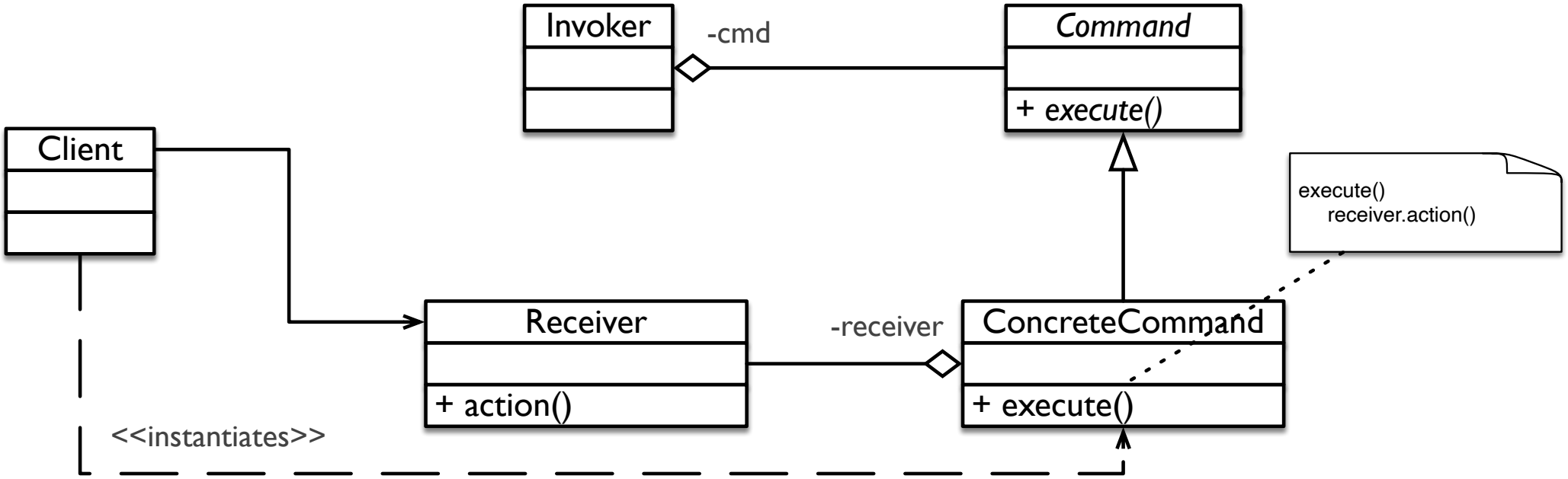
Command

- Purpose
  - Encapsulate a request in an object …
  - … allowing to: configure clients with different requests, maintain a history of the requests, manage the undo

- If a request, a command, is an object,
  - it has its own "life": it can be memorised, passed as an argument, …

Command diagram



Mechanism

- An instance of Client creates an instance of ConcreteCommand passing the Receiver
  - The instance of Command is a command; at this point, the command is an object …
- Afterwards, an instance of Invoker (which keeps track of the commands instantiated so far) through polymorphism invokes execute() on one ConcreteCommand…
- … finally the instance of ConcreteCommand invokes action on the Receiver.

Comments

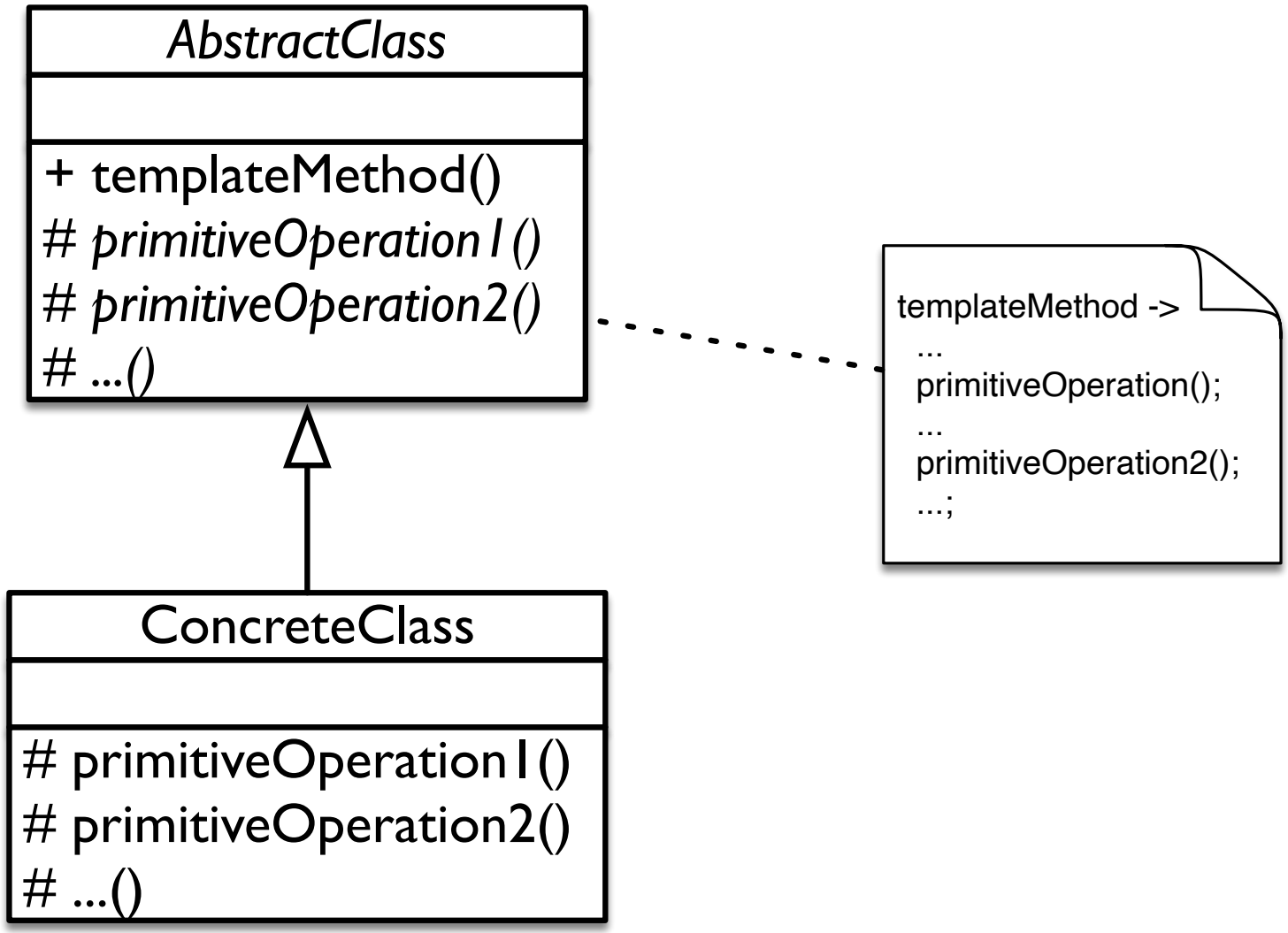
- Command de-couples Invoker (the one invoking the request) from the Receiver (the one executing the request)
- Commands are objects. Therefore:
  - it is possible to compose commands in a single command (Composite pattern)
  - it is easy to add new commands
  - it is possible to keep track of the executed commands to for instance allow undo

Template method

- Purpose
  - Define the structure of an algorithm in one method …
  - … leaving some parts undefined.
  - The implementation of the unspecified parts is contained in other methods which implementation is delegated to subclasses.

- The subclasses redefine only some parts of the algorithm, not the general structure.

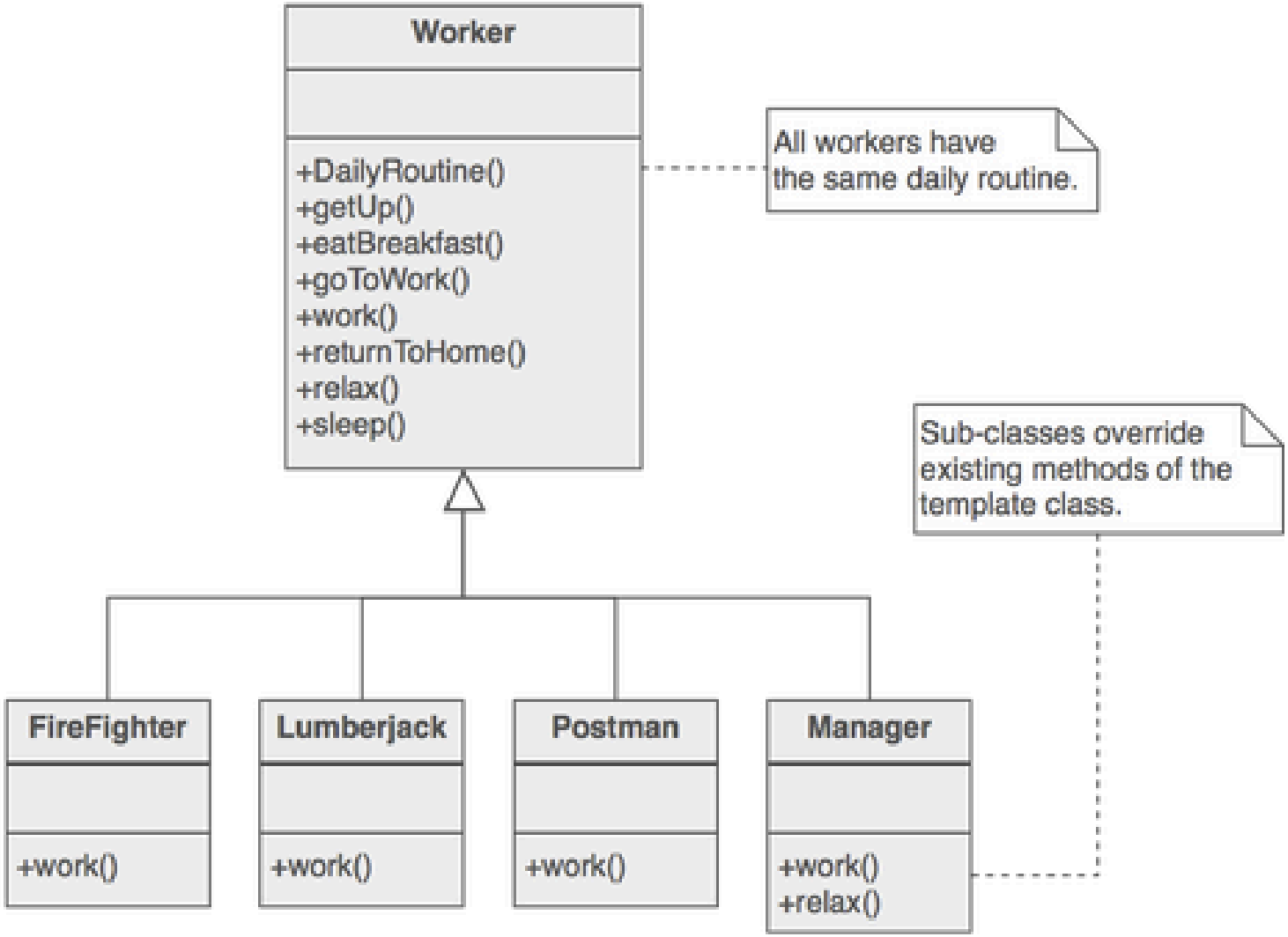
Template method class diagram



Comments

- Very useful pattern in frameworks.
- You can have many more that just one concrete subclass.
- primitiveOperation()
  - are also called hook methods
  - they can also be public …
  - … or they can also be stub methods (not abstract but {})
- Avoid imposing the definitions of too many methods to the subclass(es).

An example


Worker
+DailyRoutine()
+getUp()
+eatBreakfast()
+goToWork()
+work()
+returnToHome()
+relax()
+sleep()
All workers have the same daily routine.
Sub-classes override existing methods of the template class.
FireFighter
+work()
Lumberjack
+work()
Postman
+work()
Manager
+work()
+relax()

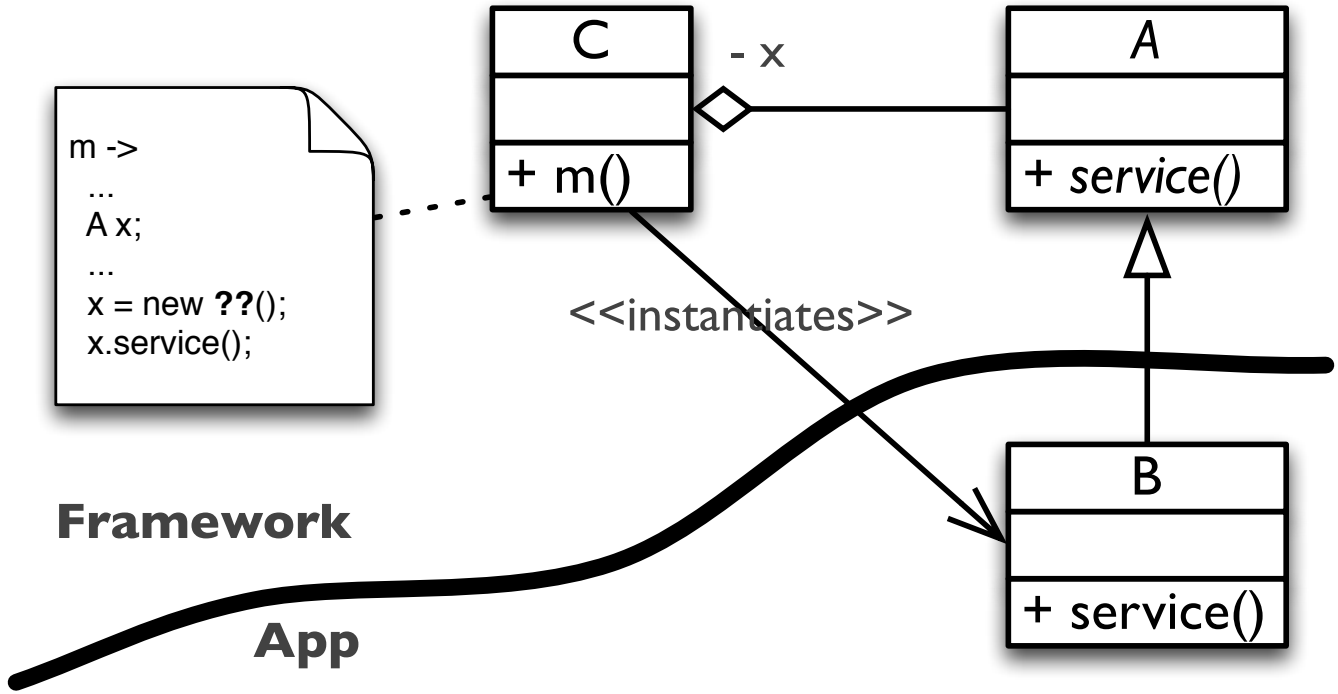
Factory method

- Purpose
  - Delegate the creation of a class to one of the subclasses
  - The subclass(es) decide(s) which instance to create, which constructor to call.
- Typically used in OO frameworks.

What is an OO framework?

- A set of classes
- Difference with a library:
  - Library: the developer of a system, calls the methods of the library
  - Framework: the developer of a system
    - writes methods that are called by the framework
    - "fills the gaps"
    - "Do not call us, we will call you"

Typical problem in a framework

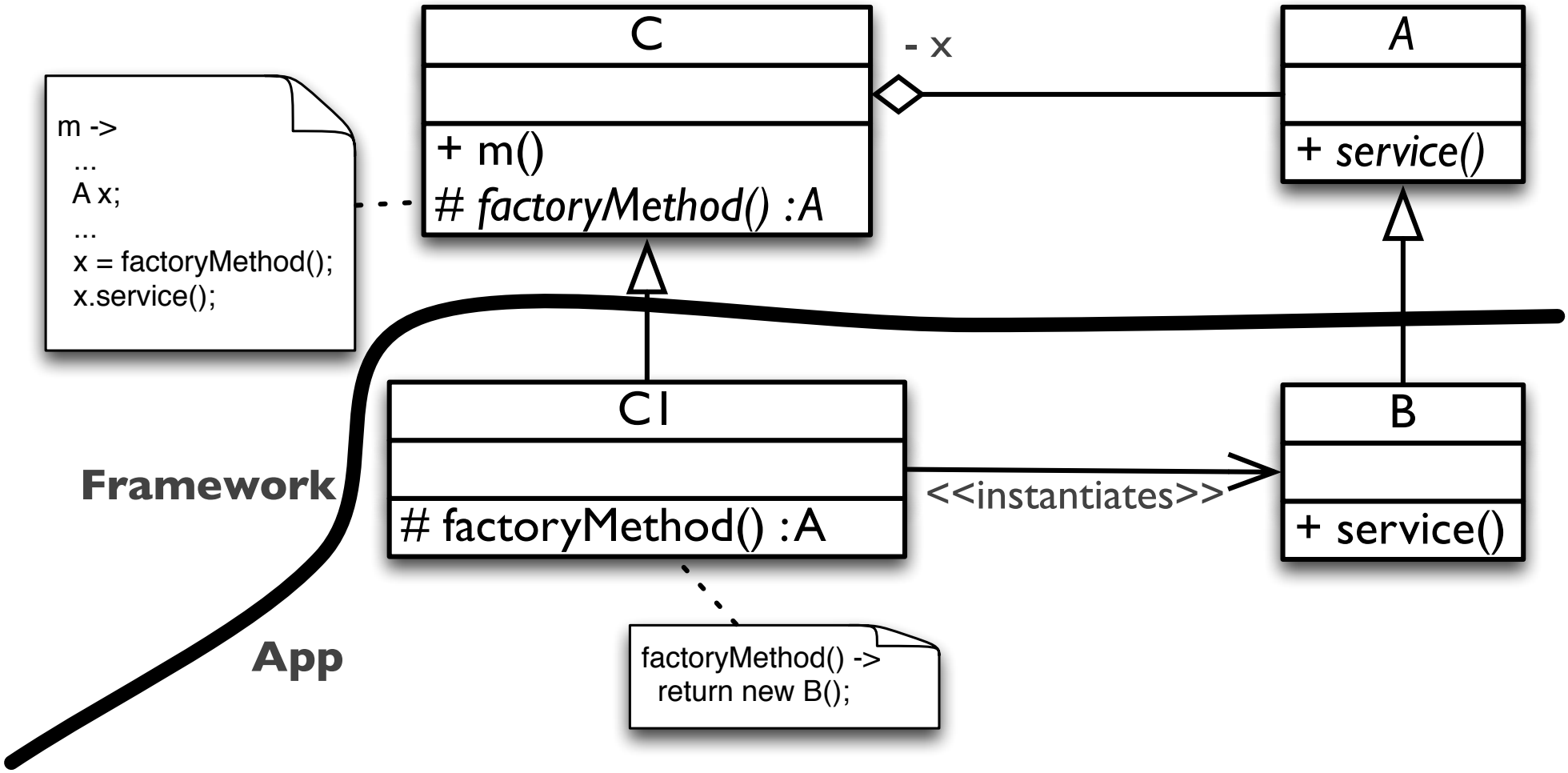


- The framework developer writes a method in a class C …
- … the method needs to create, at a certain precise point, an instance of a subclass B of a base class A …
- … but which subclass B to use is unknown as it is a decision that the developer of the application needs to take, not the framework developer.

Solutions?

- First (non) solution:
  - the framework developer gives up and lets the application developer do all the work.
- Second solution:
  - the framework developer writes what he can/knows: the invocation to an abstract factoryMethod that must create the instance …
  - … and asks the application developer to write, together with the subclass B, also the subclass C1 of C that implements the factoryMethod.

Solutions


C
+ m()
# factoryMethod() :A
- x
A
+ service()
m ->
...
A x;
...
x = factoryMethod();
x.service();
C1
factoryMethod() :A
<<instantiates>>
B
+ service()
Framework
App
factoryMethod() ->
return new B();

Comments

- Instead of having explicit knowledge on the creation of a class C …
- … the creation is delegated to a subclass C1.
- Reason: while writing the class C, the knowledge about which subclass to instantiate is missing.

Factory method [GoF]

- Purpose
  - Define an interface for the creation of an object (the factoryMethod)…
  - … leaving to the subclasses the decision about the class that needs to be instantiated.
  - Allows to define the instantiation of a class at the subclasses level.

Template method versus Factory method

- The basic idea is pretty similar:
  - Given a class, have its abstract methods called by another method and defined in the subclass(es).

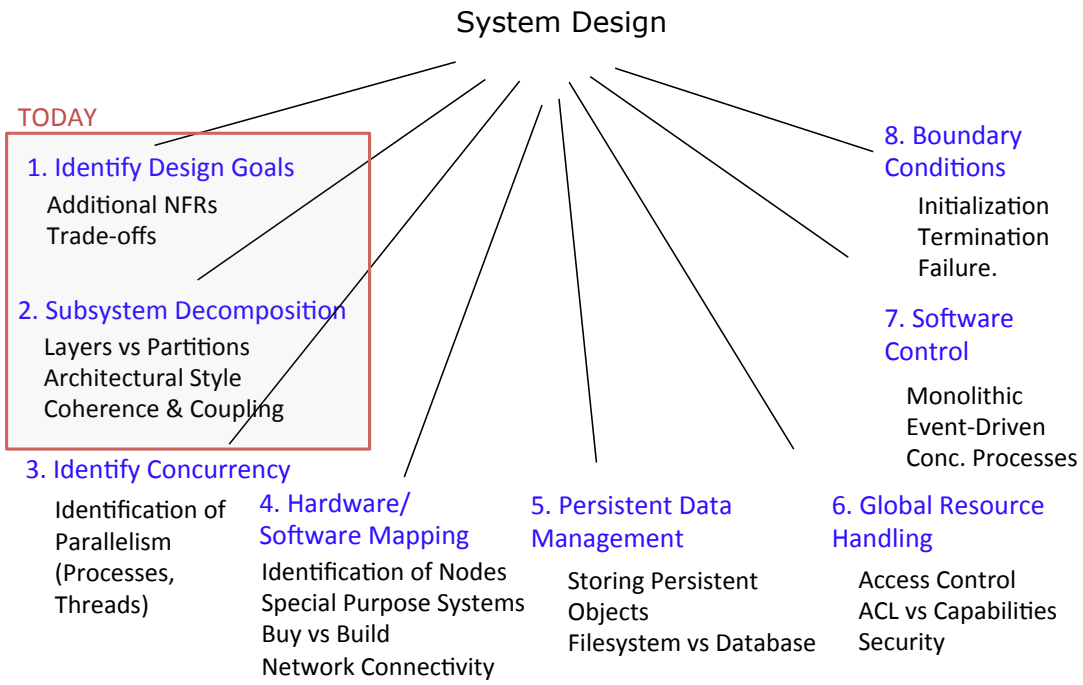
- However, they are different
  - The Template method is the one invoking the abstract methods.
  - The Factory method is the abstract method …
  - … which has to create and return the instance.

Concluding

Key points I

- We are moving from analysis to design, from application to system domain.

- System qualities and trade-offs.
- Subsystems and Classes
- Services and Subsystems Interfaces
- Coupling and Cohesion
- Layers and Partitions
- Architectural Styles



Key points II

- A software architecture is a description of how a software system is organized
  - mainly into components and connectors
- Architectural design decisions
  - the type of application, the distribution of the system, the architectural styles to be used.
- Architectures may be documented from several different perspectives or views
  - a conceptual view, a logical view, a process view, and a development view (4…)
  - and the use cases (…+1)
- Architectural and design patterns
  - means of reusing knowledge about generic system architectures.
  - describe the architecture, explain when it may be used
  - describe its advantages and disadvantages.

Key points III

- Architectural patterns or styles
  - Model–View–Controller (MVC) <— with Rasmus
  - Layered
  - Onion and Clean architecture
  - Repository
  - Client-Server
  - Pipe & Filter
  - Model–view–viewmodel (MVVM) <— with Rasmus

- Design patters
  - Observer
  - Strategy
  - Command
  - Template method
  - Factory method
  - ….

Structural	Behavioural	Creational
Adapter	**Strategy**	Builder
Façade	State	Prototype
Composite	**Command**	**Factory method**
Decorator	**Observer**	Abstract factory
Bridge	Memento	
Singleton	Interpreter	
Proxy	Iterator	
Flyweight	Visitor	
	Mediator	
	Template method	
	Chain of responsibility	