Technische Hochschule Rosenheim

Modul - Objektorientierte Programmierung

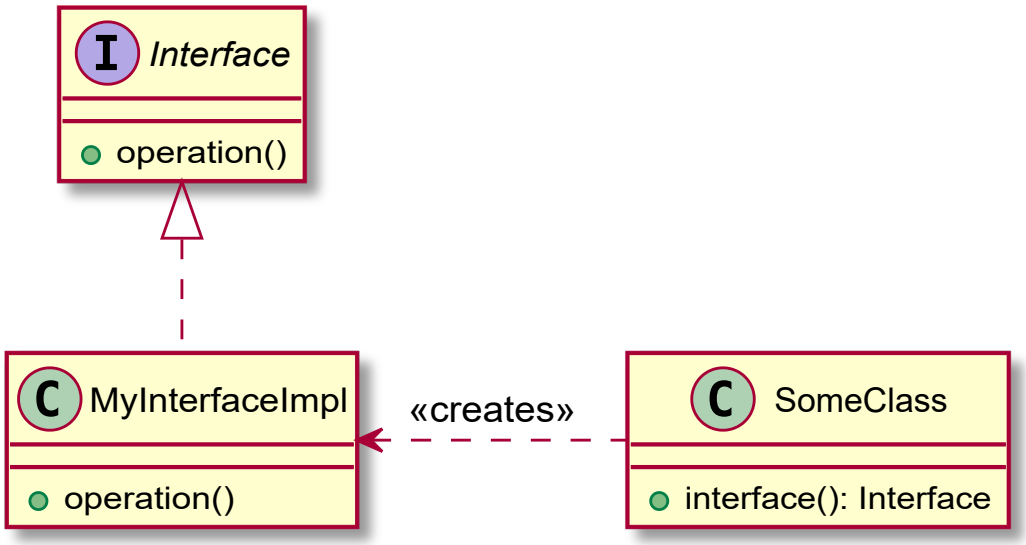
Bachelor Wirtschaftsinformatik

05 - Factory, Iterator und Klassen

Prof. Dr. Marcel Tilly

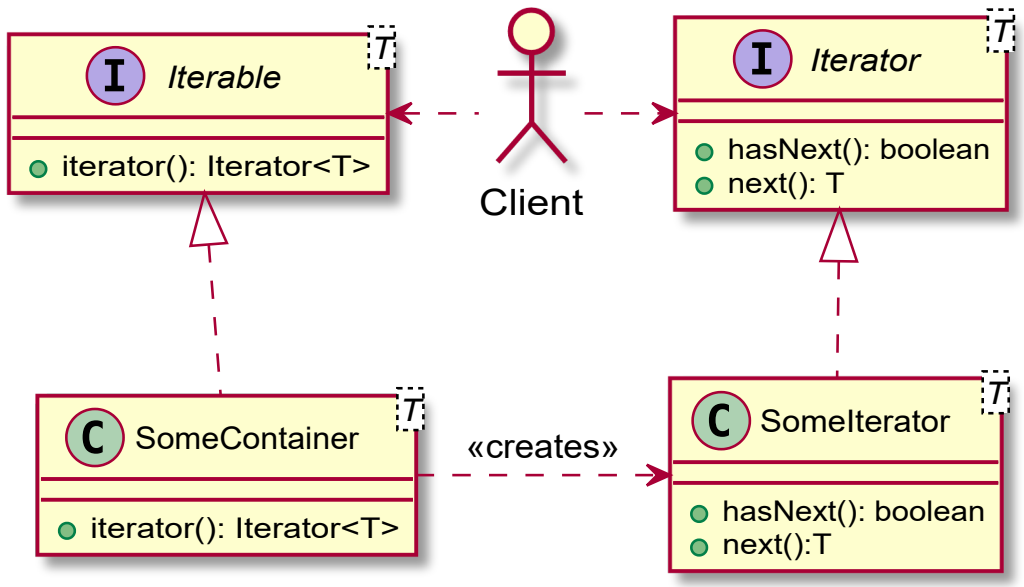
Fakultät für Informatik, Cloud Computing

Factory Method



- Methode gibt Instanz zu Interface zurück
- Aufrufer der Methode kennt nur das *Interface* als Rückgabetyp bekannt ist, nicht aber die der Instanz zu Grunde liegende *implementierende Klasse*.
- Fabrikmethode ist ein Erstellungsmuster (*creational pattern*).

Iterator



- Modellierung des sequenziellen Zugriffs auf eine Containerstruktur
- Setzt `Iterable` und `Iterator` in Beziehung
- Verhaltensmuster (*behavioral pattern*)
- Verwendet dabei das Factory Method Pattern.

Verwendung

Technische Hochschule Rosenheim

Regulär, mit `while` Schleife

```
Iterator<Integer> it = intset.iterator();
while (it.hasNext())
    System.out.println(it.next());
```

Mit `for-each` Schleife (erfordert `implements Iterable<T>`):

```
for (int i : intset)
    System.out.println(i);
```

Syntax: Klassen

Technische Hochschule Rosenheim

In Java sind folgende Klassen möglich:

- (normale) Klasse
- *innere* Klasse
- *statische innere* Klasse
- *anonyme innere* Klasse

(Normale) Klassen

Technische Hochschule Rosenheim

```
class MeineKlasse {

}
```

- **Eine** normale, nicht-statische Klasse pro `.java` Datei
- Klassenname muss gleich dem Dateinamen sein; Übersetzung von `MeineKlasse.java` in `MeineKlasse.class` (Bytecode).
- Kann beliebig viele innere Klassen enthalten

Innere Klassen

Technische Hochschule Rosenheim

```
class MeineKlasse {
    private MeineInnere {

    }

    private static MeineStatischeInnere {

    }
}
```

- Beliebig viele innere Klassen
- Sichtbarkeit und Gültigkeit analog zu Attributen
- **Innerhalb** einer normalen Klasse, **ausserhalb** von Methoden
- `Innere` kann **nur** in Instanz von `Aeussere` existieren
- Im Prinzip beliebig schachtelbar (innere in inneren in inneren, ...)

(Nicht-statische) Innere Klasse

```
class Aeussere {
    private String attribut = "A";
    private static String ATTRIBUT = "B";

    class Innere {
        String attribut = "X";

        // Compilerfehler! Innere können keine static haben
        static String ATTRIBUT = "Y";

        void zugriff() {
            System.out.println(this.attribut);  // "X"
            System.out.println(Aeussere.this.attribut);  // "A"
        }
    }
}
```

(Nicht-statische) Innere Klasse

- Zugriff auf alle Attribute der `Aeussere` Instanz
- Bei Namenskonflikten mit `<KlassenName>.this.*` disambiguieren
- keine `static` Attribute in inneren Klassen

Statische Innere Klasse

```
class Aeussere {
    Strint attribut = "A";
    static class Statische {
        String attribut = "Z";
        void zugriff() {
            System.out.println(attribut);  // "Z"
            // Compilerfehler! Statische Innere hat keinen Aeussere-Scope
            System.out.println(Aeussere.this.attribut);
        }
    }
}
```

- Können ohne Instanz der äußeren Klasse verwendet werden

```
Aeussere.Statische inst = new Aeussere.Statische();
```

- Folglich kein direkter Zugriff auf die Attribute der äußeren Klasse

Anonyme Innere Klasse

```
interface Intf {
    void methode();
}
```

```
final int aussen = 10;
Intf inst = new Intf() {
    int attr;
    {
        // Konstruktor, wenn nötig
        attr = aussen;  // Zugriff nur auf quasi-final!
    }
    public void methode() {
        System.out.println(attr);
    }
};
```

Anonyme Innere Klasse

- Erstellt ein Objekt das ein Interface implementiert
- Klassendefinition aber zur Laufzeit unbekannt ("anonym")
- Zugriff auf äußere Attribute nur, wenn diese quasi-final sind.

Zusammenfassung

Technische Hochschule Rosenheim

- Die **Iteration** für Containerstrukturen (wie z.B. Listen oder Sets) ist eine Abstraktion, welche dem Benutzer sequenziellen Zugriff auf die enthaltenen Elemente gibt, ohne die innere Struktur zu kennen.
- Der **Iterator** als **Verhaltensmuster** (behavioral pattern) beschreibt dabei den Zusammenhang der Interfaces `Iterator<T>` und `Iterable<T>`.
- Die **Fabrikmethode** (factory method) ist ein **Erstellungsmuster** (creation pattern) welches sich auch im Iterator Muster wiederfindet.
- **Iteratoren für sequenzielle Datenstrukturen** sind im Allgemeinen einfach zu implementieren: sie erinnern die aktuelle Position.
- **Iteratoren für Baumstrukturen**, also Datenstrukturen deren Elemente mehr als einen Nachfolger haben, verwenden hingegen eine **Agenda:** eine Liste von noch zu besuchenden Elementen.
- Es gibt *normale*, *innere*, *statische innere* und *anonyme innere* Klassen

Fragen?