

Sitzung 2: rse_annotations – Forschungscode markieren und prüfen

Grundlagen: Decorators, Immutabilität und Dataclasses

Dr. Julian Dehne

Gesellschaft für Informatik e.V. (GI) · deRSE
julian.dehne@gi.de · ORCID 0000-0001-9265-9619

Wiederholung: Decorators

Was ist ein Decorator?

Mit Decorator

```
@calculations
def normalize(x, lo, hi):
    return (x - lo) / (hi - lo)
```

Ohne: dasselbe von Hand

```
def normalize(x, lo, hi):
    return (x - lo) / (hi - lo)

normalize = calculations(normalize)
```

- `@calculations` über `def` ist nur Kurzschrift für den Aufruf rechts.
- Ein Decorator ist ein **aufrufbares Objekt**: Es bekommt eine Funktion und gibt eine Funktion zurück.
- Was zurückkommt, steht danach unter dem alten Namen.

Mit und ohne Argumente

```
@data_input
def read_rows(path): ...

@data_input(fields={"path": "CSV file to read", "rows": "parsed records"})
def read_rows(path): ...
```

- **Ohne Klammern:** `data_input(read_rows)` bekommt die Funktion und markiert sie.
- **Mit Klammern:** `data_input(fields=...)` bekommt **keine** Funktion. Es muss einen Decorator zurückgeben, der dann auf die Funktion angewendet wird.
- **Zwei Schritte:** `read_rows = data_input(fields=...)(read_rows)`
- Die Argumente (hier `fields`) werden als Metadaten mit der Markierung gespeichert.

Im Prototyp: Die Instanz ist der Decorator

```
class Concern:
    def __call__(self, fn=None, **meta):
        def mark(f):
            marks = getattr(f, "__rse_concerns__", ())
            f.__rse_concerns__ = marks + ((self.name, meta),)
            return f # unverändert
        return mark(fn) if callable(fn) else mark
```

- `__call__` macht ein Objekt aufrufbar: `@calculations` ruft `calculations.__call__(normalize)`.
- `callable(fn)` trennt die Fälle: Funktion da → markieren, sonst `mark` zurückgeben.
- Mehrere Markierungen sammeln sich im Tupel `__rse_concerns__`.
- Ein neuer Concern erbt `__call__` und schreibt keinen eigenen Decorator.

Java-Annotation vs. Python-Decorator

	Java: <code>@Override</code> , <code>@Test</code>	Python: <code>@calculations</code>
Was ist es?	Metadaten, deklariert mit <code>@interface</code>	ein aufrufbares Objekt, beliebiger Code
Wann wirkt es?	Compiler, Annotation Processor, Reflexion	beim Ausführen von <code>def</code> (Import)
Ändert es Verhalten?	nur über ein Framework (JUnit, Spring)	ja, wenn es umhüllt

- `rse_annotations` nutzt Decorators **wie** Java-Annotationen: nur markieren, nichts ändern.
- Der Scan liest die Markierung aus dem Quelltext (`ast`), ohne den Code zu importieren oder auszuführen.
- `__rse_concerns__` dient der Reflexion zur Laufzeit, etwa in Tests.

- **GoF-Decorator** (Gamma et al. 1994): Ein Objekt umhüllt ein anderes mit derselben Schnittstelle und ergänzt Verhalten. Python-Decorators sind verwandt, aber allgemeiner: Sie dürfen auch nur markieren.
- **AOP** (Sitzung 1, *Querschnittsbelange*):
 - ▶ **Join Point**: die markierte Funktion. Die Autorin setzt ihn selbst, statt ihn per Pointcut auszuwählen.
 - ▶ **Advice**: die Prüfung durch Scan und Review. Sie läuft statisch, nicht zur Laufzeit.
 - ▶ **Concern**: das Risiko, das geprüft wird, z. B. ein ungesedeter Zufallszug.
- Dasselbe Idiom kennen Sie aus `@pytest.fixture` oder `@app.route` in Flask.

Technische Modellierung

Concern, Verdict, Project

- **Concern:** eine Art von Schritt, von der abhängt, ob man einem Ergebnis trauen kann: ein Zufallszug, ein Modellaufruf, eine Rechnung, eine menschliche Entscheidung. Jeder Concern hat eine **Behauptung** (assertion), die für jede damit markierte Funktion gelten soll.
- **Verdict:** das Urteil eines Menschen, ob die Behauptung für **eine** markierte Funktion gilt: `accepted`, `declined` oder `pending`.
- **Project:** der geprüfte Code: ein Verzeichnis, die Concerns, nach denen gesucht wird, und die Verdicts, die dazu gesammelt wurden.

	Wer	Wo
Markieren	Autor:in des Forschungscode	<code>@calculations</code> über der Funktion
Entdecken	das Werkzeug	<code>detect</code> im statischen Scan
Prüfen	Reviewer:in	<code>--inspect: <i>Does the assertion "... hold?</i></code>

Concern: der Begriff

- Ein Concern benennt ein **Risiko**, keinen Fehler. `@model_call` heißt „hier wird ein Sprachmodell aufgerufen“, nicht „hier ist etwas falsch“.
- Die Behauptung macht das Risiko prüfbar:

Concern	Behauptung (assertion)
<code>calculations</code>	This is a pure function whose formula matches the method described in the paper.
<code>model_call</code>	Model, version, prompt and parameters of this language-model call are recorded.

- Die Menge der Concerns ist offen: Ein neuer Concern ist ein neuer Ordner.

Verdict: der Begriff

- Der Scan liefert nur **Vermutungen**. Ob eine Behauptung stimmt, entscheidet ein Mensch.
- Ein Verdict gehört zu genau einem Paar (Funktion, Concern). Eine Funktion mit zwei Markierungen bekommt zwei Verdicts.
- `accepted`: Behauptung gilt. `declined`: Behauptung gilt nicht. `pending`: noch offen (übersprungen).
- Die Verdicts liegen in `inspection.yaml` neben dem Code. Ein zweiter Durchlauf schlägt die alten Urteile als Vorgabe vor; so kann das Review über mehrere Sitzungen laufen.

Immutabilität

Was heißt unveränderlich?

- Ein **unveränderliches** (immutable) Objekt wird nach seiner Erzeugung nicht mehr geändert.
- Eine „Änderung“ erzeugt ein **neues** Objekt; das alte bleibt, wie es war.

	unveränderlich	veränderlich
Python	int, str, tuple, frozenset	list, dict, set
Java	String, record, List.of(...)	ArrayList, HashMap

```
s = "abc"
t = s.upper()      # neues Objekt; s ist weiterhin "abc"
```

frozen=True in Python

```
from dataclasses import dataclass

@dataclass(frozen=True)
class Concern:
    name: str
    help: str

c = Concern("stochastic", "Ist der Zufall geseedet?")
c.name = "random"
# dataclasses.FrozenInstanceError: cannot assign to field 'name'
```

- Jede Zuweisung an ein Feld wirft `FrozenInstanceError`.
- `frozen=True` erzeugt zusätzlich `__hash__` aus den Feldern.

Grenze: nur flach (shallow)

```
@dataclass(frozen=True)
class FunctionView:
    reads: list[str]

v = FunctionView(["open"])
v.reads = []           # FrozenInstanceError
v.reads.append("load") # erlaubt: die Liste ist veränderlich
```

- `frozen` schützt die **Felder**, nicht die Objekte, auf die sie zeigen.
- **Echte Unveränderlichkeit**: `tuple` statt `list`, `frozenset` statt `set`.
- Im Prototyp hält `FunctionView` noch Listen – geschützt nur durch die Konvention „Detektoren lesen nur“.

Werte im Konstruktor anpassen

```
@dataclass(frozen=True)
class FunctionView:
    reads: tuple[str, ...]

    def __post_init__(self):
        # self.reads = ... wäre ein FrozenInstanceError
        object.__setattr__(self, "reads", tuple(sorted(self.reads)))
```

- `__post_init__` läuft nach dem generierten `__init__`.
- Innerhalb der Klasse geht Zuweisen nur über `object.__setattr__`. Das ist eine bewusste Hintertür, nur für die Konstruktion.

Vergleich: Java

```
public record FunctionView(String name, List<String> reads) {  
    public FunctionView {                                // compact constructor  
        reads = List.copyOf(reads);                    // unveränderliche Kopie  
    }  
}
```

- Ein `record` hat nur `final`-Felder und generiert Konstruktor, Getter, `equals`, `hashCode`, `toString`.
- Auch hier flach: Ohne `List.copyOf` könnte der Aufrufer die Liste später ändern.
- Python `__post_init__` mit `object.__setattr__` entspricht dem *compact constructor*.

Dataclasses

Was `@dataclass` generiert

Von Hand

```
class Concern:
    def __init__(self, name, help):
        self.name = name
        self.help = help

    def __repr__(self):
        return f"Concern({self.name!r})"

    def __eq__(self, other):
        return (self.name, self.help) == \
            (other.name, other.help)
```

Mit Dataclass

```
from dataclasses import dataclass

@dataclass
class Concern:
    name: str
    help: str
```

- `__init__`, `__repr__`, `__eq__` aus den annotierten Feldern
- mit `frozen=True` zusätzlich `__hash__`

Felder steuern mit *field()*

```
from dataclasses import dataclass, field

@dataclass
class Record:
    qualname: str
    marked: list[str] = field(default_factory=list)
    source: str = field(default="", repr=False)
```

- **Defaults:** Felder mit Default müssen nach denen ohne stehen.
- `default_factory=list`: jede Instanz bekommt eine eigene Liste (nie = []).
- `repr=False`: das Feld erscheint nicht in der Ausgabe, z.B. langer Quelltext oder eine Funktion.

Beispiel: *Concern* im Prototyp

```
@dataclass(frozen=True)
class Concern(ABC):
    name: str
    assertion: ClassVar[str]                # setzt jede Unterklasse

    @abstractmethod
    def detect(self, v: FunctionView) -> Optional[Evidence]: ...
    def __call__(self, fn=None, **meta):    # die Instanz *ist* der Decorator
        ...

stochastic = Stochastic("stochastic")      # erzeugt registry.py
@stochastic
def sample(papers): ...
```

- Ein Feld (`name`). Die **Behauptung** (`assertion`) und `detect` liefert jede Unterklasse.
- `frozen`: Kein Plugin kann einen Concern zur Laufzeit umbauen.

Beispiel: *FunctionView* und eine Fabrikmethode

```
@dataclass(frozen=True)
class FunctionView:
    name: str
    params: List[str]
    reads: List[str]
    writes: List[str]
    called: Set[str]
    ...

    @classmethod
    def of(cls, node: ast.AST) -> "FunctionView":
        ... # AST einmal durchlaufen
        return cls(name=..., params=..., reads=..., ...)
```

- `of` ist eine **Fabrikmethode**: Die Arbeit (AST lesen) steht dort, der generierte `__init__` speichert nur.
- Java: statische Methode `FunctionView.of(node)` vor dem Konstruktor des Records.

Abstrakte Klassen

Eine Methode ohne Rumpf

```
from abc import ABC, abstractmethod

class Concern(ABC):
    @abstractmethod
    def detect(self, v):
        """Liefert (Grund, Konfidenz) oder None."""
```

- ABC macht die Klasse **abstrakt**; @abstractmethod markiert die Methode, die jede Unterklasse schreiben **muss**.
- Der Rumpf ist leer, nur ein Docstring oder ...: Er beschreibt den Vertrag, nicht die Umsetzung.
- Die Basisklasse legt fest, *was* ein Concern ist (Name, Behauptung, Decorator). *Wie* er erkannt wird, entscheidet die Unterklasse.

Unterklasse implementiert *detect*

```
class Stochastic(Concern):
    assertion = ("The seed is an explicit parameter, "
                 "so the result can be regenerated.")

    def detect(self, v):
        hits = sorted(v.called & RNG_CALLS)
        if not hits:
            return None
        if _seed(v):
            return f"nutzt {hits[0]}, Seed als Parameter", "high"
        return f"nutzt {hits[0]} ohne Seed-Parameter", "high"
```

- Wie in Java heißt die Datei wie die Klasse: `stochastic/Stochastic.py`. `registry.py` lädt sie aus jedem Concern-Ordner und erzeugt die eine Instanz: `Stochastic("stochastic")`.
- `--inspect` fragt zu jeder markierten Funktion: *Does the assertion “...” hold?*

Was passiert, wenn *detect* fehlt?

```
class Vergessen(Concern):  
    assertion = "..."  
  
Vergessen("vergessen")  
# TypeError: Can't instantiate abstract class Vergessen  
#           without an implementation for abstract method 'detect'
```

- Der Fehler kommt beim **Erzeugen**, nicht erst beim ersten Aufruf von `detect`.
- Mit `raise NotImplementedError` statt `@abstractmethod` würde das Plugin geladen und erst mitten im Scan scheitern.
- ABC erzwingt nur Methoden, keine Attribute: Eine fehlende `assertion` meldet die Registry.
- Python prüft das zur Laufzeit beim Laden der Registry, Java schon beim Kompilieren.

Vergleich: Java

Abstrakte Klasse

```
public abstract class Concern {  
    final String name;  
  
    Concern(String name) {  
        this.name = name;  
    }  
  
    // kein Rumpf  
    abstract String assertion();  
    abstract Optional<Evidence>  
        detect(FunctionView v);  
}
```

Interface

```
interface Detector {  
    Optional<Evidence>  
        detect(FunctionView v);  
}
```

- Python kennt kein `interface`; eine ABC nur aus abstrakten Methoden spielt dieselbe Rolle.
- Python erlaubt Mehrfachvererbung, also auch mehrere ABCs.

Dataclass und ABC zusammen

```
@dataclass(frozen=True)
class Concern(ABC):
    name: str
    assertion: ClassVar[str]          # Klassenattribut, kein Feld im __init__

    @abstractmethod
    def detect(self, v): ...

class Stochastic(Concern):            # kein eigenes @dataclass nötig
    assertion = "The seed is an explicit parameter, ..."
    def detect(self, v): ...

Stochastic("stochastic")             # geerbtes __init__
```

- Die Unterklasse **erbt** das generierte `__init__`, `__repr__`, `__eq__` und `frozen`.
- Ein Java-`record` kann weder abstrakt sein noch erben. In Python lassen sich Wertklasse und abstrakte Basisklasse verbinden.

Das Modell im Prototyp

Convention over Configuration

```
def load_one(name: str) -> Concern:
    """Import ``<name>/<ClassName>.py`` and create its one instance."""
    cls = class_name(name)
    sub = getattr(importlib.import_module(f"__package__.{name}.{cls}"), cls, None)
    if not (isinstance(sub, type) and issubclass(sub, Concern)):
        raise ValueError(f"{name}/{cls}.py must define class {cls}(Concern)")
    c = sub(name)
    if not getattr(c, "assertion", ""):
        raise ValueError(f"concern {name!r} sets no assertion")
    return c
```

- Keine Konfigurationsdatei, keine Liste der Concerns: Der **Ordnername** legt alles fest.
- stochastic/ → Stochastic.py → class Stochastic(Concern) →
@stochastic
- Wer sich an die Konvention hält, registriert nichts. Wer abweicht, bekommt beim Laden einen klaren Fehler.

Verdict im Code

```
@dataclass
class Verdict:
    function: str
    location: str          # datei.py:zeile
    concern: str
    formula: str = ""      # nur @calculations
    verdict: str = "pending"
    note: str = ""

    @property
    def key(self):
        return (self.function,
                self.location,
                self.concern)
```

```
# inspection.yaml
inspections:
- function: "normalize"
  location: "sample_pipeline.py:22"
  concern: "calculations"
  formula: "normalize(...) =
           (value - lo) / (hi - lo)"
  verdict: "accepted"
  note: ""
```

- `key` findet das alte Urteil beim nächsten Durchlauf wieder.
- **Nicht** frozen: Das Review setzt `verdict`, nachdem die Antwort da ist.

Project: eine Klasse für den geprüften Code

```
@dataclass(frozen=True)
class Project:
    root: Path
    concerns: Dict[str, Concern]

    def __post_init__(self):
        object.__setattr__(self, "root", Path(self.root).resolve())

    def files(self): ...           # alle *.py ohne Tests
    def scan(self): ...           # -> (records, errors)
    def verdicts(self): ...       # inspection.yaml lesen
    def save(self, verdicts): ... # inspection.yaml schreiben
```

- **Vorher:** `scan(root, concerns)` und `review(root, concerns)`, und jede Funktion rechnete selbst aus, wo `inspection.yaml` liegt.
- Jetzt bündelt **ein** Objekt Pfad, Concerns und Ablage: Daten und die Operationen darauf gehören zusammen.

Wie die drei zusammenspielen

```
concerns = load()                # {name: Concern}
project  = Project("examples", concerns)

records, errors = project.scan()  # Concern.detect für jede Funktion
review(project)                  # pro (Funktion, Concern) ein Verdict
                                # -> project.save(...) nach jeder Antwort
```

- `python -m rse_annotations PATH` macht genau das: `--analyze` **bis** `scan`, `--inspect` **bis** `review`.
- **Concern** ist unveränderlich und für alle Projekte gleich, **Project** ist unveränderlich und beschreibt einen Prüflauf, **Verdict** ist das einzige Objekt, das sich im Lauf ändert.

Mitmachen



github.com/juliandehne/rse_code_annotations

Plan für den Kurs

- Jede Gruppe **forkt** das Repository und arbeitet in ihrem Fork.
- Neue Concerns und Verbesserungen kommen per **Pull Request** zurück ins Repository.
- Probleme mit dem Basiscode: ein **Issue** im Repository.