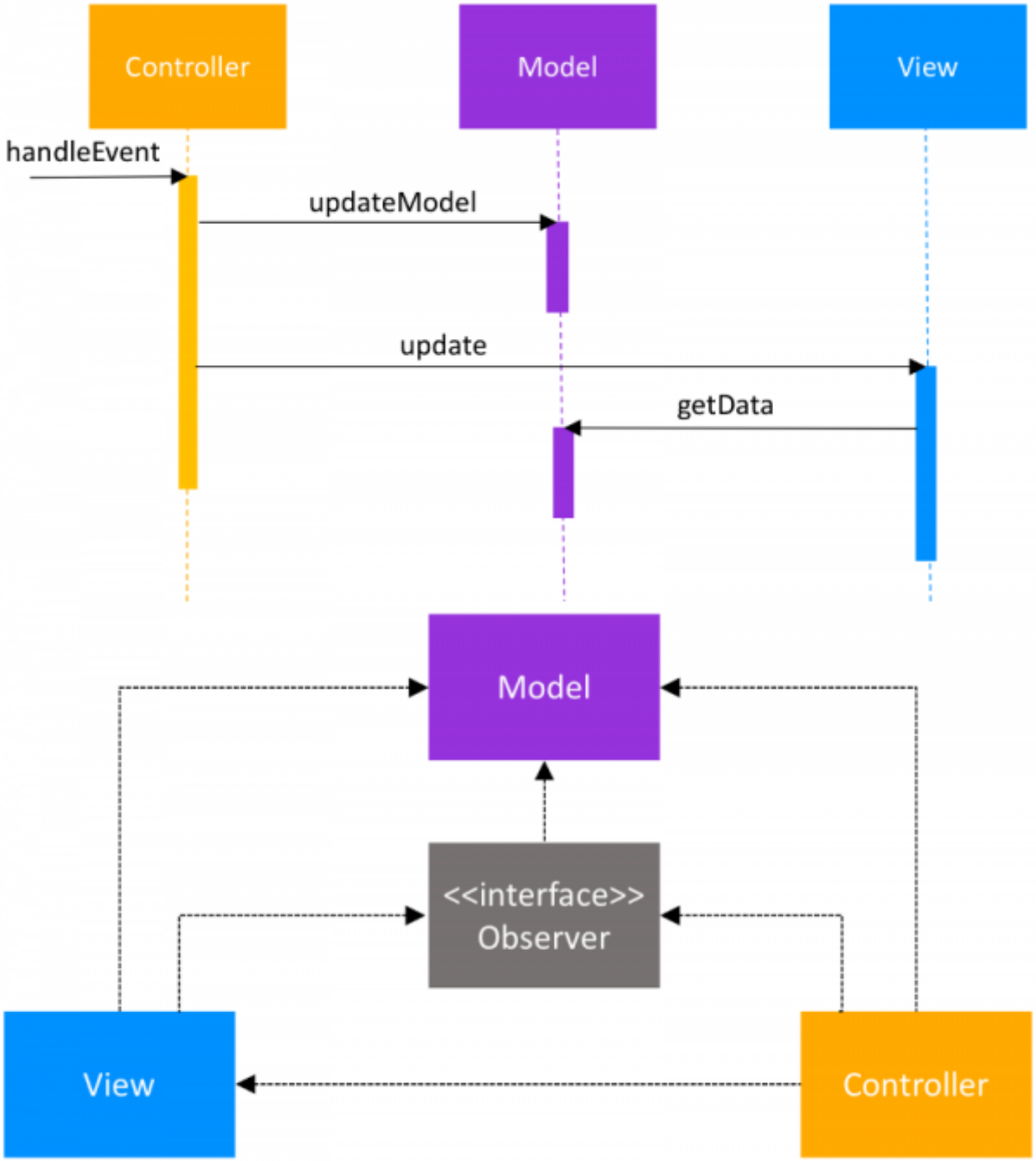
Model-View Controller (MVC) Change Propagation

Die Change Propagation stellt die Konsistenz zwischen der Benutzeroberfläche und dem Modell sicher. Dieser Mechanismus wird normalerweise mithilfe des Observer-Musters (Observer pattern) und dem Publisher-Subscriber-Musters (Publisher Subscriber pattern) implementiert. Dabei ist die Grundidee, eine Ansicht „registriert“ sich am Modell. Wenn das Verhalten eines Controllers vom Status des Modells abhängt, „registriert“ sich der Controller beim Mechanismus für die Weitergabe von Änderungen.

Verwenden Sie das MVC-Muster zum Erstellen interaktiver Anwendungen mit einer flexiblen „Mensch-Computer“-Schnittstelle. Wenn die gleichen Informationen unterschiedlich dargestellt werden sollen, zum Beispiel in verschiedenen Fenstern. Oder die Anzeige und das Verhalten einer Anwendung die Veränderungen von Daten sofort widerspiegeln müssen. Oder zum Portieren der Benutzeroberfläche da dann der Code im Kern der Anwendung nicht beeinflusst wird.

Die Structure des Model-View Controller (MVC)





Quelle : medium.com

Während der Controller und die Ansicht direkt mit dem Modell gekoppelt sind, ist das Modell nicht direkt mit dem Controller oder der Ansicht gekoppelt.

Verbindlichkeiten von Model-View Controller (MVC)

Erhöhte Komplexität durch die Verwendung separater Ansichts- und Steuerungskomponenten, ohne zugewinn von Flexibilität. Möglicherweise entstehen übermäßig viele Update-Befehle, beachten Sie hierbei dass nicht alle Ansichten sind immer an allen Änderungen interessiert sind. Es kommt zu „intimen Verbindungen“ zwischen View und Controller.

Benefits von Design Patterns

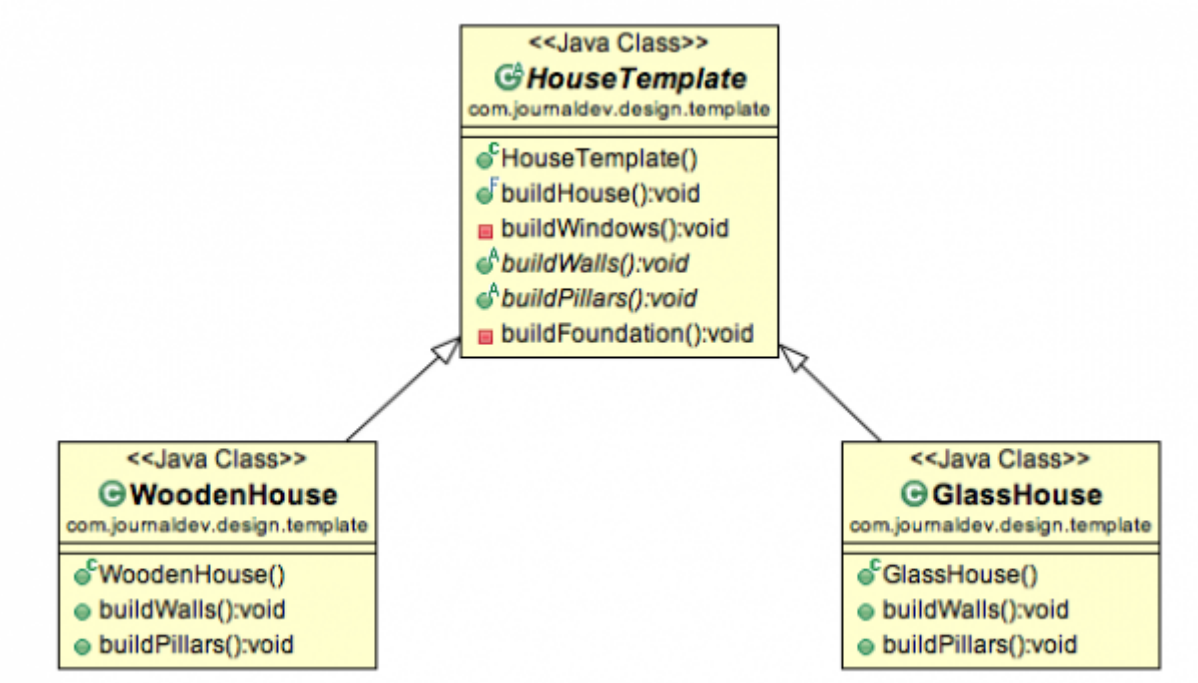
- Systematische (Software-) Entwicklung
- leichtere Dokumentation durch Expertenwissen
- Verwendung generischer Lösungen
- Erhöhung der Abstraktionsebene

1. Name des Design Patter: Eine kurze Mnemonik zur Erweiterung Ihres Designvokabulars.
2. Problem: Beschreibung, warum das Muster angewendet werden soll. Eine Bedingungen, die erfüllt sein müssen, bevor das Muster angewendet werden kann
3. Lösung: Die Elemente, aus denen das Design besteht, deren Beziehungen, Verantwortlichkeiten und Zusammenarbeit.
4. Folgen: Kosten und Nutzen der Anwendung des Musters. Sprach- und Implementierungsprobleme sowie Auswirkungen auf Systemflexibilität, Erweiterbarkeit oder Portabilität. Ziel ist es, ein Muster zu verstehen und zu bewerten.

Template für ein Design Patterns

1.	Name und Absicht
2.	Motivation und Anwendung
3.	Struktur, Teilnehmer, Zusammenarbeit und Implementierung
4.	Folgen oder Konsequenzen
5.	Bekannte Verwendungen und Verwandte Muster

Um ein verwendetes Entwurfsmuster zu dokumentieren, verwenden Sie die Namen des Musters, um die Rolle der Klasse bei der Implementierung von Mustern anzugeben.



Quelle: journaldev.com

Levels of Consciousness eines Design Pattern

1. Unschuld
2. Bekannte Tricks
3. Kompetente Trickanwendung
4. Anwendbarkeit & Konsequenzen bekannt
5. Breites Wissen über Muster und deren Interaktion
6. Fähigkeit, Wissen in gebildeter Form zu erfassen

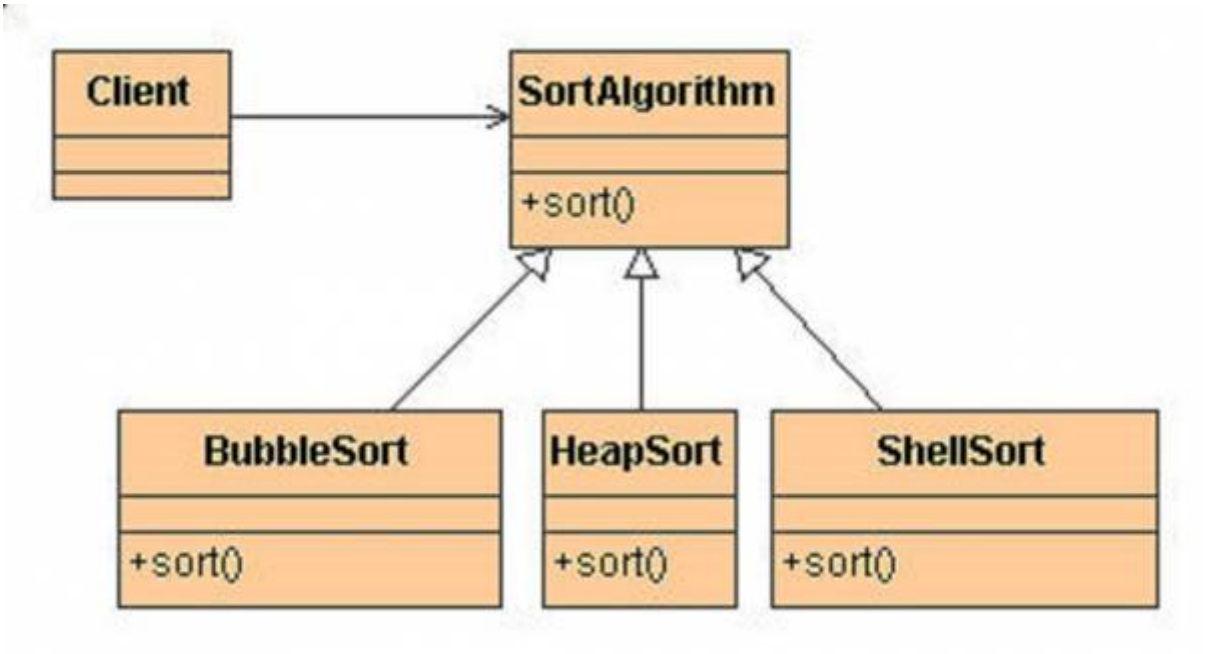
Muster ermöglichen den Aufbau hochwertiger Softwarearchitekturen.

Ein Softwareentwurfsmuster beschreibt eine häufig wiederkehrende Struktur interagierender Softwarekomponenten, die ein allgemeines Softwareentwurfsproblem in einem bestimmten Kontext lösen.

Idiome, Designmuster und Architekturmuster helfen Ihnen, wiederkehrende Probleme auf verschiedenen Abstraktionsstufen zu lösen und die Vorteile und Kompromisse sofort zu verstehen. Mithilfe von Mustern können Sie auf einer höheren Abstraktionsstufe über das Design Ihrer Anwendung sprechen.

Das Strategy Design Pattern

Das Strategy Design Pattern unterstützt verschiedene Arten von externen Fremdservices, z.B. zur Berechnung von Steuern. Ebenso unterstützt es verschiedene Arten von Datenbankverbindungen oder wenn unterschiedliche Werte sortiert werden sollen. Definieren Sie eine Familie von Algorithmen, kapseln Sie diese und machen Sie diese austauschbar. Das Strategy Design Pattern lässt den Algorithmus unabhängig von den Clients variieren.



Quelle: blogspot.com

Eine Alternative zum Unterklassen

Unterklassen mischen die Implementierung des Algorithmus mit dem Context. dadurch ist der Kontext schwieriger zu verstehen, zu pflegen und zu erweitern.

- Bei der Verwendung von Unterklassen können wir den Algorithmus nicht dynamisch variieren
- Unterklassen führen zu vielen verwandten Klassen
- Sie unterscheiden sich lediglich in dem verwendeten Algorithmus oder Verhalten.

Einschluss des Algorithmus in Strategie lässt den Algorithmus unabhängig vom Kontext variieren und erleichtert dadurch das Wechseln, Verstehen, Wiederverwenden und Erweitern des Algorithmus.

Wann ist das Strategy Design Pattern anzuwenden?

- bei vielen verwandten Klassen, welche sich nur in ihrem Verhalten unterscheiden, anstatt verschiedene verwandte Abstraktionen zu implementieren.
- Strategy Design Pattern ermöglichen das Konfigurieren einer Klasse mit einem von vielen Verhalten.
- Sie benötigen verschiedene Varianten eines Algorithmus
- Strategy Design Pattern können verwendet werden, wenn Varianten von Algorithmen als Klassenhierarchie implementiert werden.
- eine Klasse definiert viele Verhalten, die in ihren Vorgängen als mehrere bedingte Anweisungen erscheinen
- Verschieben Sie verwandte bedingte Verzweigungen in eine Strategie.

Clients müssen sich der unterschiedlichen Strategien und deren Unterschiede bewusst sein, um die passende Strategie auszuwählen Aber Clients können auch Implementierungsproblemen ausgesetzt sein. Strategien sollten nur verwendet werden, wenn eine Verhaltensvariation für den Client relevant ist.

Optionale Strategieobjekte Kontext prüft, ob eine Strategie vorhanden ist, bevor auf diese zugegriffen wird. Falls eine vorhanden ist, verwenden Sie normalerweise Kontext. Falls keine Strategie vorhanden

ist, führt der Kontext das Standardverhalten aus. Der Vorteil ist, dass Kunden sich nicht mit Strategieobjekten befassen müssen, es sei denn, sie sind dazu verpflichtet.

- erhöhen Sie die Anzahl der (Strategie-)Objekte
- Manchmal können stateless-Strategien reduziert werden, um den Kontext gemeinsam nutzen können.
- Jeder Status wird von Kontext verwaltet und übergibt diesen für jede Anforderung an das Strategieobjekt. Dadurch entsteht keine oder nur wenige Kopplung zwischen Strategieimplementierungen und Kontext.
- Geteilte Strategien sollten den Status nicht über Aufrufe hinweg aufrechterhalten

Die Strategie-Schnittstelle wird von allen Concrete-Strategy-Klassen gemeinsam genutzt, unabhängig davon, ob die von ihnen implementierten Algorithmen trivial oder komplex sind. Einige ConcreteStrategies verwenden nicht alle Informationen, die an sie weitergegeben werden Simple ConcreteStrategies verwendet möglicherweise keine davon. Der Kontext erstellt bzw. initialisiert auch Parameter, die niemals verwendet werden. Wenn dies ein Problem ist, verwenden Sie eine engere Kopplung zwischen Strategie und Kontext. Lassen Sie die Strategie mehr über den Kontext wissen.

Vergessen Sie aber dabei den Communication Overhead nicht!

Strategiesichtbarkeit für die Kontextinformationen, die die Strategie benötigt; zwei mögliche Strategien:

Übergeben Sie die benötigten Informationen als Parameter …

- Kontext und Strategie entkoppelt
- Kommunikationsaufwand
- Der Algorithmus kann nicht an spezifische Kontextanforderungen angepasst werden

Kontext übergibt sich selbst als Parameter oder Strategie hat einen Bezug zu seinem Kontext …

- Reduzierter Kommunikationsaufwand
- Kontext muss eine komplexere Schnittstelle zu seinen Daten definieren
- Engere Kopplung von Strategie und Kontext

Bei Verwendung des Strategiemusters hängen sowohl die Vorlage als auch die detaillierten Implementierungen von Abstraktionen also den Schnittstellen ab.