Go to **www.menti.com** and use the code **37 42 91 7**

What comes to your mind when hearing 'patterns' in the context of SwEng?

Mentimeter

Patterns

Patterns are ways to describe best practices and good design.
They capture experience in a way that is possible for others to reuse.

- stylised, abstract description of good practice;
- tried and tested in different environments;
- system organisation that has proved to be successful;
- describe when to use it – and when not;
- strengths and weaknesses.

Patterns

Reporting patterns

Pattern name: Observer

Description: Separates the display of the state of an object from the object itself and allows alternative displays to be provided. When the object state changes, all displays are automatically notified and updated to reflect the change.

Problem description: In many situations, you have to provide multiple displays of state information, such as a graphical display and a tabular display. Not all of these may be known when the information is specified. All alternative presentations should support interaction and, when the state is changed, all displays must be updated.

This pattern may be used in all situations where more than one display format for state information is required and where it is not necessary for the object that maintains the state information to know about the specific display formats used.

Solution description: This involves two abstract objects, Subject and Observer, and two concrete objects, ConcreteSubject and ConcreteObject, which inherit the attributes of the related abstract objects. The abstract objects include general operations that are applicable in all situations. The state to be displayed is maintained in ConcreteSubject, which inherits operations from Subject allowing it to add and remove Observers (each observer corresponds to a display) and to issue a notification when the state has changed.

The ConcreteObserver maintains a copy of the state of ConcreteSubject and implements the Update() interface of Observer that allows these copies to be kept in step. The ConcreteObserver automatically displays the state and reflects changes whenever the state is updated.
The UML model of the pattern is shown in Figure 7.12.

Consequences: The subject only knows the abstract Observer and does not know details of the concrete class. Therefore there is minimal coupling between these objects. Because of this lack of knowledge, optimizations that enhance display performance are impractical. Changes to the subject may cause a set of linked updates to observers to be generated, some of which may not be necessary.

Overview

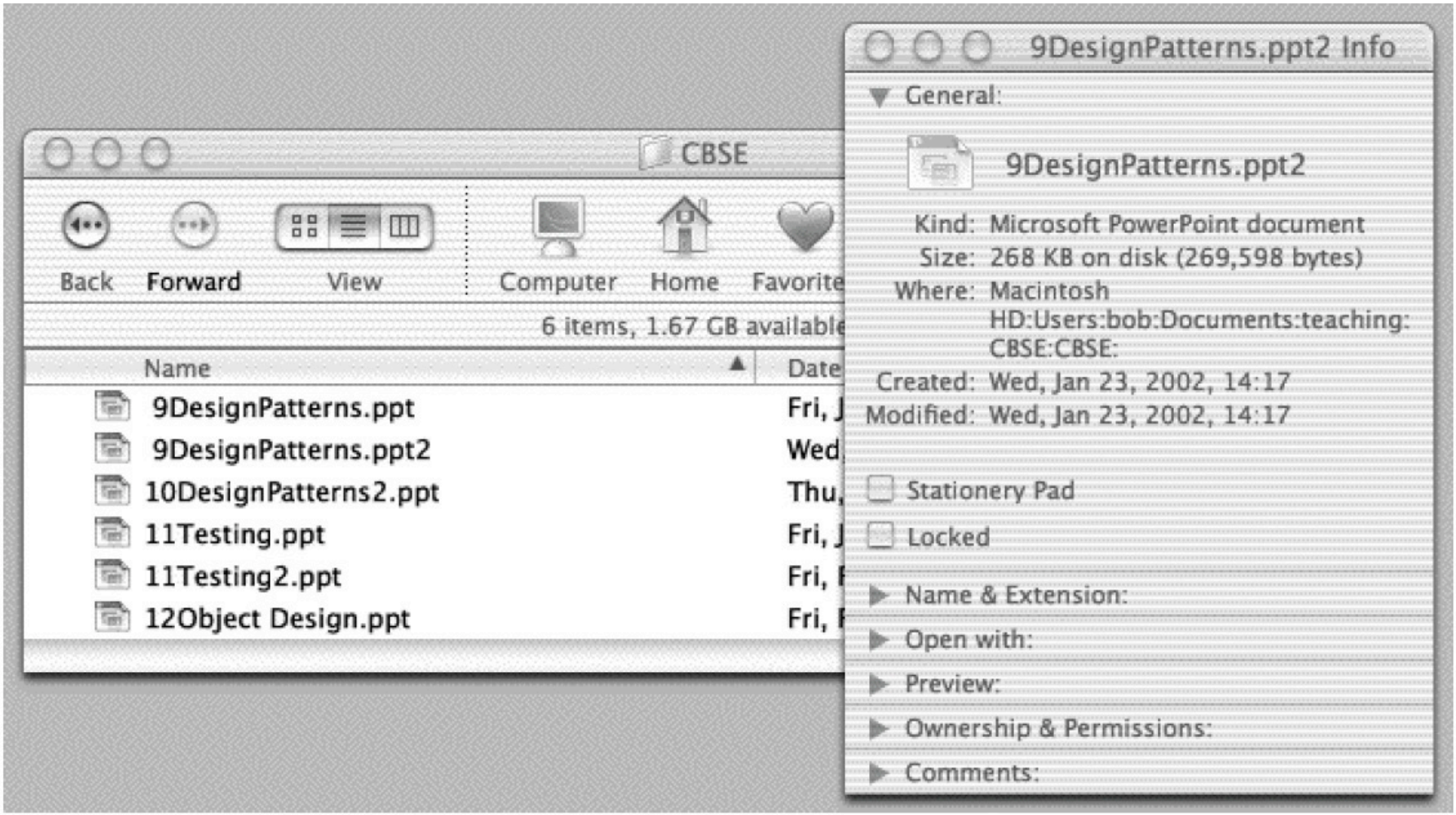
- Architectural patterns or styles
  - Model–View–Controller (MVC) <— with Rasmus
  - Layered
  - Onion and Clean architecture
  - Repository
  - Client-Server
  - Pipes & Filters
  - Model–view–viewmodel (MVVM) <— with Rasmus

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

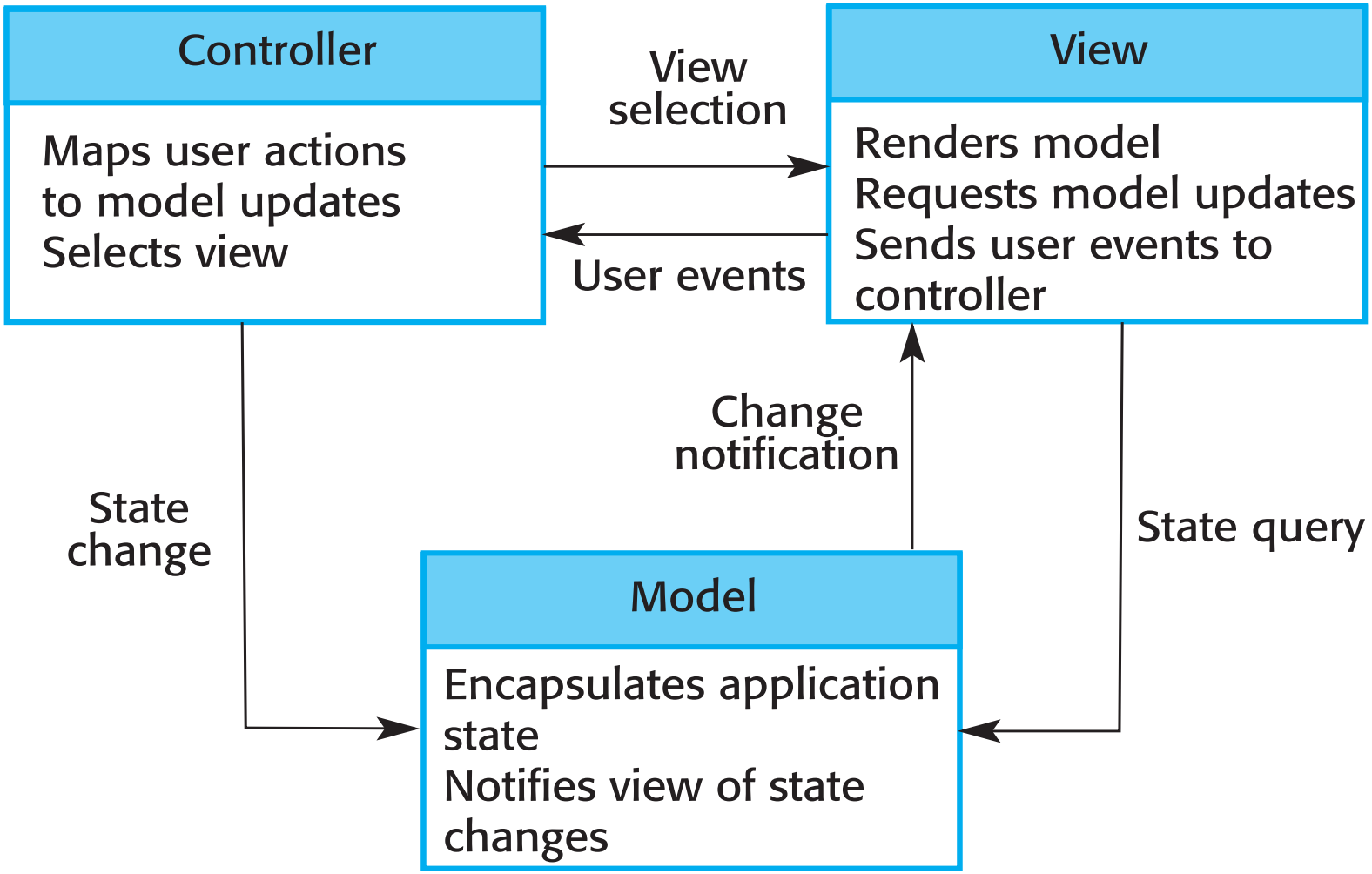
Structural	Behavioural	Creational
Adapter	**Strategy**	Builder
Façade	State	Prototype
Composite	**Command**	**Factory method**
Decorator	**Observer**	Abstract factory
Bridge	Memento	
Singleton	Interpreter	
Proxy	Iterator	
Flyweight	Visitor	
	Mediator	
	Template method	
	Chain of responsibility	

Architectural patterns (styles)

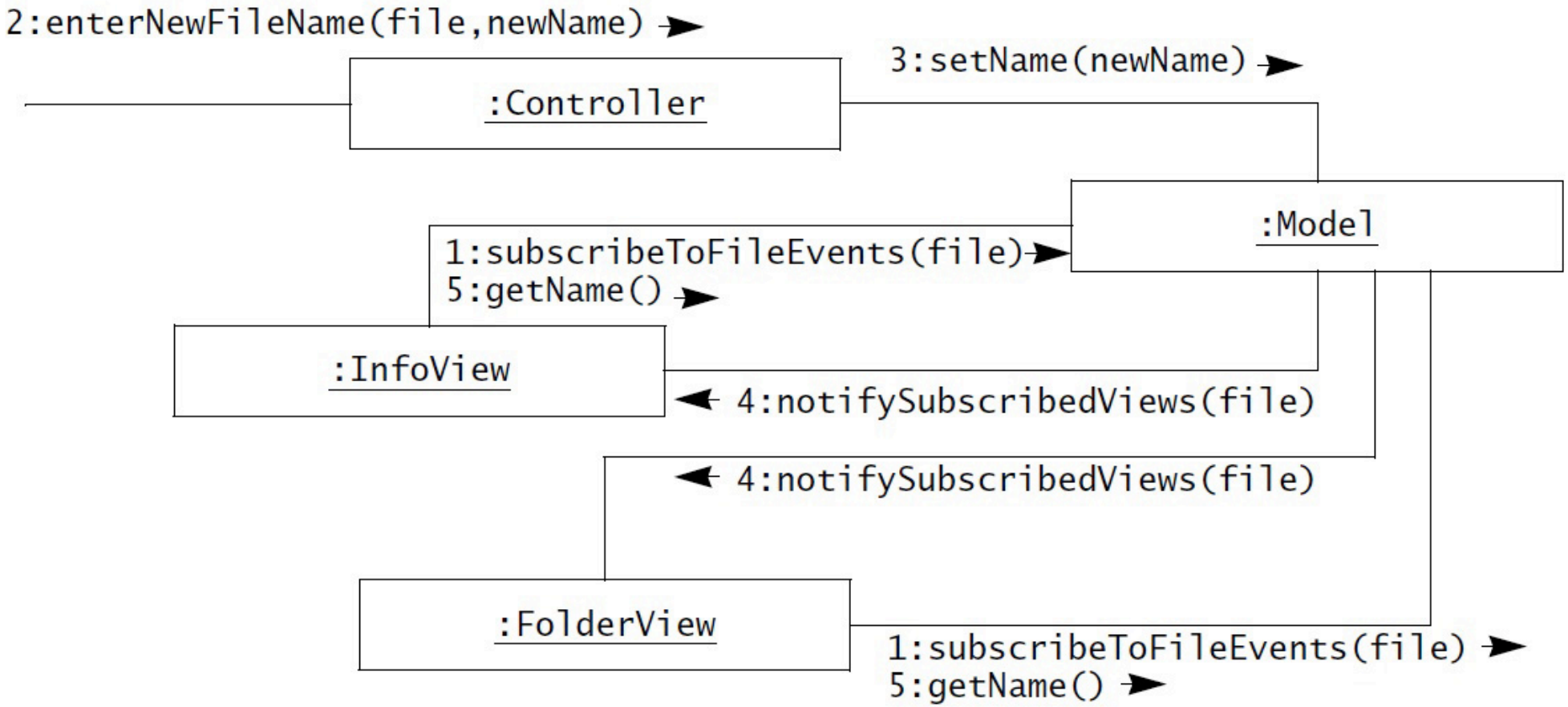
Model-View-Controller (MVC)



The organisation of the Model-View-Controller (MVC)



Model-View-Controller (MVC)



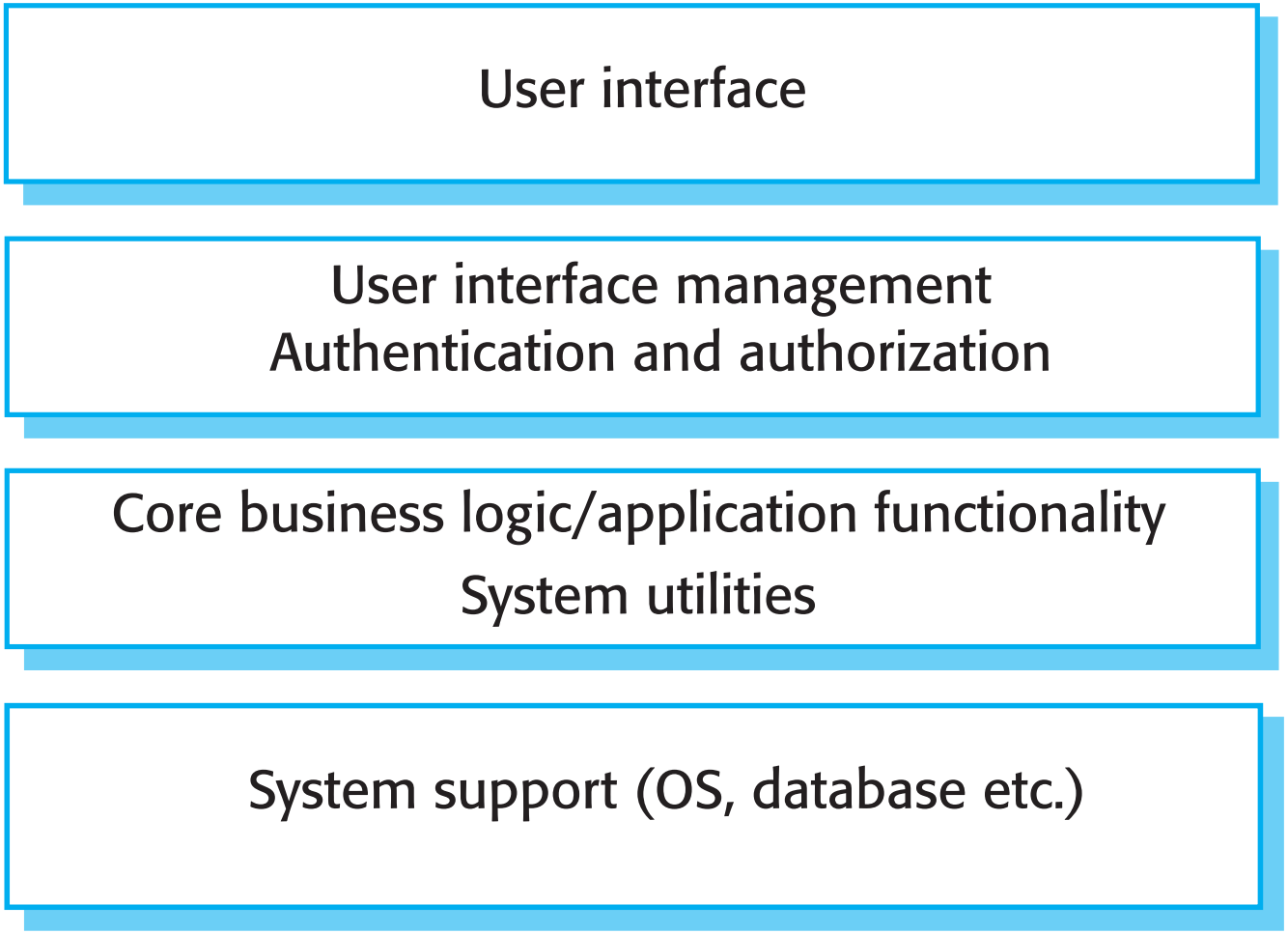
The MVC pattern

Name	MVC (Model-View-Controller)
Description	Separates presentation and interaction from the system data. The system is structured into three logical components that interact with each other. The Model component manages the system data and associated operations on that data. The View component defines and manages how the data is presented to the user. The Controller component manages user interaction (e.g., key presses, mouse clicks, etc.) and passes these interactions to the View and the Model. See Figure 6.3.
Example	Figure 6.4 shows the architecture of a web-based application system organized using the MVC pattern.
When used	Used when there are multiple ways to view and interact with data. Also used when the future requirements for interaction and presentation of data are unknown.
Advantages	Allows the data to change independently of its representation and vice versa. Supports presentation of the same data in different ways with changes made in one representation shown in all of them.
Disadvantages	Can involve additional code and code complexity when the data model and interactions are simple.

Layered architecture

- Used to model the interfacing of sub-systems.
- Organises the system into a set of layers (or abstract machines) each of which provide a set of services.
- Supports the incremental development of sub-systems in different layers. When a layer interface changes, only the adjacent layer is affected.
- However, it can be artificial to structure systems in this way.

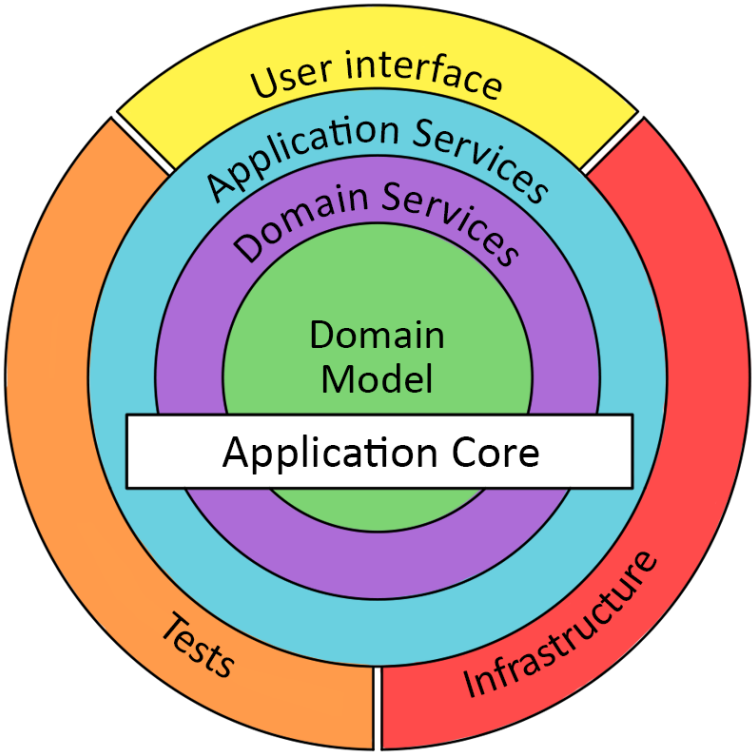
A generic layered architecture



The layered architecture pattern

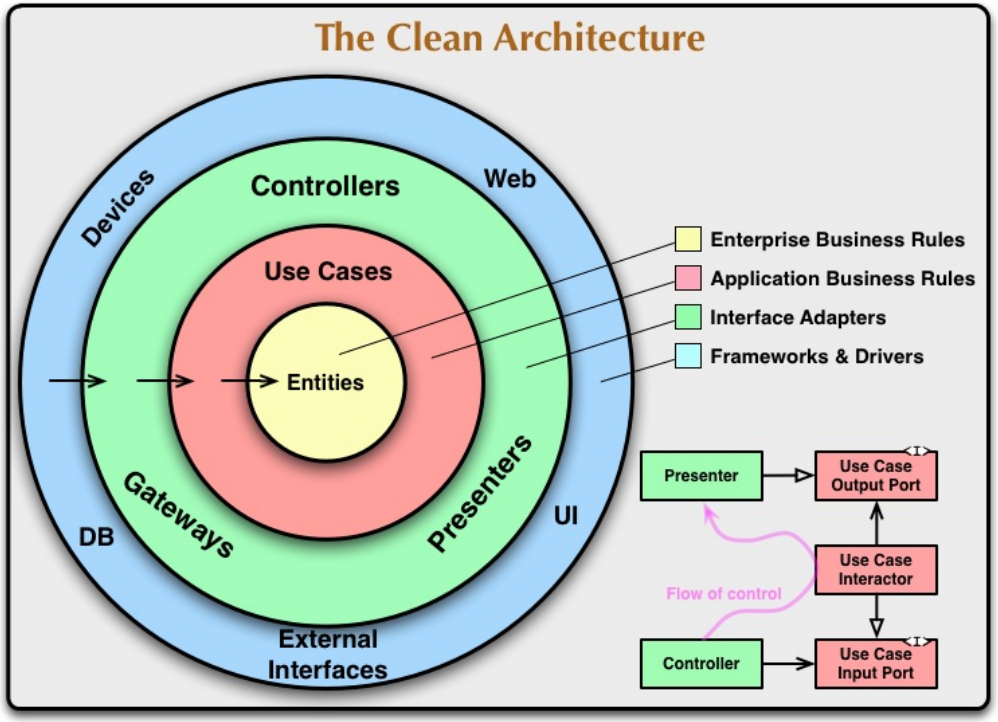
Name	Layered architecture
Description	Organizes the system into layers with related functionality associated with each layer. A layer provides services to the layer above it so the lowest-level layers represent core services that are likely to be used throughout the system. See Figure 6.6.
Example	A layered model of a system for sharing copyright documents held in different libraries, as shown in Figure 6.7.
When used	Used when building new facilities on top of existing systems; when the development is spread across several teams with each team responsibility for a layer of functionality; when there is a requirement for multi-level security.
Advantages	Allows replacement of entire layers so long as the interface is maintained. Redundant facilities (e.g., authentication) can be provided in each layer to increase the dependability of the system.
Disadvantages	In practice, providing a clean separation between layers is often difficult and a high-level layer may have to interact directly with lower-level layers rather than through the layer immediately below it. Performance can be a problem because of multiple levels of interpretation of a service request as it is processed at each layer.

Onion and Clean architecture



Onion architecture

- by Jeffrey Palermo
- Goal:
  - Change should happen at the edge of the onion not the core
  - A layer can depend on the lower layers
  - A layer must be independent of the outer layers
  - Reduce coupling by injecting dependencies



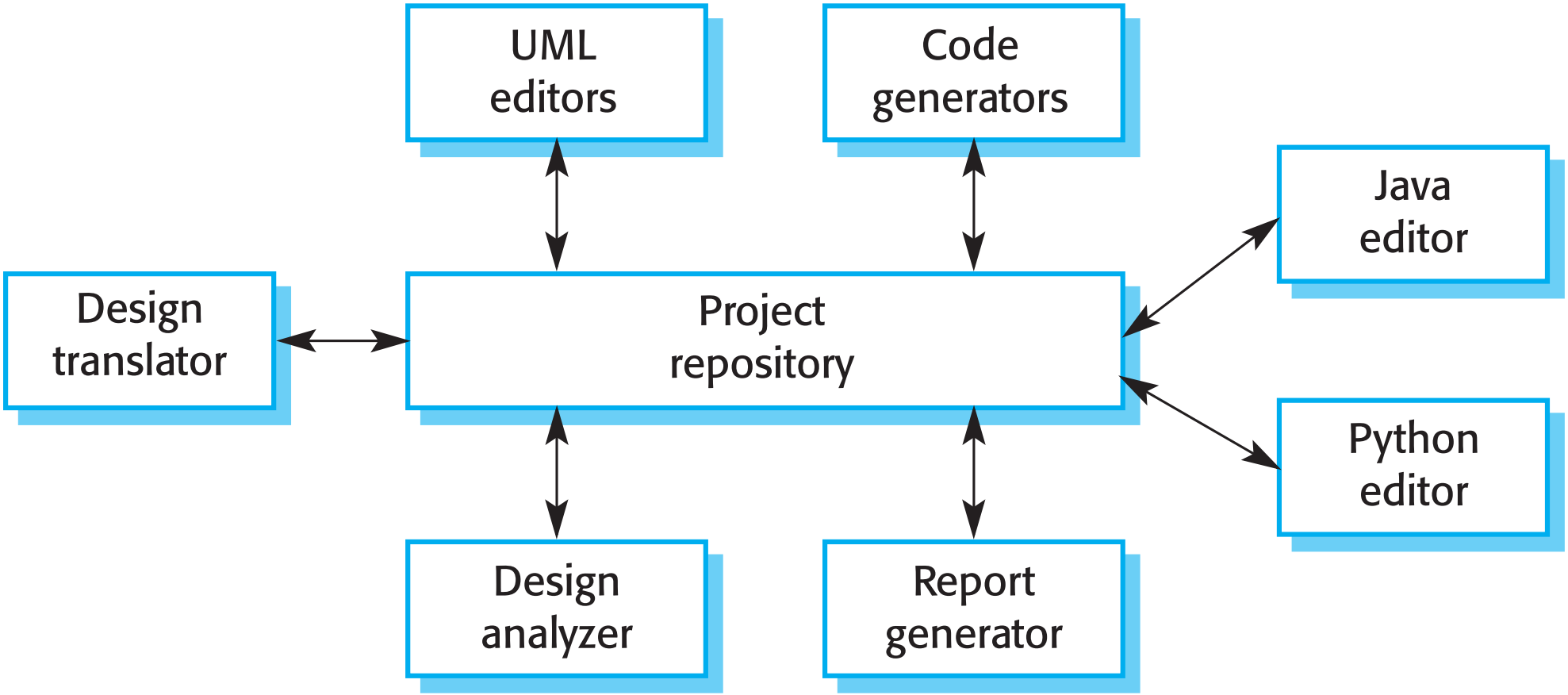
Clean architecture

- by Robert C. Martin (Uncle Bob)
- Extension of onion and hexagonal architecture (by Alistair Cockburn)
- Goal:
  - Provide a way to organise code to encapsulates business logic, while keeping the delivery mechanism separated

Repository architecture

- Sub-systems must exchange data. This may be done in two ways:
  - Shared data is held in a central database or repository and may be accessed by all sub-systems;
  - Each sub-system maintains its own database and passes data explicitly to other sub-systems.
- When large amounts of data are to be shared, the repository model of sharing is most commonly used as this is an efficient data sharing mechanism.

An example of a repository architecture



The repository pattern

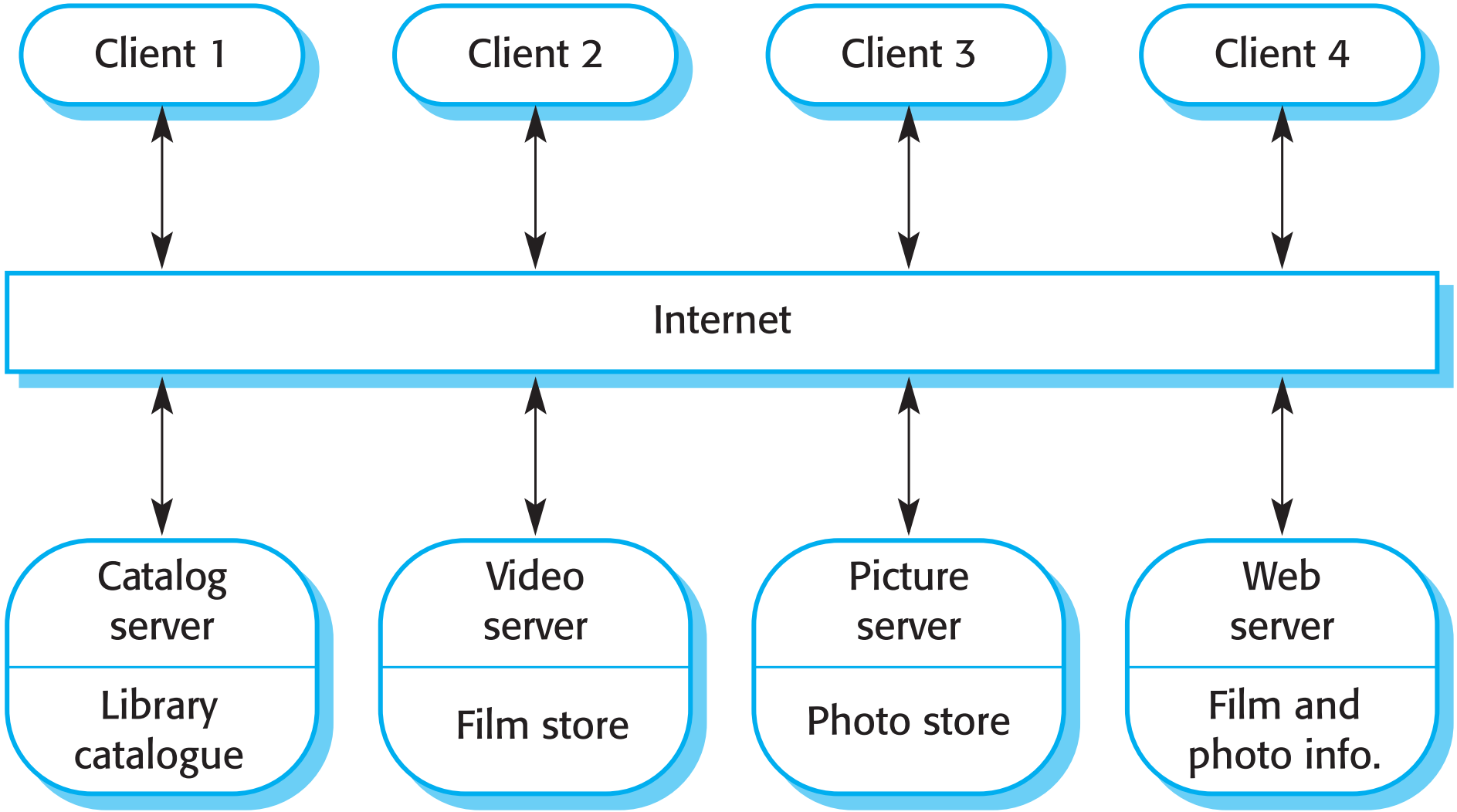
Name	Repository
Description	All data in a system is managed in a central repository that is accessible to all system components. Components do not interact directly, only through the repository.
Example	Figure 6.9 is an example of an IDE where the components use a repository of system design information. Each software tool generates information which is then available for use by other tools.
When used	You should use this pattern when you have a system in which large volumes of information are generated that has to be stored for a long time. You may also use it in data-driven systems where the inclusion of data in the repository triggers an action or tool.
Advantages	Components can be independent—they do not need to know of the existence of other components. Changes made by one component can be propagated to all components. All data can be managed consistently (e.g., backups done at the same time) as it is all in one place.
Disadvantages	The repository is a single point of failure so problems in the repository affect the whole system. May be inefficiencies in organizing all communication through the repository. Distributing the repository across several computers may be difficult.

Client-Server architecture

- Distributed system model which shows how data and processing is distributed across a range of components.
  - Can be implemented on a single computer.

- Three parts:
  - Set of stand-alone servers which provide specific services such as printing, data management, etc.
  - Set of clients which call on these services.
  - Network which allows clients to access servers.

A client–server architecture for a film library



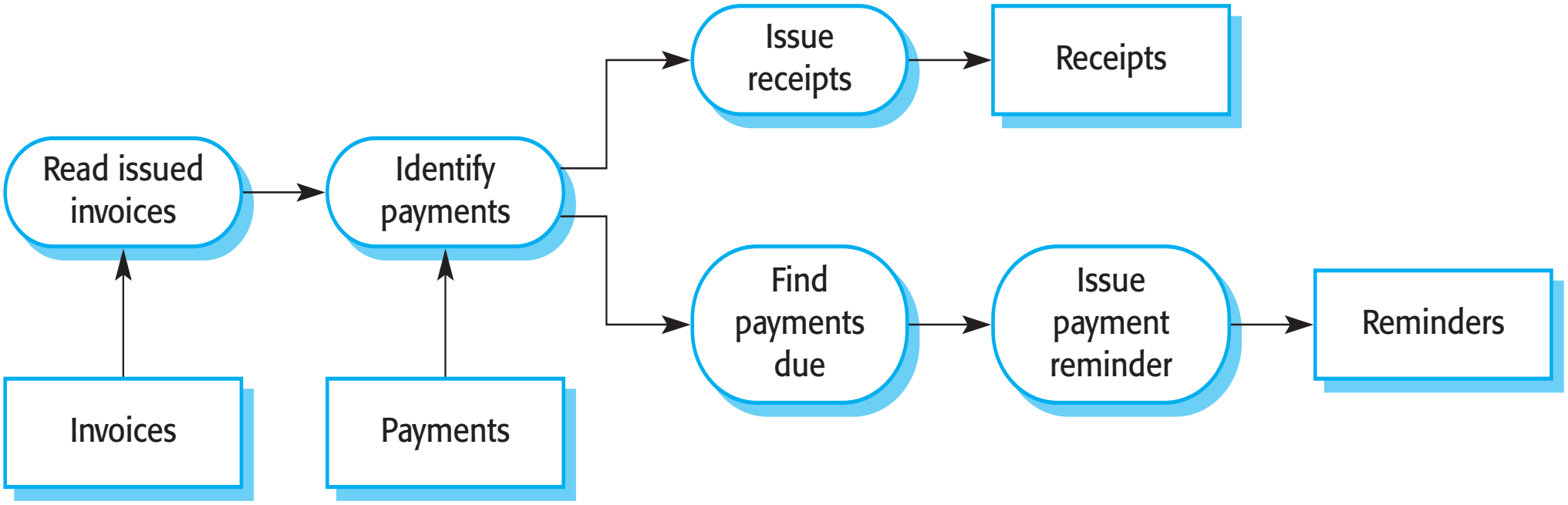
The client–server pattern

Name	Client-server
Description	In a client–server architecture, the functionality of the system is organized into services, with each service delivered from a separate server. Clients are users of these services and access servers to make use of them.
Example	Figure 6.11 is an example of a film and video/DVD library organized as a client–server system.
When used	Used when data in a shared database has to be accessed from a range of locations. Because servers can be replicated, may also be used when the load on a system is variable.
Advantages	The principal advantage of this model is that servers can be distributed across a network. General functionality (e.g., a printing service) can be available to all clients and does not need to be implemented by all services.
Disadvantages	Each service is a single point of failure so susceptible to denial of service attacks or server failure. Performance may be unpredictable because it depends on the network as well as the system. May be management problems if servers are owned by different organizations.

Pipes and Filters architecture

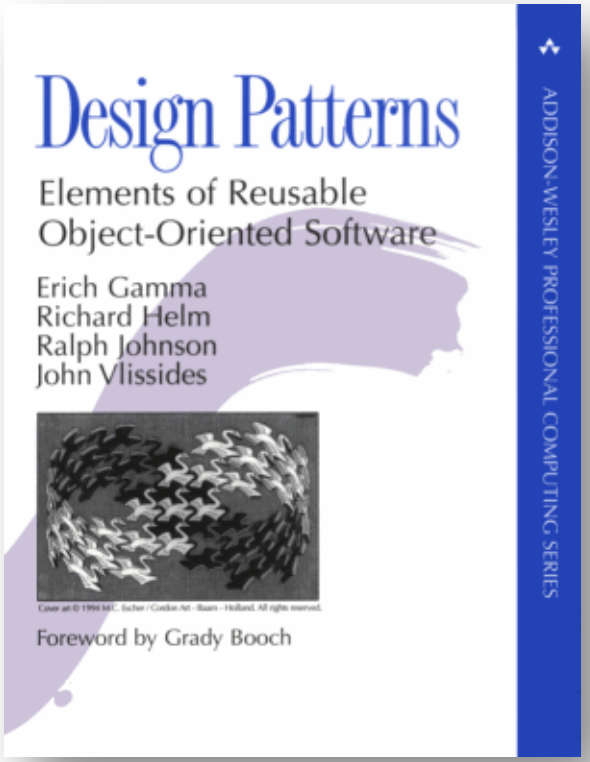
- Functional transformations process inputs to produce outputs.
- Variants of this approach are very common. When transformations are sequential, this is a batch sequential model which is extensively used in data processing systems.
- Not really suitable for interactive systems.

An example of the pipes and filters architecture



The pipes and filters pattern

Name	Pipe and filter
Description	The processing of the data in a system is organized so that each processing component (filter) is discrete and carries out one type of data transformation. The data flows (as in a pipe) from one component to another for processing.
Example	Figure 6.13 is an example of a pipe and filter system used for processing invoices.
When used	Commonly used in data processing applications (both batch- and transaction-based) where inputs are processed in separate stages to generate related outputs.
Advantages	Easy to understand and supports transformation reuse. Workflow style matches the structure of many business processes. Evolution by adding transformations is straightforward. Can be implemented as either a sequential or concurrent system.
Disadvantages	The format for data transfer has to be agreed upon between communicating transformations. Each transformation must parse its input and unparse its output to the agreed form. This increases system overhead and may mean that it is impossible to reuse functional transformations that use incompatible data structures.



Originally proposed by the "Gang-of-Four" (GoF)

Erich Gamma, Richard Helm,
Ralph Johnson & John Vlissides

Design patterns

Overview

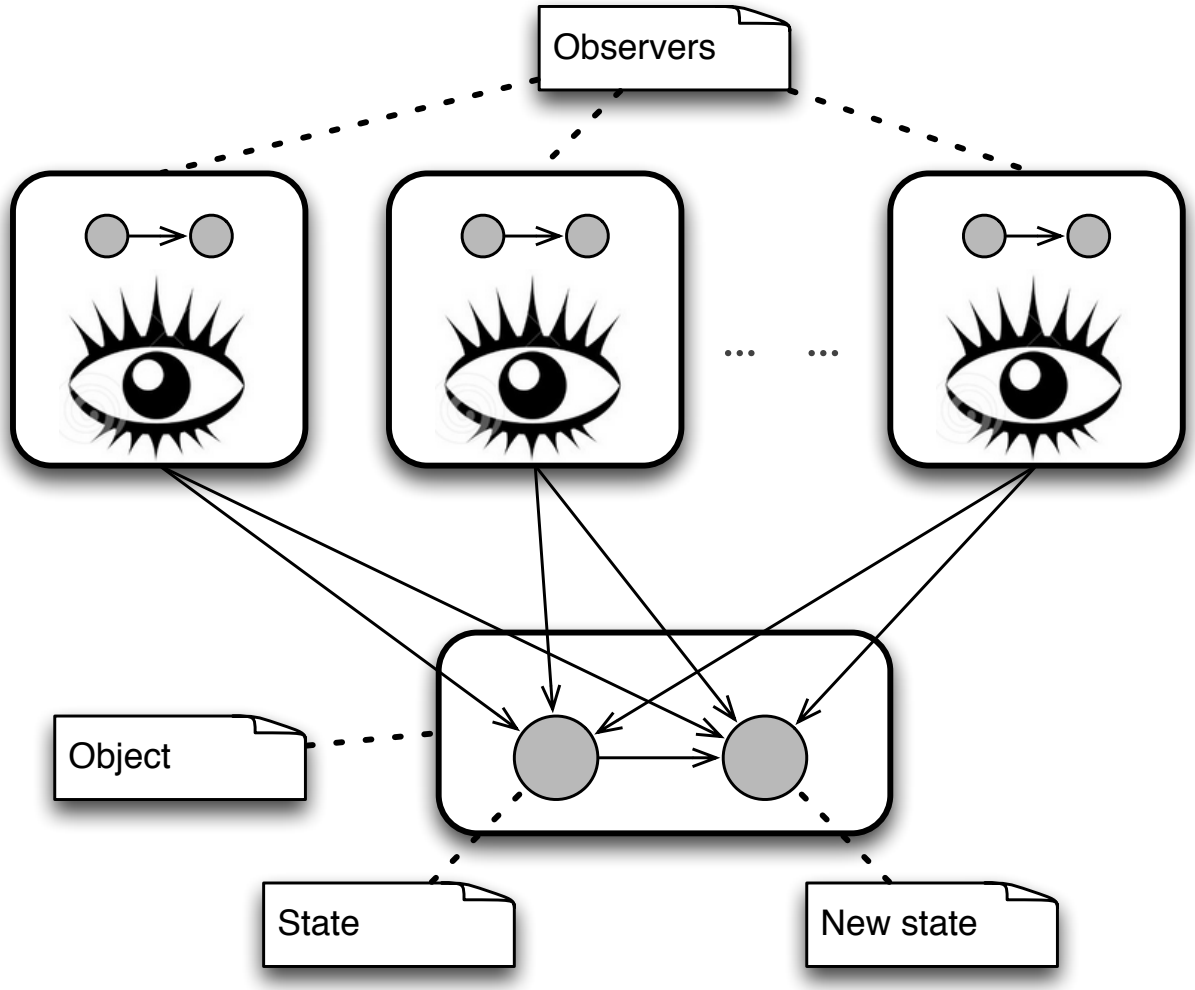
Structural	Behavioural	Creational
Adapter	**Strategy**	Builder
Façade	State	Prototype
Composite	**Command**	**Factory method**
Decorator	**Observer**	Abstract factory
Bridge	Memento	
Singleton	Interpreter	
Proxy	Iterator	
Flyweight	Visitor	
	Mediator	
	Template method	
	Chain of responsibility	

Observer

- Purpose
  - Have a set of objects "observing" the state of an object …
  - … so that they will be notified and updated immediately when the state changes.

- The name is counter intuitive …
  - … the observer does not observe, but awaits to be notified to go looking.

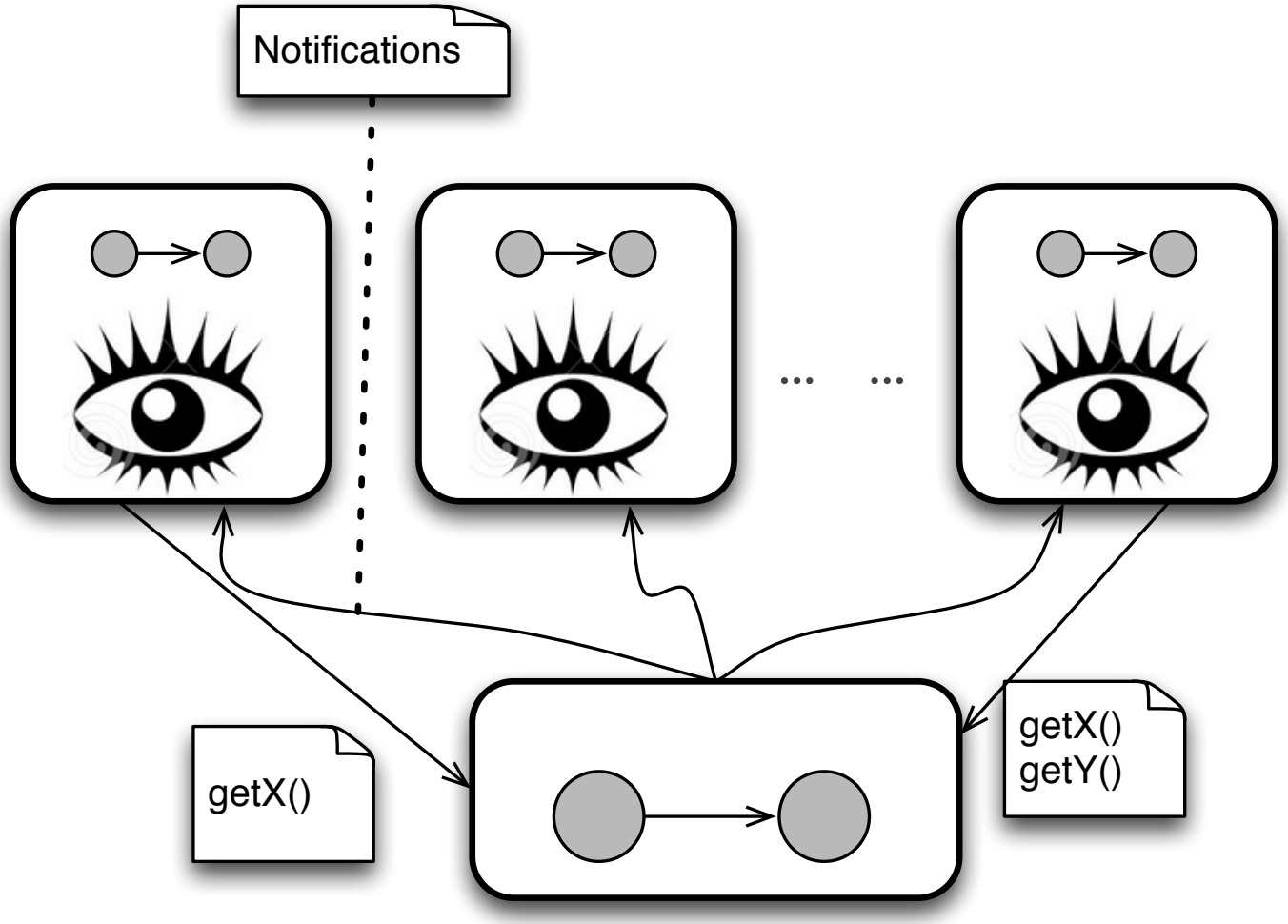
An idea? (naive)



Let's try

- Public attributes? Rejected
- The observers constantly invoke a method *changed* of the observed object?
- Problems! The observer could:
  - be bothering too much the observed object
  - discover the change in the state too late
  - loose one of many rapid changes in the state
  - And if there are many observers? The solution is not scalable!
    - The observer would be spending most of the execution time replying to the observers (potentially to report no variation).

Revisited solution



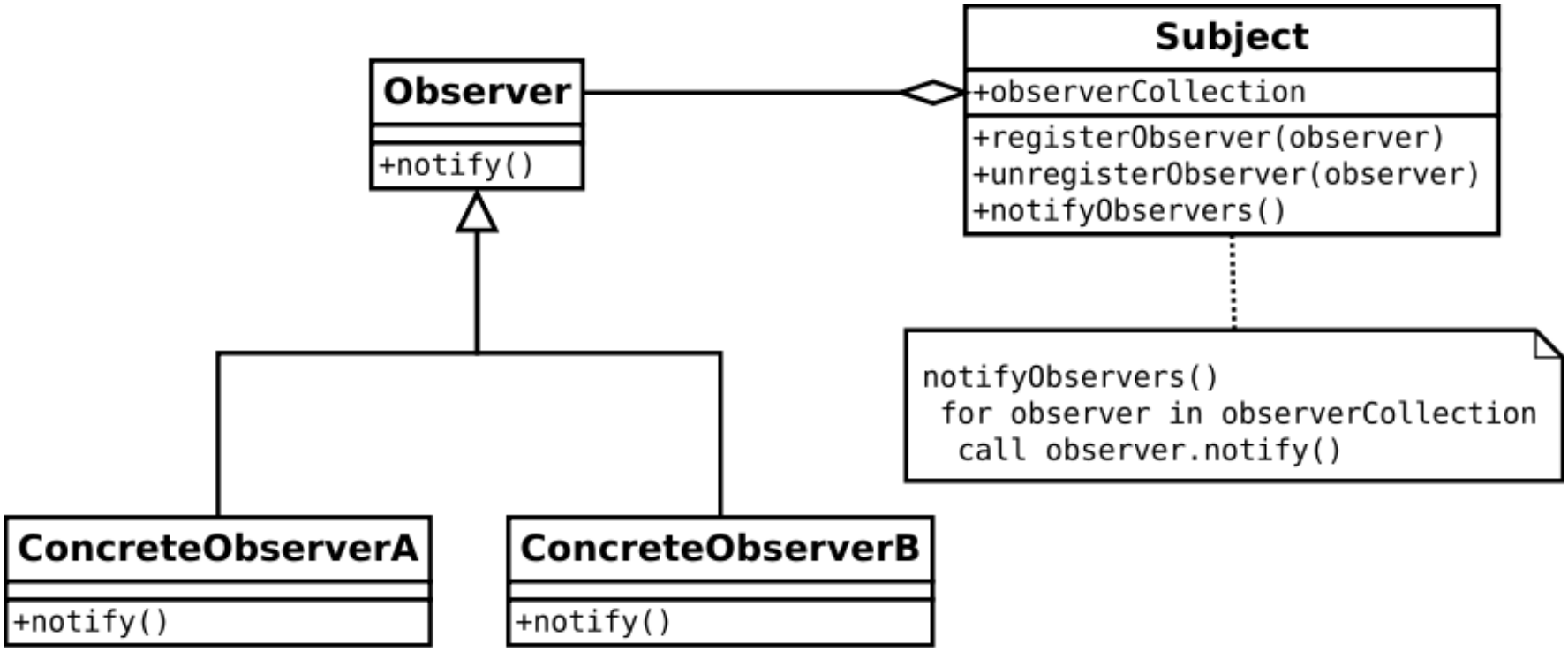
… therefore

- (Prerequisite: the observers need to register)
- When the state of the object changes, it notifies all the observers (by calling a method)
- When notified, the observers decide what to do with the knowledge
  - Nothing
  - Request information about the state of the object
  - …

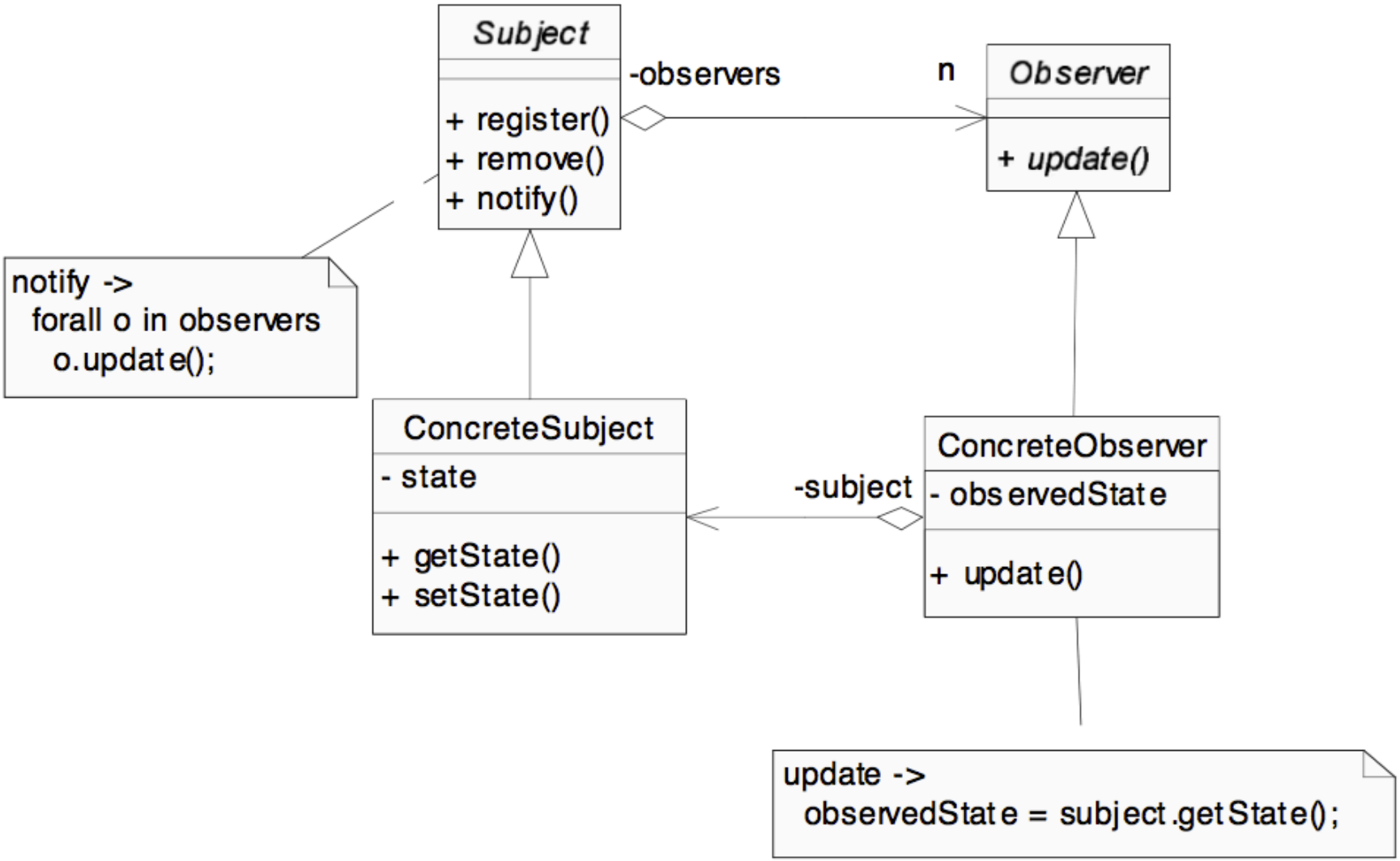
… therefore

- The observers could be “views” on the same object
  - Data on a spreadsheet and their tabular, graphical, … representations
  - Visualisation of the file name in a folder and info box
- The observed object does not know the exact identity of the observers
- Registration and unregistration to the object happens at runtime

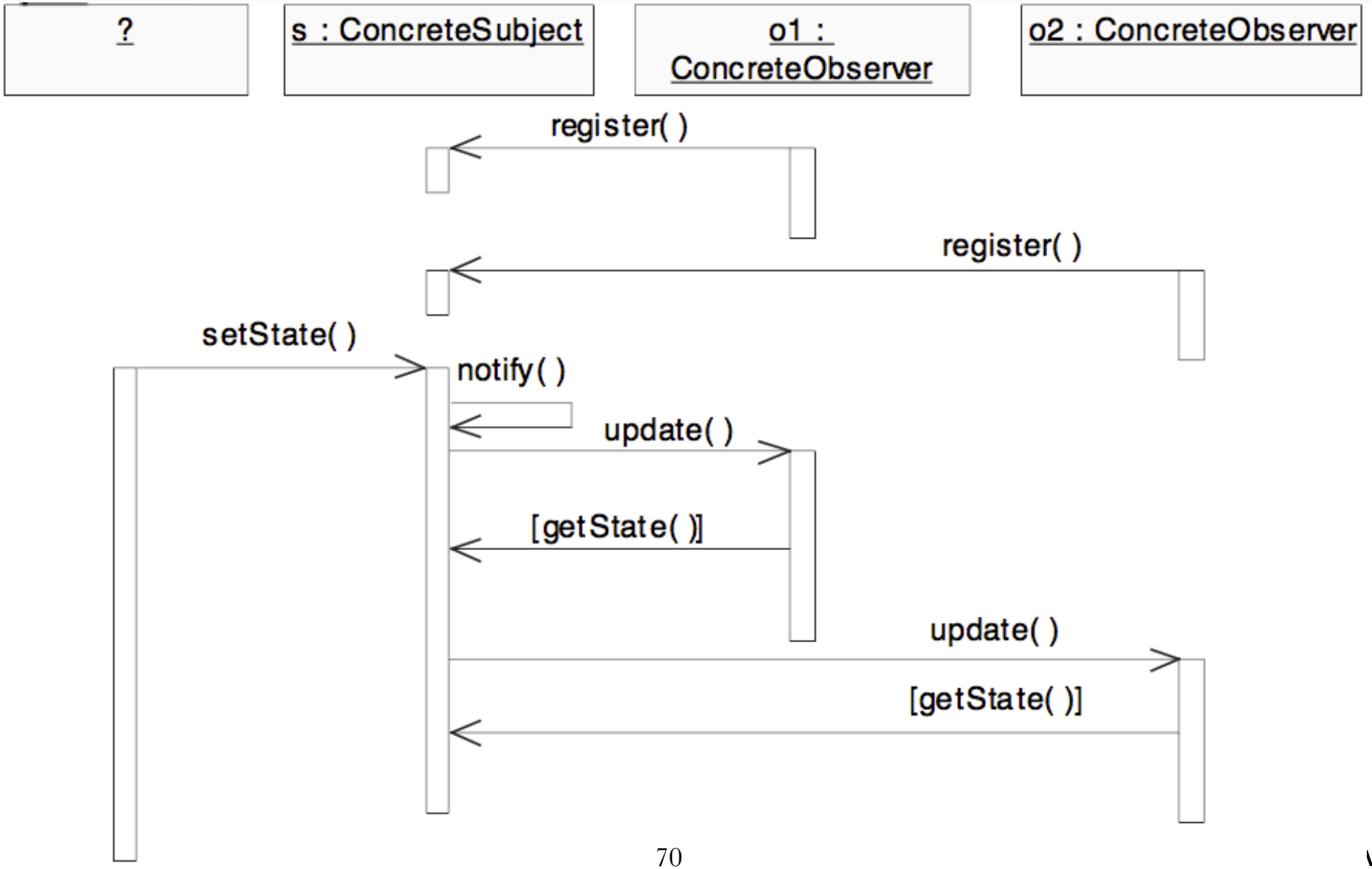
Observer class diagram



Observer class diagram



Observer sequence diagram



Comments

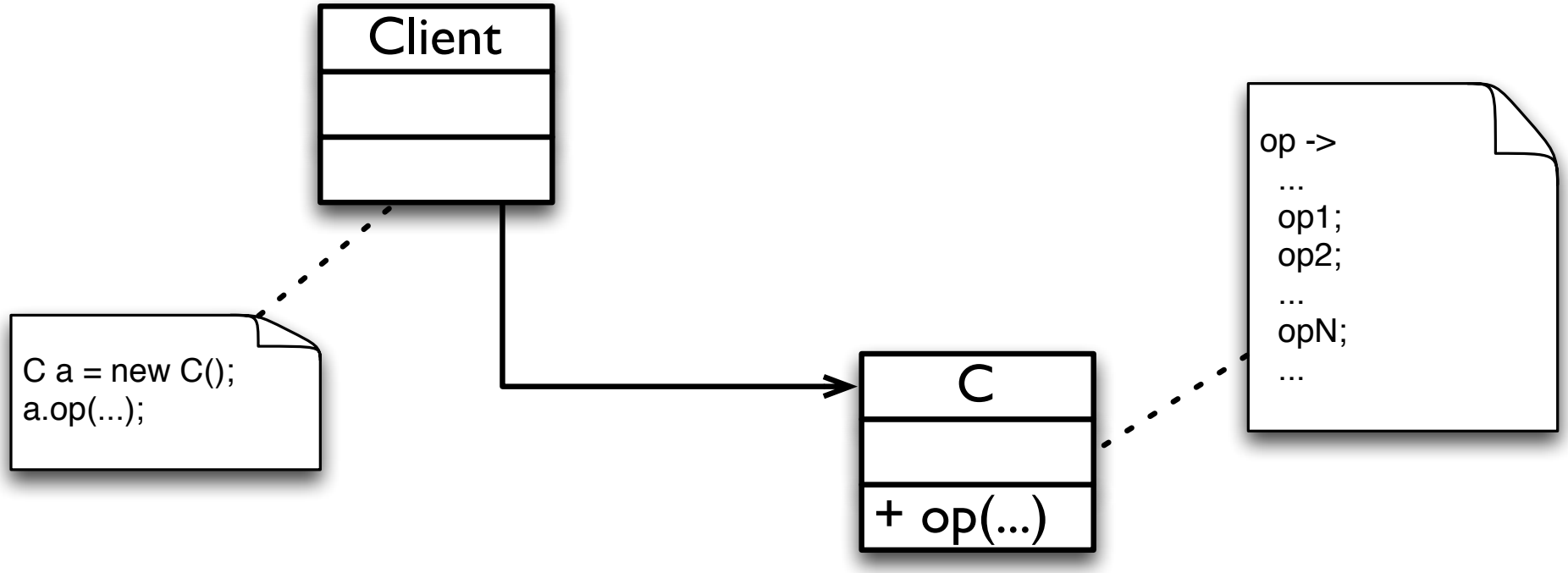
- Who invokes setState()?
  - Anybody
  - … also one of the observers could
- Who invokes notify()?
  - Every method of the object observed that modifies the state
  - … or, the client after having finished a sequence of modifications.

Strategy

- Purpose
  - Define a family of algorithms …
  - … encapsulate them …
  - … and make them interchangeable.
  - Allow the algorithms to vary.

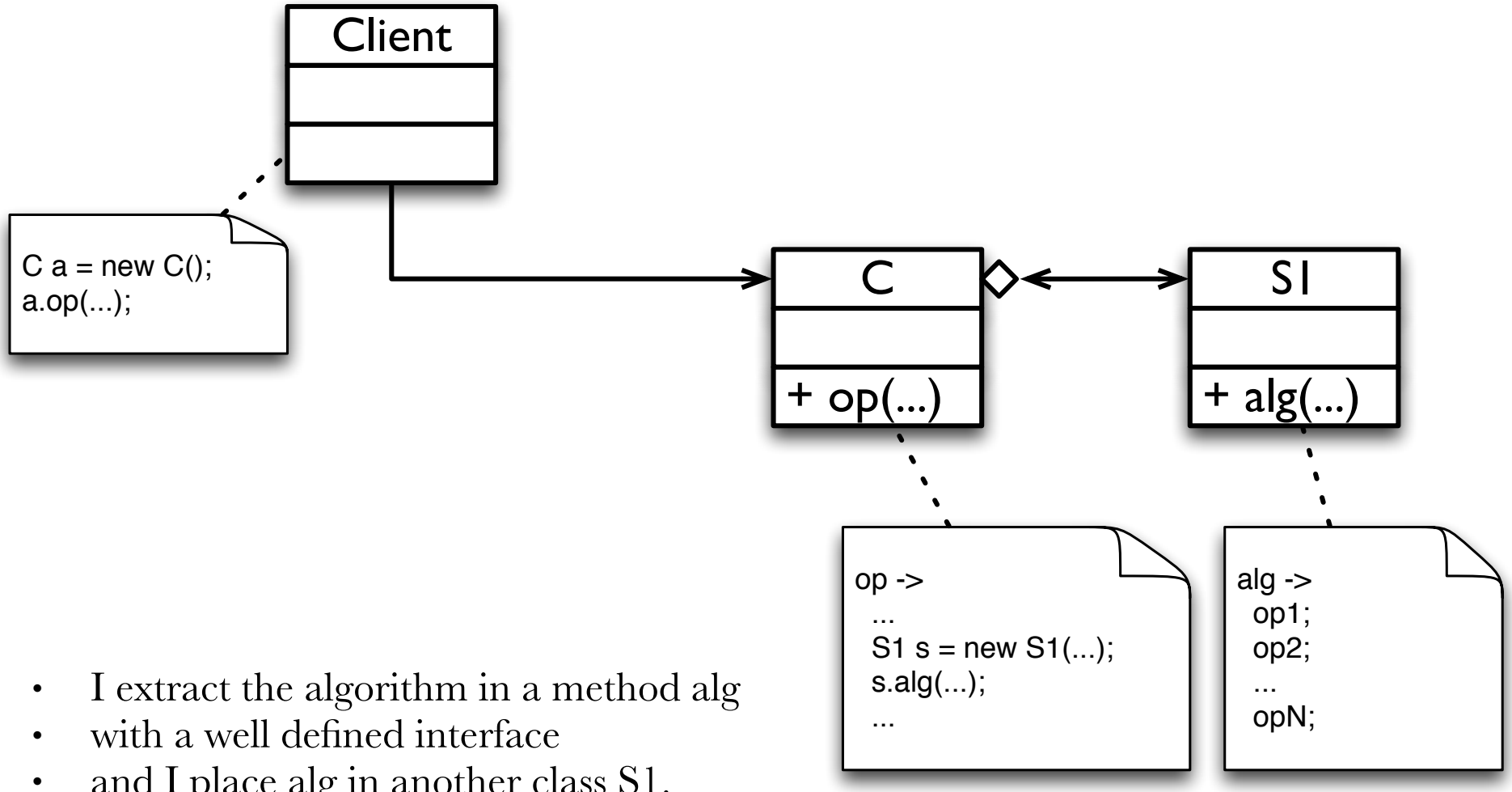
- Differently from the Template method that uses inheritance, Strategy uses composition.

Let's understand Strategy (1/6)



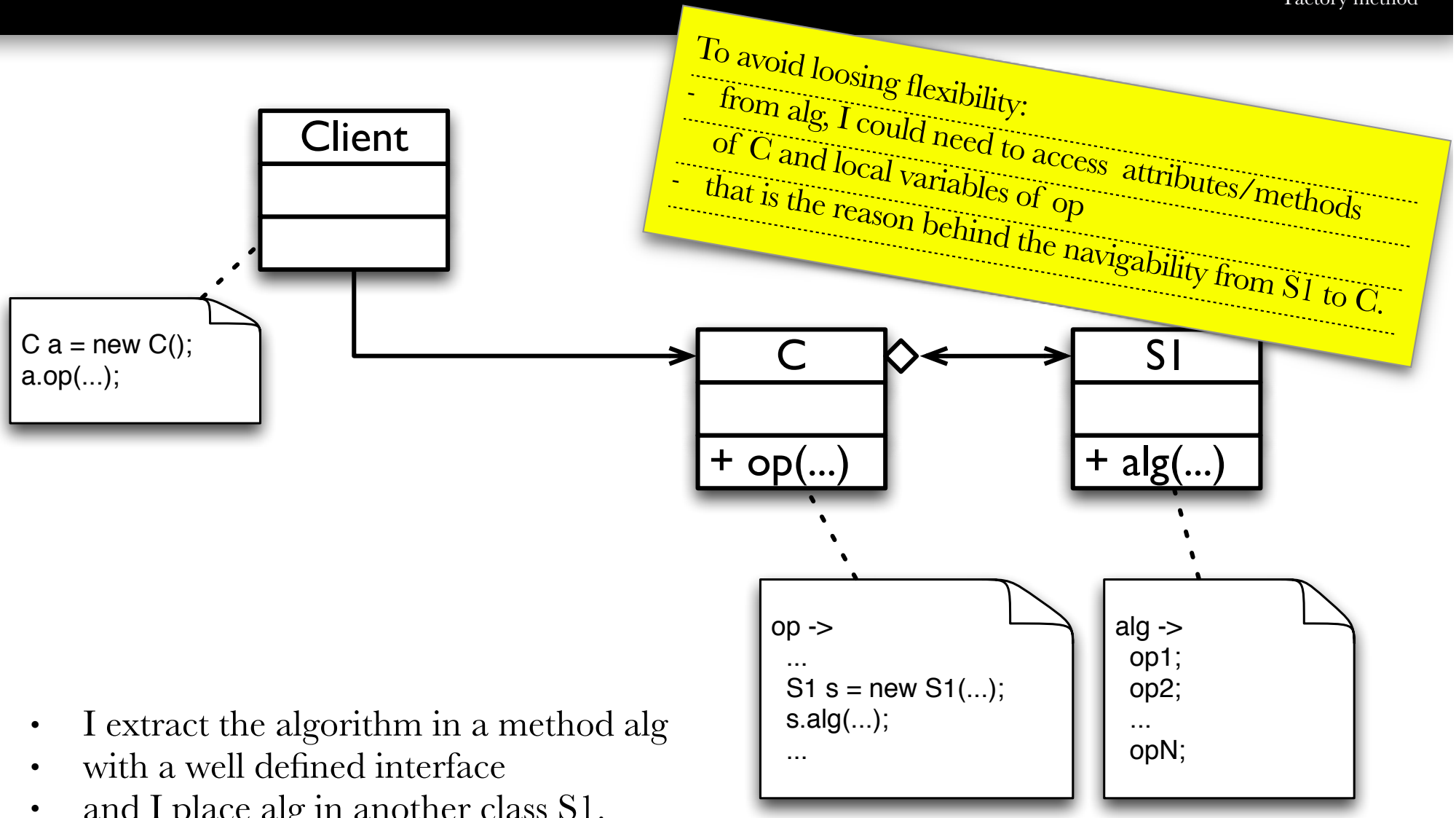
- The algorithm is complex and it is wired in the code of a method op of a class C (used by a Client).

Let's understand Strategy (2/6)



- I extract the algorithm in a method alg
- with a well defined interface
- and I place alg in another class S1.

Let's understand Strategy (2/6)

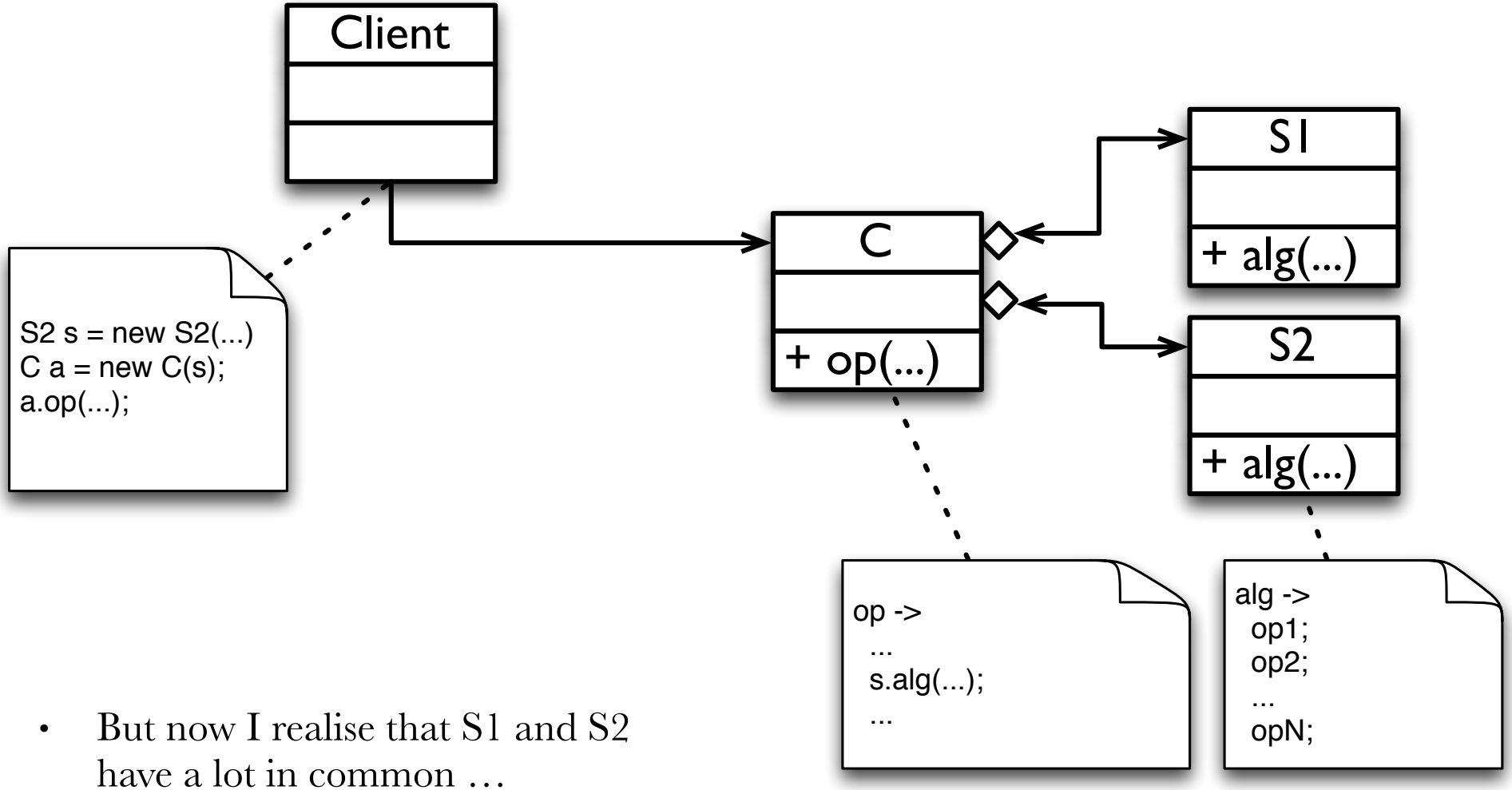


- I extract the algorithm in a method alg
- with a well defined interface
- and I place alg in another class S1.

Let's understand Strategy (3/6)

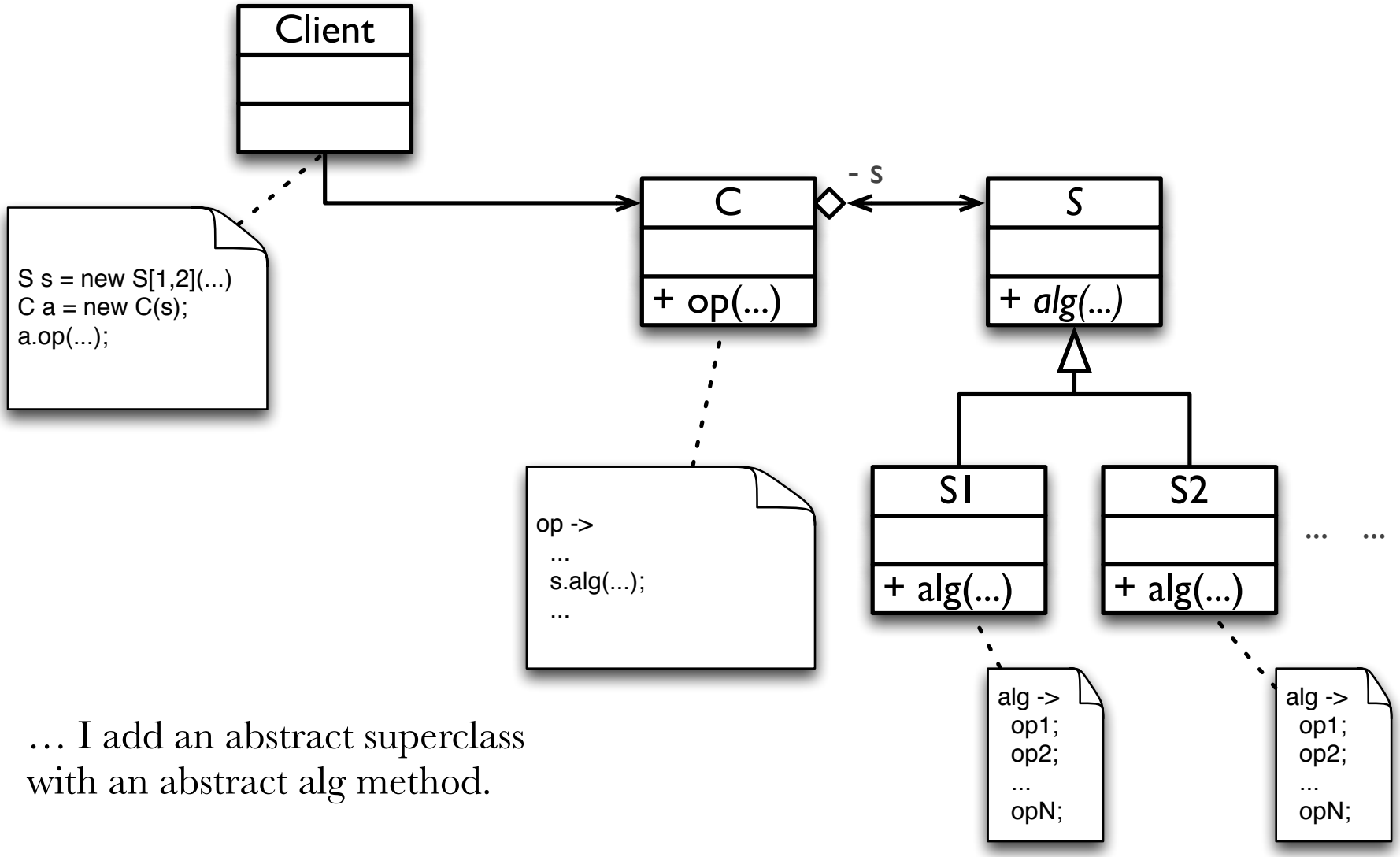
- Additional complications:
  - C could be configured by the Client to use S1 (how??).
  - This would allow me in the future to add S2 with a different implementation of alg with very little effort.

Let's understand Strategy (4/6)



- But now I realise that S1 and S2 have a lot in common …

Let's understand Strategy (5/6)



- … I add an abstract superclass with an abstract alg method.

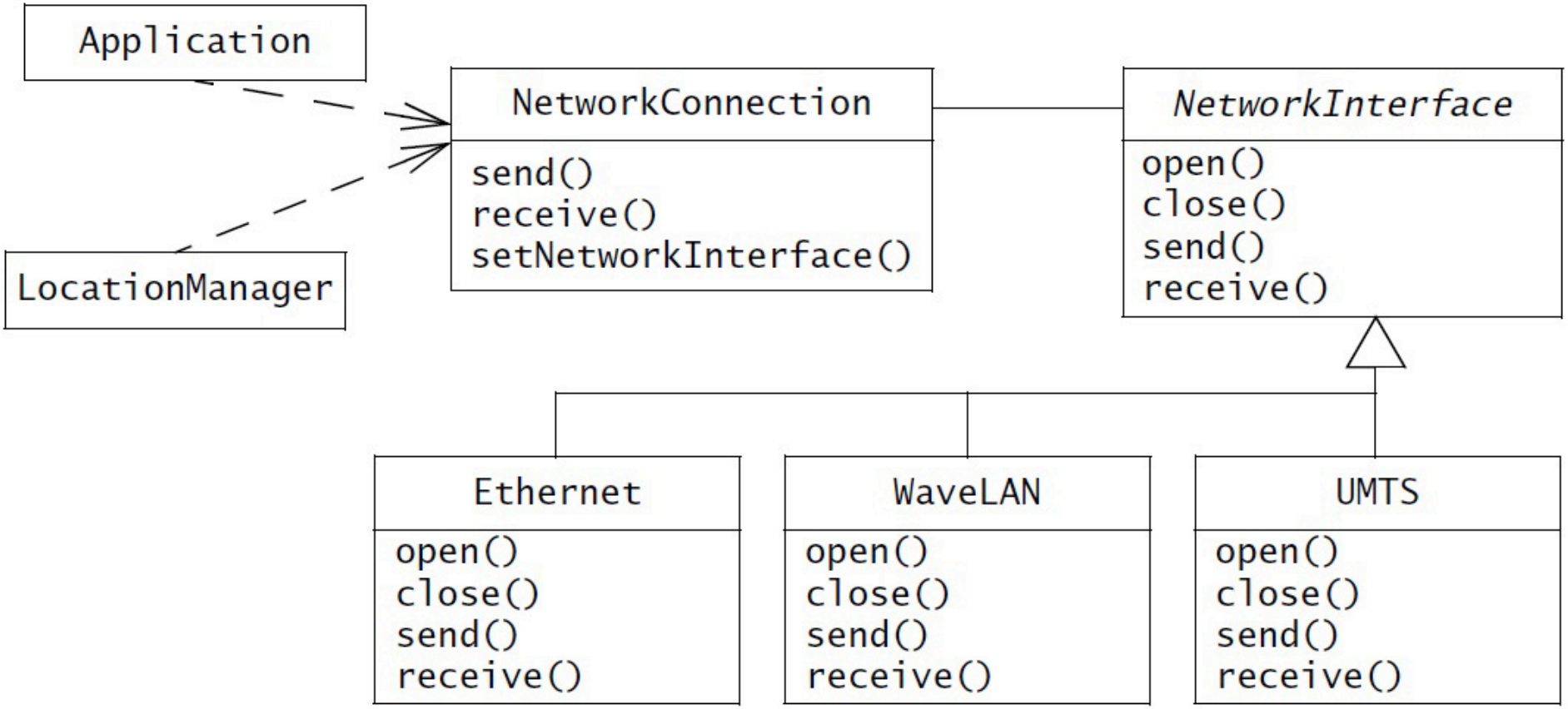
Let's understand Strategy (6/6)

- Using this approach:
  - C does not know about the subclasses and invokes only the base abstract class (remember that C could be configured by the Client)
  - Now – and in the future – I can add other alg in other subclasses
  - With very little modification, the Client can use another algorithm.

Comments

- The Client can use different algorithms with different characteristics (memory usage, speed, etc) …
  - … in a transparent manner.
- The strategy can be selected at run-time (and it would not be the case without the hierarchy generated from C)
- It is called Strategy not Algorithm, …
  - … strategy is a much more generic term
  - … let's see another application.

Another example



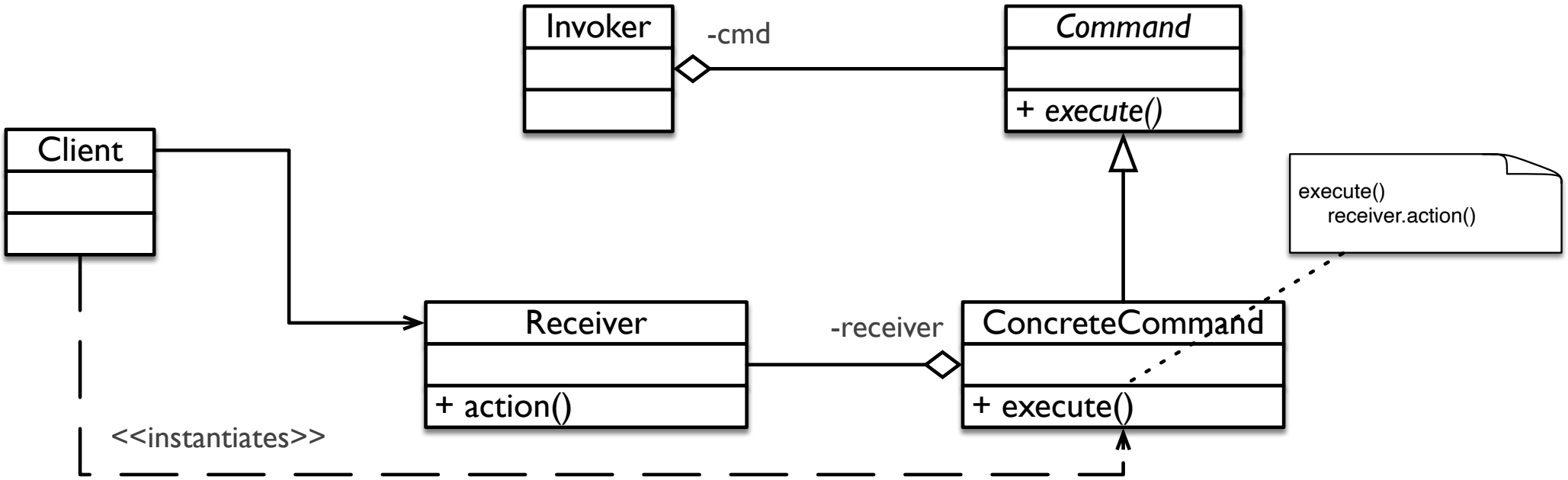
- Dynamic switching of network interfaces depending on context/location.

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

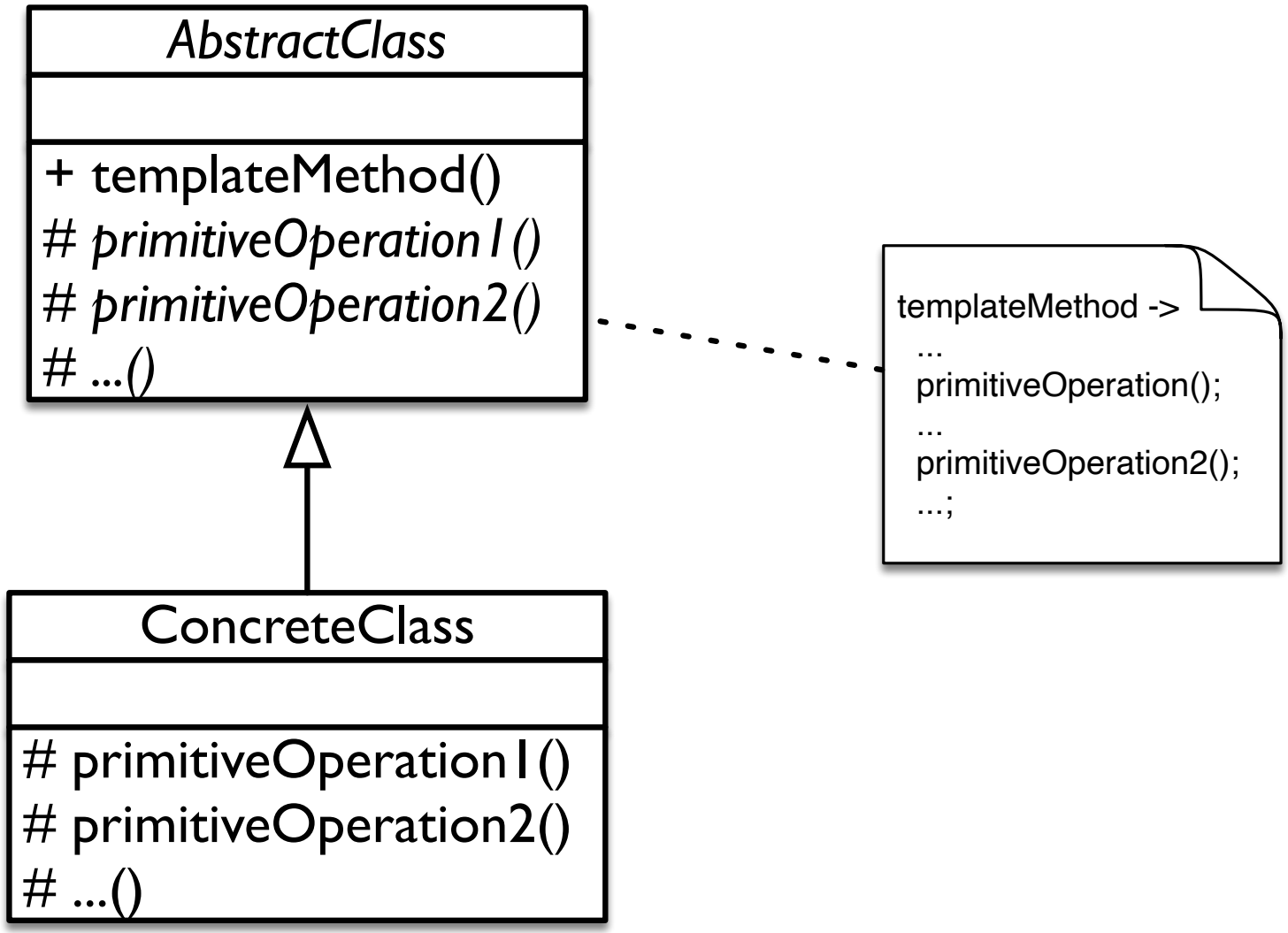
- Command de-couples Invoker (the one invoking the request) from the Receiver (the one executing the request)
- Commands are objects. Therefore:
  - it is possible to compose commands in a single command (Composite pattern)
  - it is easy to add new commands
  - it is possible to keep track of the executed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

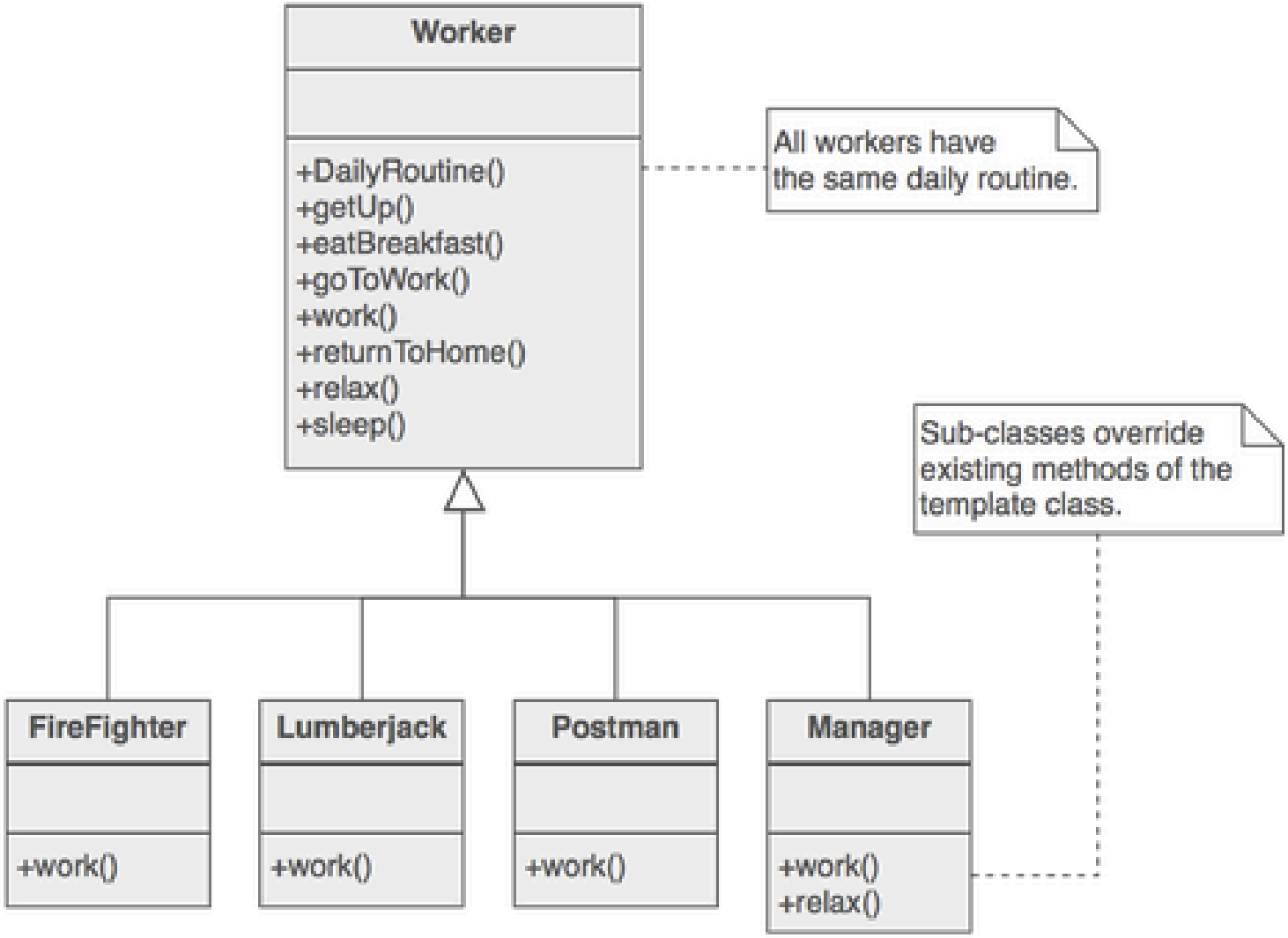
Template method class diagram



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()

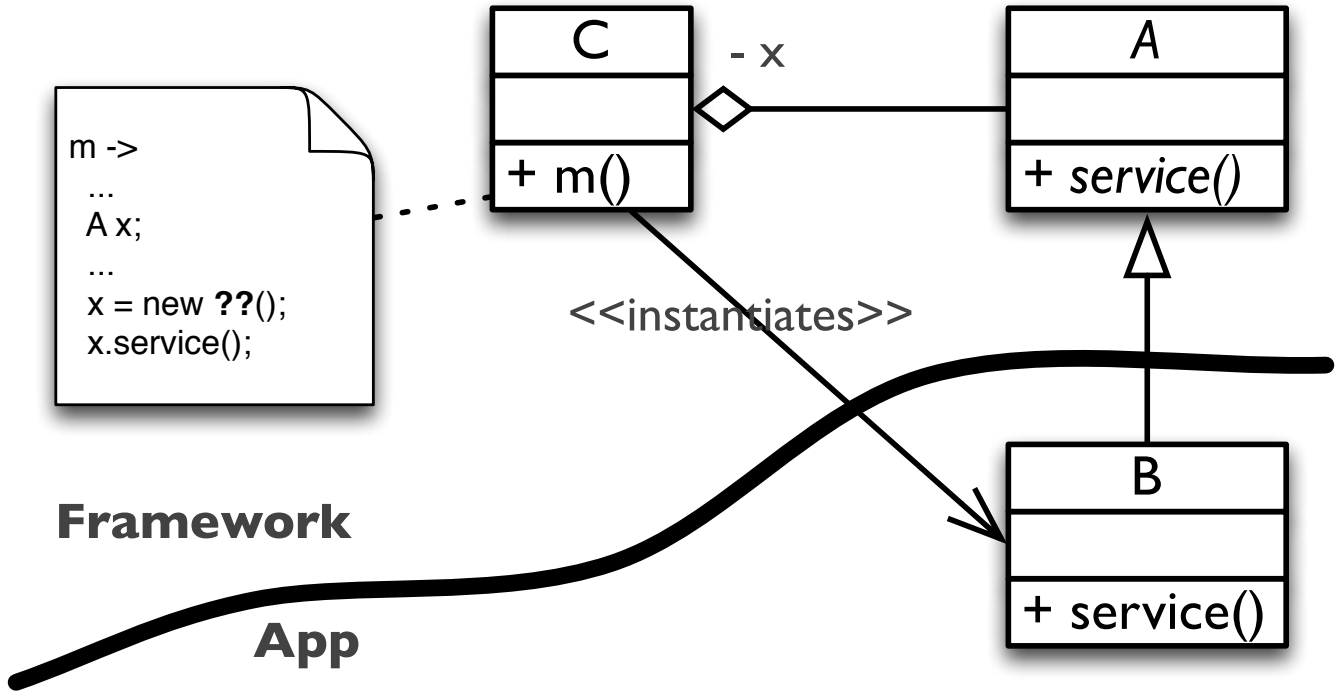
Factory method

- 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

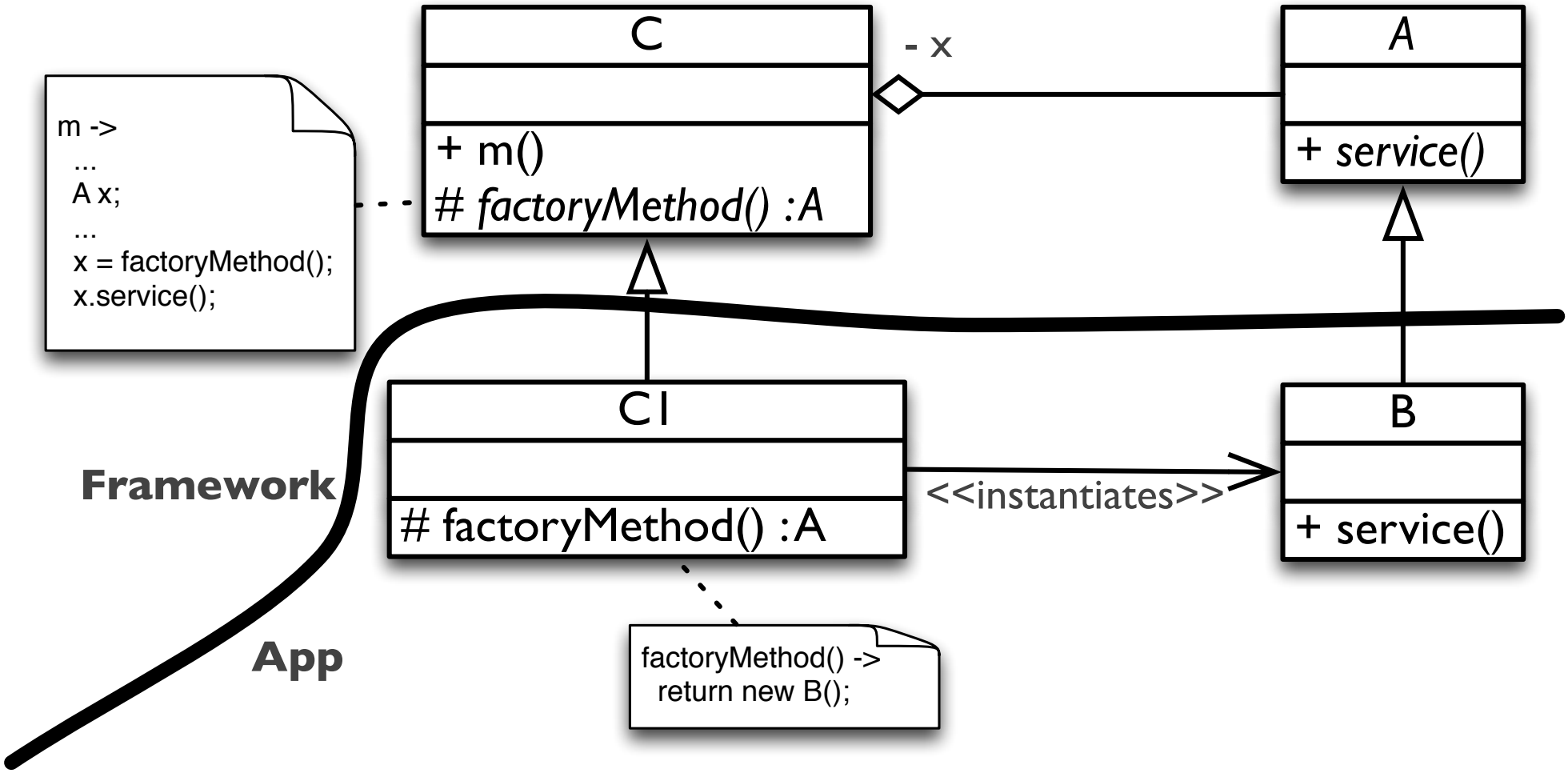


- The framework developer writes a method in a class C …
- … the method needs to create, at a certain precise point, an instance of a subclass B of a base class A …
- … but which subclass B to use is unknown as it is a decision that the developer of the application needs to take, not the framework developer.

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).

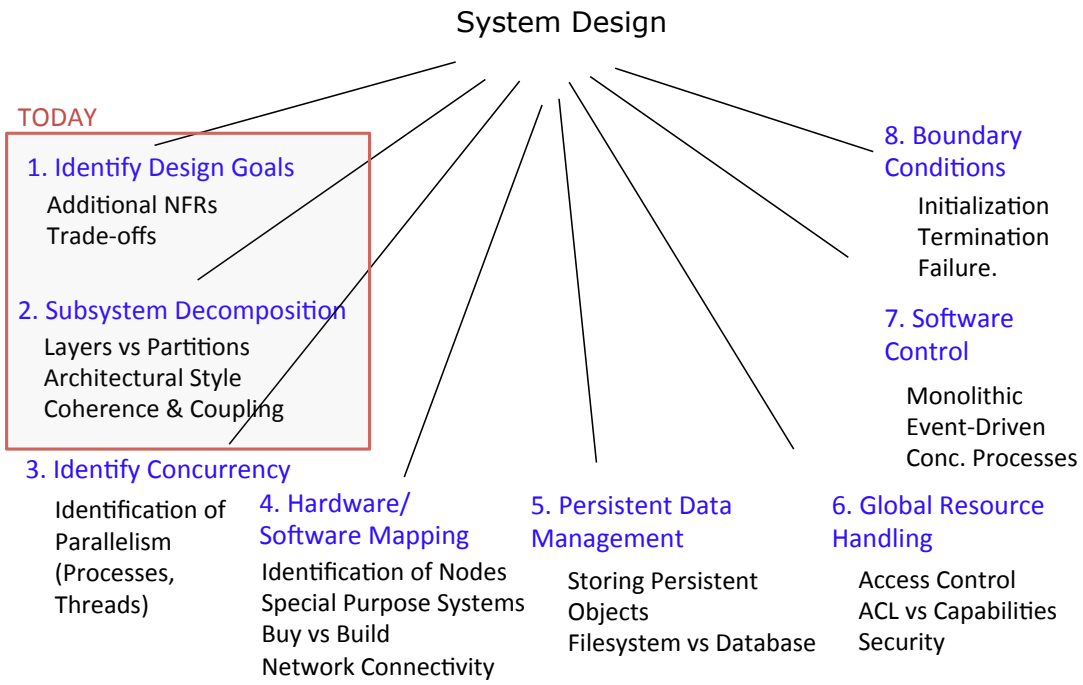
- However, they are different
  - The Template method is the one invoking the abstract methods.
  - The Factory method is the abstract method …
  - … which has to create and return the instance.

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	