2013

APR- Quakologie

Protokoll zur Arbeitsdurchführung

Patrick Mühl, Ahmed Aly
TGM
01.01.2013

Inhalt

Aufgabenstellung

Implementieren Sie die Quakologie in C++ und dokumentieren Sie nach den Protokollregeln!

Gesamtpunkte (16)

Funktionalität 4Pkt

Kommentare (doxygen [1]) 2Pkt

Protokoll 10Pkt

- UML/Beschreibung und Identifikation der Patterns ... 6Pkt

- restliche Protokollierung ... 4Pkt

[1] http://www.stack.nl/~dimitri/doxygen

Arbeitsschritte/ Arbeitsaufteilung

Arbeitsschritte	Ahmed Aly	Patrick Mühl
UML Diagramm	x	
Implementierung des Codes	x	
Code Coverage	x	
Ausarbeiten der Pattern		x
Code Dokumentation		x
Protokoll		x

Zeitaufteilung

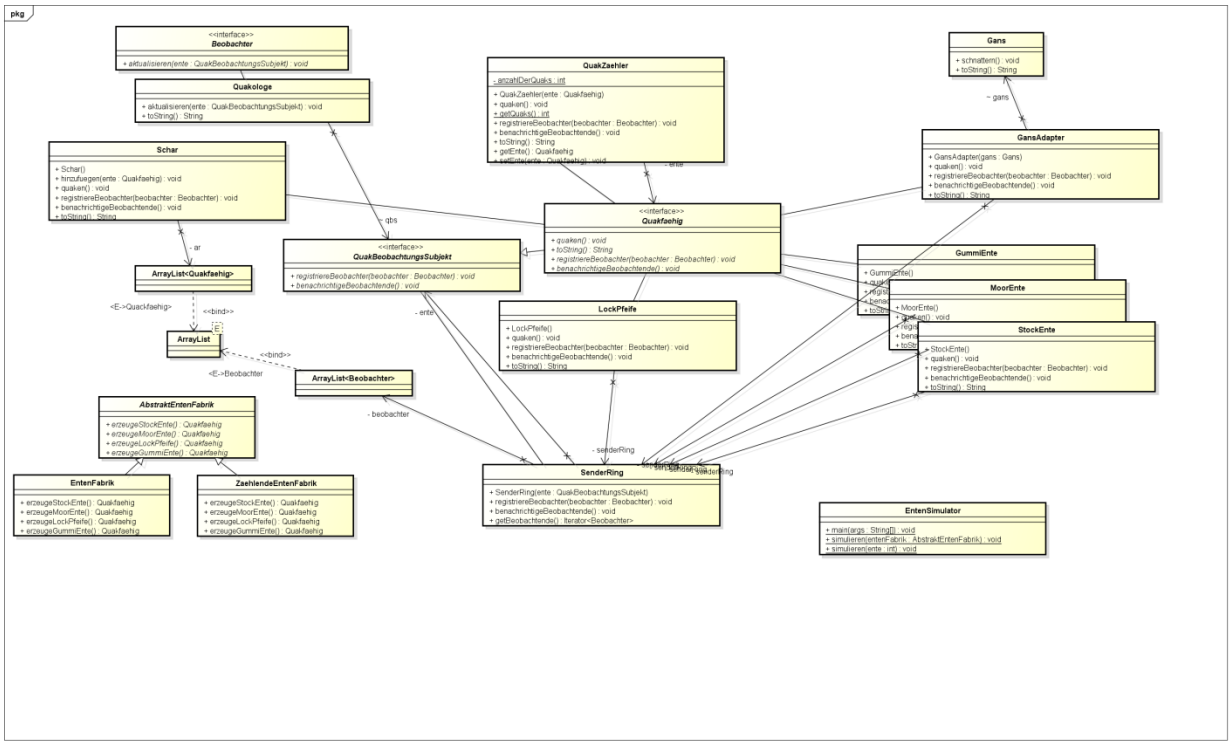
Ahmed Aly

Arbeitsschritte	Geschätzte Zeit	Benötigte Zeit
UML Diagramm	30 min	30 min
Implementierung des Codes	3h	4h
Code Coverage	3 h	3 h 30 min
Gesamt	**6h 30min**	**8h**

Patrick Mühl

Arbeitsschritte	Geschätzte Zeit	Benötigte Zeit
Ausarbeitung der Pattern	3h	3h
Code Dokumentation	2h	1h 30 min
Protokoll	30 min	30 min
Gesamt	**5h 30 min**	**5h**

Designüberlegung



Arbeitsdurchführung

Erfolge

- Ausarbeitung der Pattern war durch die vorhandene Ausarbeitung des letzten Jahres nicht so langwierig

Niederlagen

- doxygen funktioniert nicht

Probleme

- C++ lange nicht mehr programmiert
- Implementierung von Header Files oft vergessen
- Pointer nicht beachtet
- virtuelle Methoden danach =0 setzen

Technologienbeschreibung

Decorator Pattern[1]

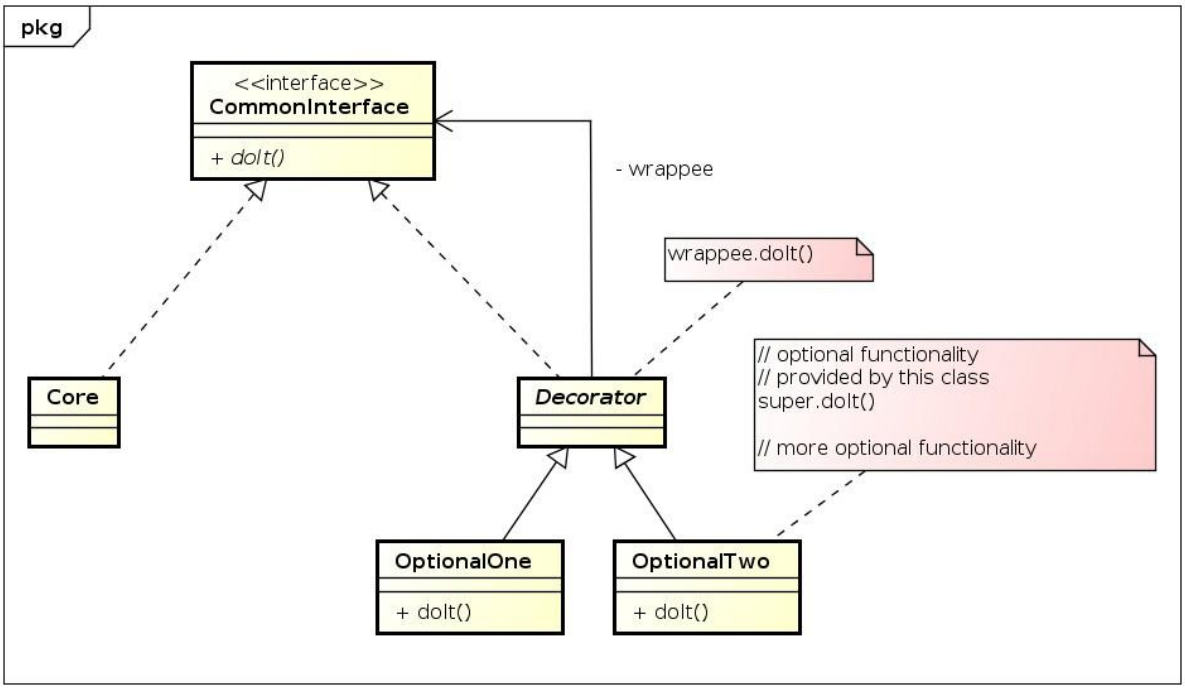
Das Decorator Pattern bearbeitet Probleme der Erweiterbarkeit des Codes.

Hat man z.B. Klassen die während der Laufzeit erweitert werden sollen, benötigt man für jeden möglichen Zustand eine neue Klasse. Dies macht den Code sehr unübersichtlich und schwer Wartbar.

Daher wurde das Decorator Pattern als Lösung für solche Probleme entwickelt.

Das Decorator Pattern stellt die Möglichkeit zur Verfügung, Klassen mit Objekten anderer Typen zu "dekorieren". Nehmen wir das Beispiel Kaffeehaus: Ein Kunde möchte zu seinem Kaffee auch noch Milchschaum und Schokostreusel. Hier wird der Kaffee als Supertyp gesehen und dann mit den jeweiligen Klassen verknüpft,
Dies verringert den Wartungsaufwand sehr, denn mögliche Änderungen betreffen nicht mehr alle Klassen des Programms.

UML:



Das Klassendiagramm des Decorator Pattern zeigt ganz klar das die zusätzlichen Optionen von der Klasse Decorator erben, welches wiederum das Interface CommonInterface implementiert. die DoIt() Methoden in den beiden Optionen werden von der superklasse angeboten
Ein neues Prinzip beim Decorator Pattern ist das Offen/Geschlossen Prinzip

Composite Pattern[2]

Das Kompositum-Entwurfsmuster (engl. Composite-Pattern) zählt zu der Klasse der Strukturmuster (Structural Patterns). Der Grundgedanke dieses Musters besteht darin, aus einer Grundmenge von primitiven Objekten, mittels geschickter Kombination und Verkettung, komplexe Strukturen zu erstellen. Dabei wird versucht, Objekte zu Baumstrukturen zusammenzufügen, um sogenannte Teil-Ganzes-Hierarchien darzustellen. Dabei geht man im Allgemeinen von einer abstrakten Oberklasse -

der Komponente - aus, welche dem Klienten eine einheitliche Schnittstelle mit vordefinierten Methoden zur Verfügung stellt. Diese Methoden werden später an die Subklassen, nämlich die Konkreten Komponenten und die Kompositionen weiter vererbet. Dies ermöglicht dem Klienten, sowohl einzelne Objekte, als auch deren Kompositionen einheitlich zu behandeln. Das Kompositionsmuster gehört außerdem zu den sogenannten GoF- (Gang of Four) Mustern.

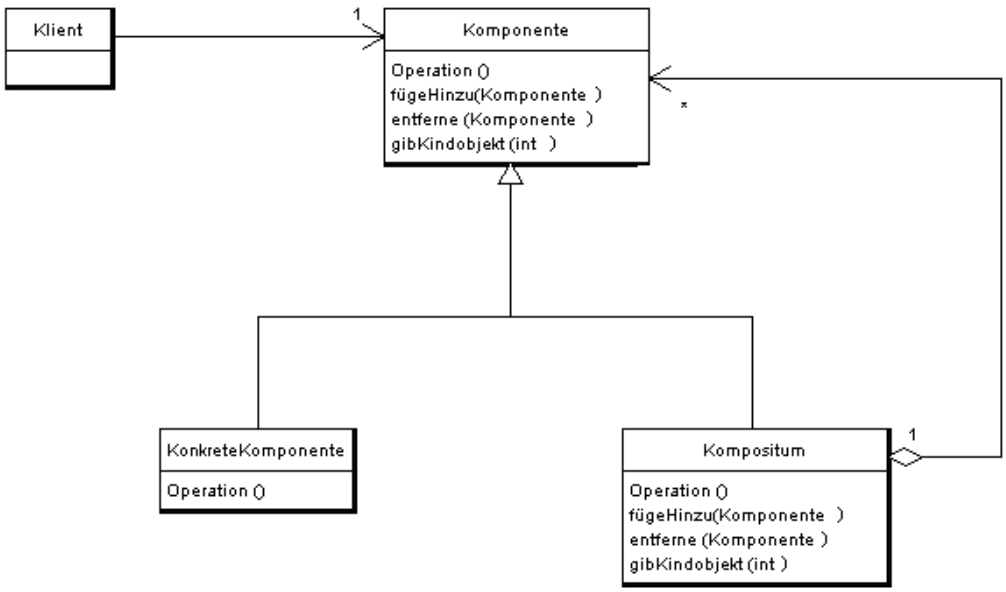
Anwendung

- Implementierung von Teil-Ganzes-Hierarchien
- Wenn der Klient Objekte einheitlich behandeln soll, unabhängig ob er einzelne oder zusammengesetzte Objekte benutzt

Konkrete Beispiele für die Umsetzung dieses Musters

- hierarchische Dateisysteme, speziell in Dateimanagern oder bei Filebrowsern als Dateien und Verzeichnisse
- das in Java verwendete Abstract Window Toolkit (AWT), welches die Möglichkeit bietet, durch Zusammensetzung einfacher Klassen, d.h. durch die Komposition einzelner Grafikelemente, eine komplette Oberfläche zu schaffen

UML:



Factory Pattern[3]

Das Problem mit new

Im Prinzip gibt es kein Problem mit dem 'new'- Operator. Allerdings wiederspricht er unserem Prinzip programmieren auf Interfaces nicht auf Implementierungen.
durch aufrufen von 'new' erzeugt man in seiner Klasse eine Instanz auf eine Konkrete Klasse. Code der viele konkrete Klassen verwendet verursacht Probleme beim hinzufügen von neuen konkreten Klassen.

Daher werden Dinge die sich ändern gekapselt voneinander verarbeitet
Der 'new' Operator wird in eine extra Klasse gepackt die einfach erstellte Referenzen zurückliefert

Simple Factory

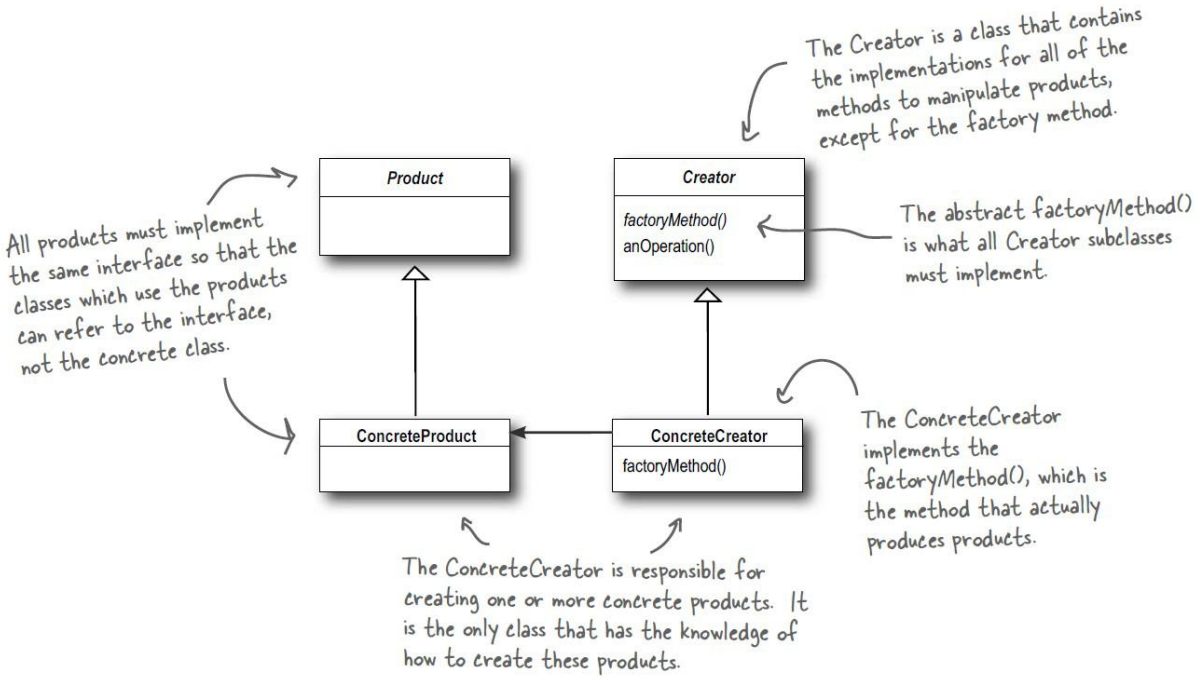
Die Simple Factory ist kein Pattern bringt aber einige Verbesserungen im Vergleich zu Hardcoding. Man verschiebt die Probleme zwar nur in ein anderes Objekt, die Erweiterung des Codes wird dadurch aber um einiges einfacher . Die Konkrete Instantierung wird ebanfalls aus dem Client- Code entfernt.

Factory Method

Die Factory Method ist ein Pattern, welches die Unterklassen entscheiden lässt was zu tun ist.
Definition(aus dem Foliensatz übernommen, da es eine Sehr gute und Verständliche Definition ist):
Die Factory Method Definiert eine Schnittstelle zur Erstellung eines Objekts, lässt aber die Unterklassen entscheiden, welche Klassen instantiiert werden. Factory Method ermöglicht einer Klasse, die Instantiierung in Unterklassen zu verlangen.

Sollte es im Programm mehrere Variationen eines Bestimmten Objektes geben, werden der Typ des Objektes an die Unterklassen übergeben. Die Unterklassen entscheiden dann: Passt dieser Typ zu mir? wenn ja wird er verarbeitet wenn nein ignoriere ich den Befehl.
Klassendiagramm(ebenfalls aus dem Foliensatz übernommen)

UML:



Ohne einer Factory würde der Code unübersichtlich und schwer wartbar werden, da alle instantierungen in einer Klasse stattfinden.

Neues Prinzip:

Umkehrung der Abhängigkeiten.

Je höher eine Komponente im Programm steht desto weniger soll sie von den unteren Komponenten abhängen. Hier soll man sich mehr auf Abstraktion stützen.
Ziel dieses Prinzips soll es sein Abhängigkeiten von konkreten Klassen zu reduzieren

Richtlinien für dieses Prinzip:

- Keine Variable soll eine Referenz auf eine Konkrete Klasse halten ('new' vermeiden)
- Keine Klasse soll von einer konkreten Klasse abgeleitet sein
- Keine Methode sollte eine implementierte Methode einer ihrer Basisklassen überschreiben

Würde man sich an diese Richtlinien halten hätte man keinen Code.

Abstract Factory

Definition :
Bietet eine Schnittstelle zum Erstellen von Familien verwandter oder zusammenhängender Objekte an, ohne konkrete Klassen anzugeben.

Observer Pattern[4]

Das Observer- Pattern übernimmt den Part des Aufpassers und des Nachrichtendienstes im Programm. Sollten sich Objekte ändern werden alle Objekte benachrichtigt, die etwas mit dem geänderten Objekt zu tun haben.

Alle Klassen die überwacht werden sollen werden beim Observer registriert. Sollte sich nun etwas bei registrierten Klassen ändern wird der Observer benachrichtigt und setzt den Status der Objekte neu.

Weiters findet hier das Prinzip der losen Kopplung Verwendung.
Die Klassen sind locker gebunden. Soll heißen es findet eine Interaktion der Objekte statt ohne ein großes Detailwissen über die anderen Klassen zu haben.

Lockere Kopplung zwischen Subjekt und Beobachter:
Das Subjekt kennt nur die Beobachter- Schnittstelle.
Das Subjekt muss für neue Beobachter nicht verändert werden.
Das Subjekt und der Beobachter sind unabhängig voneinander verwendbar.

Hier gibt es wieder Interfaces. Observable und Observer.
Observable wird vom Subject implementiert, Observer vom Konkreten Observer.

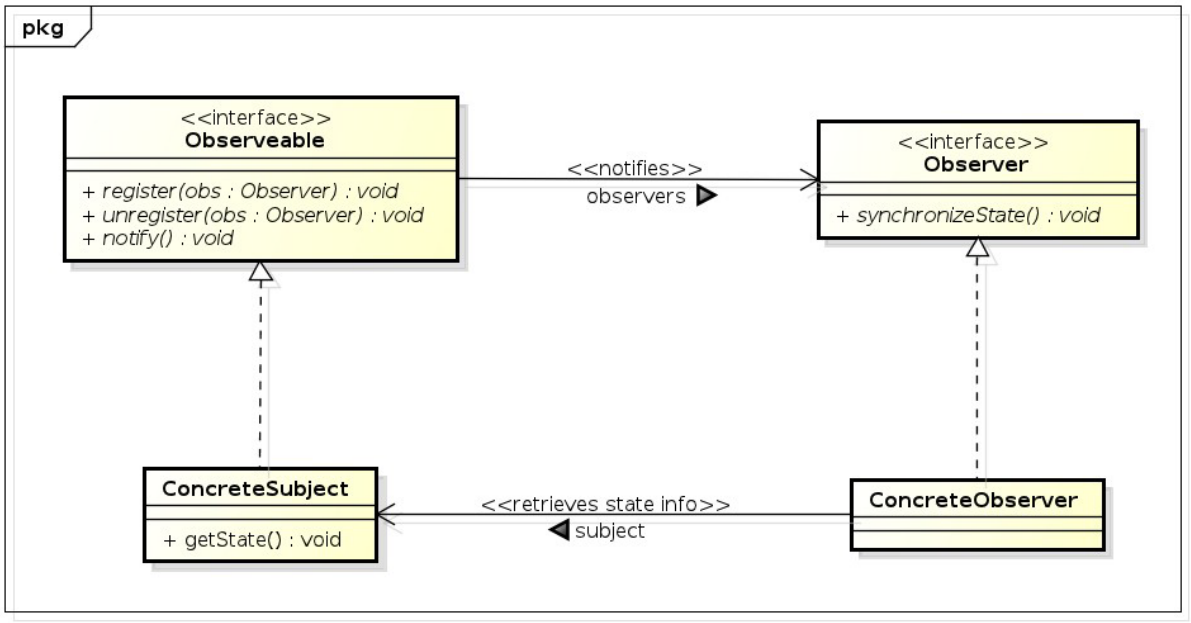
Sollte das Subjekt bereits beim Observer registriert sein und eine Zustandsänderung stattfinden, wird über das Observable- Interface eine Benachrichtigung an den Observer übergeben. Dieser ändert

sofort den Status und alle Klassen die beim Observer registriert sind werden über die Änderung benachrichtigt.

Der Observer findet auch bei der Gestaltung von Grafischen Oberflächen Verwendung.
Der ActionListener eines Buttons ist ebenfalls ein Observer, welcher Statusänderungen an der GUI durchführt oder Funktionalität im Hintergrund startet.

Das Observable Interface implementiert alle Klassen die benötigt werden um sich bei dem Observer zu registrieren.

UML



- Lose Kopplung zwischen Objekten die miteinander kommunizieren.

Iterator Pattern[5]

Als Iterator (auch als Cursor bekannt) bezeichnet man in der Softwareentwicklung ein spezielles Entwurfsmuster, welches zur Kategorie der Verhaltensmuster (Behavioural Patterns) gehört und von der "Gang of Four" (GoF) eingeführt wurden ist. Es stellt eine Art Zeiger dar, mit dem über Elemente einer Menge, häufig auch als Aggregat bezeichnet, iteriert werden kann.

Im Gegensatz zu einem Index oder Schlüssel kann man mit einem Iterator direkt auf das zugehörige Element zugreifen, ohne dabei die Datenstruktur selber zu kennen, wobei bei einem Index/Schlüssel immer der jeweiligen Index und die Datenstruktur benötigt wird. Eine Instanz eines Iterators ist nur

für genau eine Datenstruktur gültig. Ein Index jedoch kann auch auf andere Datenstrukturen übertragen werden.

Kategorien

Input

Diese iteratoren bieten einen lesenden Zugriff an. Sobald eine Element aus dem vorgesehenen Bereich gelesen wurde, kann der Iterator nicht mehr zurückgehen. Um auf vergangene Elemente zurückzuspringen, muss der komplette Bereich von vorne gelesen werden

Output

Diese Iteratoren bieten nur einen Schreibzugriff. Auch diese sind nur für den einfachen Durchgang konzipiert

Forward

Diese Iteratoren bieten eine Kombination aus Input- und Output- Iteratoren an.
Dies ermöglicht lesende und schreibende Zugriffe zu gleich.
Ausserdem können sie mehrmals den vorgesehenen Elementbereich durchiterieren.

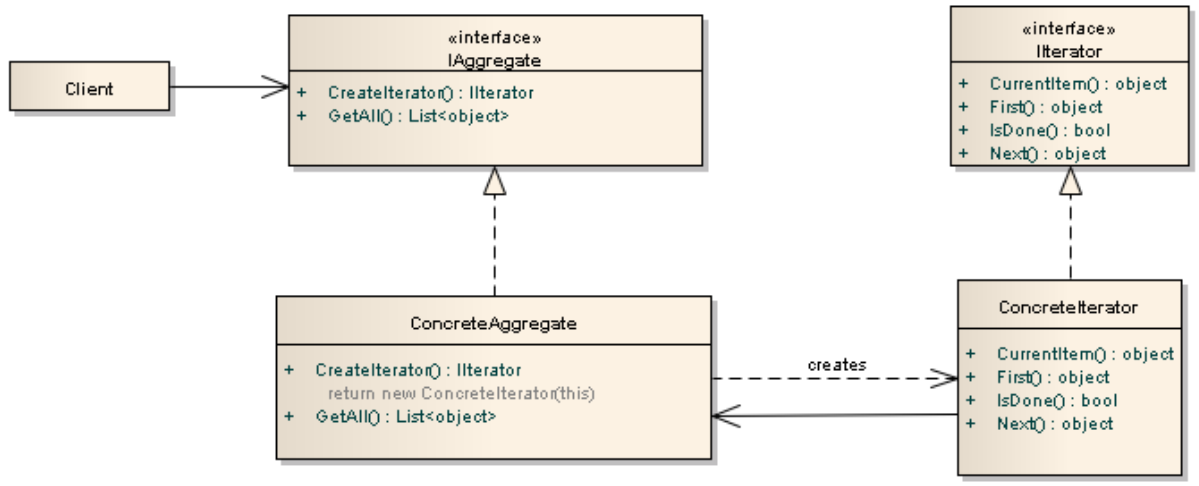
Bidirectional

Hierbei handelt es sich um Forward- Iteratoren mit der Fähigkeit, den vorgesehenen Elementbereich auch rückwärts durchgehen zu können.

Random Access

Als echte zeigerähnliche Objekte weisen sie ähnliche Fähigkeiten wie die bidirektionalen Iteratoren auf und beherrschen zudem die Zeiger-Arithmetik.

UML:

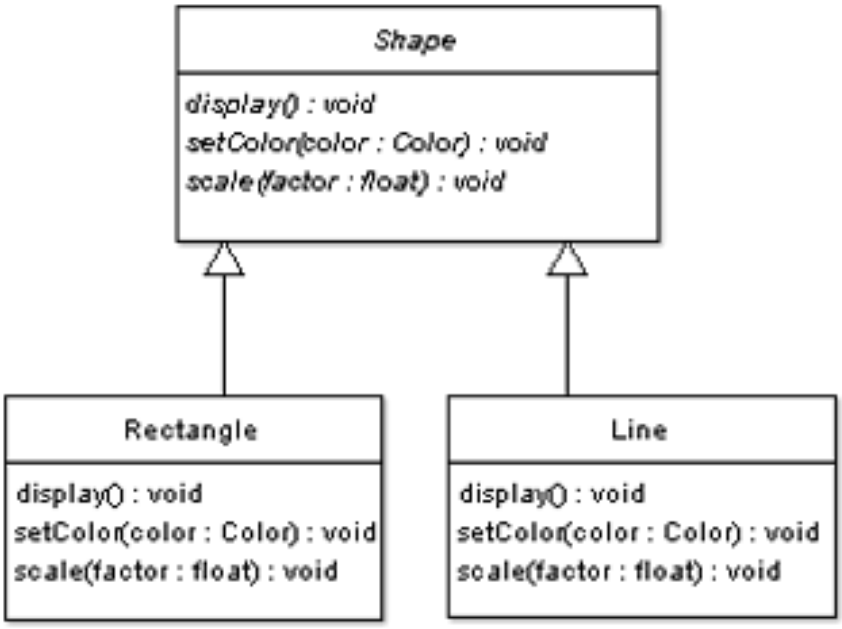


Adapter Pattern[6]

Greift ein Client auf eine andrere Klasse zu, so erwartet er sich eine bestimmte Schnittstelle nach außen. Hierbei kann es nun vorkommen, dass diese Klasse zwar die benötigten Funktionalitäten zur Verfügung stellt, aber die erwartete Schnittstelle nicht besitzt.

An diesem Punkt kommt das Adapter Pattern ins Spiel. Dieses Pattern erlaubt es, unterschiedliche Klassen trotz "inkompatibler" Schnittstellen zusammenarbeiten zu lassen. Ein gängigerer Begriff zu diesem Pattern ist der Wrapper.

UML:



Literaturverzeichnis

[1] Foliensatz zum Decorator Pattern des 4. Jahrgangs
[2] http://www.imn.htwk-leipzig.de/~weicker/pmwiki/pmwiki.php/Main/Composite-Pattern
[3] Foliensatz zum Factory Pattern des 4. Jahrgangs
[4] Foliensatz zum Observer Pattern des 4. Jahrgangs
[5] http://www.imn.htwk-leipzig.de/~weicker/pmwiki/pmwiki.php/Main/Iterator-Pattern
[6] http://magazin.c-plusplus.de/artikel/Einf%FChrung%20in%20Design%20Patterns