

Das Command Design Pattern

 Send to Kindle

Studienprojekt von Philipp Hauer. 2009 - 2010. ©

Zur Katalogübersicht

Strategy, Observer, Decorator, Factory Method, Abstract Factory, Singleton, Command, Composite, Facade, State

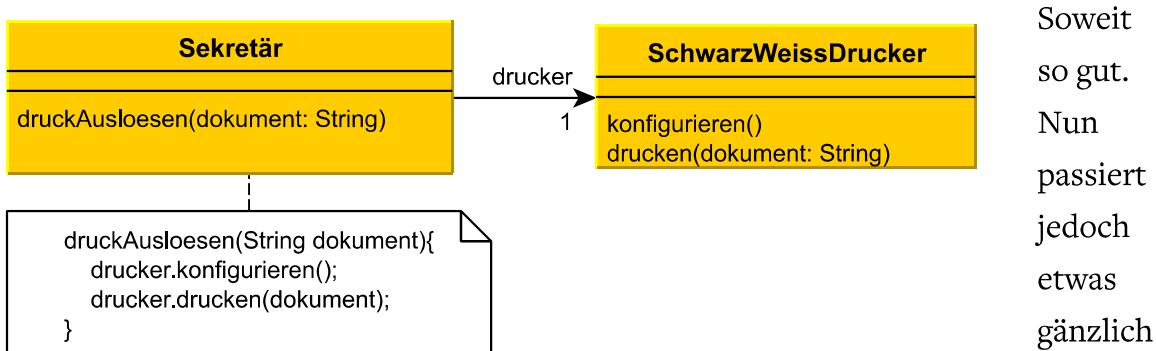
Literaturverzeichnis, Philipps Blog

Inhalt

Einführung

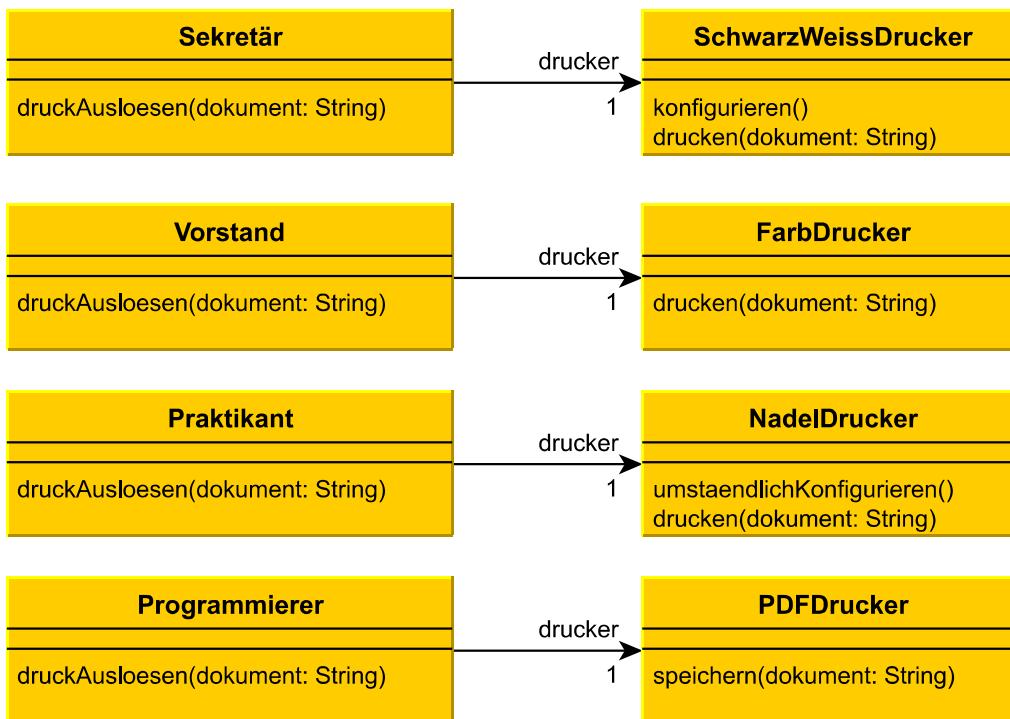
Wir wollen Mitarbeiter modellieren. Dabei interessiert uns - im ersten Schritt - nur die Fähigkeit des Sekretärs, etwas ausdrucken zu können: Wird ihm gesagt, er soll etwas ausdrucken, so geht er zu dem firmeninternen Schwarz-Weiß-Drucker, nimmt Einstellungen vor und druckt das gewünschte Dokument aus.

Damit der Sekretär seinen Drucker auch kennt, geben wir ihm eine Referenz auf den SchwarzWeißDrucker.



Unerwartetes: Die Firma erwirbt einen modernen Farbdrucker. Allerdings dürfen nur ausgewählte Mitarbeiter in den Genuss des Farbdruckers kommen, wie der Vorstand. Um die entstandenen Anschaffungskosten zu kompensieren, wird gleichzeitig ein billiger Nadeldrucker angeschafft, damit der Schwarz-Weiß-Drucker entlastet wird. Praktikanten "dürfen" mit Nadeldruckern "arbeiten".

Nun, offensichtlich ist dies ein suboptimaler Entwurf. Warum?



- *Inflexibilität.* Es wird klar, dass eine harte Zuordnung von Mitarbeiter und Drucker äußerst ungünstig ist. Was, wenn ein Sekretär plötzlich doch etwas farbig ausdrucken *muss*? Die Zuordnung zum Drucker kann nicht dynamisch zur Laufzeit erfolgen. Noch schlimmer: Er kann selbst zur Compilezeit nur mit mittelgroßen, destruktiven und irreversiblen Codeänderungen durchgeführt werden. Dies ist durch die enge Kopplung im Code bedingt: Mitarbeiter und Drucker sind auf ewig fest aneinander gekettet.
- Keine *Wiederverwendbarkeit*. Befehle können nicht wiederverwendet werden. Soll nun der Werkstudent ebenfalls den Schwarz-Weiß-Drucker verwenden, so muss die Methode `druckAuslösen()` für den Schwarz-Weiß-Drucker im Werkstudent komplett neu implementiert werden. Es folgen *Coderedundanzen* (mehrmals die identische Bedeutung eines speziellen Druckers implementieren) und damit die Gefahr von *Inkonsistenzen*.

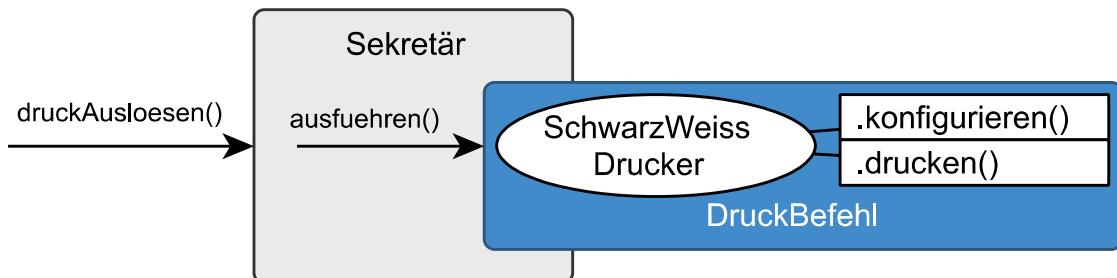
Das *Strategy* Design Pattern würde diesen Umstand versuchen aufzulösen, in dem es eine Druckerschnittstelle einführt. Den Mitarbeitern ist fortan nur noch die Druckerschnittstelle bekannt und sie delegieren ihren Druckaufruf an ihr Druckerobjekt. Über Getter und Setter wird der Drucker gesetzt.

Warum kann das nicht funktionieren?

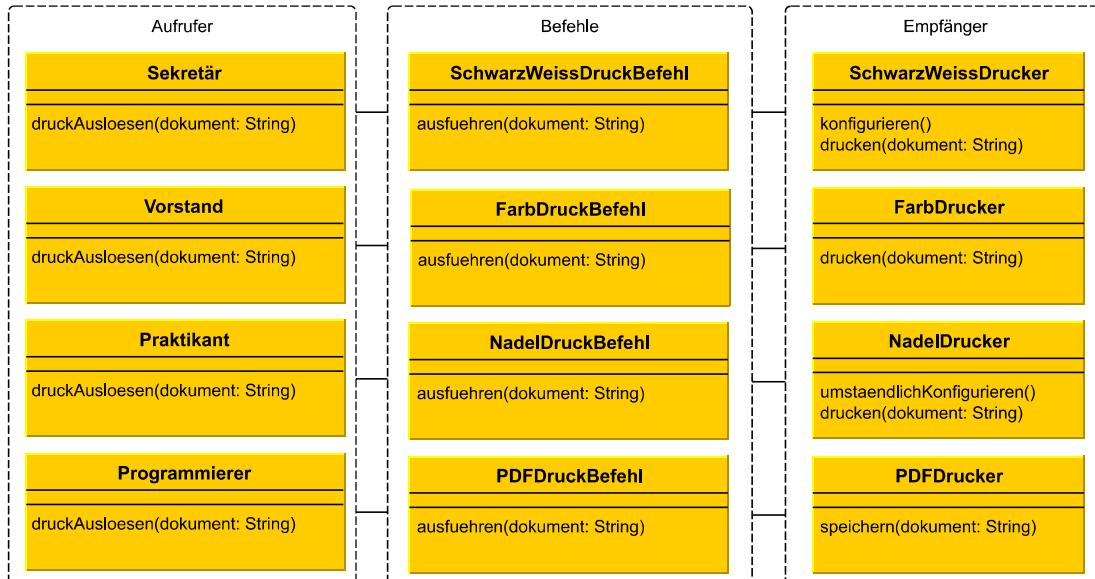
- Bei den Druckern handelt es sich *nicht* um ein *Verhalten*, das zu einem Mitarbeiter gehört, sondern um ein separates Objekt, an dem der Mitarbeiter seine Nachricht sendet.
- Außerdem sind für das Drucken mehrere Arbeitsschritte je nach Zieldrucker notwendig (`konfigurieren()`, dann `drucken()`). Dazu kommt, dass diese

Schritte druckerspezifisch sind (mal umständlich Konfigurieren(), mal nur konfigurieren(), mal gar nichts (oder speichern() statt drucken()) beim PDFDrucker) und sich nicht in eine allgemeine Druckerschnittstelle abstrahieren lassen. Jeder Drucker muss **anders behandelt** werden.

Die Lösung liegt in der Einführung einer neuen Schicht zwischen Mitarbeiter und den Druckern: Der DruckBefehl. Dieser kapselt seinen Zieldrucker und weiß *allein*, welche druckerspezifischen Schritte notwendig sind.



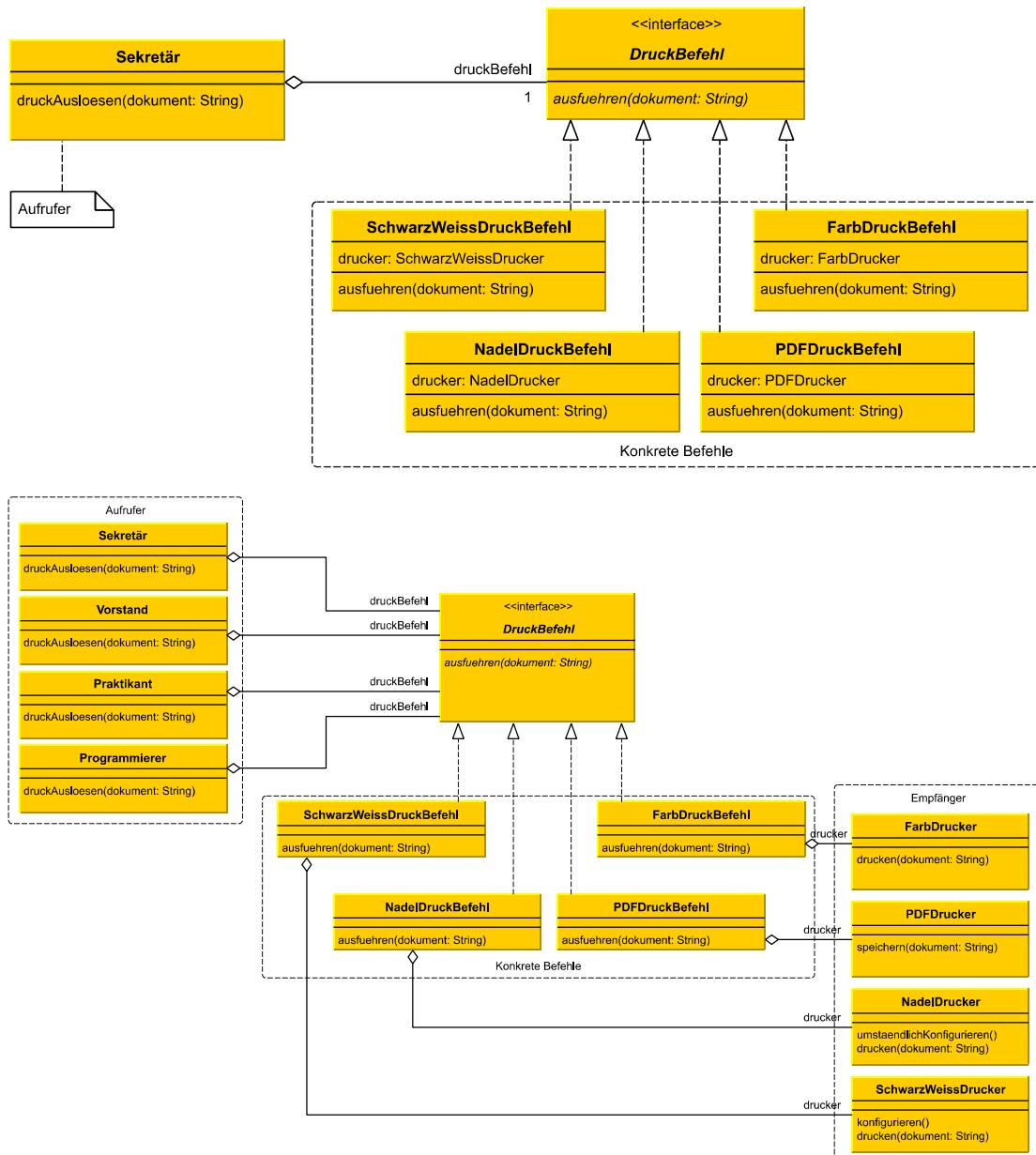
Noch einmal: **Allein der Befehl** kennt seinen **Empfänger** (Drucker) und die durchzuführenden **Schritte** (konfigurieren(), drucken()). Der Mitarbeiter, unser Befehlsaufrufer, ist von diesem Wissen befreit. Er muss lediglich den gewünschten DruckBefehl anstoßen.



Dieser Entwurf schreit förmlich nach der Einführung einer abstrakten Schnittstelle für die DruckBefehle (DruckBefehl). Die aufrufenden Mitarbeiter arbeiten folglich nur noch gegen diese Schnittstelle und delegieren die Druckarbeit an ihren Befehl. Sie wissen nicht, welcher konkrete Befehl sich dahinter verbirgt. Den Mitarbeitern ist es auch gleich, denn der Befehl weiß selbst, was mit wem zu tun ist.

In der Zusammenschau ergibt sich folgendes Klassendiagramm.

Jetzt bedarf es schließlich nur noch einer Instanz, die unser System initialisiert und



konfiguriert. Danach kann das System genutzt werden.

Die erreichten Vorteile sind umwerfend:

Hinweis: Diese Einführung behandelte nicht ein mächtiges Feature des Command Design Patterns: Die Rückgängig/Undo-Funktion, siehe dazu Variationen.

Es zeigt sich, dass durch den Einsatz des Command Patterns, ein hohes Maß an Flexibilität und Dynamik gewonnen wird, während zeitgleich die Wartung erleichtert wird und Erweiterungen schnell und unkompliziert möglich sind.

Nach dieser Einführung wird im folgenden Abschnitt das Command Design Pattern formalisiert, näher analysiert und diskutiert.

Das Druckerbeispiel mit Command Pattern Termini

Klasse	Command Teilnehmer
Sekretär, Vorstand, Praktikant, Programmierer	Aufrufer/Invoker
SchwarzWeissDrucker, Farbdrucker, Nadeldrucker, PDFDrucker	Empfänger/Receiver
DruckBefehl	Befehl/Command

Exkurs: Cyclomatic Complexity und Null-Objekt

Schauen wir uns folgenden Code von unserer Praktikanten-Klasse an.

Betrachten wir die Methode druckAusloesen(): In ihr wird geprüft, ob das Field druckBefehl null ist, bevor der eigentliche Code ausgeführt wird. Damit wird eine NullpointerException verhindert. Dies geschieht allerdings auf Kosten der Lesbarkeit. Warum? Code ist dann für den Menschen gut zu erfassen, wenn er eine geringe Verschachtelungstiefe und -breite (if, else, switch, for, while, &&, ||) ausweist. Faustregel: *So wenig wie möglich schachteln*. Eine Software-Metrik, die diese Verschachtelungstiefe misst, ist die **Cyclomatic Complexity**. Sie beziffert die minimale Anzahl an möglichen Pfaden durch den Code.

Bei einer hohen Cyclomatic Complexity sollte sich gefragt werden, ob nicht Teile des Verschachtelungsbaums in sprechende Methoden ausgelagert werden können, um die Lesbarkeit, Übersichtlichkeit und Verständlichkeit zu sichern (Top-Down-Vorgehensweise).

Zurück zu unserer Klasse: Eine Lösung für das Verschachtelungsproblem ist denkbar simpel:

Der objektorientierte Lösungsansatz dieses Problems kommt sogar ganz ohne Null-Check aus. Wir führen sogenannte **Null-Objekte** ein, die zwar das Interface DruckBefehl korrekt implementieren, jedoch **nichts tun**. Mit diesem Null-Objekt initialisieren wir unsere Instanzvariable.

Unser Code wird wunderbar **schlank** und **absturzsicher**.

Darüber hinaus kann das Null-Objekt erweitert werden, sodass es zwar weiterhin nichts tut, aber seinen Aufruf loggt. Dadurch kann erkannt werden, wo im System eine Klasse uninitialisiert verwendet wird.

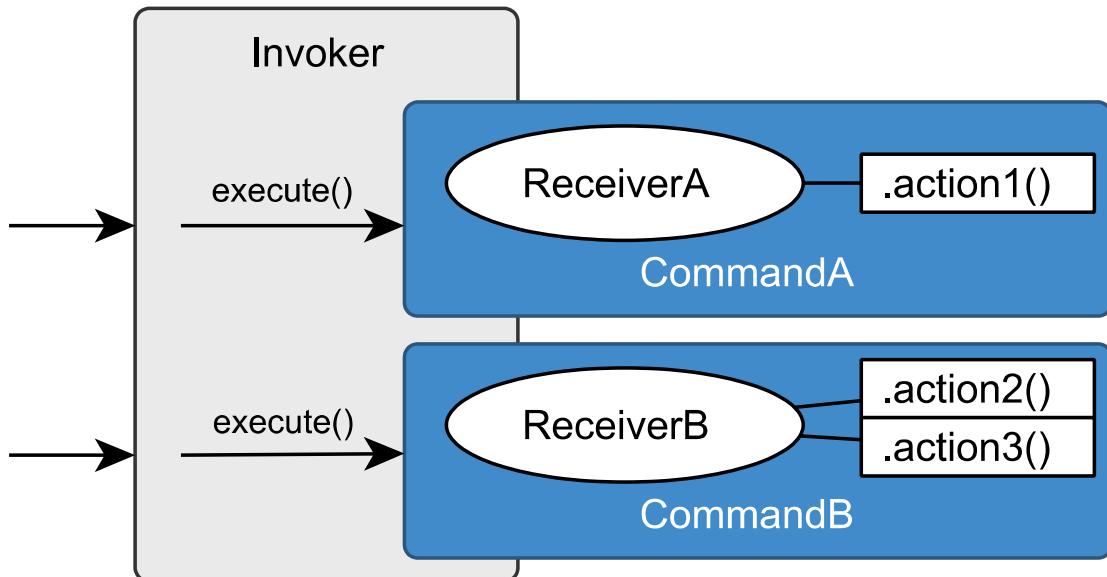
Analyse und Diskussion

Gang Of Four-Definition

Command: "Kapsle einen Befehl als ein Objekt. Dies ermöglicht es, Klienten mit verschiedenen Anfragen zu parametrisieren, Operationen in eine Schlange zu stellen, ein Logbuch zu führen und Operationen rückgängig zu machen."

([GoF], Seite 273)

Beschreibung



Das Command Design Pattern ermöglicht die Modularisierung von Befehlen und Aufrufen. Auf elegante Weise können Befehle rückgängig gemacht, protokolliert oder in einer Warteschlange gelegt werden.

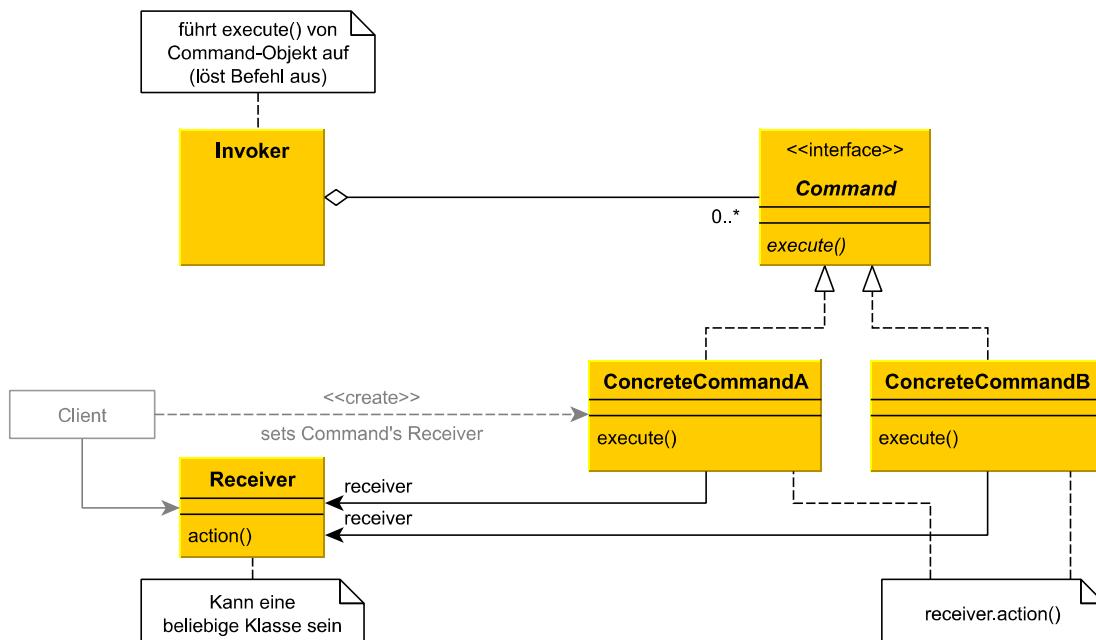
Invoker (Aufrufer) und Receiver (Empfänger) werden entkoppelt. Dazu werden die namensgebenden Command-Objekte (Befehle) zwischengeschaltet. Nur das Command-Objekt allein weiß, welche Aktionen auf welchem Empfänger auszuführen sind. So kennt ein SelectAllBefehl das gewünschte TextField und ruft `selectAll()`, die Aktion, auf diesem auf.

Dabei kennt der Aufrufer den Empfänger nicht - sie sind entkoppelt -, sondern nur die Command-Objekte. Genauer gesagt ein Commandinterface. Dadurch kann der Empfänger (beispielsweise eine Schaltfläche in einer GUI) beliebig mit Befehlen geladen werden oder bestehende können ausgetauscht werden.

Realisiert wird das Command Design Pattern wie folgt: Der Aufrufer hat ein oder mehrere Befehle (Commands), wobei er nur die Schnittstelle zu den Befehl kennt. Möchte er nun einen Befehl auslösen, so ruft er die `execute()`-Methode der

Befehlsabstraktion auf. Der dahinterliegende konkrete Befehl, kennt seinen Empfänger (beispielsweise über eine Instanzvariable) und ruft nun die gewünschten Aktionen auf diesem auf (.action()). Er **delegiert** damit den Aufruf an den Empfänger.

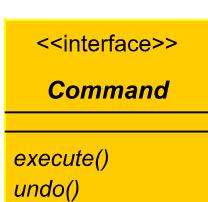
Damit diese Interaktion stattfinden kann, muss vorher der Aufrufer mit den gewünschten **Commands geladen** werden. Dazu wird vorher ein konkretes Commandobjekt erstellt, und ihm ein Empfänger (Receiver) zugewiesen.



Variationen

Rückgängig- und Wiederherstellen-Funktion (Undo, Redo)

Das wohl mächtigste Feature des Command Design Patterns ist eine elegante Rückgängig-Funktion. Dafür wird die Schnittstelle des Commandinterfaces um eine undo()-Methode erweitert.

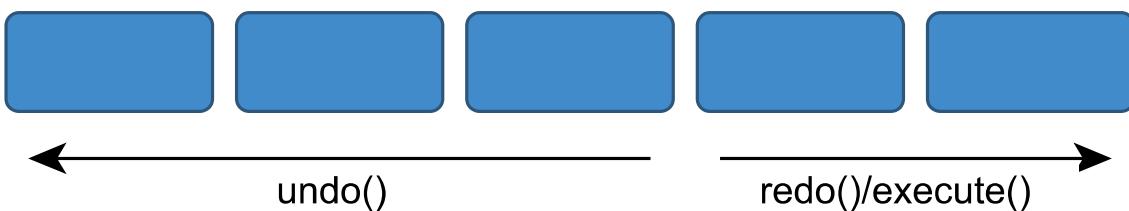


Diese Undo()-Methode enthält die exakte Umkehrung der execute()-Methode. Sorgt execute() dafür, dass eine Checkbox gesetzt wird, so deaktiviert undo() diese Checkbox wieder. In den meisten Fällen, muss sich das Befehlsobjekt die ursprünglichen Einstellungen (die während execute()) überschrieben wurden) merken, um sie korrekt wiederherstellen zu können.

So oder so muss der Aufrufer ebenfalls um ein Befehlsgedächtnis erweitert werden. Im einfachsten Fall (Rückschrittfunktion mit einer Tiefe von 1) ist das ein zusätzliches Attribut im Aufrufer. Dieses wird bei jeder Befehlausführung mit dem aktuellen Befehl gesetzt.

Denken wir diesen Gedanken konsequent zu Ende, so lassen sich mit Hilfe des Command Entwurfsmusters ganze Undo/Redo-Histories aufbauen. Der Aufrufer merkt sich in einer Liste (die History) alle bisher ausgeführten Befehle. Die Methoden `undo()` und `execute()/redo()` ermöglichen die Navigation in der Befehlshistoryliste. Um die letzten drei Befehle rückgängig zu machen, wird auf den letzten drei Befehlen in der Liste die `undo()`-Methode aufgerufen. Möchte man diese wiederum rückgängig machen (also Wiederherstellen), so wird in der Liste nach vorne navigiert und nochmals `execute()` auf den Befehlen aufgerufen, um die Änderungen wiederherzustellen.

Commandhistory:



Es gilt: Je länger die Liste der CommandHistory ist, umso mehr Undo/Redo-Ebenen sind möglich.

Intelligenz der Commandobjekte

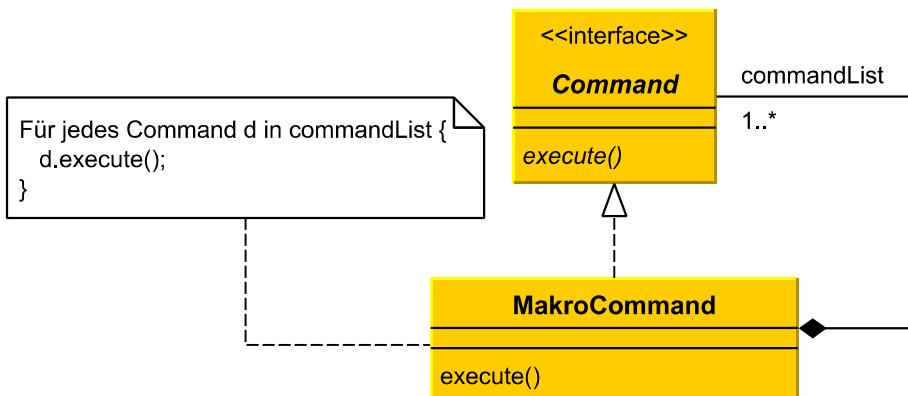
Grundsätzlich gibt es zwei Tendenzen im Funktionsumfang der Commands.



Makro-Befehle

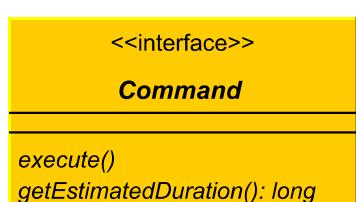
Besonders interessant ist die Möglichkeit die modularen Befehle beliebig miteinander zu kombinieren und in einem neuen Befehlsobjekte zu kapseln. Man spricht hierbei von sogenannten Makro-Befehlen.

Ein Makrobefehl hält eine Liste von Befehlen. Bei Ausführung (`execute()`) wird über die Befehlsliste iteriert und auf jedem Befehl `execute()` aufgerufen. Somit kann mit einem einzigen `execute()` eines MakroBefehls gleich mehrere Befehle auf einmal ausgeführt werden.



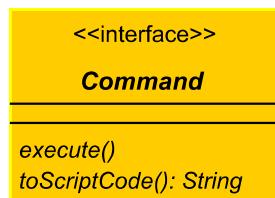
Anwendungsfälle

- **Grafische Benutzeroberflächen.**
 - In komfortablen Benutzeroberflächen kann man durch verschiedene Schalter eine identische Funktion auslösen (Kontextklick, Schaltfläche, Menüpunkt, Hotkey). Statt die gewünschte Funktion für jeden Schalter erneut zu implementieren, können sie als Invoker fungieren und Befehlsobjekte aufnehmen, die sie beim entsprechendem User-Event ausführen. Damit lassen sich alle Schaltflächen mit einem einzigen Commandobjekt laden. Weiterhin können sie flexibel und problemlos ausgetauscht werden.
 - Wegen der herausragenden Eignung des Command Patterns zur Realisierung der Rückgängig-Funktion findet es in vielen Anwendungsprogrammen Verwendung.
- **Transaktionssysteme.** Commandobjekte können in Fällen angewandt werden, in denen eine *Reihe* von Befehlen ausgeführt werden muss und es darauf ankommt, dass *alle* korrekt ausgeführt werden. Schlägt beispielsweise die letzte Aktion fehl, so können alle bis dahin durchgeföhrten Transaktionen zurückgerollt werden.
-  Bestimmung von Prozessdauer für **Fortschrtsbalken.** Besteht ein Vorgang aus mehreren Befehlen, so kann die Commandschnittstelle um eine Methode getEstimatedDuration() erweitert werden. In dieser Methode gibt jeder Befehl seine geschätzte Abarbeitungszeit zurück. Die Gesamtdauer kann damit elegant und genau ermittelt werden.



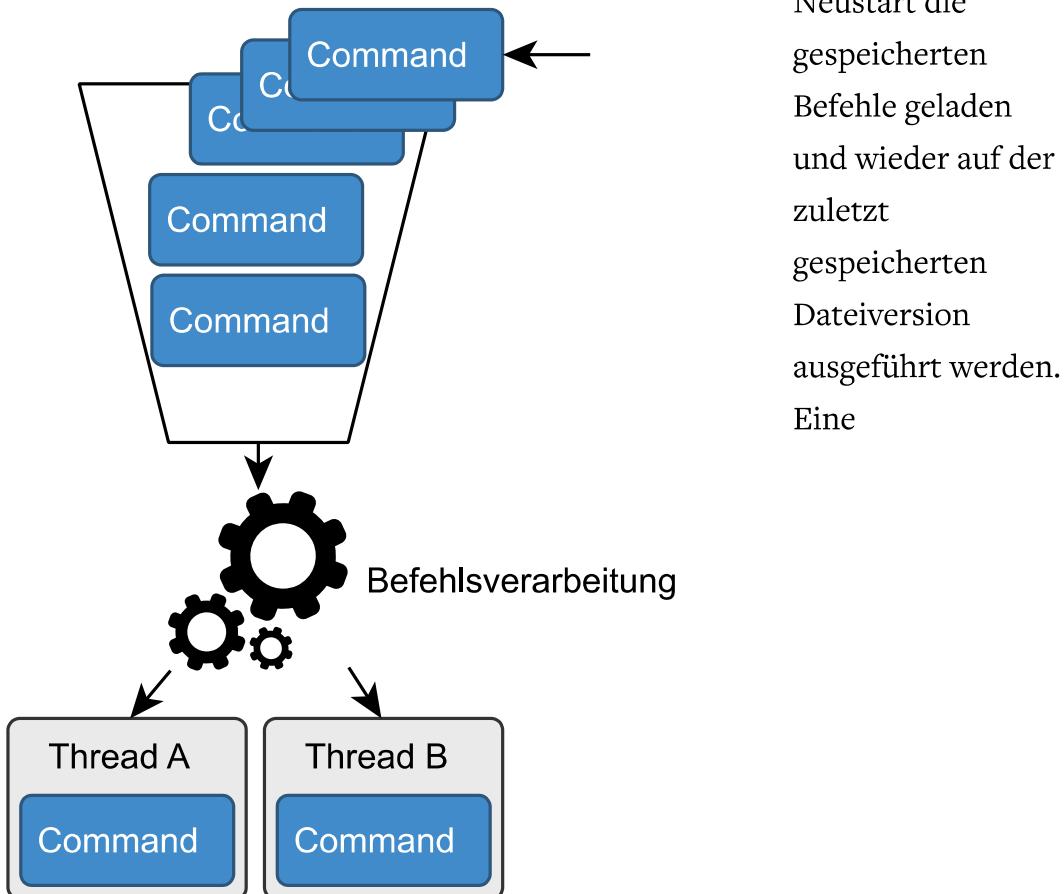
- **Makro-Recording und Scripting.**

- Viele Programme wie Photoshop oder FixFoto bieten die Möglichkeit, eine Folge von Benutzeraktionen aufzunehmen und als Makro abzuspeichern. Später kann diese Aktionsreihenfolge beliebig ausgeführt werden. Diese Funktion kann dank Command Entwurfsmuster sehr elegant implementiert werden. Es müssen lediglich alle während der Aufnahme ausgeführten Befehle in einer Liste gespeichert werden.
- Auch ist eine Speicherung dieser Befehlsreihe in einer beliebigen Skriptsprache möglich. Dazu bedarf es wieder einer Erweiterung der Commandschnittstelle.

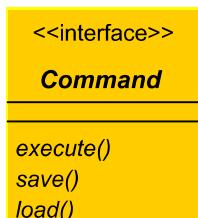


Jeder Befehl in der Liste kennt seine Entsprechung in der Skriptsprache und kann sie zurückgeben.

- **Wizards.** Wizards automatisieren Benutzeraktionen. Jede Seite des Wizards ergibt ein oder mehrere Befehlsobjekte, die ebenfalls in einer Liste gespeichert werden können. Nach der letzten Seite, wird jeder Befehl der Liste ausgeführt. Durch die mehrfache Verwendung von Befehlsobjekten (durch normaler GUI, Wizards etc.) ergibt sich ein zentraler Punkt für Wartung und Änderungen.
- **Befehlswarteschlangen und Threadpools.** Befehlsobjekten können ebenfalls in einer Warteschlange gesammelt werden. Wird ein Thread frei, nimmt er den Befehl am Kopf der Schlange ab und verarbeitet ihn. Sinnvoll ist die Verwendung eines gemeinsamen Interfaces für alle Befehle, damit die Threads sie verarbeiten können. Im Falle von Java wäre es `java.lang.Runnable`, wobei die `run()`-Methode den Verarbeitungscode des Befehls enthält. Der Thread (die Verarbeitungseinheit) ist damit vollkommen von der eigentlich auszuführenden Tätigkeit entkoppelt.
(frei nach [VKBF], Seite 228)
- **Parallele Verarbeitung.** Befehle können in einem eigenen Thread ausgeführt werden, sodass die Befehlsausführung im Hintergrund erfolgen kann.
- **Netzwerkversand.** Befehlsobjekte können serialisiert und über das Netzwerk verschickt werden, um sie in anderen JVMs auszuführen. Auch ist eine Realisierung von Remotekontrolle von entfernen Systemen denkbar.
- **Protokollierung/Logging von Befehlen.** Stürzt eine Applikation ab, so sind oft die ungespeicherten Änderungen verloren. Dem kann man entgegenwirken, in dem parallel zur Ausführung die Befehle ab dem letzten Speichern der Datei auf die Festplatte geschrieben werden (beispielsweise durch Serialisierung der Befehlsobjekte). Stürzt das System ab, kann nach einem



Wiederherstellungsfunktion wurde damit implementiert.



Vorteile

Im Zentrum des Command Design Patterns steht die **Entkopplung** von Aufrufer (Auslösen der Anfrage) und Empfänger (Ausführen der Anfrage) durch Befehlsobjekte.

Die resultierenden Vorteile sind:

- **Modularität und Wiederverwendbarkeit** von Befehlsobjekten. Befehle können von verschiedenen Aufrufern (Kontextklick, Schaltfläche, Menüpunkt, Hotkey) wiederverwendet werden.
- **Flexibilität und Dynamik.**
 - Das Austauschen von Befehlen eines Aufrufers ist sehr einfach und kann ohne Brechen von weiterem Code durchgeführt werden.
 - Auch sind **neue Befehle** schnell erstellt (einfach Command realisieren und execute() implementieren) und ins System integriert. Wieder muss dazu kein Code angefasst werden.
 - Weiterhin können Befehle **dynamisch zur Laufzeit** gewechselt werden.
- **Kombination** von Befehlen zu Makrobefehlen.
- Einfache Implementierung von **Rückgängig**- oder **Loggingfunktionalitäten**.

- Durch **Reduzierung der Abhängigkeiten** (mittels Delegation) wird die **Kohäsion** jedes Teilnehmers (Invoker, Command, Receiver) erhöht.
- Vermeidung von **Coderedundanz** und **Inkonsistenz** durch zentrale Befehlsobjekte.
- Befehlsobjekte können wie normale Objekte verwendet werden.
 - Erweiterung/Spezialisierung
 - Speichern, Laden, Versenden
 - Verändern, Filtern, Sortieren

Nachteile

- **Hohe Klassenanzahl.** Da jeder Befehl eine neue Klasse bildet, kann es sehr schnell zu einer unüberschaubaren Vielfalt von konkreten Befehlen kommen. Die Schachtelung der modularen Befehle in Form von Makrobefehlen kann diesen Effekt bedingt verringern.

Anwendung in der Java Standardbibliothek

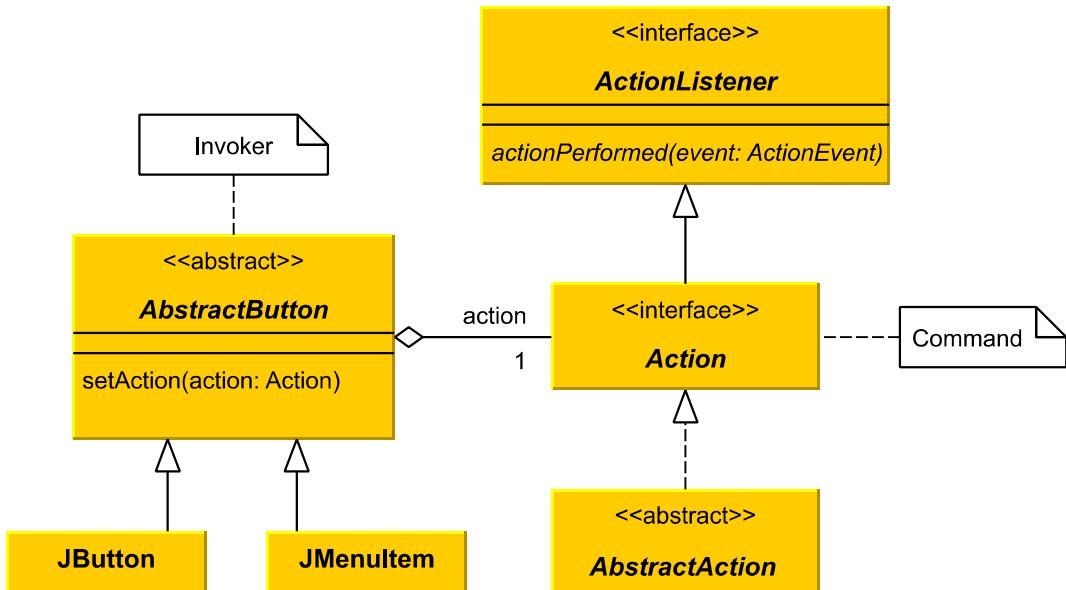
Swing: Action-Klasse (`javax.swing.Action`)

Swing Komponenten nutzen das *Observer* Design Pattern, um andere Klassen über Änderungen zu informieren (beispielsweise beim Drücken eines Buttons werden alle registrierten ActionListener informiert). Diese informierte Klasse weiß genau, was zu tun ist bzw. auf welchen Empfänger nun eine Aktion ausgeführt werden soll. Da auch in diesem Fall Aufrufer (Subjekt, Button) und Empfänger (Observer, ActionListener) voneinander entkoppelt sind, kann auch dies als eine Art Realisierung des Command Patterns betrachtet werden. Zumal auch selbe Observer/Commands für verschiedene Aufrufer/Subjekte genutzt werden können.

Aber Swing Komponenten unterstützen zum Eventhandling neben den Listener noch ein Konzept, das direkt dem Command Pattern entspricht: Action und AbstractAction.

So kann ein JButton und ein JMenuItem mit ein und derselben Action geladen werden. Eigentlich sind Actions nichts anderes als ActionListener (sie erweitern sogar dieses Interface) und werden intern vom JButton genauso gehandhabt. Sie werden ebenfalls in die ActionListenerListe aufgenommen.

Jedoch unterscheiden sich das Action- zum ActionListener-Konzept hinsichtlich folgender Aspekte:



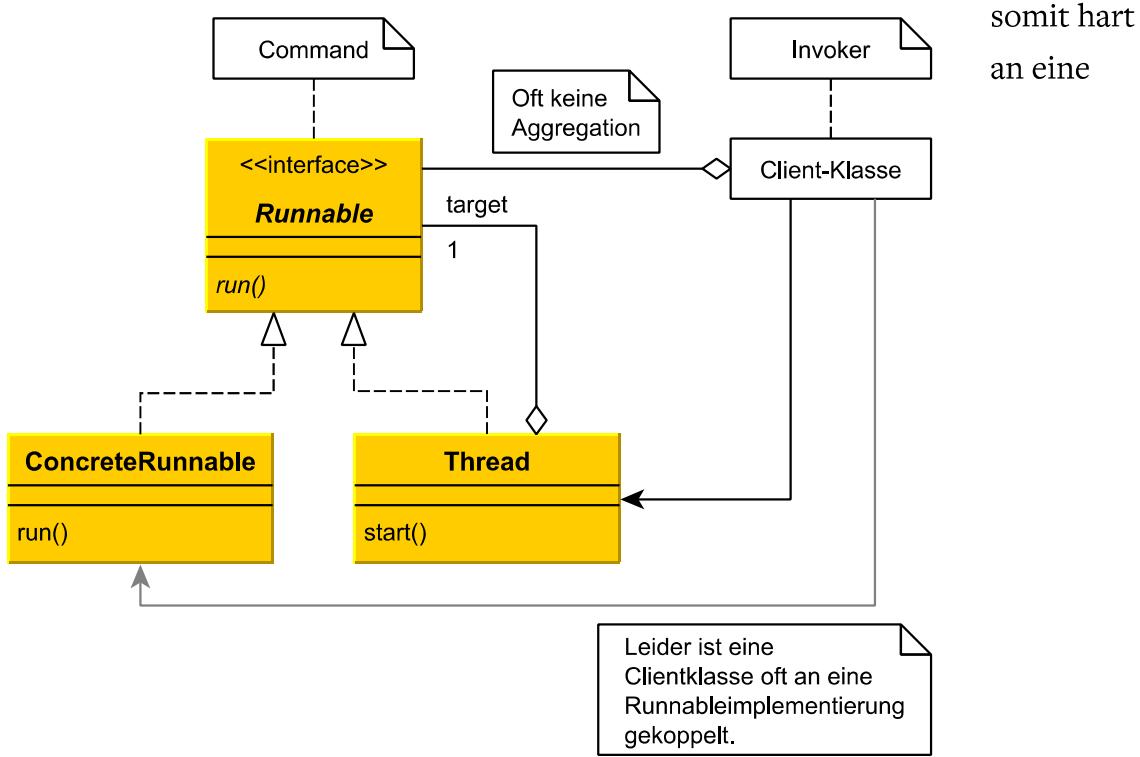
- Eine Komponente (JButton, JMenuItem, JComboBox, JTextField) kann immer nur mit *einer* Action geladen werden.
- Actions bieten die Möglichkeit, den Zustand einer ActionEvent-feuernden Komponente durch Setzen der Action zu überschreiben. Der Zustand eines Buttons umfasst beispielsweise den Anzeigetext, Tooltip, Icons, Hotkey, Enabled- und Selectionstatus. Die gewünschte Konfiguration für diese Zustände können in einem Actionobjekte abgelegt und bequem auf verschiedenen Componenten (JButton, JMenuItem) übertragen werden. Das Codebeispiel ist dem Sun Tutorial "How to Use Actions" entnommen. Weiterführende Informationen sind dort zu finden.

java.lang.Runnable

Es wurde bereits angedeutet, dass das Command Pattern sich sehr gut für Multithreading eignet. Nun handelt es sich beim Interface Runnable um nichts anderes als um ein *Commandinterface*, das zudem direkt in einem eigenen Thread ausgeführt werden kann.

Eine wie auch immer geartete Klasse des Clients fungiert als Invoker. Er stößt den Thread (und damit die Befehlsausführung) an, dem er eine Runnableimplementierung übergeben hat. Bei kluger Entkopplung kennt die Clientklasse, unser Invoker, nur die Runnableschnittstelle. Sie ist von der eigentlichen Tätigkeit der Runnable (bzw. des Threads) und dessen Empfänger entkoppelt.

Leider wird in der Praxis viel zu oft das volle *Command-Potenzial nicht ausgeschöpft*. Das liegt daran, dass die Clientklasse häufig nicht nur den Thread mit der Runnable abschickt (Command anstoßen), sondern auch die konkrete Runnableimplementierung selbst vorher instanziert (Command laden). Da sie



Implementierung gekoppelt ist, kann die Clientklasse nicht dynamisch mit einer neuen Runnable geladen werden.

Ich möchte diese oft zweckmäßige Vorgehensweise nicht verteufeln, sondern nur auf das Potenzial des Runnable-Interfaces hinweisen.

Zur Katalogübersicht

Strategy, Observer, Decorator, Factory Method, Abstract Factory, Singleton, Command, Composite, Facade, State

Literaturverzeichnis, Philipps Blog

<https://www.philippbauermann.de/study/se/design-pattern/command.php>