Software Engineering Essentials

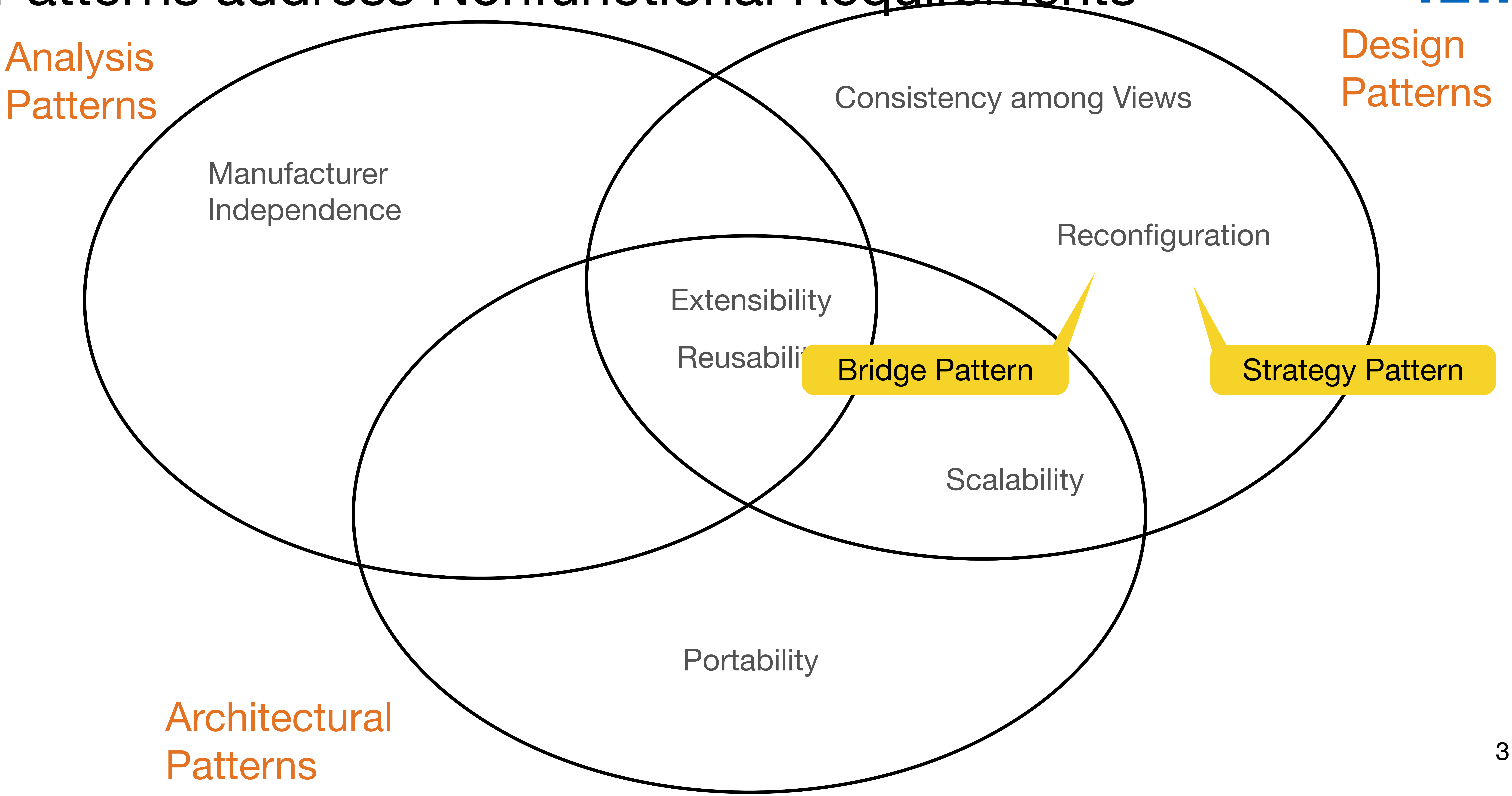
Design Patterns

Bernd Bruegge, Stephan Krusche, Andreas Seitz, Jan Knobloch
Chair for Applied Software Engineering — Faculty of Informatics

Learning Goals

1) Understand why design patterns are useful in software engineering
2) Analyze the different types of design patterns
3) Apply the bridge pattern and strategy pattern

Patterns address Nonfunctional Requirements



Why are Design Patterns good?

- They are generalizations of detailed design knowledge from existing systems
- They provide a shared vocabulary
- They provide examples of reusable designs
  - Polymorphism (inheritance, sub-classing)
  - Delegation (or aggregation)

Many Design Patterns and (Architectural Patterns) use a combination of polymorphism and delegation.

3 Types of Design Pattern (“GoF Patterns”)

Structural Patterns

- Reduce coupling between two or more classes
- Introduce an abstract class to enable future extensions
- Encapsulate complex structures

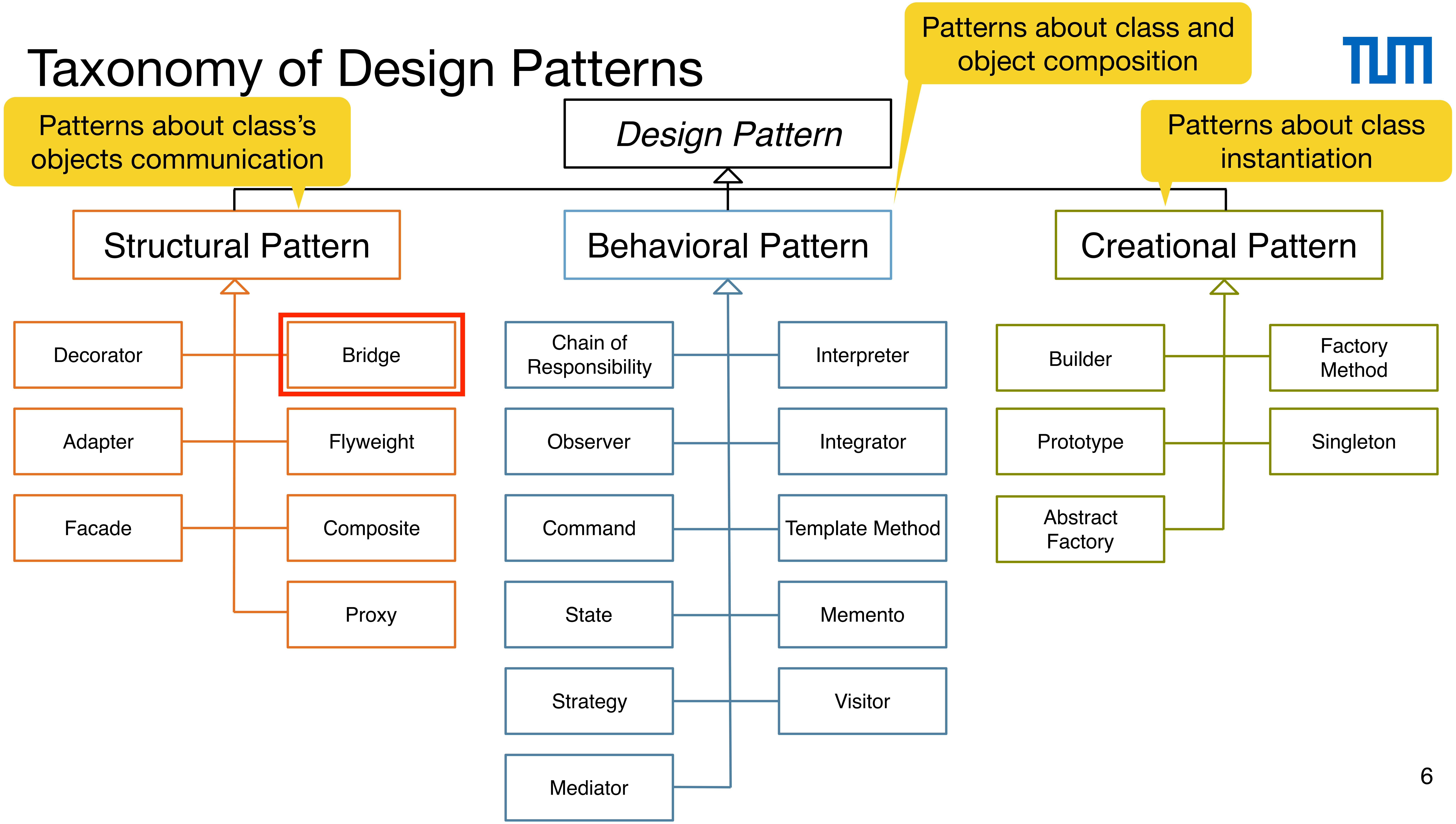
Behavioral Patterns

- Allow a choice between algorithms and the assignment of responsibilities to objects (“Who does what?”)
- Simplify complex control flows that are difficult to follow at runtime

Creational Patterns

- Allow a simplified view from complex instantiation processes
- Make systems independent from the way its objects are created, composed and represented

Taxonomy of Design Patterns



Bridge Pattern

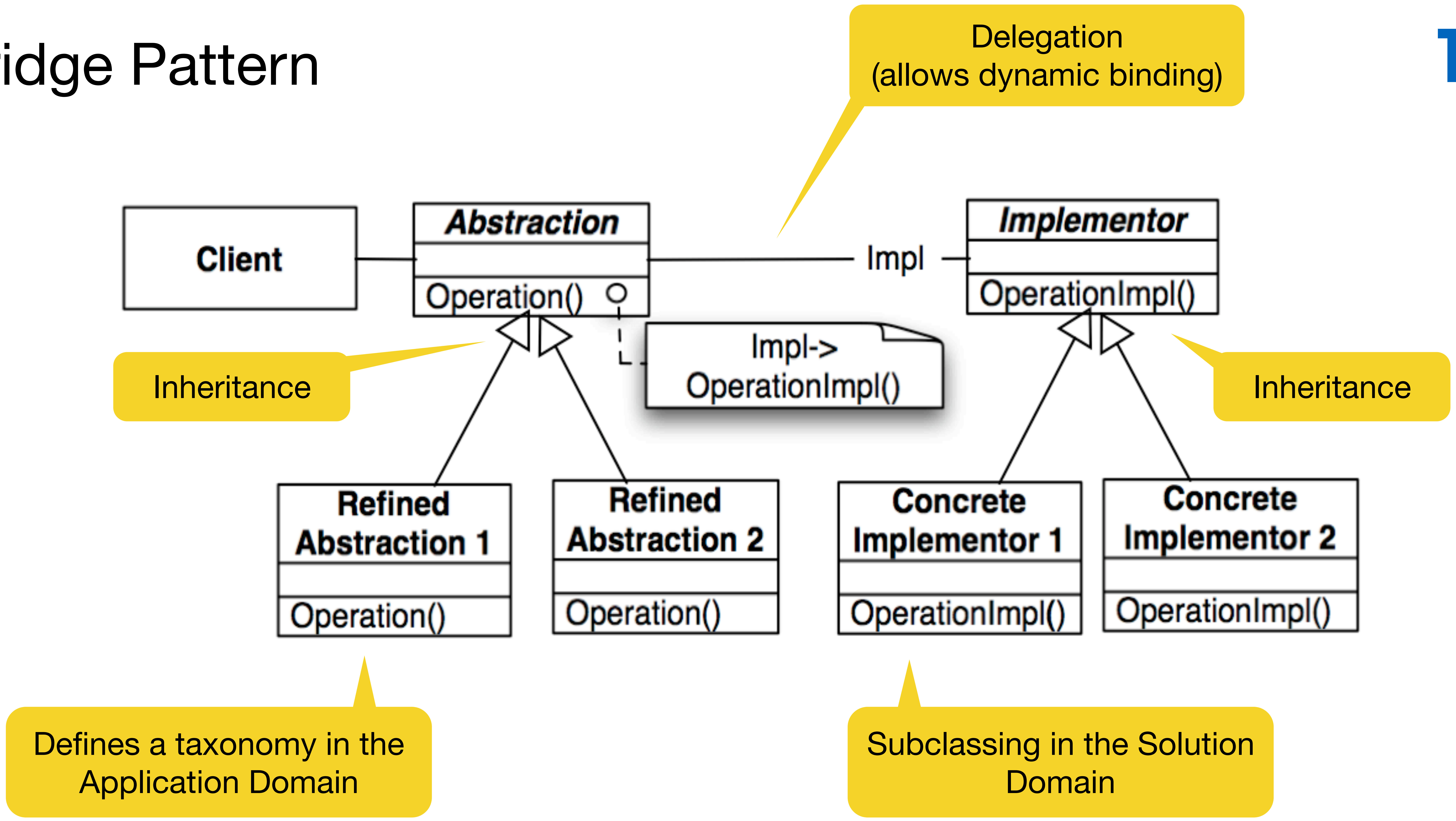
The Bridge Pattern allows to postpone design decisions to the startup time of a system

Problem: Many design are made final at design time or at compile time

- Often it is desirable to delay design decisions until run time
  - Example: We want to support two types of clients:
    - Client 1 uses a very old implementation of an algorithm
    - Client 2 uses a modern implementation of the algorithm
- The Bridge Pattern allows to delay the binding between a interface and its implementation to the startup time of the system

e.g. in the constructor of the implementation class

Bridge Pattern

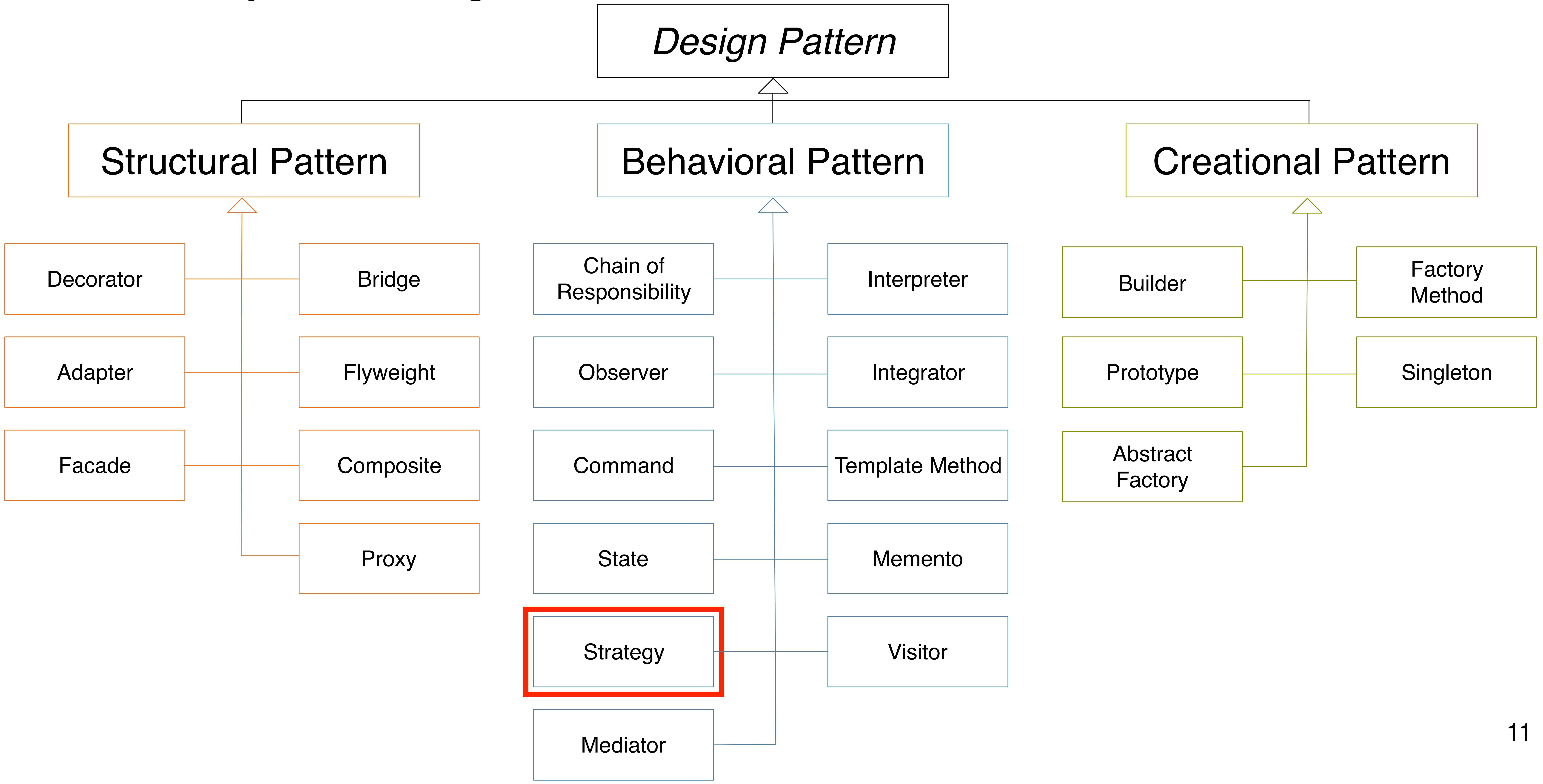


Why the name Bridge Pattern ?

It provides a bridge between the abstraction (in the application domain) and the implementor (in the solution domain)

Client
Abstraction
Impl
Implementor
OperationImpl()
Operation()
ation()
Concre
OperationI
tionImpl()

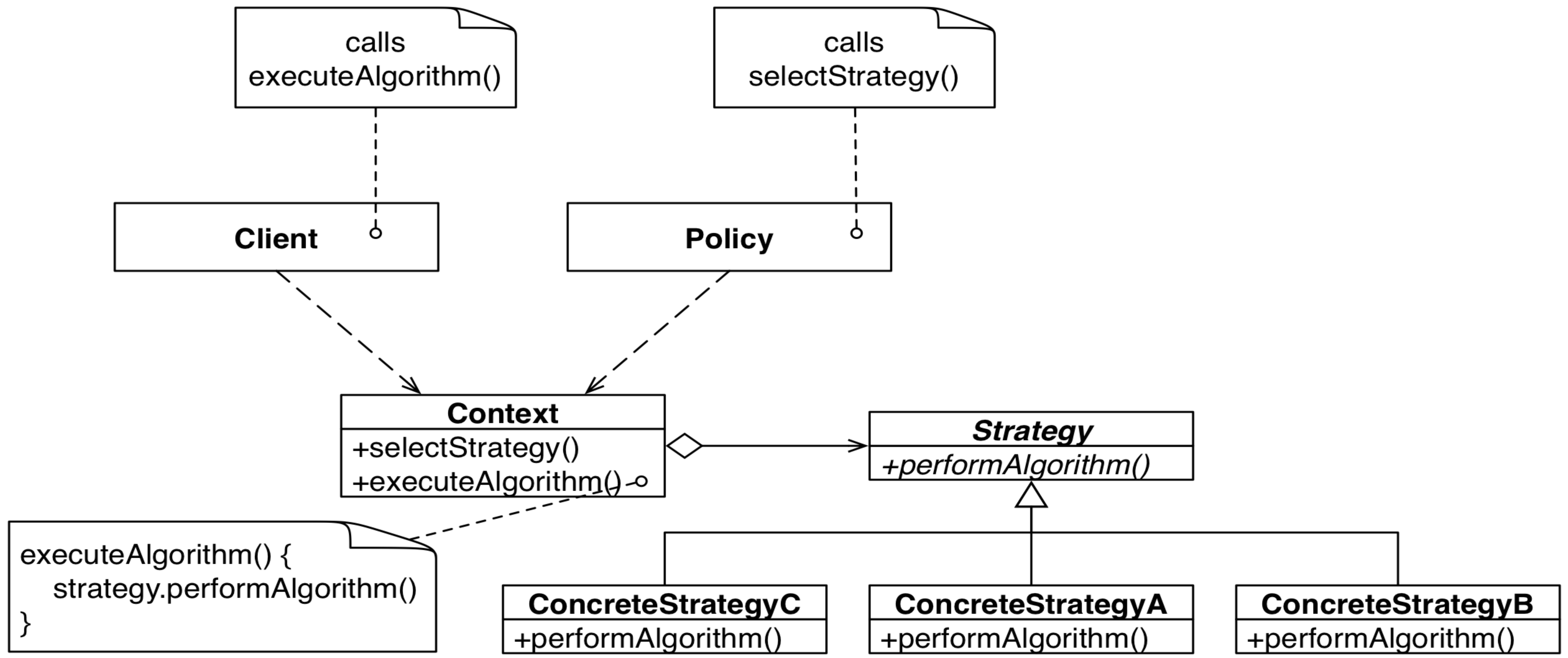
Taxonomy of Design Patterns



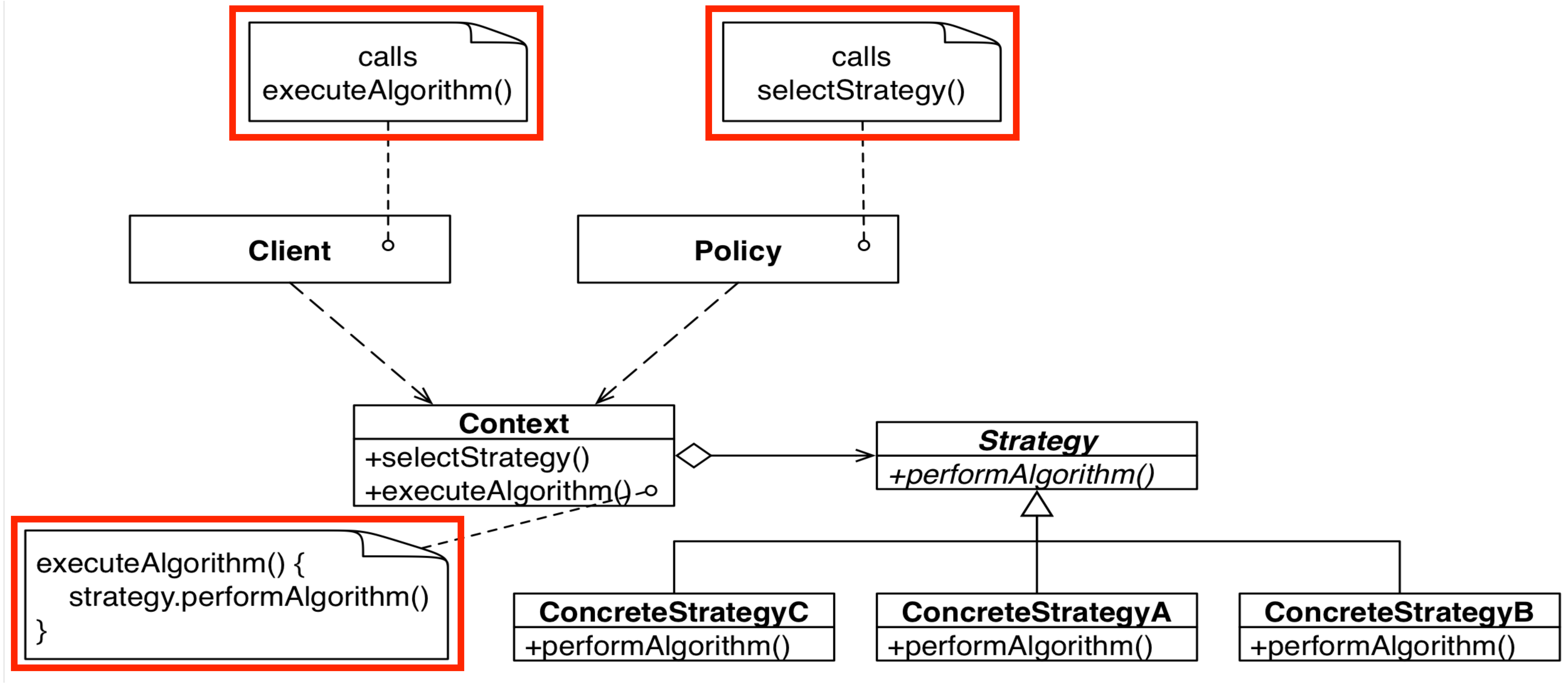
Strategy Pattern

- Situations, where different algorithms exist for a specific task
- Example: Sorting a list of students
  - Algorithms: bubble sort, quick sort, merge sort
- Different variants of an algorithm that describes trade-offs between space and time
  - A specific implementation is selected based on the current context at runtime
- Different algorithms will be appropriate at different times
  - Use an algorithm that is slow, but can be implemented fast for rapid prototyping
  - Use an algorithm that is fast, but takes some time to implement for the delivery of the final product
- When we add a new algorithm, we want to add it easily without disturbing existing applications that are using already another algorithm

Strategy Pattern (Responsibilities)



Strategy Pattern (Responsibilities)



```
executeAlgorithm() {
    strategy.performAlgorithm()
}
```

Comparison: Bridge vs. Strategy Pattern

- The bridge pattern is used for structural decisions
  - It decouples abstractions from their implementations
  - Used to delay system design decisions all the way to system startup
    - Depending on the client a specific implementation is chosen at $\boxed{\text{startup}}$ time
- The strategy pattern is used for behavioral decisions
  - Depending on the policy, a specific algorithm is chosen at $\boxed{\text{runtime}}$
  - The choice of the algorithm depends on the policy used in the application and is independent from the client using it

Software Engineering Essentials

Design Patterns

Bernd Bruegge, Stephan Krusche, Andreas Seitz, Jan Knobloch
Chair for Applied Software Engineering — Faculty of Informatics