Materialpaket 08_PTRN – Patterns

C/C++, Autor: Prof. Dr.-Ing. Carsten Link

Version 1.3.1 March 3, 2019

Contents

1 Kompetenzen und Lernegebnisse

Durch das Bearbeiten dieses Materialpaketes erwerben Sie diese Kompetenzen (Wissen, Fähigkeiten, Fertigkeiten zur Problemlösung):

Sie können Quellcode derart strukturieren, dass er gängige Muster einsetzt.

Die oben genannten Kompetenzen erwerben Sie, indem Sie Lernziele erreichen, welche sich prüfen lassen. Lernegebnisse: Sie können nachweislich[1]:

- Zweck und Nutzen von Idiomen und Design Patterns erläutern

[1] Sie können das Erzielen der einzelnen Lernergebnisse beispielsweise bei einem Testat im Praktikum oder einer Aufgabe in der Modulprüfung nachweisen

- einige wichtige C++-Idiome anwenden
- einige wichtige Design Patterns in C++ anwenden
- aus einer umgangssprachlichen Beschreibung eines Sachverhaltes die Struktur eines dazugehörigen C++-Programmes herleiten und diese implementieren

2 Konzepte

In diesem Materialpaket werden aus dem Themenbereich Softwaredesign die Konzepte Software-Architektur, C++-Idiome sowie Design Patterns behandelt.

2.1 Software-Architektur

Generelle Leitlinien:

- *Divide and Conquer:* das Problem in kleinere Teilprobleme zerlegen, die so klein sind, dass sie leicht zu beherrschen sind
- *Modularisierung:* kleine, in sich geschlossene Komponenten entwickeln, welche eine nach außen sichtbare Schnittstelle haben. Hierbei soll ein Modul wenig von anderen Modulen abhängen (Ziel: geringe Kopplung) und nur eigene Daten und Berechnungen verwenden (Ziel: hohe Köhäsion)
- *Separation of Concerns:* Zuständigkeiten sollten nicht auf mehrere Module verteilt werden
- *Encapsulation, Separation of Interface and Implementation:* die Schnittstelle von Klassen sollte aus Methoden bestehen und nicht interne Strukturen/Mechanismen offenlegen
- *Programming against Interfaces not Implementations:* es sollte gegen Klassen programmiert werden, die sich weit oben in der Vererbungshierarchie befinden. Oft sind diese Klassen abstrakt (in Java können Interfaces verwendet werden). Auf diese Weise entfallen viele Typabhängigkeiten

2.2 Design Patterns

Design Patterns sind programmiersprachenunabhängige architekturelle Muster, die sich in vielen Softwareprojekten wiederfinden. Im Standardwerk [DPTRN] hierzu sind 23 klassische Softwaremuster aufgeführt. Hier einige wichtige Software Design Patterns(siehe auch[2]):

- Singleton[3]: Klasse, von der nur ein einziges Objekt erzeugt wird, welches an vielen Stellen im Programm verwendet wird

[2] https://en.wikipedia.org/wiki/Software_design_pattern
[3] https://sourcemaking.com/design_patterns/singleton

- Factory: Erzeugung von Objekten, wobei die konkrete Klasse von externer Information abhängt
- Facade[4]: bündelt eine oder mehrer externe Schnittstellen
- Observer[5]: Propagation von Objektänderungen
- Interpreter[6]: Unterstützung programmspezifischer Sprachen
- Proxy[7]: wird einem Objekt vorgeschaltet, um beispielsweise Zugriffe zu kontrollieren
- Model-View-Controller[8]: Trennung von Daten und deren Darstellung/Bearbeitung
- Dependency injection[9]: Klassen, von denen Objekte erzeugt werden sollen, werden nicht bei `new` angegeben, sondern in einer Konfigurationsdatei. Programmierung erfolgt so gegen Interfaces; nicht gegen konkrete Klassen

Die beiden Patterns *Singleton* und *Factory* werden im Folgenden kurz besprochen.

2.2.1 Singleton

In Beispielprojekt `PTRN` wird das Singleton-Pattern in den Dateien `Logger.hpp` und `Logger.cpp` vereinfacht angewendet:

```
1  // file Logger.hpp
2  #ifndef Logger_hpp
3  #define Logger_hpp
4
5  #include <iostream>
6
7  class Logger {
8    static Logger* _theInstance;
9  public:
10   virtual void log(std::string message, std::string file="", int line=0) = 0;
11   static Logger* getInstance();
12 };
13
14 class StdoutLogger: public Logger {
15 public:
16   void log(std::string message, std::string file="", int line=0);
17 };
18
19 #endif
```

[4] https://sourcemaking.com/design_patterns/facade
[5] https://sourcemaking.com/design_patterns/observer
[6] https://sourcemaking.com/design_patterns/interpreter
[7] https://sourcemaking.com/design_patterns/proxy
[8] https://developer.apple.com/library/content/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html
[9] https://de.wikipedia.org/wiki/Dependency_Injection

Die statische Methode `Logger::getInstance()` gibt dem verwendenden Code Zugriff auf das einzige existierende `Logger`-Objekt. Dieses ist vom Typ einer Klasse, welche von `Logger` abgeleitet ist – in diesem Fall `StdoutLogger`, welche auf std::cout die Log-Nachrichten ausgibt. Andere Subklassen sind denkbar, die verschiedene Kanäle zur Log-Ausgabe verwenden.

In folgendem Code ist ersichtlich, dass die Singleton-Ausprägung (-Instanz) bei der ersten Verwendung angelegt wird:

```
1  // file Logger.cpp
2  #include "Logger.hpp"
3
4  Logger* Logger::_theInstance = nullptr;
5
6  Logger* Logger::getInstance(){
7    if (!_theInstance){
8      _theInstance = new StdoutLogger();
9    }
10   return _theInstance;
11 }
12
13 void StdoutLogger::log(std::string message, std::string file, int line){
14   std::cerr << message << " in " << file << " line " << line << std::endl;
15 }
```

Der Logger kann nun (wie in `main()`) so verwendet werden:

```
1    Logger::getInstance()->log("work naively", __FILE__, __LINE__);
```

Die beiden Makros `__FILE__` und `__LINE__` sorgen für die Angabe der Quelltextdatei und der Zeilennummer:

```
work naively in ./IdiomsAndPatterns/main.cpp line 66
```

2.2.2 Factory

Die einfachste Form des Factory-Patterns wird in diesem Artikel[10] behandelt.

In `main()` wird die Factory-Funktion `vehicleForPayload()` verwendet, um abhängig von Daten (hier: dem geforderten Gewicht) die Klasse des zu erzeugenden Objektes herauszusuchen:

```
1    vehicles.push_back(vehicleForPayload("Magirus Deutz", 18*1000));
```

Dabei ist diese Funktion wie folgt definiert:

```
1  /*
2    A factory creates objects based un runtime information
```

[10] https://en.wikipedia.org/wiki/Factory_(object-oriented_programming)

```
   a) within the process (data values, Types i.e.e virtual methods)
   b) input from files or user input
   Th is simple Factory differentiates only based on a value.
 */
Vehicle* vehicleForPayload(std::string model, int weight){
  if (weight < 3500){
    return new Car(model, weight);
  }else{
    return new Truck(model, weight);
  }
}
```

2.3 C++-Idiome RAII und Virtual Constructor

Das Wort *Idiom* kommt aus den Sprachwissenschaften und meint eine Spracheigenheit.

Idiome ähneln Design Patterns dahingehend, dass sie Muster beschrieben, die in Software vorkommen. Idiome sind allerdings sprachspezifisch, betreffen also Muster im C++-Quelltext. Dementsprechend sind Idiombeschreibungen auch nicht so umfangreich wie Patternbeschreibungen.

Ein kleines Beispiel soll die Verwendung der C++-Idiome RAII und Virtual Constructor verdeutlichen. Zunächst eine naive Implementierung einer Methode, die sich Kopien von Objekten erzeugt, um darauf zu arbeiten ohne die Originale zu verändern.

```
void workOnCopy_naive(std::vector<Vehicle*> vehicles){
  std::vector<Vehicle*> tempVec;
  for (int i=0; i<vehicles.size(); i++){
    Vehicle * currentVehicle = vehicles[i];
    Car*   carToCopy   = dynamic_cast<Car*>(currentVehicle);  // RTTI type check
    Truck* truckToCopy = dynamic_cast<Truck*>(currentVehicle);// may be nullptr
    if(carToCopy){
      tempVec.push_back(new Car(carToCopy->model(), carToCopy->maxWeight())); // type depend
    }else if(truckToCopy){
      tempVec.push_back(new Truck(truckToCopy->model(), truckToCopy->payload_kg())); // type
    }else{
      // TODO: what do we do here?
      // Did someone add a new class to the code base?
      return; // BUG: copies aren't free'd
    }
  }
  // work on copies ...
  // ...
  for(auto vehi : tempVec){
```

```
20      std::cout << vehi->model() << " " << vehi->payload_kg() << " kg" << std::endl;
21    }
22    for (int i=0; i<tempVec.size(); i++){
23      delete tempVec[i];
24    }
25    tempVec.clear();
26  }
```

Der Code oben weist einige gravierende Probleme auf:

1. Die Ausdrücke `new Car()` und `new Truck()` ergeben Abhängigkeiten zu den Klassen `Car` und `Truck`. Dies ist unnötig, da der Rest des Codes nur mit der abstrakteren Oberklasse `Vehicle` auskommt
2. Wird eine weitere Klasse von `Vehicle` abgeleitet, so muss der Code angepasst werden (siehe Kommentar `TODO:`)
3. Es besteht die Möglichkeit, dass die erzeugten Kopien nicht freigegeben werden, da die Freigabe am Ende der Funktion mit `return` übersprungen wird (siehe Kommentar `BUG:`)

Die ersten beiden Probleme werden vom Idiom *virtual constructor* gelöst; das dritte Problem vom Idiom *RAII*. Beide werden nun vorgestellt.

Das Idiom *Resource acquisition is initialization (RAII)* ermöglicht es, Ressourcen garantiert freizugeben, auch wenn Teile des Codes übersprungen werden (z.B. durch Exceptions). Hierbei wird die Tatsache ausgenutzt, dass der Compiler garantiert, dass lokale Objekte bei Verlassen des Gültigkeitsbereichs zerstört werden – die Ressourcenfreigabe geschieht dann im Destruktor des automatisch zerstörten Objektes.

Im folgenden Code ist das RAII-Idiom mittels der lokalen Klasse `RAIIvector` umgesetzt:

```
1   void workOnCopy_smart(std::vector<Vehicle*> vehicles){ // uses RAII, virtual ctor
2
3     class RAIIvector { // local RAII helper class
4     public:
5       std::vector<Vehicle*> tempVec;
6       ~RAIIvector(){
7         for(auto vehi : tempVec){
8           delete vehi;
9         }
10        tempVec.clear();
11      }
12    };
13    RAIIvector rv;
14
15    for(auto vehi : vehicles){
16      rv.tempVec.push_back(vehi->clone());
17    }
```

```
18    // work on copies ...
19    // ...
20    for(auto vehi : rv.tempVec){
21      std::cout << vehi->model() << " " << vehi->payload_kg() << " kg" << std::endl;
22    }
23    // compiler WILL invoke RAIIvector::~RAIIvector() here
24  }
```

Der Compiler sorgt dafür, dass das lokale Objekt `rv` zerstört wird, sobald die Funktion verlassen wird – egal wie. Dies impliziert, dass der Destruktor von `RAIIvector` die in `tempVec` enthaltenen Objekte mit `delete` löschen kann.

Das Idiom virtual constructor spiegelt sich im Aufruf der virtuellen Methode `clone()` wider. Typabhängigkeiten entfallen, da der aufrufende Code keine Kenntnis über den konkreten Laufzeittyp des Objektes haben muss, das am Pointer `vehi` hängt. Da `clone()` in `class Car` und `class Truck` jeweils überschrieben ist, wird eine typgleiche Kopie erstellt, wie beispielsweise in `Truck.cpp` und `Car.cpp`:

```
1  Vehicle* Truck::clone(){
2    return new Truck(model(),_payload_kg);
3  }
4
5  Vehicle* Car::clone(){
6    return new Car(model(), _maxWeight);
7  }
```

Das Idiom virtual constructor löst auch das Problem, dass der aufrufende Code die Klasse des zu erzeugenden Objektes womöglich nicht kennt – also der Klassenname `X` für `new X()` unbekannt ist.

```
1
```

3 Material zum aktiven Lernen

Da eine Programmiersprache nur durch aktive Verwendung erlernt werden kann, werden im folgenden Aufgaben zum praktischen Üben vorgestellt. Zunächst wird ein Grundgerüst (C++Programm) erstellt, welches dann auf mehrere Arten Modifiziert wird. Insbesondere die Modifikationen ermöglichen es dem Lernenden (und auch dem Lehrenden), die Qualität des Kompetenzerwerbs bzgl. dieses Materialpakets bewerten zu können.

3.1 Aufgabe: Grundgerüst

Sie sollen eine C++Bibliothek erstellen, die es ermöglicht, einfache Simulationen von Geldflüssen zu Programmieren. Sie stellen also Simulationsprogrammierern

Abstraktionen, Schnittstellen und Algorithmen zur Verfügung, die einerseits die Problemdomäne angemessen modellieren und andererseits anhand gängiger C++-Vorgehensweisen einfach zu benutzen ist.

Szenario: Als Zahlungsmittel hat sich Bargeld etabliert. Dieses tritt in Form von Geldscheinen und Münzen in Erscheinung. Geldscheine haben eine Seriennummer, Münzen nicht. Bargeld kann auf ein Konto eingezahlt werden, welches bei einer Bank (Finanzinstitut) geführt wird. Kontostände haben keine Seriennummer.

Optionale Erweiterungen: Banken werden von einer Zentralbank mit Geld versorgt (ggf. eine pro Land). Es existieren verschiedene Währungen. Geld verschiedener Währungen kann eingetauscht werden; hierbei kann zur Ermittlung des Wechselkurses der Preis eines Big Macs in Landeswährung herangezogen werden (siehe[11] [12] und[13]). Der Wert eines Kontostands, einer Münze oder eines Geldscheines wird numerisch mit einer Festkommazahl (Festkomma – nicht Fließkomma) mit zwei Stellen hinter dem Komma dargestellt.

Hinweis: betreiben Sie kein Over-Engineering! Oft reicht ein einfaches Sprachkontrukt aus. Ebenso muss Ihre Lösung die Realwelt nur grob (und näherungsweise) modellieren.

Das Grundgerüst können Sie in Einzel- oder Gruppenarbeit erstellen. Nutzen Sie zur Klärung der Details der Anforderungen das Diskussionsforum. Klärungsbedarf besteht hier:

- Welche Begriffe aus der Beschreibung sollen von C++-Abstraktionen repräsentiert werden?
- Welche C++-Sprachmittel eignen sich jeweils?
- Wie hängen die gewählten Abstraktionen strukturell zusammen (Abhängigkeiten, Vererbung)
- Welche Interaktionen sollen zur Laufzeit möglich sein?
- Können Idiome und Patterns helfen? Welche?

Verwenden sie möglichst viele Konzepte aus diesem und den vorangegangenen Materialpaketen angemessener Art und Weise:

- benutzerdefinierte Datentypen
- Operatorenüberladung
- Subtyping Polymorphism
- Stack- und Heapobjekte
- `std::`-Container, `std::`-Algorithmen
- Separation of Concerns
- Encapsulation, Separation of Interface and Implementation
- Patterns und Idiome

[11] Big-Mac-Preis in inländischer Währung auf https://de.wikipedia.org/wiki/Big-Mac-Index

[12] The Econimist, Interactive currency-comparison tool, http://www.economist.com/content/big-mac-index

[13] The Econimist, Interactive currency-comparison tool, http://www.economist.com/content/big-mac-index

3.2 Aufgabe: Modifikationen

Stellen Sie sicher, dass Sie jede einzelne der nachfolgenden Modifikationen innerhalb weniger Minuten (5 - 10) vor Zuschauern (Testatsituation) umsetzen können. Konkret sollen Sie im Testat in der Lage sein, das gegebene Grundgerüst um mindestens eine zufällig ausgewählte Modifikation zu erweitern. Bereiten Sie dazu auf ihrer Arbeitsumgebung ein Verzeichnis vor, welches ausschließlich das Grundgerüst enthält.

Modifikationen:

1. Simulieren Sie Abhebe- und Einzahlungsvorgänge von einigen Tausend Konsumenten (= Bürger, Personen) an Geldautomat und Schalter.
2. Fügen Sie der Simulation Wechselstuben hinzu, an denen die Konsumenten Teile ihres Bargeldes in eine andere Währung eintauschen.
3. Fügen Sie der Simulation eine Falschgelddruckerei hinzu.
4. Fügen Sie der Simulation Geschäfte des Einzelhandels hinzu.
5. Ermitteln Sie gelegentlich die Geldmengen M0 (Bargeld) und M1 (M0 + Sichteinlagen)

3.3 Verständnisfragen

Nach Bearbeitung des Kapitels “Konzepte”, der Erstellung des Grundgerüsts sowie dem Üben der Modifikationen sollten Sie in der Lage sein, die folgenden Fragen zu beantworten.

1. Was sind die wesentlichen Unterschiede zwischen Idiomen und Entwurfsmustern?
2. Warum sollte bei Vorhandensein einer virtuellen Methode dafür gesorgt werden, dass der Destruktor virtuell ist?
3. Welche C++-Eigenschaft macht sich das Idiom RAII zunutze?
4. Welche Probleme löst das Idiom `virtual constructor`?
5. Wie kann *Divide and Conquer* bei einer Funktion umgesetzt werden?
6. Was ist der Unterschied zwischen Köhäsion und Kopplung?
7. Wie äußert sich *Programming against Interfaces not Implementations* in C++?
8. Was ist das besondere bei der Verwendung einer Factory?
9. Welche Unterschiede bestehen zwischen Factory und virtual constructor?

4 Nützliche Links

- [MIDIO] More C++ Idioms, https://en.wikibooks.org/wiki/More_C%2B%2B_Idioms

5 Literatur

- [DPTRN] Gamma et al.: Design Patterns – Elements of Reusable Object-Oriented Software
- [PPP] Stroustrup, Bjarne: Programming - Principles and Practice using C++
- [TCPL] Stroustrup, Bjarne: The C++ Programming Language, Fourth Edition
- [SMPTRN] Design Patterns Explained Simply, https://sourcemaking.com/design_patterns/