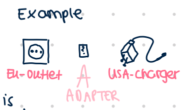
LECTURE 7

10.06.2021

OUTLINE
1) Adapter pattern
2) Observer pattern
4) Strategy pattern

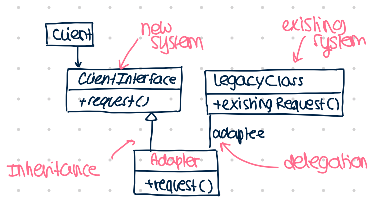
1) ADAPTER PATTERN

Example



- PROBLEM: An existing component offers functionality, but is not compatible with the new system being developed
- SOLUTION: Adapter pattern connects incompatible components
  - reuse of existing components
  - converts provided interface into required one

→ also called WRAPPER



definition: Legacy system

= an old system that continues to be used, even though newer technology or more efficient methods are available

- Often designed without modern software design methodologies or source code was lost
  → high maintenance cost
- Irreplaceable because the re-implementation would be expensive or impossible

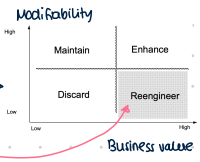
⚡ Problems with legacy systems:

Reasons for the continued use:

- System cost: still makes money
- Poor engineering (or management): lost source code
- Availability: system requires 100% availability
- Pragmatism: system is installed and working

⚡ BUT: change is required due to new functional-, non functional- or pseudo requirements

What to do with them? →



system = irreplaceable
⟹ ADAPTER

Comparison: adapter vs. bridge pattern

- Similarities: → Both hide details of underlying implementation
- Differences:

ADAPTER:	BRIDGE:
→ makes unrelated components work together	→ used to let abstractions and implementations vary independently
Inheritance → delegation	Delegation → Inheritance

2) OBSERVER PATTERN

PROBLEM: - Object that often changes its state
- Multiple views of current state

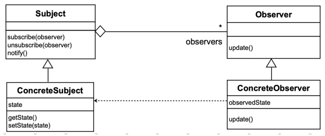
Requirements:
- Consistency across the (redundant) views, whenever observed object changes
- Highly extensible system design
- Possibility to add new elements

SOLUTION: model 1-to-many dependency between objects
⇒ connect the state of an observed object, the subject with many observing objects, the observers

Benefits:
- maintain consistency across redundant observers
- optimize a batch of changes to maintain consistency

Also called: „Publish and Subscribe"

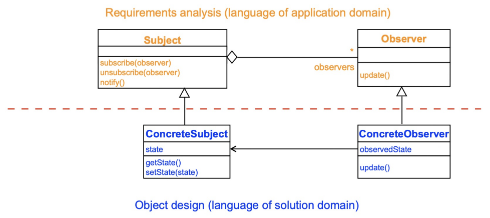
The observer pattern decouples a subject from its observer



- Subject represents entity object
  → state contained in ConcreteSubject
- observer subscribes() to subject
- Each concrete observer has a different view of the state of

3 Variants:

1. Notification + pull: When state changes → notify() is called → calls update() in each observer
   → observer can decide whether to pull the state by calling getState()
2. Notification with push: Subject also includes the state that has been changed in each update(state) call
3. Periodic pull: observer periodically (e.g. every 5s) pulls state of subject by calling getState()



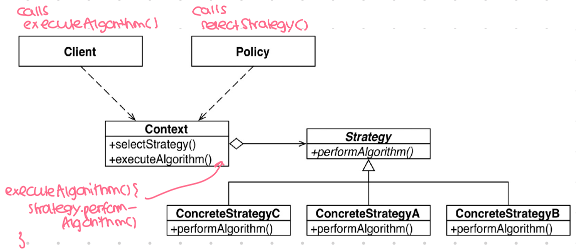
Strategy pattern

3. STRATEGY PATTERN

PROBLEM: Different algorithms exist for a specific task
(Example: sorting)

If we need a new algorithm, we want to add it without changing the rest of the application or the other algorithms

SOLUTION: strategy pattern allows to switch between different algorithms at run time based on the context and a policy



good luck ♡

Clues for the use of design patterns

- **Text:** “complex structure”, “must have variable depth and width”
  ➡ **Composite Pattern**
- **Text:** “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime”
  ➡ **Strategy Pattern**
- **Text:** “must be location transparent”
  ➡ **Proxy Pattern**
- **Text:** “states must be synchronized”, “many systems must be notified”
  ➡ **Observer Pattern** (MVC architectural pattern)
- **Text:** “must interface with an existing object”
  ➡ **Adapter Pattern**
- **Text:** “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility”
  ➡ **Bridge Pattern**
- **Text:** “must interface to an existing set of objects”, “must interface to an existing API”, “must interface to an existing service”
  ➡ **Façade Pattern**

Summary

- Design patterns combine inheritance and delegation
- Adapter Pattern: connects incompatible components and allows the reuse of existing components
- Observer Pattern: maintains consistency across multiple observers: basis for MVC
- Strategy Pattern: switches between multiple implementations of an algorithm at runtime based on context and a policy