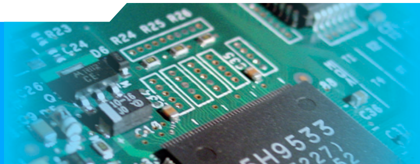
Basic design patterns

Noël PLOUZEAU

IRISA/ISTIC



Concept of design pattern

- We have already seen a form of design patterns
  - canned wisdom
  - harvested from designs and code made by thousands of developers in the last 30 years

The real design patterns

- Traditionally the term is used for
  - a reusable solution
  - to a frequently occurring problem
  - in a given context

Origin

- Proposed by an architect (Ch. Alexander) around 1979
- Applied to software design starting from 1987
- Gained popularity with this book:
  - Gamma, Erich; Richard Helm, Ralph Johnson, and John Vlissides (1995). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley. ISBN 0-201-63361-2.

A form of template

- Main features to describe a DP
  - name, intent (what is the goal), motivation (problem), applicability (where)
  - participants, collaboration, structure
  - implementation example
  - consequences (side effects, dependencies)

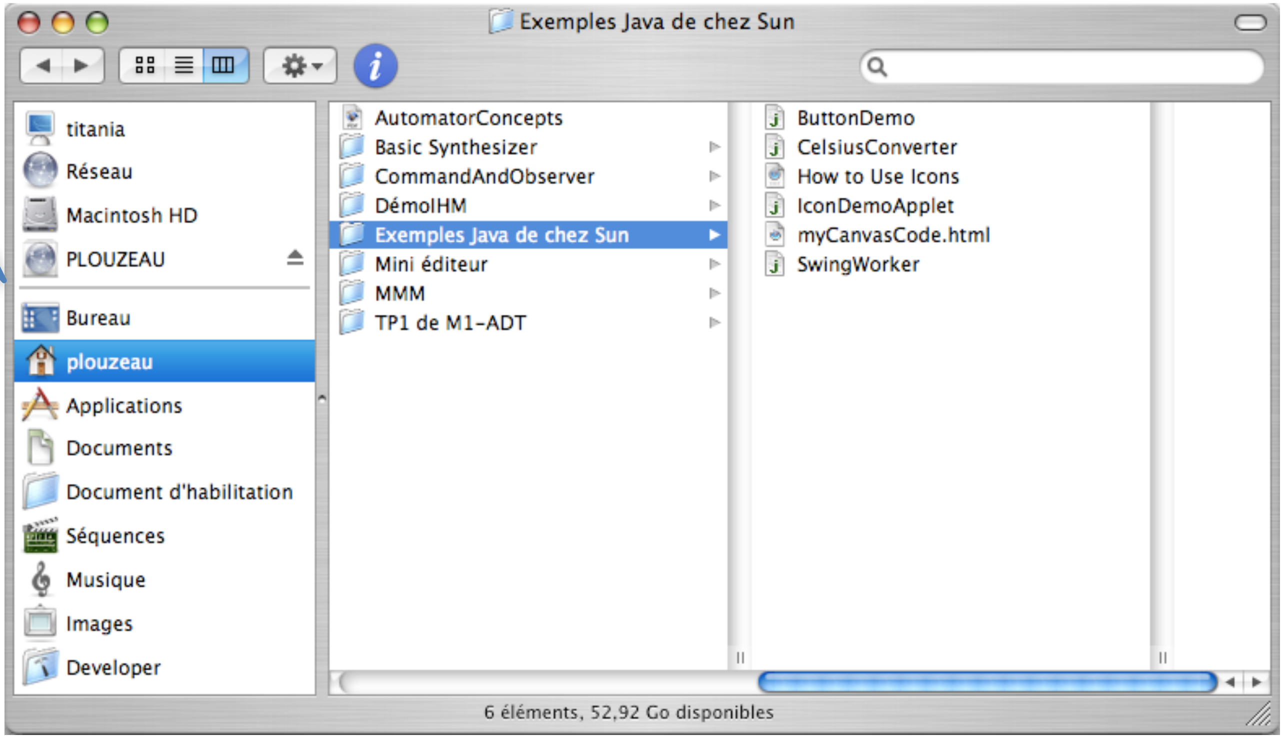
A first example: Observer

- Motivation
  - some objects must keep their state consistent with the state of other objects (the subjects)
  - when the subjects change, the dependent object must update their state

Example

View

File system



Files, directories and views must be kept in sync

Another example

Data must be sent
to the clients
when it changes

Twitter

- Source of data
  - Each user's history of tweets, on the user's personal computer
- Sink of data
  - The computer of users that follow a set of users
- Fast and scalable push technique

General classes of solution

- Data source, data sink
- Who has the initiative?
    - Data source -> “push”
    - Data sink -> “pull”
    - Another object -> “pull/push”
- Observer is based on “push”

The Observer PC

- Intent: to propagate data changes of an object to other objects
- Motivation: some objects must keep their state in sync with others' states
- Participants: mutator, subject, observer, concrete subject, concrete observer

The CRC template

- To describe a pattern one can use the CRC (class, responsibilities, collaborations) template
  - class: these are the types (interfaces, implementation classes)
  - responsibilities: what each type must ensure
  - collaborations: an implementation to guarantee that the responsibilities are fullfilled

Classes (types)

- mutator, subject and observer define a protocol
  - rules of interaction
  - interfaces between participants
  - one participant often plays the role of "the outside of the PC"

Details of responsibilities

- subject
  - manages observers' subscription (interface plus storage of subscriptions)
- observer
  - provides an interface to receive update notification from subjects

Details of responsibilities (cont'd)

- mutator
  - the outside, the reason why subject states change
- mutator, subject and observer are declarations, not implementations (therefore Java/UML interfaces)

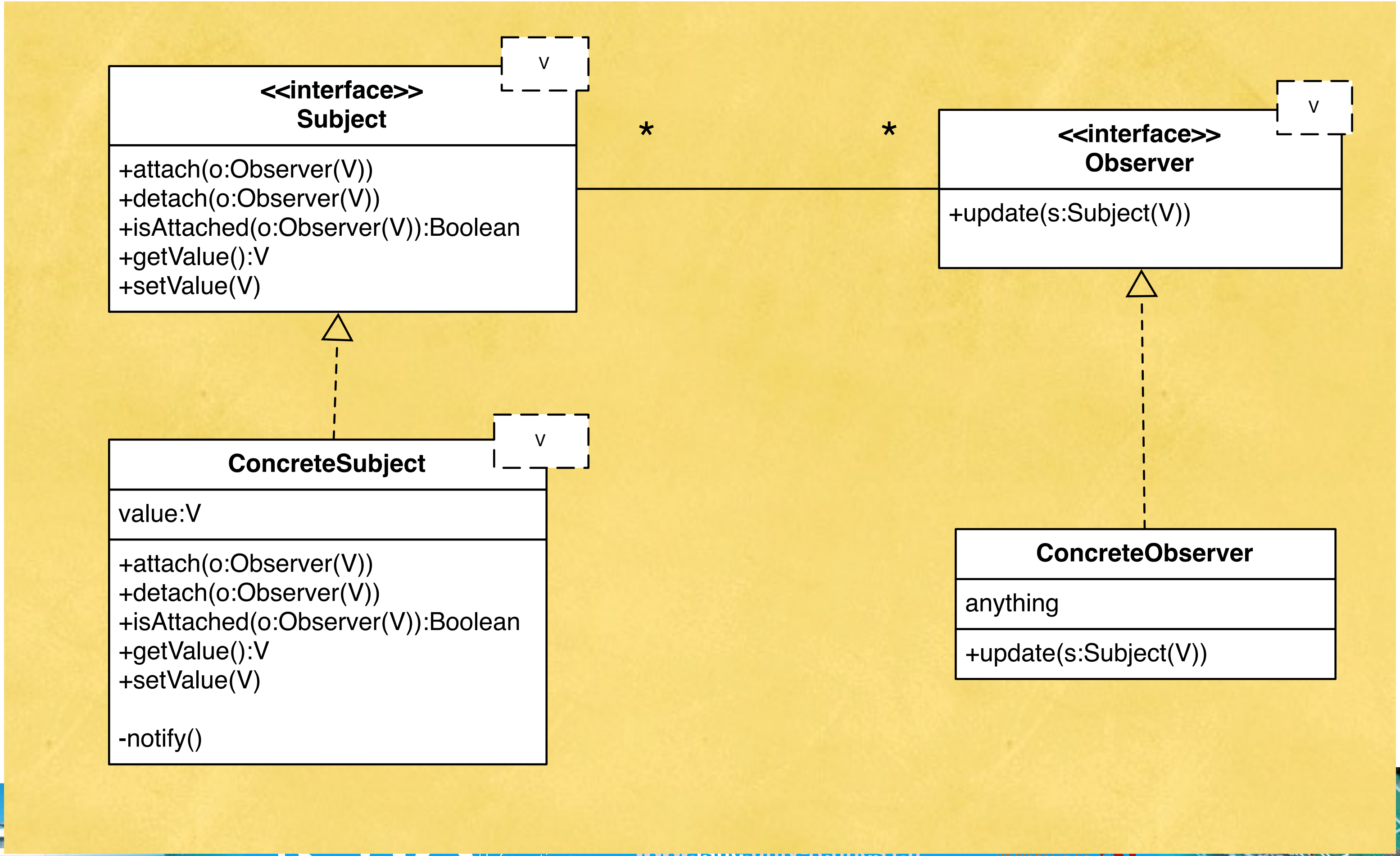
Details of responsibilities (cont’d)

- concrete subject
  - stores a data state, provides R/U operations for it
  - must notify observers when its state changes

Details of responsibilities (cont'd)

- concrete observer
  - has a state that depends on the subject's state
  - provides a method for the operation that the subject can call to notify changes

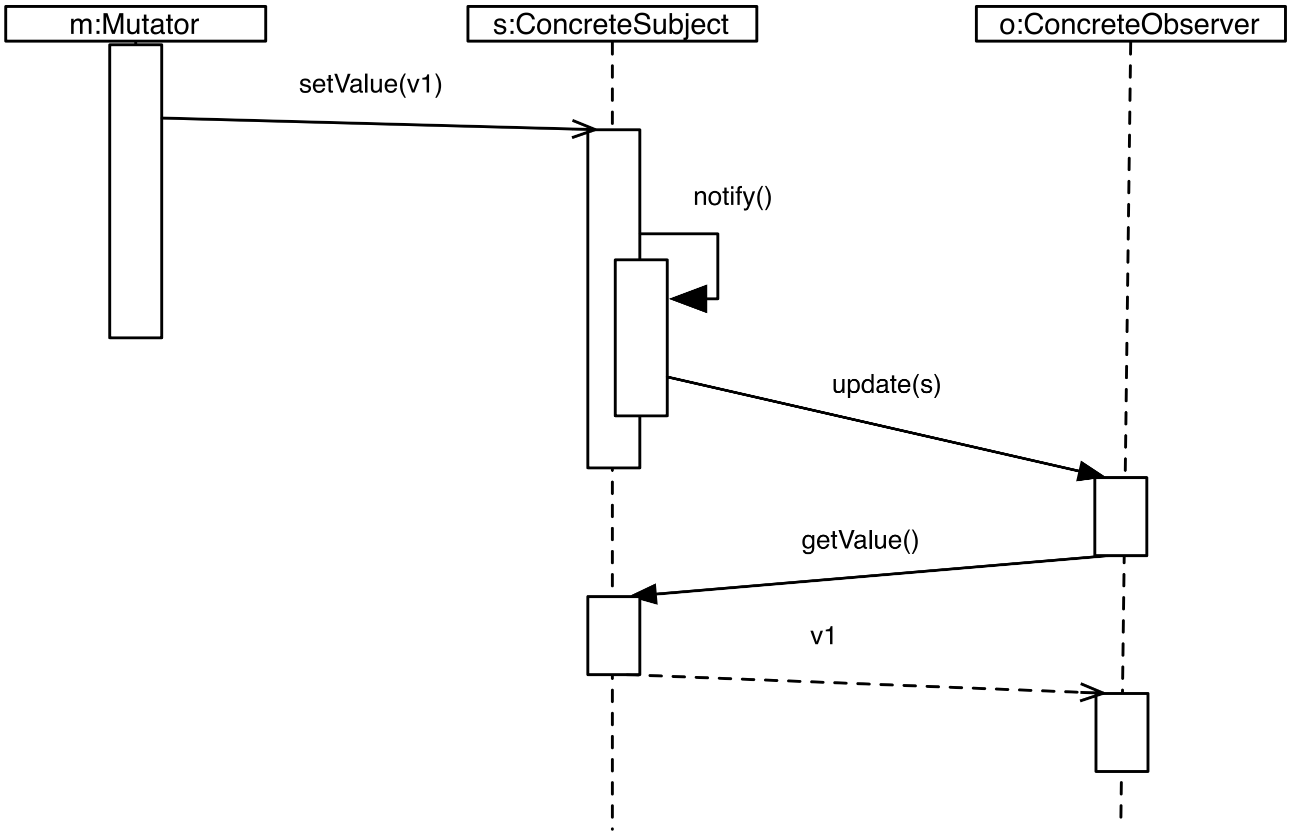
Structure


V
<<interface>>
Subject
+attach(o:Observer(V))
+detach(o:Observer(V))
+isAttached(o:Observer(V)):Boolean
+getValue():V
+setValue(V)
*
*
V
<<interface>>
Observer
+update(s:Subject(V))
V
ConcreteSubject
value:V
+attach(o:Observer(V))
+detach(o:Observer(V))
+isAttached(o:Observer(V)):Boolean
+getValue():V
+setValue(V)
-notify()
ConcreteObserver
anything
+update(s:Subject(V))

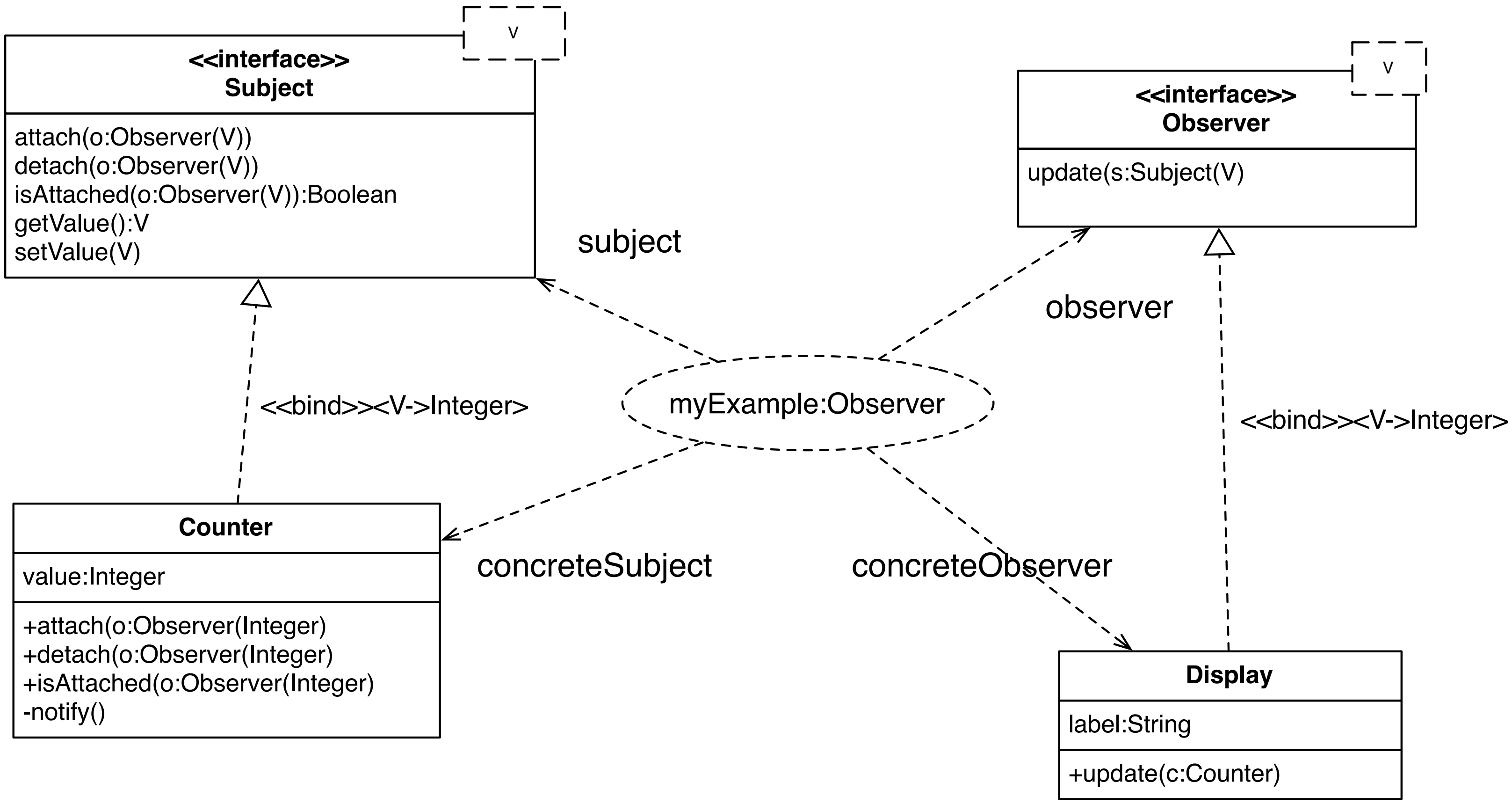
Collaborations

- There are many ways to describe them
  - informal description using text
  - UML sequence diagrams (including HMSC constructs)
  - UML collaboration diagrams
  - statecharts
  - …

Observer collaboration



Example of instantiation



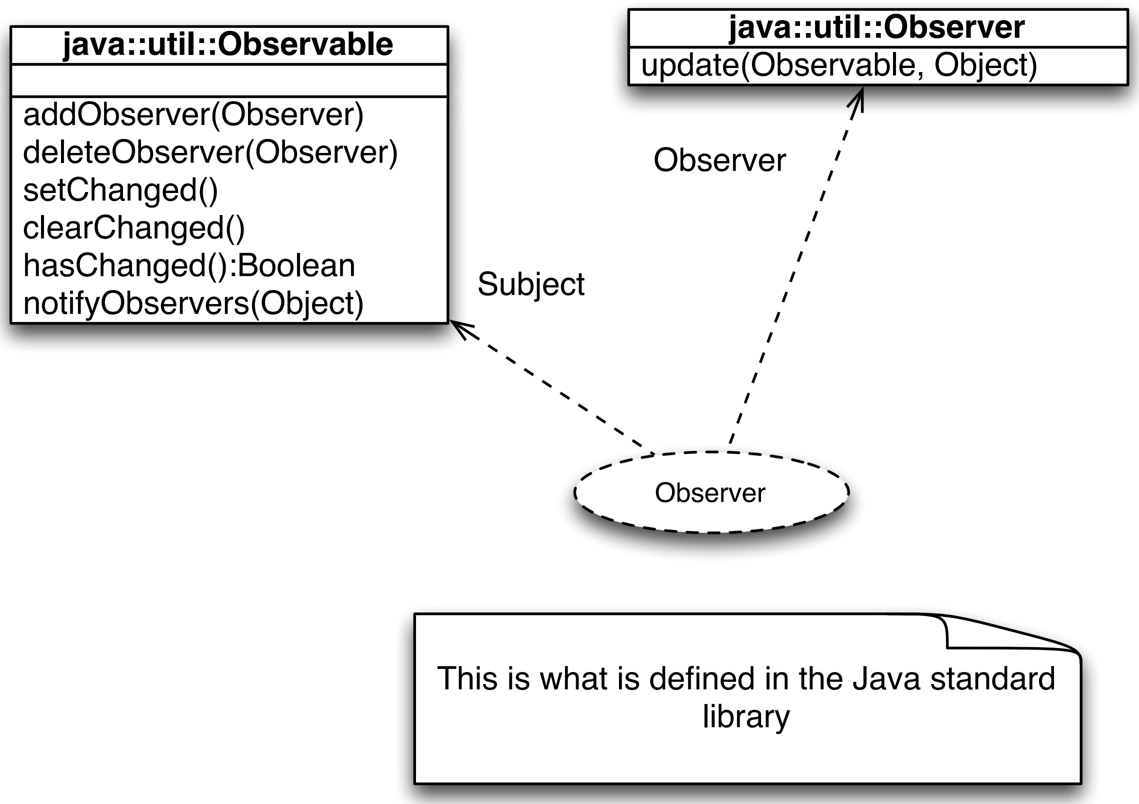
Implementation possibilities

- As Observer is often used one could try to build a reusable implementation
  - Oracle has one but it is awkward
  - Factorize the notification code into a stateless concrete subject
  - Use genericity

An example of implementation

- See the ObserverExample module in the ACO2018 repository

About the Observer implementation in the standard library



Good and bad choices

- Observer is an interface: good
- Observable (= Subject) is a class: not so good, as method inheritance is consumed
  - a concrete subject cannot inherit methods from problem-related classes
  - implementation is mentioned in parameters and attributes (see rule #EJ52)

A much better implementation: JavaFX

- JavaFX is the current Oracle library for graphical user interface design
- This library has good implementations of several design patterns, including Observer
- We will have a look at this library in a moment

The Command design pattern

- Intent
  - Reify an operation concept into an object
- Motivation
  - Often one needs to choose an operation and call it later

Classical example

- In most frameworks for computer-human interfaces
  - a click on menu item must trigger some operation into the application
  - at interface creation this operation is represented by a command object
  - on click the command object is executed

Additional benefits

- A command object can store parameters as its attributes
- A command can implement undo & redo operations

Participants

- Protocol
  - command
  - invoker
  - receiver
- Remember: no implementation in the protocol definition, only operations and rules of collaboration

Participants (cont'd)

- Implementation
  - concrete command
  - client

Responsibilities

- command
  - define operations common to all commands, for invocation
  - act as a relay between invoker and receiver
  - minimum: execute()
- invoker
  - when appropriate requests execution from a command

Responsibilities (cont'd)

- receiver
  - performs the task upon request by a command execution
- client
  - creates and configure concrete commands
  - register commands with the invoker

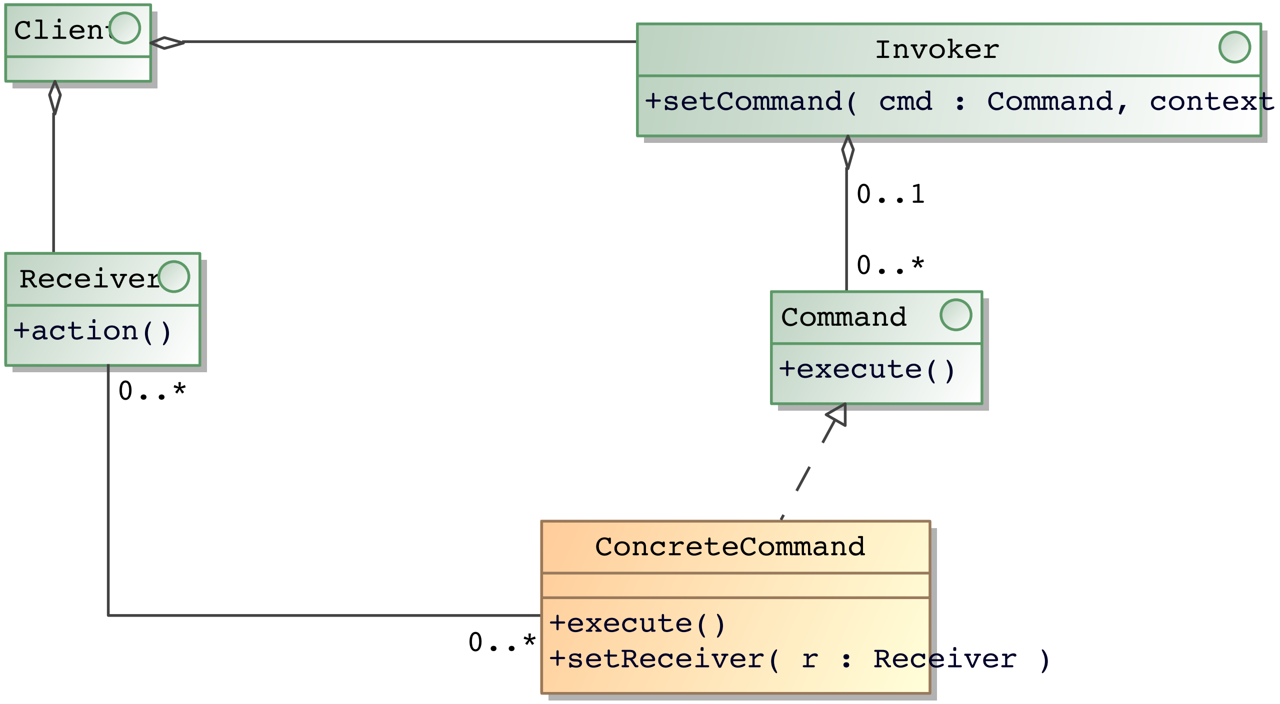
Responsibilities (cont'd)

- concrete command
  - knows which receiver to use and what operation to call
  - implements the command operation to forward calls to receivers

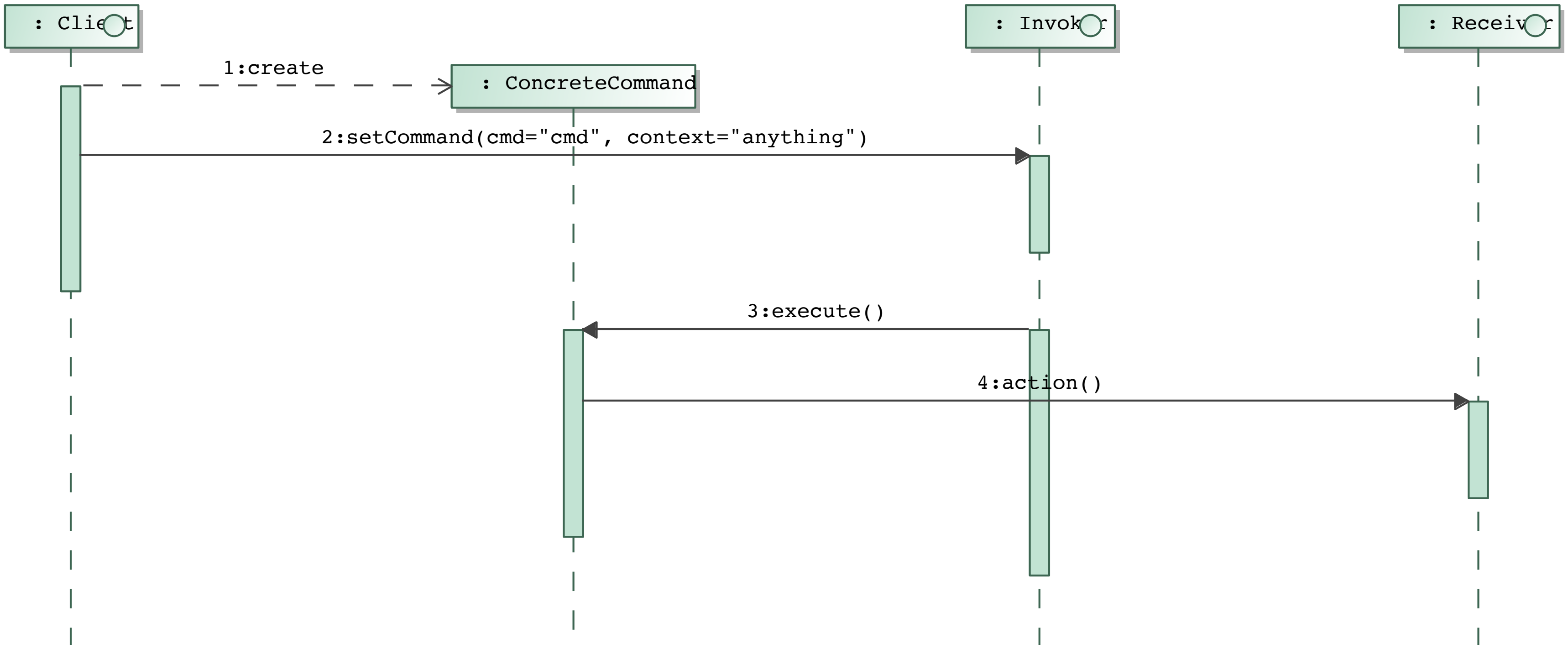
Collaborations

- the client creates command and set up the invoker
- when the time comes
  - the invoker detects this
  - retrieves the command set up by the client
  - call the execute() operation of this command
- the execute() method calls a given operation on the receiver
- the receiver executes the given operation

Structure


Client
Invoker
+setCommand(cmd : Command, context
0..1
0..*
Receiver
+action()
0..*
Command
+execute()
ConcreteCommand
0..*
+execute()
+setReceiver(r : Receiver)

Collaboration as a sequence diagram

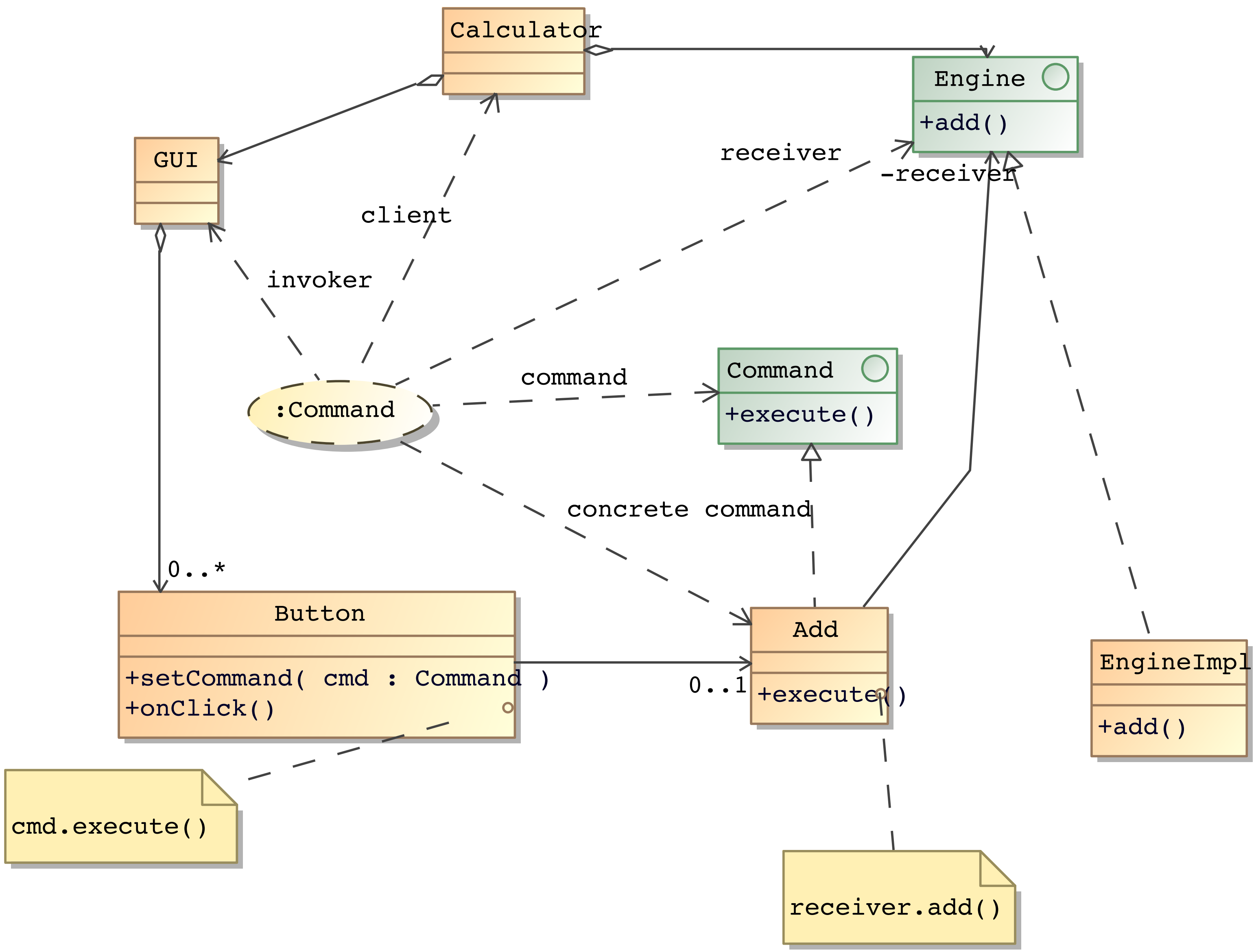


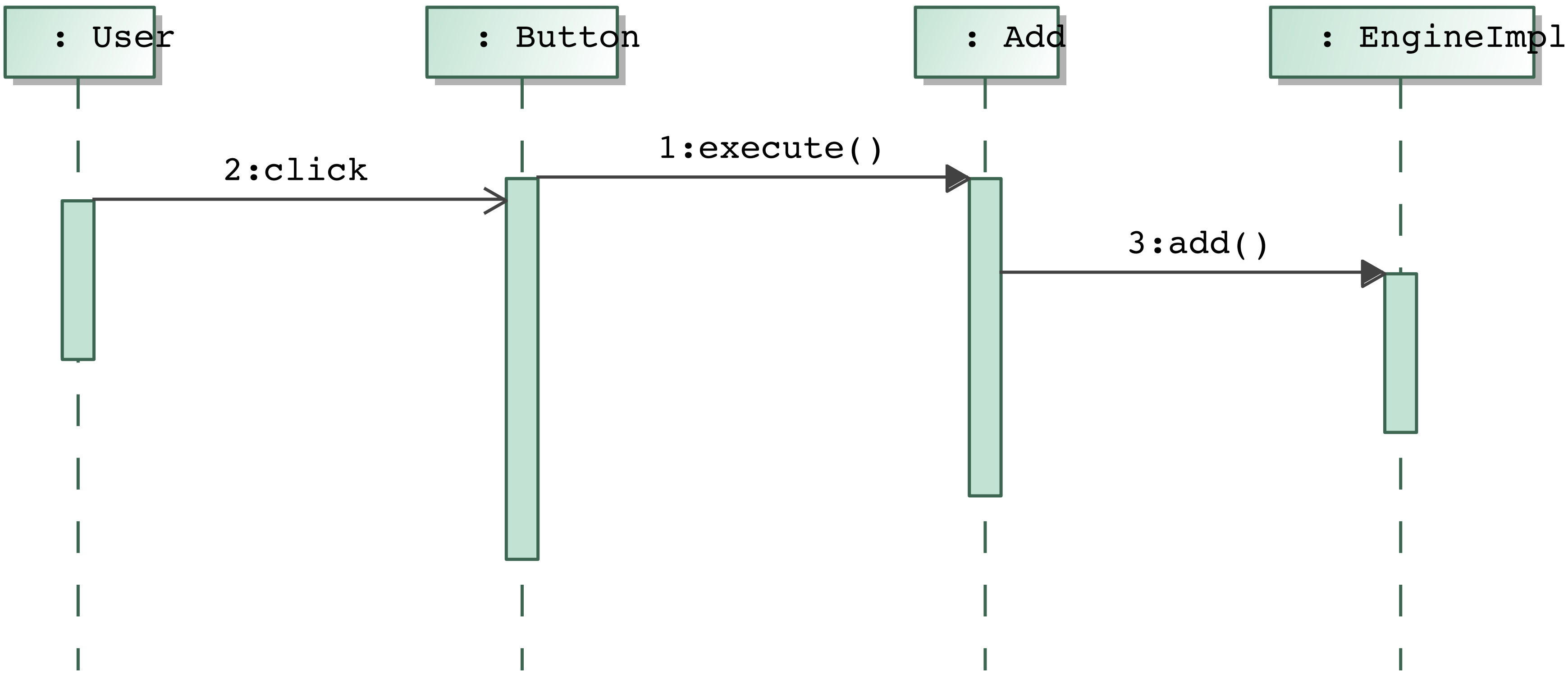
Example: pocket calc

- We design a very simple application for simple computation
- Basic operations
- Number input

Principles of design

- A GUI (graphical user interface)
- A computation engine (a simple stack-based engine)
- GUI and engine are connected
  - by commands (GUI->engine)
  - by observer (engine->GUI)
- A full fledged implementation should use MVC and a-likes (see separate GUI course)


Calculator
Engine
+add()
GUI
receiver
-receiver
client
invoker
command
Command
+execute()
:Command
concrete command
0..*
Button
+setCommand(cmd : Command)
+onClick()
Add
+execute()
0..1
EngineImpl
+add()
cmd.execute()
receiver.add()



: User
: Button
: Add
: EngineImpl
1:execute()
2:click
3:add()

Benefits

- The GUI and the engine are not directly connected
- The connection is made by commands set up by the application
- This maximise reusability of both GUI and engine

The Command design pattern and Java 8

- The concepts of lambda expressions and operation reference (cf previous part) simplifies Command a lot
- Concrete commands are just lambdas or operation reference
- For example
  - interface Command {
    - void execute();
  - }

Concrete commands beforeJava 8

- "Old way"
  - class Add implements Command {...}
- More recent but still old
  - Command cmd = new Command() {
    - @Override
    - void execute() { receiver.doIt();}
  - }

Concrete commands since Java 8

- With lambda expressions
- Command cmd = () -> receiver.doIt();
  - This implicitly declares an anonymous class with a method for the Command::execute operation, thanks to type inference
-

Concrete commands since Java 8

- With operation reference (aka "method reference")
- See example LambdasAndStuff in the ACO2018 repository
  - Consider types Receiver, ReceiverImpl, BasicAction and test4 operation
- Oracle forgot to define a (void) -> void operation type, hence the BasicAction declaration in the example

The Type/instance design pattern

- Intent
  - To implement dynamic types using static types
- Motivation
  - Static types limit reusability and extensibility
  - Some languages do not directly support dynamic creation of new types (eg Java, C#, C++)

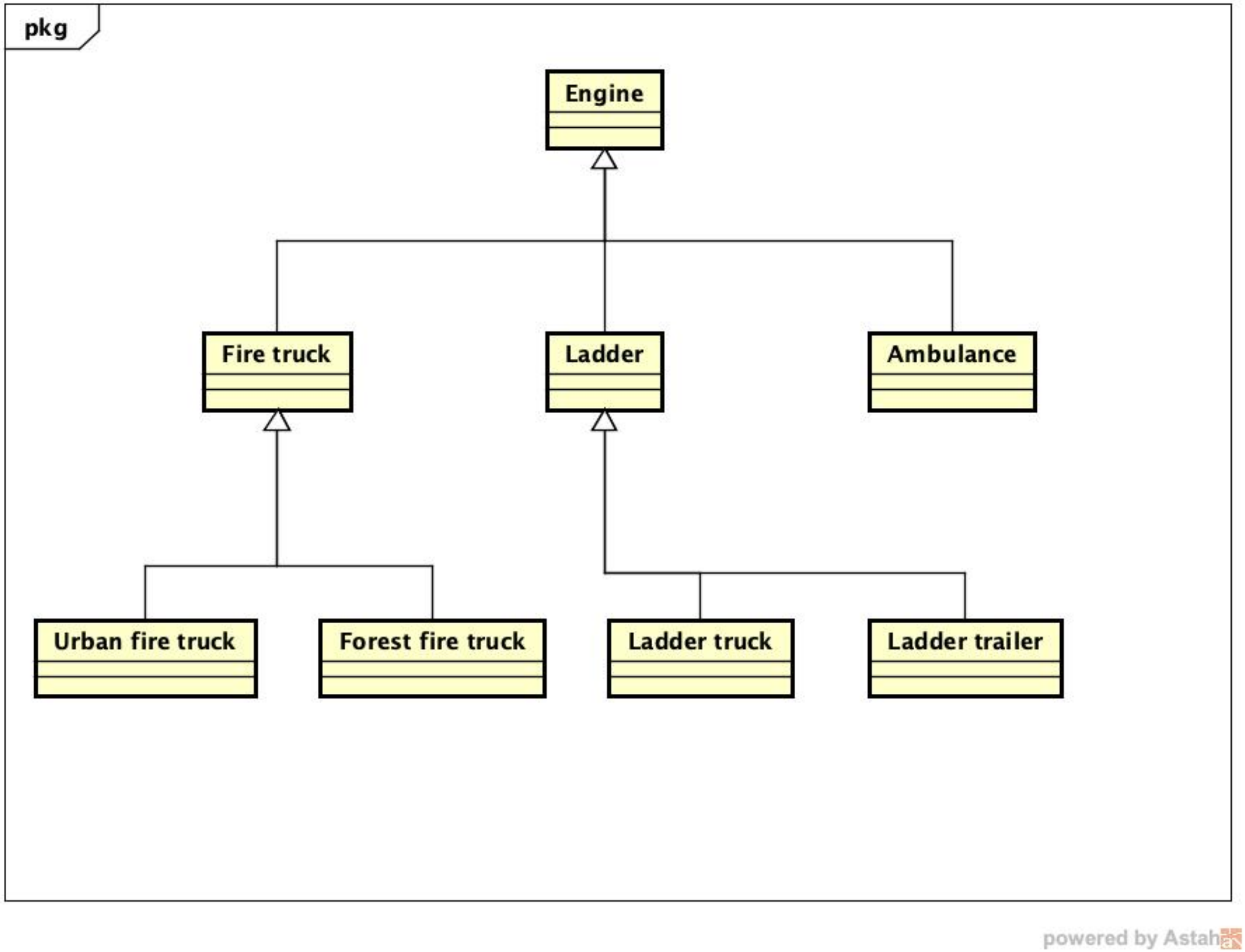
Example

- E-commerce system (A*zon, etc)
  - Millions of product **types**
  - Many variants of a same product type (colour, size, optional features, etc)
- Application of OO principles to represent shared features (data, behaviour)
  - Millions of classes
  - Addition of new classes many times each day

Another concrete example

- Firefighter crisis management system
  - Engines and other resources
  - Persons (capabilities, location...)
  - Types of calls (fire, medical, pollution, etc)
  - Types of locations (private house, public place, high rise, hospital, etc)
- All theses types have complex properties and from an OO point of view various complex behaviours (methods!)

Another concrete example (continued)



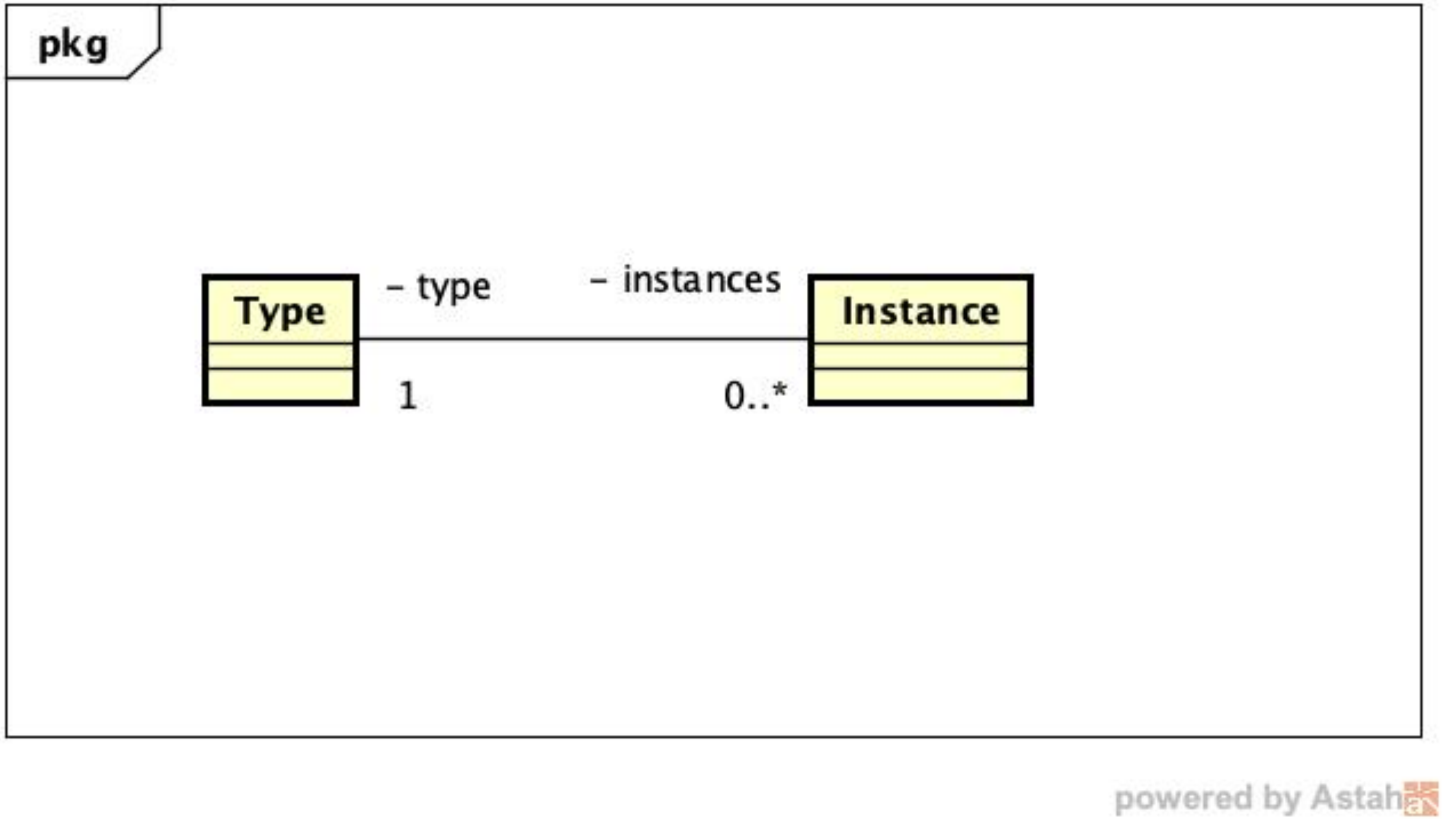
Another concrete example (continued)

- The hierarchy factories common attributes and methods
- Subtypes redefine some behaviours
- There are complex rules encoded in the code of these classes, like what qualifications are needed to board a given engine in a given situation
- A very expensive software, and now you need to adjust it to new types of vehicles, new qualifications, new rules...

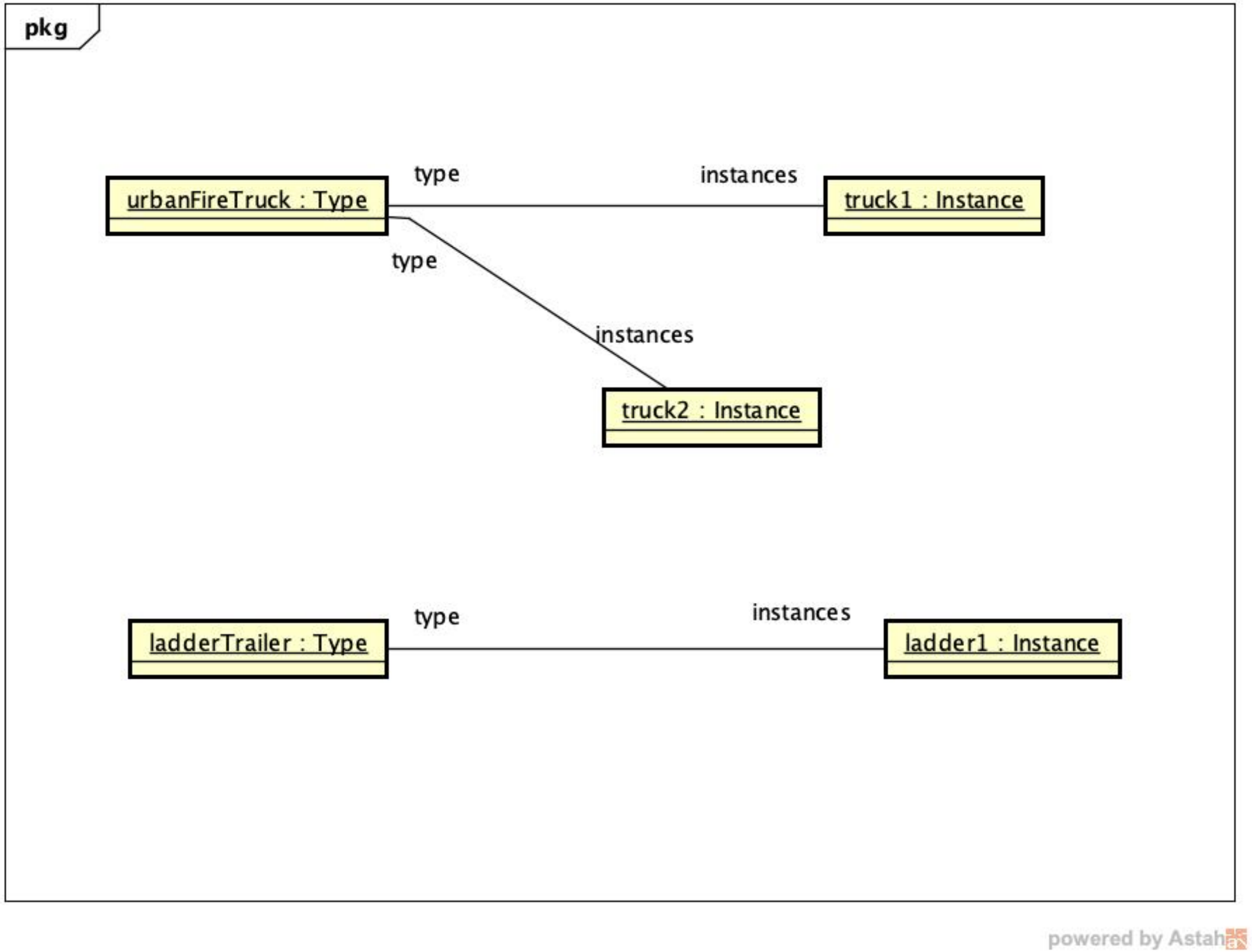
Description of the type/instance pattern

- General idea:
  - no more inheritance hierarchy with hundreds of classes
  - instead each class is defined by… an object, an instance of a single class Type!
  - each instance is also an object (of course) but they are all of the same type: a single class Instance

Structure



Application



Advantages and drawbacks

- New types of engine, location, etc can be added or modified at any time without source code changes
- Mapping to files and databases is easy
- But we lost the advantages of OO
  - type checking
  - dynamic dispatch of methods
  - this means that we have to recode this (cf the lab project)