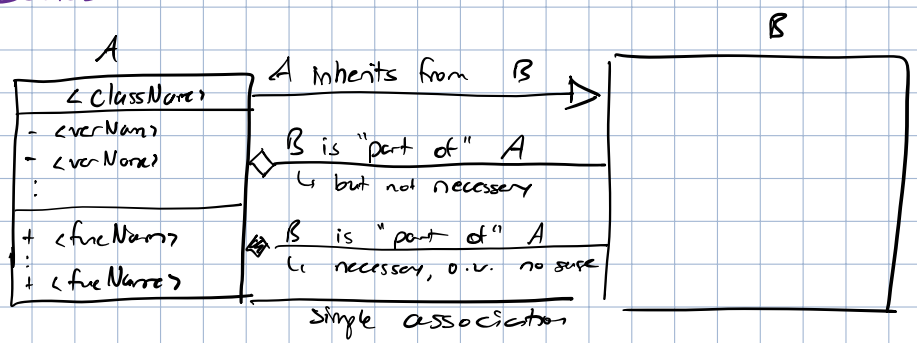
Contents:

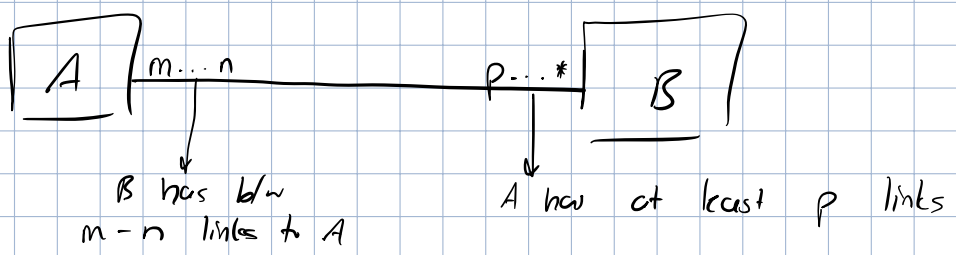
- Design pattens : usage, ~~UML diagram~~
- Architectural styles
- UML diagram basics

DESIGN PATTERNS

UML Diagram Basics

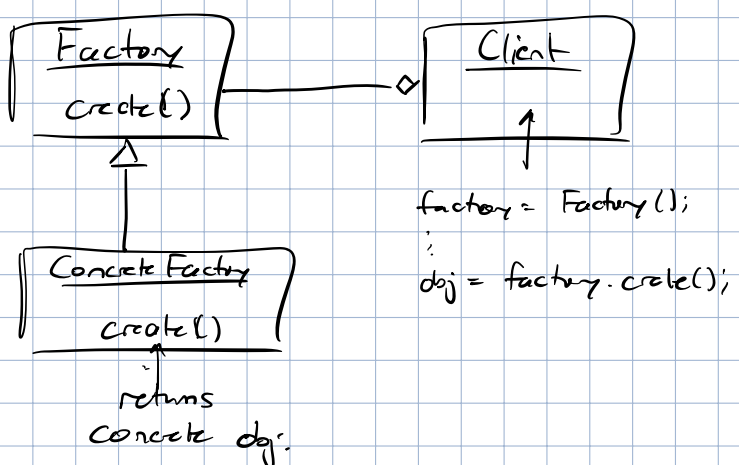


Multiplicities:



Creational Patterns

① Factory

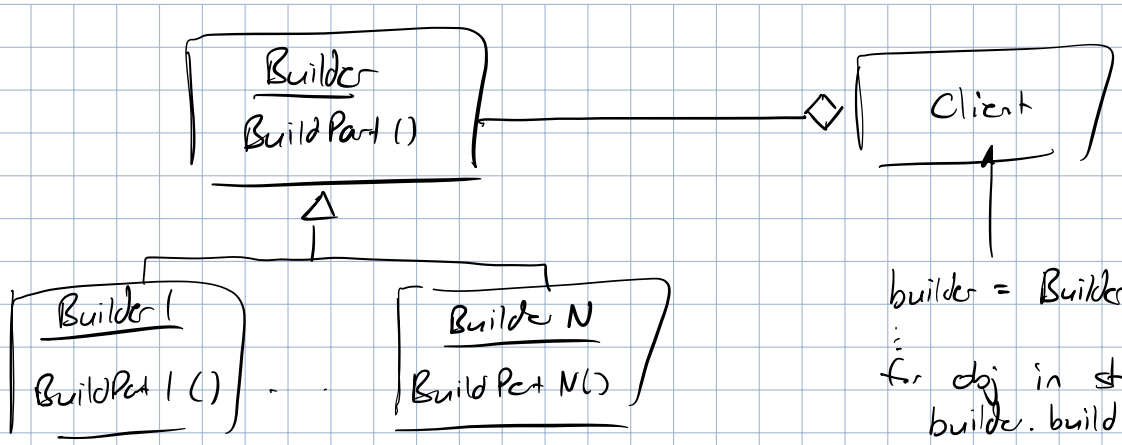


Abstract factory: abstract Factory class & sub'class Factories from main class

Pros: complex creation → Factory can handle it so client doesn't need to worry

Problem solved: complex object creation

② Builder patten.

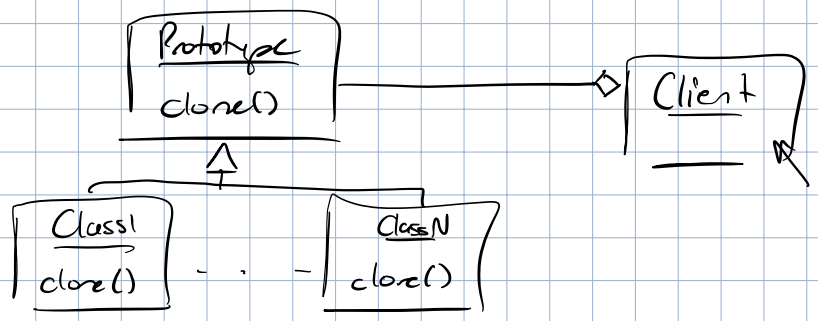



```
builder = Builder();
⋮
for obj in struct:
    builder.build();
```

Problem: Hard to have custom steps in building obj.

Pros: custom/dynamic building

③ Prototype pattern




```
cloner = Prototype();
⋮
copy = cloner.clone();
```

Problem: cloning is too coupled w/ class

Pros: copying & cloning delegated to class rather than client code.

④ Singleton

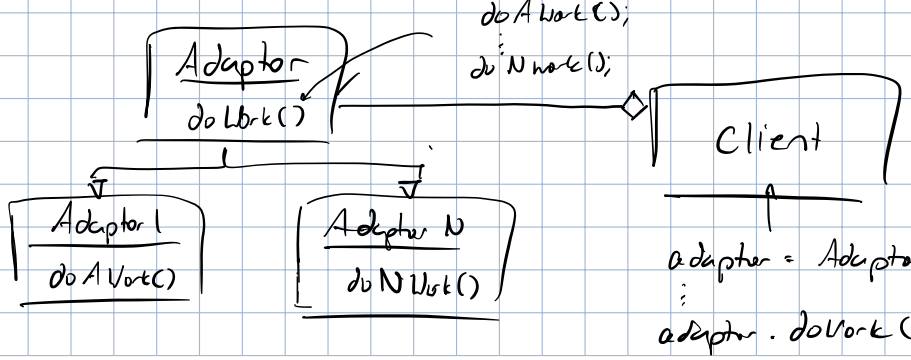
Create 1 instance that is shared across entire program lifecycle.

Problem: Hard access to global stuff.

Pros: global class w/ global logic

Structural Patterns

① Adaptor pattern



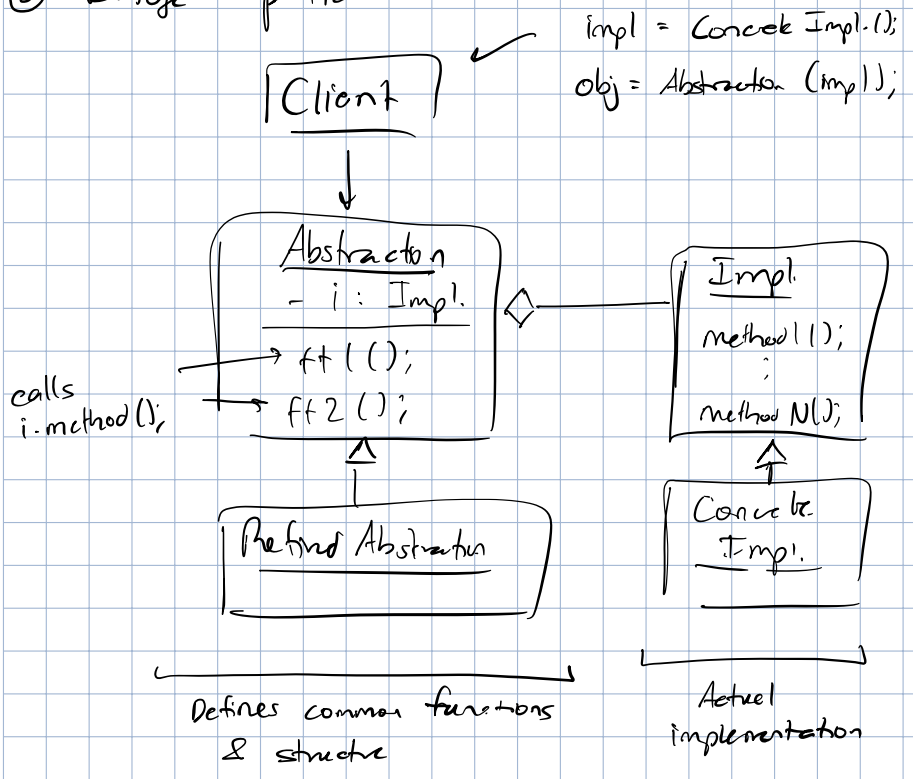

```
adapter = Adaptor();
⋮
adapter.doWork();
```

Problem: classes have incompatible interfaces

Pros: hides conversion complexity b/w diff. interfaces

Object adapter pattern: adapter has adaptees objects to call methods rather than using multiple inheritance to directly call methods.

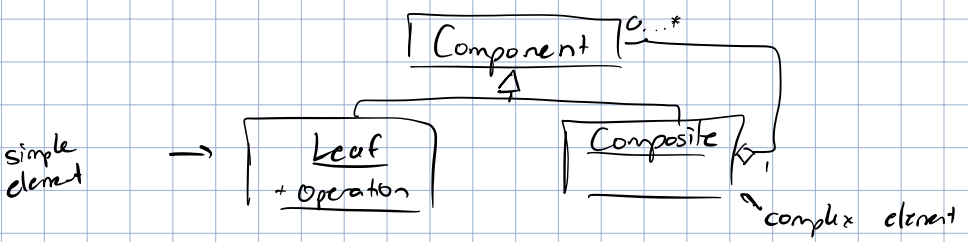
② Bridge pattern



Problem: want to separate class structure from class behavior — s.o. will lead to too many classes

Pros: impl. & structure separated.

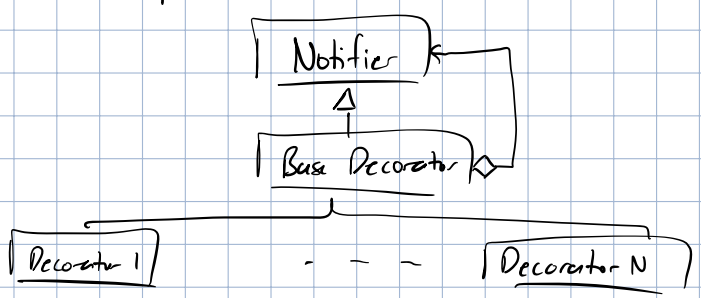
② Composite pattern



Problem: nested structure of code is hard to make computations

Pros: can work w/ tree structures

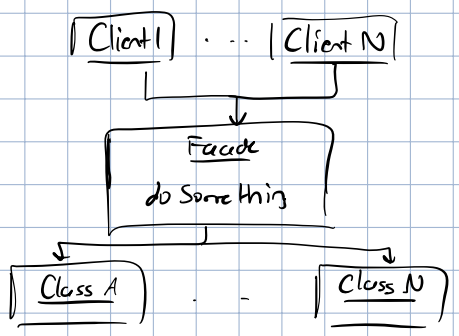
④ Decorator pattern



Problem: add custom behavior on an object w/out creating subclasses

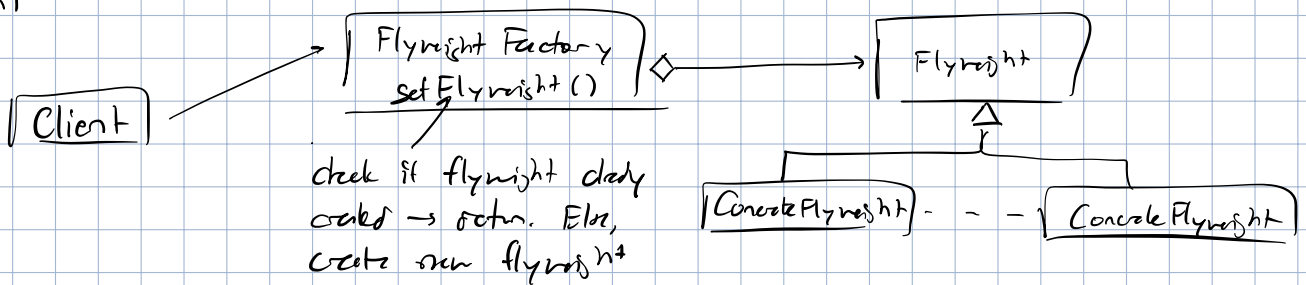
Pros: Extra behavior w/out too much code at runtime

⑤ Facade pattern



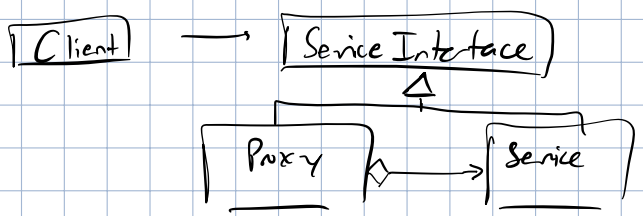
Problem: interface to library is too complex

⑥ Flyweight



Problem: don't want to keep generating new objects w/ shared state → too much memory used. Share memory b/w objects

⑦ Proxy



Problem: want to wrap actual service w/ custom behaviour / lazy init / control lifecycle of underlying service

Behavioural patterns

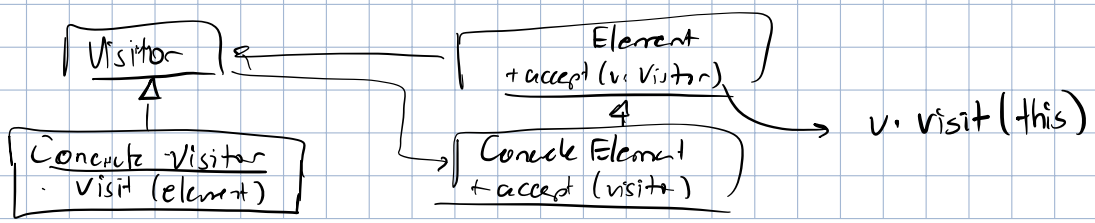
① Interpreter pattern

Implement specialized language to solve specific problem

② Template method pattern

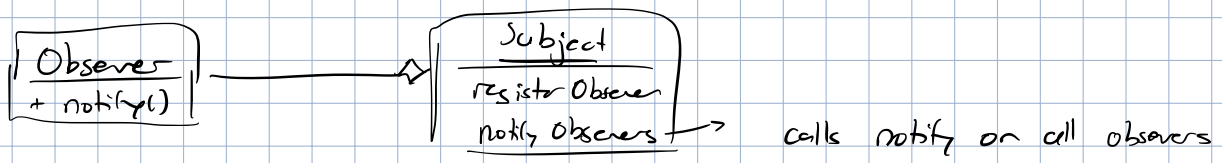
Create a template class that outlines steps of algo & let subclasses override specific steps

③ Visitor pattern



Problem: perform op. on complex object structure w/out modifying behaviour

④ Observer pattern



ARCHITECTURAL STYLES

① Layered

② Batch-sequential

③ Pipe-and-filter

④ Blackboard

⑤ Rule-based

⑥ Interpreter

⑦ Implicit invocation

⑧ P2P

ARCHITECTURAL MODELING

① C4: context → container → component → code

② 4+1: logical + physical + process + development + use scenarios

DESIGN PRINCIPLES

S ingleton
T ightly coupled
U ntestable
P remature optimization
I ndescriptive names
D uplication

S ingle responsibility
O pen/closed
L iskov substitutability
I nterface segregation
D ependency inversion

SECURITY PRINCIPLES

① Least privilege

② Fail-safe default

③ Economy of mechanism

④ Complete mediation

⑤ Open design

⑥ Separation of privilege

⑦ Least common mechanism

⑧ Psychological acceptability

⑨ Defense in depth

TYPE SYSTEMS

① Type hierarchy : subtypes

② Type rules : what rules are illegal

③ Type introduction : types of literals

④ Dataflow : run-time tests