

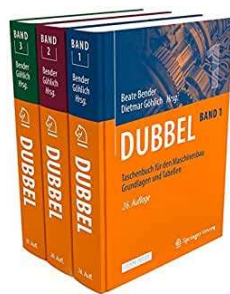
Kurseinheit 11: Design Patterns

1. Motivation
2. Singleton
3. Composite
4. Observer

1. Motivation

Lösungsmuster

In vielen technischen Bereichen gibt es Lösungsmuster für häufig wiederkehrende Problemstellungen.



Maschinenbau:
„Der Dubbel“
Erstauflage **1914**
Aktuelle 26. Auflage von 2020



Elektronik
Tietze, Schenk: Halbleiter-Schaltungstechnik
Erstauflage **1969**
Aktuelle 16. Auflage von 2019

Gibt es auch so etwas für die Objektorientierte Softwareentwicklung?

1. Motivation

3

Lösungsmuster



OO Softwareentwicklung

„GoF Buch“

GoF: Gang of Four

Erstauflage 1994

Gamma, Helm, Johnson und Vlissides (GoF) haben in ihrem Buch Design Patterns 23 Design Patterns vorgestellt und damit einen Pattern-Hype ausgelöst.

Während in dem Buch OO-orientierte Designmuster vorgestellt wurden, erschienen später von anderen Autoren Veröffentlichungen zu Analyse-Mustern, Test-Mustern, Anti-Mustern, ...

Ebenso erschienen weitere Veröffentlichungen zu Entwurfsmustern von anderen Autoren, wobei die 23 GoF Design Pattern immer noch Hauptbestandteil blieben.

1. Motivation


4


GoF-Design Patterns - Klassifizierung

Die GoF-Design Patterns sind wie folgt klassifiziert:

- **Creational Patterns:** Erzeugungsmuster
Erzeugung und Instanziierung von Objekten
- **Structural Patterns:** Strukturmuster
Beziehungen zwischen Klassen um umfangreiche komplexe Strukturen abzubilden
- **Behavioral Patterns:** Verhaltensmuster
Modellierung von komplexem Verhalten – Algorithmen und Verantwortlichkeiten

Beschrieben werden die Design Patterns überwiegend mit UML-Klassendiagrammen.

1. Motivation			5
GoF-Design Patterns - Klassifizierung			
Creational Patterns	FoU	C++	
Abstract Factory	5	-	
Builder	2	-	
Factory Method	5	-	
Prototype	3	-	
Singleton	4	Wird in dieser Kurseinheit behandelt	
Structural Patterns	FoU	C++	
Adapter	4		
Bridge	3		
Composite	4	Wird in dieser Kurseinheit behandelt	
Decorator	3	Wurde bereits im Labor behandelt	
Facade	5		
Flyweight	1		
Proxy	4		
FoU: Frequency of Use (5: oft – 1: selten)			
 C++ - KE11: Design Patterns		Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2	

1. Motivation			6
GoF-Design Patterns - Klassifizierung			
Behavioral Patterns	FoU	C++	
Chain of Responsibility	2		
Command	4		
Interpreter	1		
Iterator	5	Indirekt bei STL verwendet	
Mediator	2		
Memento	1		
Observer	5	Wird in dieser Kurseinheit behandelt	
State	3		
Strategy	4	Wurde bereits im Labor behandelt	
Template Method	3		
Visitor	1		
FoU: Frequency of Use (5: oft – 1: selten)			
Studierende aus AI werden im Rahmen des Studiums nahezu alle GoF-Design Patterns behandeln. Studierende aus EI, EI+, EI3nat und MKA sollten sich bei Bedarf in die anderen Patterns selbst einarbeiten.			
 C++ - KE11: Design Patterns		Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2	

1. Motivation

7

Empfehlung: E-Book und Sourcecode

Modern Book on Design Patterns

Hey, I have just reduced the price for all products. Let's prepare our programming skills for the post-COVID era. [Check it out >](#)

English Español Français
Polski Português-BR Русский
Українська 中文

Facebook Twitter

Contact us Log in

<https://refactoring.guru/design-patterns/book> [AS]

Dive Into DESIGN PATTERNS

An ebook on design patterns and the principles behind them

Design patterns help you solve commonly-occurring problems in software design. But you can't just find a pattern and copy it into your program, the way you can with off-the-shelf functions or libraries. A pattern is not a specific piece of code, but a general concept for solving a particular problem. They are like pre-made blueprints that you can customize to solve a recurring design problem in your code.

The book *Dive Into Design Patterns* illustrates 22 classic design patterns, solve it using one of the patterns.

~~€39.95~~
€19.95 Buy now

COVID relief amount

Buy as a gift

Log in

Wer später SW entwickeln will, sollte dieses E-Book selbst kaufen!

Elektrotechnik, Medientechnik und Informatik C++ - KE11: Design Patterns Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2

2. Singleton

8

„Highlander: Es kann nur einen geben“



In einem Softwaresystem darf/soll es nur ein Objekt einer Klasse geben.

- Hauptfenster
- Verbindung zur Datenbank
- Fehlerausgabe (Logger)
- ...

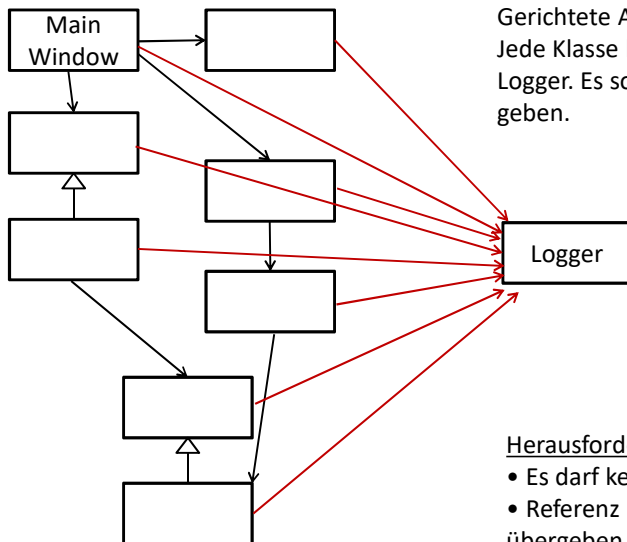
Oft soll von überall im System darauf zugegriffen werden.

Singleton wird heutzutage auch **oft als Anti-Pattern angesehen**, da es mit den objektorientierten Prinzipien bricht („Globales Objekt“).

2. Singleton

9

Problemstellung



Gerichtete Assoziation!

Jede Klasse hat eine Referenz auf die Klasse Logger. Es soll nur ein Objekt dieser Klasse geben.

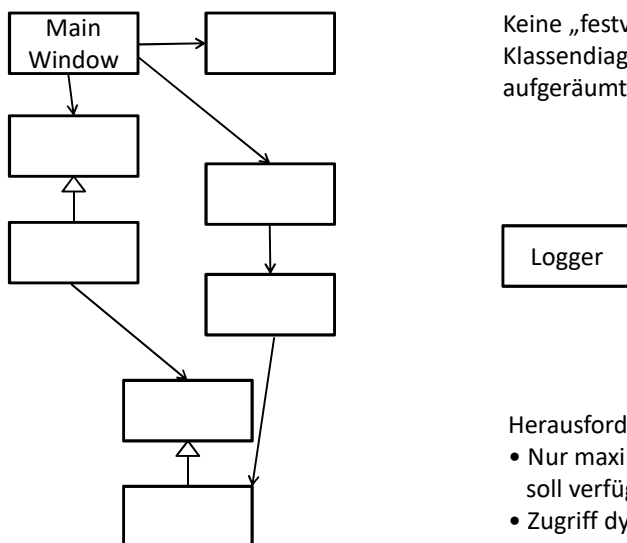
Herausforderungen:

- Es darf kein zweites Objekt erstellt werden.
- Referenz (oder Zeiger) müsste immer übergeben werden (Constructor Injection).

2. Singleton

10

Lösungsansatz



Keine „festverdrahteten“ Assoziationen. Klassendiagramm sieht deutlich aufgeräumter aus.

Herausforderungen

- Nur maximal ein Objekt der Klasse Logger soll verfügbar sein
- Zugriff dynamisch zur Laufzeit

2. Singleton

11

Original aus dem GoF-Buch

```
class Singleton
{
public:
    static Singleton* GetInstance();

protected:
    Singleton();

private:
    static Singleton* _instance;
};
```

.h

Klassenmethode!
Ohne Instanz von überall aufrufbar, da public.

Klassenvariable!
Ohne Instanz vorhanden. Zugriff aber nur aus der Klasse heraus, da private.

```
Singleton* Singleton::_instance = 0;

Singleton::Singleton()
{
    std::cout << "Con Singleton\n";
}
```

.cpp

```
Singleton* Singleton::GetInstance()
{
    if (_instance == 0)
    {
        _instance = new Singleton;
    }
    return _instance;
}
```

.cpp

2. Singleton

12

Original aus dem GoF-Buch: Doch kein Highlander?

```
int main()
{
    //Singleton MySingleton1(); //Error
    //Singleton* pMySingleton2 = new Singleton(); //Error

    Singleton* pMySingleton3 = Singleton::GetInstance();
    Singleton* pMySingleton4 = Singleton::GetInstance();

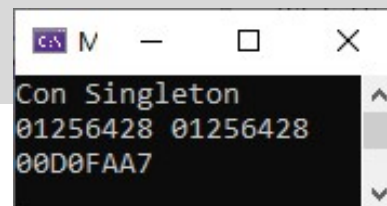
    std::cout << pMySingleton3 << " " << pMySingleton4 << std::endl;

    //Ups
    Singleton MySingleton5(*pMySingleton3);
    std::cout << &MySingleton5 << std::endl;

    return 0;
}
```

Wie lassen sich weitere Instanzen anlegen?

Durch den Default-Kopierkonstruktor!



2. Singleton

13

Lösung 1 aus: DIVE INTO Design Patterns

```
class Singleton
{
protected:
    Singleton
    {}
    static Singleton* psingleton_;

public:
    Singleton(Singleton &other) = delete;
    void operator=(const Singleton &) = delete;
    static Singleton *GetInstance();

    void SomeBusinessLogic()
    {
        // ...
    }
}
```

Default-CopyConstructor und Default-CopyAssign werden gelöscht.

In einer **Multiple-Threading-Anwendung** können aber trotzdem noch weitere Singletons instanziiert werden.

2. Singleton

14

Lösung 1 aus: DIVE INTO Design Patterns

Wie können trotzdem in einer Multiple-Threading-Anwendung Singletons instanziiert werden?

```
Singleton* Singleton::GetInstance()
{
    if (_instance == 0)
    {
        _instance = new Singleton;
    }
    return _instance;
}
```

Einprozessorsystem (Quasi-Parallelität):

T1 kommt zur Abfrage (ist true), wird unterbrochen, T2 kommt zu Zuge (new Singleton), später wird wieder auf T1 umgeschaltet und dieser springt in den Wahr-Block (new Singleton).

Mehrprozessorsystem (echte Parallelität):

T1 und T2 fragen gleichzeitig _instance ab. Ist true, T1 und T2 führen Wahr-Block aus.

2. Singleton

15

Lösung 2 aus: DIVE INTO Design Patterns

```
class Singleton
{
private:
    static Singleton * pinstance_;
    static std::mutex mutex_;

// ... Same as before
}
```

Kann nicht gleichzeitig von zwei Threads durchgeführt werden. T2 wartet bis lock von T1 out of scope wird und kann dann erst fortfahren (pinstance_ ist dann kein nullptr.)

Ähnlich zu unique_ptr:
Lokales Objekt lock erhält Mutex und gibt wieder frei, wenn lock out of scope wird.

```
Singleton *Singleton::GetInstance()
{
    if (pinstance_ == nullptr)
    {
        std::lock_guard<std::mutex> lock(mutex_);
        if (pinstance_ == nullptr)
        {
            pinstance_ = new Singleton();
        }
    }
    return pinstance_;
}
```



Dokumentation zu lock_guard: https://en.cppreference.com/w/cpp/thread/lock_guard

Elektrotechnik, Medizintechnik und Informatik

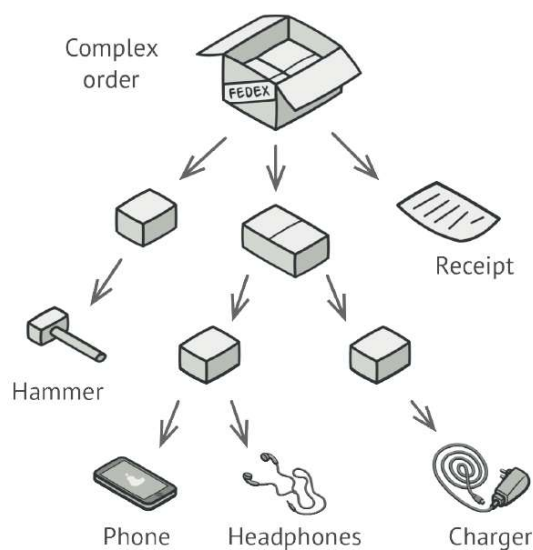
C++ - KE11: Design Patterns

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2

3. Composite

16

Problemstellung



Ein Karton kann weitere Kartons enthalten, wobei in einem Karton auch Einzelteile liegen können.

Diese Problemstellung findet sich in der realen Welt sehr häufig:

- Abteilungen mit Unterabteilungen und Mitarbeitern
- Dateisystem: Ein Verzeichnis enthält Unterverzeichnisse und Dateien
- Eine Maschinenkomponente besteht aus weiteren Komponenten und Einzelteilen

Quelle: [AS]

Elektrotechnik, Medizintechnik und Informatik

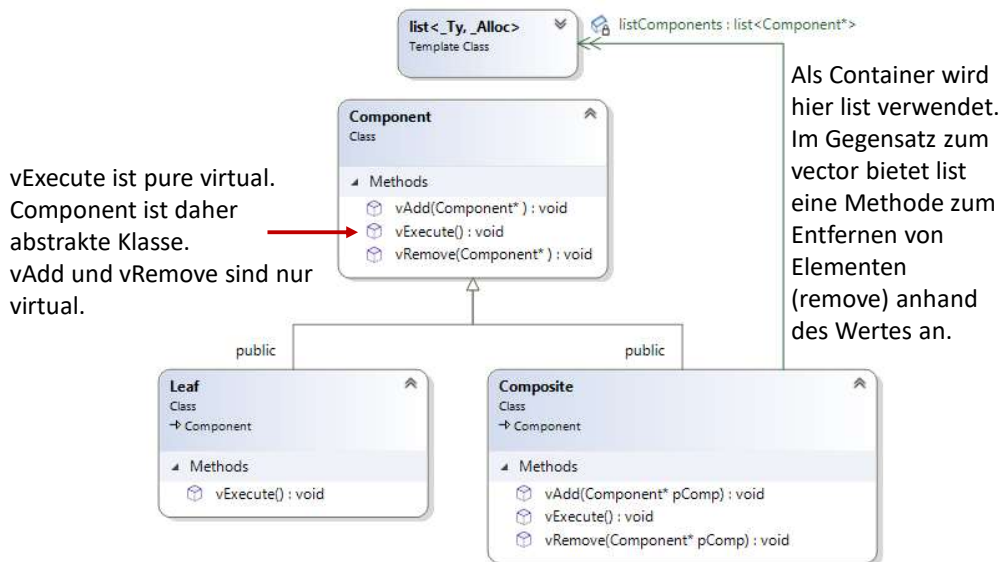
C++ - KE11: Design Patterns

Prof. Dr.-Ing. Daniel Fischer - Version 3.0.2

3. Composite

17

Generisches Composite



3. Composite

18

Generisches Composite

```
namespace MyDesignPatterns
{
class Component
{
public:
    virtual void vExecute(void) const = 0;
    virtual void vAdd(Component*) { };
    virtual void vRemove(Component*) { };
};

class Leaf : public Component
{
public:
    void vExecute(void) const override
    {
        std::cout << "vExecute Leaf" << std::endl;
    }
};
```

Leere Defaultimplementierung

Falls Kindklasse nicht überschreibt, wird diese Defaultimplementierung geerbt.
Leaf leaf;
leaf.vAdd(..) ist möglich.

3. Composite

19

Generisches Composite

```
class Composite : public Component
{
public:
    void vExecute(void) const override
    {
        std::cout << "vExecute Composite" << std::endl;
        for (auto ait : listComponents_)
        {
            ait->vExecute();
        }
    }

    void vAdd(Component* pComp) override {listComponents.push_back(pComp);}
    void vRemove(Component* pComp) override {listComponents.remove(pComp);}

private:
    std::list<Component*> listComponents_;
};
//end namespace
```

auto ranged loop über
Containerelemente!!!

3. Composite

20

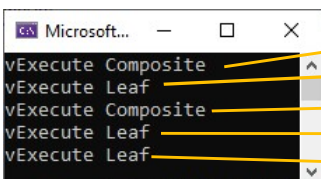
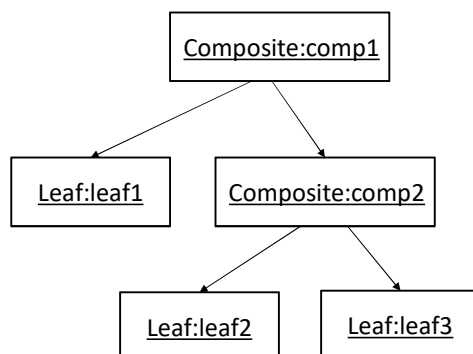
Generisches Composite - Anwendung

```
MyDesignPatterns::Leaf leaf1;
MyDesignPatterns::Leaf leaf2;
MyDesignPatterns::Leaf leaf3;
MyDesignPatterns::Composite comp1;
MyDesignPatterns::Composite comp2;
```

```
comp1.vAdd(&leaf1);
comp2.vAdd(&leaf2);
comp2.vAdd(&leaf3);
```

```
comp1.vAdd(&comp2);
comp1.vExecute();
```

UML-Objektdiagramm:

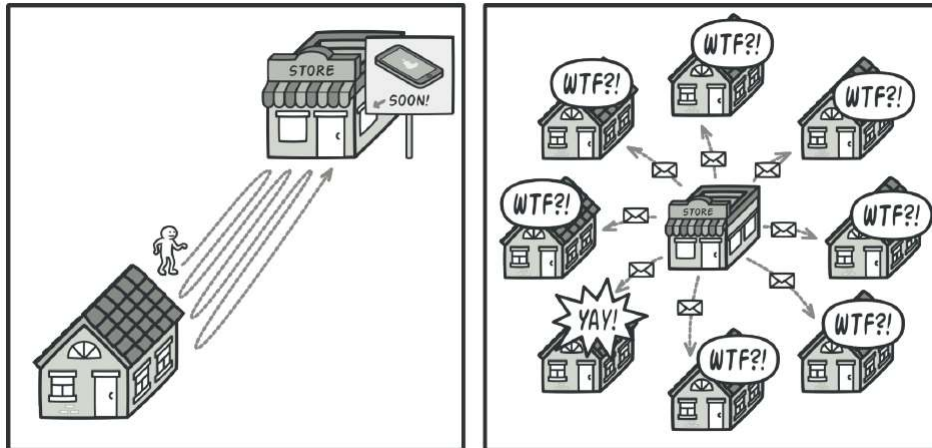


comp1
leaf1
comp2
leaf2
leaf3

4. Observer

21

Problemstellung 1



Eine **Kunde** will ein neues Avocado Smart Phone kaufen und fragt immer im Avocado **Store** nach, ob dieses schon da ist. Der Shop könnte aber an alle Haushalte eine Nachricht schicken, dass dieses da ist. Dies würde bei den meisten Haushalten (WTF) unnötig sein. Quelle: [AS]

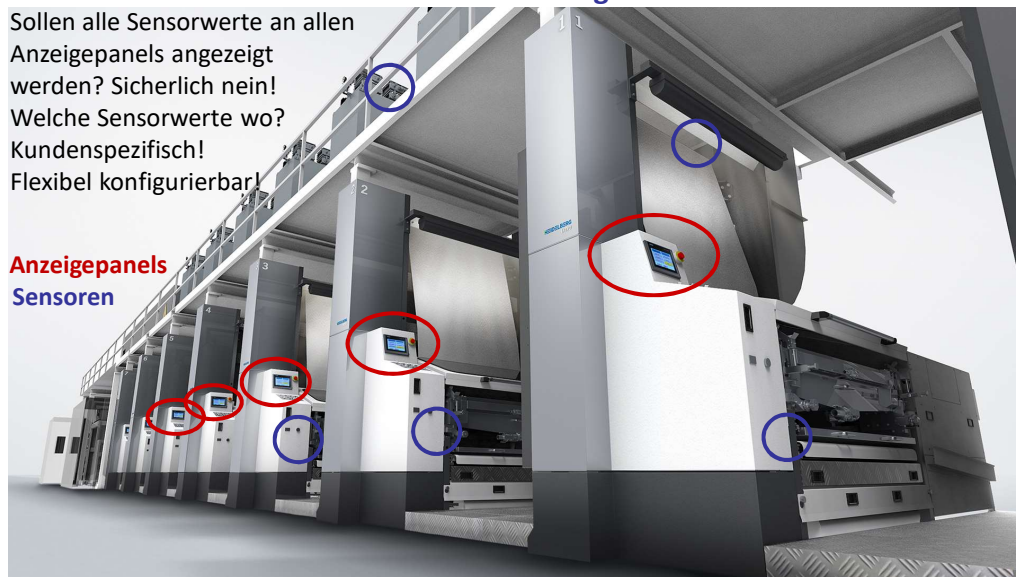
4. Observer

22

Problemstellung 2

Sollen alle Sensorwerte an allen
Anzeigepanels angezeigt
werden? Sicherlich nein!
Welche Sensorwerte wo?
Kundenspezifisch!
Flexibel konfigurierbar!

Anzeigepanels
Sensoren

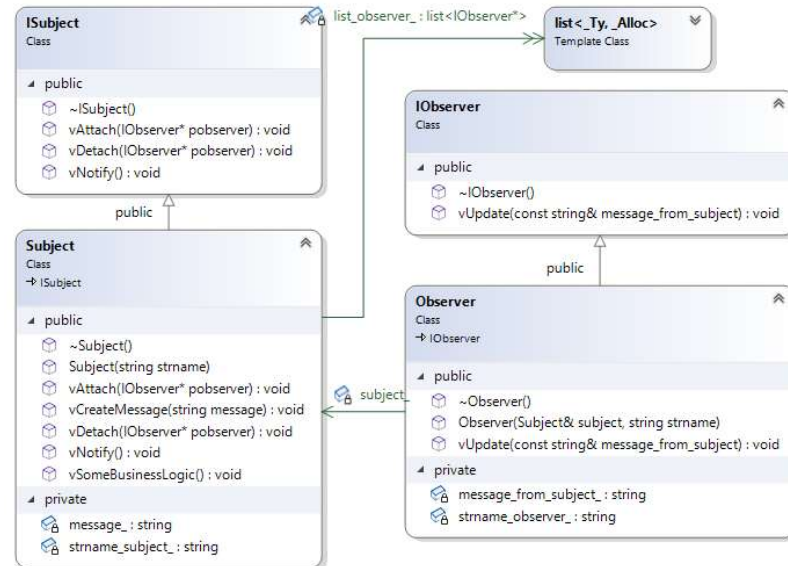


https://www.heidelberg.com/global/media/global_media/products_flexo_print/jpg_115/Intro_Seitenansicht.jpg

4. Observer

23

Generisches Observer



Problemstellung 1:
Subject ist Shop
Observer ist Kunde

Problemstellung 2:
Subject ist Sensor
Observer ist Anzeigepanel

4. Observer

24

Generisches Observer - Interfaces

```

class IObservable
{
public:
    virtual ~IObservable() {};
    virtual void vUpdate(const std::string& message_from_subject) = 0;
};

class ISubject
{
public:
    virtual ~ISubject() {};
    virtual void vAttach(IObservable* pobserver) = 0;
    virtual void vDetach(IObservable* pobserver) = 0;
    virtual void vNotify() = 0;
};
    
```

C++ hat **kein** Schlüsselwort Interface. Eine Klasse, die nur rein virtuelle Methoden (außer Destruktor) enthält, ist dann ein **Interface**. Man schreibt zur besseren Kennzeichnung dann ein „I“ vor den Klassennamen (I für Interface). Enthält eine Klasse rein virtuelle Methoden und weitere (virtuelle und nicht virtuelle) Methoden, so handelt es sich „nur“ um eine **abstrakte Klasse** (siehe Component).

Generisches Observer – Subject

```

class Subject : public ISubject
{
public:
    Subject(std::string strname);
    virtual ~Subject();
    void vAttach(IObserver* pobserver) override;
    void vDetach(IObserver* pobserver) override;
    void vNotify() override;
    void vCreateMessage(std::string message = "Empty");
    void vSomeBusinessLogic();

private:
    std::list<IObserver*> list_observer_;
    std::string message_;
    std::string strname_subject_;
};

```

Verschickt die neue Nachricht (message_) an alle Observer

Generisches Observer – Observer

```

class Observer : public IObserver
{
public:
    Observer(Subject& subject, std::string strname);
    virtual ~Observer();
    void vUpdate(const std::string& message_from_subject) override;

private:
    std::string message_from_subject_;
    Subject& subject_;
    std::string strname_observer_;
};

```

Die erhaltene Nachricht vom Subject.

Zwei Varianten des Observer Patterns:

- **Push-Variante:** Subject übergibt die Daten direkt (bei einfachen Daten) – hier vorgestellt.
- **Pull-Variante:** Wird bei komplexeren Daten verwendet. Subject informiert Observer, dass neue Daten vorhanden sind. Dieser holt sich dann seine spezifischen Daten vom Subject ab. Hier muss der Observer sein Subject kennen (subject_).

4. Observer

27

Generisches Observer – Wichtige Subject-Methoden

```
void Subject::vNotify()
{
    for (auto ait : list_observer_)
    {
        ait->vUpdate(message_);
    }
}
```

auto ranged loop
Iteriert über
Container

```
void Subject::vCreateMessage(std::string message)
{
    message_ = message;
    vNotify();
}
```

Daten sind hier nur
std::string

```
void Subject::vSomeBusinessLogic()
{
    message_ = "change message message";
    vNotify();
    std::cout << "I'm about to do something important\n";
}
```

Jedes nicht
generische Subject
hat natürlich
„Business Logic“

4. Observer

28

Generisches Observer – Anwendung

```
MyDesignPatterns::Subject subjectA1("Avacoda-Shop");
```

```
MyDesignPatterns::Observer observerPeter(subjectA1, "Peter");
```

```
MyDesignPatterns::Observer observerPaul(subjectA1, "Paul");
```

```
MyDesignPatterns::Observer observerMarie(subjectA1, "Marie");
```

```
subjectA1.vAttach(&observerPeter);
```

```
subjectA1.vAttach(&observerPaul);
```

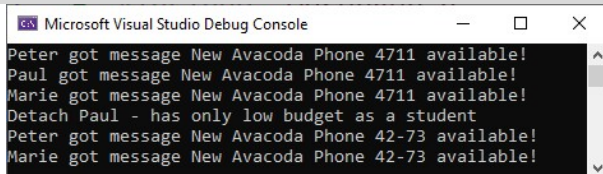
```
subjectA1.vAttach(&observerMarie);
```

```
subjectA1.vCreateMessage("New Avacoda Phone 4711 available!");
```

```
std::cout << "Detach Paul - has only low budget as a student\n";
```

```
subjectA1.vDetach(&observerPaul);
```

```
subjectA1.vCreateMessage("New Avacoda Phone 42-73 available!");
```



```
Microsoft Visual Studio Debug Console
Peter got message New Avacoda Phone 4711 available!
Paul got message New Avacoda Phone 4711 available!
Marie got message New Avacoda Phone 4711 available!
Detach Paul - has only low budget as a student
Peter got message New Avacoda Phone 42-73 available!
Marie got message New Avacoda Phone 42-73 available!
```