Entwurfsmuster

Andreas Zeller

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Entwurfsmuster

In diesem Kapitel werden wir uns mit typischen Einsätzen von objektorientiertem Entwurf beschäftigen – den sogenannten *Entwurfsmustern* (*design patterns*).

Der Begriff *Entwurfsmuster* wurde durch den Architekten Christopher Alexander geprägt:

> Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.

Ein Muster ist eine *Schablone,* die in vielen verschiedenen Situationen eingesetzt werden kann.



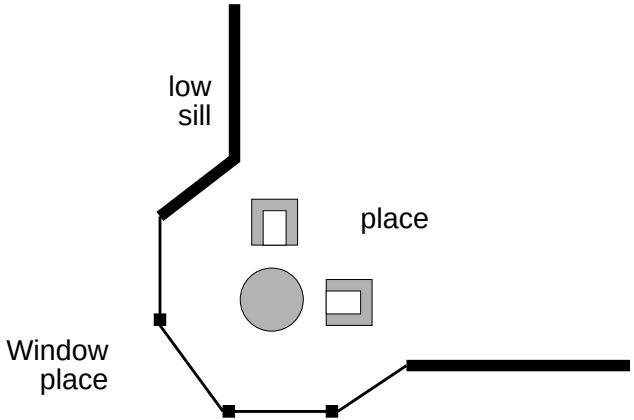




Muster in der Architektur: Window Place

Everybody loves window seats, bay windows, and big windows with low sills and comfortable chairs drawn up to them

In every room where you spend any length of time during the day, make at least one window into a "window place"



Muster im Software-Entwurf

In unserem Fall sind Entwurfsmuster

Beschreibungen von kommunizierenden Objekten und Klassen, die angepasst wurden, um ein allgemeines Entwurfsproblem in einem bestimmten Kontext zu lösen.







Die Elemente eines Musters

Muster sind meistens zu *Katalogen* zusammengefasst: Handbücher, die Muster für zukünftige Wiederverwendung enthalten.

Jedes Muster wird mit wenigstens vier wesentlichen Teilen beschrieben:

- Name
- Problem
- Lösung
- Folgen







Die Elemente eines Musters (2)

Der Name des Musters wird benutzt, um das Entwurfsproblem, seine Lösung und seine Folgen in einem oder zwei Worten zu beschreiben.

- ermöglicht es, auf einem höheren Abstraktionsniveau zu entwerfen,
- erlaubt es uns, es unter diesem Namen in der Dokumentation zu verwenden,
- ermöglicht es uns, uns darüber zu unterhalten.

Das Problem beschreibt, wann ein Muster eingesetzt wird.

- beschreibt das Problem und dessen Kontext
- kann bestimmte Entwurfsprobleme beschreiben
- kann bestimmte *Einsatzbedingungen* enthalten

Die Elemente eines Musters (3)

Die Lösung beschreibt die Teile, aus denen der Entwurf besteht, ihre Beziehungen, Zuständigkeiten und ihre Zusammenarbeit – kurz, die *Struktur* und die *Teilnehmer*:

- nicht die Beschreibung eines bestimmten konkreten Entwurfs oder einer bestimmten Implementierung,
- aber eine *abstrakte Beschreibung* eines Entwurfsproblems und wie ein allgemeines Zusammenspiel von Elementen das Problem löst.

Die Folgen sind die Ergebnisse sowie Vor- und Nachteile der Anwendung des Musters:

- *Abwägungen bezüglich Ressourcenbedarf* (Speicherplatz, Laufzeit)
- aber auch die Einflüsse auf *Flexibilität*, *Erweiterbarkeit* und *Portierbarkeit.*

Fallstudie: Die Textverarbeitung Lexi

Als Fallstudie betrachten wie den Entwurf einer “What you see is what you get” (“WYSIWYG”) Textverarbeitung namens *Lexi.*

Lexi kann Texte und Bilder frei mischen in einer Vielzahl von möglichen Anordnungen.

Wir betrachten, wie Entwurfsmuster die wesentlichen Lösungen zu Entwurfsproblemen in *Lexi* und ähnlichen Anwendungen beitragen.







lexi

File Edit Style Symbol

Align left
Center
Align right
✓Justify

✓Roman
Boldface
Italic
Typewriter
Sans serif

✓Gnu
Gnu
Gnu
Gnu
Gnu
Gnu

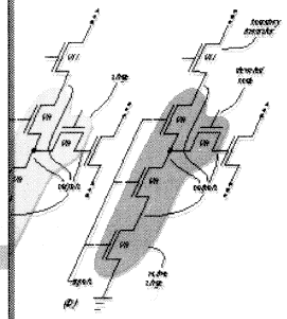


Figure 4: A gratuitous idraw drawing

the internal representation of the TextView. The draw operation (which is not shown) simply calls draw on the TEBox.

The code that builds a TextView is similar to the original draw code, except that instead of calling functions to draw the characters, we build objects that will draw themselves whenever necessary. Using objects solves the redraw problem because only those objects that lie within the damaged region will get draw calls. The programmer does not have to write the code that decides what objects to redraw – that code is in the toolkit (in this example, in the implementation of the Box draw operation). Indeed, the glyph-based implementation of TextView is even simpler than the original code because the programmer need only declare *what* objects he wants – he does not need to specify *how* the objects should interact.

?.? Multiple fonts

Because we built TextView with glyphs, we can easily extend it to add functionality that might otherwise be difficult to implement. For example, Figure 4 shows a screen dump of a version of TextView that displays EUC-encoded Japanese text. Adding this feature to a text view such as the Athena Text Widget would require a complete rewrite. Here we only add two lines of code. Figure 5 shows the change.

Character glyphs take an optional second constructor parameter that specifies the font to use when drawing. For ASCII-encoded text we create Characters that use the 8-bit ASCII-encoded "a14" font; for JIS-encoded text (kanjii and kana characters) we create Characte that use the 16-bit JIS-encoded "k14" font.

?.? Mixing text and graphics

We can put any glyph inside a composite glyph; th it is straightforward to extend TextView to displ embedded graphics. Figure 6 shows a screen dump a view that makes the whitespace characters in a fi visible by drawing graphical representations of spac newlines, and formfeeds. Figure 7 shows the modifi code that builds the view.

A Stencil is a glyph that displays a bitmap, an HRu draws a horizontal line, and VGlue represents vertic blank space. The constructor parameters for Rule a

```
while ((c = getc(file)) != EOF) {
  if (c == '\n') {
      line = new LRBox();
+ } else if (!isascii(c)) {
+     line->append(
        new Character(
          tojis(c, getc(file)), k14
        )
      );
  } else {
      line->append(
        new Character(c, a14)
      );
  }
}
```

Figure 5: Modified TextView that displays Japanese tex

Herausforderungen

Dokument-Struktur. Wie wird das Dokument intern gespeichert?

Formatierung. Wie ordnet *Lexi* Text und Graphiken in Zeilen und Polygone an?

Unterstützung mehrerer Bedienoberflächen. *Lexi* sollte so weit wie möglich unabhängig von bestimmten Fenstersystemen sein.

Benutzer-Aktionen. Es sollte ein einheitliches Verfahren geben, um auf *Lexi*s Funktionalität zuzugreifen und um Aktionen zurückzunehmen.

Jedes dieser Entwurfsprobleme (und seine Lösung) wird durch ein oder mehrere Entwurfsmuster veranschaulicht.

Struktur darstellen – das Composite*-Muster*

Ein *Dokument* ist eine Anordnung grundlegender graphischer Elemente wie Zeichen, Linien, Polygone und anderer Figuren.

Diese sind in *Strukturen* zusammengefasst—Zeilen, Spalten, Abbildungen, und andere Unterstrukturen.

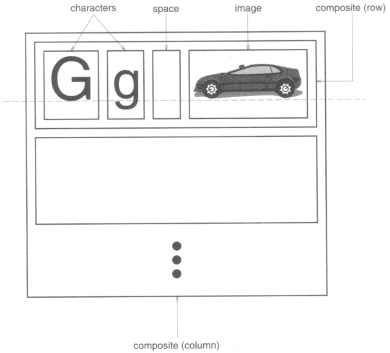
Solche hierarchisch strukturierte Information wird gewöhnlich durch *rekursive Komposition* dargestellt – aus einfachen Elementen (*composite*) werden immer komplexere Elemente zusammengesetzt.



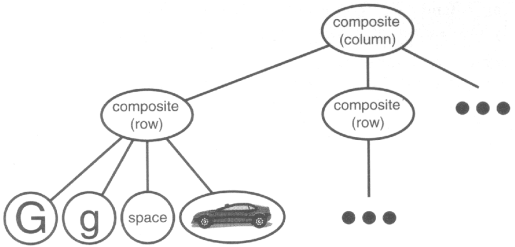




Elemente im Dokument

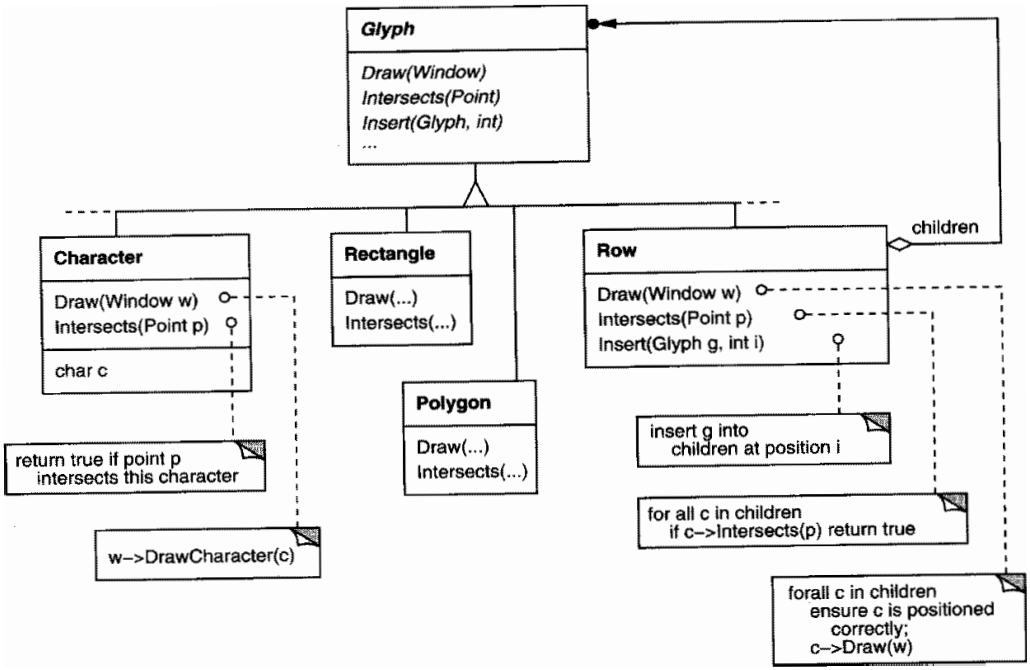


Für jedes wichtige Element gibt es ein individuelles Objekt.



Glyphen

Wir definieren eine abstrakte Oberklasse *Glyphe* (engl. *glyph*) für alle Objekte, die in einem Dokument auftreten können.



Glyphen (2)

Jede Glyphe

- weiß, wie sie sich zeichnen kann (mittels der `Draw()`-Methode). Diese abstrakte Methode ist in konkreten Unterklassen von `Glyph` definiert.
- weiß, wieviel Platz sie einnimmt (wie in der `Intersects()`-Methode).
- kennt ihre Kinder (*children*) und ihre Mutter (*parent*) (wie in der `Insert()`-Methode).

Die *Glyph*-Klassenhierarchie ist eine Ausprägung des *Composite*-Musters.

Das Composite-*Muster*

Problem Benutze das Composite-Muster wenn

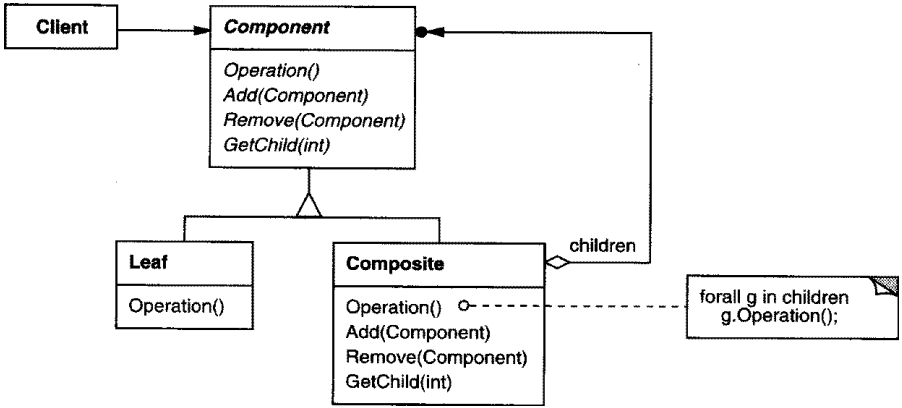
- Du *Teil-ganzes*-Hierarchien von Objekten darstellen willst
- die Anwendung Unterschiede zwischen zusammengesetzten und einfachen Objekten ignorieren soll







Struktur


Client
Component
Operation()
Add(Component)
Remove(Component)
GetChild(int)
children
Leaf
Operation()
Composite
Operation()
Add(Component)
Remove(Component)
GetChild(int)
forall g in children
g.Operation();

Teilnehmer

- **Component** (Komponente: Glyph)
  - definiert die Schnittstelle für alle Objekte (zusammengesetzt und einfach)
  - implementiert Vorgabe-Verhalten für die gemeinsame Schnittstelle (sofern anwendbar)
  - definiert die Schnittstelle zum Zugriff und Verwalten von Unterkomponenten (Kindern)
- **Leaf** (Blatt: Rechteck, Zeile, Text)
  - stellt elementare Objekte dar; ein Blatt hat keine Kinder
  - definiert gemeinsames Verhalten elementarer Objekte







Teilnehmer (2)

- **Composite** (Zusammengesetztes Element: Picture, Column)
  - definiert gemeinsames Verhalten zusammengesetzter Objekte (mit Kindern)
  - speichert Unterkomponenten (Kinder)
  - implementiert die Methoden zum Kindzugriff in der Schnittstelle von *Component*
- **Client** (Benutzer)
  - verwaltet Objekte mittels der *Component*-Schnittstelle.







Folgen

Das Composite-Muster

- definiert Klassenhierarchien aus elementaren Objekten und zusammengesetzten Objekten
- vereinfacht den Benutzer: der Benutzer kann zusammengesetzte wie elementare Objekte einheitlich behandeln; er muss nicht (und sollte nicht) wissen, ob er mit einem elementaren oder zusammengesetzten Objekt umgeht







Folgen (2)

Das Composite-Muster

- vereinfacht das Hinzufügen neuer Element-Arten
- kann den Entwurf zu sehr *verallgemeinern:* soll ein bestimmtes zusammengesetztes Element nur eine feste Anzahl von Kindern haben, oder nur bestimmte Kinder, so kann dies erst zur Laufzeit (statt zur Übersetzungszeit) überprüft werden. ⇐ *Dies ist ein Nachteil!*

Andere bekannte Einsatzgebiete: Ausdrücke, Kommandofolgen.

Algorithmen einkapseln – das Strategy-*Muster*

Lexi muss Text in Zeilen umbrechen und Zeilen zu Spalten zusammenfassen – je nachdem, wie der Benutzer es möchte. Dies ist die Aufgabe eines *Formatieralgorithmus'*.

Lexi soll mehrere Formatieralgorithmen unterstützen, etwa

- einen schnellen ungenauen („quick-and-dirty“) für die WYSIWYG-Anzeige und
- einen langsamen, genauen für den Textsatz beim Drucken

Gemäß der Separation der Interessen soll der Formatieralgorithmus unabhängig von der Dokumentstruktur sein.

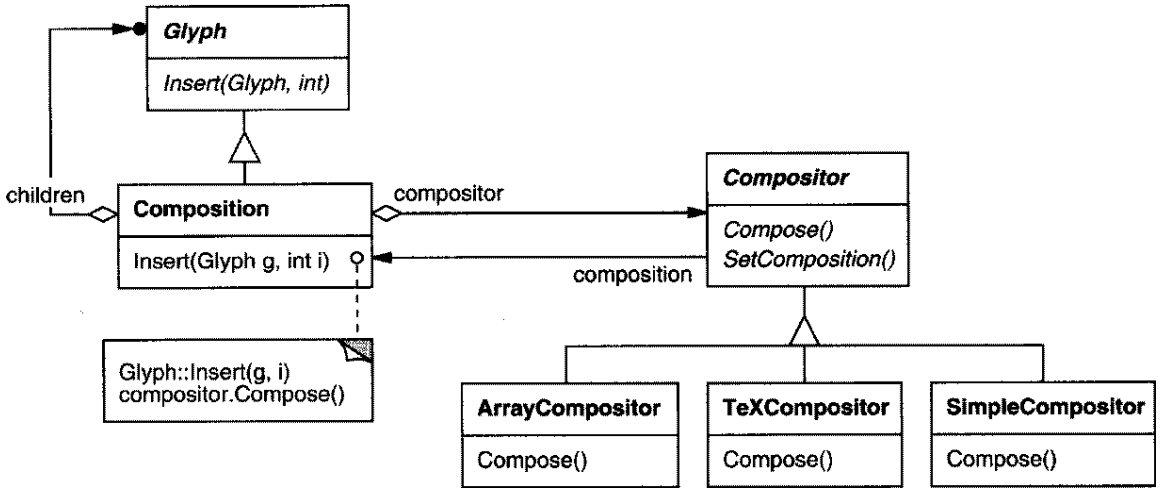






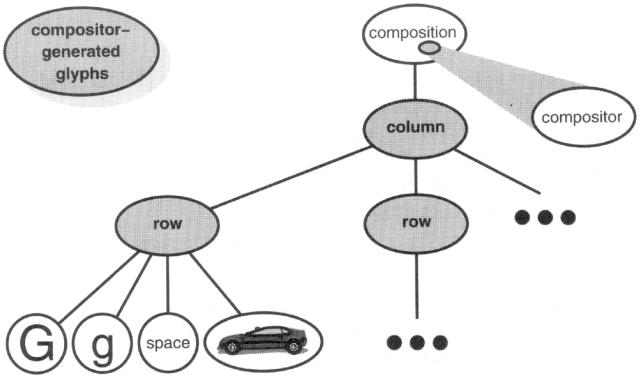
Formatieralgorithmen

Wir definieren also eine separate Klassenhierarchie für Objekte, die bestimmte Formatieralgorithmen *einkapseln*. Wurzel der Hierarchie ist eine abstrakte Klasse *Compositor* mit einer allgemeinen Schnittstelle; jede Unterklasse realisiert einen bestimmten Formatieralgorithmus.



Formatieralgorithmen (2)

Jeder Kompositor wandert durch die Dokumentstruktur und fügt ggf. neue (zusammengesetzte) Glyphen ein:



Dies ist eine Ausprägung des *Strategy*-Musters.

Das Strategy-*Muster*

Problem Benutze das Strategy-Muster, wenn

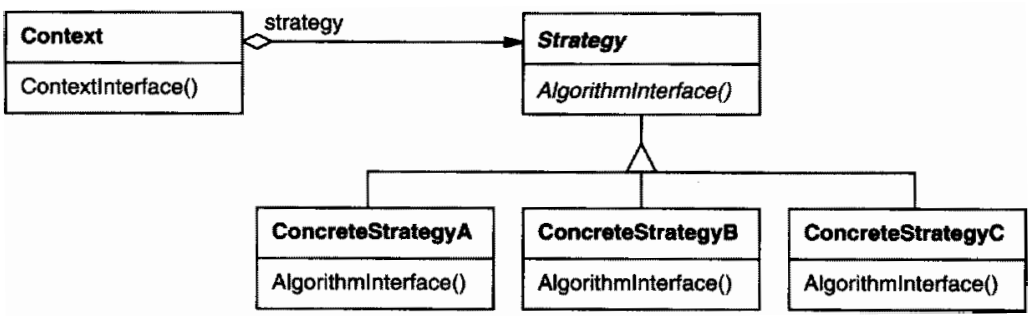
- zahlreiche zusammenhängende Klassen sich nur im Verhalten unterscheiden
- verschiedene Varianten eines Algorithmus' benötigt werden
- ein Algorithmus Daten benutzt, die der Benutzer nicht kennen soll







Struktur


Context
ContextInterface()
strategy
Strategy
AlgorithmInterface()
ConcreteStrategyA
AlgorithmInterface()
ConcreteStrategyB
AlgorithmInterface()
ConcreteStrategyC
AlgorithmInterface()

Teilnehmer

- **Strategy** (Compositor)
  - definiert eine gemeinsame Schnittstelle aller unterstützten Algorithmen
- **ConcreteStrategy** (SimpleCompositor, TeXCompositor, ArrayCompositor)
  - implementiert den Algorithmus gemäß der Strategy-Schnittstelle
- **Context** (Composition)
  - wird mit einem ConcreteStrategy-Objekt konfiguriert
  - referenziert ein Strategy-Objekt
  - kann eine Schnittstelle definieren, über die Daten für Strategy verfügbar gemacht werden.

Folgen

Das Strategy-Muster

- macht Bedingungs-Anweisungen im Benutzer-Code unnötig (if simple-composition then . . . else if tex-composition . . .)
- hilft, die gemeinsame Funktionalität der Algorithmen herauszufaktorisieren
- ermöglicht es dem Benutzer, Strategien auszuwählen. . .
- . . . aber belastet den Benutzer auch mit der Strategie-Wahl!
- kann in einem *Kommunikations-Overhead* enden: Information muss bereitgestellt werden, auch wenn die ausgewählte Strategie sie gar nicht benutzt

Weitere Einsatzgebiete: Code-Optimierung, Speicher-Allozierung, Routing-Algorithmen







Mehrfache Variation – das Bridge-*Muster*

Lexi läuft auf einem Fenstersystem und hat (mindestens) drei verschiedene Fenster-Typen:

- Ein *ApplicationWindow* zeigt das Dokument an
- Ein *IconWindow* ist zeitweise unsichtbar („ikonifiziert“)
- Ein *DialogWindow* ist ein temporäres Fenster, das auf einem *Besitzer-Fenster* liegt.

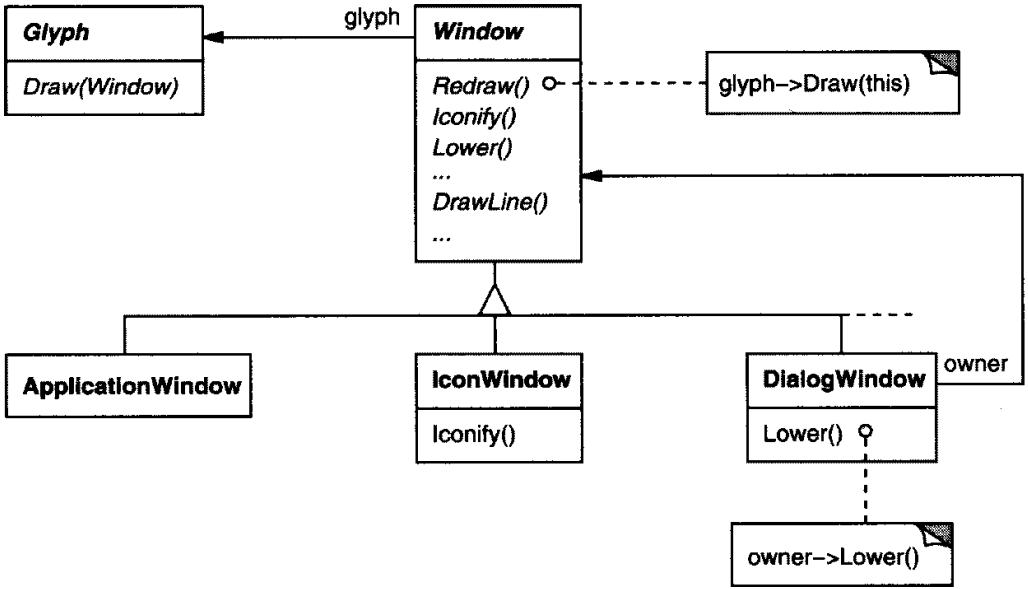






Variation durch Klassen

Diese Fenstertypen können in einer *Window*-Klassenhierarchie angeordnet werden:



Andere Variationsmöglichkeiten

Diese Fenstertypen sind nicht die einzige mögliche Variation.

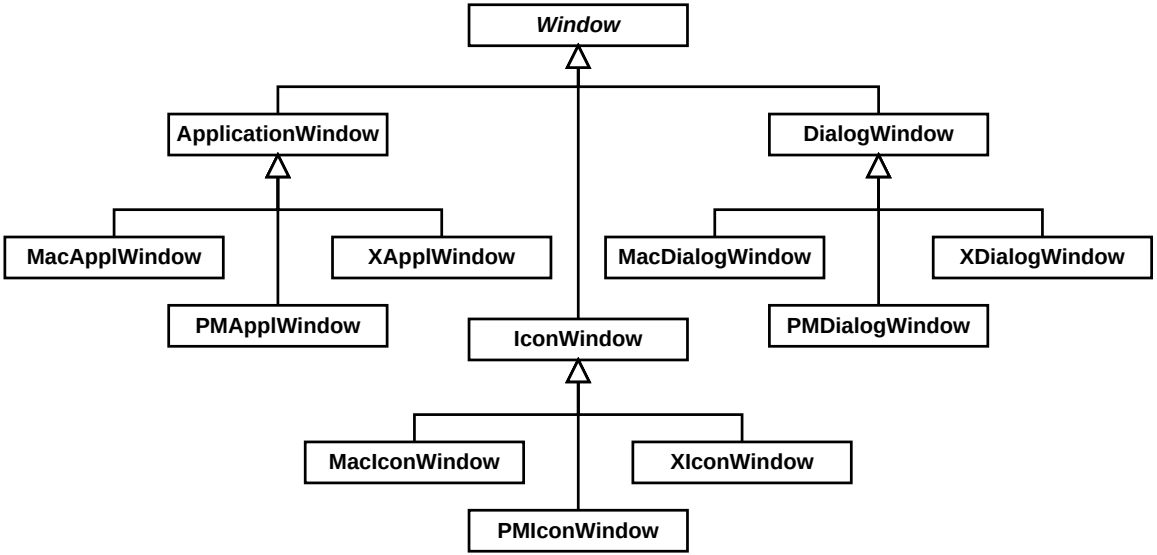
Lexi könnte auf mehreren verschiedenen *Fenstersystemen* laufen – etwa Windows Presentation Manager, Macintosh, und das X Window System.

Es gibt drei Alternativen, um diese zusätzliche Variation zu modellieren:

- Neue Unterklassen
- Mehrfach-Vererbung
- Dynamische Kopplung

Neue Unterklassen

Alternative 1: Neue Unterklassen. Führe neue Unterklassen ein, die die Variation einkapseln, wie *PMIconWindow, MacIconWindow, XIconWindow*.



Probleme

- Gemeinsamer Code für alle Fenstersysteme ist nicht in gemeinsamer Oberklasse
- Mit n Fenstertypen und m Fenstersystemen erhalten wir $n \times m$ neue Klassen

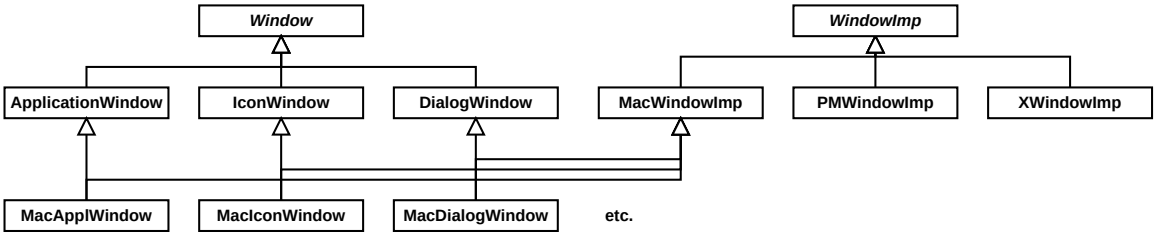






Mehrfach-Vererbung

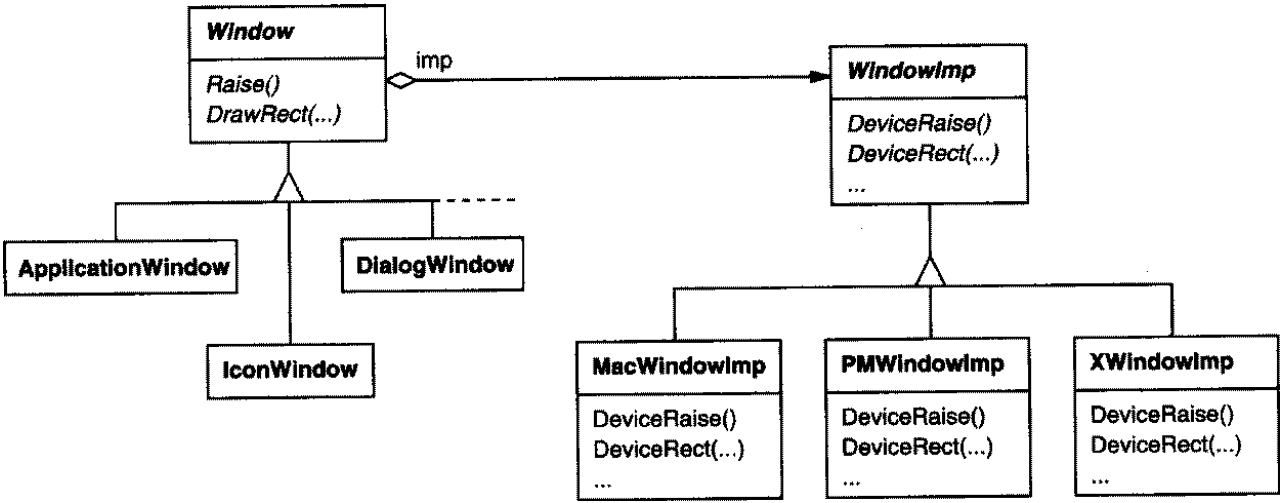
Alternative 2: Mehrfach-Vererbung. Erzeuge zwei Hierarchien für den Fenstertyp (*Window*) und das Fenstersystem (*WindowImp*) und benutze mehrfache Vererbung, um neue zusammengesetzte Klassen zu erzeugen: Jede Klasse erbt einen Fenstertyp und eine Implementierung.



- Wir bekommen nach wie vor $n \times m$ neue Klassen.

Dynamische Kopplung

Alternative 3: Dynamische Kopplung. Wir koppeln Fenstertyp und Fenstersystem dynamisch mittels eines *implementation*-Verweises; jedes *Window*-Objekt verweist auf eine bestimmte *WindowImp*-Implementierung.



Dies ist eine Ausprägung des *Bridge*-Musters.

Das Bridge-*Muster*

Problem Benutze das Bridge-Muster, wenn

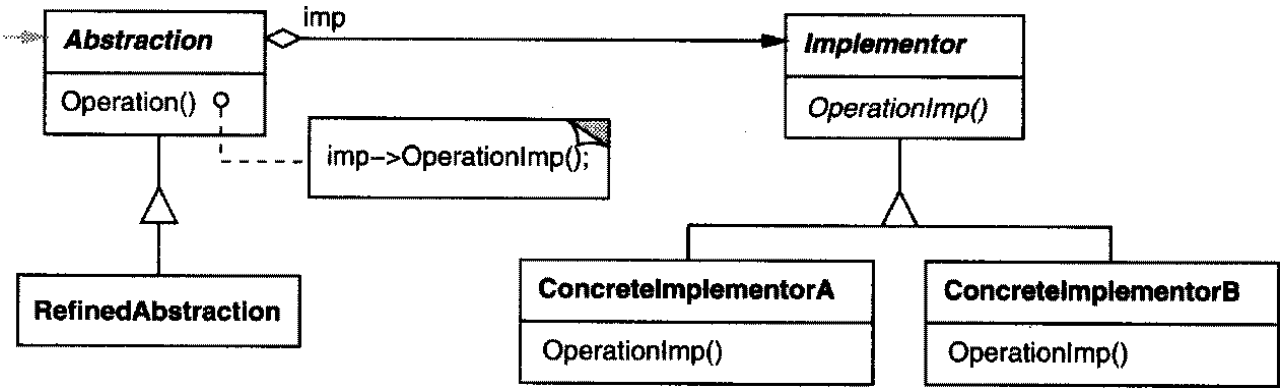
- Du eine Abstraktion von ihrer Implementierung entkoppeln willst (etwa wenn die Implementierung zur Laufzeit ausgewählt wird)
- sowohl die Abstraction als auch die Implementierung erweiterbar sein sollen.







Struktur


Abstraction
Operation()
imp
imp->OperationImp();
Implementor
OperationImp()
RefinedAbstraction
ConcreteImplementorA
OperationImp()
ConcreteImplementorB
OperationImp()

Teilnehmer

- **Abstraction** (Window)
  - definiert die Schnittstelle der Abstraktion
  - unterhält einen Verweis zu einem Objekt vom Typ Implementor
- **RefinedAbstraction** (IconWindow)
  - erweitert die Schnittstelle von Abstraction
- **Implementor** (WindowImp)
  - definiert die Schnittstelle für Implementierungs-Klassen (gewöhnlich primitive Operationen; die höheren Operationen sind in Abstraction realisiert)
- **ConcreteImplementor** (XWindowImp, PMWindowImp)
  - realisiert die Implementor-Schnittstelle und definiert ihre konkrete Implementierung

Folgen

Das Bridge-Muster

- entkoppelt Schnittstelle und Implementierung:
  - Eine Abstraktion ist nicht fest an eine Schnittstelle gebunden
  - Die Implementierung kann zur Laufzeit konfiguriert (und sogar geändert) werden
  - Wir unterstützen eine bessere Gesamt-Struktur des Systems
- verbessert die Erweiterbarkeit
- versteckt Implementierungs-Details vor dem Benutzer

Weitere Einsatzgebiete: Darstellung von Bildern (z.B. JPEG/GIF/TIFF), String-Handles

Benutzer-Aktionen – das Command-*Muster*

*Lexi*s Funktionalität ist auf zahlreichen Wegen verfügbar: Man kann die WYSIWIG-Darstellung manipulieren (Text eingeben und ändern, Cursor bewegen, Text auswählen) und man kann weitere Aktionen über Menüs, Schaltflächen und Abkürzungs-Tastendrücke auswählen.

Wir möchten eine bestimmte Aktion nicht mit einer bestimmten Benutzerschnittstelle koppeln, weil

- es mehrere Benutzungsschnittstellen für eine Aktion geben kann (man kann die Seite mit einer Schaltfläche, einem Menüeintrag oder einem Tastendruck wechseln)
- wir womöglich in Zukunft die Schnittstelle ändern wollen.







Benutzer-Aktionen (2)

Um die Sache weiter zu verkomplizieren, möchten wir auch Aktionen *rückgängig machen* können (*undo*) und rückgängig gemachte Aktionen *wiederholen* können (*redo*).

Wir möchten *mehrere Aktionen* rückgängig machen können, und wir möchten Makros (Kommandofolgen) aufnehmen und abspielen können.

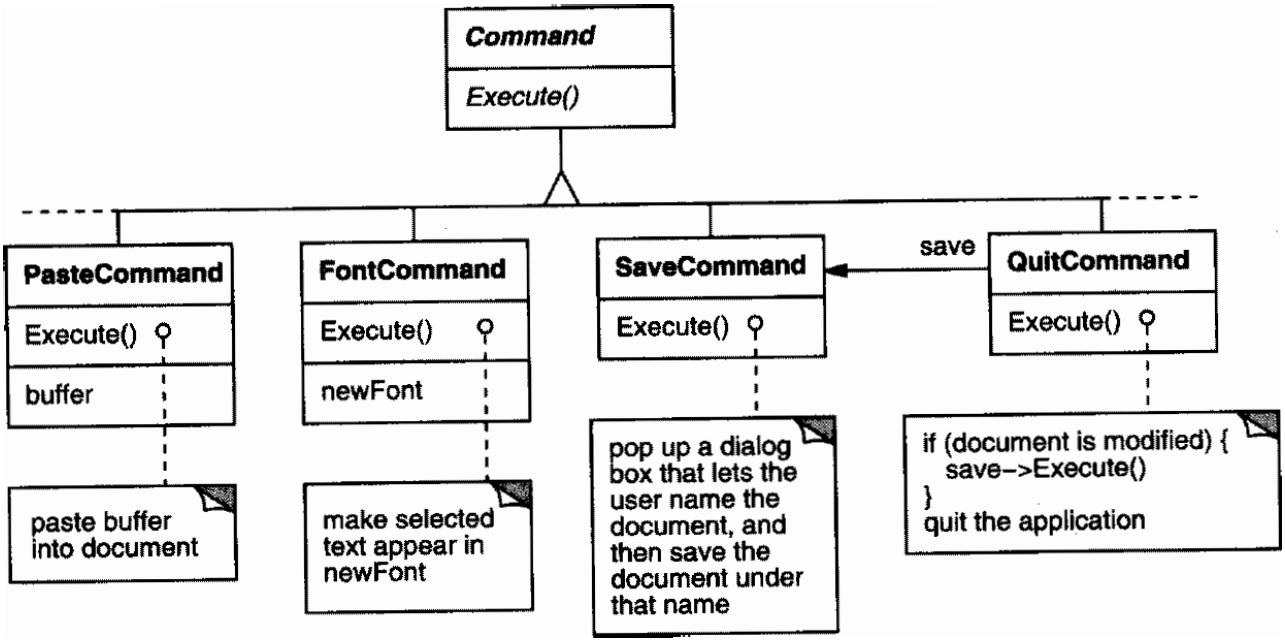






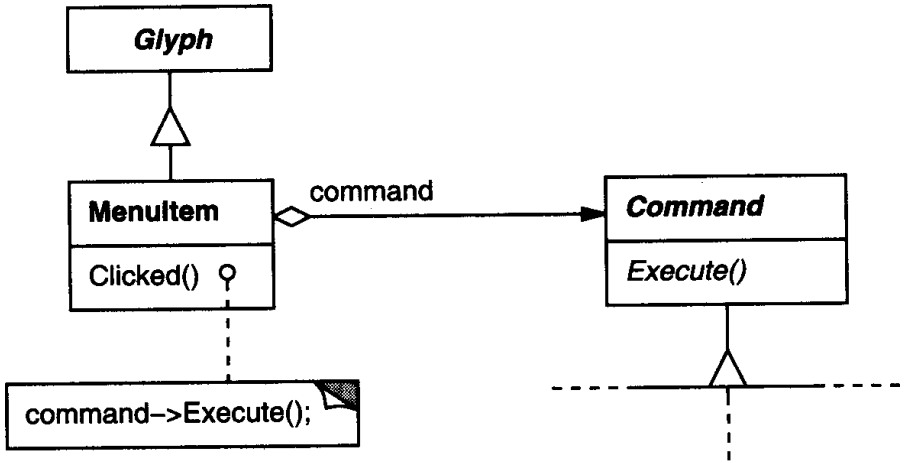
Benutzer-Aktionen (3)

Wir definieren deshalb eine *Command*-Klassenhierarchie, die Benutzer-Aktionen einkapselt.



Benutzer-Aktionen (4)

Spezifische Glyphen können mit Kommandos verknüpft werden; bei Aktivierung der Glyphen werden die Kommandos ausgeführt.



Dies ist eine Ausprägung des *Command*-Musters.

Das Command-*Muster*

Problem Benutze das Command-Muster wenn Du

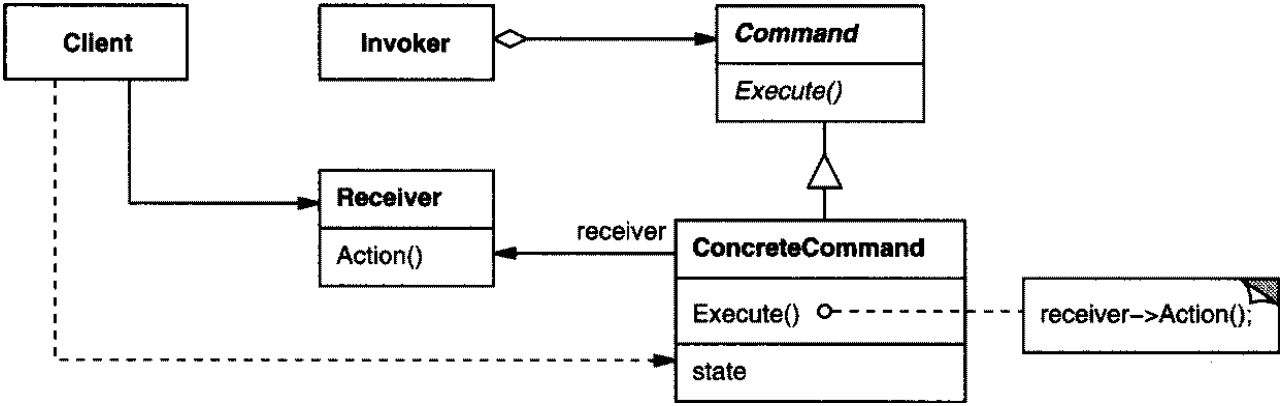
- Objekte parametrisieren willst mit der auszuführenden Aktion
- Kommandos zu unterschiedlichen Zeiten auslösen, einreihen und ausführen möchtest
- die Rücknahme von Kommandos unterstützen möchtest (siehe unten)
- Änderungen protokollieren möchtest, so dass Kommandos nach einem System-Absturz wiederholt werden können.







Struktur


Client
Invoker
Command
Execute()
Receiver
Action()
receiver
ConcreteCommand
Execute()
state
receiver->Action();

Teilnehmer

- **Command** (Kommando)
  - definiert eine Schnittstelle, um eine Aktion auszuführen
- **ConcreteCommand** (PasteCommand, OpenCommand)
  - definiert eine Verknüpfung zwischen einem Empfänger-Objekt und einer Aktion
  - implementiert `Execute()`, indem es die passenden Methoden auf Receiver aufruft
- **Client** (Benutzer; Application)
  - erzeugt ein ConcreteCommand-Objekt und setzt seinen Empfänger







Teilnehmer (2)

- **Invoker** (Aufrufer; MenuItem)
  - fordert das Kommando auf, seine Aktion auszuführen
- **Receiver** (Empfänger; Document, Application)
  - weiß, wie die Methoden ausgeführt werden können, die mit einer Aktion verbunden sind. Jede Klasse kann Empfänger sein.







Folgen

Das Command-Muster

- entkoppelt das Objekt, das die Aktion auslöst, von dem Objekt, das weiß, wie die Aktion ausgeführt werden kann
- realisiert Kommando als *first-class*-Objekte, die wie jedes andere Objekt gehandhabt und erweitert werden können
- ermöglicht es, Kommandos aus anderen Kommandos zusammenzusetzen (siehe unten)
- macht es leicht, neue Kommandos hinzuzufügen, da existierende Klassen nicht geändert werden müssen.

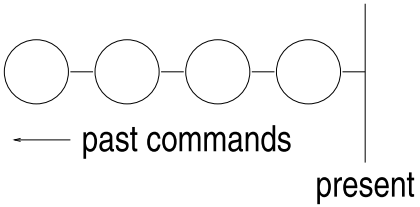






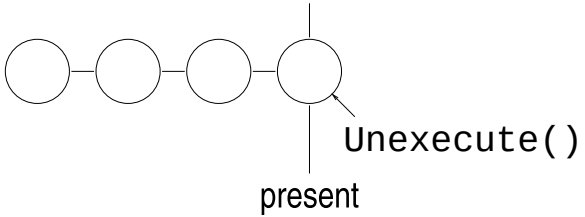
Kommandos rückgängig machen

Mittels einer *Kommando-Historie* kann man leicht rücknehmbare Kommandos gestalten. Eine Kommando-Historie sieht so aus:



Kommandos rückgängig machen (2)

Um das letzte Kommando rückgängig zu machen, rufen wir `Unexecute()` auf dem letzten Kommando auf. Das heißt, daß jedes Kommando genug Zustandsinformationen halten muß, um sich selbst rückgängig zu machen.



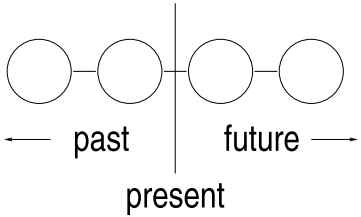






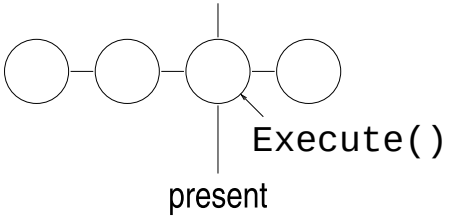
Kommandos rückgängig machen (3)

Nach dem Rücknehmen bewegen wir die „Gegenwarts-Linie“ ein Kommando nach links. Nimmt der Benutzer ein weiteres Kommando zurück, wird das vorletzte Kommando zurückgenommen und wir enden in diesem Zustand:



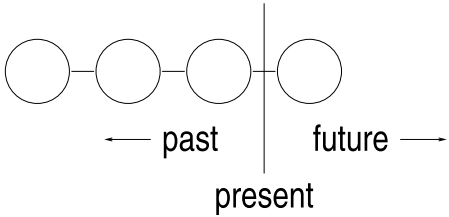
Kommandos rückgängig machen (4)

Um ein Kommando zu wiederholen, rufen wir einfach `Execute()` des gegenwärtigen Kommandos auf . . .



Kommandos rückgängig machen (5)

. . . und setzen die „Gegenwarts-Linie“ um ein Kommando nach vorne, so dass das nächste Wiederholen die `Execute()`-Methode des nächsten Kommandos aufruft.



Auf diese Weise kann der Benutzer vorwärts und rückwärts in der Zeit gehen – je nachdem, wie weit er gehen muss, um Fehler zu korrigieren.

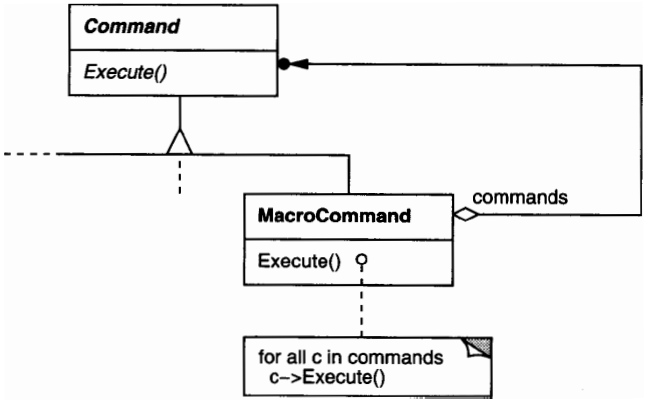






Makros

Zum Abschluss zeigen wir noch, wie man Makros (Kommandofolgen) realisiert. Wir setzen das *Composite*-Muster ein, um eine Klasse *MacroCommand* zu realisieren, die mehrere Kommandos enthält und nacheinander ausführt:



Wenn wir nun noch dem MacroCommand eine `Unexecute()`-Methode hinzufügen, kann man ein Makro wie jedes andere Kommando zurücknehmen.

Zusammenfassung

In *Lexi* haben wir die folgenden Entwurfs-Muster kennengelernt:

- *Composite* zur Darstellung der internen Dokument-Struktur
- *Strategy* zur Unterstützung verschiedener Formatierungs-Algorithmen
- *Bridge* zur Unterstützung mehrerer Fenstersysteme
- *Command* zur Rücknahme und Makrobildung von Kommandos

Keins dieser Entwurfsmuster ist beschränkt auf ein bestimmtes Gebiet; sie reichen auch nicht aus, um alle anfallenden Entwurfsprobleme zu lösen.







Zusammenfassung (2)

Zusammenfassend bieten Entwurfsmuster:

Ein gemeinsames Entwurfs-Vokabular. Entwurfsmuster bieten ein gemeinsames Vokabular für Software-Entwerfer zum Kommunizieren, Dokumentieren und um Entwurfs-Alternativen auszutauschen.

Dokumentation und Lernhilfe. Die meisten großen objekt-orientierten Systeme benutzen Entwurfsmuster. Entwurfsmuster helfen, diese Systeme zu verstehen.

Eine Ergänzung zu bestehenden Methoden. Entwurfsmuster fassen die Erfahrung von Experten zusammen – unabhängig von der Entwurfsmethode.

"The best designs will use many design patterns that dovetail and intertwine to produce a greater whole."