Objektorientiertes Programmieren mit C++ für Fortgeschrittene

Kapitel 9

9. Entwurfsmuster (*Design Pattern*)

9.1. Allgemeines und Überblick

9.2. Beispiele

Entwurfsmuster – Allgemeines

• Allgemeines

◇ **Entwurfsmuster** (**Design Pattern**) sind wiederverwendbare **bewährte** generische **Lösungen** für bestimmte immer wiederkehrende Entwurfsprobleme. Dabei ist charakteristisch, dass in **unterschiedlichen Anwendungsbereichen** auftretende **gleichartige Probleme** durch Anwendung des gleichen Entwurfsmusters **gleichartig gelöst** werden können.

◇ Entwurfsmuster können auf unterschiedlichen Abstraktionsebenen gebildet werden.
Im engeren Sinn versteht man darunter *Beschreibungen von interagierenden **Objekten und Klassen, die so aufeinander abgestimmt sind, dass sie ein allgemeines Entwurfsproblem in einem bestimmten Kontext lösen*** können

◇ Ein Entwurfsmuster **identifiziert** und **abstrahiert** die **Kern-Aspekte** einer allgemeinen Entwurfsstruktur.
Es identifiziert und benennt die beteiligten Klassen und Objekte, ihre Rollen, Verantwortlichkeiten und ihre Beziehungen.

> Entwurfsmuster **dokumentieren** erprobte **Entwurfserfahrungen**.
> Sie fördern dadurch in besonderen Maße die **Wiederverwendbarkeit** bewährter Lösungsstrukturen und ermöglichen einen Entwurf auf **höherem Abstraktionsniveau**.
> Ihre Kenntnis und sinnvolle Anwendung **erleichtert** und **beschleunigt** den **Entwicklungsprozess** und **verhindert** gleichzeitig den Einsatz **"schlechterer" Lösungsalternativen**.

• Beschreibungselemente

◇ Im Laufe der Zeit sind zahlreiche Entwurfsmuster "entdeckt" und in Büchern und Fachzeitschriften veröffentlicht worden.
Neue Muster kommen laufend hinzu.

◇ Die Beschreibung eines Entwurfsmusters sollte wenigstens die folgenden vier Elemente umfassen :

▻ **Name des Musters** (*pattern name*)
Prägnante Zusammenfassung des Entwurfsproblems, seiner Lösung und deren Anwendungskonsequenzen in ein oder zwei Worten.

▻ **Problembeschreibung** (*problem*)
Darstellung des Anwendungsbereichs des Entwurfsmusters durch Erläuterung des Problems, seines Kontexts und gegebenenfalls spezifischer Entwurfsprobleme

▻ **Problemlösung** (*solution*)
Beschreibung der Komponenten (Objekte und Klassen) der Entwurfsstruktur, ihrer Verantwortlichkeiten, ihrer Beziehungen und ihrer Zusammenarbeit.
Dabei bezieht sich die Beschreibung weder auf einen konkreten Entwurf noch auf eine konkrete Implementierung, da ein Entwurfsmuster wie eine Schablone in verschiedenen – aber strukturell ähnlichen - Situationen anwendbar sein soll.
Entwurfsmuster beruhen auf einer abstrakten Darstellung eines Entwurfsproblems und stellen eine allgemeine Anordnung von Elementen (Klassen und Objekten), die das Problem lösen kann, zur Verfügung.
Prinzipielle Realisierungsbeispiele – in einer konkreten Implementierungssprache – können gegebenenfalls angegeben werden.

▻ **Anwendungskonsequenzen** (*consequences*)
Auflistung der Ergebnisse und Folgen ("*trade-offs*"), die sich aus der Anwendung des Entwurfsmusters ergeben.
Die Kenntnis der Konsequenzen sind für eine kritische Untersuchung von Entwurfsalternativen und die Abwägung von Kosten und Nutzen einer Lösung wichtig.
Häufig betreffen die Konsequenzen Speicher- und/oder Zeiteffizienz.
Sie können sich aber auch auf Sprach- und Implementierungsgesichtspunkte beziehen.
Darüber hinaus ist es grundsätzlich wichtig, den Einfluss eines Entwurfsmusters auf die Flexibilität, die Erweiterbarkeit und die Portabilität eines Systems zu kennen.

Entwurfsmuster – Klassifizierung

- **Klassifikation**

  ◇ Die verschiedenen bisher veröffentlichten Entwurfsmuster **differieren** bezüglich ihres **Anwendungsbereiches** und ihrer **strukturellen Auflösung** (*granularity*).
  Sie lassen sich nach **unterschiedlichen Gesichtspunkten klassifizieren.**

  ◇ In Anlehnung an das Standardwerk von Gamma/Helm/Johnson/Vlissides ("Design Patterns") ist vor allem eine Klassifizierung nach zwei Gesichtspunkten üblich :

  ▻ **Einsatzzweck** (*purpose*)

  ▻ **Geltungsbereich** (*scope*)

  ◇ Der **Einsatzzweck** spiegelt wieder, ***was*** ein Entwurfsmuster ***bewirkt***. Man unterscheidet :

  ▻ **Erzeugende Muster** (*creational patterns*)
  Sie beziehen sich auf die Erzeugung von Objekten

  ▻ **Strukturelle Muster** (*structural patterns*)
  Sie befassen sich mit der Zusammensetzung (*composition*) von Klassen und Objekten

  ▻ **Verhaltensmuster** (*behavioral patterns*)
  Sie bestimmen die Zusammenarbeit zwischen Klassen bzw. Objekten und die Verteilung ihrer Verantwortlichkeiten

  ◇ Der **Geltungsbereich** legt fest, ***worauf*** sich ein Entwurfsmuster primär ***bezieht*** (auf Klassen oder auf Objekte):

  ▻ **Klassenmuster** (*class patterns*)
  Sie befassen sich primär mit Vererbungsbeziehungen zwischen Klassen.
  Diese Beziehungen sind statisch und liegen zur Compile-Zeit fest.

  ▻ **Objektmuster** (*object patterns*)
  Sie befassen sich primär mit den Nutzungsbeziehungen zwischen Objekten.
  Diese Beziehungen sind i.a. zur Laufzeit änderbar und daher dynamisch.
  Die meisten Entwurfsmuster gehören dieser Kategorie an.

- **Klassifizierung der Standard-Entwurfsmuster nach Gamma (u.a.)**

		Geltungsbereich	
		Klassenmuster	**Objektmuster**
Einsatzzweck	**Erzeugende Muster**	Factory Method	Abstract Factory Builder Prototype Singleton
	Strukturelle Muster	Adapter (class)	Adapter (Object) Bridge Composite Decorator Facade Flyweigth Proxy
	Verhaltensmuster	Interpreter Template Method	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategie Visitor

Entwurfsmuster – Überblick (1)

- **Überblick über die Standard-Entwurfsmuster nach Gamma (u.a.)**

Erzeugende Muster (*Creational Patterns*)

◇ **Abstract Factory** (*Kit*) : Stellt ein Interface zur Erzeugung ganzer Familien verwandter oder voneinander abhängiger Objekte zur Verfügung, ohne dass deren konkrete Klassen spezifiziert werden müssen

◇ **Builder** : Trennt die Konstruktion eines komplexen Objekts von seiner Repräsentation, so dass derselbe Konstruktionsprozess zur Erzeugung unterschiedlicher Repräsentationen eingesetzt werden kann.

◇ **Factory Method** (*Virtual Constructor*) : Definiert ein Interface zur Erzeugung eines Objekts, überträgt aber Subklassen die Entscheidung welches konkrete Objekt tatsächlich erzeugt wird.

◇ **Prototype** : Instantiiert Objekt-Prototypen und erzeugt neue Objekte durch Kopieren dieser Prototypen.
Dadurch wird die Erzeugung von erst zur Laufzeit spezifizierter Objekte ermöglicht.

◇ **Singleton** :Stellt sicher, dass nur eine Instanz einer Klasse erzeugt wird und stellt einen globale Zugriffsmöglichkeit zu dieser Instanz zur Verfügung.

Strukturelle Muster (*Structural Patterns*)

◇ **Adapter** (*Wrapper*) : Wandelt das Interface einer Klasse in ein anderes von einem Client erwartetes Interface um. Adapter ermöglichen die Zusammenarbeit von Klassen, die sonst wegen inkompatibler Interfaces nicht zusammenarbeiten könnten.

◇ **Bridge** (*Handle, Body*) : Entkoppelt eine Abstraktion (Klassenhierarchie) von ihrer Implementierung (Klassenhierarchie), so dass beide unabhängig voneinander variiert werden können.

◇ **Composite** (*Recursive Composition*) : Ermöglicht die Repräsentation von Teil-Ganzheits-Beziehungen durch den Aufbau baumartiger Objektstrukturen. Dadurch können Clients individuelle Objekte und Zusammensetzungen von Objekten gleichartig behandeln.

◇ **Decorator** : Ergänzt ein Objekt dynamisch um zusätzliche Fähigkeiten.
Das Entwurfsmuster stellt bezüglich dieser Erweiterung eine flexible Alternative zur Vererbung dar.

◇ **Facade** : Kapselt einen Satz von Interfaces innerhalb eines Subsystems nach außen durch ein vereinfachtes einheitliches Interface. Dadurch wird ein Interface höherer Ebene definiert, das die Benutzung des Subsystems erleichtert.

◇ **Flyweight** : Ermöglicht die Mehrfachnutzung von Objekten mit gleichem inneren Zustand aber unterschiedlichem äußeren Zustand (Kontext). Dadurch werden Strukturen mit einer großen Anzahl "einfacherer" Objekte effektiv unterstützt.

◇ **Proxy** (*Surrogate*) : Stellt einen Platzhalter (Stellvertreter) für ein Objekt zur Verfügung, um den Zugriff zu diesem Objekt zu kontrollieren.

Entwurfsmuster – Überblick (2)

- **Überblick über die Standard-Entwurfsmuster nach Gamma (u.a.)**, Forts.

Verhaltensmuster (*Behavioral Patterns*)

◇ **Chain of Responsibility** : Ermöglicht, dass mehr als ein Objekt die Gelegenheit zur Reaktion auf eine Botschaft erhält. Die möglichen Botschaftenempfänger werden miteinander verkettet und die Botschaft wird die Kette entlang gereicht, bis ein Objekt diese bearbeitet.

◇ **Command** (*Action, Transaction*) : Kapselt eine Botschaft als Objekt. Dadurch werden Sender und Empfänger einer Botschaft voneinander entkoppelt. Dies ermöglicht u.a. die Parametrisierung von Client-Objekten (Anwendungen) mit unterschiedlichen Botschaften für unterschiedliche daraus folgende Operationen.

◇ **Interpreter** : Beschreibt die Definition der Grammatik einer einfachen Sprache sowie die Darstellung von Sätzen in der Sprache zusammen mit einem Interpreter zu ihrer Interpretation.

◇ **Iterator** (*Cursor*) : Ermöglicht den sequentiellen Zugriff zu den Elementen eines Aggregat-Objekts ohne dessen internen Aufbau offenzulegen.

◇ **Mediator** : Definiert ein Objekt, das die Interaktion einer Objektmenge kapselt. Das Entwurfsmuster unterstützt eine lose Objektkopplung durch Verhinderung eines expliziten Bezugs der Objekte aufeinander. Es ermöglicht eine unabhängige Änderung der Objektinteraktion.

◇ **Memento** (*Token*) : Ermöglicht das Festhalten und die äußere Darstellung des inneren Zustands eines Objekts zu seiner späteren Wiederherstellung, ohne die Objektkapselung zu verletzen.

◇ **Observer** (*Dependents, Publish-Subscribe*) : Ermöglicht die dynamische Registrierung von Objektabhängigkeiten, so dass bei einem Zustandswechsel eines Objekts alle von ihm abhängigen Objekte benachrichtigt werden.

◇ **State** (*Object for States*) : Kapselt die Zustände eines Objekts in separate Objekte, die jeweils zu einer Zustandsklasse gehören, die von einer gemeinsamen abstrakten Basis-Zustandsklasse abgeleitet sind. Dieses Entwurfsmuster ermöglicht einem Objekt, sein Verhalten bei einer Zustandsänderung zu ändern. Nach außen scheint das Objekt dadurch seine Klasse zu ändern.

◇ **Strategy** (*Policy*) : Kapselt verwandte Algorithmen in jeweils einer eigenen Klasse, die von einer gemeinsamen Basisklasse abgeleitet ist. Dies ermöglicht die Auswahl und Änderung der Algorithmen zur Laufzeit ohne dass Änderungen an den Clients, die sie benutzen, notwendig sind.

◇ **Template Method** : Definiert das Gerüst eines Algorithmus in einer abstrakten Basisklasse, wobei die Konkretisierung einiger Algorithmus-Schritte an abgeleitete Klassen übertragen wird, ohne dass dadurch die Struktur des Algorithmus geändert werden muss.

◇ **Visitor** : Implementiert eine Operation, die mehrere Objekte in einer komplexen Struktur betrifft, in einem separaten Objekt. Das Entwurfsmuster ermöglicht die Definition neuer Operationen ohne Änderung der Klassen der Objekte, auf die sich die Operation bezieht.

Entwurfsmuster : Singleton (1)

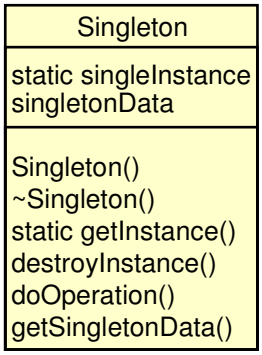
- **Name : Singleton**

- **Kurzbeschreibung**
  Stellt sicher, dass **nur eine Instanz** einer Klasse erzeugt wird und stellt einen **globale Zugriffsmöglichkeit** zu dieser Instanz zur Verfügung.

- **Problembeschreibung**
  Oft ist es wichtig, dass von einer Klasse nur eine einzige Instanz erzeugt wird.
  Zusätzlich besteht häufig die Forderung, dass diese Instanz global zugreifbar sein soll.
  Beispielsweise sollte in einem zentralisierten Workflow-Managementsystem nur ein Manager-Objekt vorhanden sein.

- **Problemlösung**
  In einer sinnvollen Lösung ist die Klasse selbst dafür verantwortlich, dass sie nur einmal instantiiert wird.
  Dies wird dadurch erreicht, dass alle Konstruktoren der Klasse privat sind
  Zur Objekterzeugung und zum Zugriff auf das Singleton-Objekt dient eine öffentliche statische Memberfunktion.
  Diese überprüft zuerst, ob bereits eine Instanz der Klasse angelegt ist.
  Falls nein wird eine Instanz – durch Aufruf eines privaten Konstruktors - angelegt und ein Pointer auf diese in einer privaten statischen Membervariablen abgelegt. Dieser Pointer wird gleichzeitig als Funktionswert zurückgegeben.
  Falls ja, gibt sie den in der statischen Membervariablen gespeicherten Pointer auf die Instanz zurück.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Da die Singleton-Klasse ihre einzige Instanz kapselt, hat sie die strikte Kontrolle über den Zugriff zu dieser
  - ▻ Die Singleton-Klasse kann leicht zu einer Klasse erweitert werden, die eine definierte Anzahl von Instanzen größer als eins erzeugen kann.
  - ▻ Der Einsatz einer Singleton-Klasse als Basisklasse kann problematisch sein.
    In diesem Fall darf der Konstruktor nicht privat sein. Außerdem lässt sich die statische Objekterzeugungs-Funktion nicht überschreiben.

Entwurfsmuster : Singleton (2)

- **Implementierungsbeispiel**

  ◇ **Definition der Singleton-Klasse :**

```
class Singleton
{
  public :
    static Singleton* getInstance();
    void destroyInstance();
    // weitere nicht-statische Memberfunktionen
  private :
    static Singleton* singleInstance;
    // weitere nicht-statische Datenkomponenten
    Singleton();
    ~Singleton();
 };
```

  ◇ **Implementierung der Singleton-Klasse :**

```
Singleton* Singleton::singleInstance = 0;   // == NULL-Pointer

Singleton::Singleton()
{
  // privater Konstruktor
}

Singleton::~Singleton()
{
  // privater Destruktor
}

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

void Singleton::destroyInstance()
{
  delete this;
  singleInstance=0;       // == NULL-Pointer
}
```

Entwurfsmuster : Factory Method (1)

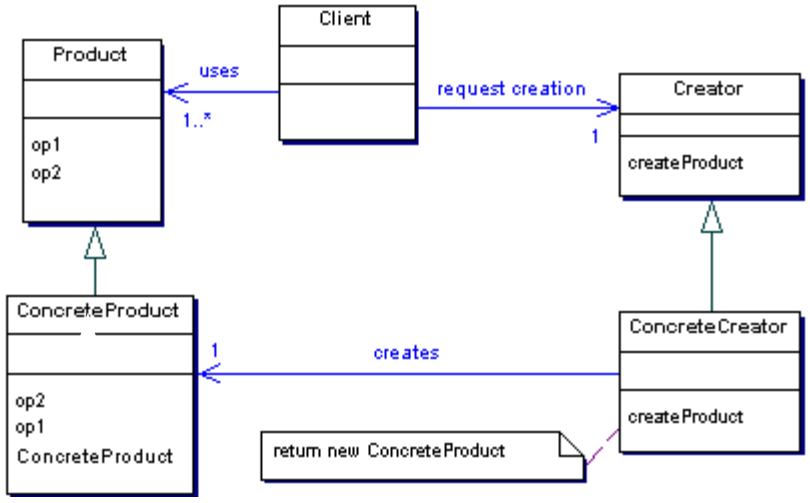
- **Name : Factory Method** (*Virtual Constructor*)

- **Kurzbeschreibung**
  Ermöglicht die Erzeugung **applikationsspezifischer Objekte** durch **applikationsunabhängigen** und damit auch von den Klassen der erzeugten Objekte unabhängigen **Code**.

- **Problembeschreibung**
  Es gibt zahlreiche Situationen, in denen ein Objekt einer Klasse (Client) – je nach Applikation in der sie eingesetzt wird – mit unterschiedlichen Objekten, die aber alle eine gemeinsame Basisklasse haben, arbeiten soll. Das jeweils zu verwendende Objekt muss von dem verwendenden Objekt (Client) erzeugt werden. Die Klasse des verwendenden Objekts (Client) kennt zwar den Zeitpunkt der Objekterzeugung, jedoch nicht dessen konkrete Klasse. Um unabhängig von der Klasse des zu erzeugenden Objekts zu sein und damit allgemein verwendbar zu bleiben, muss die eigentliche Objekterzeugung entweder in eine eigene Erzeugerklasse ausgelagert werden oder durch Unterklassen realisiert werden.
  Typisch für solche Situationen sind z. B. Frameworks, die es ermöglichen ganze Gruppen verwandter Anwendungen mit gemeinsam nutzbaren Klassen, zu realisieren. Beispiel sei ein Framework für Desktop-Applikationen, die jeweils mit unterschiedlichen Dokumenten-Objekten arbeiten. Die Framework-Klassen sollten unabhängig von der Klasse des konkret zu erzeugenden applikationsspezifischen Dokumenten-Objekts sein.

- **Problemlösung**
  Definition eines **Interfaces** in einer – häufig abstrakten – Klasse **`Creator`** zur Erzeugung eines Objekts der Klasse **`Product`** mittels einer (rein) **virtuellen Funktion** (→ ***factory method***). Diese Methode wird in von `Creator` abgeleiteten Klassen (**`ConcreteCreator`**) so **überschrieben**, dass das jeweils benötigte konkrete Objekt (abgeleitete Klasse **`ConcreteProduct`**) erzeugt wird. Die `Creator`–Klasse verschiebt also die konkrete Objekterzeugung zu ihren abgeleiteten Klassen, die somit die Applikationsabhängigkeit enthalten.
  Der applikationsunabhängige Code (der sich in einer eigenen Anwendungsklasse **`Client`** befindet oder auch in der `Creator`–Klasse angesiedelt sein kann) ruft zur Erzeugung eines Objekts über das Interface tatsächlich die überschreibende Methode auf.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Der die Objekterzeugung anfordernde Code (Client) ist unabhängig von der Klasse des konkret erzeugten Objekts. Er ist damit abstrakter und wiederverwendbarer.
  - ▻ Es existieren **zwei Hauptvarianten** dieses Entwurfsmusters :
    - Der die Objekterzeugung anfordernde Code befindet sich in einer von `Creator` getrennten eigenen Klasse
    - Der die Objekterzeugung anfordernde Code befindet sich in der `Creator`–Klasse (`Client`–Klasse existiert nicht)
  - ▻ Die `Creator`–Klasse kann abstrakt sein (*factory method* ist rein virtuell). Sie kann aber auch eine konkrete Klasse sein und eine Default-Implementierung dieser Methode bereitstellen.
  - ▻ Das Entwurfsmuster erhöht die Klassen-Komplexität, da Subklassen allein zur Objekterzeugung benötigt werden.
  - ▻ Die Objekterzeugungsfunktion (*factory method*) kann auch **parameterisiert** werden. Die Parameter können entweder an den Konstruktor der Klasse des zu erzeugenden Objekts weitergereicht werden oder/und zur Auswahl zwischen verschiedenen Klassen dienen → Erzeugung von Objekten unterschiedlichen Typs durch eine Methode.

Entwurfsmuster : Factory Method (2)

- **Implementierungsbeispiel** (Teil 1)

◇ **Definition einer – abstrakten – Produkt-Klasse** (**`Product`**)

```
class RawDiskReader
{ public :
    virtual ~RawDiskReader() {};
    virtual bool openDisk(char*) = 0;
    virtual bool readSector(unsigned long, char*) = 0;
};
```

◇ **Definition einer konkreten Produkt-Klasse** (**`ConcreteProduct`**)

```
class WinRawDiskReader : public RawDiskReader
{ public :
    WinRawDiskReader(char* = NULL);
    ~WinRawDiskReader();
    bool openDisk(char*);
    bool readSector(unsigned long, char*);
  private :
    // Datenkomponenten und private Memberfunktionen
};
```

◇ **Definition einer – abstrakten – Creator-Klasse** (**`Creator`**)

```
class RDRCreator
{ public :
    virtual RawDiskReader* createRDR()=0;   // factory method (Interface)
};
```

◇ **Definition einer konkreten Creator-Klasse** (**`ConcreteCreator`**)

```
class WinRDRCreator : public RDRCreator
{ public :
    RawDiskReader* createRDR();             // factory method
};
```

◇ **Definition einer Anwendungsklasse** (**`Client`**)

```
class DiskDump
{ public :
    DiskDump(RDRCreator*);
    ~DiskDump();
    void doDump(char*, unsigned);
    // weitere öffentliche Memberfunktionen
  private :
    RDRCreator* m_pCre;        // Pointer auf Creator-Objekt
    // weitere Datenkomponenten und private Memberfunktionen
};
```

Entwurfsmuster : Factory Method (3)

- **Implementierungsbeispiel** (Teil 2)

◇ **Implementierung der Klasse `WinRawDiskReader`** (Auszug) (applikationsabhängig)

```
WinRawDiskReader::WinRawDiskReader(char* diskname)
{ if (diskname!=NULL)
    openDisk(diskname);
  else
  { m_hDisk=INVALID_HANDLE_VALUE;
    m_pcDiskName=NULL;
  }
}

WinRawDiskReader::~WinRawDiskReader()
{ CloseHandle(m_hDisk);
  delete [] m_pcDiskName;
  // ...
}

bool WinRawDiskReader::openDisk(char* diskname) { /* ... */ }

bool WinRawDiskReader::readSector(unsigned long sec, char* buff) { /* ... */ }
```

◇ **Implementierung der Klasse `WinRDRCreator`** (applikationsabhängig)

```
RawDiskReader* WinRDRCreator::createRDR()     // factory method
{
  return new WinRawDiskReader();
}
```

◇ **Implementierung der Klasse `DiskDump`** (Auszug) (applikationsunabhängig)

```
DiskDump::DiskDump(RDRCreator* cr)
{ m_pCre=cr;
}

void DiskDump::doDump(char* dname, unsigned size)
{ // ...
  RawDiskReader* pRDR=m_pCre->createRDR();   // Erzeugung eines Reader-Objekts
  if (pRDR->openDisk(dname))
  // ...
}
```

◇ **Beispiel für verwendenden Code** (Funktion **`main()`** eines Programms `diskdump`) (applikationsabhängig)

```
int main(int argc, char** argv)
{ if (argc==1)
    cout << "\nAufruf : diskdump <diskname>\n";
  else
  { RDRCreator* cr = new WinRDRCreator;   // einzige applikationsabhängige Zeile
    DiskDump dd(cr);
    dd.doDump(argv[1], SEC_SIZE);
  }
  return 0;
}
```

Entwurfsmuster : Composite (1)

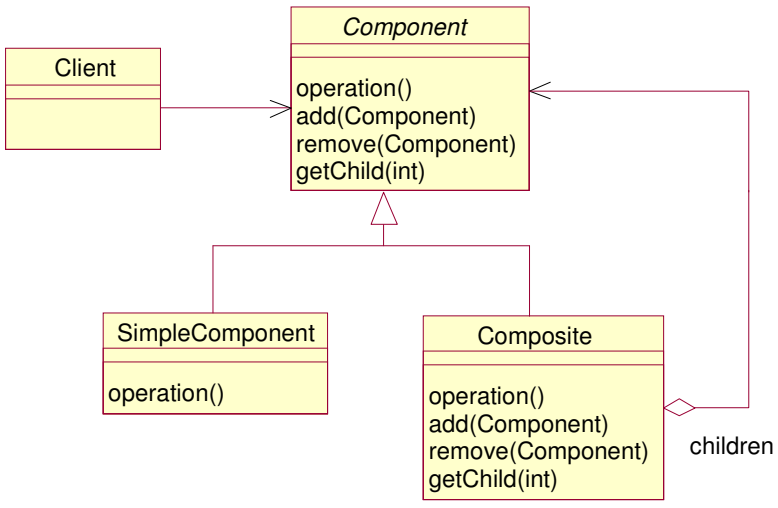
- **Name : Composite** (*Recursive Composition*)

- **Kurzbeschreibung**
  Ermöglicht die Repräsentation von Teil-Ganzheits-Beziehungen durch den Aufbau baumartiger Objektstrukturen. Dadurch können Clients individuelle Objekte und Zusammensetzungen von Objekten gleichartig behandeln.

- **Problembeschreibung**
  In vielen Anwendungsbereichen treten Strukturen auf, die aus einfachen Objekten und zusammengesetzten Objekten (Containern) aufgebaut sind. Dabei können i.a. die zusammengesetzten Objekte wiederum zusammengesetzte Objekte als Komponenten enthalten ⇒ Rekursive Teil-Ganzheits-Hierarchien.
  (Beispiel : Fenster-System → Ein Fenster enthält u.a. Text, einfache Steuerungselemente und "Unter"-Fenster).
  Es ist sinnvoll und effektiv, wenn Code, der die entsprechenden Klassen benutzt (Client), einfache Objekte und zusammengesetzte Objekt völlig gleichartig behandeln kann, d.h. beim Aufruf von Komponenten-Methoden nicht zwischen einfachen und zusammengesetzten Objekten unterscheiden muß.

- **Problemlösung**
  Die Klassen für einfache Komponenten und für zusammengesetzte Komponenten werden von einer gemeinsamen abstrakten Basisklasse **`Component`** abgeleitet.
  Diese definiert sowohl das "allgemeine" Komponenten-Anwendungs-Interface als auch das spezielle "Management"-Interface für zusammengesetzte Komponenten.
  Von der gemeinsamen Komponenten-Basisklasse **`Component`** können durchaus mehrere Klassen für einfache Komponenten abgeleitet sein (im untenstehenden Klassendiagramm nur die Klasse **`SimpleComponent`**).
  Die Klasse für zusammengesetzte Komponenten **`Composite`** kann selbst wieder eine (abstrakte) Basisklasse für mehrere (konkrete) Composite-Klassen sein.
  Die Klasse für zusammengesetzte Komponenten **`Composite`** verwaltet ihre enthaltenen Objekte (Kindkomponenten, "children") als Instanzen der gemeinsamen Komponenten-Basisklasse **`Component.`**
  An sie gerichtete Anwendungs-Methodenaufrufe ("`operation()`") delegiert sie an alle in ihr enthaltenen Komponenten.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Client-Code kann immer dann, wenn er ein einfaches Objekt erwartet auch mit einem zusammengesetzten Objekt arbeiten.
  - ▻ Da Clients einfache und zusammengesetzte Objekte gleich behandeln können, vereinfacht sich der Client-Code (z. B. keine `switch-case`-Anweisungen zu ihrer Unterscheidung notwendig).
  - ▻ Neue Komponenten-Klassen können einfach ohne Änderungen des Client-Codes hinzugefügt werden.
  - ▻ Andererseits ist es schwierig, Kindkomponenten auf bestimmte Typen zu begrenzen (→ zusätzliche Laufzeitüberprüfungen notwendig).

Entwurfsmuster : Composite (2)

- **Implementierungsbeispiel** (Teil 1)

◇ **Definition einer Komponenten-Basisklasse** (**`Component`**) **:**

```
class Equipment
{
  public :
    virtual ~Equipment ();
    const char* getName() const { return m_szName; }
    virtual double getPrice() const = 0;
    // weitere Anwendungs-Memberfunktionen
    virtual void add(Equipment*);
    virtual void remove(Equipment*);
    virtual Iterator<Equipment*>* createIterator() const;
  protected :
    Equipment (const char*);
  private :
    const char* m_szName;
};
```

◇ **Definition einer einfachen Komponentenklasse** (**`SimpleComponent`**) **:**

```
class BasicPart : public Equipment
{
  public :
    BasicPart(const char*, double);
    ~BasicPart();
    virtual double getPrice() const;
    // weitere Anwendungs-Memberfunktionen
  private :
    double m_dPrice;
};
```

◇ **Definition einer zusammengesetzten Komponentenklasse** (**`Composite`**) **:**

```
class CompositeEquipment : public Equipment
{
  public :
    CompositeEquipment(const char*);
    virtual ~CompositeEquipment();
    virtual double getPrice() const;
    // weitere Anwendungs-Memberfunktionen
    virtual void add(Equipment*);
    virtual void remove(Equipment*);
    virtual Iterator<Equipment*>* createIterator() const;
  private :
    List<Equipment*> *m_pclEquipment;
};
```

Entwurfsmuster : Composite (3)

- **Implementierungsbeispiel** (Teil 2)

◇ **Implementierung der Klasse `Equipment` (Auszug):**

```
Equipment:: Equipment(const char* name)
{
  m_szName=name;
}
```

◇ **Implementierung der Klasse `BasicPart` (Auszug):**

```
BasicPart::BasicPart(const char* name, double price) : Equipment(name)
{
  m_dPrice=price;
}

double BasicPart::getPrice() const
{
  return m_dPrice;
}
```

◇ **Implementierung der Klasse `CompositeEquipment` (Auszug):**

```
CompositeEquipment::CompositeEquipment(const char* name) :
  Equipment(name), m_pclEquipment(0) { }

double CompositeEquipment::getPrice() const
{
  Iterator<Equipment*>* i = createIterator();
  double sum=0.0;
  for (i->first(); i->hasMore(); i->next())
    sum+=i->current()->getPrice();
  delete i;
  return sum;
}
```

◇ **Beispiel für Anwendungscode :**

```
CompositeEquipment* cabinet     = new CompositeEquipment("PC Cabinet");
CompositeEquipment* motherboard = new CompositeEquipment("PC Motherboard");
BasicPart* emptyCabinet  = new BasicPart("Cabinet (empty)", 110.00);
BasicPart* emptyBoard    = new BasicPart("ASUS-Board Pentium II 350", 370.00 );

cabinet->add(emptyCabinet);
cabinet->add(motherboard);
motherboard->add(emptyBoard);

CompositeEquipment* bus = new CompositeEquipment("PCI-Bus");
bus->add(new BasicPart("ATI Graphic Card", 50.00));
motherboard->add(bus);
motherboard->add(new BasicPart("3.5 Floppy Disc", 65.00));

cout << endl << "The total price is : " << cabinet->getPrice() << endl;
```

Entwurfsmuster : Strategy (1)

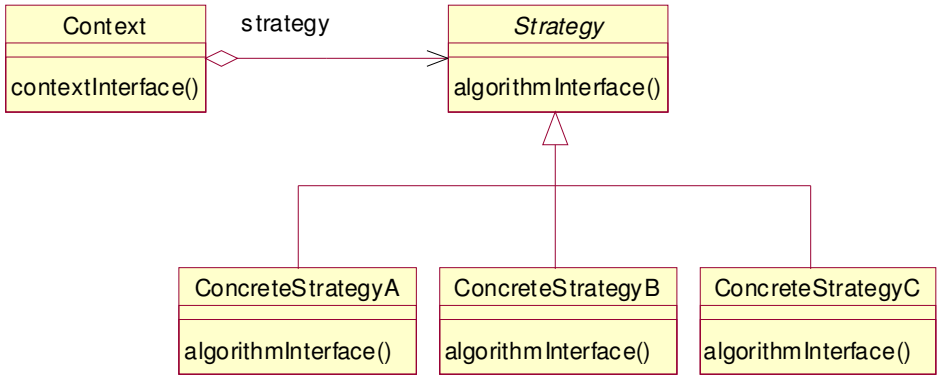
- **Name : Strategy** (*Policy*)

- **Kurzbeschreibung**
  Kapselt verwandte Algorithmen in jeweils einer eigenen Klasse, die von einer gemeinsamen Basisklasse abgeleitet ist. Dies ermöglicht die Auswahl und Änderung der Algorithmen zur Laufzeit ohne dass Änderungen an den Clients, die sie benutzen, notwendig sind.

- **Problembeschreibung**
  Häufig sollen zur Realisierung bestimmter Funktionalitäten einer Klasse unterschiedliche Algorithmen einsetzbar sein. (Beispiel : Unterschiedliche Auswahlstrategien eines Workflow-Managers zur Bestimmung der nächsten zu bearbeitenden Task.)
  Eine Codierung sämtlicher möglichen Algorithmen in einer Klasse führt zu einer aufwendigen, umständlichen und wenig änderungsfreundlichen Implementierung.
  Die Erzeugung mehrerer ähnlicher, sich nur im jeweils verwendeten Algorithmus unterscheidenden, Klassen stellt ebenfalls eine unhandliche und inflexible Lösung dar.

- **Problemlösung**
  Auslagerung der Funktionalität in eine eigene (abstrakte) Basisklasse **`Strategy`**, von der für jeden in Frage kommenden Algorithmus eine eigene konkrete Strategie-Klasse abgeleitet wird (**`ConcreteStrategyA`**, **`ConcreteStrategyB`** usw).
  Die Klasse, die diese Funktionalität eigentlich ausführen soll, (**`Context`**) wird mit einer Referenz (oder einem Pointer) auf ein konkretes Strategie-Objekt konfiguriert, an das sie Anforderungen zur Ausführung der betreffenden Funktionalität weiterleitet.

- **Struktur**



- **Anwendungskonsequenzen**
  - ▻ Die Isolierung der Algorithmen-"Familie" in einer eigenen Klassenhierarchie erlaubt ihre einfachere Wiederverwendung.
  - ▻ Das Entwurfsmuster stellt eine sehr flexible Alternative zur Vererbung dar : Es wird nur eine Kontext-Klasse benötigt die ohne Änderungen an ihr vorzunehmen, mit unterschiedlichen Strategien konfiguriert werden kann.
  - ▻ Die jeweils verwendete Strategie kann leicht – auch zur Laufzeit - ausgetauscht werden.
  - ▻ Es ist möglich, daß das von der Klasse **`Strategy`** definierte Interface, Parameter enthält, die nicht von allen konkreten Strategie-Objekten benötigt werden.⇒ Kommunikations-Overhead zwischen **`Context`** und **`Strategy`**
    Abhilfe : **`Context`** definiert ein Interface für den Zugriff durch **`Strategy`**, über das sich **`Strategy`** nur die wirklich benötigte Information beschafft. (→ engere Kopplung zwischen **`Strategy`** und **`Context`**).

Entwurfsmuster : Strategy (2)

- **Implementierungsbeispiel** (Teil 1)

  ◇ **Definition einer abstrakten Strategy-Basisklasse (`Strategy`) :**

```
class TaskScheduler
{
  public :
    virtual ~TaskScheduler();
    virtual Task* determineNextTask(TaskList*) = 0;
    // gegebenenfalls weitere Memberfunktionen
  protected :
    TaskScheduler(WFManager*);
  private :
    WFManager* m_pclManager;
};
```

  ◇ **Definition einer konkreten StrategyKlasse (`ConcreteStrategyA`) :**

```
class SimpleTaskScheduler : public TaskScheduler
{
  public :
    SimpleTaskScheduler(WFManager*);
    ~SimpleTaskScheduler();
    virtual Task* determineNextTask(TaskList*);
    // gegebenenfalls weitere Memberfunktionen
  private :
   // gegebenenfalls weitere Datenkomponenten;
};
```

  ◇ **Definition einer Context-Klasse (`Context`) :**

```
class WFManager
{
  public :
    WFManager();
    WFManager(TaskScheduler*);
    ~WFManager();
    void setTaskScheduler(TaskScheduler*);
    // weitere Memberfunktionen
  private :
    void handleWaitingTasks();
    TaskScheduler* m_pclScheduler;
    TaskList* m_pclTasks;
    // weitere Datenkomponenten
};
```

Entwurfsmuster : Strategy (3)

- **Implementierungsbeispiel** (Teil 2)

  ◇ **Implementierung der Klasse `TaskScheduler`** (Auszug)**:**

```
TaskScheduler ::TaskScheduler(WFManager* manager)
{
  m_pclManager=manager;
}
```

  ◇ **Implementierung der Klasse `SimpleTaskScheduler`** (Auszug)**:**

```
SimpleTaskScheduler::SimpleTaskScheduler(WFManager* manager) :
  TaskScheduler(manager) { }

Task* SimpleTaskScheduler::determineNextTask(TaskList* tasks)
{ Task* nextTask;
  // Implementierung eines einfachen Auswahlalgorithmus (z.B.FIFO)
  return nextTask;
}
```

  ◇ **Implementierung der Klasse `WFManager`** (Auszug)**:**

```
WFManager::WFManager()
{
  // Realisierung des Default-Konstruktors
}

WFManager::WFManager(TaskScheduler* scheduler)
{
  m_pclScheduler=scheduler;
  // restliche Initialisierung
}

WFManager::~WFManager()
{
  delete m_pclScheduler;
  // weitere Funktionalität des Destruktors
}

void WFManager::setTaskScheduler(TaskScheduler* scheduler)
{ if (m_pclScheduler!=0)
    delete m_pclScheduler;
  m_pclScheduler=scheduler;
}

void WFManager::handleWaitingTasks()
{ // ...
  Task* nextTask;
  nextTask=m_pclScheduler->determineNextTask(m_pclTasks);
  // ...
}
```

Entwurfsmuster : Observer (1)

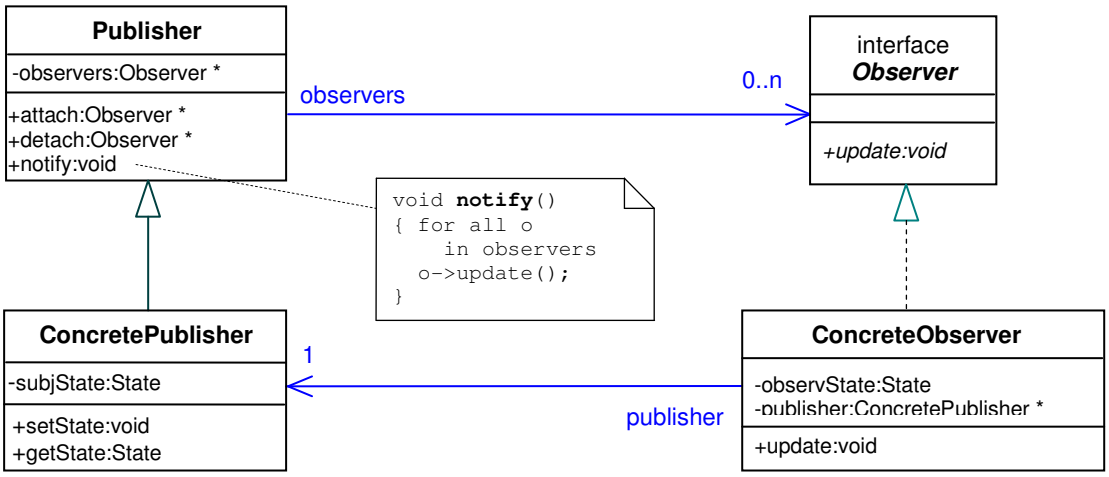
- **Name : Observer** (*Publish-Subscribe*, *Dependents*)

- **Kurzbeschreibung**
  Ermöglicht die **dynamische Registrierung** von **Objektabhängigkeiten**, so dass bei einem **Zustandswechsel** eines Objekts alle von ihm **abhängigen Objekte benachrichtigt** werden.

- **Problembeschreibung**
  In vielen Anwendungsbereichen soll sich der Zustandswechsel eines Objekts direkt auf den Zustand bzw. das Verhalten anderer Objekte auswirken.
  Beispiel : Ein Datenbestand und seine – u.U. gleichzeitige – Darstellung in verschiedenen Formaten in einem GUI-System (z. B. Tabelle, Balkendiagramm, Tortendiagramm). Jede Änderung des Datenbestandes muss sich umgehend auf alle Darstellungen auswirken.
  Eine enge Kopplung der beteiligten Objekte (→ die beteiligten Objekte haben voneinander Kenntnis) ist meist nicht wünschenswert, da dadurch die unabhängige Verwendung und Modifikation ihrer jeweiligen Klassen stark eingeschränkt wird. Außerdem müssen in diesem Fall die Abhängigkeiten bereits zur Compilezeit bekannt sein, was eine flexible dynamische Anpassung zur Laufzeit verhindert.

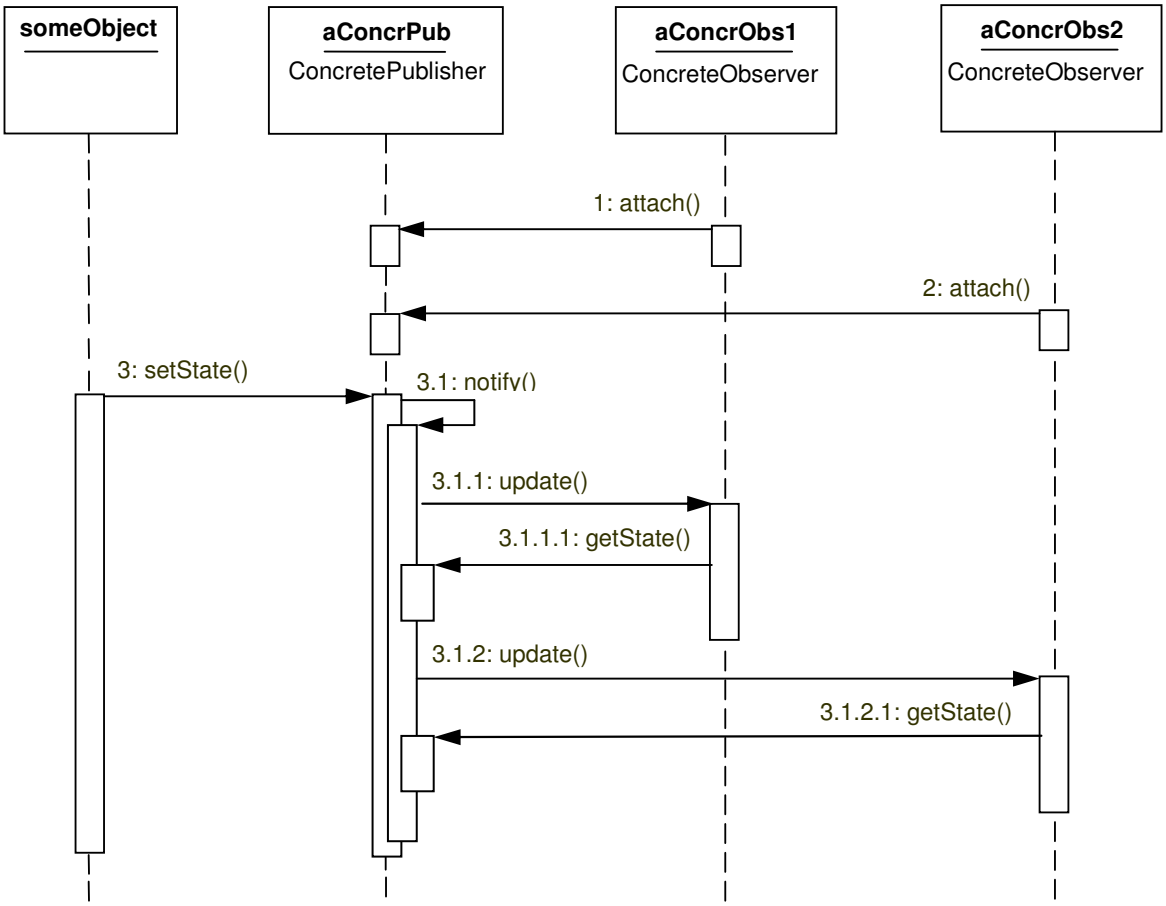
- **Problemlösung**
  Bei dem Objekt, von dessen Zustand andere Objekte abhängen (***Publisher*-Objekt**), wird eine **Liste der abhängigen Objekte** geführt. In diese Liste tragen sich alle Objekte, die über den Zustand dieses Objekts auf dem laufenden gehalten werden wollen (weil sie von ihm abhängen), ein (***Subscriber*-Objekte**, ***Observer*-Objekte**).
  Bei einem **Wechsel** seines Zustands **informiert** das *Publisher*-Objekt alle eingetragenen *Observer*-Objekt hierüber (***notify***). Diese können dann den neuen Zustand vom *Publisher*-Objekt ermitteln und entsprechend der eingetretenen Änderung reagieren (z.B. ihren eigenen Zustand an den Zustand des *Publisher*-Objektes anpassen).
  Die Anzahl der *Observer*-Objekte muss dem *Publisher*-Objekt a priori nicht bekannt sein. Bei ihm können sich zur Laufzeit beliebig viele *Observer*-Objekte an- bzw. auch wieder abmelden.
  Publisher-Objekt und *Observer*-Objekte sind von einander entkoppelt und können unabhängig voneinander modifiziert werden.
  Die ***Publisher*-Schnittstelle** wird durch eine geeignete Klasse (**`Publisher`**) zur Verfügung gestellt. Diese Klasse dient als **Basisklasse** für konkrete *Publisher*-Klassen (**`ConcretePublisher`**), die die Datenkomponenten für den jeweiligen Zustand und die Methoden zum Setzen und Ermitteln derselben bereitstellen.
  Das **Interface** zum Informieren von *Observer*-Objekten wird durch eine abstrakte Klasse (***`Observer`***) definiert
  Dieses Interface wird von konkreten *Observer*-Klassen (**`ConcreteObserver`**) implementiert. Objekte dieser Klassen enthalten – neben ihren eigentlichen Zustands-Datenkomponenten – eine Referenz auf das konkrete *Publisher*-Objekt, bei dem sie sich eingetragen haben.

- **Struktur**



Entwurfsmuster : Observer (2)

- **Ablauf der Interaktion**



- **Anwendungskonsequenzen**
  - ***Publisher*** und ***Observer*** können **unabhängig** voneinander **geändert** und **ausgetauscht** werden.
    Publisher können wiederbenutzt werden, ohne ihre *Observer* wiederzubenutzen und umgekehrt.
    Observer können hinzugefügt werden, ohne den *Publisher* oder andere *Observer* zu verändern.
  - Die **Kopplung** zwischen *Publisher* und *Observer* ist **abstrakt** und **minimal**.
    Ein *Publisher*-Objekt weiß lediglich, dass es eine Liste von *Observer*-Objekten besitzt, die das *Observer*-Interface implementieren. Es kennt nicht die konkrete Klasse seiner *Observer*.
    Publisher und Observer können unterschiedlichen Abstraktions-Ebenen (Schichtenmodell !) angehören.
  - *Observer*-Objekte können gegebenenfalls selbst auch einen Zustandswechsel beim *Publisher*-Objekt bewirken
  - Die Information über den Zustandswechsel (**Notifikation**) durch das *Publisher*-Objekt ist automatisch eine **Broadcast-Kommunikation**. Es werden immer alle eingetragenen *Observer*-Objekte informiert, unabhängig davon, wieviel es sind. Es obliegt einem *Observer*-Objekt, eine Notifikation gegebenenfalls zu ignorieren.
  - Falls **sehr viele** *Observer*-Objekte eingetragen sind, kann eine Notifikation und der dadurch ausgelöste *Observer*-Update u.U. eine **längere Zeit** dauern.
  - Da das eine Zustandsänderung bewirkende Objekt keine Information über die *Observer* und die mit diesen verbundenen "Update"-Kosten hat, können **"leichtfertige" Zustandsänderungen** einen erheblichen zeit- und damit kostenintensiven **Aufwand** bewirken.
  - Das **einfache Notifikations-Protokoll** liefert **keine Informationen** darüber, **was** sich im Publisher-Objekt **geändert** hat. Dies festzustellen, obliegt dem Observer-Objekt, was den Update-Aufwand gegebenenfalls noch erhöht.
    Als Alternative kann u.U. ein **erweitertes Protokoll** definiert werden, dass detailliertere Zustandsänderungs-Info liefert.
  - Es gibt Fälle, bei denen sinnvoll ist, dass sich **ein *Observer***-Objekt bei **mehreren *Publisher*-**Objekten für eine Notifikation einträgt. In diesen Fällen muss die Update-Botschaft eine Information über das absendende Publisher-Objekt enthalten

Entwurfsmuster : Observer (3)

- **Implementierungsbeispiel** (Teil1)

  ◇ **Definition einer Publisher-Basisklasse** (Publisher)

```
class Publisher
{ public :
    virtual ~Publisher() {};
    virtual void attach(Observer*);
    virtual void detach(Observer*);
    virtual void notify();
  protected :
    Publisher() {};
  private :
    Set<Observer*> observs;       // Liste der eingetragenen Observer-Objekte
};
```

  ◇ **Implementierung der Klasse `Publisher`**

```
void Publisher::attach(Observer* no)
{ observs.insert(no);
}

void Publisher::detach(Observer* no)
{ observs.remove(no);
}

void Publisher::notify()
{ Observer** ppo;
  for(ppo=observs.first(); ppo; ppo=observs.next())
    (*ppo)->update(this);
}
```

  ◇ **Definition einer konkreten Publisher-Klasse** (ConcretePublisher)

```
class ClockTimer : public Publisher
{ public :
    ClockTimer();
    ~ClockTimer();
    int getHour();
    int getMinute();
    int getSecond();
    void tick();          // Veränderung des Objekt-Zustands
  private :
    // Datenkomponente(n) zur Speicherung der Zeit
};
```

  ◇ **Implementierung der Klasse `ClockTimer`**

```
// Implementierung der uebrigen Memberfunktionen

void ClockTimer::tick()
{
  // update der privaten Datenkomponente(n) zur Speicherung der Zeit
  notify();
}
```

Entwurfsmuster : Observer (4)

- **Implementierungsbeispiel** (Teil2)

◇ **Definition einer abstrakten Observer-Basisklasse** (*Observer*)

```
class Publisher;

class Observer
{ public :
    virtual ~Observer() { };
    virtual void update(Publisher*) = 0;
  protected :
    Observer() {};
};
```

◇ **Definition einer konkreten Observer-Klasse** (ConcreteObserver)

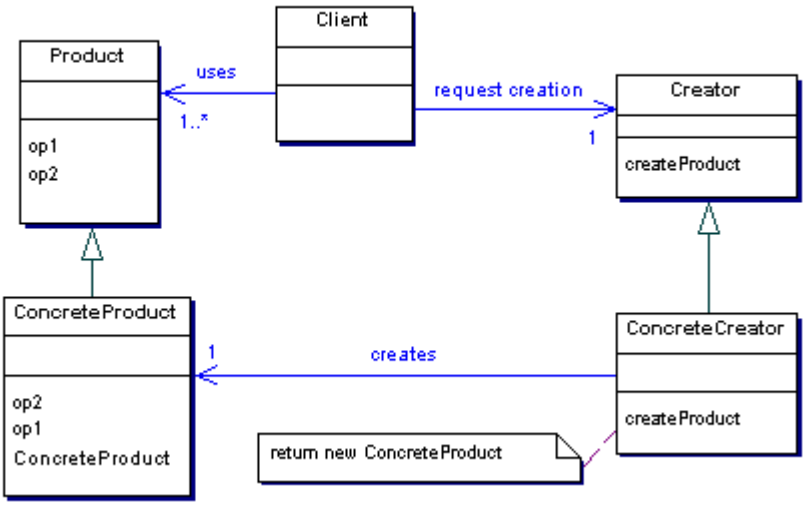
```
class Widget;      // Fenster-Basisklasse mit graph. Fähigkeiten
class ClockTimer;

class DigitalClock : public Observer, public Widget
{ public :
    DigitalClock(ClockTimer*);
    ~DigitalClock();
    void update(Publisher*);
    void draw();             // virtuelle Methode von Widget
  private :
    ClockTimer* m_publ;
};
```

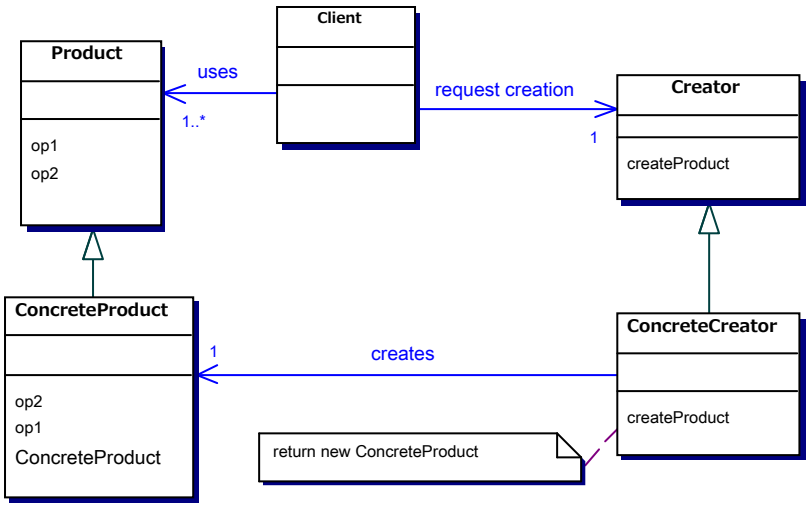
◇ **Implementierung der Klasse `DigitalClock`**

```
DigitalClock::DigitalClock(ClockTimer* ct)
{ m_publ=ct;
  m_publ->attach(this);
}

DigitalClock::~DigitalClock()
{ m_publ->detach(this);
}

void DigitalClock::update(Publisher* pub)
{ if (pub==m_publ)
    draw();
}

void DigitalClock::draw()              // virtuelle Methode von Widget
{ int h = m_publ->getHour();
  int m = m_publ->getMinute();

  // Zeichnen der Digital-Uhr
}
```

◇ **Beispiel für Anwendungscode** (Auszug)

```
  ClockTimer timer;
  DigitalClock digClock(&timer);
  // bei jedem Aufruf von timer.tick() stellt digClock die neue Zeit dar
```


Client
Product
uses
1..*
request creation
1
Creator
op1
op2
createProduct
ConcreteProduct
1
creates
ConcreteCreator
op2
op1
ConcreteProduct
createProduct
return new ConcreteProduct


Client
Product
uses
1..*
request creation
1
Creator
op1
op2
createProduct
ConcreteProduct
1
creates
ConcreteCreator
op2
op1
ConcreteProduct
createProduct
return new ConcreteProduct