Objektzugriff

Decorator, Proxy und Master-Slave

Mario Heidenreich

Fakultät für Mathematik und Informatik - Universität Leipzig

10. Juni 2009

Inhalt

1. Einführung
   - Strukturmuster
   - Zugriffspatterns
2. Die Pattern
   - Decorator
   - Proxy
   - Master-Slave
3. Zusammenfassung
   - Die Patterns im Überblick
   - Abgrenzung der Patterns untereinander

Überblick

1. Einführung
   - Strukturmuster
   - Zugriffspatterns
2. Die Pattern
   - Decorator
   - Proxy
   - Master-Slave
3. Zusammenfassung
   - Die Patterns im Überblick
   - Abgrenzung der Patterns untereinander

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind
- beschäftigen sich damit, wie auf eine Funktionalität zugegriffen wird

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind
- beschäftigen sich damit, wie auf eine Funktionalität zugegriffen wird
- verschiedene Ebenen möglich: klassenbasierte Strukturmuster / objektorientierte Muster

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind
- beschäftigen sich damit, wie auf eine Funktionalität zugegriffen wird
- verschiedene Ebenen möglich: klassenbasierte Strukturmuster / objektorientierte Muster
- andere Muster in diser Kategorie:

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind
- beschäftigen sich damit, wie auf eine Funktionalität zugegriffen wird
- verschiedene Ebenen möglich: klassenbasierte Strukturmuster / objektorientierte Muster
- andere Muster in diser Kategorie:
  - Adapter

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind
- beschäftigen sich damit, wie auf eine Funktionalität zugegriffen wird
- verschiedene Ebenen möglich: klassenbasierte Strukturmuster / objektorientierte Muster
- andere Muster in diser Kategorie:
  - Adapter
  - Facade

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind
- beschäftigen sich damit, wie auf eine Funktionalität zugegriffen wird
- verschiedene Ebenen möglich: klassenbasierte Strukturmuster / objektorientierte Muster
- andere Muster in diser Kategorie:
  - Adapter
  - Facade
  - Flyweight

Strukturmuster

- befassen sich alle mit der Art und Weise, wie Klassen und Objekte untereinander angeordnet sind
- beschäftigen sich damit, wie auf eine Funktionalität zugegriffen wird
- verschiedene Ebenen möglich: klassenbasierte Strukturmuster / objektorientierte Muster
- andere Muster in diser Kategorie:
  - Adapter
  - Facade
  - Flyweight
  - Kompositum

Überblick

1 Einführung
- Strukturmuster
- Zugriffspatterns

2 Die Pattern
- Decorator
- Proxy
- Master-Slave

3 Zusammenfassung
- Die Patterns im Überblick
- Abgrenzung der Patterns untereinander

Begriffe

Client

Klasse oder Komponente, die eine Methode oder den Service einer anderen Klasse oder Komponente nutzt

Component

Klasse oder Komponente, die Methode oder Service anbietet

Begriffe

Client

Klasse oder Komponente, die eine Methode oder den Service einer anderen Klasse oder Komponente nutzt

Component

Klasse oder Komponente, die Methode oder Service anbietet

- Zugriff eines Clients auf eine Komponente erfolgt normalerweise direkt

Begriffe

Client

Klasse oder Komponente, die eine Methode oder den Service einer anderen Klasse oder Komponente nutzt

Component

Klasse oder Komponente, die Methode oder Service anbietet

- Zugriff eines Clients auf eine Komponente erfolgt normalerweise direkt
- direkter Zugriff ist aber nicht immer erwünscht, inpraktikabel oder nicht möglich

Begriffe

Client

Klasse oder Komponente, die eine Methode oder den Service einer anderen Klasse oder Komponente nutzt

Component

Klasse oder Komponente, die Methode oder Service anbietet

- Zugriff eines Clients auf eine Komponente erfolgt normalerweise direkt
- direkter Zugriff ist aber nicht immer erwünscht, inpraktikabel oder nicht möglich
- weitere Patterns in dieser Kategorie: Facade, Iterator

	Architectural Patterns	Design Patterns	Idioms
Creation		*Abstract Factory* *Prototype* *Builder*	*Singleton* *Factory Method*
Structural Decomposition		Whole-Part (225) *Composite*	
Organization of Work		Master-Slave (245) *Chain of Responsibility* *Command* *Mediator*	
Access Control		Proxy (263) *Facade* *Iterator*	
Service Variation		*Bridge* *Strategy* *State*	*Template Method*
Service Extension		*Decorator* *Visitor*	
Management		Command Processor (277) View Handler (291) *Memento*	
Adaptation		*Adapter*	

Abbildung: Einteilung nach [POSA96]

Überblick

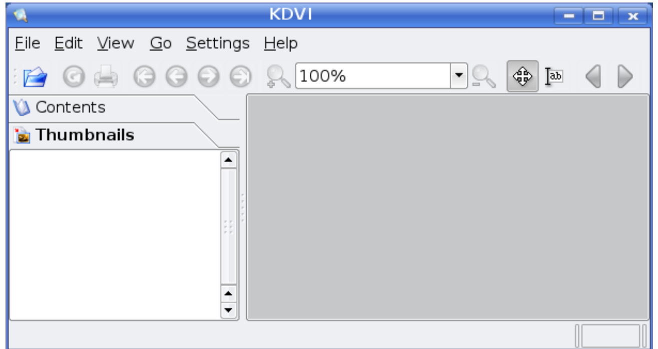
1 Einführung
- Strukturmuster
- Zugriffspatterns

2 Die Pattern
- Decorator
- Proxy
- Master-Slave

3 Zusammenfassung
- Die Patterns im Überblick
- Abgrenzung der Patterns untereinander

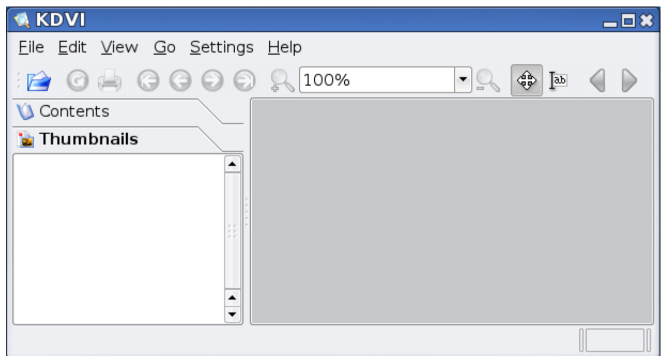
Motivation

- die PaintMethode eines WindowToolKits soll je nach Art eines Fensters dieses mit Rahmen oder Scrollbars versehen



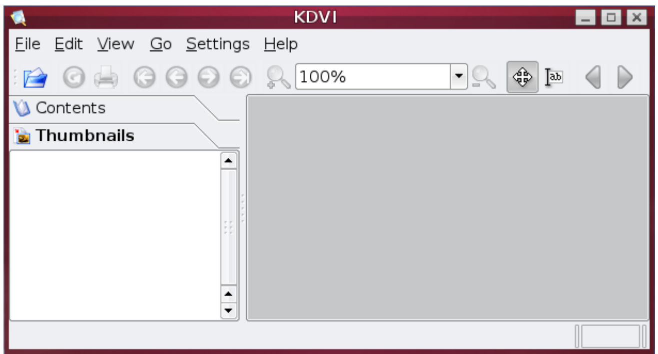
Motivation

- die PaintMethode eines WindowToolKits soll je nach Art eines Fensters dieses mit Rahmen oder Scrollbars versehen
- nicht jedes Fenster brauch Rahmen oder Scrollbars und es exisiteren verschiedene Unterarten von Rahmen



Motivation

- die PaintMethode eines WindowToolKits soll je nach Art eines Fensters dieses mit Rahmen oder Scrollbars versehen
- nicht jedes Fenster brauch Rahmen oder Scrollbars und es exisiteren verschiedene Unterarten von Rahmen
- das Aussehen der Fenster soll dynamisch änderbar sein

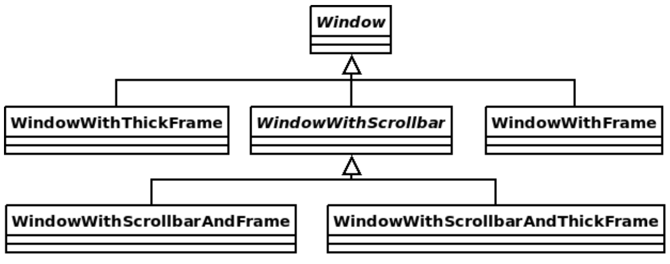


Motivation

- Ansatz: lege für jede Unterart eine eigene Klasse an

Motivation

- Ansatz: lege für jede Unterart eine eigene Klasse an
- Folge: Vererbungshierarchie ufert schnell aus



Motivaton

Naiver Ansatz

Wir führen ein zusätzliches Attribut ein, dass uns die Erweiterung an und abschalten lässt

Example

```
class clazz{
  [..]
  boolean isCritical=true;
  [..]
  public void accessThis (Requester requester){
    if(this.isCritical){
      checkPermissions(this, requester)
    }
    [..] }}
```

Motivation

Vorteile des naiven Ansatzes...

- simpler Ansatz, einfache Umsetzung
- Status einer Instanz leicht erkennbar

Motivation

Vorteile des naiven Ansatzes...

- simpler Ansatz, einfache Umsetzung
- Status einer Instanz leicht erkennbar

...und die Nachteile

- unser Objekt wird mit zusätzlichen Attributen überfrachtet
- zusätzlicher Overhead und "aufgeblähter" Code
- nur die vorgesehenen Methoden möglich

Motivation

Vorteile des naiven Ansatzes...

- simpler Ansatz, einfache Umsetzung
- Status einer Instanz leicht erkennbar

...und die Nachteile

- unser Objekt wird mit zusätzlichen Attributen überfrachtet
- zusätzlicher Overhead und "aufgeblähter" Code
- nur die vorgesehenen Methoden möglich → keine zuätzlichen Erweiterungen während der Laufzeit möglich

Das Problem allgemein gefasst

Grundlegende Ziele

- Funktionalität einer Klassenmethode soll erweitert werden
- Vererbung ist nicht immer möglich
- Vererbung ist unflexibel und führt mitunter zu unübersichtlich vielen Unterklassen

Das Problem allgemein gefasst

Grundlegende Ziele

- Funktionalität einer Klassenmethode soll erweitert werden
- Vererbung ist nicht immer möglich
- Vererbung ist unflexibel und führt mitunter zu unübersichtlich vielen Unterklassen

Gewünschte Eigenschaften der Lösung

- Pre- und Postprocessing von Methodenaufrufen
- soll dynamische Erweiterung von Klassenmethoden erlauben
- soll nur die Klasseninstanzen erweitern, die es auch nötig haben

Die Lösung: Decorator

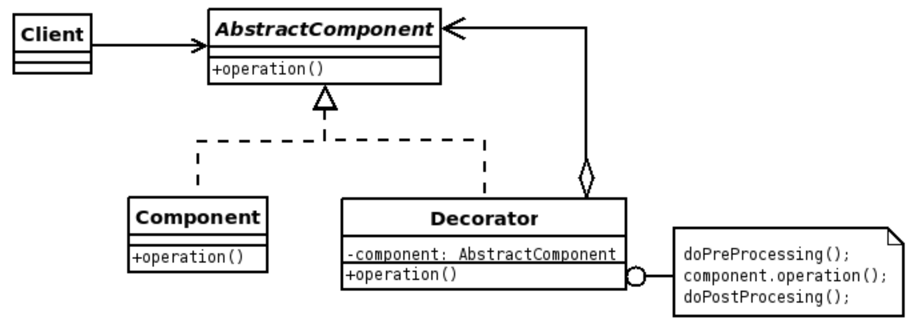
- aka Wrapper

Die Idee

- bau einen Wrapper für das originale Objekt, der das selbe Interface implementiert
- implementiere in diesem Wrapper die zusätzlichen Funktionen
- leite Aufrufe des Clients an das Interface auf das originale Objekt weiter

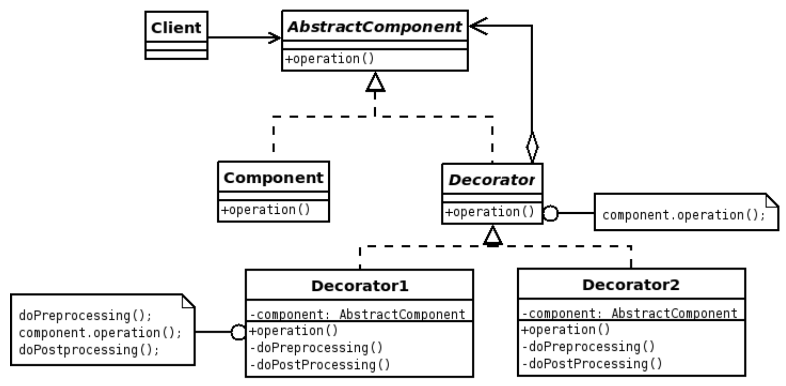
Die Lösung: Decorator

- aka Wrapper

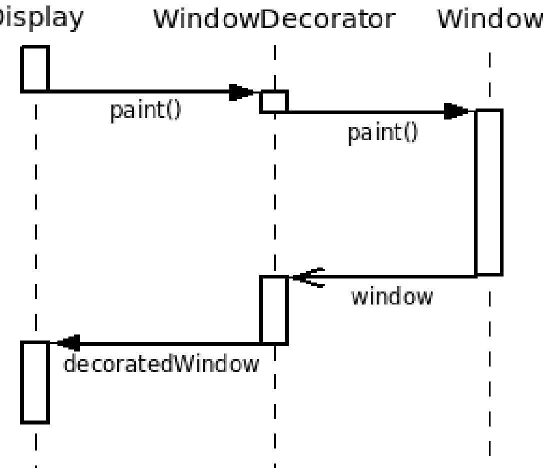


Verbesserte Version

- erlaubt mehrere Decorators
- erlaubt Unterscheidung zwischen Decorator und Component



Beispielaufruf



Anwendung

- WindowToolKits (Borders, Scrollbars, Verhalten)

org.eclipse.jface.viewers

Interface IBaseLabelProvider

All Known Subinterfaces:
IDebugModelPresentation, ILabelDecorator, ILabelProvider, ITableLabelProvider

All Known Implementing Classes:
DecoratingLabelProvider, FileEditorMappingLabelProvider, LabelProvider

Anwendung

- WindowToolKits (Borders, Scrollbars, Verhalten)
- Java I/O: OutputStream, InputStream

java.io
Class FilterInputStream

java.lang.Object
 └ java.io.InputStream
   └ **java.io.FilterInputStream**

All Implemented Interfaces:
Closeable

Direct Known Subclasses:
BufferedInputStream, CheckedInputStream, CipherInputStream, DataInputStream, DeflaterInputStream, DigestInputStream, InflaterInputStream, LineNumberInputStream, ProgressMonitorInputStream, PushbackInputStream

Vorteile des Decorator Patterns

- höhere Flexibilität gegenüber normaler Vererbung
  - Funktionalität kann zur Laufzeit hinzugefügt werden
  - mehrfache Verschachtelung von Decoratorn möglich

Vorteile des Decorator Patterns

- höhere Flexibilität gegenüber normaler Vererbung
  - Funktionalität kann zur Laufzeit hinzugefügt werden
  - mehrfache Verschachtelung von Decoratorn möglich
- Funktionen werden erst dann hinzugefügt, wenn sie benötigt werden $\rightarrow$ Klassen bleiben einfach

Vorteile des Decorator Patterns

- höhere Flexibilität gegenüber normaler Vererbung
  - Funktionalität kann zur Laufzeit hinzugefügt werden
  - mehrfache Verschachtelung von Decoratorn möglich
- Funktionen werden erst dann hinzugefügt, wenn sie benötigt werden $\rightarrow$ Klassen bleiben einfach
- Decorator bleibt für den Client transparent

Nachteile des Decorator Patterns

- mehrere kleine Objekte nehmen mehr Platz weg als ein großes

Nachteile des Decorator Patterns

- mehrere kleine Objekte nehmen mehr Platz weg als ein großes
- ein Objekt ist nicht identisch mit seinem dekorierten Objekt

Nachteile des Decorator Patterns

- mehrere kleine Objekte nehmen mehr Platz weg als ein großes
- ein Objekt ist nicht identisch mit seinem dekorierten Objekt
- Code unter Umständen schwerer zu verstehen und zu debuggen

Überblick

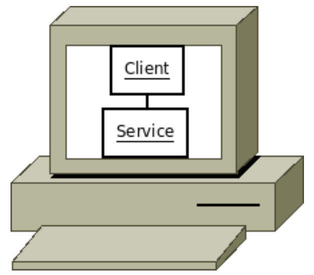
1 Einführung
- Strukturmuster
- Zugriffspatterns

2 Die Pattern
- Decorator
- Proxy
- Master-Slave

3 Zusammenfassung
- Die Patterns im Überblick
- Abgrenzung der Patterns untereinander

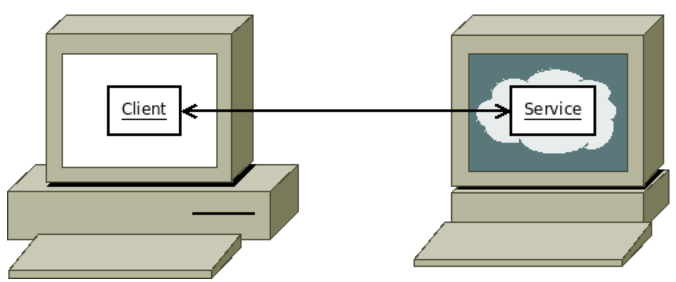
Motivation

- Beispiel: ein Client möchte Funktion einer Component nutzen



Motivation

- Beispiel: ein Client möchte Funktion einer Component nutzen
- Component wurde aber auf einen Server migriert
- Client muss nun Remote auf die Funktionen zugreifen



Motivation

Naiver Ansatz

Client implementiert zusätzlichen Aufrufmechanismus

Beispiel

```
class Client { [..]
  private String serverAddr= "192.168.2.210"
  private String serverPort = "4223"
  public Object doSomething(Object[] params)
throws RandomException {
    socket = new Socket(serverAddr, serverPort);
    OutputStream out = socket.getOutputStream();
    [..] // call the method
    InputStream in = sockert.getInputStream();
    result = computeResult(in);
    return object;} }
```

Vorteile des Naiven Ansatzes...

- Overheadvermeidung bei vereinzelten Aufrufen

Vorteile des Naiven Ansatzes...

- Overheadvermeidung bei vereinzelten Aufrufen
- leicht verständlich, einfach zu debuggen

Vorteile des Naiven Ansatzes...

- Overheadvermeidung bei vereinzelten Aufrufen
- leicht verständlich, einfach zu debuggen
- Vermeidung zusätzlicher Schnittstellen

Vorteile des Naiven Ansatzes...

- Overheadvermeidung bei vereinzelten Aufrufen
- leicht verständlich, einfach zu debuggen
- Vermeidung zusätzlicher Schnittstellen

Vorteile des Naiven Ansatzes...

- Overheadvermeidung bei vereinzelten Aufrufen
- leicht verständlich, einfach zu debuggen
- Vermeidung zusätzlicher Schnittstellen

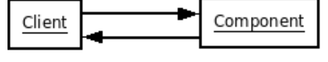
...und die Nachteile

- Client wird mit unnötiger Logik beladen, Klasse wird aufgebläht
  - Fehlerbehandlung
  - Sicherheitsaspekte
- Client muss für Aufruf Serveradresse wissen
- mehrere Clients bauen auch mehrere Verbindungen auf
  - Connection Pooling?
  - Synchronisierung?
  - Caching?

Das Problem allgemein gefasst

Problem

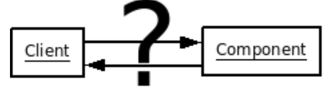
- ein Klient möchte Zugriff auf eine andere Komponente



Das Problem allgemein gefasst

Problem

- ein Klient möchte Zugriff auf eine andere Komponente
- direkter Zugriff wäre möglich ist aber inpraktikabel



Anforderungen an die Lösung

1. Zugriff auf die Komponente sollte ressourcenschonend bleiben/sein
2. Zugriff sollte auf beiden Seiten sicher sein
3. keine Änderungen im Interface, d.h. keine Änderungen für den Client im Aufruf

Die Lösung: ein Proxy

Die Idee

- aka Stellvertreter (“surrogate”), Repräsentant (“ambassador”)

füge eine zusätzliche Komponente mit dem benötigten Interface zw. Client und Component ein, die zusätzliche Aufgaben implementiert

Die Lösung: ein Proxy

Die Idee

- aka Stellvertreter ("surrogate"), Repräsentant ("ambassador")

füge eine zusätzliche Komponente mit dem benötigten Interface zw. Client und Component ein, die zusätzliche Aufgaben implementiert

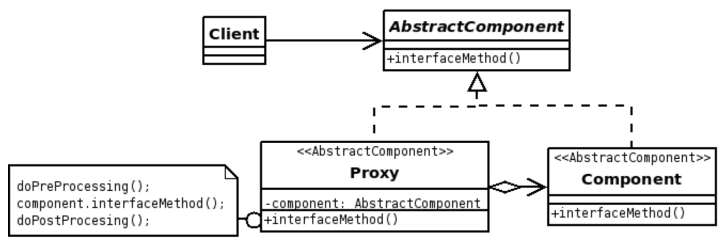
- Proxy erhält Reference auf die Component(s), die er verwalten soll
- Client schickt Interfaceaufruf an den Proxy statt direkt zur Component
- Proxy behandelt den Funktionsaufruf, ruft eventuell Methoden der verwalteten Component(s) auf und schickt Antwort an den Client

Die Lösung: ein Proxy

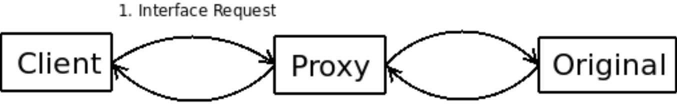
Die Idee

- aka Stellvertreter ("surrogate"), Repräsentant ("ambassador")

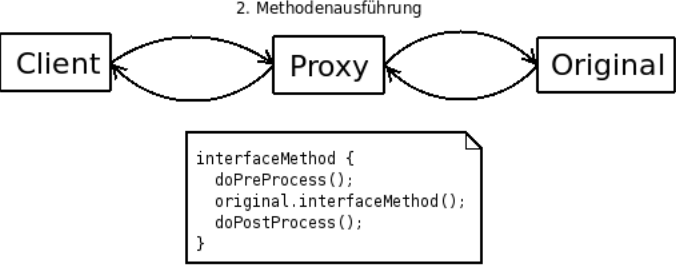
füge eine zusätzliche Komponente mit dem benötigten Interface zw. Client und Component ein, die zusätzliche Aufgaben implementiert



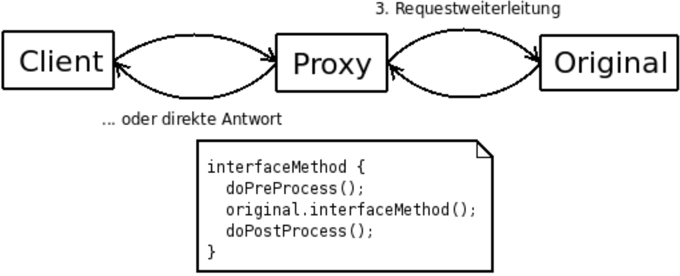
Beispielausführung



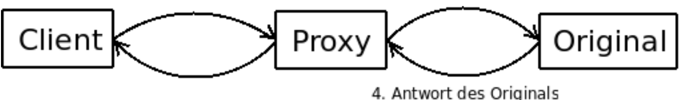
Beispielausführung



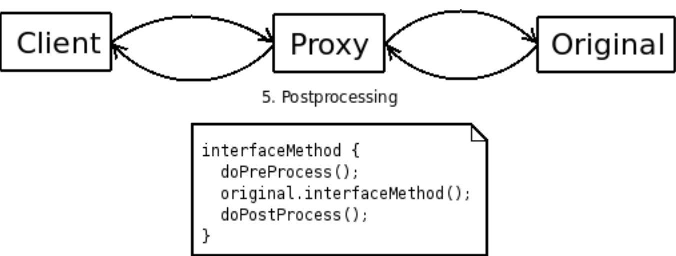
Beispielausführung



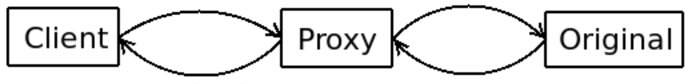
Beispielausführung



Beispielausführung



Beispielausführung



Unterarten des Proxypatterns

Remote Proxy

RPC und Netzwerkattribute sollen im Fall von Remote Components vor dem Client verborgen werden

Unterarten des Proxypatterns

Remote Proxy

RPC und Netzwerkattribute sollen im Fall von Remote Components vor dem Client verborgen werden

Protection Proxy

Originalkomponente soll vor unbefugten Zugriff geschützt werden

Unterarten des Proxypatterns

Remote Proxy

RPC und Netzwerkattribute sollen im Fall von Remote Components vor dem Client verborgen werden

Protection Proxy

Originalkomponente soll vor unbefugten Zugriff geschützt werden

Cache Proxy

mehrere Clients teilen sich eine read-only Component

Unterarten des Proxypatterns

Remote Proxy

RPC und Netzwerkattribute sollen im Fall von Remote Components vor dem Client verborgen werden

Protection Proxy

Originalkomponente soll vor unbefugten Zugriff geschützt werden

Cache Proxy

mehrere Clients teilen sich eine read-only Component

Synchronization Proxy

mehrere Clients teilen sich eine synchronisierte Component

Unterarten (Fortsetzung)

Counting Proxy

Proxy sammelt Statistiken-z.B. Anzahl der Zugriffe

Unterarten (Fortsetzung)

Counting Proxy

Proxy sammelt Statistiken-z.B. Anzahl der Zugriffe

Virtual Proxy

der Proxy stellt "Metainformationen" über die Component oder Teile davon bereit um direkt Zugriff möglichst zu sparen
aka "Lazy Loading"

Unterarten (Fortsetzung)

Counting Proxy

Proxy sammelt Statistiken-z.B. Anzahl der Zugriffe

Virtual Proxy

der Proxy stellt "Metainformationen" über die Component oder Teile davon bereit um direkt Zugriff möglichst zu sparen
aka "Lazy Loading"

Firewall Proxy

lokale Components werden vor externen Zugriffen geschützt

Konkrete Anwendungen

- häufig übernimmt ein Proxy mehrere der genannten Rollen z.B. Remote Proxy und Protection Proxy, Virtual Proxy und Cache Proxy

Konkrete Anwendungen

- häufig übernimmt ein Proxy mehrere der genannten Rollen z.B. Remote Proxy und Protection Proxy, Virtual Proxy und Cache Proxy

Beispiele

- Stubs - z.B. SOAP, RMI, CORBA

Konkrete Anwendungen

- häufig übernimmt ein Proxy mehrere der genannten Rollen z.B. Remote Proxy und Protection Proxy, Virtual Proxy und Cache Proxy

Beispiele

- Stubs - z.B. SOAP, RMI, CORBA
- Datenbanken - z.B. Hibernate

Konkrete Anwendungen

- häufig übernimmt ein Proxy mehrere der genannten Rollen z.B. Remote Proxy und Protection Proxy, Virtual Proxy und Cache Proxy

Beispiele

- Stubs - z.B. SOAP, RMI, CORBA
- Datenbanken - z.B. Hibernate
- Webproxy - Http Responses cachen

Vorteile des Proxy Patterns

- spart Ressourcen, indem es Zugriffe oder Kopien spart (Virtual Proxy, Cache)

Vorteile des Proxy Patterns

- spart Ressourcen, indem es Zugriffe oder Kopien spart (Virtual Proxy, Cache)
- Components werden von sich wiederholenden Aufgaben befreit, die vieleicht auch gar nix direkt mit den eigentlichen Methoden zu tun haben

Vorteile des Proxy Patterns

- spart Ressourcen, indem es Zugriffe oder Kopien spart (Virtual Proxy, Cache)
- Components werden von sich wiederholenden Aufgaben befreit, die vieleicht auch gar nix direkt mit den eigentlichen Methoden zu tun haben
- Kommunikation zwischen Proxy und Component ist für Client transparent $\rightarrow$ der Client ist unabhängig von z.B. der Adresse des Servers

Nachteile des Proxy Patterns

- ein indirekter Zugriff bedeutet immer zusätzlichen Overhead

Nachteile des Proxy Patterns

- ein indirekter Zugriff bedeutet immer zusätzlichen Overhead
- Virtual Proxies, Cache Proxies sind nicht in jedem Fall von Vorteil

Nachteile des Proxy Patterns

- ein indirekter Zugriff bedeutet immer zusätzlichen Overhead
- Virtual Proxies, Cache Proxies sind nicht in jedem Fall von Vorteil
- Proxies als Flaschenhals

Nachteile des Proxy Patterns

- ein indirekter Zugriff bedeutet immer zusätzlichen Overhead
- Virtual Proxies, Cache Proxies sind nicht in jedem Fall von Vorteil
- Proxies als Flaschenhals

Nachteile des Proxy Patterns

- ein indirekter Zugriff bedeutet immer zusätzlichen Overhead
- Virtual Proxies, Cache Proxies sind nicht in jedem Fall von Vorteil
- Proxies als Flaschenhals

Der Einsatz dieses Patterns sollte gut überlegt sein, kann aber große Vorteile bringen.

Überblick

1. Einführung
   - Strukturmuster
   - Zugriffspatterns
2. Die Pattern
   - Decorator
   - Proxy
   - Master-Slave
3. Zusammenfassung
   - Die Patterns im Überblick
   - Abgrenzung der Patterns untereinander

Motivation

Bildberechnung

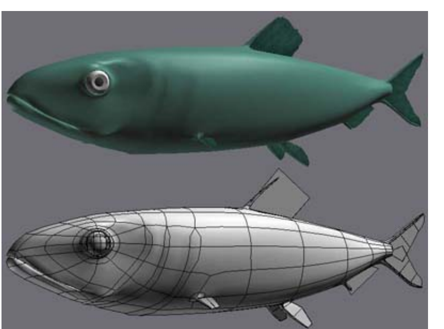


Abbildung: http://ccny.acm.org

Motivation

Bildberechnung

- rendern eines 3D Bildes ist aufwändig

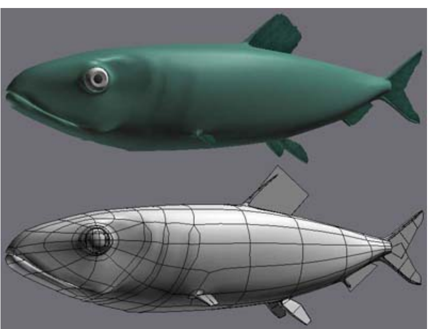


Abbildung: http://ccny.acm.org

Motivation

Bildberechnung

- rendern eines 3D Bildes ist aufwändig
- häufig Millionen Polynome pro Bild

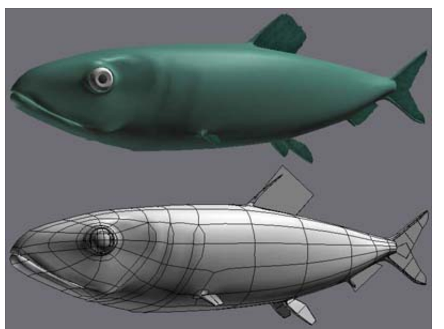


Abbildung: http://ccny.acm.org

Motivation

Bildberechnung

- rendern eines 3D Bildes ist aufwändig
- häufig Millionen Polynome pro Bild
- Bild kann in unabhängige Bereiche zerlegt werden

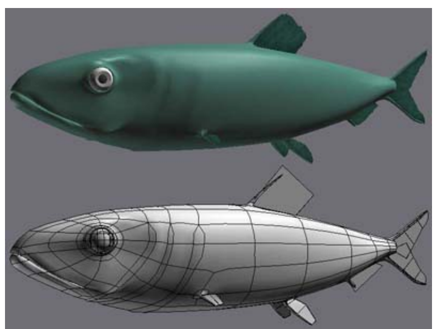


Abbildung: http://ccny.acm.org

Motivation

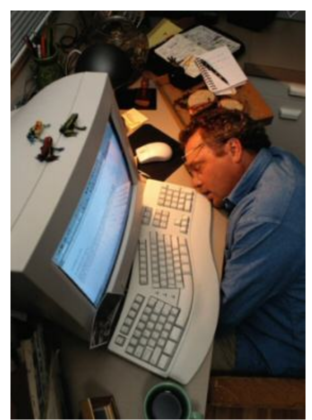
Naiver Ansatz

eine Component berechnet alles alleine...

Motivation

Naiver Ansatz

eine Component berechnet alles alleine...
...und der Client wartet



Motivation

Vorteile des naiven Ansatzes...

- Component erhält volle Kontrolle über die Verarbeitung der Aufgabe
  - wenig Overhead
  - einfache Implementierung, leicht nach zu verfolgen
- keine Probleme durch zusätzliche Kommunikation oder Synchronisation mit anderen Prozessen/Threads

Motivation

Vorteile des naiven Ansatzes...

- Component erhält volle Kontrolle über die Verarbeitung der Aufgabe
  - wenig Overhead
  - einfache Implementierung, leicht nach zu verfolgen
- keine Probleme durch zusätzliche Kommunikation oder Synchronisation mit anderen Prozessen/Threads

...und die Nachteile

- Component muss sich um Ressourcenteilung kümmern oder Ressourcen liegen brach
- Implementierung der Berechnung statisch
- Component ist Single Point of Failure

Das Problem allgemein gefasst

Das Problem

- Aufgabe(n) sollen in identische Subtasks aufgelöst werden
- das voranschreiten der Teilprozesse geschieht unabhängig voneinander
- das Gesamtresultat berechnet sich aus den Subtasks

Das Problem allgemein gefasst

Das Problem

- Aufgabe(n) sollen in identische Subtasks aufgelöst werden
- das voranschreiten der Teilprozesse geschieht unabhängig voneinander
- das Gesamtresultat berechnet sich aus den Subtasks

Anforderungen an die Lösung

- Aufgabenverteilung bleibt für den Client transparent
- weder Client noch die Teilberechnung sind abhängig vom Verteilungsalgorithmus oder von der finalen Berechnung
- die eigentliche Berechnung der Subprozesse ist implementierungsunabhängig
- Teilberechnungen müssen event. koordiniert werden

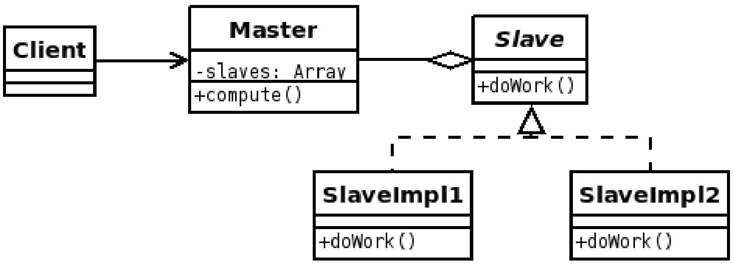
Die Lösung: Das Master-Slave Pattern

- eine Aufgabe wird auf mehrere Subtasks aufgeteilt
- Einführung einer koordnierenden Komponente zwischen Client und der eigentlichen Berechnung
- "Master" Komponente erzeugt identische[1] Subprozesse ("Slaves") und berechnet aus deren Resultaten das Gesamtergebnis

[1] sie haben das selbe Interface

Die Lösung: Das Master-Slave Pattern

- eine Aufgabe wird auf mehrere Subtasks aufgeteilt
- Einführung einer koordnierenden Komponente zwischen Client und der eigentlichen Berechnung
- "Master" Komponente erzeugt identische[1] Subprozesse ("Slaves") und berechnet aus deren Resultaten das Gesamtergebnis



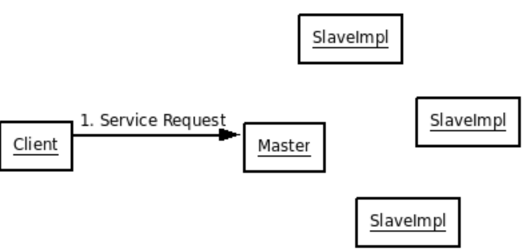
[1] sie haben das selbe Interface

Anwendungsfälle

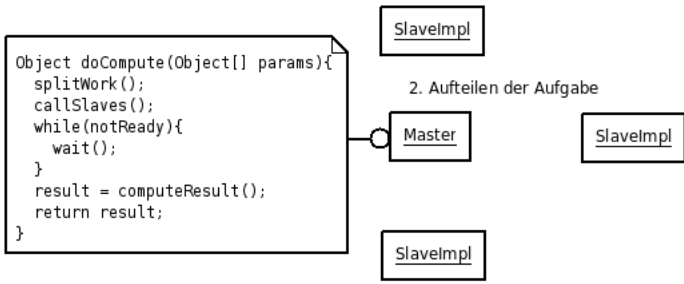
Erhöhung der Gesamtgeschwindigkeit

- Aufgabe wird in verschiedene Subtasks aufgeteilt
- Gesamtergebnis benötigt alle Teilergebnisse
- Master unterbricht seine Ausführung, bis er das Gesamtergebnis berechnen kann

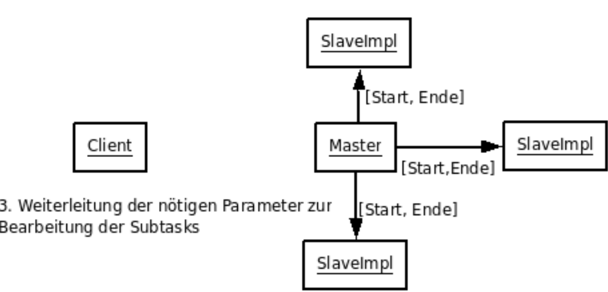
Anwendungsfälle



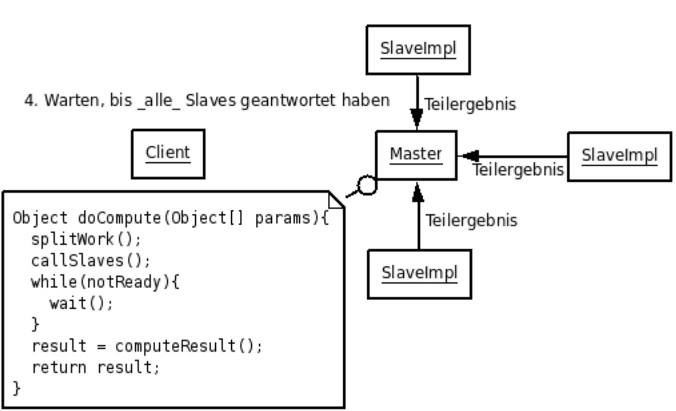
Anwendungsfälle



Anwendungsfälle



Anwendungsfälle



Anwendungsfälle

5. Berechnen des Gesamtergebnis und Antwort

SlaveImpl

Client

Ergebnis

Master

SlaveImpl

SlaveImpl

```
Object doCompute(Object[] params){
  splitWork();
  callSlaves();
  while(notReady){
    wait();
  }
  result = computeResult();
  return result;
}
```

Anwendungsfälle

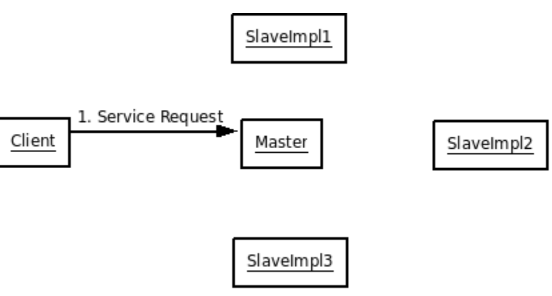
Erhöhung der Genauigkeit einer Berechnung

- die Aufgabe/ der Service wird vom Master an eine Zahl[a] verschiedene Implementierungen weitergegeben
- der Master wartet auf alle Ergebnisse und berechnet das Gesamtergebnis
- verschiedene Ansätze denkbar: Durchschnittsbildung, Zähler für das häufigste Ergebnis

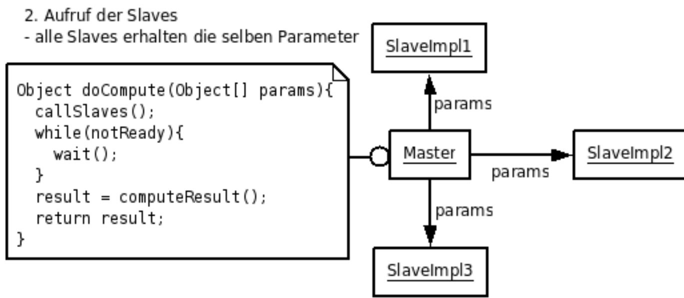
---

[a] *[..] at least three[..][POSA96]*

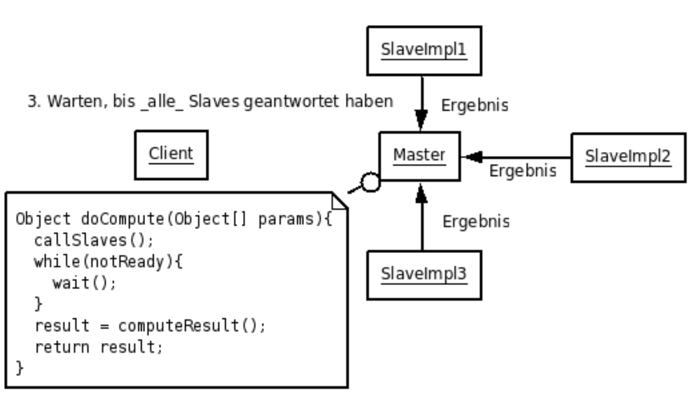
Anwendungsfälle



Anwendungsfälle



Anwendungsfälle



Anwendungsfälle

SlaveImpl1

4. Berechnen des Gesamtergebnis und Antwort

Client

Ergebnis

Master

SlaveImpl2

```
Object doCompute(Object[] params){
  callSlaves();
  while(notReady){
    wait();
  }
  result = computeResult();
  return result;
}
```

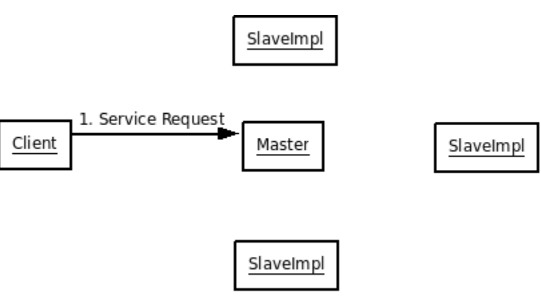
SlaveImpl3

Anwendungsfälle

Fehler -und Ausfallprävention eines Services

- die Masterkomponente erzeugt mehrere gleichartige Slaves und lässt diese das Gleiche berechnen
- das Ergebnis des ersten korrekt terminierenden Slaves wird über den Master dem Client zurückgegeben
- Fehlermeldung erst, wenn alle Slaves fehlschlagen

Anwendungsfälle

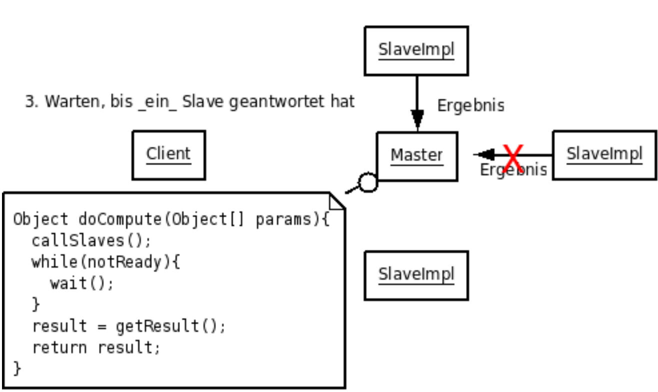


Anwendungsfälle

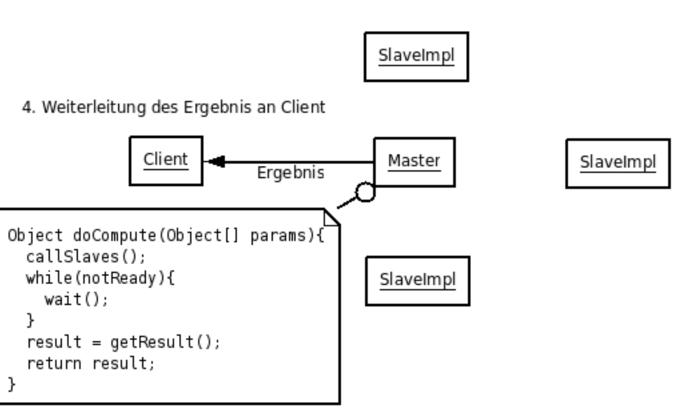
2. Aufruf der Slaves
- alle Slaves erhalten die selben Parameter

```
Object doCompute(Object[] params){
  while(notReady){
    wait();
  }
  result = getResult();
  return result;
}
```

Anwendungsfälle



Anwendungsfälle



Konkrete Anwendungsfälle

- Matrixberechnung

Konkrete Anwendungsfälle

- Matrixberechnung
- Bildberechnung/-verarbeitung
  - ParaGL 3D Real Time Rendering auf Cluster
  - K-3D Open Source 3D Modellierung und Animation erlaubt Network Rendering

Konkrete Anwendungsfälle

- Matrixberechnung
- Bildberechnung/-verarbeitung
  - ParaGL 3D Real Time Rendering auf Cluster
  - K-3D Open Source 3D Modellierung und Animation erlaubt Network Rendering
- alle möglichen parallelisierbaren Algorithmen
  - Divide-and-Conquer
  - Dynamische Programmierung
  - Monte Carlo Algorithmen
  - Brute Force Ansätze

Vorteile des Master-Slave Patterns

- Erhöhung der Auslastung / Erhöhung der Berechnungsgeschwindigkeit (parallele Verarbeitung)
- Implementierung und Ausführungsort sind transparent für Client
- Austauschbarkeit der ausgeführten Implementierungen
- Client wird von Nebenaspekten (Fehlerbehandlung, Kommunikation etc.) befreit

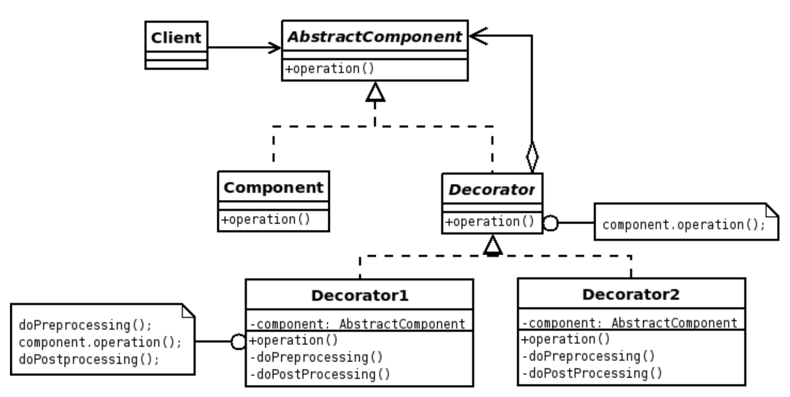
Nachteile des Master-Slave Patterns

- Machbarkeit: Aufgabenverteilung, benötigte Ressourcen
- Abhängigkeit von der unterliegenden Architektur (parallele Verarbeitung)
- hoher Implementierungsaufwand (Aufgabenteilung, Kommunikation, Synchronisation etc.)
- eingeschränkte Portabilität, erhöhtes Fehlerpotential, erschwertes Debugging

Überblick

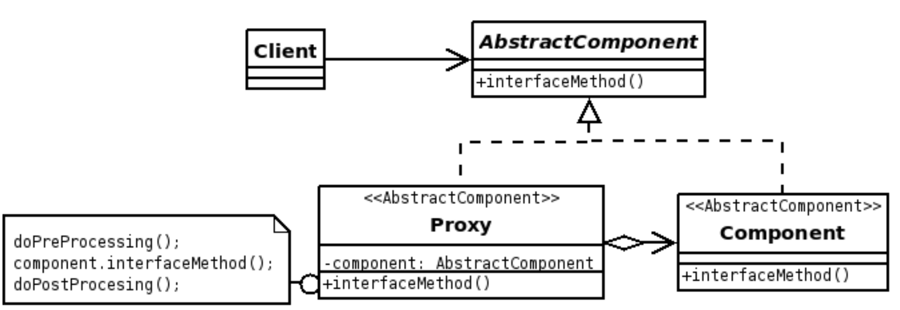
1. Einführung
   - Strukturmuster
   - Zugriffspatterns
2. Die Pattern
   - Decorator
   - Proxy
   - Master-Slave
3. Zusammenfassung
   - Die Patterns im Überblick
   - Abgrenzung der Patterns untereinander

Decorator



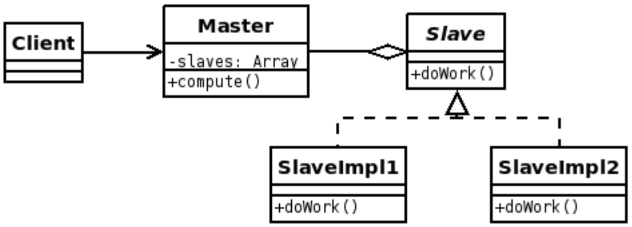
- erweitert Objekt dynamisch um Funktionen
- implementiert das selbe Interface wie die Component, die es verwaltet
- erlaubt Verschachtelungen

Proxy



- dient als Zugriffskontrolle auf ein Objekt
- übernimmt Aufgaben, die nicht in die aufgerufene Component gehören
- implementiert das selbe Interface wie die Component, die es verwaltet

Master-Slave



- Masterkomponente teilt Aufgabe in mehrere Teilaufgaben
- Slaves implementieren alle das selbe Interface
- Master meist als Singleton umgesetzt

Überblick

1 Einführung
- Strukturmuster
- Zugriffspatterns

2 Die Pattern
- Decorator
- Proxy
- Master-Slave

3 Zusammenfassung
- Die Patterns im Überblick
- Abgrenzung der Patterns untereinander

Was ist der Unterschied zwischen...

... Decorator und Kompositum

Decorator ist nicht nur für Objektaggragation verantwortlich und fügt Funktionalität hinzu

Was ist der Unterschied zwischen...

... Decorator und Kompositum

Decorator ist nicht nur für Objektaggragation verantwortlich und fügt Funktionalität hinzu

...Decorator und Adapter

Decorator ändert nicht die Schnittstelle eines Objektes

Was ist der Unterschied zwischen...

... Decorator und Kompositum

Decorator ist nicht nur für Objektaggragation verantwortlich und fügt Funktionalität hinzu

...Decorator und Adapter

Decorator ändert nicht die Schnittstelle eines Objektes

...Decorator und Strategie

Strategie ändert nur "innere" Teile eines Objekts, der Decorator bildet einen "Wrapper"

Was ist der Unterschied zwischen...

...Proxy und Adapter

Adapter ändern eine Schnittstelle, ein Proxy bietet die Selbe an oder eine Untermenge davon (Protection Proxy)

Was ist der Unterschied zwischen...

...Proxy und Adapter

Adapter ändern eine Schnittstelle, ein Proxy bietet die Selbe an oder eine Untermenge davon (Protection Proxy)

...Proxy und Decorator

Decorator erweitert eine Klasse dynamisch um Zuständigkeiten, Proxyies versuchen das Objekt von Zuständigkeiten zu befreien

Literatur

Buschmann, F.; Meunier R. et al.
Pattern-Oriented Software Architecture
A System of Patterns
John Wiley & Sons, 1996

Buschmann, F.; Henney, K.; Schmidt, D.C.
Pattern-Oriented Software Architecture
A Pattern Language for Distributed Computing
John Wiley & Sons, 2007

Gamma, E.; Helm, R.; Johnson, R.; Vlissides, J.
Entwurfsmuster
Addison-Wesley, 2004

David Geary
http://www.javaworld.com/javaworld/jw-10-2001/jw-1012-designpatterns.html?
(09.06.2009)