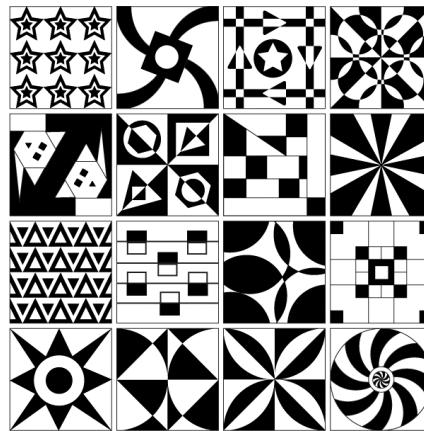




Media Engineering Design Patterns

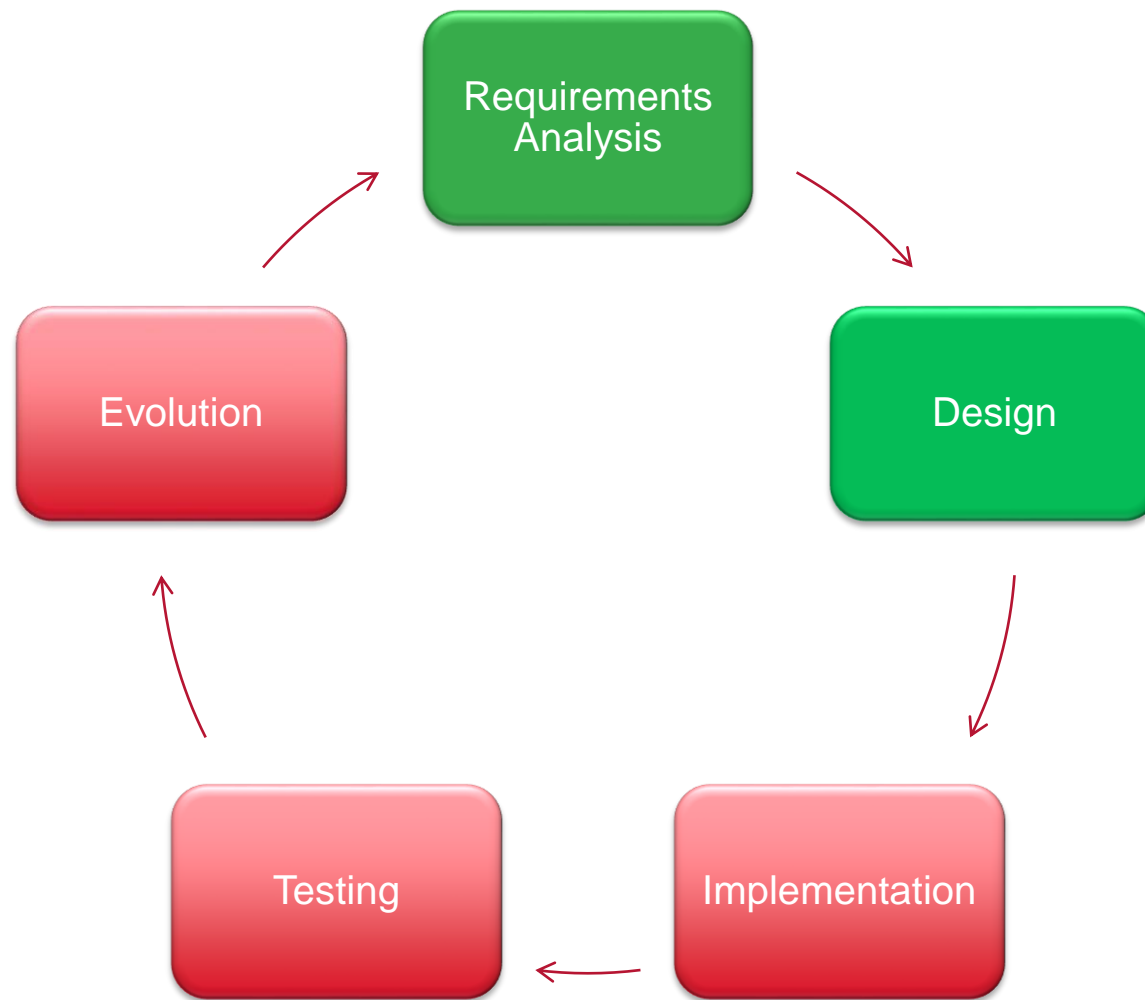


R. Weller

University of Bremen, Germany

cgvr.cs.uni-bremen.de

Der Software Development-Lifecycle



Wo stehen wir?

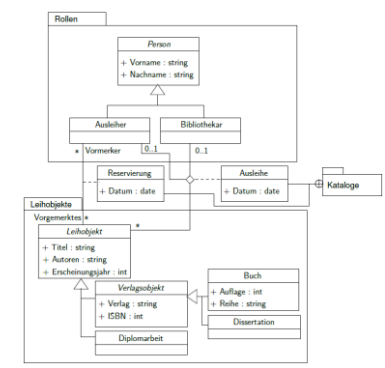
- Wir wissen **was** wir machen wollen
 - Requirements Engineering => Pflichtenheft



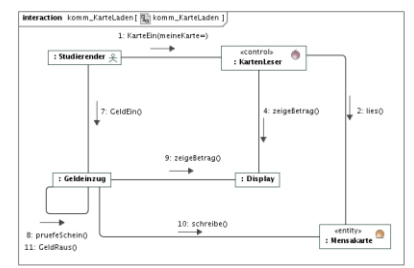
- Wir wissen **wie** es **aussehen** soll
 - GUI-Design => Prototypen



- Wir kennen die statische **Struktur**
 - Objektorientiertes Design => Strukturdiagramme



- Wir wissen, **wie** es sich **verhalten** soll
 - Objektorientiertes Design => Verhaltensdiagramme



Also endlich?



Design Patterns

- 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.

Christopher Alexander (Architekt und Mathematiker),
“A pattern language”, 1977



- Definitionsversuch
 - **Design Pattern** (auch Entwurfsmuster genannt) sind **Musterlösungen** für **wiederkehrende Entwurfsprobleme**.
- 1994 übertragen Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides das Prinzip der Design Pattern von Christopher Alexander auf den Softwareentwurf.
 - Das Buch "*Design Patterns. Elements of Reusable Object-Oriented Software*" gilt nach wie vor als Standardwerk



Was Design Pattern sind bzw nicht sind

- Design Pattern verwendet man in der Design-Phase
 - Recht hoher Abstraktionsgrad
 - Unabhängig von Programmiersprache und Implementierung
 - Unabhängig von Vorgehensmodellen (SCRUM, Wasserfall, RUP,...), aber dazu später mehr
 - Grob gesagt: Beschreiben eher das Verhalten von und zwischen Objekten, und nicht die Implementation eines einzelnen Objekts
- Was Design Pattern nicht sind
 - Datenstrukturen (Listen, Hash-Tables, binäre Bäume,...)
 - Algorithmen (binäre Suche, Sortieren,...)

- **Name**
 - Möglichst prägnant, also kurz und beschreibend
- **Problem**
 - Definiert, **was** das Entwurfsmuster löst
- **Lösung**
 - Beschreibt, **wie** das Entwurfsmuster das Problem löst
- **Konsequenzen**
 - Beschreibt Folgen und Kompromisse des Musters

Ein einfaches Beispiel

- Erinnern wir uns an das Bibliothekbeispiel aus dem OOAD-Kapitel
- Dort gab es eine Klasse „Bibliothek“, in der alle Bücher stehen

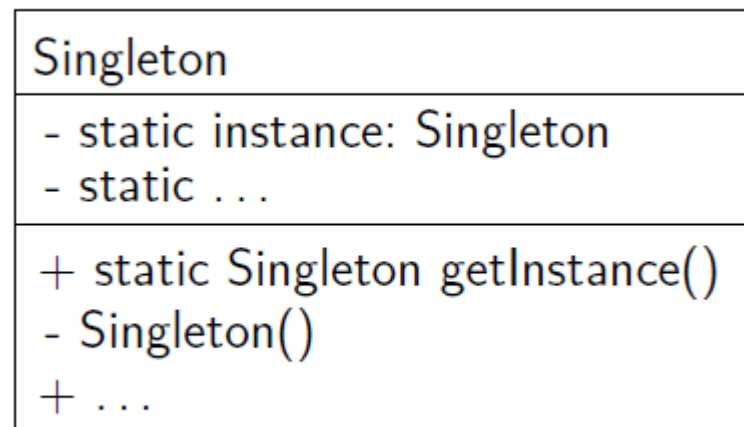


- Von dieser Klasse soll es nur **eine einzige** Instanz (Objekt) geben.
- Wie kann man das erreichen?

Design Pattern: Singleton

- **Name:** Singleton
- **Problem:** Stellt sicher, dass es von einer Klasse nur eine Instanz gibt
- **Lösung:**
 - Konstruktor privat machen
 - Statisches Attribut speichert die einzige Instanz
 - Statische Methode liefert einzige Instanz zurück

- In UML:



```

class Singleton
{
    public:
        static Singleton* getInstance();

    protected:
        Singleton() {}

    private:
        static Singleton *instance;
};

Singleton* Singleton::instance = 0;

Singleton* Singleton::getInstance()
{
    if( instance == 0 )
        instance = new Singleton();
    return instance;
}

```

```

//Nicht:
Bibliothek *bib = new Bibliothek();
//Funktioniert nicht, da der Konstruktor privat ist =>
Compilerfehler

//Stattdessen:
Bibliothek *bib = Bibliothek::getInstance();
    
```

■ Konsequenzen:

■ Vorteile im Vergleich zu globalen Variablen:

- Bietet Zugriffskontrolle
- Kann durch Unterklassen spezialisiert werden
- Leichte Änderung, wenn man doch mehrere Objekte benötigt

■ Nachteile

- Abhängigkeiten werden verschleiert
 - D.h., man sieht nicht direkt am Interface einer Klasse, ob sie ein Singleton ist, sondern nur an der internen Implementierung
- Einzigartigkeit wirklich sicherzustellen kann schwierig sein
 - Verteilte Umgebungen
- Weniger performant als globale Variablen

Ein weiteres Beispiel: Undo/Redo

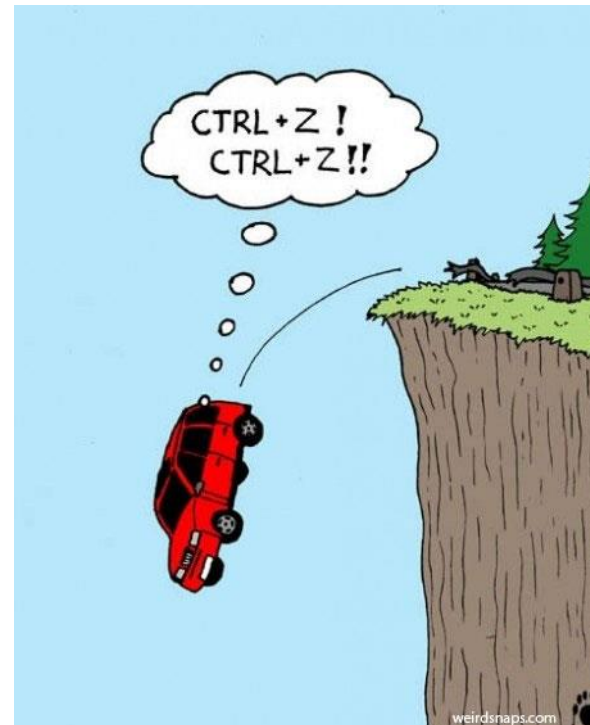
- Zurücknahme einer oder mehrerer Aktionen
- Oft mit einer Redo-Funktion kombiniert, die vorangehendes Undo aufhebt

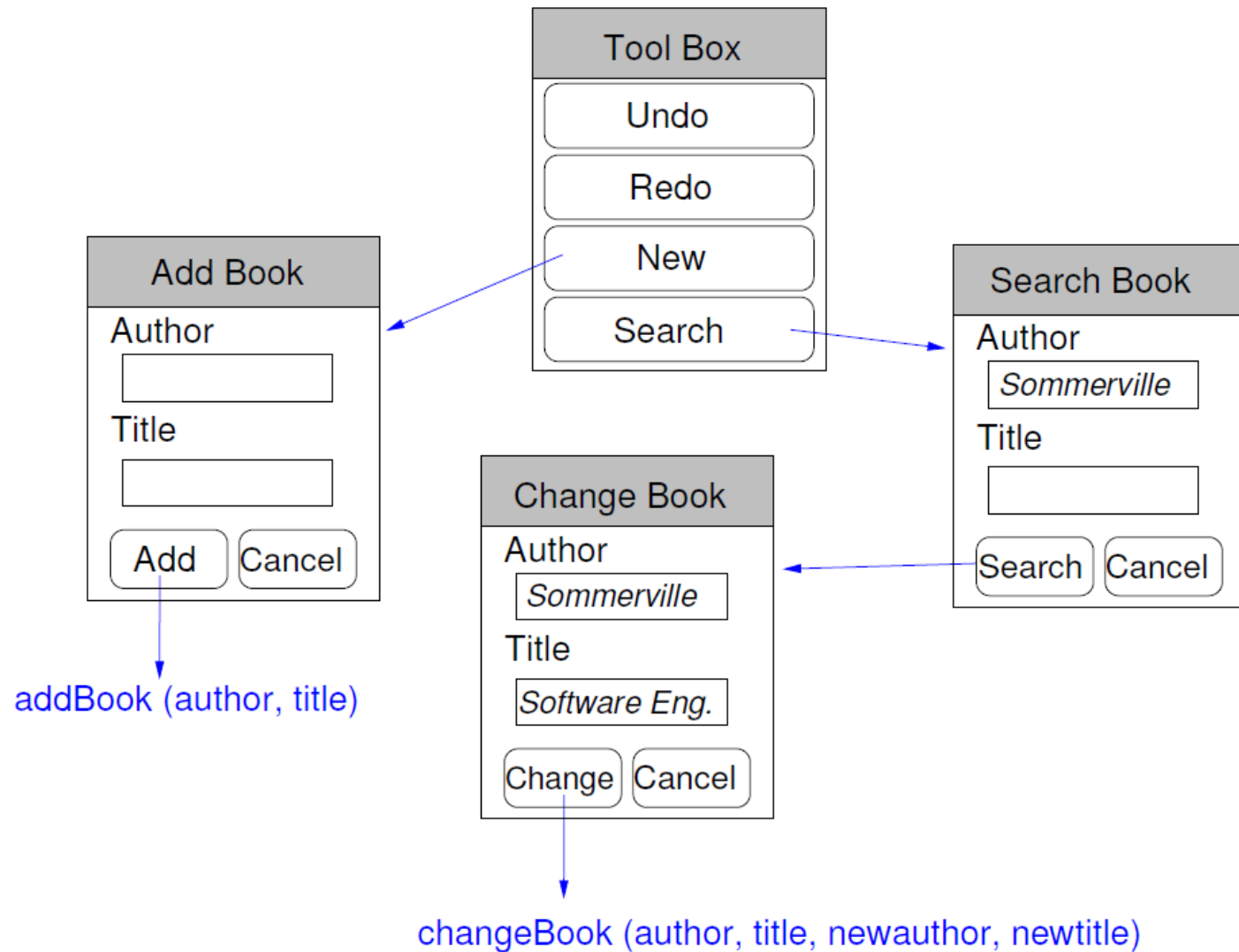


- Historisches:
 - Erstmals vorgeschlagen 1977 von Lance A. Miller (IBM)
 - Die Belegung von Ctrl + Z wurde vom Xerox PARC eingeführt

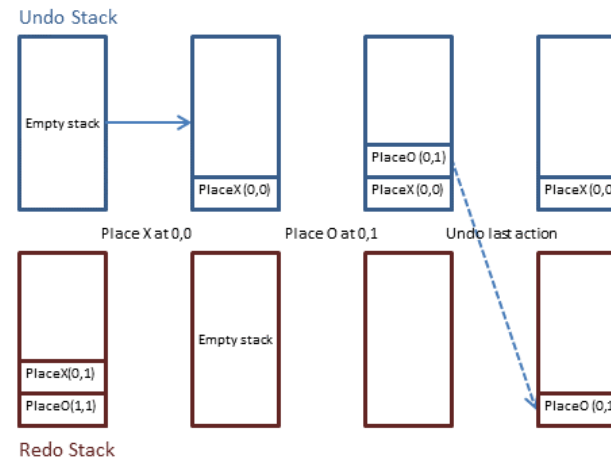
Undo/Redo Basisfunktionalität

- Aufzeichnung aller ausgeführten Benutzerinteraktionen
 - Inklusive der Argumente
- Bereitstellung einer Umkehraktion für jede mögliche Aktion

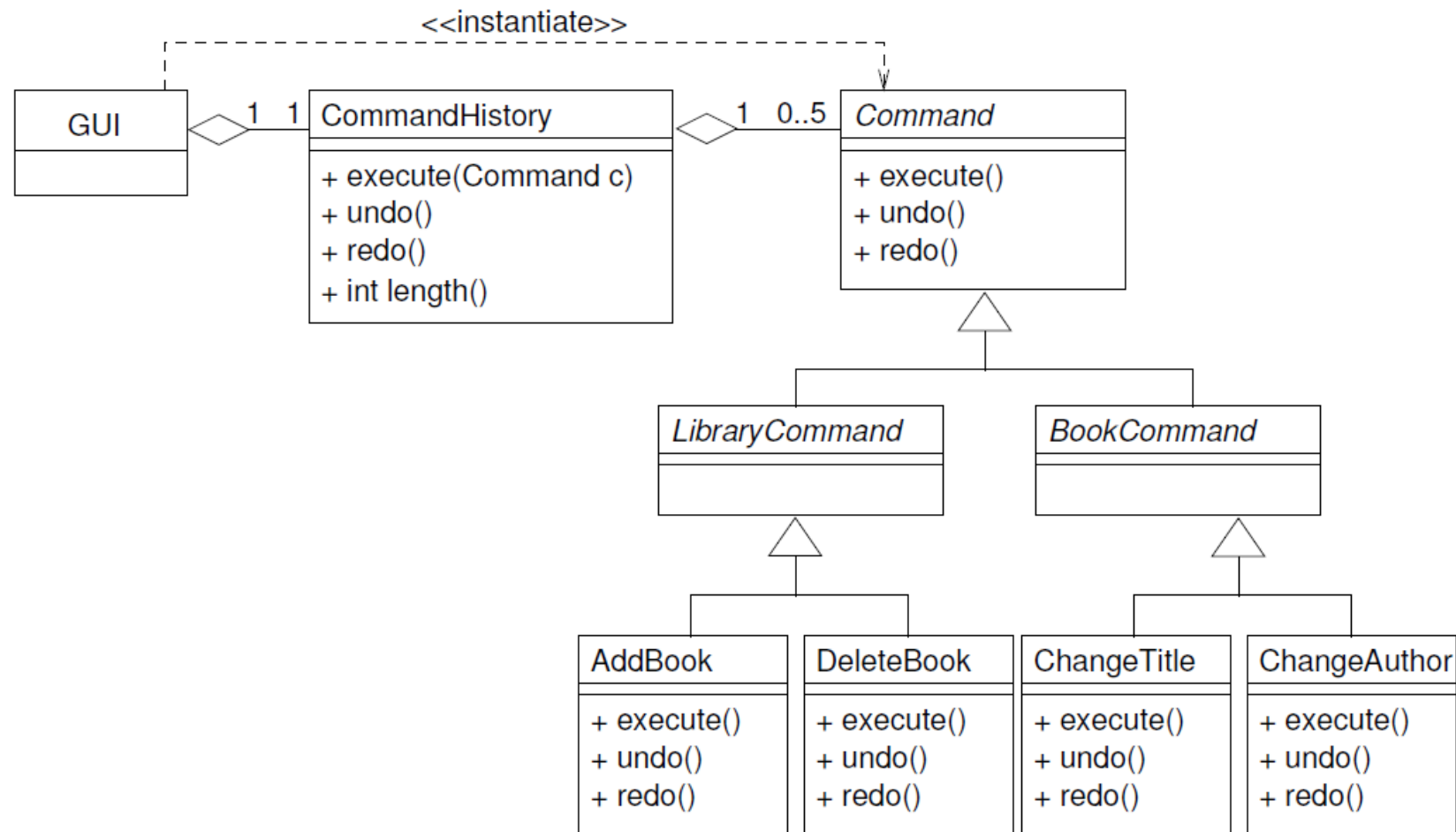




- Undo-Mechanismus
 - Speichere bei jeder Aktion alle nötigen Daten im Objekt
 - Sichere das Objekt auf einem Stapel
- Redo-Mechanismus
 - Verwende einen zweiten Stapel für die rückgängig gemachten Befehle



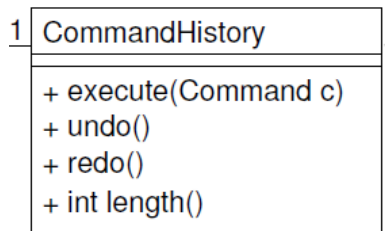
- Grundidee: Repräsentiere Befehle durch Objekte!



Implementierung in C++ - CommandHistory

```
// a bounded history of commands
class CommandHistory
{
    // upper bound of commands
    private static int maxCommands = 5;
    //primitive command stack
    private Command [] commands = new Command[maxCommands];
    //the current command in the stack
    private int current = -1;

    //adding commands to the history and execute
    public void execute( Command command )
    {
        current++;
        //in case of full history shift
        if( current == maxCommands )
        {
            for( int i = 0; i < maxCommands-1; i++ )
            {
                commands[i] = commands[i+1];
            }
            current = maxCommands - 1;
        }
        commands[current] = command;
        commands[current].execute();
    }
}
```

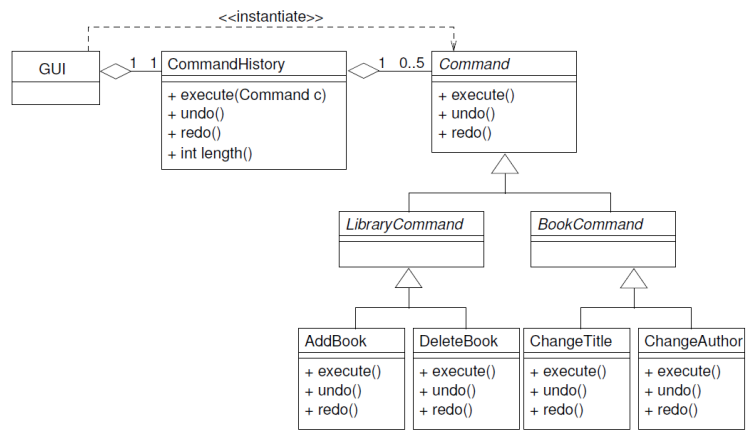


Implementierung in C++ - CommandHistory

```

// a bounded history of commands
class CommandHistory
{
    ...

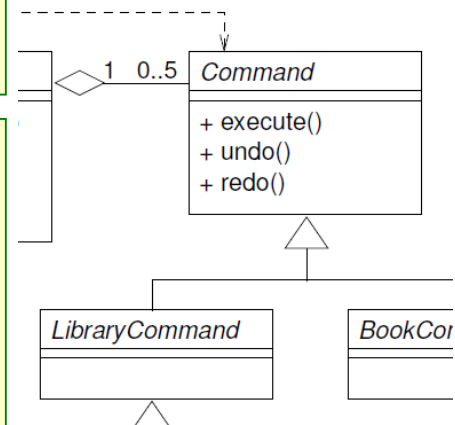
    //undo for current command
public void undo ()
{
    commands[current].undo ();
    current--;
}
}
    
```



```
// super class of all commands
class Command
{
    public virtual void execute() = 0;
    public virtual void undo () = 0;
}
```

```
//super class of all commands that manipulate the library
class LibraryCommand : public Command
{
    protected String author; //command argument
    protected String title; //command argument
    protected Library receiver; //receiver of command

    protected void init( String author, String title,
Library receiver )
    {
        this.author = author;
        this.title = title;
        this.receiver = receiver;
    }
}
```



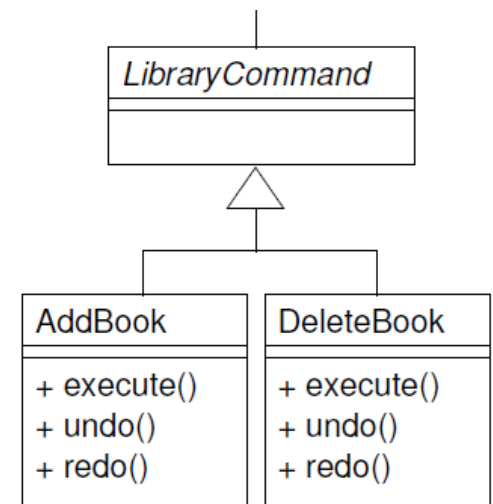
```

//command class to add new book to library
class addBook : public LibraryCommand
{
    //Constructor
    public AddBook( String author, String title,
Library receiver )
    {
        init( author, title, library );
    }

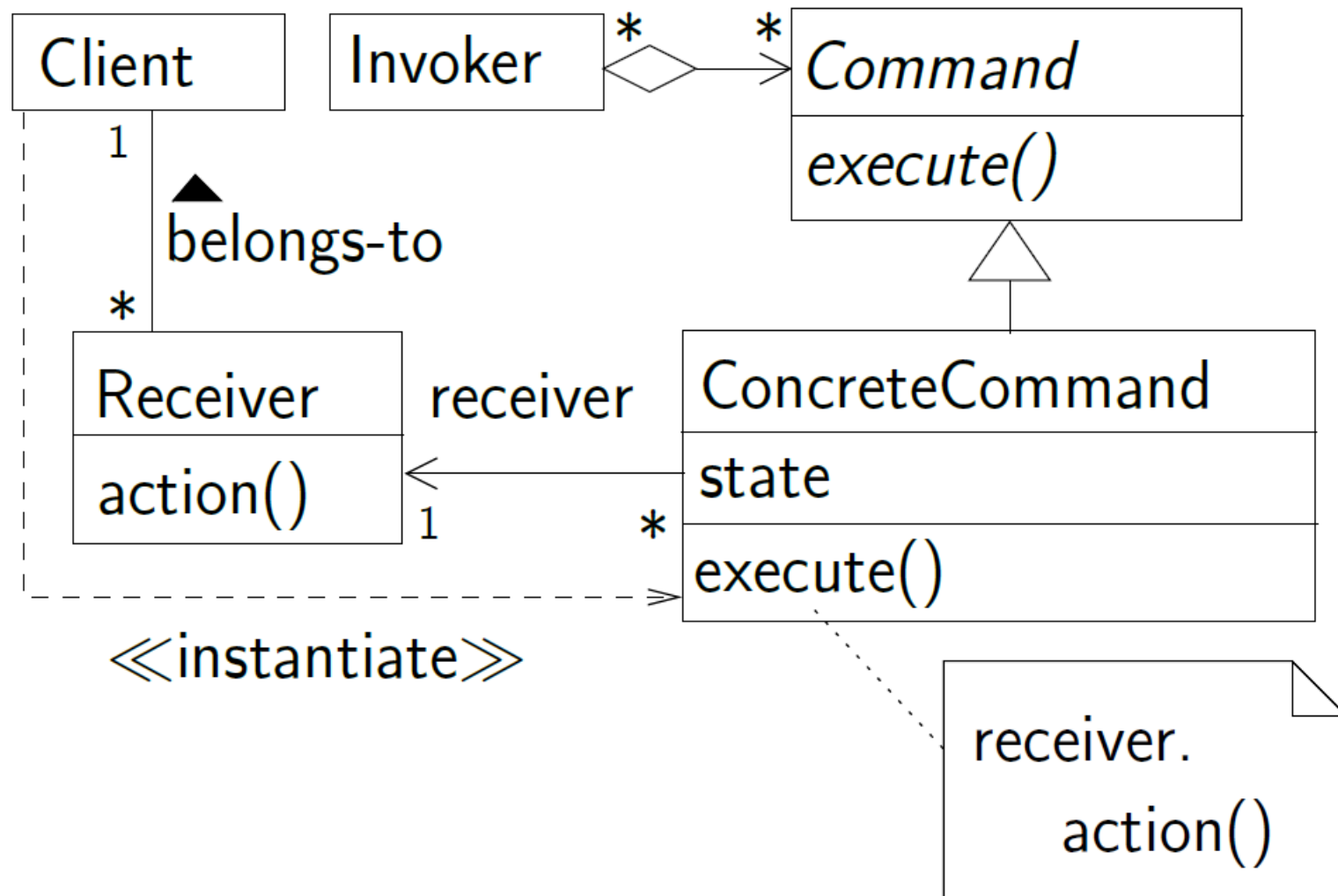
    //Overriden execute function
    public void execute()
    {
        receiver.add( author, title );
    }

    //Override undo function
    public void undo()
    {
        receiver.delete( author, title );
    }
}

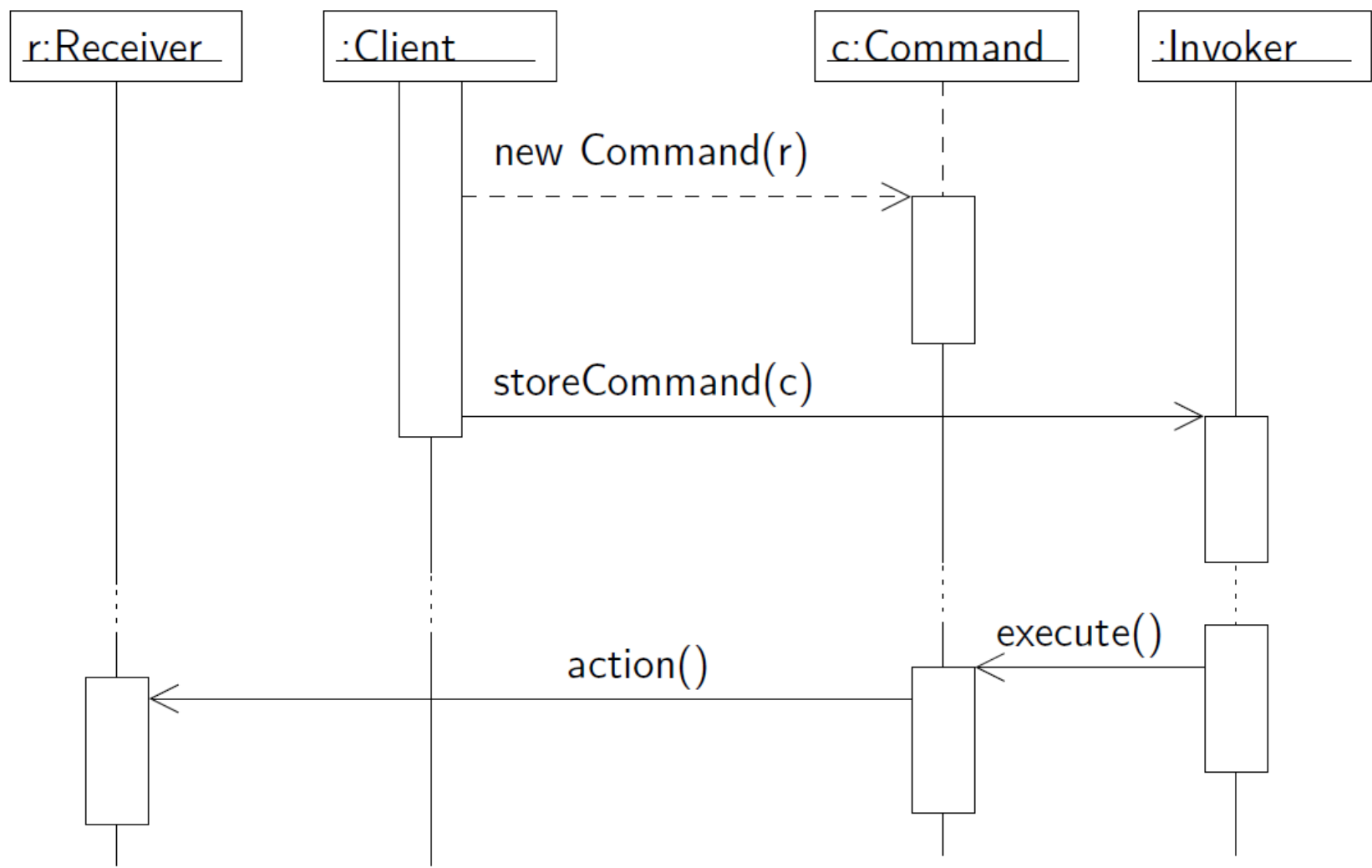
```



Der Command-Pattern im Allgemeinen



Command-Pattern Sequenzdiagramm



■ Konsequenzen:

■ Vorteile:

- Auslösender und Ausführer sind entkoppelt
- Befehle sind Objekte und können wie diese auch verwendet werden (Zwischenspeichern, Hierarchie (Vererbung), Filtern,...)
- Einfache Befehlsobjekte können leicht zu komplexeren Befehlen kombiniert werden (Makros)

■ Nachteile

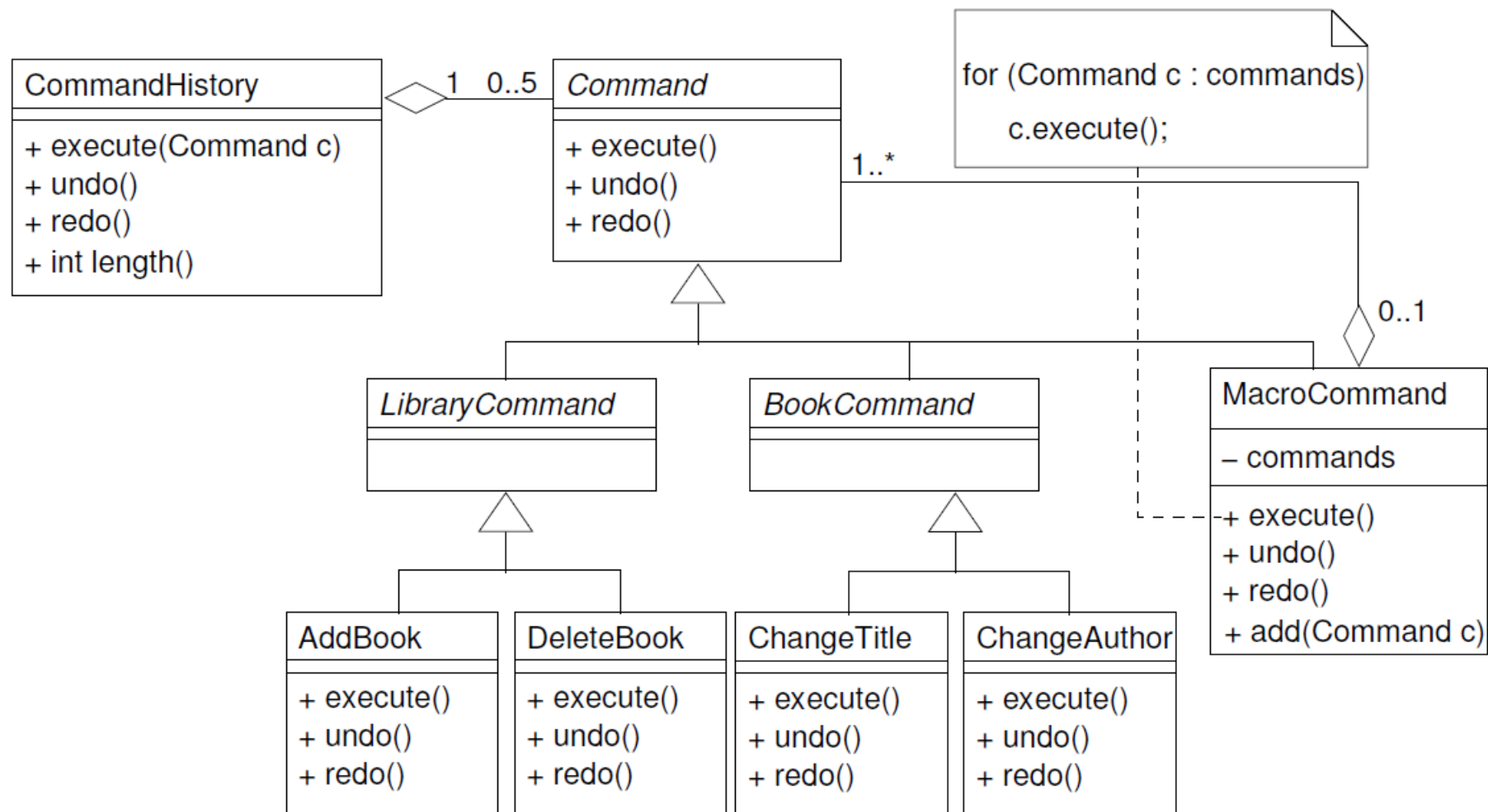
- Für jeden Befehl muss eine neue Klasse erstellt werden

Beispiel 3: Skriptsprache

- Motivation: Manchmal möchte man komplexere Befehle ausführen, die aus Basisbefehlen zusammengesetzt sind
 - Beispiel: Suche alle Bücher eines bestimmten Autors und lösche sie

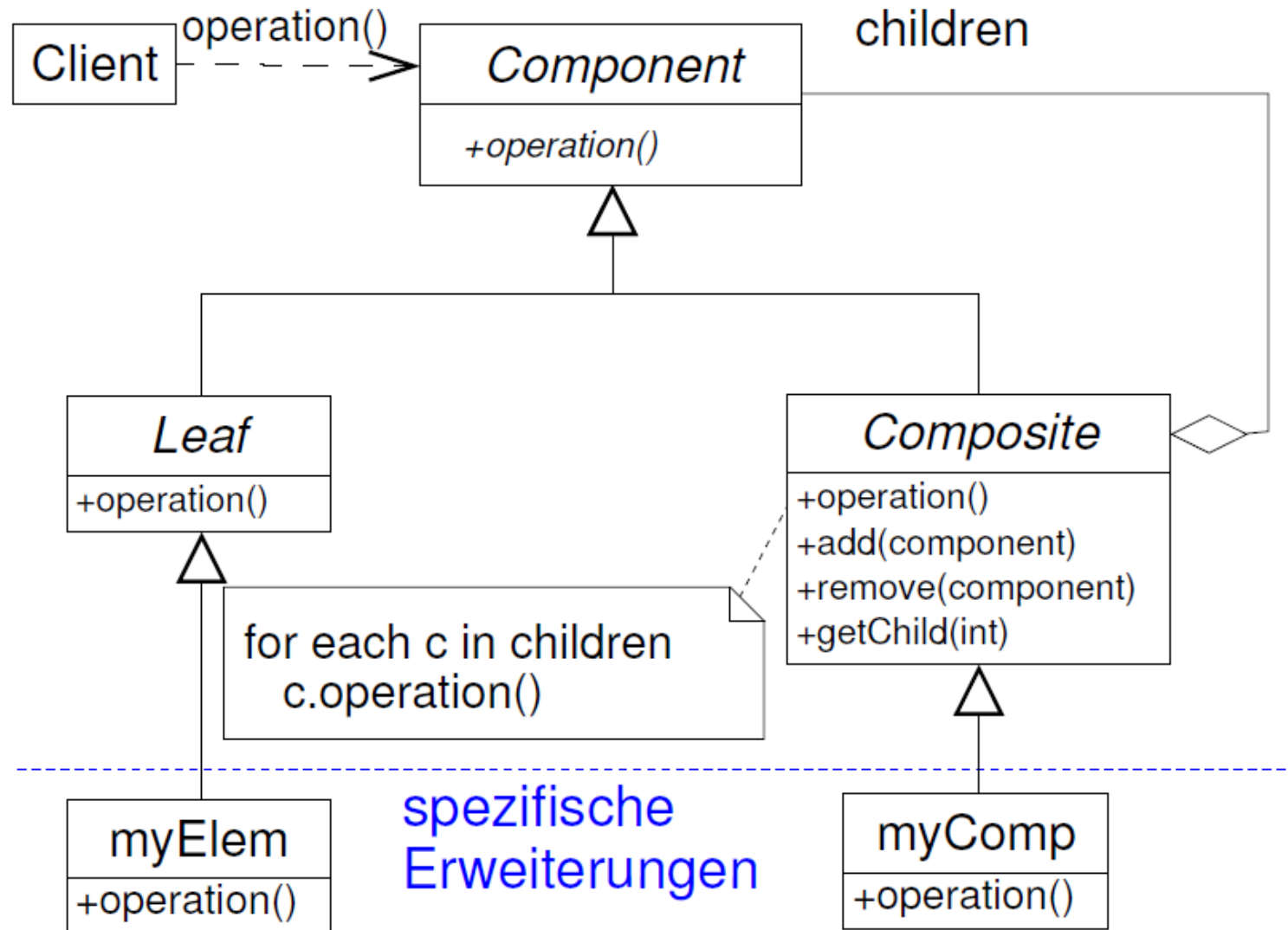


- Wir wollen zusammengesetzte Kommandos
 - Erweiterung um Befehlsstrukturen
 - If-Anweisungen
 - Schleifen
- Undo/Redo soll weiterhin funktionieren
 - Die CommandHistory soll sich nicht damit auseinandersetzen müssen, ob sie ein zusammengesetztes Kommando oder ein einzelnes Kommando bearbeitet



- **Name:** Composite
- **Problem:** Beschreibt eine Teil-von-Hierarchie mit einheitlicher Schnittstelle
 - Überall wo ein Ganzes eingesetzt werden kann, kann auch nur ein Teil verwendet werden und umgekehrt
 - Verbirgt Unterschiede zwischen einzelnen und zusammengesetzten Objekten
- **Lösung:**
 - (abstrakte) Basisklasse definiert gemeinsames Verhalten aller Teilnehmer
 - Leaf-Klasse repräsentiert Einzelobjekt (z.B. atomares Kommando)
 - Composite-Klasse repräsentiert Mengen von Blättern und anderen Komposita

Klassendiagramm: Composite-Pattern



Implementierung Composite-Pattern

- Client: manipuliert Objekte über die gemeinsame Schnittstelle von Composite
- Component
 - Deklariert einheitliche Schnittstelle
 - Optional: Standardverhalten

```
// super class of all components
class Component
{
    public virtual void operation() = 0;
}
```

- Leaf
 - Repräsentieren atomare Komponenten
 - Verhalten dieser atomaren Komponenten

```
// leaf class represents atomar operation
class Leaf : Component
{
    public void operation();
}
```

Implementierung Composite-Pattern (cont)

- Composite
 - Speichert Teile
 - Definiert Standardverhalten für zusammengesetzte Komponenten
 - Implementiert Operationen zur Verwaltung von Teilen
 - Hinzufügen
 - Entfernen
 - ...

```
class Composite : Component
{
    //data structure that stores the commands
    private std::set<Component> children;
```



```

...
//the interface for component
public void operation()
{
    for( std::set<Component>::iterator c = children.begin();
        c != children.end(); ++c )
    {
        c->operation();
    }
}

//adding new components
public void add( Component c )
{
    children.insert( c );
}

//removing new components
public void remove( Component c )
{
    children.erase( c );
}
}

```

■ Konsequenzen:

■ Vorteile:

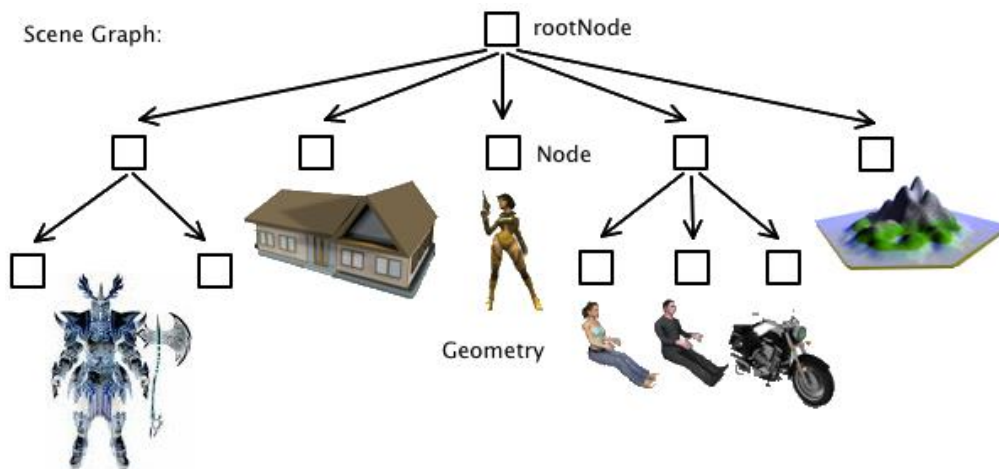
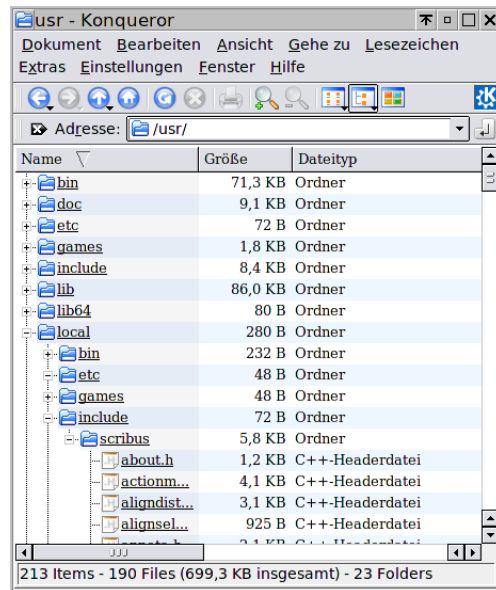
- Einheitliche Schnittstelle für zweigeteilte Klassenhierarchie (Leaf und Composite)
 - => Uniforme Verwendung auf Seiten des Benutzers
- Neue Komponenten können leicht hinzugefügt werden

■ Nachteile

- Sehr allgemeiner Entwurf
 - Oftmals darf nicht jede beliebige Component immer Teil eines Composite sein
 - => Compiler kann nicht prüfen, welche es dürfen und welche nicht
 - => Typüberprüfung zur Laufzeit notwendig

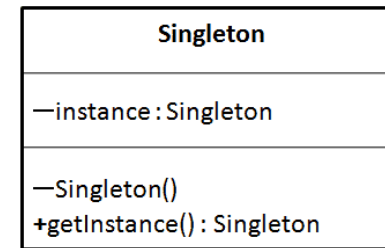
Weitere Anwendungen von Composite

- Skripting-Erweiterung mit Undo/Redo
- Repräsentation hierarchischer Dateisysteme in Dateimanagern
- Bibliotheken zur graphischen Darstellung
 - GUI-Systeme z.B. AWT in Java
 - Viele Szenegraphen



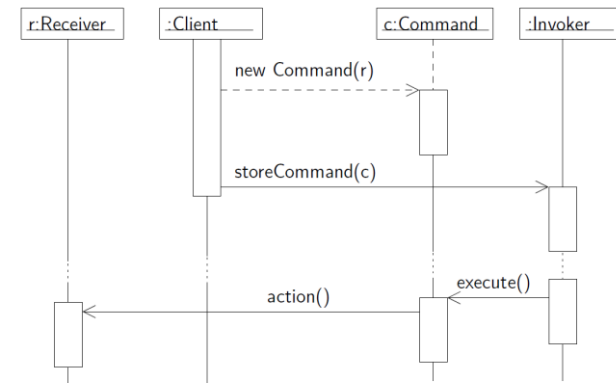
- Singleton

- Bezieht sich auf nur eine Klasse



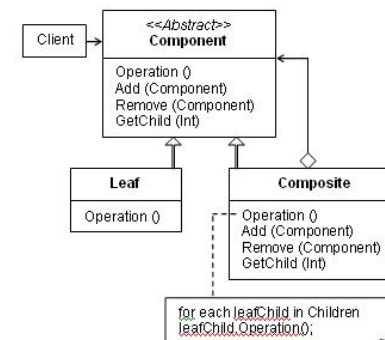
- Command

- Beschreibt die Interaktion verschiedener Klassen miteinander
 - Zur Beschreibung wurden UML-Interaktionsdiagramme benötigt



- Composite

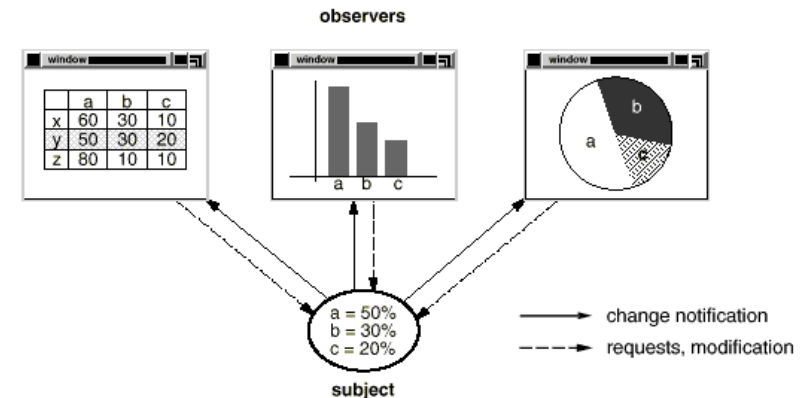
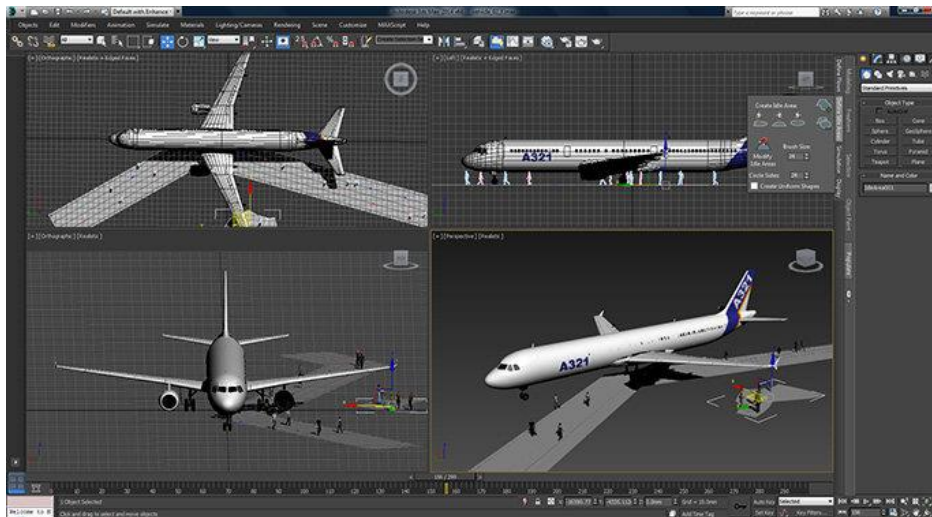
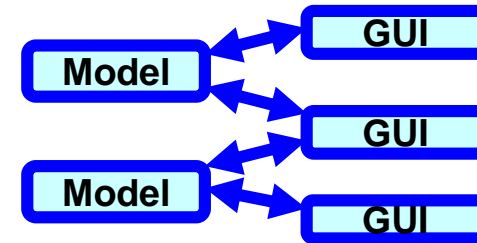
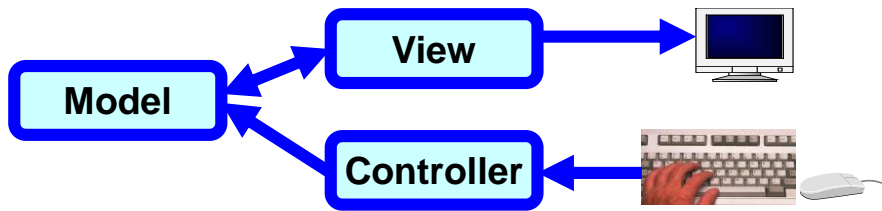
- Beschreiben, wie man mehrere Klassen geschickt anordnet



Kategorien von Entwurfsmustern

- Erzeugungsmuster
 - Betreffen die Erzeugung von Objekten oder Klassen
 - Beispiel: Singleton
- Strukturelle Muster
 - Fassen Klassen und Objekten zu größeren Strukturen zusammen
 - Beispiel: Composite
- Verhaltensmuster
 - Beschreiben Interaktion und Verantwortlichkeiten von Objekten oder Klassen
 - Beispiel: Command

- Zur Erinnerung: Model-View-Konzept



Ein Bibliothek-Beispiel

- In der GUI soll stets die aktuelle Anzahl Bücher in der Bibliothek angezeigt werden
- Beim Hinzufügen oder Löschen eines Buchs muss die Anzeige angepasst werden
- In Zukunft will man eventuell auch noch andere Darstellungen haben
 - Z.B. Balken der Füllstand visualisiert

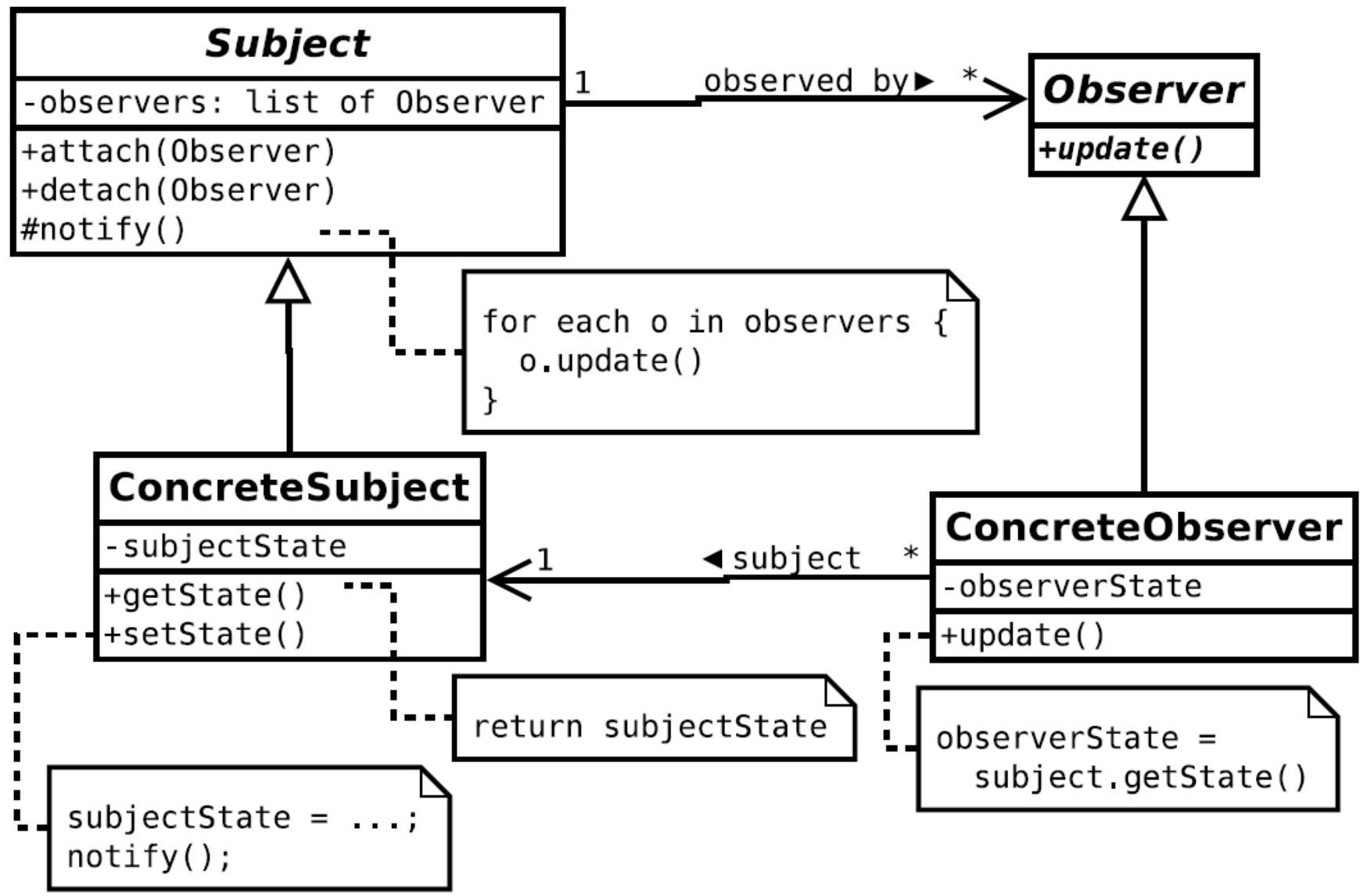


Design Pattern: Observer



- **Name:** Observer (auch publish-subscribe)
- **Problem:**
 - Komponenten hängen von anderen Komponenten ab
 - Änderungen der einen Komponente zieht Änderung der anderen Komponenten nach sich
 - Komponenten sind lose gekoppelt, d.h., Abhängigkeiten sind nicht im Voraus bekannt und ergeben sich erst zur Laufzeit, nicht zur Kompilierzeit
- **Lösung:**
 - Abhängige Komponenten (Beobachter) registrieren sich bei der beobachteten Komponente (Publisher)
 - Publisher informiert alle abhängigen Beobachter über Zustandsänderung

Klassendiagramm Observer Pattern

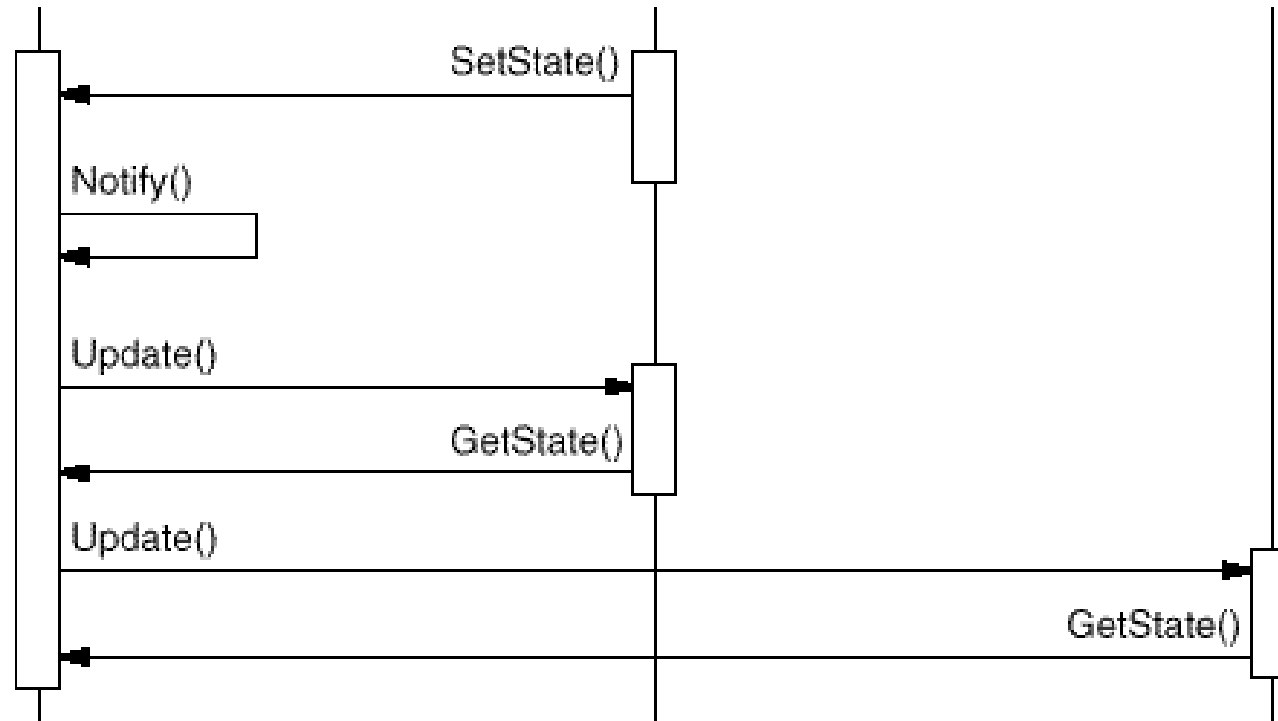


Sequenz-Diagramm Observer Pattern

aConcreteSubject

aConcreteObserver

anotherConcreteObserver



Implementierung Observer-Pattern: Subject

- Kennt alle ihre Observer (zur Laufzeit)
 - Potentiell beliebige viele
- Stellt für Observer Schnittstellen zum registrieren und entfernen zur Verfügung
- Stellt für abgeleitete konkrete Subjects Schnittstelle zur Benachrichtigung der Observer zur Verfügung

```
class Subject
{
    //List of all observers
    private std::list<Observer> list;

    public void attach(Observer *o);
    public void detach(Observer *o);
    protected void notify();
};
```

- Implementierung der Subject-Funktionalität:

```

void Subject::attach(Observer *o)
{
    list.push_back(o);
}

void Subject::detach(Observer *o)
{
    list.erase(o);
}

void Subject::notify()
{
    for(std::list<Observer*>::iterator it = list.begin(); it != list.end();
        ++it)
    {
        (*it)->update();
    }
}

```

Implementierung Observer-Pattern: Observer

- Abstrakte Klasse
- Deklariert Schnittstelle für die Update-Nachricht
 - Wird aufgerufen, wenn sich im Subject was geändert hat und dies dem Observer per notify mitgeteilt wurde.

```
class Observer
{
public:
    virtual void update() = 0;
};
```

Implementierung Observer-Pattern: Library

- Die ConcreteSubject-Klasse
 - Abgeleitet von Subject
 - Hat den beobachteten Wert

```

class Library : public Subject
{
    //attribute to store the number of books
    private int nrBooks;
    //Set with all books in the library
    std::set<Books*> books;

    //setState functions
    //Adds a new book to the library
    public void addBook( String author, string title );
    //Deletes a book from the library
    public void deleteBook( String author, string title );

    //getState function
    public int getNrBooks ();
};
    
```

Implementierung Observer-Pattern: Library

- Wenn sich am beobachteten Wert was ändert => rufe notify auf
- Stellt Schnittstelle zum Abfragen des Werts zur Verfügung

```

public void Library::addBook( String author, string title )
{
    Book *book = new Book( author, title );
    books.add( book );
    nrBooks++;
    notify();
}

public void Library::deleteBook( String author, string title )
{
    Book *book = books.find( author, title );
    books.remove( book );
    nrBooks--;
    notify();
}

public void Library::geNrBooks ()
{
    return nrBooks;
}

```

- Die ConcreteObserver-Klasse
 - Registriert sich beim Subject als Observer
 - Implementiert die update-Funktion

```

class BookCounter : public Observer
{
    //The observed library
    private Library *subject;

    //Constructor
    public BookCounter( Library *subject );

    //Called by subject in case of changes
    public void update();
};
    
```


- Falls per notify der Subject-Klasse mitgeteilt wurde, dass sich was geändert hat => rufe update auf, um Wert zu aktualisieren

```

//Constructor
public BookCounter::BookCounter( Library *subject )
{
    this.subject = subject;
    subject.attach(this);
}

//Called by subject in case of changes
public BookCounter::update( Library *subject )
{
    //Get new state from observer
    int nrBooks = library.geNrBooks();

    //Do something with new state
    cout << "Es sind " << nrBooks << " Buecher in der Bibliothek" << endl;
}

```

■ Konsequenzen:

■ Vorteile:

- Subjekte und Beobachter können unabhängig variiert werden
- Sind nur auf minimale Weise miteinander gekoppelt
- Subjekt braucht keinerlei Kenntnis über Struktur seiner Beobachter
- Unterstützt Broadcasts

■ Nachteile

- Bei großer Beobachterzahl hohe Änderungskosten, da alle Beobachter informiert werden, auch wenn nicht alle die geänderte Information benötigen
- Keine Information, was sich genau geändert hat, sondern nur, dass sich was geändert hat
- Gefahr von Endlosschleifen, wenn Beobachter eines Subjekts auch gleichzeitig von diesem beobachtet wird

■ Pull-Modell

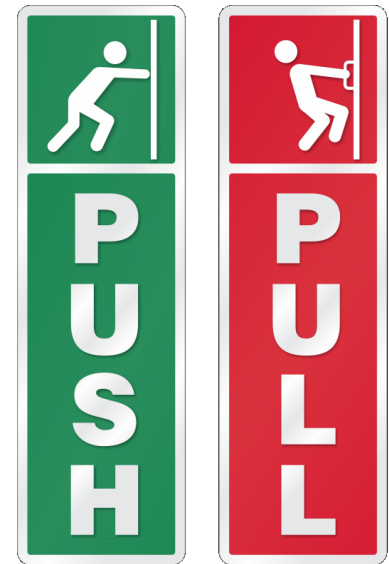
- Subject sendet nur minimale Beschreibung der Änderungen
- Observer holt sich die Informationen per `getState()`
- Nachteil: Observer benötigt Referenz auf Subject

■ Push-Modell

- Subject sendet detaillierte Änderungen an Observer
- Z.B. `update(int length, int width,...)`
- Vorteil: Observer braucht keine Informationen mehr über Subject
- Nachteil: Erweiterbarkeit, vermehrter Datenverkehr

■ Explizite Interessen

- Observer meldet Interesse an spezifischem Aspekt an
- Dieser Aspekt wird zusätzlicher Parameter von Update
- Nachteil: Schwieriger zu implementieren



■ Es gibt noch zahllose weitere Design Patterns:

■ Erzeugungsmuster:

- Factory Method
- Abstract Factory
- Buildem
- Prototype
- ...

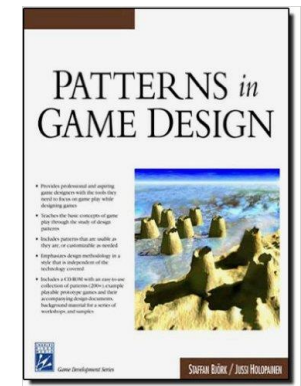
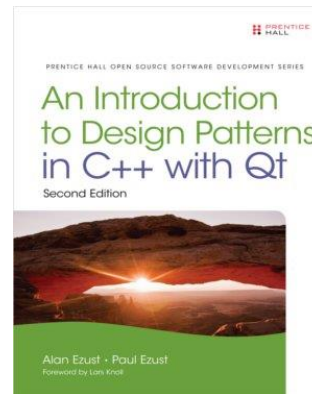
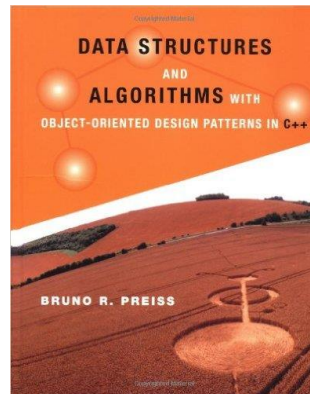
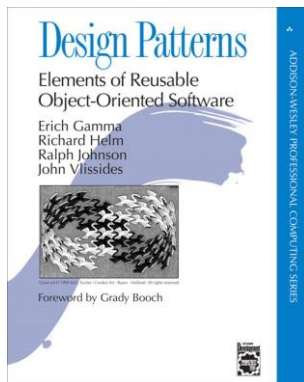
■ Strukturmuster:

- Wrapper
- Bridge
- Facade
- Flyweight
- ...

■ Verhaltensmuster:

- Interpreter
- Template Method
- Mediator
- Iterator
- ...

■ Wen es interessiert:





THE HYPE IS LONG GONE BUT DESIGN PATTERNS ARE STILL USEFUL