Winter Semester 16/17

Software Engineering Design & Construction

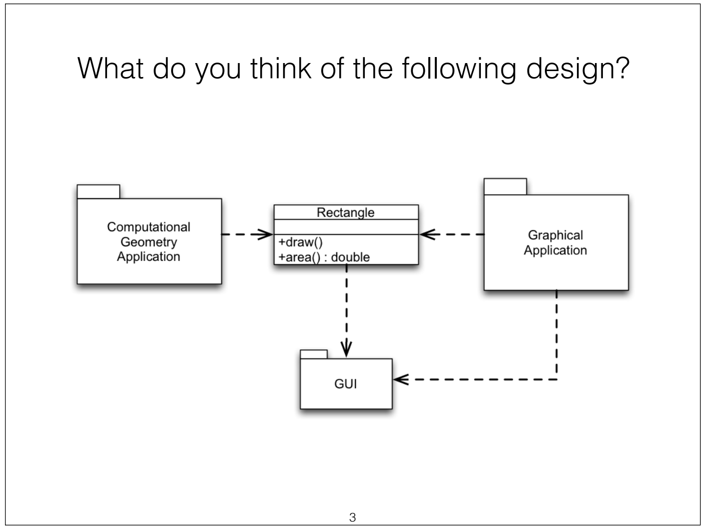
Dr. Michael Eichberg
Fachgebiet Softwaretechnik
Technische Universität Darmstadt

Single Responsibility Principle

***S**ingle **R**esponsibility **P**rinciple*

A class should have only one reason to change.

–Agile Software Development; Robert C. Martin; Prentice Hall, 2003



What do you think of the following design?

Observation

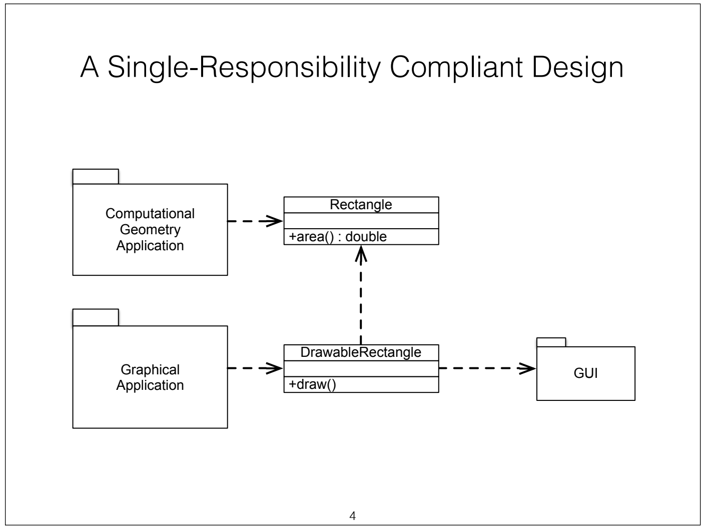
- `Rectangle` provides a method to draw rectangle shapes on the screen. For that, Rectangle uses GUI to implement draw().
- `GeometricApplication` is a package for geometrical computations, which also uses Rectangle (area()).
- `GeometricApplication` **depends on GUI (GUI has to be deployed along with Rectangle) even if it only needs the geometrical functions of rectangles.**

Evaluation

- `Rectangle` has multiple responsibilities: (1) Geometrics of rectangles: `area()` and (2) Drawing of rectangles: `draw()`
- `Rectangle` has low cohesion!
- It is not a representation of a coherent concept, but a point to bundle needed functionality without consideration of their cohesion. Geometrics and drawing do not naturally belong together.

Problems

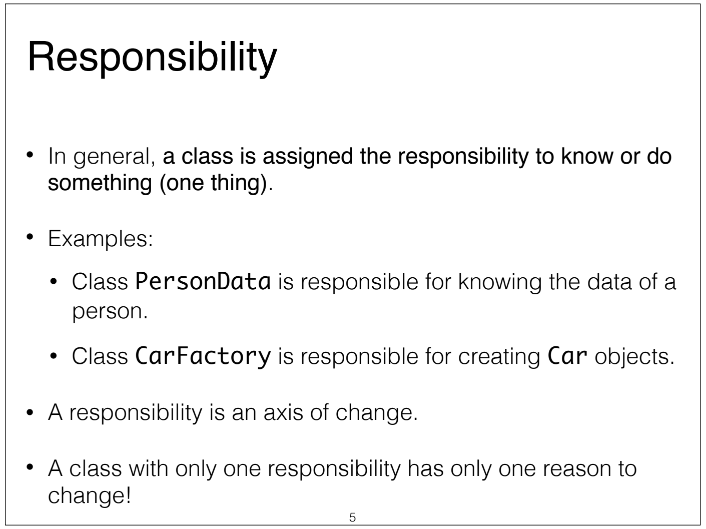
- `Rectangle` has multiple reasons to change.
- If drawing functionality changes in the future, we need to retest and redeploy Rectangle in context of GeometricalApplication!



A Single-Responsibility Compliant Design

Assessment

- Split responsibilities:
  - `Rectangle` models geometric properties of rectangles.
  - `DrawableRectangle` models visual properties of graphical rectangles.
- Computational Geometry Application uses only `Rectangle`. It only depends on the geometrical aspects.
- Graphical Application uses `DrawableRectangle` and indirectly `Rectangle`. It needs both aspects and therefore depends on both.
- Both classes can be reused easily!
  Only changes to the responsibilities we use will affect us.
- Both classes are easily understood!
  Each implements one concept.
  `Rectangle` represents a rectangle shape by its geometric properties.
  `DrawableRectangle` represents a rectangle by its visual properties.

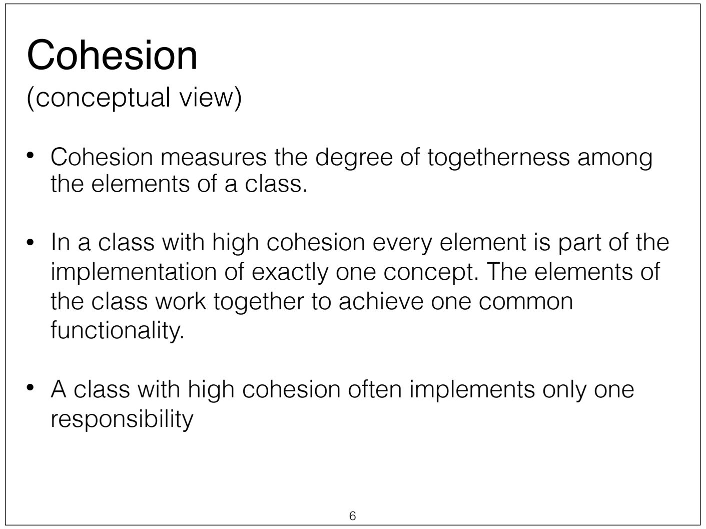


Responsibility

- In general, a class is assigned the responsibility to know or do something (one thing).
- Examples:
  - Class `PersonData` is responsible for knowing the data of a person.
  - Class `CarFactory` is responsible for creating `Car` objects.
- A responsibility is an axis of change.
- A class with only one responsibility has only one reason to change!

5

In general, if new functionality must be achieved, or existing functionality needs to be changed, the responsibilities of classes must be changed.



Cohesion
(conceptual view)

- Cohesion measures the degree of togetherness among the elements of a class.
- In a class with high cohesion every element is part of the implementation of exactly one concept. The elements of the class work together to achieve one common functionality.
- A class with high cohesion often implements only one responsibility

6

Cohesion actually *measures* the extent to which operations and data within a class belong to the concept this class is representing. Therefore, a class with low cohesion – i.e., a class where the operations and data actually belongs to several concepts – violates the single-responsibility principle.

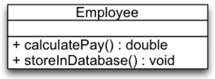
Common metrics that are defined to measure the cohesion (such as LCOM(*)) are usually not working at the conceptual level and hence, would identify a class such as `PersonData` that stores information regarding a person and which usually offers a large number of "getter" and "setter" methods as non-cohesive. But, from a conceptual perspective this class is cohesive.

SRP and Cohesion

- Applying the single-responsibility principle maximizes the cohesion of classes.
- Classes with high cohesion ...
  - can be reused easily,
  - are easily understood,
  - protect clients from changes, that should not affect them.

In other words, applying the SRP improves reusability and comprehensibility (→ maintainability).

Should we split the responsibilities of this class?

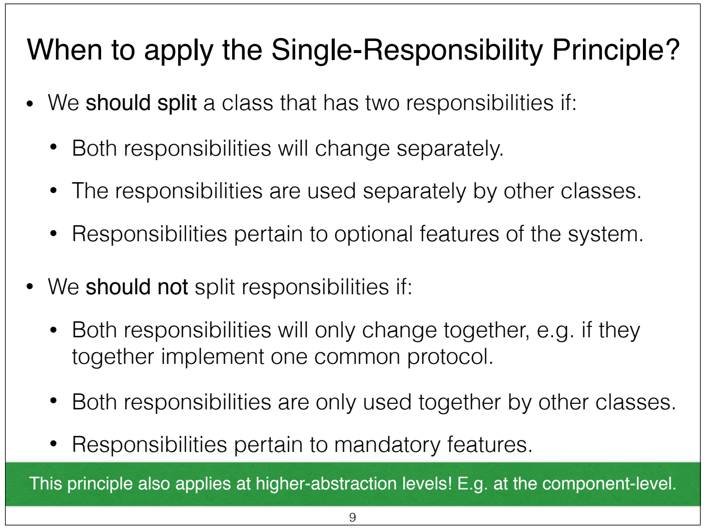


The class `Employee` which has two responsibilities:

1. Calculating an employee’s payment.
2. Storing employee data in the database.

Calculating the payment is part of the business rules. It corresponds to a real-world concept the application shall implement. Storing information in the database is a technical aspect. It is a necessity of the IT architecture that we have selected; does not correspond to a real-world concept.

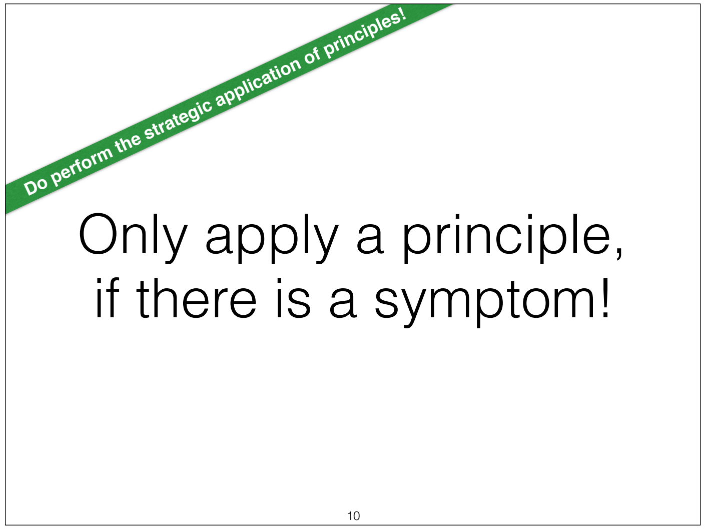
Mixing business rules and technical aspects is calling for trouble! **From experience**, we know that both aspects are extremely volatile. Hence, most probably we should split the class in this case.



When to apply the Single-Responsibility Principle?

- We **should split** a class that has two responsibilities if:
  - Both responsibilities will change separately.
  - The responsibilities are used separately by other classes.
  - Responsibilities pertain to optional features of the system.
- We **should not** split responsibilities if:
  - Both responsibilities will only change together, e.g. if they together implement one common protocol.
  - Both responsibilities are only used together by other classes.
  - Responsibilities pertain to mandatory features.

This principle also applies at higher-abstraction levels! E.g. at the component-level.



Do perform the strategic application of principles!

Only apply a principle,
if there is a symptom!

Be agile and modify the design when needed.

Choose the kinds of changes to guide your application of the single-responsibility principle. Guess the most likely kinds of changes derived from experience. Experienced designers hope to know the user and an industry well enough to judge the probability of different kinds of changes.
Invoke the single-responsibility principle against the most probable changes.

An axis of change is an axis of change only if the change actually occurs.