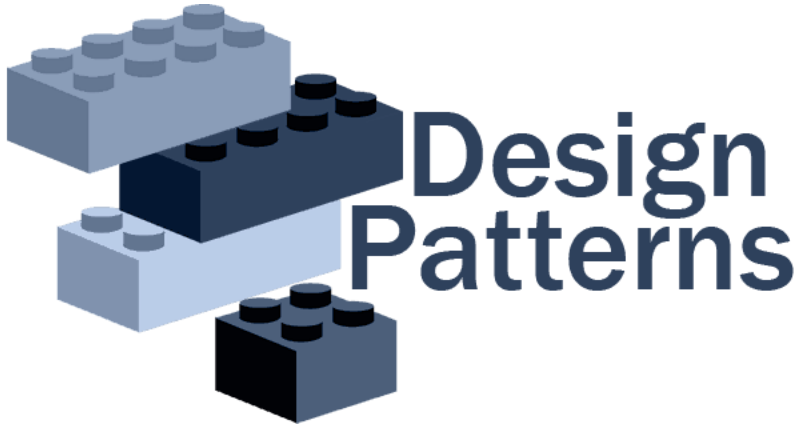
Advanced Software Development

Johan Bontes (johan@digitsolutions.nl)

CSC3003S

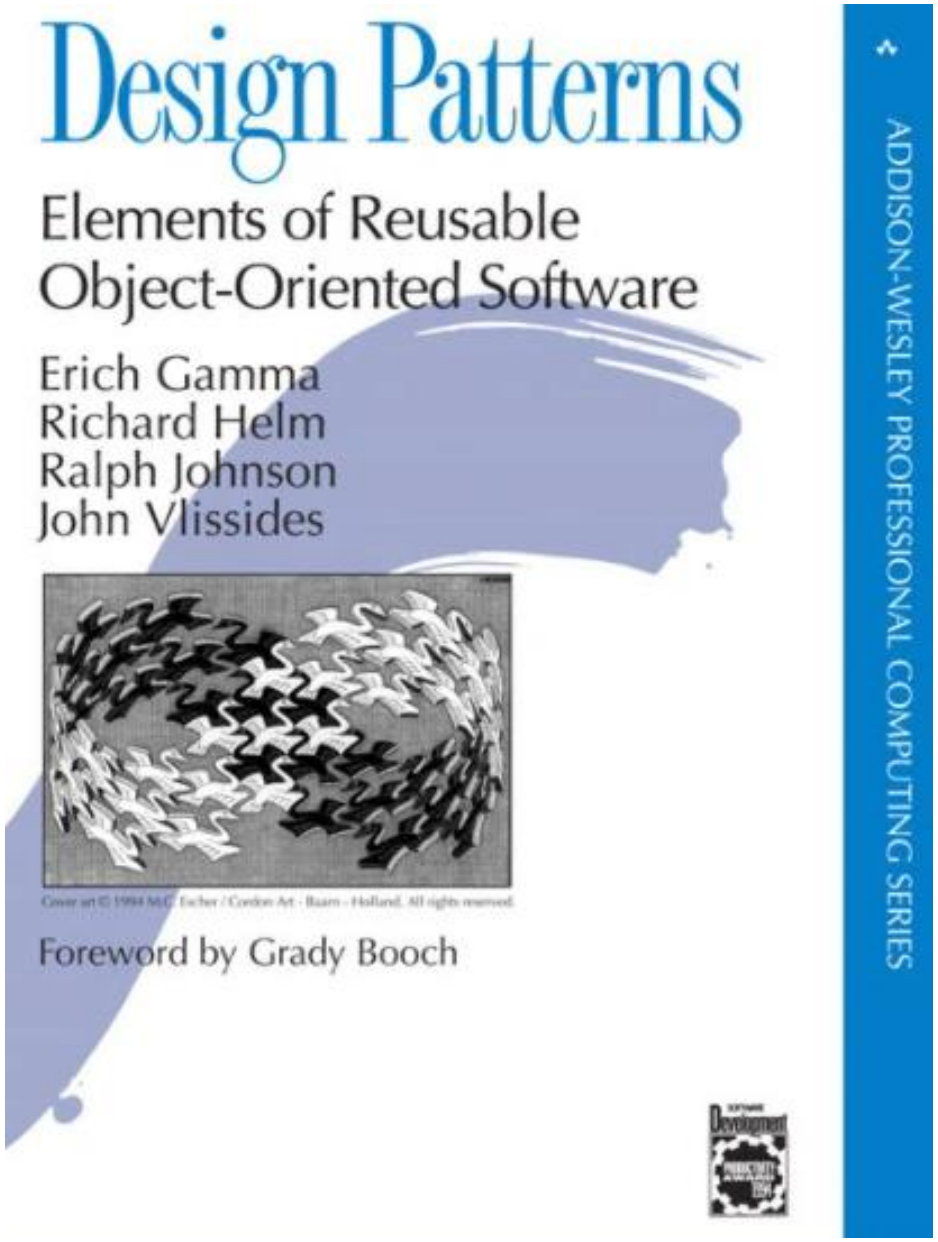
CSC3003S ASD12
GoF Design Patterns



A. What are Design Patterns

B. How are they Classified

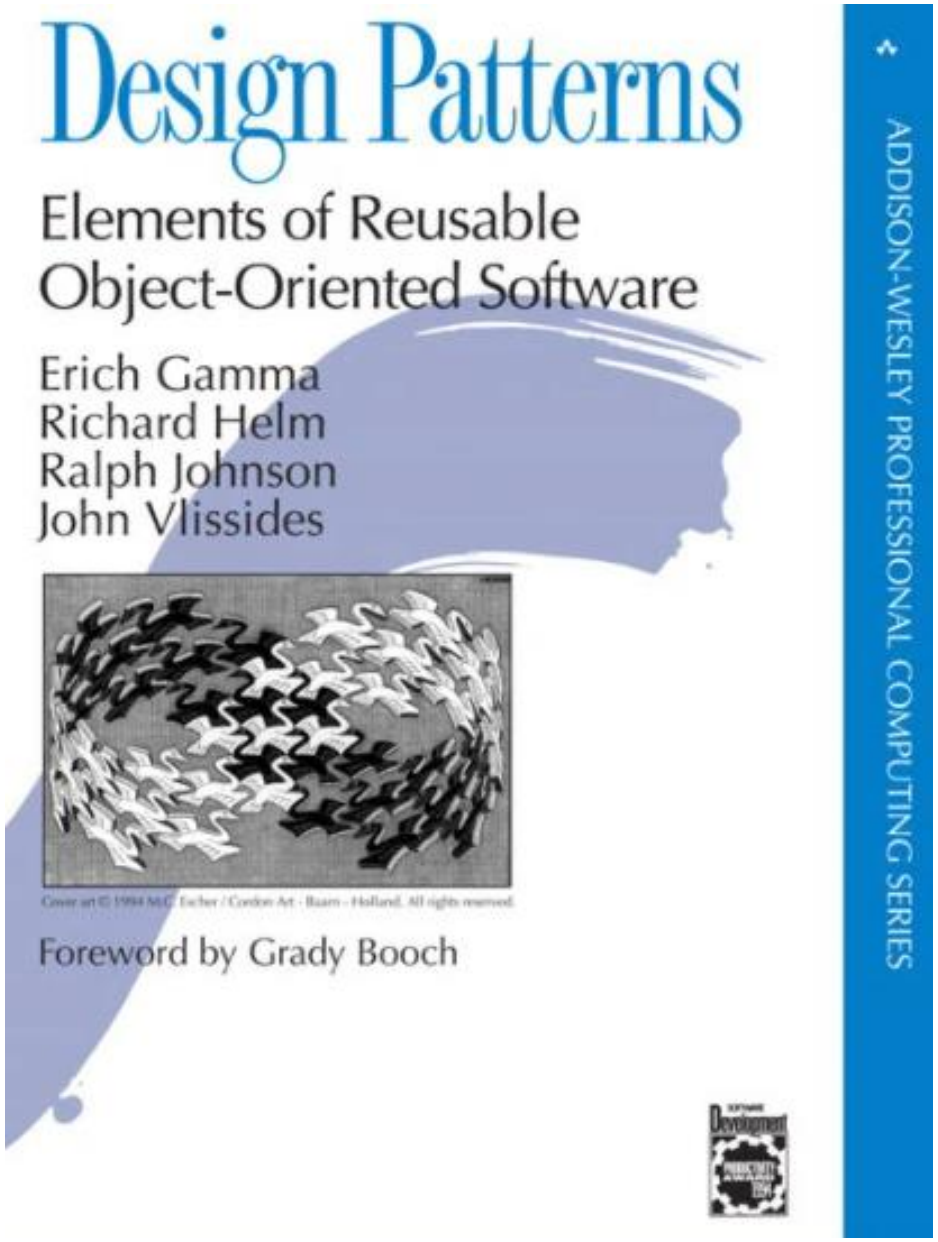
C. Examples



Design patterns are **solutions to general problems** that software developers face

"Patterns … are ways to describe **best practices, good designs**, and capture experience in a way that it is possible **for others to reuse** this experience"

Hillside.net/patterns



1994

Design patterns are ~~solutions~~ **workarounds** to general ~~problems~~ **to missing features in your language** that software developers face

New language features (esp. lambdas) have made many patterns obsolete

Lambdas λ

Lisp: (1960)

C++11 (2012)

Java SE 8 (2014)

Python 2.2 (2002) *(Broken version already in 1994)*

Von Neumann computer

Code and data are the same

OOP: separation between code and data

```
class X {
   code: methods = fixed
   data: members: variable
}
```

Functional code: data can also be a function

Anatomy of a lambda

Lambda: anonymous function that can capture outside variables

```
int i;                    external variable, may be captured
const auto X =            named lambda, no dedicated type
  [=/&,...]               capture: [=] by value, [&]: by reference
   (params)               optional parameters
   const/mutable          capture by value: const or not.
{  //error in Java
    i++;                  body, outside i has been captured
    int a = i + 2; //valid in Java
}
```

Note: two identical lambdas have different types

Lambdas in Java

- Somewhat broken, because captures always const

- Lambdas are not more powerful than named functions

Lambdas in other languages

- Can capture values by reference

= Changes inside lambda also change value outside

=> very powerful, e.g.: sorting

Lambdas are data

- Can add them to a collection of ‘workers’
- Call each worker using for in loop

```
for (auto a: WorkList) { a(); } //C++
for (var a: WorkList) { a(); } Java
```

Captures

What can be captured?

Anything that was in scope when lambda was defined.

When in doubt, just pass items as parameters

```
int i1;          //C++ can pass anything by ref
MyInteger i2; //Object can be passed by reference
for (auto a: WorkList) { a(i1); } //C++
for (var a: WorkList) { a(i2); } //Java
```

Lambda demo

DEMO

Lambdas internally

Every capture is added as a member field

```
int i = 0; int r = 10;
auto lambda = [&](){
  i++; r++;
};
class lambda {
  int& i;
  int& r;
  lambda(int& i, int& r): i(i), r(r) {}
  void dostuff() { i++; r++; }
};
```

Three pattern classes

Creational

- Create objects without having to know all the details

Structural

- Assemble objects into larger structures, which are flexible and efficient

Behavioral

- Communication between objects

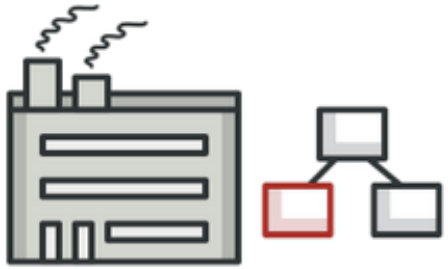
Disclaimer

Most content in this lecture adapted from

Dive into design patterns

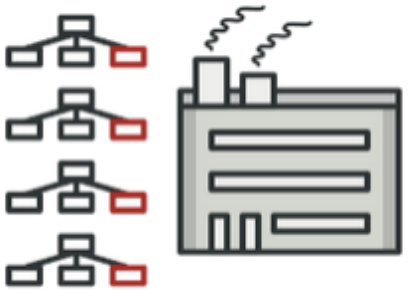
https://refactoring.guru/design-patterns/book

Creational Patterns



Factory method

Polymorphic construction

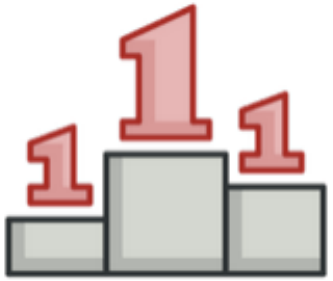


Abstract factory

More complex construction

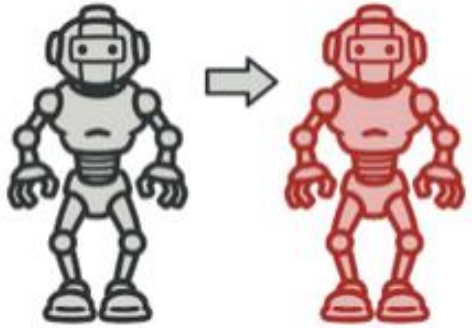
Builder

Build in stages



Singleton

“There can be only one”



Prototype

Java: `object.clone()`

Factory method

Polymorphic construction

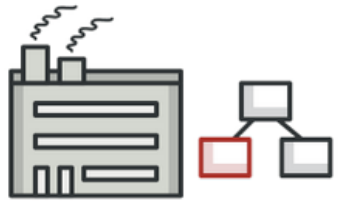
- Polymorphic => virtual methods
- Java, C++:
  no virtual constructors
  - Other languages do, e.g.: Pascal

Why C++/Java not?

C++: Child object is created first (incorrect order)

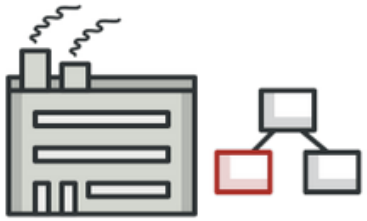
Pascal: Parent object first (Got objects later, flaw fixed)

```
//Class type, describes family
type TParentClass = class of TParent;
constructor TParent.Create() virtual;
begin
  Writeln('Parent.Create');
end;

constructor TChild.Create() override;
begin
  //inherited Create();
  Writeln('Child.Create');
end;

var
  firstClass: TParentClass;
  first: TParent;
begin
  firstClass := TChild;
  first := firstClass.Create();
end.
```



Factory method

Java



Text file with serialised object data
=> recreate objects from that file.

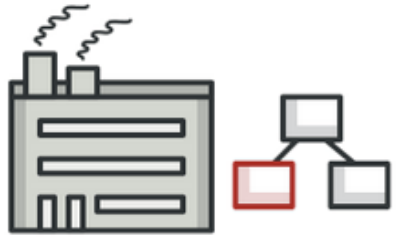
Problem:

- cannot restrict constructor signature; not virtual.
- cannot call Parent.MakeChild method without naming child
- no method like Object.fromString() => new object

Solution: keep the constructor details in one place, a factory

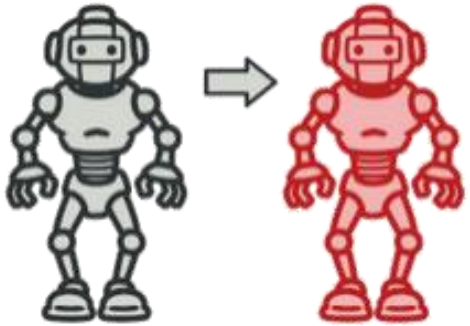
Factory method: Java

```
public Shape shapeFactory(ShapeId shapeId, String params)
  throws UnknownShapeException {
  switch (shapeId) {
    case ID_oval: return new RoundRect(params, 100);
    case ID_rect: return new RoundRect(params, 0);
    case ID_blob: return new Blob(params);
    //shift left: never use default!
  } //case
  throw new UnknownShapeException(shapeId, params);
}

public class Circle extends Shape {
  public Circle(String params) { .... }
}
```



Prototype

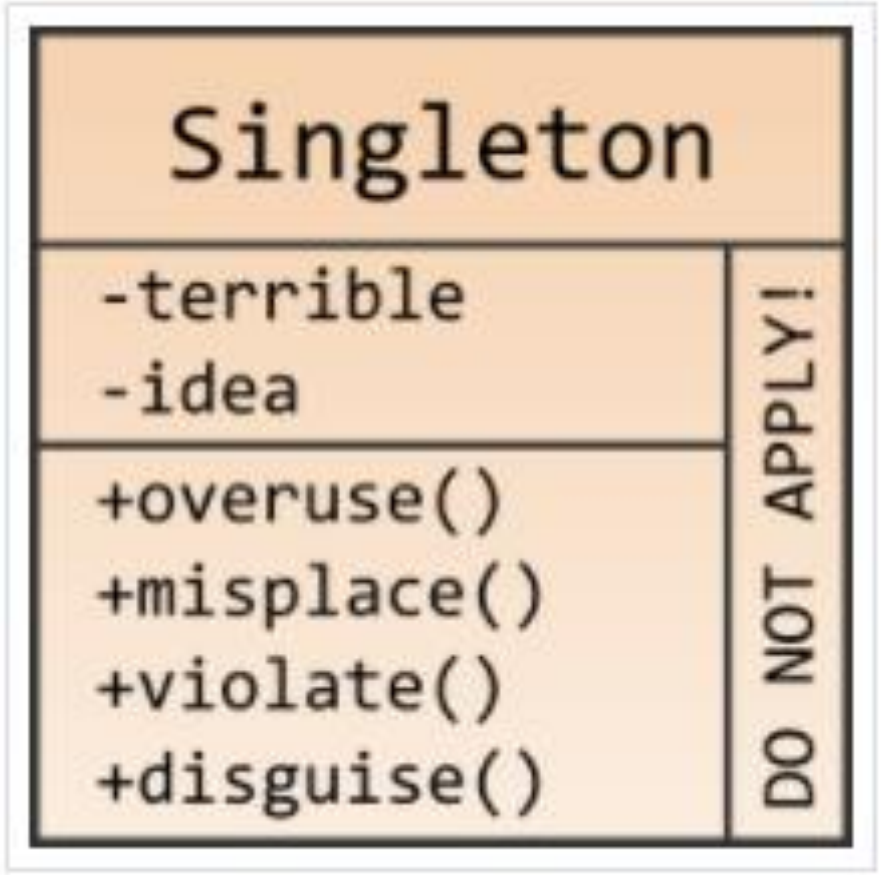
Java: single object hierarchy + virtual methods = polymorph

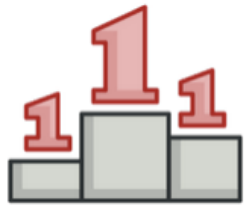


```
object.clone();
```

```
C++: //no virtual methods needed
//duck typing, restrict method signature
template <typename T>
concept Clonable = requires (const T t) {
    { {t.clone()} -> same_as<T&>; }
};
//clone factory method
auto clone(const Clonable auto& object) {
  return object.clone();
}
```

https://godbolt.org/z/E4TxMcqqE



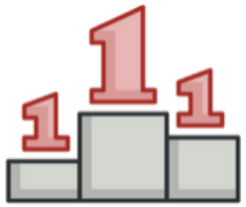


It allows your code to lie about its dependencies

- can't trust other programmers to not add mutable state to previously immutable singletons.

https://softwareengineering.stackexchange.com/questions/252/when-is-singleton-appropriate

Singleton



Very useful pattern

One pattern: CEO of the application

=> All decisions are made by the CEO
=> God object = anti-pattern

Global state is bad. Too temping

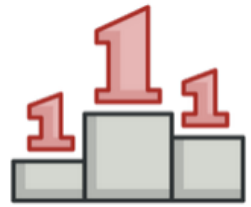
Hard to test code without including singleton as well

(fix by mocking singleton)

Used correctly for

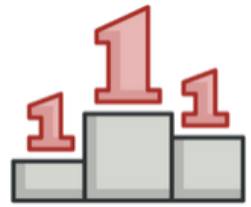
- logging
- database connection

Singleton: pattern or anti-pattern ?



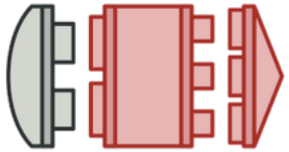
- ✓ memory usage, performance, consistency
- global shared resource accessible everywhere
- ☹ When GOF appeared, far too much
- ☹ If more than one actually needed
- ☹ If not correctly used, hidden dependencies & tight coupling

Singleton Java



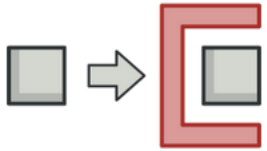

```
public /*final*/ class Singleton {
   private static bool isInitDone = false; //lazy init
   private static final Singleton instance = new Singleton();

   //private constructor clients cannot create more
   private Singleton(){}

   public static Singleton getInstance() {
      if (!isInitDone) { //do initialization }
      return instance;
   }
}
```

Structural patterns



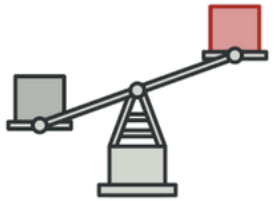
Adapter

Match objects with different interfaces



Proxy

Gatekeeper restricting access to class



Flyweight

Cache redundant data to save space

Composite

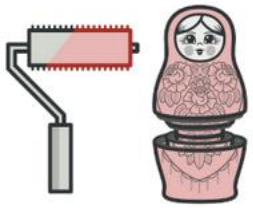
Split big object into parts

Facade

Simplified interface around object

Bridge

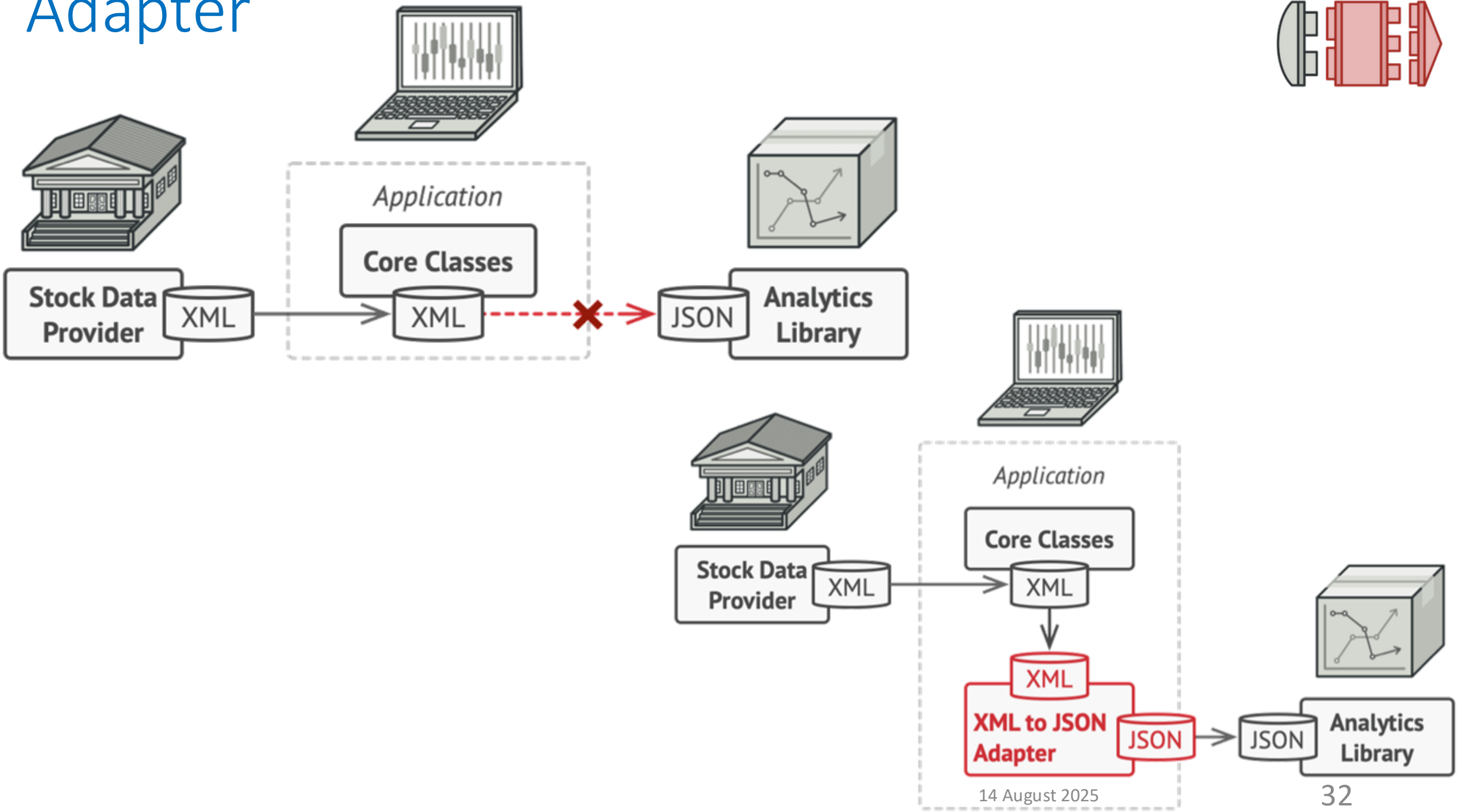
Prevent combinatorial explosion



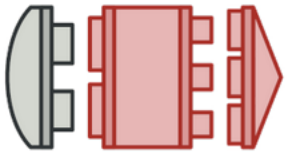
Decorator

Wrapper to add new functionality

Adapter


Application
Core Classes
Stock Data Provider
XML
XML
JSON
Analytics Library
Application
Core Classes
Stock Data Provider
XML
XML
XML
XML to JSON Adapter
JSON
JSON
Analytics Library

Adapter implementation

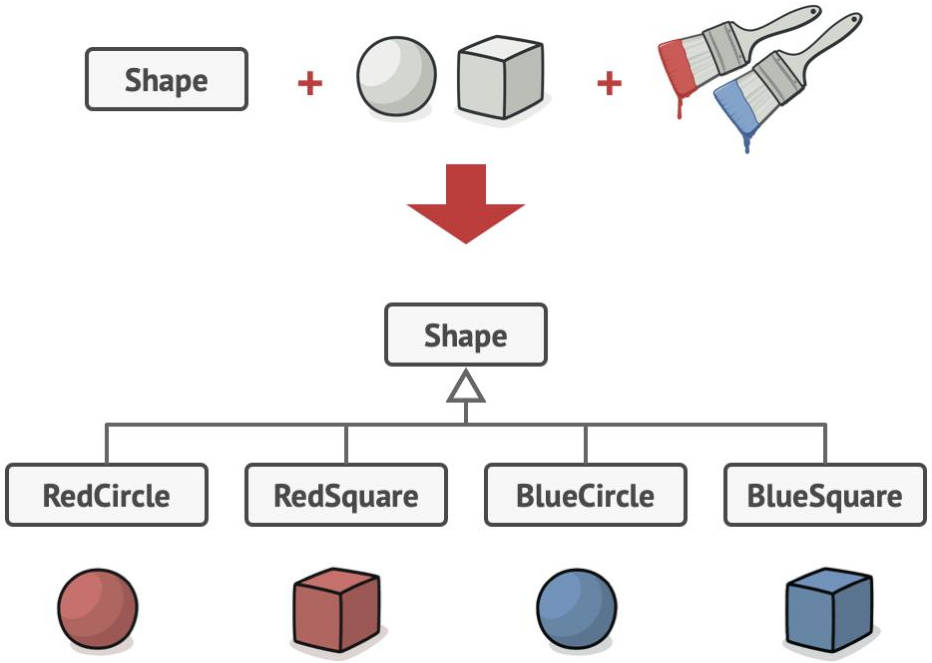


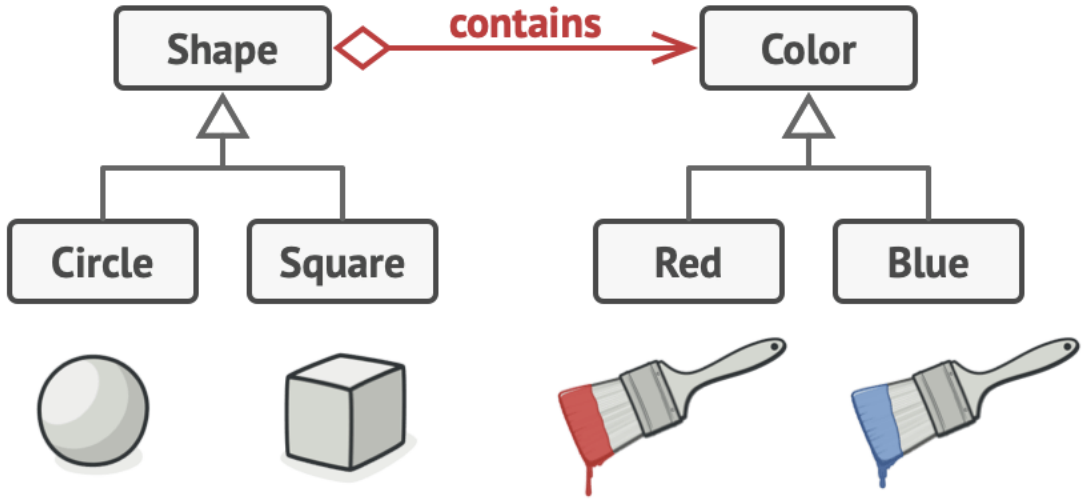
```
public class Adapter
implements IXML_er, IJSON_er {

   public void writeToDB(XML xdata) {
      JSON jdata = toJSON(xdata);
      json_db.write(jdata);
   }
}
```

Bridge

1. Use inheritance only for IS_A relationships
2. These relationships are rare
3. Very rare, outside of UI
4. Bridge: use composition instead of inheritance

   Allows you to add new dimensions to your model



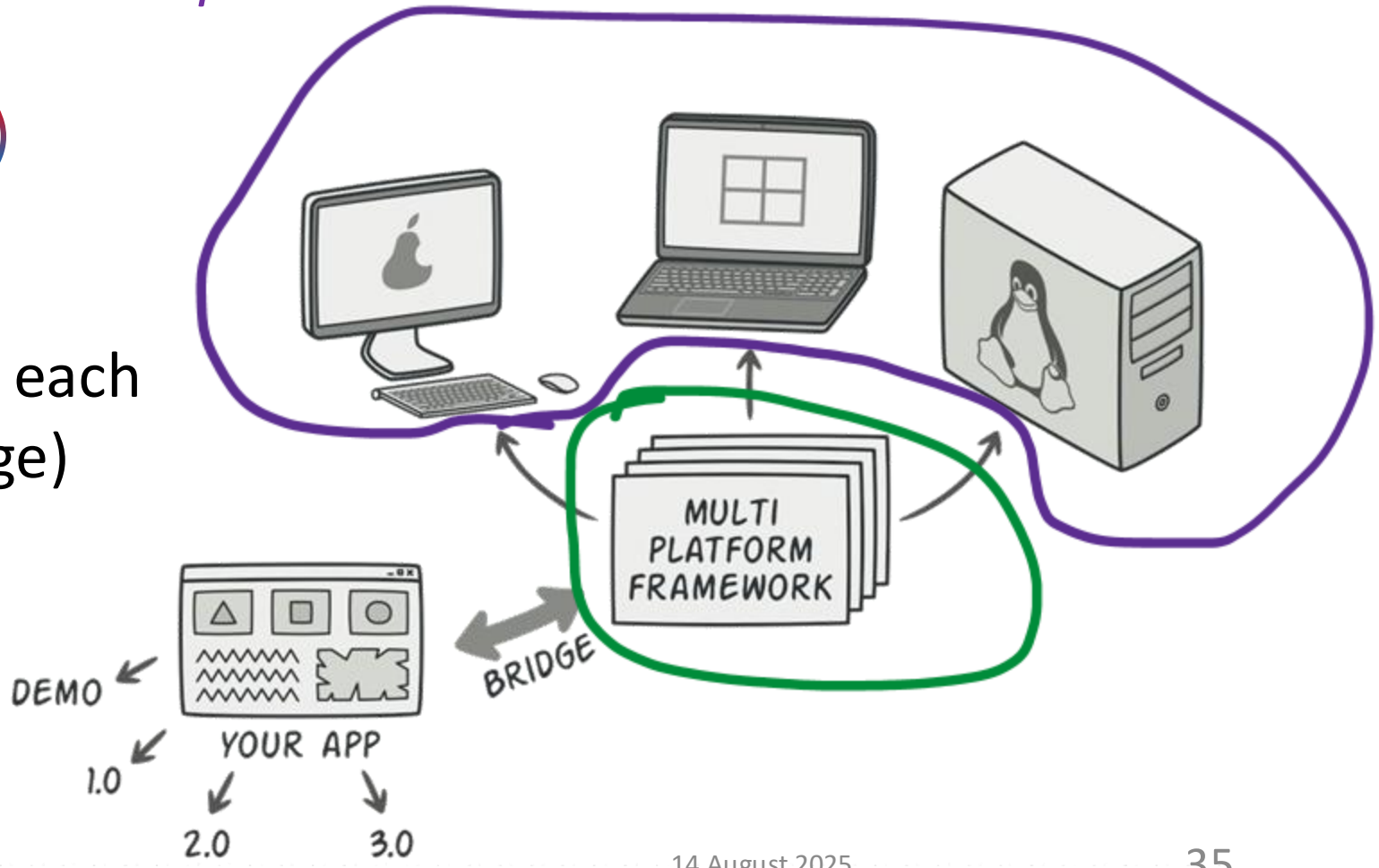


Bridge

GoF books says:
"bridge separates abstraction from implementation"

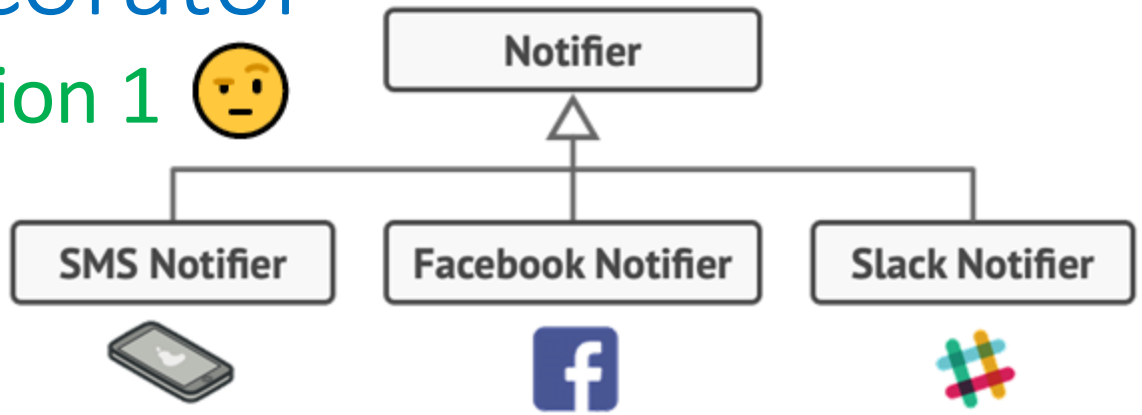
Imagine cross-platform (colors)
GUI (shapes ) app.

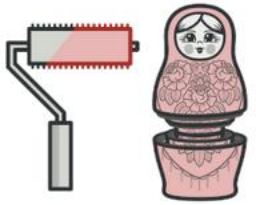
Instead of creating controls for each OS, create a display layer (bridge) that abstracts the OS' (impl.) display layer



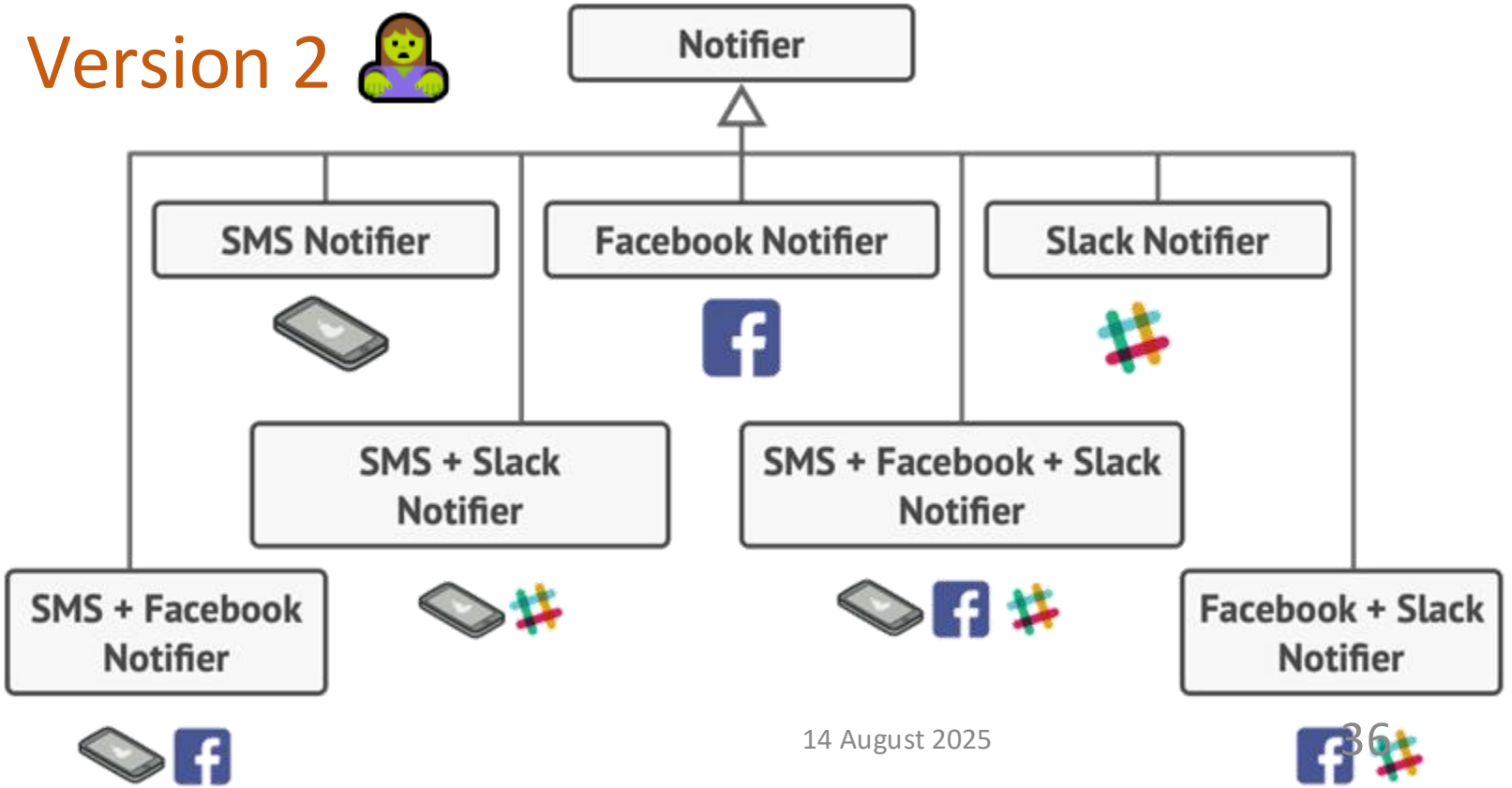
Decorator

Version 1

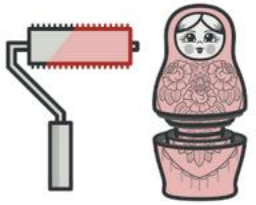




Version 2

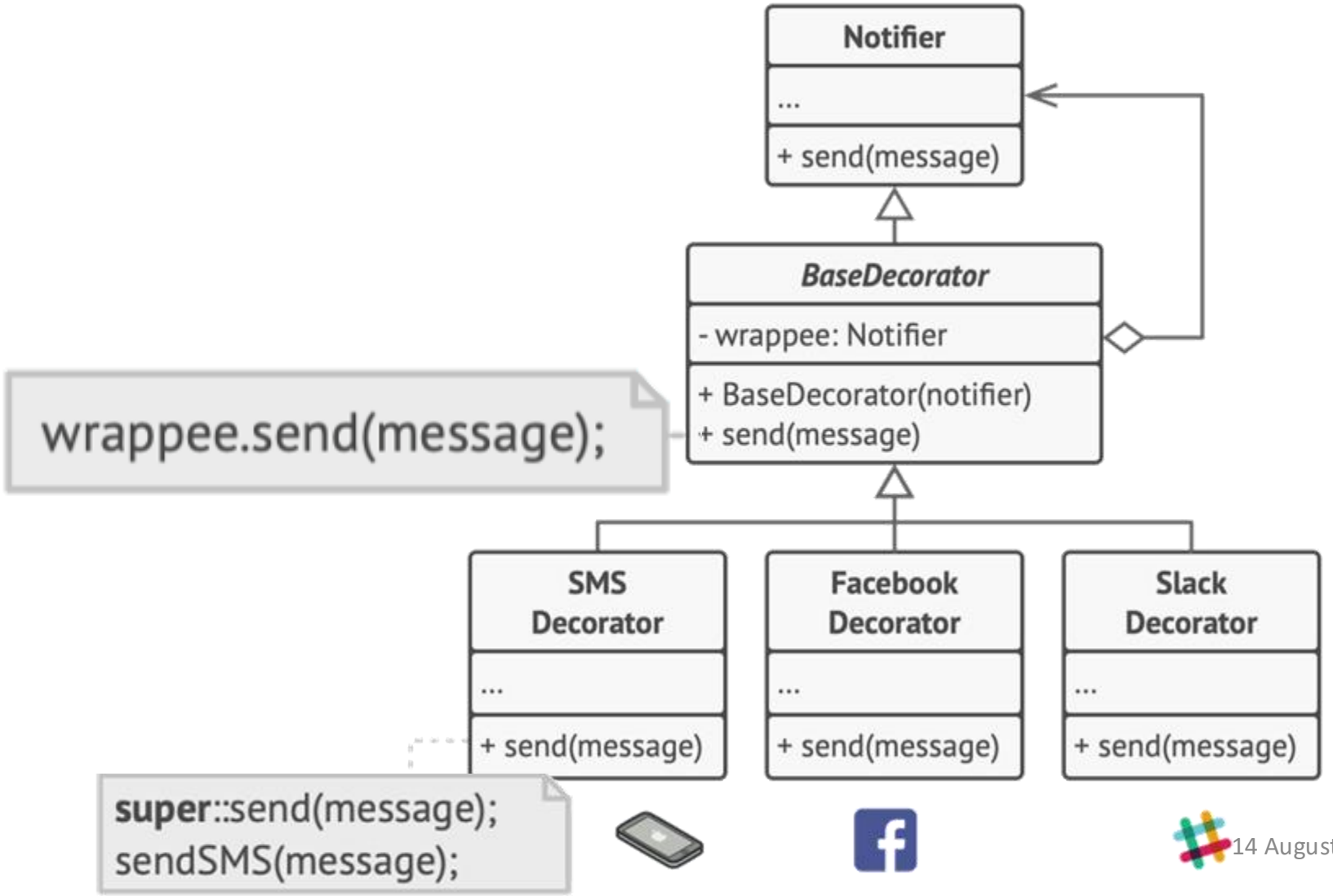


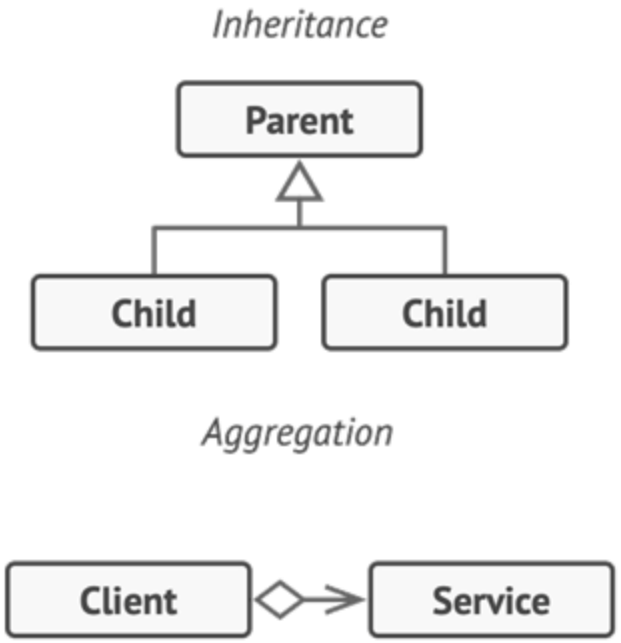
Decorator



Inheritance is rarely the best solution

To combine traits, use aggregation





Facade

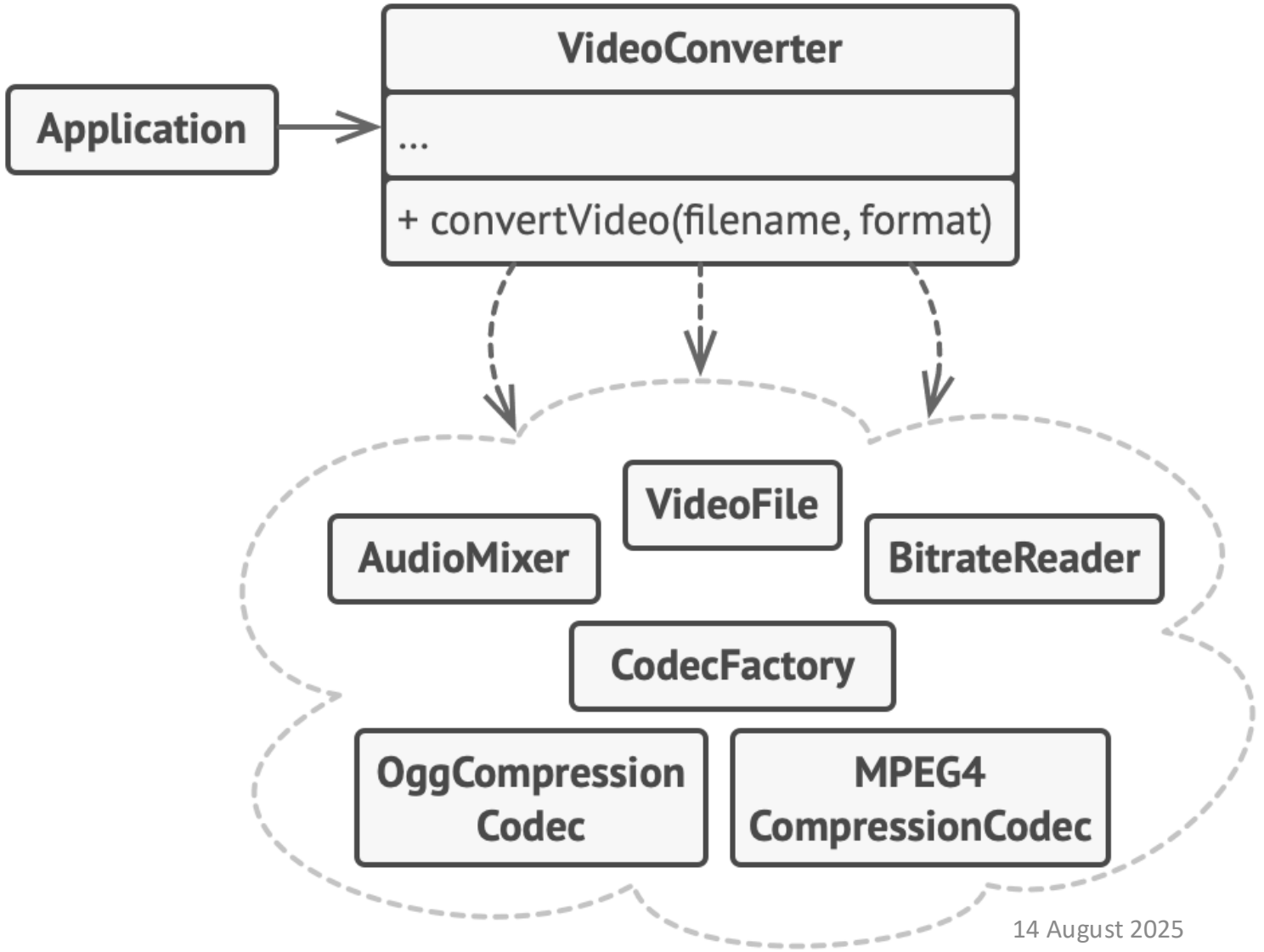
Imagine a webshop

As a client you order
a product from a vendor
to a location by shipping company
with VAT payable to SARS
paid online with creditcard via your bank

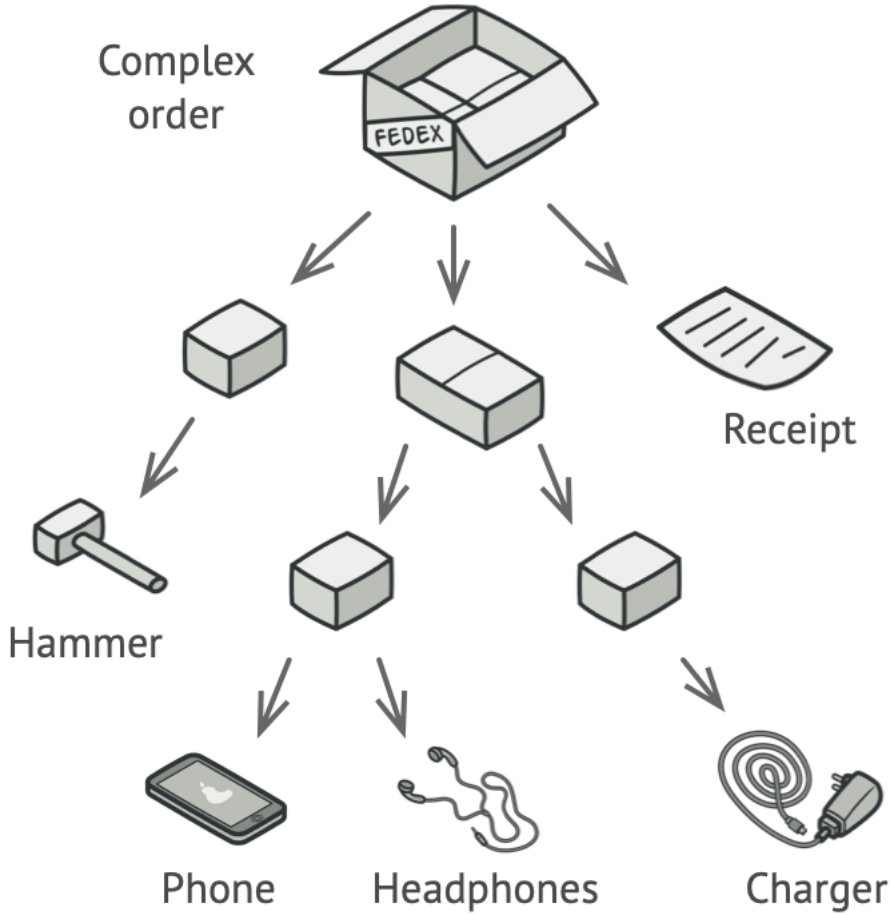
None of these entities and actors are your problem,
You just fill in a few fields on a form = facade

Facade

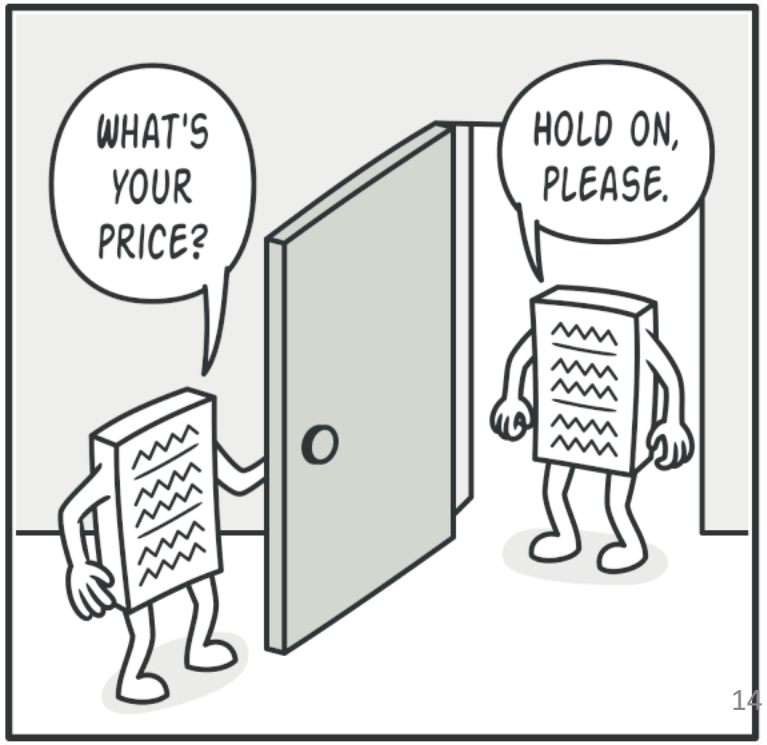
Simple interface around complex model

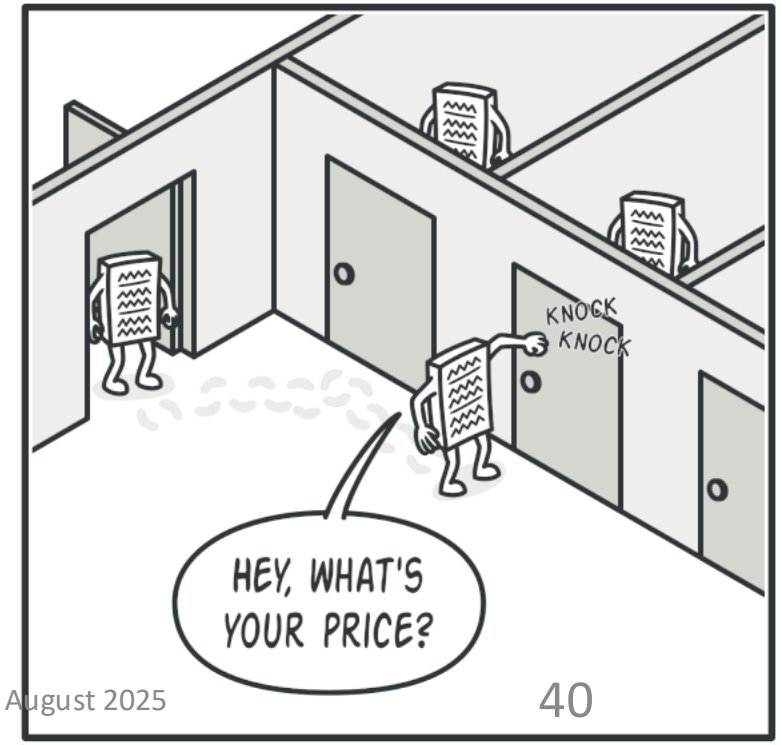


Composite

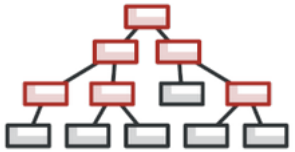


Iterate over all objects inside container





Composite



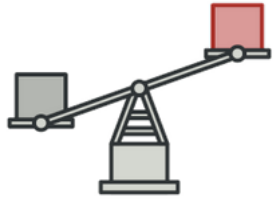
Composite class is a container

- list
- tree

That may contain other containers

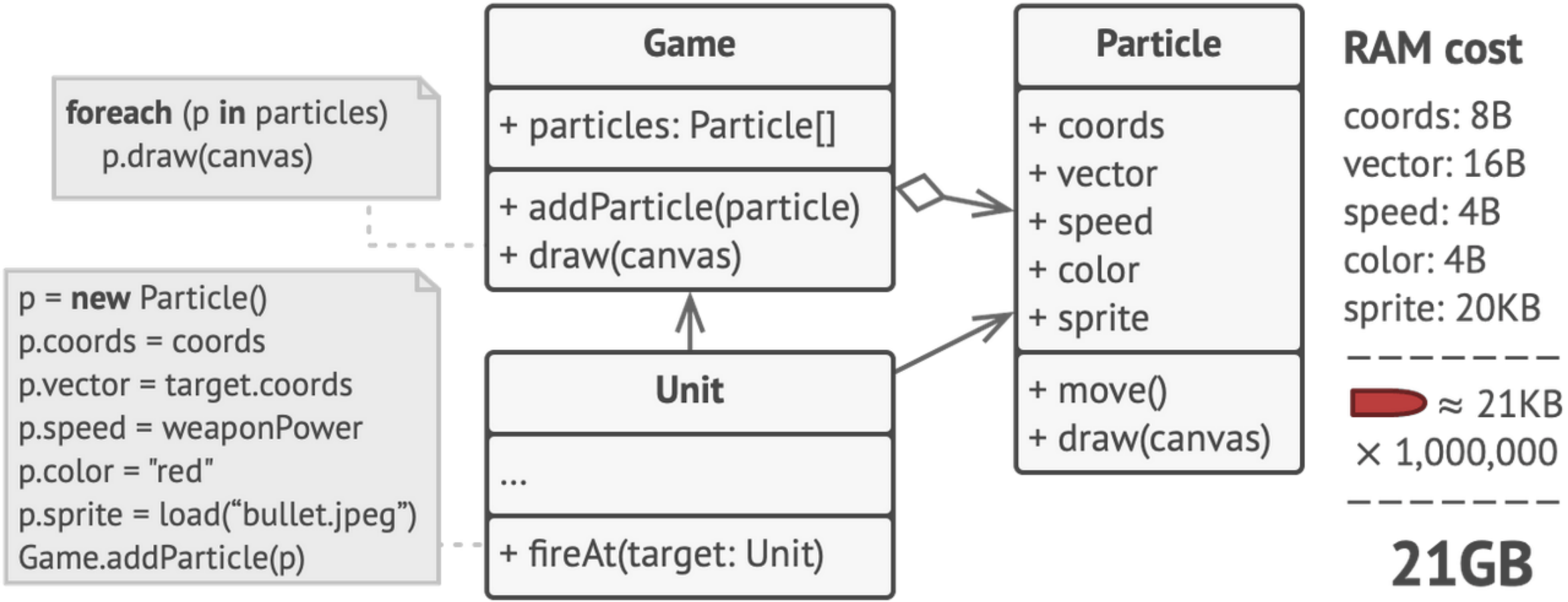
Iterate recursively over the container

Flyweight



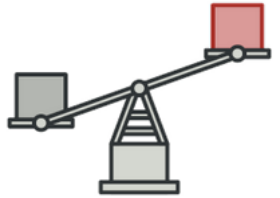
Problem: many objects contain duplicate info

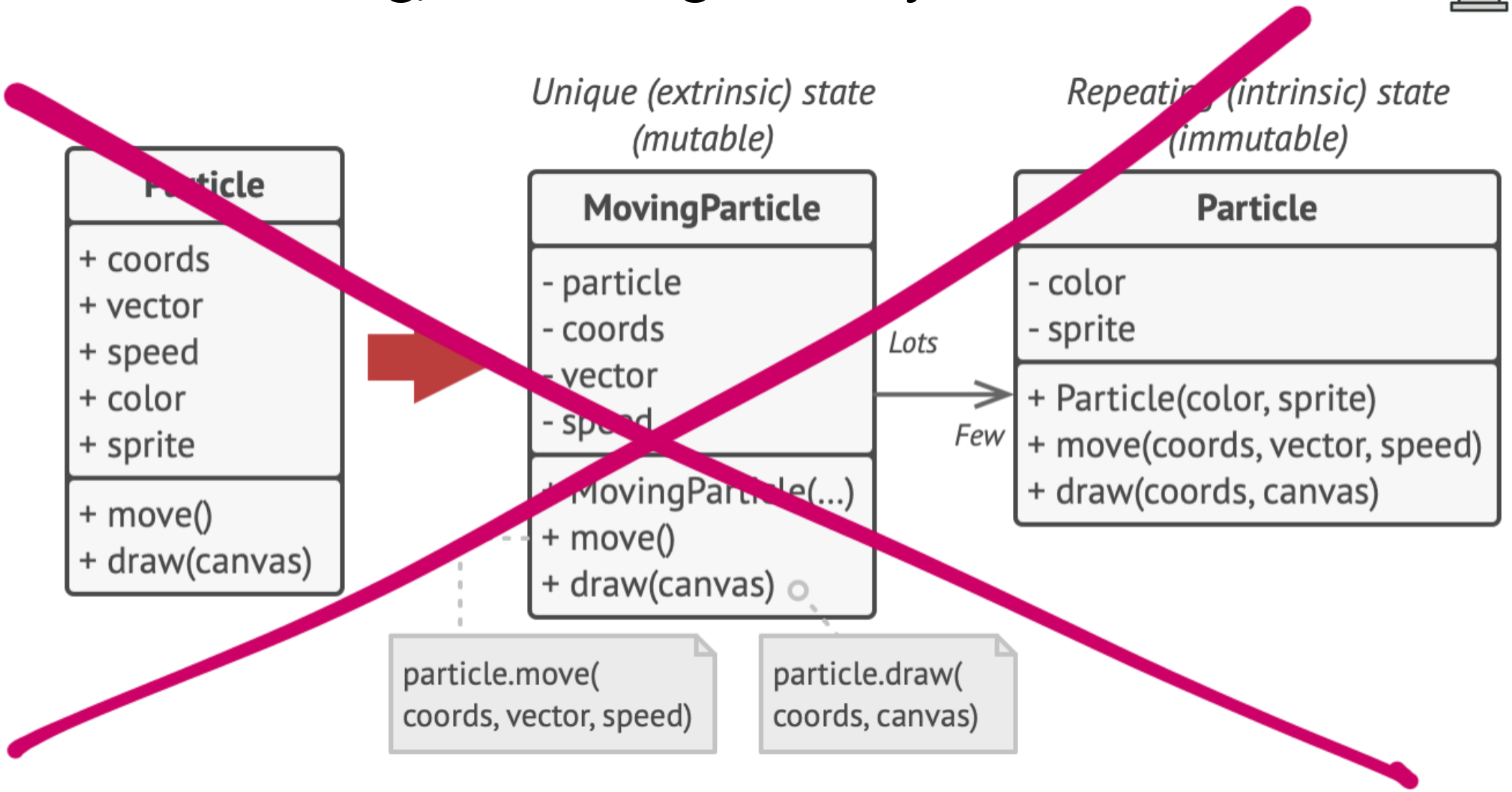
Consumes too much memory



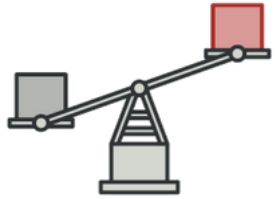
Flyweight

Solution: Something, something new object





~~Flyweight~~



No, games use

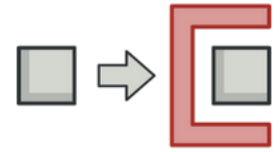
Data oriented design (DOD)

Different OOP paradigm, review DOD lecture

Why not flyweight?

- Iterating 1’000’000 objects too slow
- AoS solution aka pointer chasing
- Games use SoA => linear access

Proxy

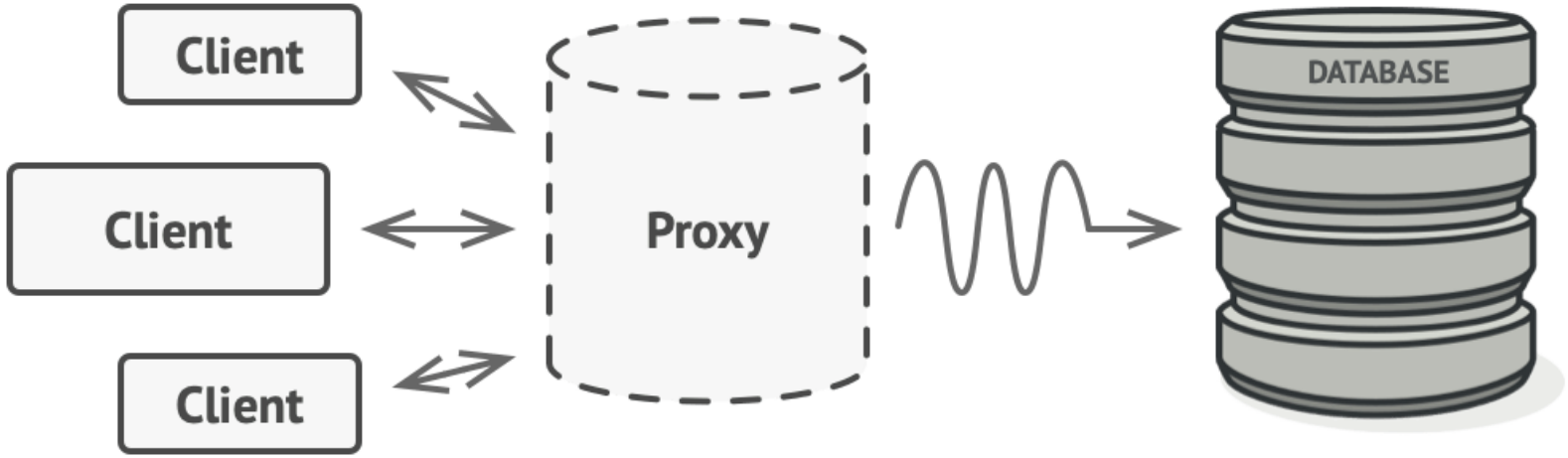


Webproxy: Nginx, Squid

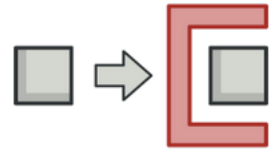
Database proxy

Disk proxy

Cache between program and slow resource



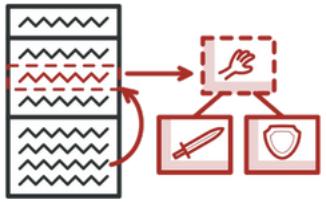
Proxy



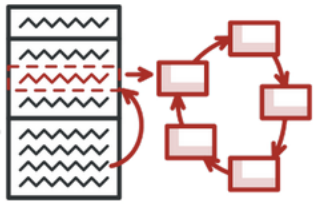
```
interface IDisk {
  List<String> getFileList(String path);
  Vector<byte> readData(String file);
}

class DiskProxy implements IDisk {
  List<Vector<byte>> Cache;
  Disk hd;
  Vector<byte> readData(String file) {
    if (file in Cache) { return Cache.getFile(file); }
    else {
      data = hd.getFile(file);
      Cache.Add(data);
      return data;
    }
```

Behavioral patterns

Command
Action objects



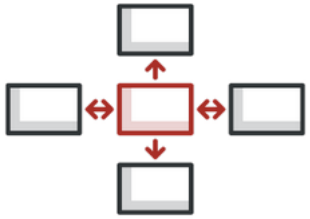
Strategy
Object per algorithm



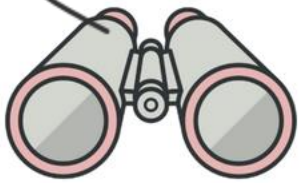
State
Finite state machine

Template
Algorithm blueprint

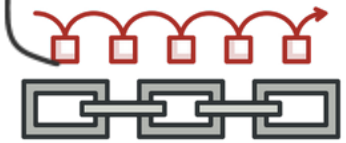
Memento
Snapshot/undo



Mediator
Reduce cross-talk



Observer
Subscribe to events



Chain of responsibility
Message passing

Visitor
Command++

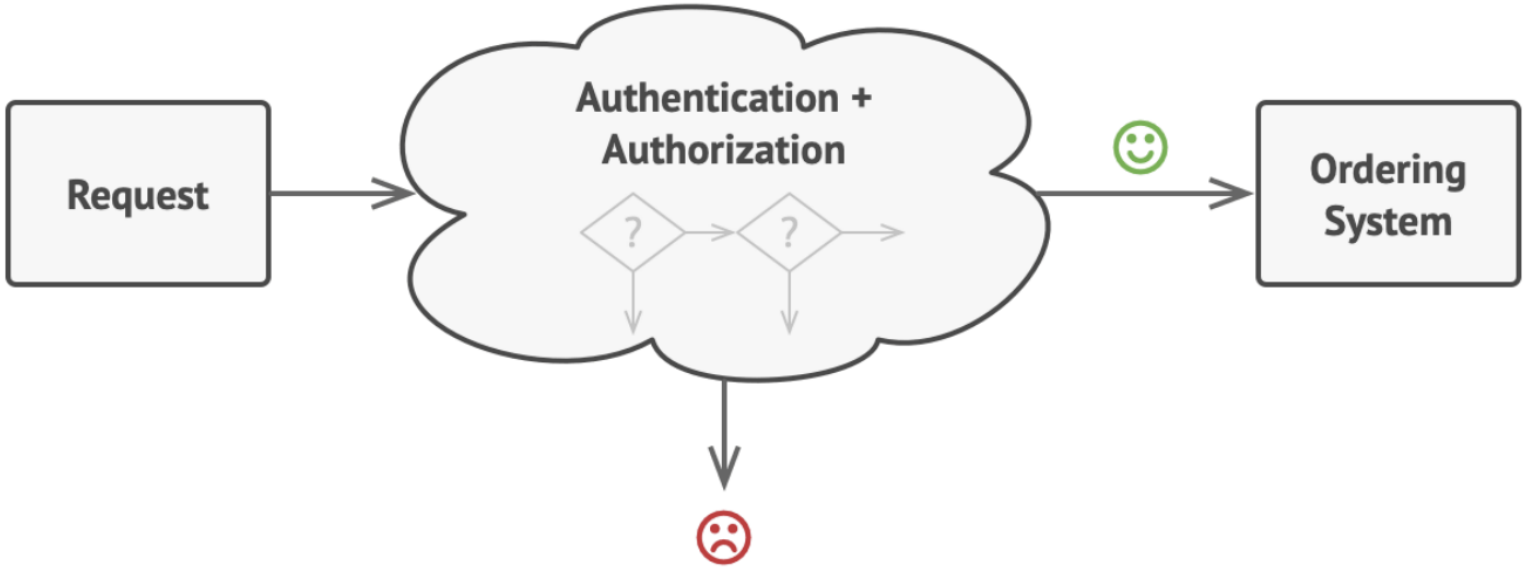
Iterator
Visit every element

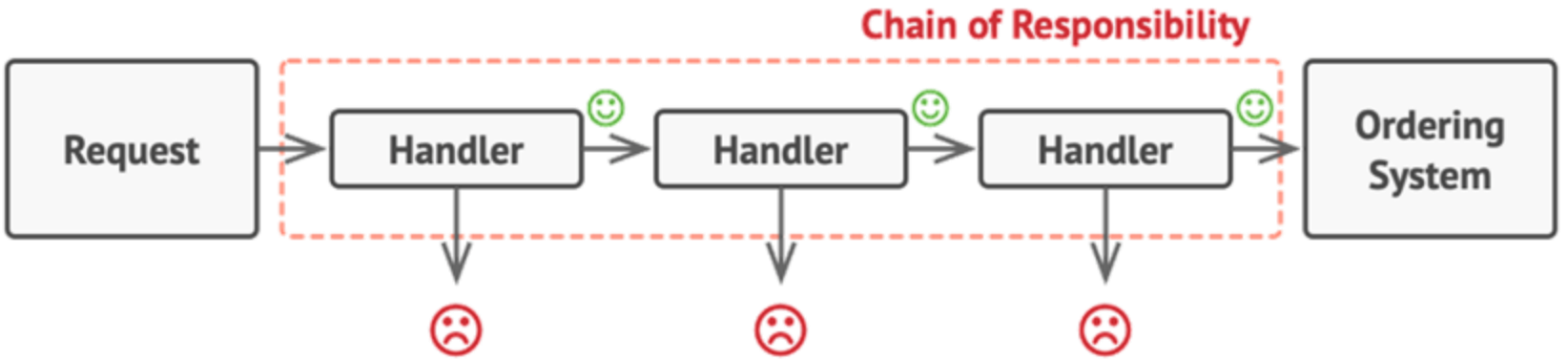
Chain of responsibility

Receive a request

Process it

Pass it on, or drop it



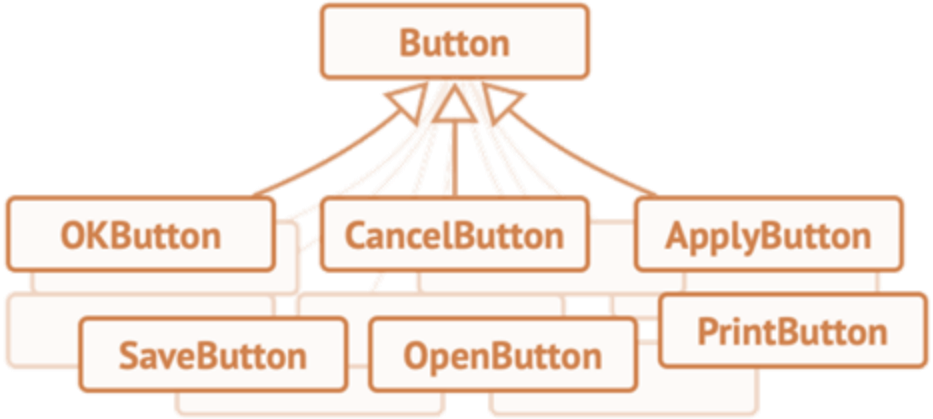


Command

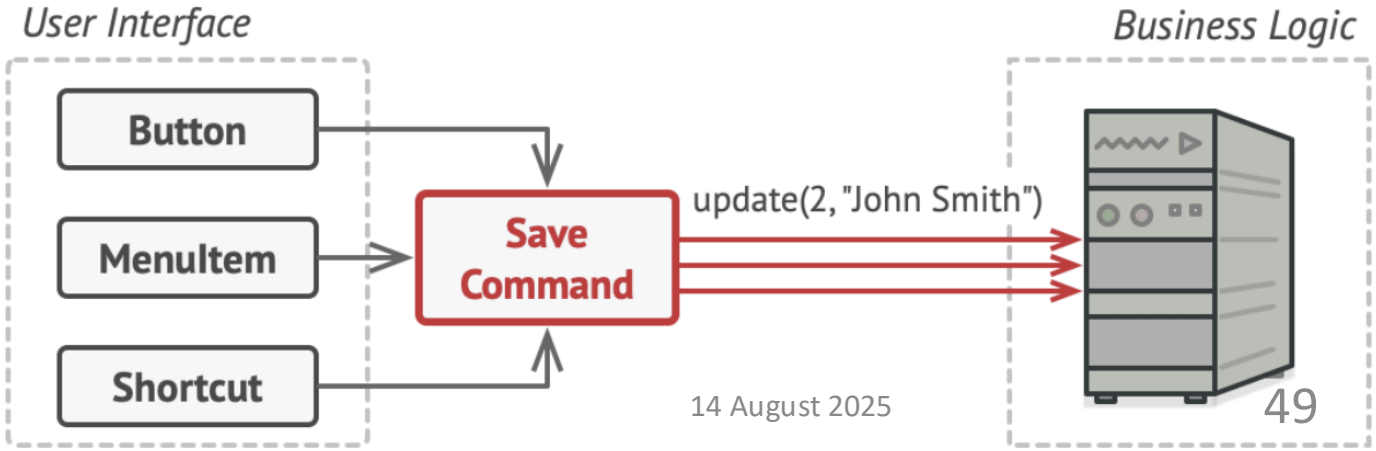
Separate commands from UI

OKButton IS_A Button? => Yikes

- Confusion of property/class
- Explosion of classes
- Business logic everywhere



- Duplicated code
- Tight coupling UI - logic



- No code duplication
- Loose coupling

Command

```
type Command: []{}-> bool; //lambda returning bool
const emptyCommand: Command = [&]{ return false; };

class UIControl: public InvisibleControl {
public
  action: Command = emptyCommand; //do nothing;
  bool execute() final {
    return action();
  }
}
//                                +--- [&] = capture context
form1.Actions.Add(ID_SaveAs, [&]{
  if (savedialog1.execute()) { //true = user pressed OK on SaveDialog
    auto filename = savedialog1.filename;
    editor1.SaveToFile(filename);
  });
```

Every modern language has lambdas, no longer needs OOP workaround

Chain of responsibility: WinAPI messages

Receive a request

Process it

Pass it on, or drop it

Form1
└ Panel1
  └ Button1

Option 1: every link in chain knows next link

```
Msg 1: Button1.OnMouseOver => draw glow effect => drop msg
Msg 2: Panel1.OnMouseOver => do nothing => call owner.OnMouseOver
          Form1.OnMouseOver => Flash caption => no owner => drop msg
```

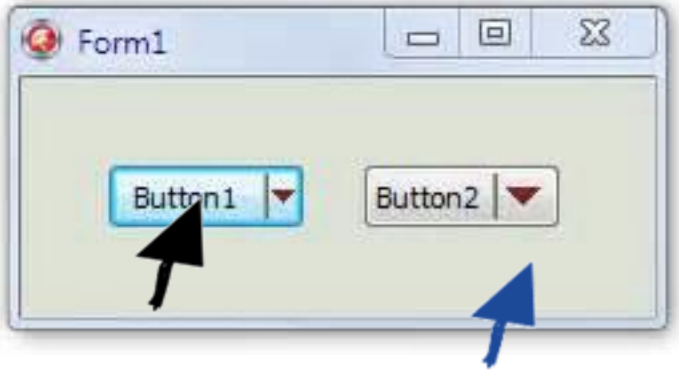
Chain of responsibility: WinAPI messages

Receive a request

Process it

Pass it on, or drop it

Form1
└ Panel1
  └ Button1



Option 2: links do not know each other

WinAPI iterates (iterator pattern) over controls topmost -> bottom most

Traverse tree: do hit test, if hit,
then traverse child until no more hit = topmost control

Traverse hit controls in reverse order

```
Msg 1: Button1.OnMouseOver => draw glow effect => return false;
Msg 2: Panel1.OnMouseOver => do nothing => return true
       Form1.OnMouseOver => Flash caption => return false;
```

Iterator

Walk through every item in a collection

Collection: first + next + current methods

Iterator: `for (a in Collection) { a.dostuff() }`

=> for in loop runs the following pseudocode

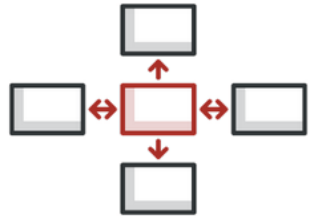
```
c.First();                  //current = c[-1]
while (c.next()) {  //current = c[0,1,2,...]
   a = c.current(); a.dostuff();
}
```

Design custom collections with `first, next, current` methods

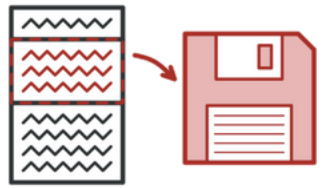
Every modern language has for in loops.
No longer needs an OOP workaround

Mediator

Fancy version of command pattern



Memento

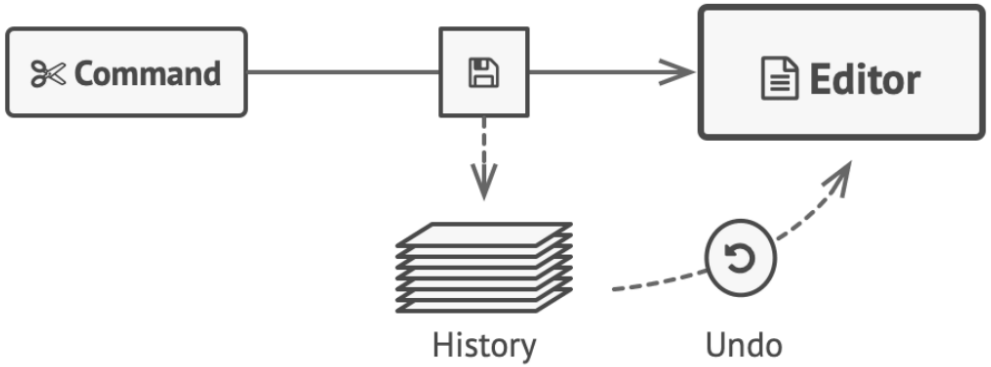


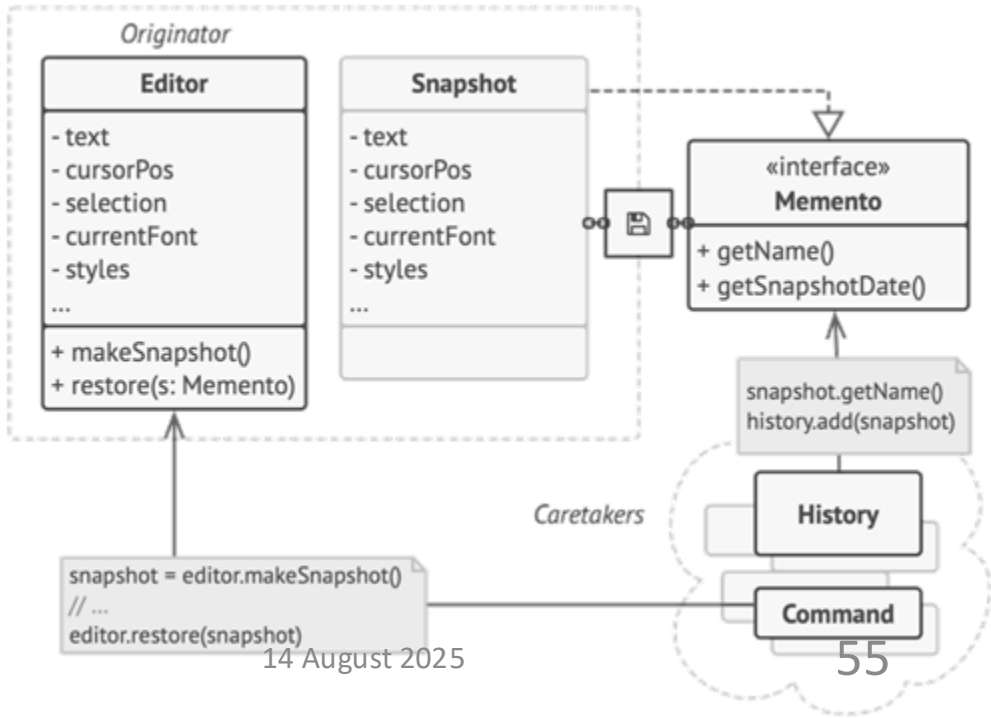
Program needs to save state from ≥ 1 objects

- Must not access object's private details
- Cloning objects not an option
  - Unsafe due to links with other objects

Solution

- Object creates '`snapshot`' to store state
- `Memento` adds metadata (date, command)
- `Caretaker` stores snapshots





Memento

```
class Editor implements IUndo {
  String text;   //gets saved
  Window owner;  //not saved
  EditorSnapshot makeSnapshot(); //IUndo
  void restoreSnapshot(EditorSnapshot restorePoint); //IUndo
}

class EditorSnapshot implements IMemento immutable {
  String text;
}

interface IMemento immutable {
  property int id;
  property IUndo source;
}

interface IMementoList immutable {  //if multiple objects must be restored
  property int id;
  property Datetime timestamp;
  property CommandId reason;
  property List<IMemento> sources;
  void restore() { for (s in sources) { s.source.restoreSnapshot(s); }
}

class Caretaker { //manages snapshots
  List<IMementoList> snapshots;
  void add(IMementoList item);
  void restore(int id) {
    restorepoint = getById(snapshots, id);
    restorepoint.restore();
```

Memento

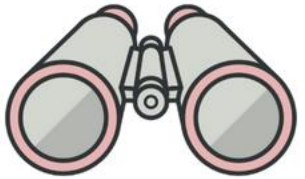
Useful for

- Undo (revert back to previous savepoint)
- Transactions (commit/rollback)
- Save to file

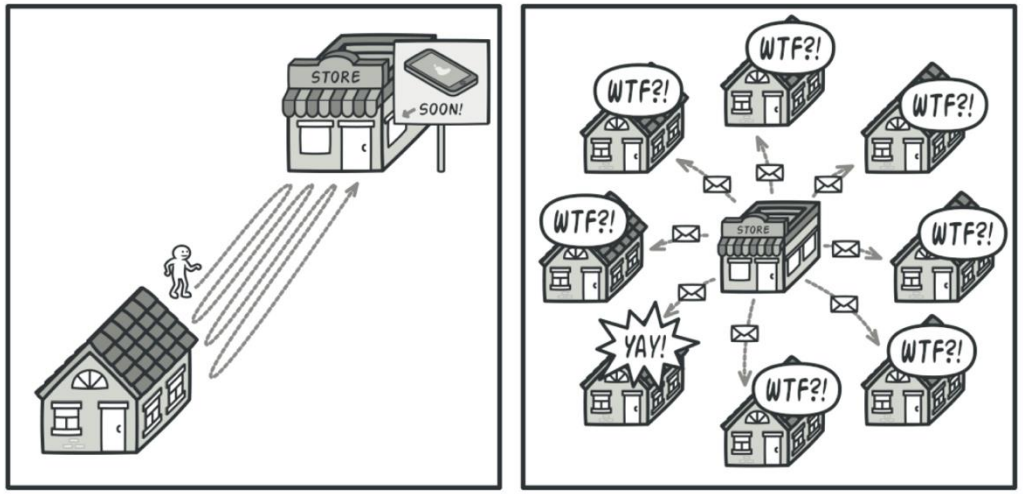
Pro	Con
Can save state, without having to know internal details of objects	Can consume a lot of RAM if too many snapshots
Caretaker tracks snapshots	May need to track original object, if these are not permanent, to destroy obsolete snapshots
	Needs immutable objects

Observer

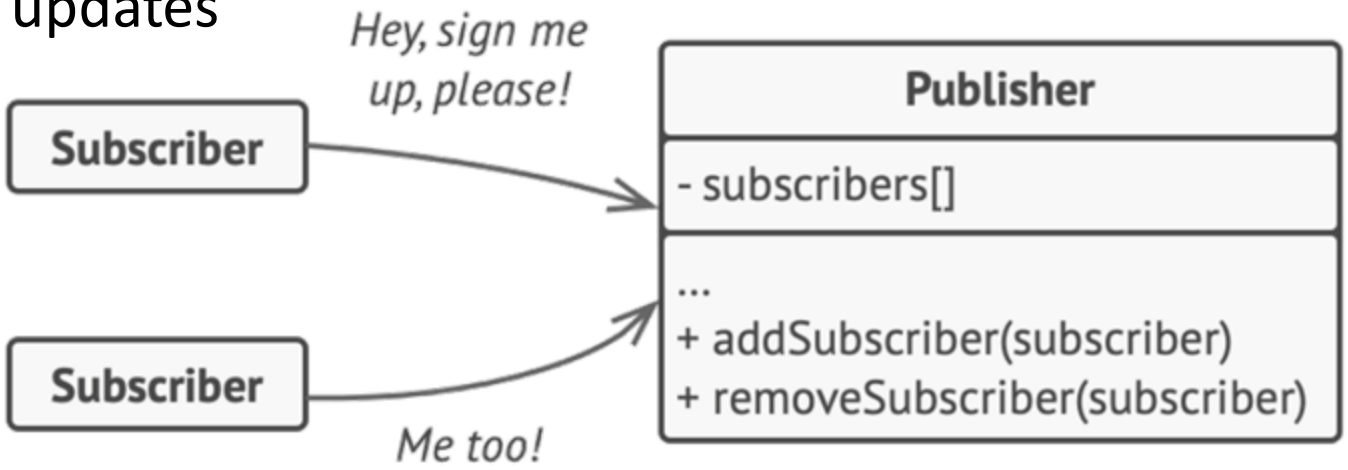
Update subscribers on the state of an object



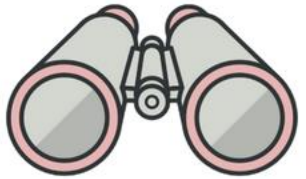
Problem: waiting for iphone 35 to arrive
Client could check in every day with the store
Store can email all possible clients on arrival



Solution: interested objects subscribe to updates



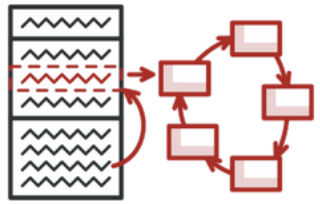
Observer



```
interface IObserver {
  void update(msg);
}

interface IPublisher {
  void subscribe(IObserver client);
  void unsubscribe(IObserver client);
}

class Publisher implements IPublisher {
  List<IObserver> clients;
  private void update(msg) {
    for (c in clients) { c.update(msg); }
  }
```

State

Finite state machine



Problem:

- complex case statement controls state

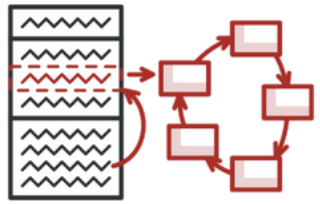
```
class Document is
    field state: string
    // ...
    method publish() is
        switch (state)
            "draft":
                state = "moderation"
                break
            "moderation":
                if (currentUser.role == "admin")
                    state = "published"
                break
            "published":
                // Do nothing.
                break
    // ...
```

Solution:

- IState interface
- One class per state
- Master object switches state
  - Calls state object to do state work

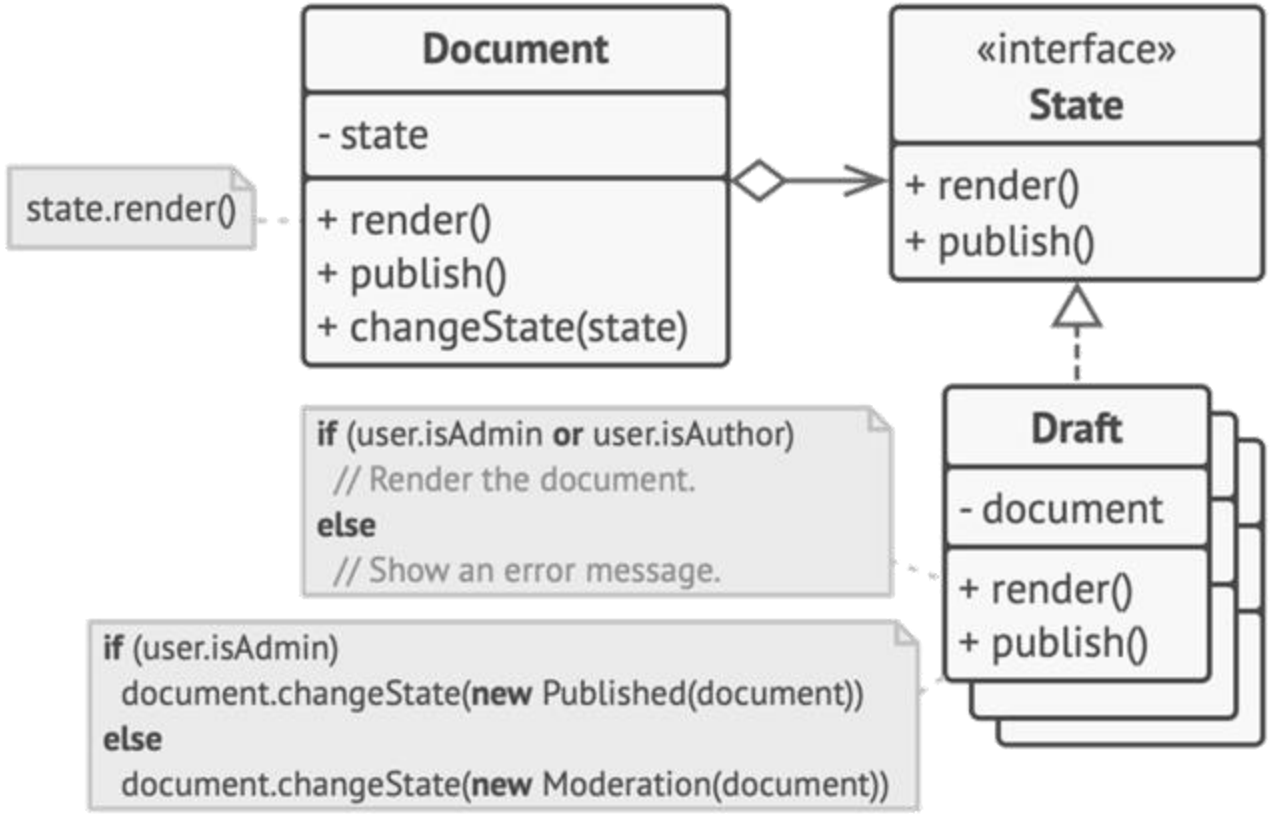
State

Finite state machine

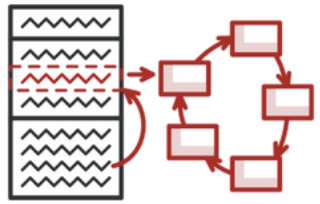


Solution:

- IState interface
- One class per state
- Master object switches state
  - Calls state object to do state work



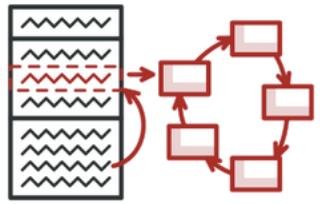
State: music player



```
interface IPlayState {
  property AudioPlayer player;
  void pressPlay(); //play/pause
}
class PlayState implements IPlayState {
  void pressPlay() { //from play -> pause
    player.setState(new PauseState());
    player.pauseMusic();
  }
}
class PauseState implements IPlayState {
  void pressPlay() {//from pause -> play
    player.setState(new PlayState()); }
    player.playMusic();
  }
}
class MusicPlayer {
  IPlayState state = new PauseState();
  Button playButton(state.pressPlay);
}
```

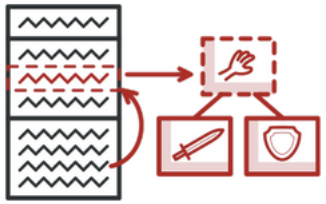
State: music player

States with shared code can inherit from or contain shared classes.



Pro	Con
Single responsibility: One state per class	Overkill if only few simple states
Open/closed: can change state details without having to change player class	
Eliminate complex state machine details (Master object can only be in a single state)	

Strategy



More complex version of State

- out of scope -

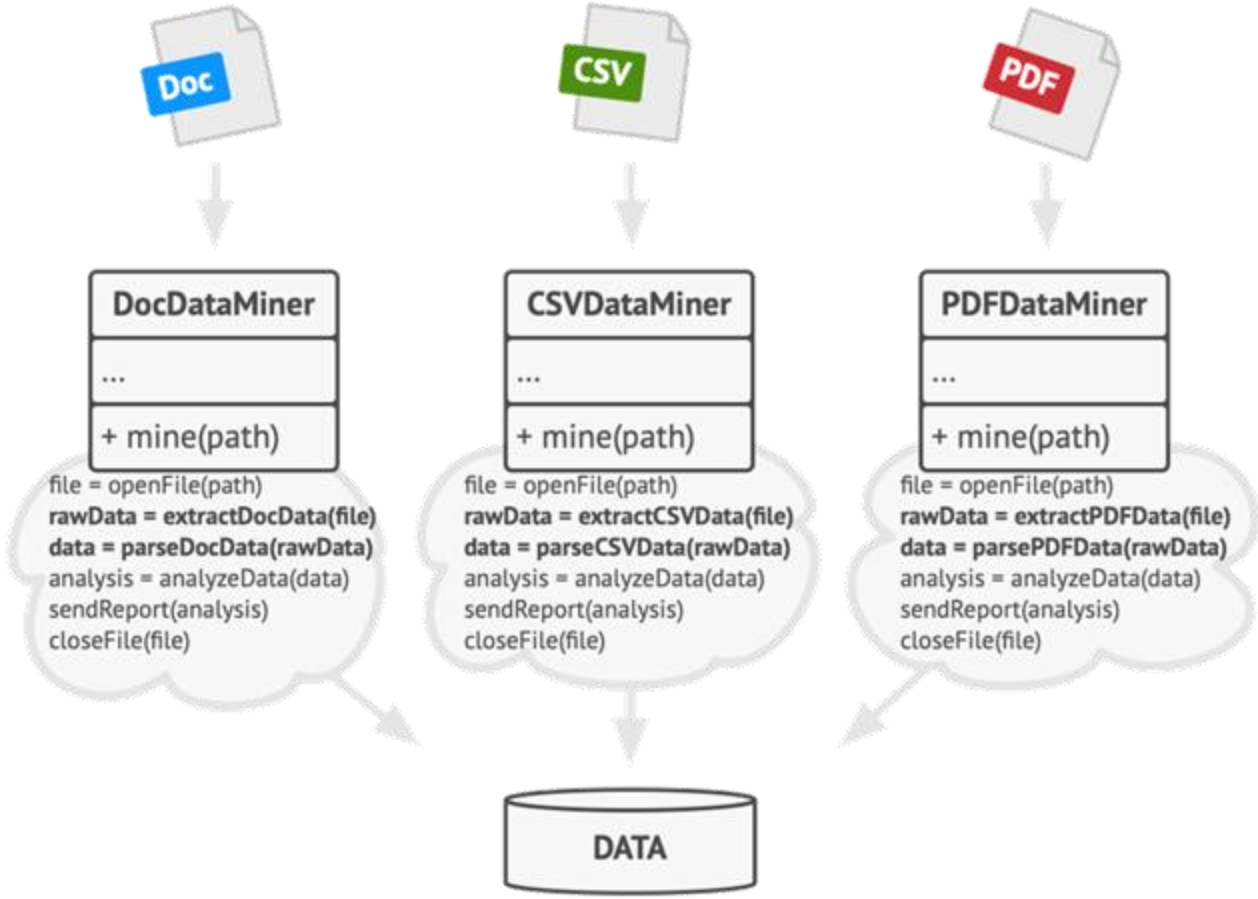
Template

Define the skeleton of an algorithm

But not the detailed steps

Problem:

- Different versions of algorithm
- Repeated code in each version
  - Violates DRY principle



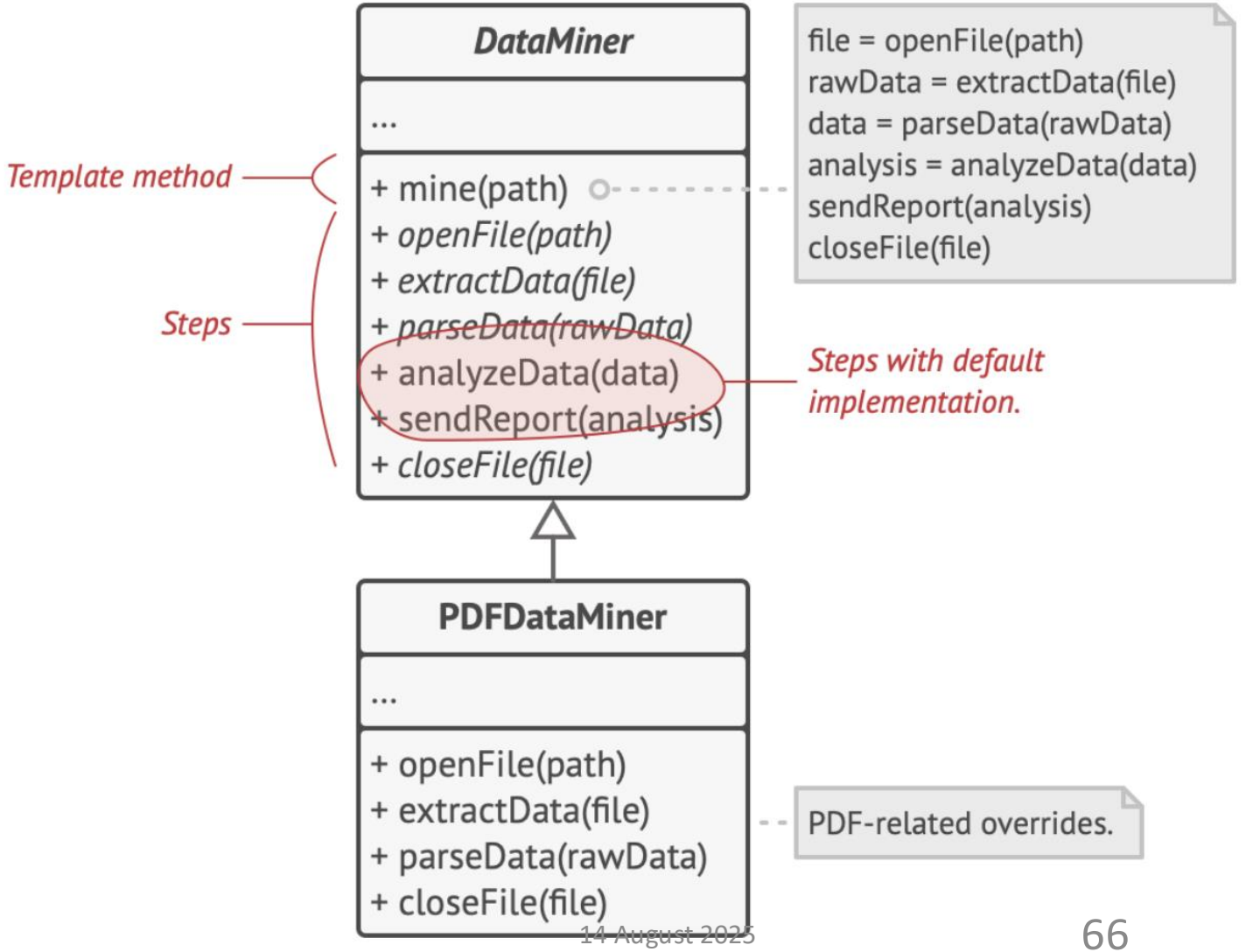
Template

Define the skeleton of an algorithm

But not the detailed steps

Solution:

- Put steps in class A
- Put details in class B



Template

```
class BrewMaster {
  void boilWater() { print("boil water"); }
  void addPlants() { /*do nothing*/ }
  void removePlants() { /*do nothing*/ }
  void addCondiments() { /*do nothing*/ }
}
class BrewTea extends BrewMaster {
  void addPlants() { hang tea bag }
  void removePlants() { remove tea bag }
  void addCondiments() { just sugar, no milk }
}
class BrewCoffee extends BrewMaster {
  void addPlants() { add coffee }
  //no sugar, no milk (duh)
}
```

Template

Define the skeleton of an algorithm

But not the detailed steps

Solution:

- Put steps in class A
- Put details in class B

Pro	Con
You can let clients override only certain parts of a large algorithm, making them less affected by changes that happen to other parts of the algorithm.	Overkill if only few simple states
Open/closed: can change state details without having to change player class	
Eliminate complex state machine details (Master object can only be in a single state)	

Visitor

More complex version of command

- out of scope -