

PÜ2

Projektübungsblatt: Gemeinsames zu PÜ21 und PÜ22

Es gibt keine einzige Ingenieursdisziplin, in der nicht die bedeutenden Werke früherer Meister genau unter die Lupe genommen werden. Auch alle angehenden Autoren müssen zunächst viel lesen. Nur die Schreiber von Programmen meinen, dass sie genial sind und nicht den Code von anderen analysieren müssen.

Grady Booch¹

PÜ2.1

Gemeinsame Lernziele und Anforderungen

Lernziele

Zu den Lernzielen gehört, dass Sie

- exemplarisch objektorientierte Techniken der Zerlegung und Abstraktion sowie das objektorientierte Entwurfsmuster Beobachter (*design pattern observer*) kennen und anwenden lernen (siehe Gamma u.a.: *Entwurfsmuster*),
- beim Anwenden dieser Techniken planvolles, zielgerichtetes Vorgehen üben.

Vorgehensweise

Gehen Sie inkrementell vor!

Aspekte trennen

- (1) Ein-/Ausgabe und funktionale Aspekte so voneinander getrennt sind, dass Ein-/Ausgabe-Teile austauschbar sind (*Leitlinie MB.1* in *MB Merkblatt*), und

Änderungen erleichtern

- (2) das Programm leicht änderbar und erweiterbar, d.h. leicht an zusätzliche Anforderungen anpassbar ist!

Softwarequalität

Dokumentieren Sie die implementierte Funktionalität! Undokumentierte Programme sind wertlos. Selbstverständlich gelten die *MB.6 Minimalanforderungen für das erste Gespräch eines Projekts* und die *MB.7 Minimalanforderungen für das letzte Gespräch eines Projekts* in *MB Merkblatt* auch hier.

PÜ2.2

Ein- und Ausgabe

Kein Verbrechen, aber Unsinn sind interaktive Anwendungen mit Eingabe von der Tastatur. Deshalb sei *Leitlinie MB.3* ausdrücklich wiederholt:

Leitlinie PÜ2.1

Einfach oder aktuell
statt umständlich und
veraltet

Die E/A-Teile der Projektprogramme verwenden Argumentübergabe von der Kommandozeile oder GUIs, aber keine Eingabe von der Tastatur.

Von jedem Bestandteil des Ampelsystems oder Fahrstuhls wird stets der aktuelle Zustand ausgegeben. (Damit der Ablauf nachvollziehbar ist, sind ggf. in die Simulation Verzögerungen einzubauen.)

Pflicht

- (1) **Text:** Die Ausgabe erfolgt textuell.

Kür

- (2) **Grafik:** Die Ausgabe erfolgt grafisch.
- (3) **GUI:** Die Ein-/Ausgabe erfolgt mit einem GUI-Dialog.

¹ Zitiert nach heise online > News > 2010 > KW 23 > 11.06.2010, <http://www.heise.de/newsticker/meldung/UML-Erfinder-Grady-Booch-zeichnet-duesteres-Bild-der-Softwareentwicklung-1020658.html>.

Nach PÜ2.1(1) sind Funktion und Ausgabe getrennt. Deshalb sei *Leitlinie MB.2* ausdrücklich wiederholt:

Leitlinie PÜ2.2

Keine Ausgabe in Funktionsteil

Schmiere nie Ausgabeanweisungen zwischen funktionale Anweisungsteile! Entwerfe Funktionsteile unabhängig von E/A-Teilen, auch GUI-Teilen!

Der erste Schritt zur Trennung ist, dass Ausgabeanweisungen nur an Enden zustandsändernder Prozeduren stehen. Der zweite Schritt ist, Ausgabeanweisungen prozedural zu trennen. Der dritte Schritt ist, Ausgabeanweisungen modular zu trennen. Der vierte Schritt ist, für die Ausgabe das Beobachter-Entwurfsmuster einzusetzen.

Zum Beobachtermuster liegen drei *Eiffel*-Varianten vor: eine einfache (*simple*) nach Gamma u.a., eine weiterentwickelte (*advanced*), und eine generische (*generic*). Jede Variante besteht aus Rollen- und Beispielklassen. Der Ampelsystem-Rohbau benutzt die dritte Ausbaustufe; diese ist auch für den Fahrstuhl empfohlen.

Aufgabe PÜ2.1

Beobachtermuster studieren

Lesen Sie in der Literatur über Zweck, Motivation, Struktur, Rollen, Interaktionen, Implementationen und Beispiele des Beobachter-Entwurfsmusters und studieren Sie die *Eiffel*-Klassen unter

http://I3/Eiffel/EiffelStudio/observer_pattern_1_simple
http://I3/Eiffel/EiffelStudio/observer_pattern_2_advanced
http://I3/Eiffel/EiffelStudio/observer_pattern_3_generic

**Leitlinie PÜ2.3**

Keine notify-Aufrufe mitten in Funktionsteil schreiben

Schreibe Aufrufe von *notify* nie mitten in funktionale Anweisungsteile, sondern immer nur ans Ende zustandsändernder Prozeduren!

notify-Aufrufe zwischen Zustandsänderungen würden Leitlinie PÜ2.2 zuwiderlaufen.

Leitlinie PÜ2.4

Beobachtbares von Beobachter entkoppeln

Lass nie beobachtete Subjekte ihre Beobachter-Objekte selbst erzeugen!

Zweck des Beobachtermusters ist die Trennung von Funktion und Ein-/Ausgabe: Beobachtete Subjekte (Funktion) sollen unabhängig von Beobachtern (Ausgabe) sein. Beobachter sollen austauschbar sein. Subjekte sind nur indirekt über *notify*-Aufrufe mit Beobachtern gekoppelt.

Wie lässt sich ein zusammengesetztes Subjekt (Ampel, System, Fahrstuhl) beobachten? Zwei Konzepte bieten sich an:

Einer für alle

- Nur an das Ganze wird ein Beobachter gehängt, der das Ganze und dessen Teile beobachtet.
 - ☺ Leicht zu konfigurieren.
 - ⊗ Der Beobachter muss die Teile seines Subjekts kennen und Zugriffsrechte darauf haben. Das kann zu Sicherheitslücken führen: Ein böser Beobachter könnte die Rechte für Zustandsänderungen am Subjekt missbrauchen. Veröffentlichte Teile könnten unerwünschterweise auch für andere Programnteile zugreifbar werden. Damit kann das Ganze seine Invariante über seine Teile nicht mehr garantieren. Die Korrektheit des Ganzen ist gefährdet.
 - ⊗ Das Gesetz von Demeter¹ ist nicht erfüllt. Ein Zugriff eines Systembeobachters könnte z.B. diese Form annehmen:


```
observed_subject.eastwest_trafficlight.red_lamp.state
```
 - ⊗ Beobachterklassen werden mit ihren Subjektklassen komplexer.
- An jedes Teil wird ein eigener Beobachter gehängt, der nur dieses Teil beobachtet.

Jedem den Seinen

¹ Das Gesetz von Demeter (*law of Demeter*) erlaubt Zugriffe der Form `object.service` und verbietet Zugriffe auf Komponenten der Form `object.part_1.part_2....part_n.service`.

- ☺ Jeder Beobachter kennt nur sein Subjekt, aber nicht dessen Teile. Wenn die Teile geschützt sind, entstehen keine Sicherheitslücken. Das Ganze kann seine Korrektheit garantieren.
- ☺ Das Gesetz von Demeter ist erfüllt. Alle Zugriffe von Beobachtern haben diese Form:
`observed_subject.state`
- ☺ Alle Beobachterklassen sind gleich einfach. Bei geschickter Lösung genügt sogar eine einzige Beobachterklasse für alle Subjektarten (*DEVICE_OBSERVER*).
- ☺ Schwer zu konfigurieren: Wie kommen Teilebeobachter an ihre Teile?
 - ◆ Indem ein Ganzes-Beobachter *start* und *stop* von *OBSERVER* redefiniert und dort die Teilebeobachter erzeugt und anhängt bzw. abhängt. Der Ganzes-Beobachter muss dann doch die Teile seines Subjekts kennen.
 - ◆ Indem der übergeordnete Beobachtungsmeister das Entwurfsmuster Abstrakte Fabrik anwendet. Das ist lehrreich, aber etwas aufwändig.

Fazit: „Einer für alle“ ist leichter zu implementieren, aber unsicherer, „Jedem den Seinen“ ist sicherer, aber aufwändiger zu implementieren.

PÜ2.3

Problem der Konsistenz zusammengesetzter Objekte

In beiden Projekten kommen Objekte vor, die sich aus anderen Objekten zusammensetzen. Ein Problem dabei verdeutlichen diese Fragmente in Pseudosprache:

```

class PART
  public
    queries
      state : STATE
    actions
      change_state
end

class COMPOUND
  public
    queries
      part : PART
    invariants
      condition_on (part)
    actions
      ...
      part.change_state -- ok, controlled by COMPOUND
      ...
end

class CLIENT_OF_COMPOUND
  ...
  compound : COMPOUND
  ...
  my_state := compound.part.state -- ok, read access cannot break invariant
  ...
  compound.part.change_state -- dangerous, may break invariant of compound!
  ...
  part : PART
  ...
  part.change_state -- ok, not part of compound
end
  
```

Ein zusammengesetztes Objekt kann invariante Bedingungen an die Konsistenz seiner Teilobjekte stellen. Das Zusammengesetzte muss seine Invarianten garantieren, gleichzeitig will es Kunden sichere Zugriffe auf seine Teile erlauben. Kunden sollen an Teilen Abfragen aufrufen dürfen, aber keine Aktionen. Leider ist die Ausdruckskraft

bekannter objektorientierter Sprachen zu beschränkt, um solche Situationen befriedigend zu formulieren. Sprachgrenzen erschweren es, die Konsistenz von Objekten zu garantieren. Im ersten Lösungsversuch schützt das Zusammengesetzte seine Teile:

Geschütztes Teil

```
class COMPOUND
  protected
    queries
      part : PART
    ...
end
```

So verbietet es zwar Aktionsaufrufe an Teilen, aber auch harmlose Abfragenaufrufe:

```
class CLIENT_OF_COMPOUND
  ...
  compound : COMPOUND
  ...
  my_state := compound.part.state -- forbidden, harmless read access
  ...
  compound.part.change_state -- forbidden, dangerous write access
end
```

Dieser Schutz ist also zu stark. Im zweiten Lösungsversuch exportiert das Teil seine Aktionen selektiv:

Selektiver Export

```
class PART
  public
    queries
      state : STATE
    public {COMPOUND}
    actions
      change_state
end
```

Damit ist

part.change_state

in COMPOUND erlaubt, aber

compound.part.change_state

in CLIENT_OF_COMPOUND verboten. Insofern ist das Problem vorerst gelöst. Kommt aber eine weitere Klasse der Art

```
class COMPOUND_OTHER
  -- similar to COMPOUND
end
```

hinzu, dann muss für korrektes Funktionieren die Kundenliste von change_state in PART erweitert werden:

```
class PART
  ...
  public {COMPOUND, COMPOUND_OTHER}
  actions
    change_state
end
```

Das ist nicht immer möglich. Falls es möglich ist, schränkt es die Wiederverwendbarkeit von PART ein. Es kann zu langen Kundenlisten führen. Im dritten Lösungsversuch exportiert PART change_state an die leere Klasse CLIENT_OF_PART:

Export an Abstraktion

```
class PART
  ...
  public {CLIENT_OF_PART}
  actions
    change_state
end
```

```

class COMPOUND extends CLIENT_OF_PART
...
    part.change_state -- ok, in a CLIENT_OF_PART
...
end
class CLIENT_OF_COMPOUND
...
    compound.part.change_state -- forbidden, not in a CLIENT_OF_PART
...
    part : PART
...
    part.change_state -- forbidden, not in a CLIENT_OF_PART
end

```

Die Lösung funktioniert partiell, ist aber schwergewichtig durch die zusätzliche Klasse CLIENT_OF_PART. Sie versagt, wenn CLIENT_OF_COMPOUND aus gutem Grund auch CLIENT_OF_PART sein muss. Sie ist leicht zu unterlaufen durch „schlechte“ Klassen:

```

class BAD extends CLIENT_OF_PART

```

Im vierten Lösungsansatz entscheidet nicht das Teil, sondern das Zusammengesetzte, welchen Kunden welche Zugriffe erlaubt sind, d.h. das Zusammengesetzte schränkt die *Art des Zugriffs* ein:

Zugriffsart Lesen

```

class COMPOUND
    public readonly
    queries
    part : PART
...
end

```

Damit ist

```

compound.part.change_state

```

in CLIENT_OF_COMPOUND verboten, weil es kein Lesezugriff ist, aber

```

my_state := compound.part.state

```

erlaubt, weil es ein Lesezugriff ist. Leider können *Component Pascal*, *Eiffel*, *Java*, *C#* das nicht ausdrücken, nur *C++* kann es mit **const**. Da COMPOUND seine Invarianten garantieren muss, muss der Programmierer den erforderlichen Schutz mit anderen Mitteln formulieren oder Sicherheitslücken offen lassen.