Gliederung

Motivation

- (lat.) das Kompositum (die ...ta, auch die ... ten) zusammengesetztes Wort, Zusammensetzung
- komplexe zusammengesetzte Strukturen spielen hier eine wichtige Rolle
- die OOP versucht vielfach zusammengesetzte Sachverhalte in ihre Einzelteile zu zerlegen (Dekomposition, nach Anforderungsanalyse)
- dient einer möglichst „realen“ Abbildung der Zusammenhänge

Motivation

- komplexe Strukturen werden meistens aus einfachen Teilen zusammengesetzt
- Beispiel Grafikeditor
- komplexe Figuren bestehen aus einfachen Teilen
- Teile befinden sich in Behältern
- Behälter enthalten wiederum Behälter (Rekursion)

Problem:
- Zugriff auf die einzelnen Objekte
- unterschiedliche Handhabung der Objekte

(Klienten müssen Objekte unterscheiden können)

Motivation

Lösung:
- Objekthandhabung der rekursiven Struktur überlassen
- Unterscheidung der Objekte den „Klienten“ abnehmen
- tatsächlichen Zugriff auf die Objekte für den Klienten transparent gestalten

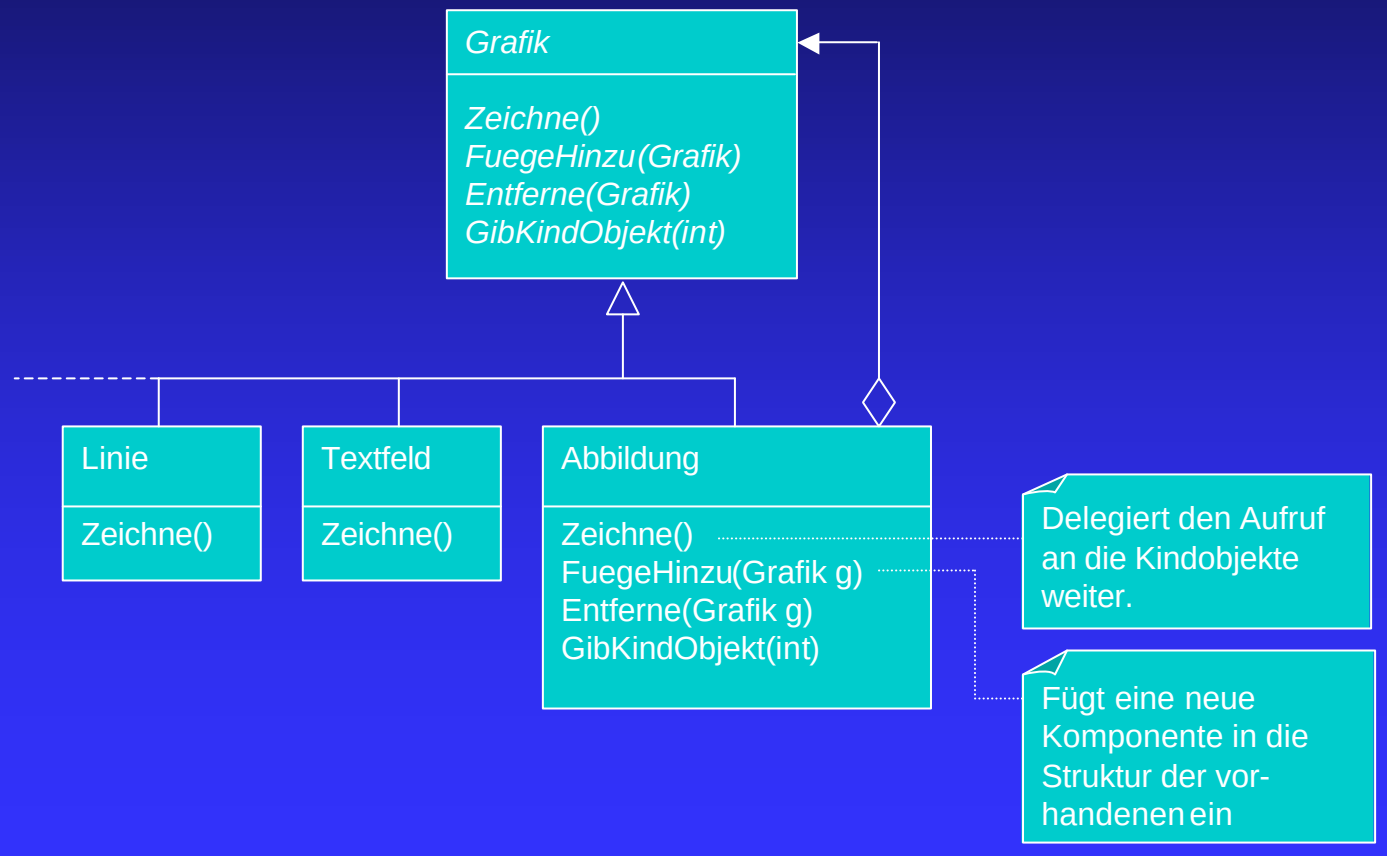
Diese Features können bei einem entsprechenden Entwurf durch das Kompositum realisiert werden.

Grundidee und Realisierung

- Grundidee trivial:
- Entwurf einer abstrakten Klasse für alle Objekte (sowohl primitive Teilobjekte als auch Behälter)
- Deklaration spezieller Methoden der primitiven als auch der zusammengesetzten Objekte in dieser einen Klasse
- Die konkreten Klassen implementieren dann die Methoden je nach ihren Anforderungen

Grundidee und Realisierung

Beispiel Grafikeditor:



Grundidee und Realisierung

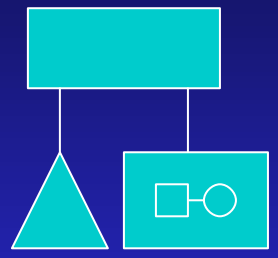
- Klassenschema enthält sowohl primitive Objekte (Linie, Textfeld) als auch komplexe Strukturen (Abbildung → aggregiert Grafiken)
- alle Klassen sind von Grafik abgeleitet
- Grafik „definiert“ Standardfunktionalität der Objekte
- Linie und Textfeld implementieren lediglich *Zeichne(),* nicht aber Kindverwaltungsmethoden
- Abbildung ist eine zusammengesetzte Struktur (spiegelt hier das Kompositum wieder)

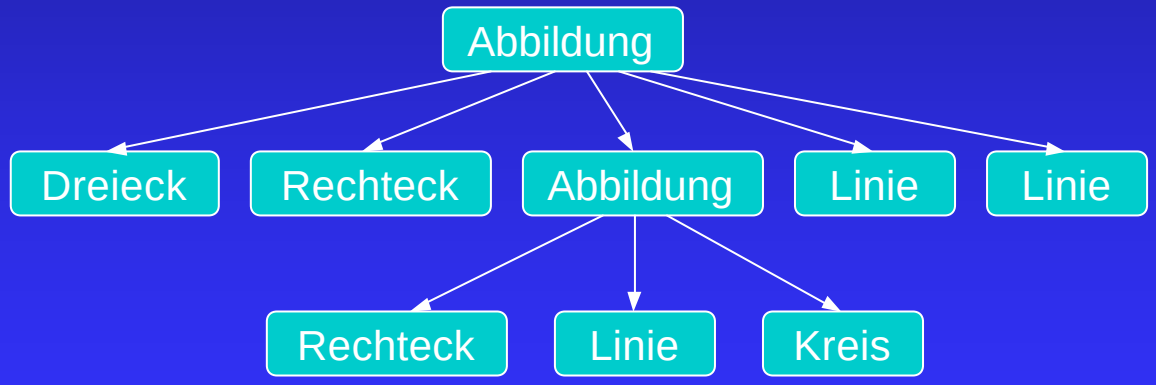
Grundidee und Realisierung

- Abbildung ist ein Aggregat (hält Referenzen auf andere *Grafik*-Objekte)
- ein Aufruf von *Zeichne()* wird an die Kindobjekte von Abbildung delegiert
- die Struktur verarbeitet alle Kindobjekte betreffenden Methodenaufrufe durch Delegation an die Objekte selbst
- die Aggregation von *Grafik* bildet eine Rekursion
- durch Anordnung von primitiven und komplexen Strukturen erhält man eine Baumstruktur

Grundidee und Realisierung

Beispiel:





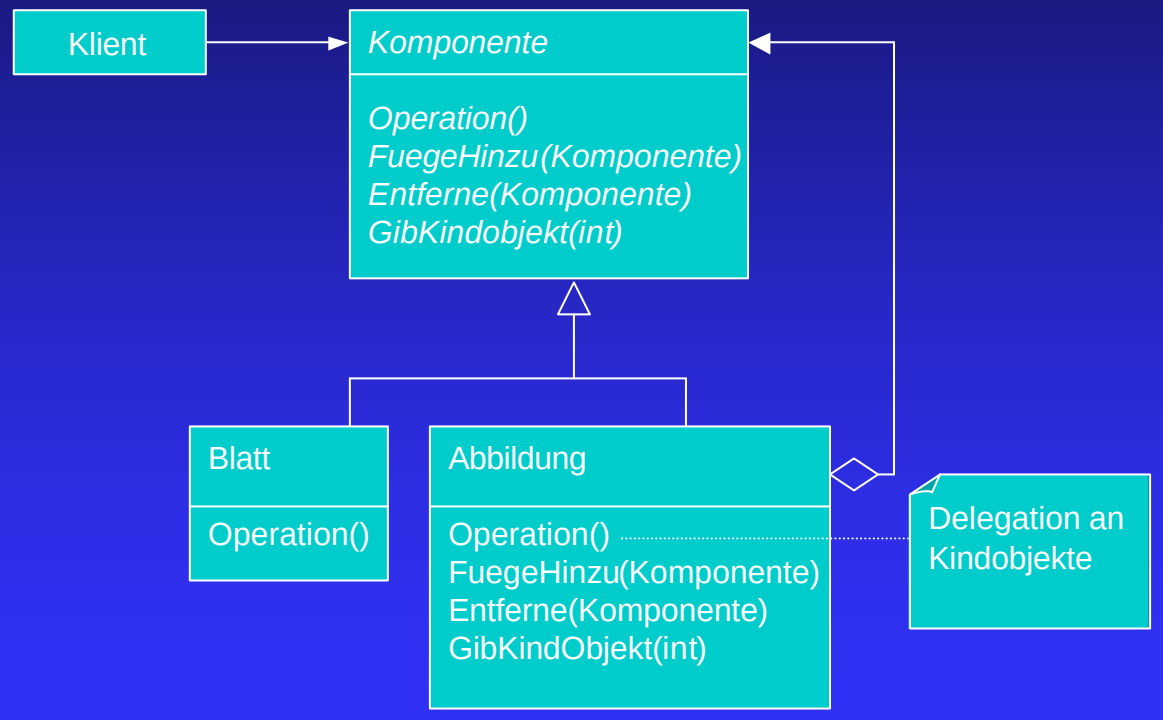
Grundidee und Realisierung

Zusammenfassung:

- Das Kompositum ist ein objektbasiertes Muster.
- Das Kompositum dient der Abbildung von Teil-Ganzes-Hierarchien.
- Das Kompositum lässt Klienten primitive und zusammengesetzte Objekte gleich behandeln.
- Das Kompositum leitet ggf. Methodenaufrufe an seine Kindobjekte (falls vorhanden) weiter.

Grundidee und Realisierung

Allgemeine Struktur:



Die Teilnehmer des Musters

Komponente (abstrakt) :

- deklariert die Schnittstellen der Objekte in der Struktur
- implementiert eventuell ein Defaultverhalten aller Objekte (gemeinsame Schnittstelle) falls dies sinnvoll erscheint
- deklariert eine Schnittstelle zum Zugriff auf eventuell vorhandene Kindobjekte
- deklariert und implementiert eventuell eine Schnittstelle zum Zugriff auf das Elternobjekt

Die Teilnehmer des Musters

Blatt :

- entspricht den Blättern des entstehenden Baumes
- implementiert das Verhalten des jeweiligen primitiven Objektes
- besitzt keine Kindobjekte

Die Teilnehmer des Musters

Kompositum :

- entspricht einer zusammengesetzten komplexen Struktur
- speichert Kindobjektreferenzen
- implementiert die Kindobjektschnittstelle von *Komponente*

Die Teilnehmer des Musters

Klient :

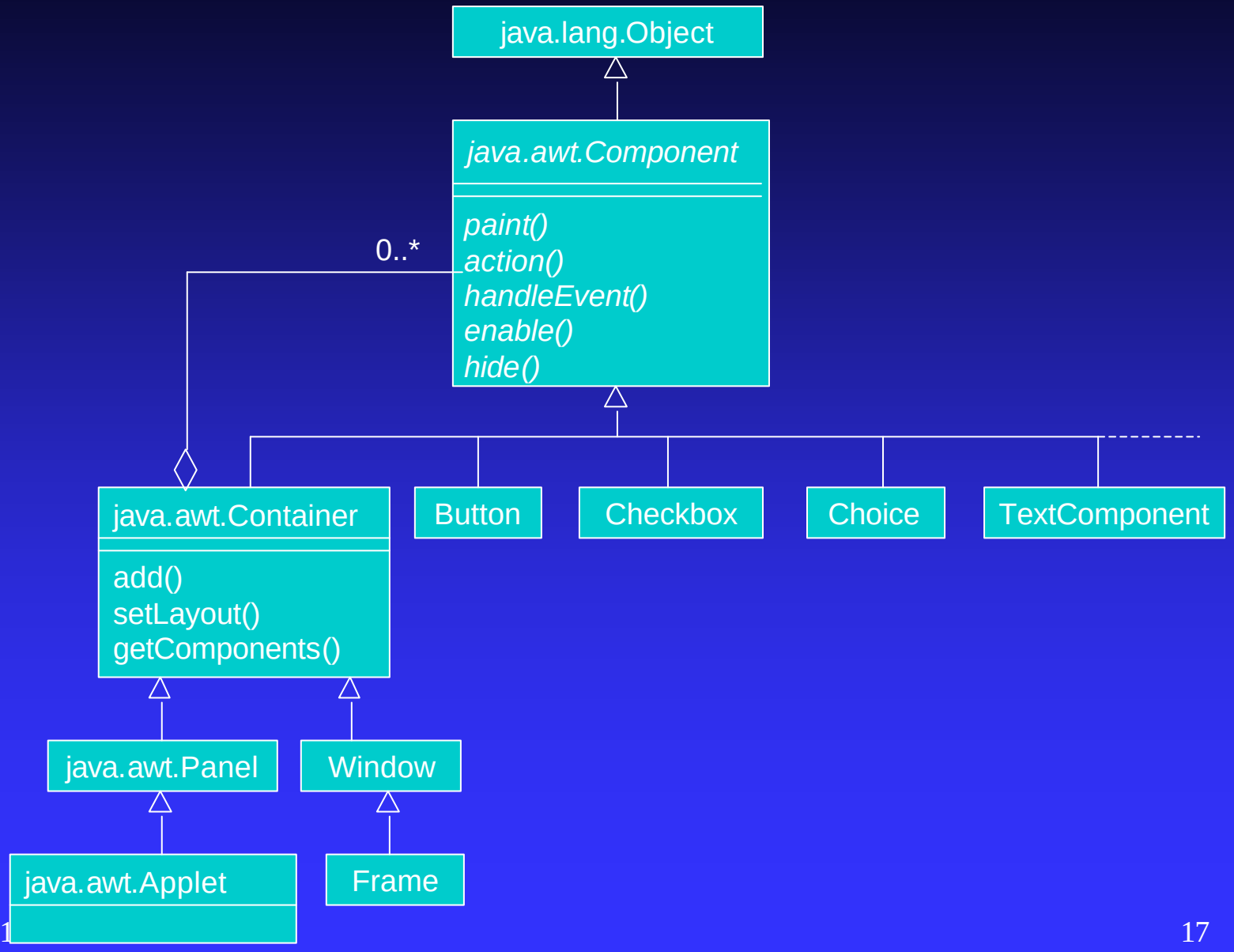
- interagiert mit Teilen der zusammengesetzten Struktur und manipuliert diese
- die tatsächliche Verarbeitung und (eventuelle) Delegation ist für den Klienten transparent
- bei der Anfrage an Kompositum-Objekt kann neben der Delegation auch zusätzlicher Code vor oder nach dem Weiterleiten ausgeführt werden

Das Java AWT

Beispielimplementierung Java AWT

- Advanced Windowing Toolkit
- ist nach dem Kompositum Muster implementiert
- die abstrakte Klasse kann in java.awt.Component wiedergefunden werden
- primitive Objekte sind z.B. Button, List, ...
- Kompositum ist die Klasse java.awt.Container

Klassenhierarchie (auszugsweise)



Konsequenzen

- das Muster definiert Klassenhierarchien mit Hilfe von primitiven und zusammengesetzten Objekten
- die primitiven Objekte können zu komplexen zusammengesetzt werden, diese können wiederum rekursiv verschachtelt werden
- Klienten unterscheiden nicht zwischen primitiven und zusammengesetzten Objekten
- der Klientencode kann schmaler gehalten werden (auf Implementierung verschiedener Behandlung der Objekte kann verzichtet werden)

Konsequenzen

- das nachträgliche Hinzufügen von Komponenten wird entschieden vereinfacht
- Komponenten müssen nur die Schnittstellen-anforderungen erfüllen (keine Anpassung der bestehenden Klassen oder Klienten nötig)

<u>Achtung</u>

- Variabilität beim Hinzufügen der neuen Komponenten macht eine Einschränkung schlecht möglich

Konsequenzen

- Typsystem reicht eventuell nicht aus, um Kompositionen auf bestimmte Elemente zu beschränken
- Verhindern wahlloser Vermengung beim Erzeugen neuer zusammengesetzter Objekte schwer
- Typüberprüfungen müssen zur Laufzeit durchgeführt werden
- kann aufgrund der Rekursion sehr teuer werden

Implementierungsvarianten

Elternobjektreferenzen:

- vereinfacht Traversieren durch die Baumstruktur (vor allem noch oben)
- üblicherweise in abstrakter *Komponente* definiert
- „Aufwärtslöschen“ muss verhindert werden
- Änderungen am Elternobjekt sollten deshalb nur beim Hinzufügen oder Entfernen der Komponente möglich sein und nur die Komponente betreffen

Implementierungsvarianten

Maximierung der Komponentenschnittstelle:

- Teilziel des Musters ist, Klassenstruktur vor Klienten zu verstecken
- möglichst vollständige Definition in abstrakter *Komponente* scheint daher sinnvoll
- steht im Widerspruch, nur solche Operationen auf höheren Schichten zu definieren, die <u>für alle Unterklassen</u> sinnvoll sind
- Lösung ist eine sinnvolle Defaultimplementierung die bei spezieller Bedeutung überschrieben wird

Beispiel: Funktion zum Zugriff auf Kindobjekte gibt den Defaultwert NULL zurück

Implementierungsvarianten

Verwaltungsoperationen für Kindobjekte:

- bei Definition auf oberster Ebene (Komponente) bleibt die Struktur transparent (einheitlich Handhabung) aber unsinnige Operationen werden möglich (Sicherheit)
- bei Definition in Kompositum haben primitive und zusammengesetzte Objekte unterschiedliche Schnittstellen (Widerspruch zur Zielstellung)

Workaround:

- Definition einer Methode, die bei Primitiven NULL und bei Zusammengesetzten Objekten das Objekt selbst (this) zurückgibt

Implementierungsvarianten

Ort des Containers für die Kindobjekte:

- bei Definition in *Komponente* wird in Primitiven Objekten Platz verschwendet
- im Falle von wenigen Kindklassen ist dies akzeptabel

Implementierungsvarianten

Verwaltung der Kindobjekte:

- die Verwaltung kann sinnvoll in einer Struktur erfolgen
- Struktur muss je nach Anforderungen gewählt werden
- es können Listen, Arrays, etc. verwendet werden, es kann aber auch auf eine Struktur verzichtet werden
- spielt die Reihenfolge der Kinder eine Rolle, muss die Schnittstelle dementsprechend entworfen werden (Iteratormuster, folgt noch)

Implementierungsvarianten

Laufzeitverhalten:

- häufiges Traversieren (rekursiv) kann Laufzeit kosten
- durch speichern von Kindinformationen können die Elternobjekte den Abstieg abkürzen oder vermeiden
- Änderungen an Kindobjekten erfordern erneute Validierung bei den Eltern (rekursiver Prozess)
- durch Referenz auf Elternobjekt wird dieser Prozess beschleunigt

Implementierungsvarianten

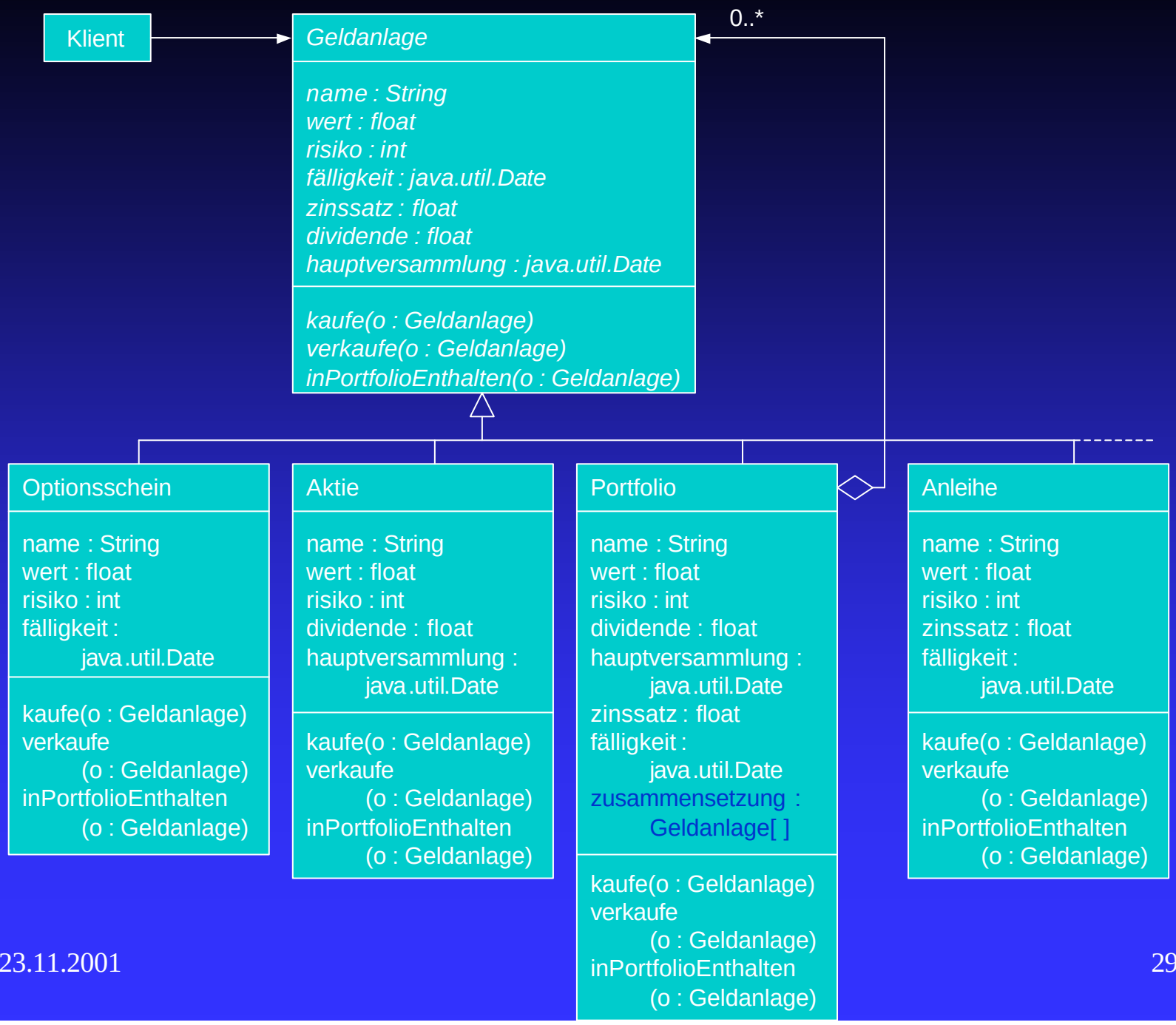
Speicherfreigabe:

- existiert keine Garbage Collection, so sollten Kompositionen sinnvollerweise beim Löschen ihre Kinder löschen (Delegation des Destruktoraufrufs)

Das Portfoliobeispiel

- ein Portfolio kann recht gut durch ein Kompositum abgebildet werden
- eine derartige Abbildung trifft die Semantik gut

Die folgende Struktur wäre beispielsweise denkbar:


Klient
Geldanlage
name : String
wert : float
risiko : int
fälligkeit : java.util.Date
zinssatz : float
dividende : float
hauptversammlung : java.util.Date
kaufe(o : Geldanlage)
verkaufe(o : Geldanlage)
inPortfolioEnthalten(o : Geldanlage)
0..*
Optionsschein
name : String
wert : float
risiko : int
fälligkeit : java.util.Date
kaufe(o : Geldanlage)
verkaufe (o : Geldanlage)
inPortfolioEnthalten (o : Geldanlage)
Aktie
name : String
wert : float
risiko : int
dividende : float
hauptversammlung : java.util.Date
kaufe(o : Geldanlage)
verkaufe (o : Geldanlage)
inPortfolioEnthalten (o : Geldanlage)
Portfolio
name : String
wert : float
risiko : int
dividende : float
hauptversammlung : java.util.Date
zinssatz : float
fälligkeit : java.util.Date
zusammensetzung : Geldanlage[ ]
kaufe(o : Geldanlage)
verkaufe (o : Geldanlage)
inPortfolioEnthalten (o : Geldanlage)
Anleihe
name : String
wert : float
risiko : int
zinssatz : float
fälligkeit : java.util.Date
kaufe(o : Geldanlage)
verkaufe (o : Geldanlage)
inPortfolioEnthalten (o : Geldanlage)

Das Portfoliobeispiel

Begründung der Definition:

- das Implementieren der Methoden ist in alle Klassen möglich, da eine Elternreferenz gehalten wird
- *kaufe()* bzw. *verkaufe()* kann sich bei primitiven Objekten wie Aktie auf die Aktie selbst beziehen
- das Portfolio kann mehrere Teilportfolios enthalten
- Verkauf eines Portfolios führt zum Verkauf aller Bestandteile (*Geldanlagen []*)
- das Risiko, die Dividende und der Zinssatz eines Portfolios können nach statistischen Formeln berechnet werden

Das Portfoliobeispiel

Begründung der Definition:

- Hauptversammlung gibt die nächste anstehende Hauptversammlung zurück (deren Datum)
- Fälligkeit gibt das älteste Datum der nächsten Fälligkeiten zurück
- der Wert des Portfolios wird durch Addieren der Werte aller Kindobjekte ermittelt

Das Portfoliobeispiel

Probleme:

- Elternreferenz des „obersten“ Portfolios müsste Referenz auf sich selbst zurückgeben
- primitive Objekte haben keine Kinder (Schnittstelle nicht einheitlich)
- das Anlegen von primitiven Objekten ohne zugehöriges Portfolio kann problematisch werden (evtl. Kontrollmechanismus implementieren Konstruktor könnte Elternreferenz verlangen)

Verwandte Muster

- Verwendung oft mit Dekorierermuster, in Form einer gemeinsam unterstützten Schnittstelle
- wird auf das Halten einer Elternreferenz verzichtet, so können nach Fliegengewichtmuster implementierte Komponenten von mehreren Eltern genutzt werden
- das Iteratormuster kann zum traversieren der Komposita verwendet werden
- das Besuchermuster kann Operationen und Verhalten lokalisieren

Quellen

Entwurfsmuster; Gamma,Helm,Johnson,Vlissides

Einführung in die Unified Modeling Language;
Prof.Dr.Jürgen Dunkel, FH Hannover

Software Engineering; Florian Matthes, Holm Wegner