Hausaufgabe 9

Tim Wende

December 4, 2021

Builder Pattern

In der Vorlesung wurde das Builder Pattern **nicht** vorgestellt.

a. Erklären Sie mit eigenen Worten, welchen Nutzen das Builder Pattern bringt und wo die typischen Einsatzgebiete liegen.

Nutzen: Mit einem Builder kann man ein Objekt kontrolliert erstellen und nach jeder Operation die Qualität des theoretische Endresultats reevaluieren. So hat man jederzeit Kontrolle darüber, ob der jeweilige Anfragende das Objekt übergeben bekommt, oder ob der Builder als einziges eine Referenz erhalten darf. So muss ein Nutzender mit der API des Builders arbeiten, bis alle nicht optionalen Operationen durchgeführt wurden.

typisches Einsatzgebiet: Builder sind nah verwandt mit Factories, wobei diese ein bisschen primitiver arbeiten. Sollte man im ctor verschiedene optionale Parameter[1] haben, entsteht hier ein sogenanntes **Telescoping Constuctor Pattern**. Dies sieht beispielsweise wie folgt aus (Spoilerwarnung: Aufgabenteil b.):

```
public Pokemon(String description){ ... }
public Pokemon(String description, Type type){ ... }
public Pokemon(String description, Type type, int hp){ ... }
public Pokemon(String description, Type type, int hp, int atk){ ... }

Pokemon p = new Pokemon("Glumanda", new Type("Feuer"), 39, 52);
```

Hier sieht man, dass dies leicht unübersichtlich wird und man sich die Reihenfolge[2] schwer merken kann. ist 39 `hp`? `atk`? Brauchen wir das? Was ist, wenn man nur `atk` und keine `hp` angeben möchte? Was ist, wenn `hp` von `atk` abhängig ist? Also zusammengefasst: das **Telescoping Constuctor Pattern** ist zu vermeiden! Aber wie?

Die eben erwähnte Factory kann hier helfen, ist aber ebenfalls nicht wirklich schön:

```
PokemonFactory pf = new PokemonFactory();
Pokemon p =  pf.newPokemon("Glumanda");
p = pf.addType(p, new Type("Feuer"))
p = pf.addHP(p, 39)
p = pf.addATK(p, 52)
```

Man sieht, dass `pf` über die gesamte Laufzeit immer wieder neu referenziert werden muss. Ebenso lässt sich der gesamte Vorgang nicht wirklich gut pipen, da wir ja auf dem Objekt selber arbeiten, und dieses die dort zu sehenden Methoden nicht kennt.

Aber wie machen wir das jetzt gut?[3] In Java? Gar nicht.

Aber der Builder kommt einem guten Endresultat doch schon ganz nah:

```
Pokemon p = new PokemonBuilder("Glumanda").with(new Type("Feuer")).withHP(39).withATK(52).build();
```

Da jede Operation in diesem Prozess den Builder an sich zurückgibt, kann man hier das Pokemon „durchreichen“. Da man jedoch nie das Pokemon an sich in der pipe hat, sondern nur den Builder, ist es nicht möglich ein halb fertiges Objekt zu erhalten. Bei großen Sinnvollen implementierungen findet man in `.build()` vermehrt `.clone()` sowie eine Überprüfung auf Integrität und Qualität des Objekts an sich. Zusätzlich hat jedes `Pokemon` einen eigenenen, für sich selbst einzigartigen und zuständigen `PokemonBuilder`.

Also zusammengefasst:

→ Objekt mit optionalen Parametern: Telescoping Constuctor Pattern

→ Zu viele Attribute: Factory

→ Zu kompliziert: Builder

Also wenn man Objekte mit beliebig vielen, gleichartigen und optionalen Parametern im Konstruktor hat, welche kompliziert zu erstellen sind oder man die Integrität unter allen Umständen beibehalten möchte und die absolute Korrektheit des Objekts unentbehrlich wichtig ist, ist das Nutzen des Builder Patterns sinnvoll[4].

[1] und man in der Sprache seiner Wahl keine default Werte hat. z.B. in Java
[2] und man in der Sprache seiner Wahl keine benannten Parameter hat. z.B. in Java
[3] Indem wir eine sinnvolle und gut Programmiersprache benutzen. z.B. nicht Java
[4] wenn man in einer veralteten schlechten Programmiersprache schreibt. z.B. Java

b. Finden Sie ein geeignetes Beispiel und erklären Sie dieses. Modellieren (Klassendiagramm) und **implementieren** Sie für diesen Anwendungsfall das Builder Pattern.

Vorab: Eine gekürzte Version zur Überschaulichen Darstellung:

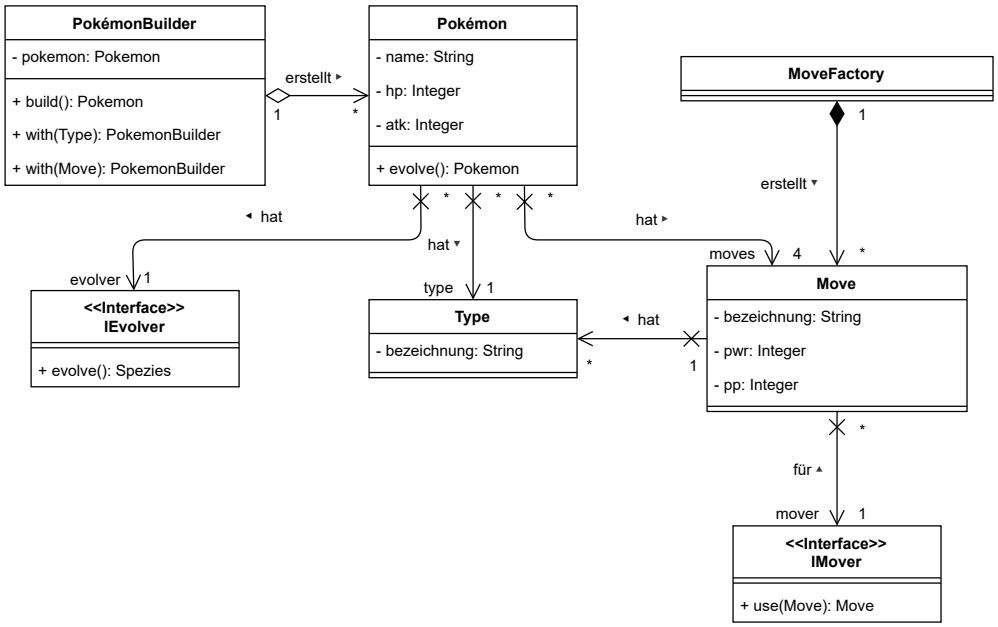

Figure 1: `class_diagram_simple`

Dieses Diagramm enthält einige Software Pattern:

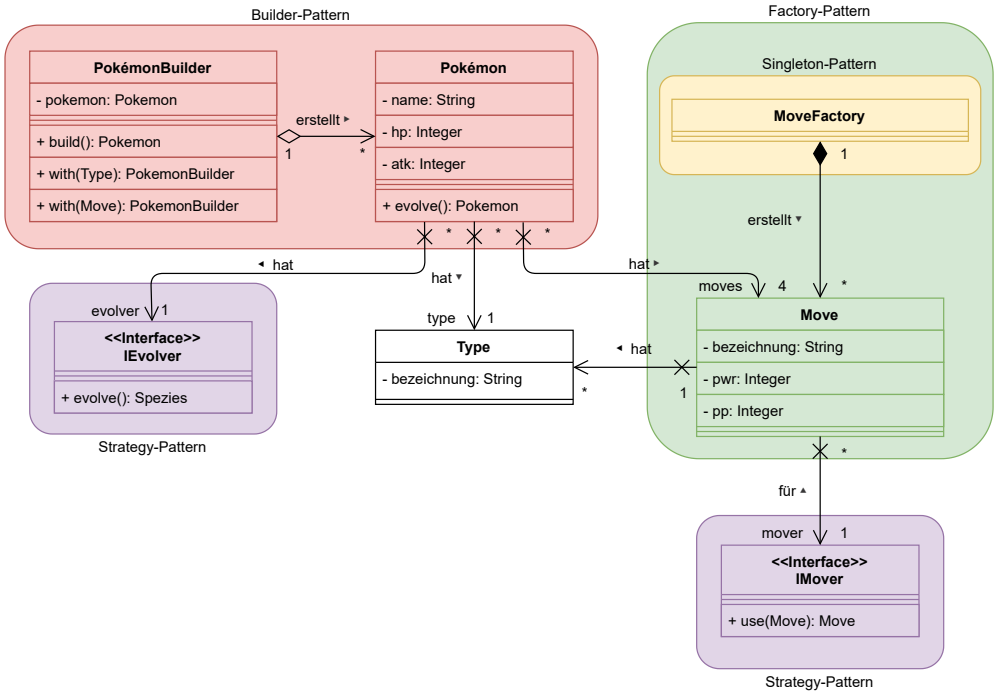

Figure 2: `class_diagram_pattern`

Diese werden hoffentlich durch den Code verständlich dargestellt.

Nun das Diagramm in voller Schönheit:

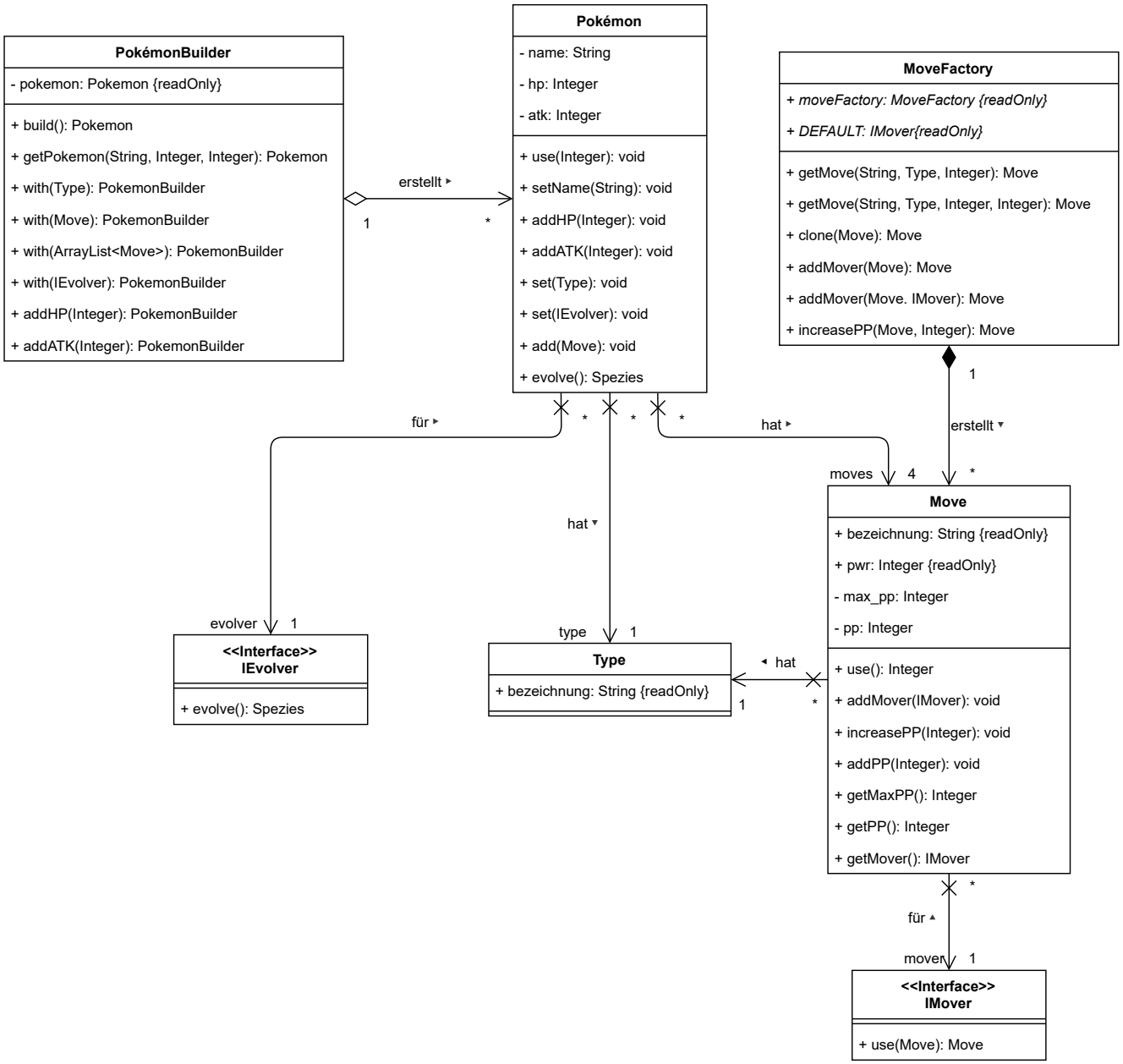

Figure 3: `class_diagram`

Schauen wir uns nun die Main an:

```
/**
 * @author Tim Wende
 */
public class Test {

	public static void main(String[] args) {
		MoveFactory moveFactory = MoveFactory.moveFactory;

		Type normal = new Type("Normal");
		Type fire = new Type("Feuer");

		Move tackle = moveFactory.addMover(moveFactory.increasePP(moveFactory.getMove(
			"Tackle", normal, 40), 35));
		Move kratzer = moveFactory.addMover(moveFactory.increasePP(moveFactory.getMove(
			"Kratzer", normal, 40), 35));
		Move glut = moveFactory.addMover(moveFactory.increasePP(moveFactory.getMove(
			"Glut", fire, 40), 25));

		Move dragon_zurstoerer_of_hellfire = moveFactory.addMover(moveFactory.getMove(
			"Platscher", normal, 0, 40), move -> {
			if (Math.random() > 0.9)
				return Integer.MAX_VALUE;
			return MoveFactory.DEFAULT.use(move);
		});

		Pokemon glumanda = new PokemonBuilder("Glumanda", 39, 52)
			.with(fire)
			.with(kratzer)
			.with(glut)
			.with(pokemon -> {
				String name = "Glutexo";
				System.out.println(pokemon + " entwickelt sich zu " + name);
				return new PokemonBuilder(name, pokemon).addHP(10).addATK(25).build(
			})
			.build();
		Pokemon bisasam = new PokemonBuilder("Bisasam").with(new Type("Pflanze"))
			.with(tackle).build();
		Pokemon schiggy = new PokemonBuilder("Schiggy").with(new Type("Wasser"))
			.with(moveFactory.clone(tackle)).build();

		System.out.println();
		for (int i = 0; i < 4; i++)
			glumanda.use(i);
		for (int i = 0; i < 4; i++)
			bisasam.use(i);
		for (int i = 0; i < 4; i++)
			schiggy.use(i);

		System.out.println();
		glumanda = glumanda.evolve();
		glumanda.add(dragon_zurstoerer_of_hellfire);

		System.out.println();
		for (int i = 0; i < 4; i++)
			glumanda.use(i);
		glumanda.use(2);
		glumanda.use(2);
	}
}
```

Diesen Code gehen wir nun einmal durch:

```
MoveFactory moveFactory = MoveFactory.moveFactory;
```

Da wir nach dem Singleton Pattern handeln, wird hier eine Referenz auf das public Attribut in MoveFactory erstellt. Da dieses final ist, darf man gerne darauf zugreifen.

```
Type fire = new Type("Feuer");
```

Type ist eine normale Klasse, hier ist nichts besonderes bei.

```
Move glut = moveFactory.addMover(
    moveFactory.increasePP(
        moveFactory.getMove(
            "Glut",
            fire,
            40
        ),
        25
    )
);
```

Ich habe es mal versucht das Factory-Pattern schön darzustellen. Man nutzt, wie oben beschrieben, für jede Operation seine Factory:

- Zuerst erstellt man einen leeren Move:
  `moveFactory.getMove("Glut", fire, 40);`
- Diesen übergibt man einen Betrag an AP:
  `moveFactory.increasePP([...], 25);`
- Und lässt schlussendlich automatisch einen IMover hinzufügen:
  `moveFactory.addMover([...]);`

Im letzten Schritt kann man ebenfalls einen eigenen IMover erstllen:

- z.B. wird hier wird (auf Grund eines Statuseffekts) jeder Schaden verdoppelt
  `moveFactory.addMover([...], move -> MoveFactory.DEFAULT.use(move) * 2);`

Nun erstellen wir ein Pokemon:

```
Pokemon glumanda = new PokemonBuilder("Glumanda", 39, 52)
    .with(fire)
    .with(kratzer)
    .with(glut)
    .with(pokemon -> {
        String name = "Glutexo";
        System.out.println(pokemon + " entwickelt sich zu " + name);
        return new PokemonBuilder(name, pokemon).addHP(10).addATK(25).build();
    })
    .build();
```

Bis zu `.build()` bearbeiten wir einen PokemonBuilder. Im Konstruktor geben wir alle nicht optionalen Parameter mit. Dann fügen wir ein Paar Attacken hinzu. Und zu guter Letzt erstellen wir ein Schema zum entwickeln. Dieses gibt einen Text aus und erstellt dank copy ctor ein neues Pokemon mit leicht erhöten Werten (ebenfalls mithilfe des PokemonBuilders).

Unter Bezugnahme Ihres Kommentars zu Hausaufgabe 7 fallen alle weiteren Klassen ab hier weg.

Jedoch hier einige Kommentare

- Das Builder-Pattern wurde in der Aggregation zwischen PokemonBuilder und Pokemon der Aufgabenstellung entsprechend realisiert
- Alle final Attribute sind public. Dies öffnet natürlich Tür und Tor für jegliche Fehler, jedoch wollte ich dies nicht über-Engineeren
- Generell sind die Klassen offensichtlich Pokémon™ nachempfunden, jedoch stark vereinfacht. Auf fachliche Korrektheit sei bitte nicht zu achten.
- PokemonBuilder bearbeitet bis zu build ein private Pokemon. Natürlich könnte man die Attribute von Pokemon ebenfalls im Builder nutzen, und so erst beim build() ein Pokemon erstellen. Dies wäre sinnvoll, wenn der ctor aus Pokemon beispielsweise dieses einer public Liste hinzufügen sollte. Da dies nicht der Fall ist, habe ich dies vernachlässigt. Alternativ könnte man die Attribute über eine abstract Klasse verteilen, von der Beide Klasse erben. Da `Java` jedoch nur von einer Klasse erben lässt, wollte ich diesen Platz nicht für einen solchen Unfug verschwenden. Zusätzlich brauchte ich keine Überprüfung der qualitativen Korrektheit in build(), da alle verpflichtenden Attribute bereits im ctor des Builders gesetzt wurden.
- MoveFactory wirft im ctor einen Fehler, sollte dieser von außerhalb aufgerufen werden. Explizit dieser sowie eventuell nicht nutzbare Konstruktoren sowie nicht zu erreichende Klassen sind public. Von einer Lösung, welche eventuell durch Sichtbereiche abgesichterte Objekte, welche durch packages realisiert werden könnte oder sonstige Abhilfe bei diesem Problem habe ich abgesehen. Durch hohe Kohäsion sind meine Klassen sowieso in einem einzigen package, da wäre dies auch schwierig umzusetzen. Hilfreiche Methoden wie `friend` aus C++ zu imitieren war mir hier zu mühselig, jedoch durch `signature security` möglich.

Der Vollständigkeit halber, hier mein Output, welcher von der Test.main() erstellt wurde:

```
Glumanda wurde gefangen
Glumanda erlernt Kratzer
Glumanda erlernt Glut

Bisasam wurde gefangen
Bisasam erlernt Tackle

Schiggy wurde gefangen
Schiggy erlernt Tackle

Glumanda (Feuer) setzt Kratzer (Normal, 34/35) ein (-20HP)
Glumanda (Feuer) setzt Glut (Feuer, 24/25) ein (-20HP)
Bisasam (Pflanze) setzt Tackle (Normal, 34/35) ein (-4HP)
Schiggy (Wasser) setzt Tackle (Normal, 34/35) ein (-4HP)

Glumanda entwickelt sich zu Glutexo

Glutexo (Feuer) setzt Kratzer (Normal, 33/35) ein (-30HP)
Glutexo (Feuer) setzt Glut (Feuer, 23/25) ein (-30HP)
Glutexo (Feuer) setzt Platscher (Normal, 39/40) ein (-0HP)
Glutexo (Feuer) setzt Platscher (Normal, 38/40) ein (-0HP)
Glutexo (Feuer) setzt Platscher (Normal, 37/40) ein (-21474835HP)
```

Sequenzdiagramm: Wetterstation

In der Vorlesung haben Sie das Observer Pattern (zu deutsch Beobachter Entwurfsmuster) am Beispiel einer Wetterstation kennengelernt. Als Beispiel für konkrete Implementierungen der Interfaces Subject und Observer werden die Klassen *WeatherData* bzw. *CurrentConditionsDisplay* aus dem Foliensatz verwendet.

Modellieren Sie in ~~Visual Paradigm~~ draw.io ein Sequenzdiagramm, das diesen Sachverhalt darstellt. Das Diagramm soll die Interaktion der Klassen für den folgenden Programmablauf modellieren:

1. Eine Wetterstation startet und erstellt jeweils eine neue Instanz der Klassen *WeatherData* und *CurrentConditionsDisplay*. (Das Display bekommt die Wetterdaten im Konstruktor übergeben und registriert sich anschließend als Observer, siehe Folien)
2. Es treten zwei Veränderungen des Wetters auf. Modellieren Sie einen Pull oder einen Push-Observer (Orientieren Sie sich an den Folien für einen Push-Observer)
3. Der Observer wird vom Subject entfernt.
4. Die Wetterstation wird abgeschaltet und der Speicher wird freigegeben

Hinweise:

- Achten Sie zudem darauf, dass Methodenaufrufe innerhalb der selben Klasse richtig dargestellt werden.
- Benutzen Sie explizite Namen mit Typ für die Benennung der Teilnehmer.
- Benutzen Sie für Nachrichten die Form <Nachrichtenname> (<Parametername>=<Argumentwert>,...)

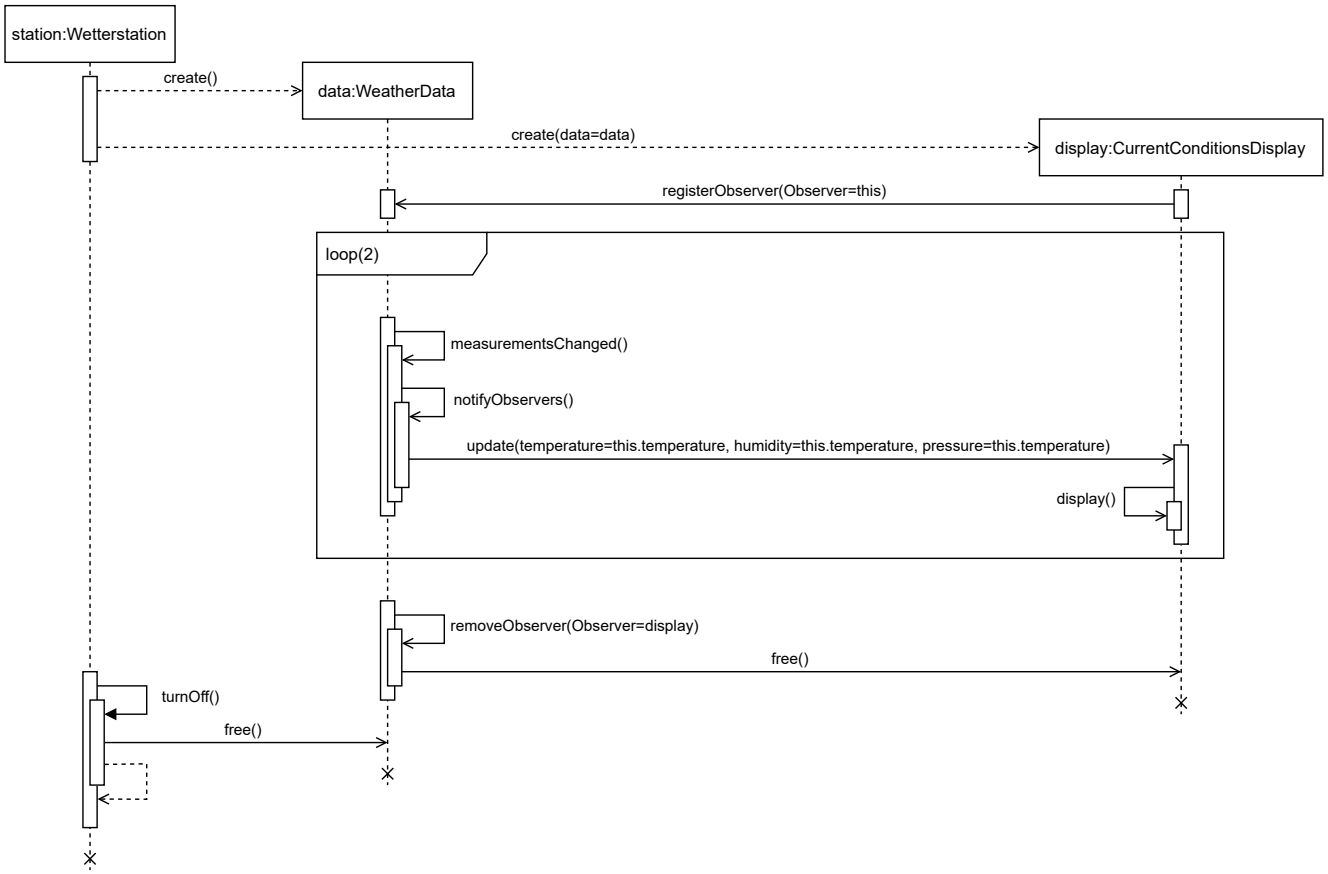

Figure 4: `sequence_diagram`