



Design patterns

Prof. Catalin Boja, PhD

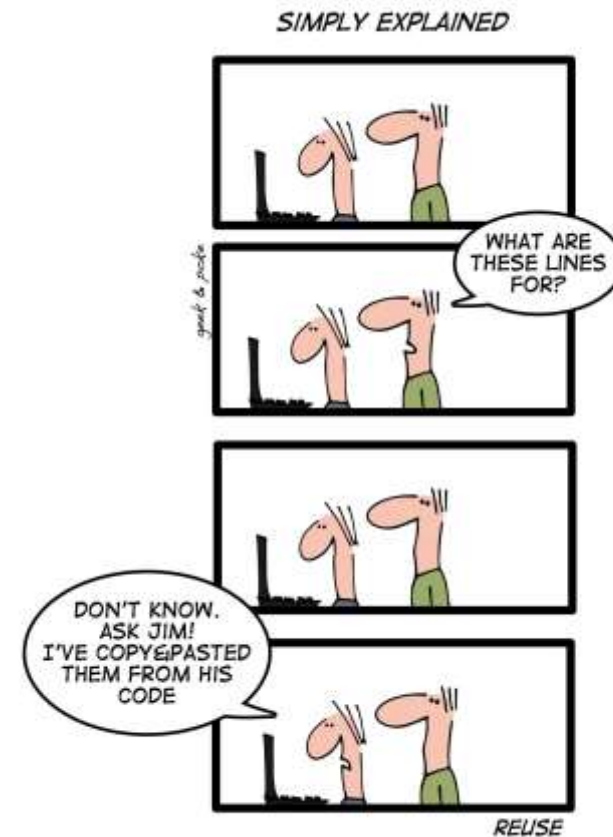
catalin.boja@ie.ase.ro

<http://acs.ase.ro/software-quality-testing>

Source code quality

Principles for writing the code:

- Easy to read / understand - clear
- Easy to modify - structured
- Easy to reuse
- Simple (complexity)
- Easy to test
- Implement patterns for the standard problem



Left: [Simply Explained: Code Reuse](#) 2009-12-03. By Oliver Widder, Webcomics Geek Aad Poke.

Source code quality

Forces that influence it:

- Available time (delivery terms)
- Costs
- The experience of the programmer
- Programmer competences
- Specifications clarity
- Solution complexity
- Change rates for specifications, requirements, team, etc

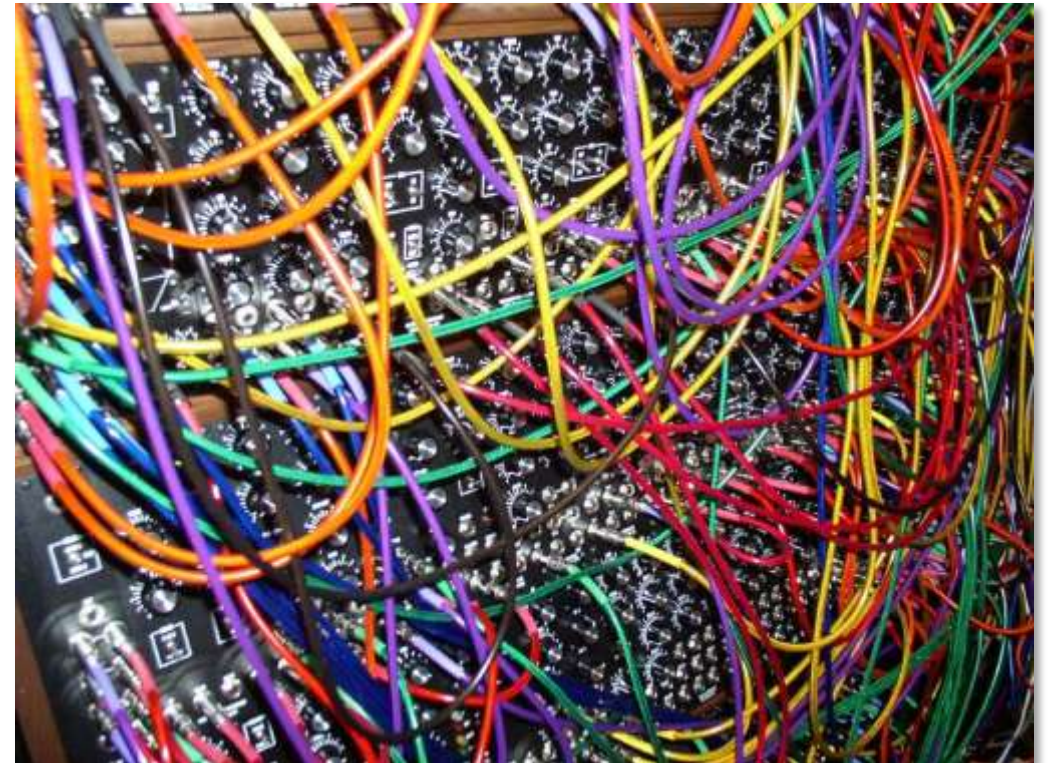


<http://khristianmcfadyen.com/>

Anti-Pattern: Big ball of mud

“A Big Ball of Mud is a haphazardly structured, sprawling, sloppy, duct-tape-and-baling-wire, spaghetti-code jungle.”

Brian Foote and Joseph Yoder, *Big Ball of Mud*,
September 1997



Anti-Pattern: Big ball of mud

Where from ? Why ?

- *Throwaway code* - Temporary (Prototyping) solutions to be replaced / rewritten
- Cut and Paste code
- Adapting code by commenting / deleting other solutions
- Very short or unrealistic deadlines
- Lack of experience
- Lack of standards / procedures



Anti-Pattern: Big ball of mud

How do you avoid it?

- Rewriting the code (**Refactoring**) to an acceptable maturity level
- Use Clean Code Principles
- Design Patterns Implementation



Design-pattern

- A **pattern** is a *reusable* solution for a *standard* problem in a given *context*
- Facilitates the reuse of architectures and software design
- **They are not** data structures



Design-pattern

“A pattern involves a general description of a recurring solution to a recurring problem with various goals and constraints. It identifies more than a solution, it also explains why the solution is needed.”

James Coplien

“... describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice”

Cristopher Alexander

Design-Pattern advantages

- Allow reuse of standard solutions at source / architecture code level
- Allow source / architecture code documentation
- Allow easier understanding of the source code / architecture
- They are universally known concepts - they define a common vocabulary
- They are tested and well documented solutions

Design-Pattern in Software Architectures

Enterprise

System

- OO Architecture

Application

- Subsystem

Macro

- Frameworks

Micro

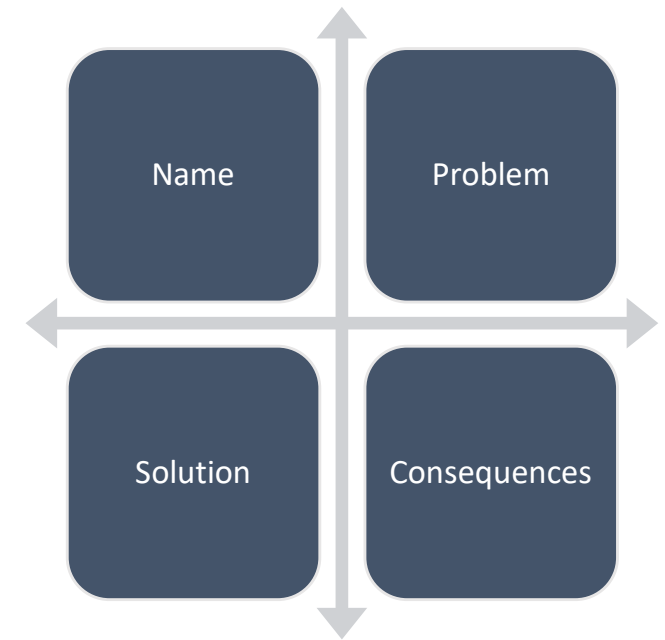
- Design-Patterns

Objects

- OOP

Design-Pattern components

- **Name:**
 - It is part of the vocabulary of a programmer / designer / software architect
 - Identifies the pattern uniquely
- **Problem:**
 - Describes the intended purpose
 - Defines the context
 - Determines when the pattern is applicable
- **Solution**
 - UML diagram, pseudo-code describing the elements
- **Consequences**
 - Results
 - Advantages and disadvantages



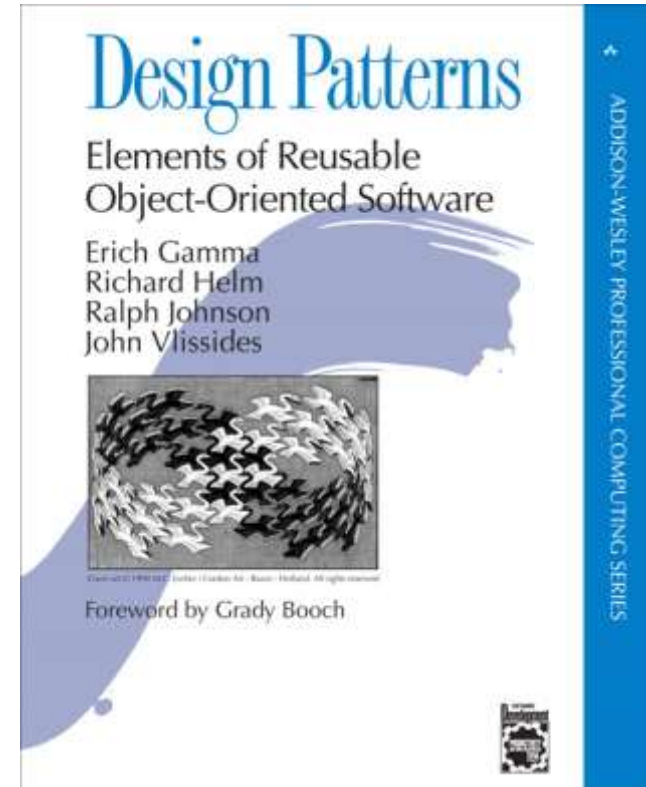
Istoric Design-Pattern

- 1970 – The first models related to the Windows and Desktop concepts (Smalltalk, Xerox Parc, Palo Alto)
- 1978 – MVC pattern (Goldberg and Reenskaug, Smalltalk, Xerox Parc)
- 1987 - Kent Beck and Ward Cunningham, “*Using Pattern Languages for Object-Oriented Programs*”, OOPSLA-87 Workshop
- 1991 - Erich Gamma, an idea for a Ph.D. thesis about patterns
- 1993 - E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Abstraction and Reuse of Object-Oriented Design. ECOOP 97 LNCS 707, Springer, 1993

Istoric Design-Pattern

Erich Gamma, Richard Helm,
Ralph Johnson & John Vlissides
(Addison-Wesley, 1995) - *Design
Patterns*

- The Gang of Four (GOF)
- The book describes 23 patterns -
problems and solutions that can
be applied in many scenarios
- The most popular computer
software book

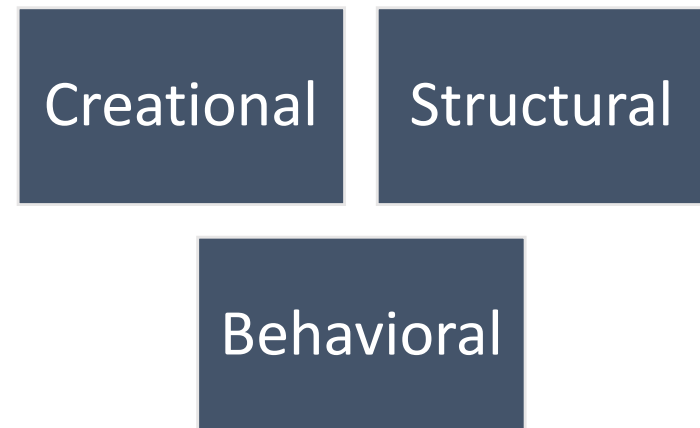


Design-Patterns implementation

- *Observer* in Java AWT si Swing for callbacks
- *Iterator* in C++ STL and Java Collections
- *Façade* in many Open-Source libraries to hide the complexity of internal routines
- *Bridge* si *Proxy* in frameworks for distributed applications
- *Singleton* in Hybernate and NHybernate

Design-Pattern types

- Creational
 - Initializing and configuring classes and objects
- Structural
 - Composition of classes and objects
 - Disconnect interfaces and classes
- Behavioral
 - Responsibility distribution
 - Interaction between classes and objects



Design-Pattern types

Creational

- Factory Method
- Abstract Factory
- Builder
- Prototype
- Singleton

Structural

- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

Behavioral

- Interpreter
- Chain of Responsibility
- Command
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Visitor
- Template

Creational Design-Patterns

- **Abstract Factory**
 - Pattern for creating objects from a large family in a specific context
- **Builder**
 - Pattern for structural (incremental) creation of complex objects (with a lot of attributes)
- **Factory Method**
 - Pattern that defines a method for creating objects from the same family (interface) in subclasses
- **Prototype**
 - Pattern for cloning new instances (clones) of an existing prototype
- **Singleton**
 - Pattern for single instance creation (unique)

Structural Design-Patterns

- **Adapter:**
 - Adapts the server / service interface to the client
- **Bridge:**
 - Decouples the abstract implementation model
- **Composite:**
 - Aggregation of several similar objects
- **Decorator:**
 - Expands an object transparently
- **Facade:**
 - Simplifies the interface of a module / subsystem
- **Flyweight:**
 - Sharing memory between similar objects.
- **Proxy:**
 - Interface to other objects / resources

Behavioral Design-Patterns

- **Chain of Responsibility:**
 - Manages the handling of an event by multiple solution providers
- **Command:**
 - Request or Action is first-class object, hence re-storable
- **Iterator:**
 - Manages scrolling through a collection of items
- **Interpreter:**
 - Interpreter for a language with a simple grammar
- **Mediator:**
 - Coordinates the interaction between multiple associates
- **Memento:**
 - It saves and restores the status of an object

Behavioral Design-Patterns

- **Observer:**
 - Defines a handler for different events
- **State:**
 - Manages objects whose behavior depends on their condition
- **Strategy:**
 - Encapsulates different algorithms
- **Template Method:**
 - Incorporates an algorithm whose steps depend on a derived class
- **Visitor:**
 - Describes methods that can be applied to a non-homogeneous structure

Creational Design-Patterns

Singleton, Abstract Factory, Factory Method, Simple Factory, Builder,
Prototype

SINGLETON Pattern

Creational Design-Patterns

SINGLETON- Problem

- You want to create a single instance for a class to manage a resource / event in a centralized manner;
- The solution is based on the existence of a single (unique) instance that can be created only once but can be referenced multiple times;
- Provides a single point of access, globally visible, to the unique object
- Examples: databases or other resources connection management; single logging mechanism; events manager; visual resources manager; configuration manager.

SINGLETON - Diagrama

```
public class Singleton {  
    private static Singleton instance = null;  
    private Singleton() { }  
  
    public static synchronized Singleton  
    getInstance() {  
        if (instance == null) {  
            instance = new Singleton ();  
        }  
        return instance;  
    }  
}
```



SINGLETON - Components

- **Singleton ()**
 - private constructor (callable only inside the class)
- **private static Singleton instance**
 - a class type static and private attribute that will reference the unique instance
- **Singleton getInstance ()**
 - public method to give access to the unique object
 - the unique object is created when the method is called for the first time

SINGLETON – Other implementations

- Version 1 – static & private attribute
- Version 2 – static & public constant attribute
- Version 3 – using an *enum* (Joshua Bloch in *Effective Java, 2nd Edition*)
- Version 4 – *Singleton collection* or *Singleton registry* – multiple unique objects are managed using a collection

SINGLETON – Advantages and Disadvantages

Advantages:

- Centralized management of resources through a single object
- Strict control of instantiations of a class – only once
- Does not allow duplication of objects
- Easy to implement
- ***Lazy instantiation*** – the object is created when is needed

Disadvantages:

- In multi-threading can generate synchronization or cooperation problems if the singleton is shared
- Can generate a *bottleneck* that will affect the app performance

SINGLETON – Scenarios

- Unique connection to the database;
- Unique management of configuration files/file;
- Unique management of preferences on the Android platform (SharedPreferences) or of application settings in a broader context;
- Unique management of network connections;
- Centralized management of access to certain resources used by the software solution;
- Single management of *expensive* objects (costly, based on the time and resources needed to create) that must have a unique instance, used several times.

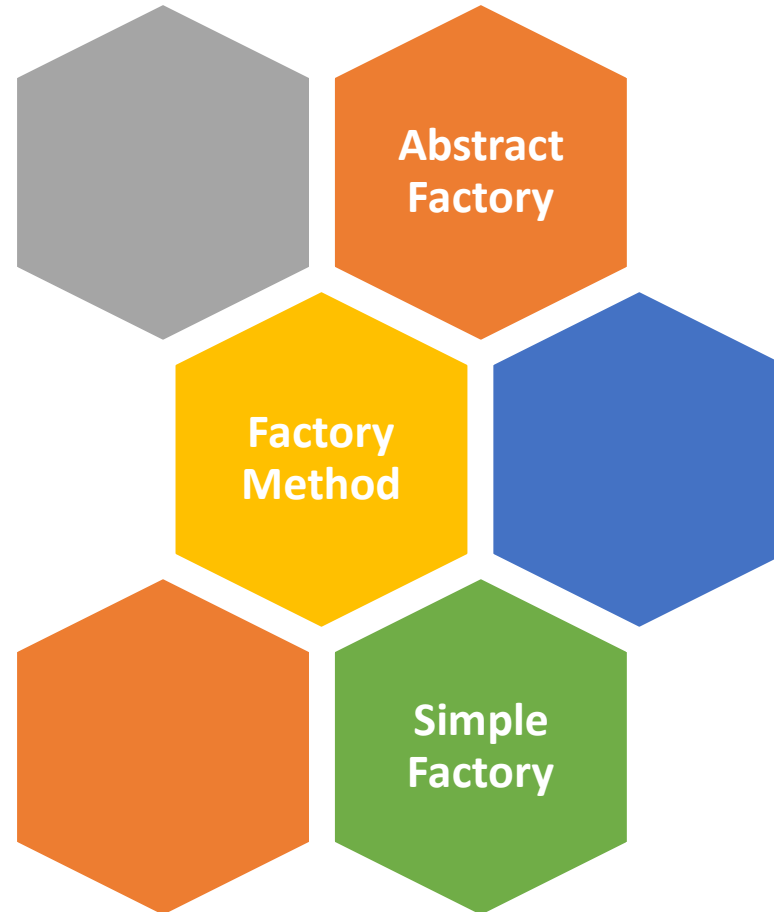
FACTORY Pattern

Creational Design-Patterns

FACTORY - Problem

- implementation of a centralized mechanism for creating objects; the process is transparent to the client; using the public interface client knows how to create objects but does not know how this is implemented;
- the solution can be extended by adding new types of concrete objects without affecting existing code;
- the creation process complexity is hidden to the customer;
- items are referenced by a common interface; concrete classes are a family of objects defined around a common interface;
- removes client code dependence on methods/classes that create the objects;

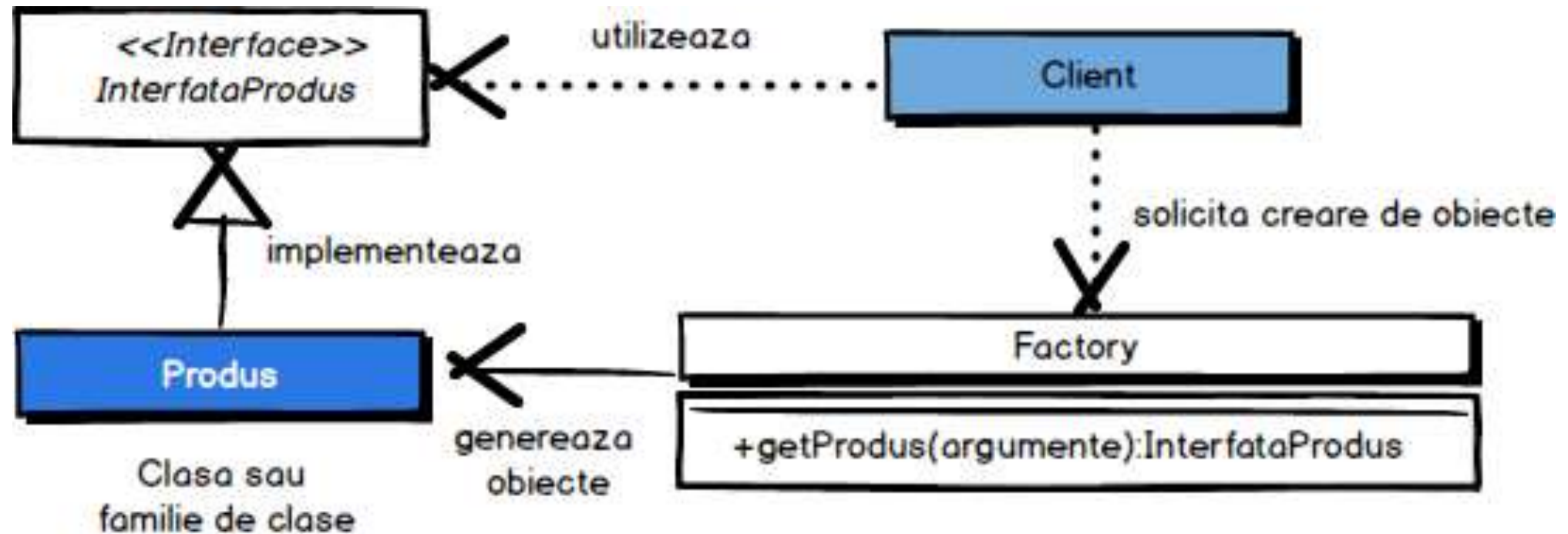
FACTORY



SIMPLE FACTORY

- particular case of the GoF *Factory pattern*
- one of the most used in production due to its simplicity of implementation and its benefits
- is not part of GoF; it's a natural derivation based on best practices

SIMPLE FACTORY - Diagram



SIMPLE FACTORY - Components

InterfataProodus

- abstract interface for *Proodus* type objects;

Proodus

- concrete classes that implement the interface; the objects will be generated by the Factory;

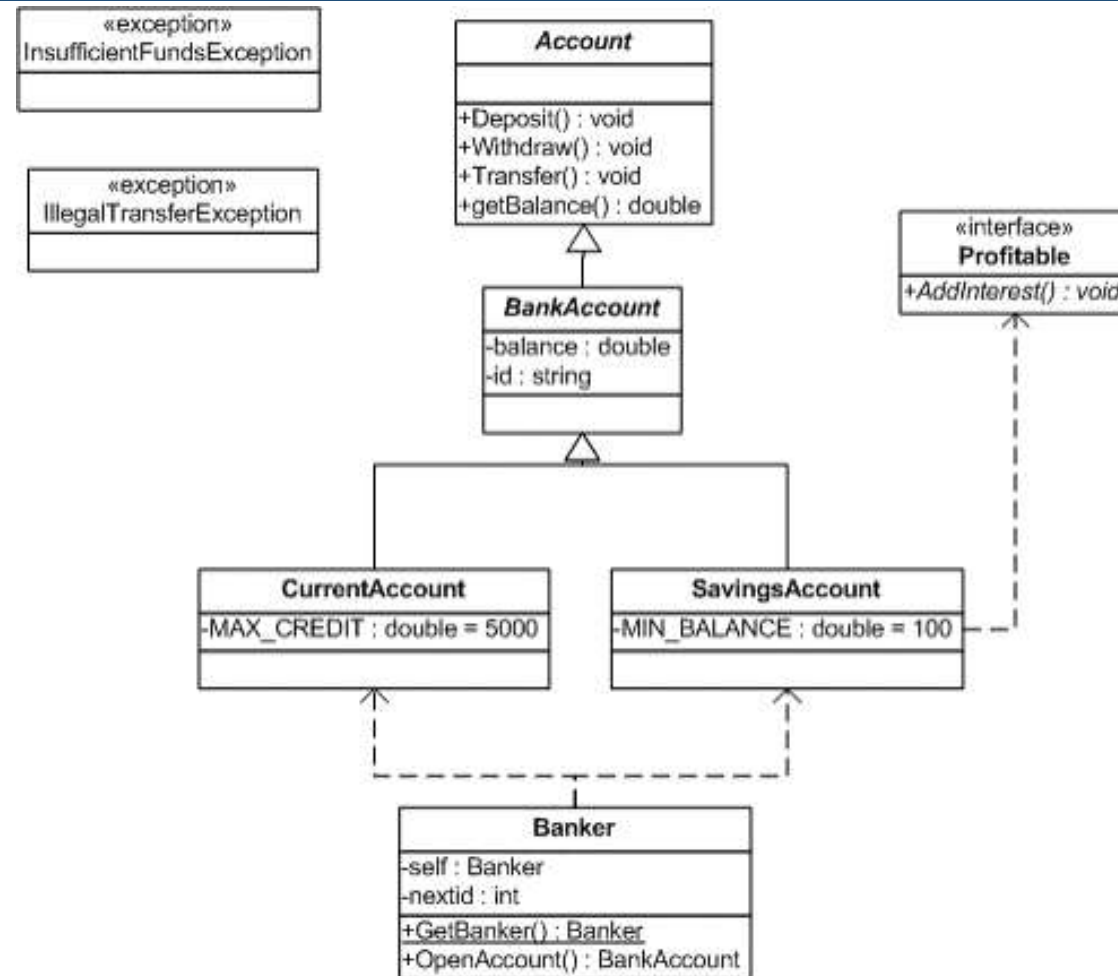
Factory

- class that implements the creation process for *InterfaceProodus* interface type objects;

Client

- a different class/method that uses the *Factory* interface to build new objects;

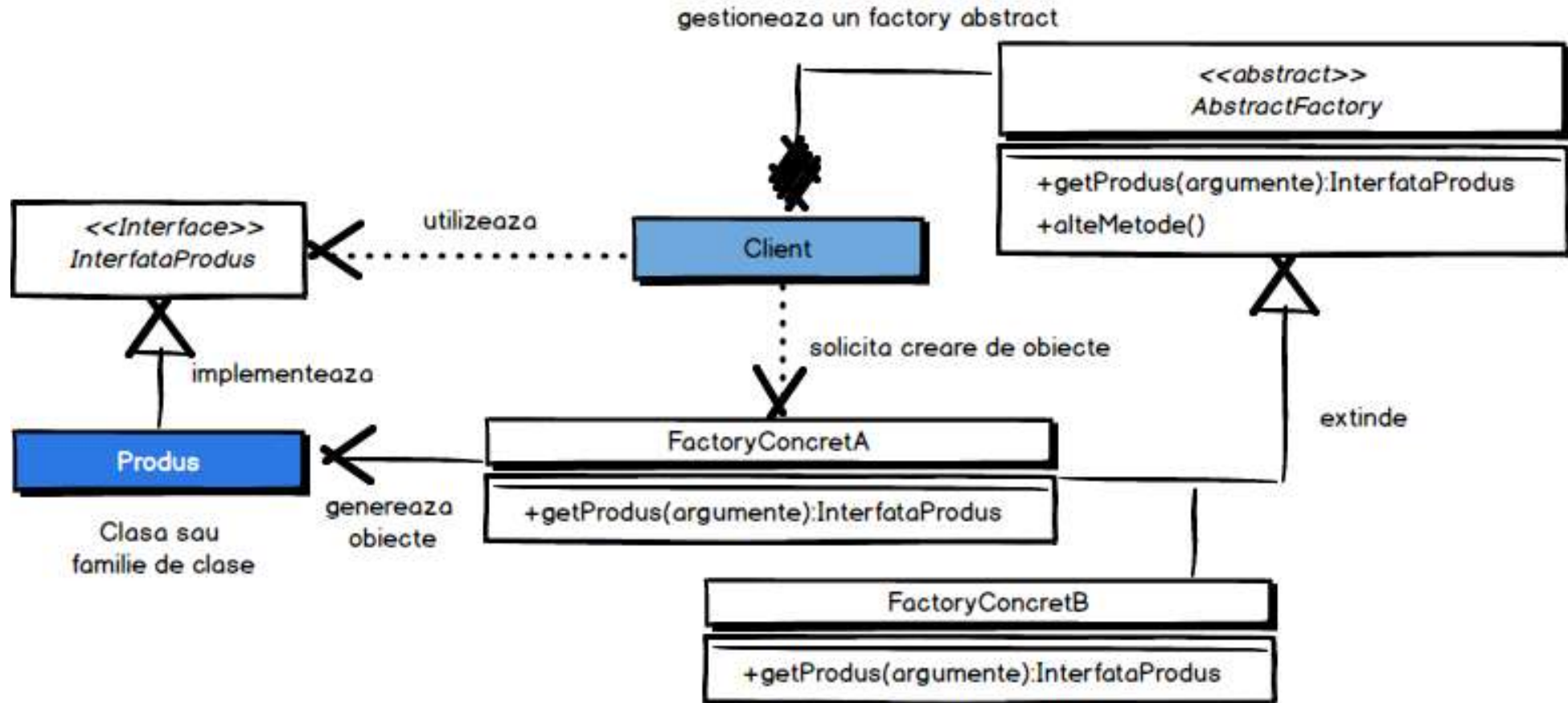
Example of SINGLETON and SIMPLE FACTORY



FACTORY METHOD Pattern (Virtual Constructor)

Creational Design-Patterns

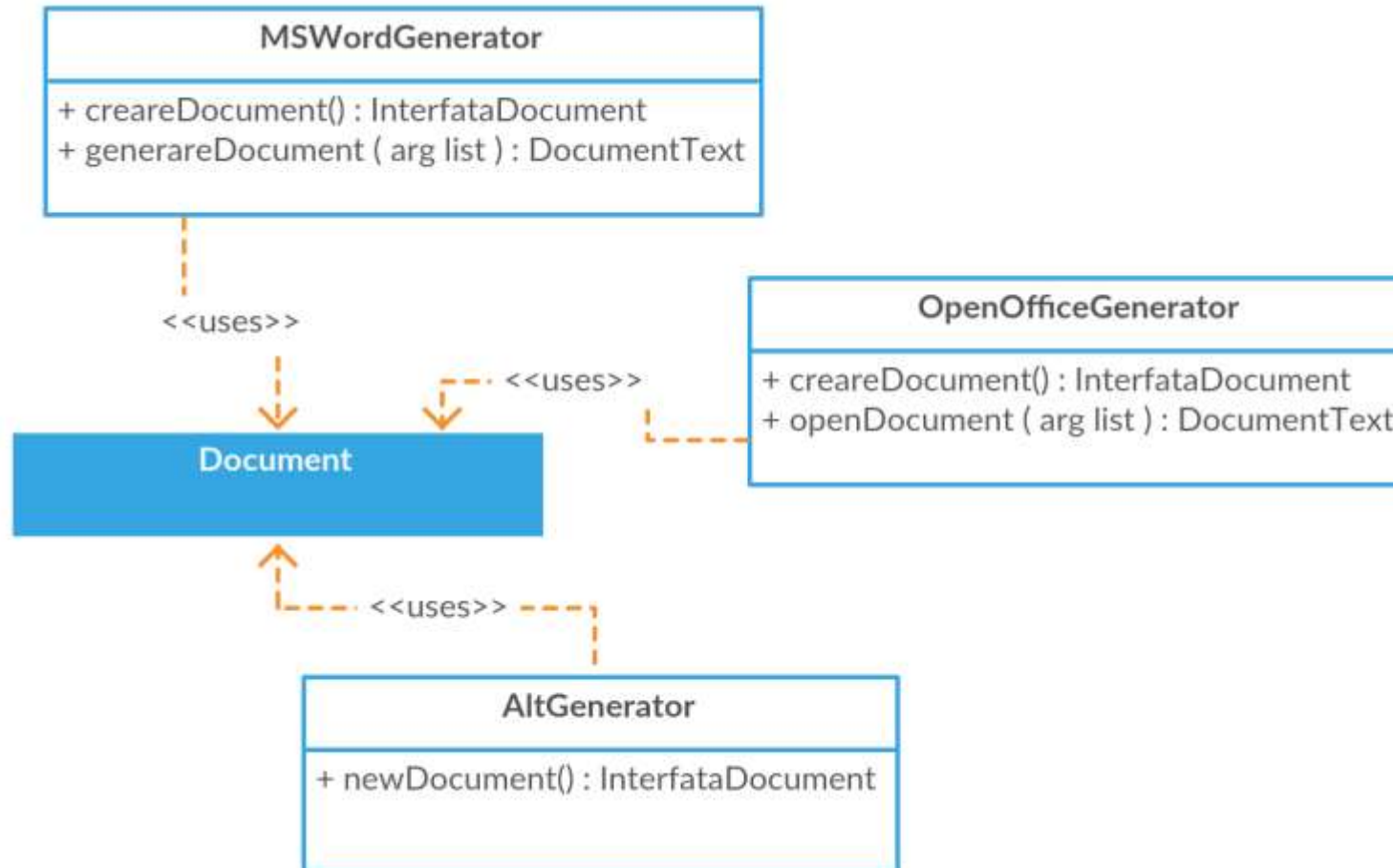
FACTORY METHOD - Diagram



FACTORY METHOD - Components

- **InterfataProbus**
 - generic interface that defines the types of objects that can be created
- **Probus**
 - concrete class that defines the type of objects that can be created
- **Factory**
 - abstract class that defines the interface of an object family generator
- **FacgtoryConcretA, FactoryConcretB ...**
 - concrete class that implements the objects generator

FACTORY METHOD - Example



FACTORY METHOD – Advantages and Disadvantages

Advantages:

- All created objects have a common interface
- Strict control of instantiations - objects are not created directly using constructor but by the factory method
- Different types of objects are managed unified by the common interface - new types of the same family can be added without costly changes
- Easy to implement
- Can be generated new objects belonging to the same family (have the common interface)
- Implements the ***Dependency Inversion*** principle

Disadvantages:

- You can't generate "new" objects (as with a different interface)
- The constructors are private – blocks extending those classes

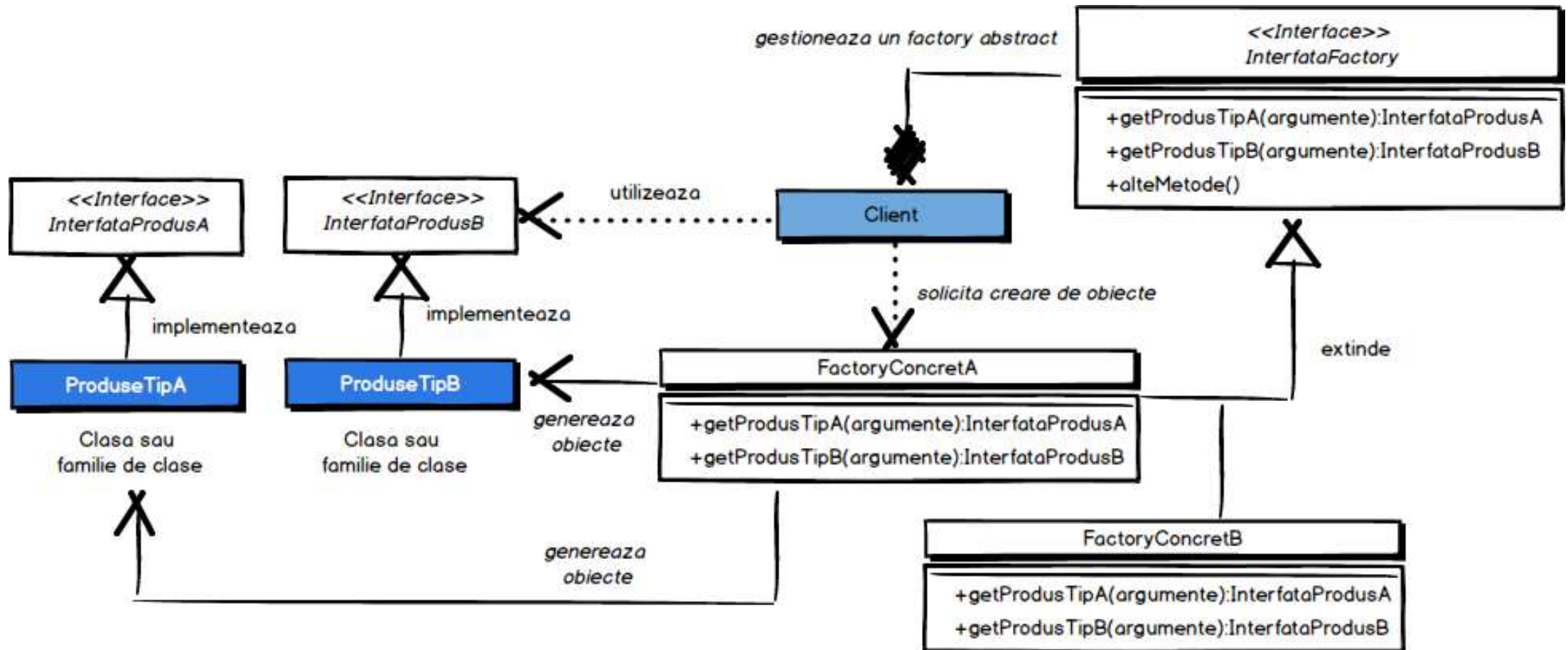
ABSTRACT FACTORY Pattern

Creational Design-Patterns

ABSTRACT FACTORY- Problem

- We want to implement a mechanism by which the creation of objects is transparent to the client
The solution can be extended by adding new concrete types of objects without affecting the written code
The complexity of creating objects is hidden from the client. He knows how to create them but he does not know how they are actually created
Object creation is disconnected from the solution and the generator can be replaced without much effort
Objects are referenced through a common interface and not directly. They form a family of objects around the common interface

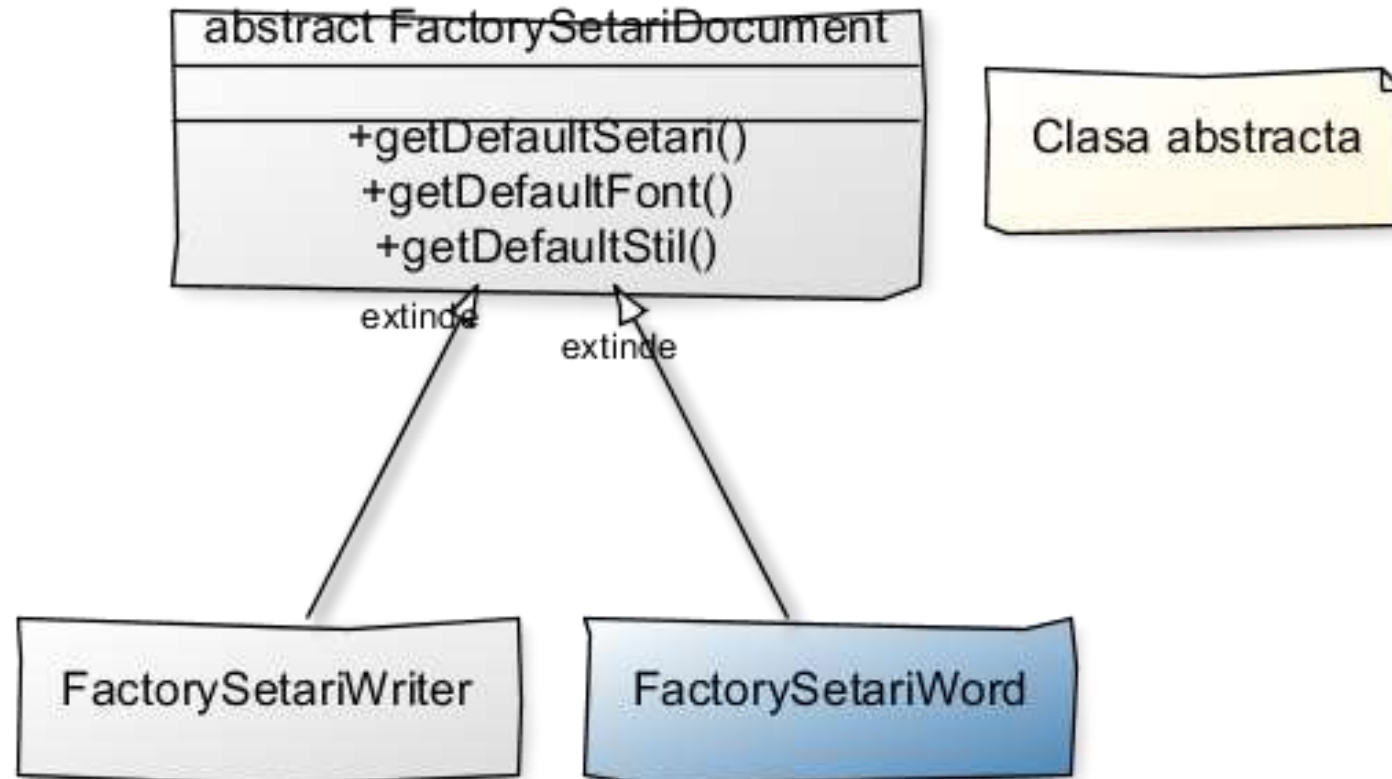
ABSTRACT FACTORY - Diagram



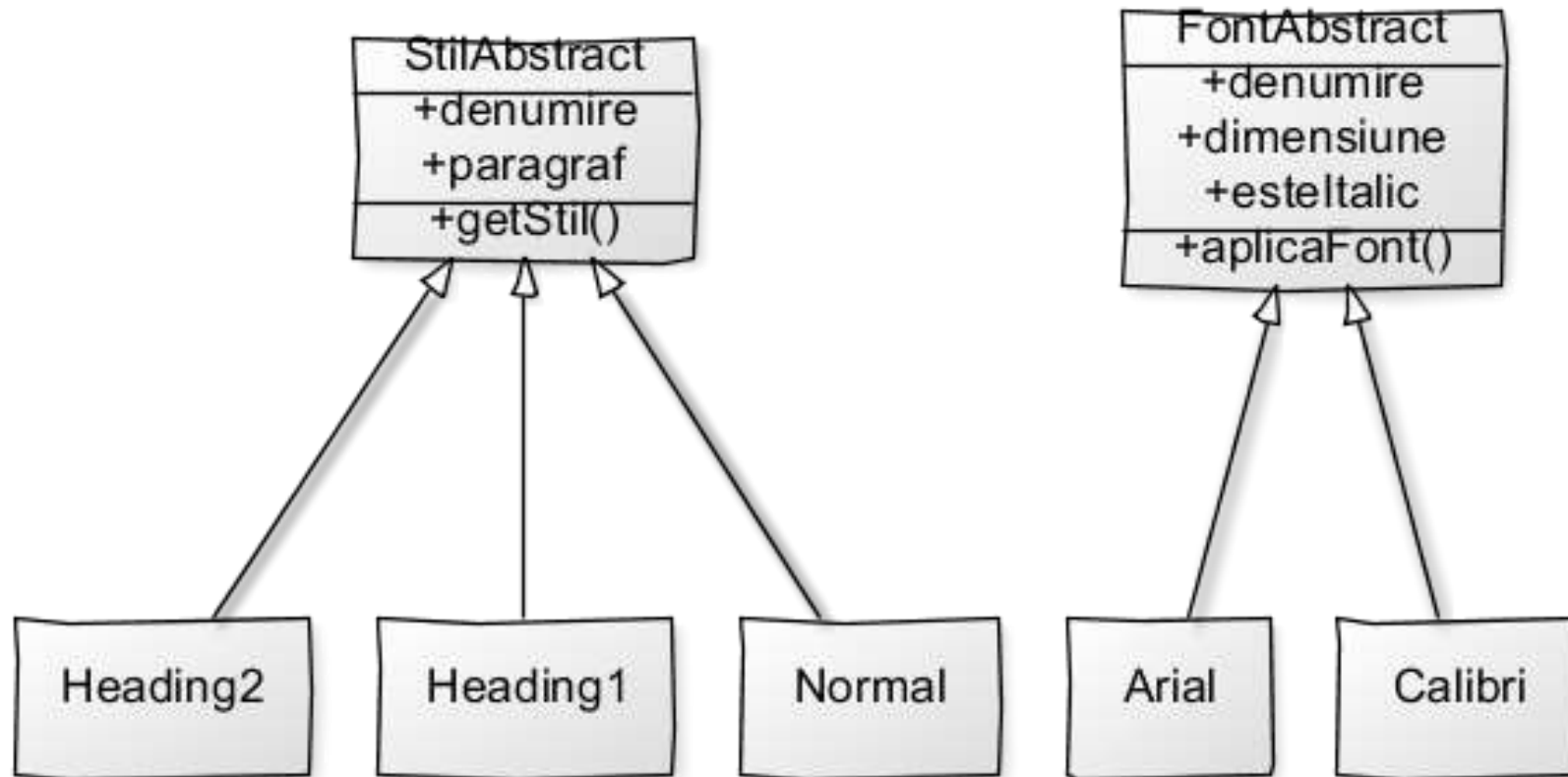
ABSTRACT FACTORY - Componente

- **InterfataFactory**
 - interface that defines abstract methods for creating instances
- **InterfataProdusA/InterfataProdusB**
 - interfaces that define the abstract types of objects that can be created
- **FactoryConcretA/FactoryConcretB**
 - concrete classes that implement the interface and methods by which objects of the type *Product* are created
- **ProduseTipA, ProduseTipB, ...**
 - concrete classes that define the different types of objects that can be created

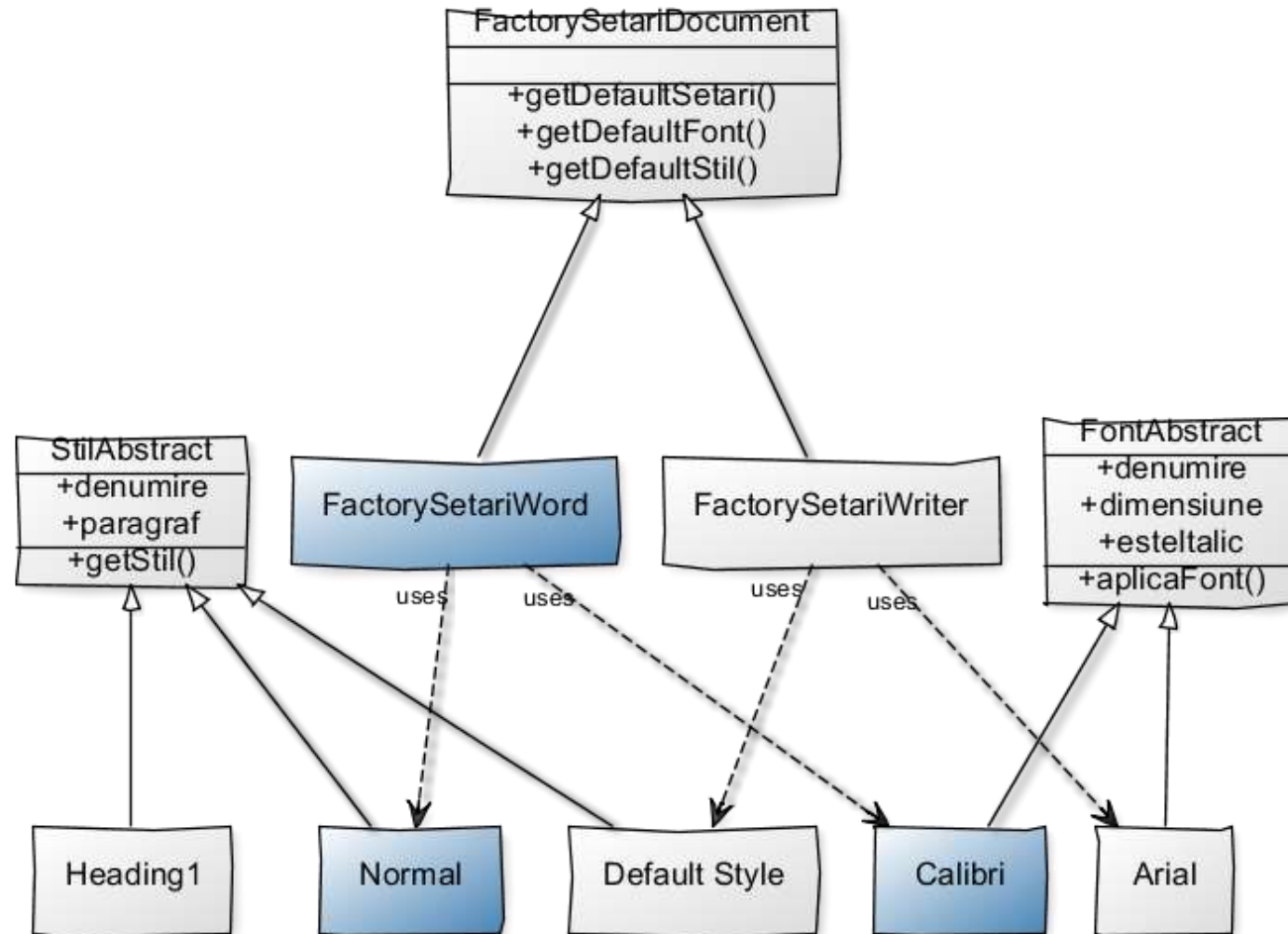
ABSTRACT FACTORY - Example



ABSTRACT FACTORY - Example



ABSTRACT FACTORY - Example



ABSTRACT FACTORY– Advantages and Disadvantages

Advantages:

- disconnects the instance generator from the client that uses them; strict control of instantiation - objects are not created directly by builders but by *factory* methods; the different types of objects are managed unitarily through the common interface - new types can be added without modifications

Disadvantages:

- large number of classes involved
- high level of complexity

ABSTRACT FACTORY – Scenarios

- Virtual store catalog product creation management
- Management creating different types of customer
- Management creating different types of reports
- Management of creating types of telephone numbers
- Management of creating types of bank accounts
- Management of creating types of pizza
- Management of creating menus in a restaurant

FACTORY

- All Factory patterns promote ***loose coupling*** and are based on ***Dependency Inversion Principle***
- Factory Method
 - *based on inheritance* - the creation of objects is done by subclasses that implement the factory method
 - It aims to delegate the creation of objects to subclasses
- Abstract Factory
 - *based on composition* - the creation of objects is done by published methods of the interface
 - It aims to create families of objects without depending on their concrete implementation

BUILDER Pattern (Adaptive Builder)

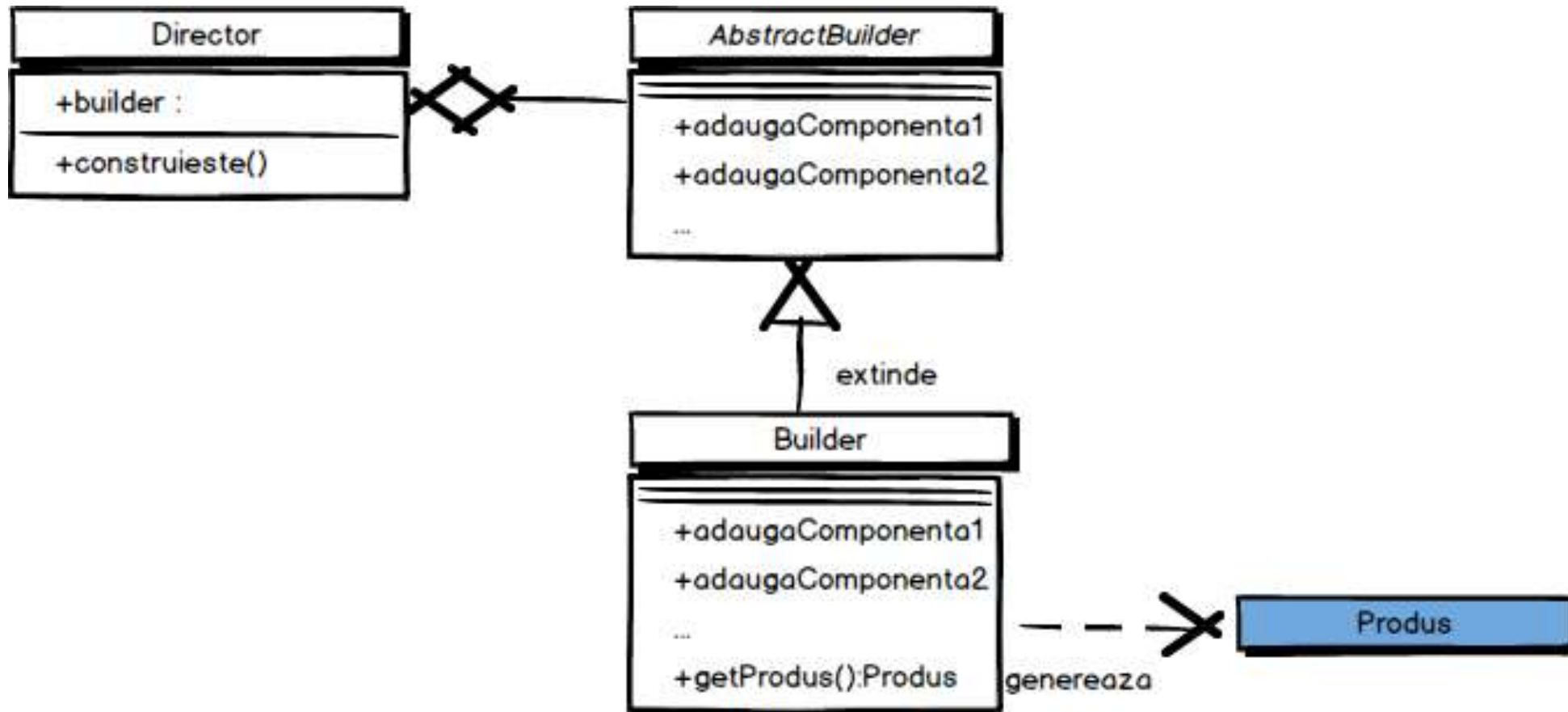
Creational Design-Patterns

BUILDER - Problem

- The solution must build complex objects through a mechanism that is independent of the process of making the objects
- The client constructs complex objects specifying only its type and value, without knowing the internal details of the object (how it stores and represents the values)
- The process of constructing objects must be able to be used to define different objects in the same family
- Objects are managed through the common interface
- The Builder instance constructs the object but its type is defined by subclasses



BUILDER- Diagram



http://en.wikipedia.org/wiki/Builder_pattern

BUILDER - Components

- **AbstractBuilder**
 - the abstract interface that defines the methods by which parts of the complex object are constructed
- **Builder**
 - the concrete class that builds the parts and based on them the final object
- **Produs**
 - the abstract class that defines the complex object that is constructed
- **Director**
 - the concrete class that builds the complex object using the *Builder* interface

BUILDER - Advantages and Disadvantages

Advantages:

- Complex objects can be created independently of the parts that make it up (an object can contain all or only one part)
- The system allows different representation of objects created through a common interface
- The object creation algorithm is flexible because the client chooses which parts to create

Disadvantages:

- **Be careful when creating objects - attributes can be omitted**

PROTOTYPE Pattern

Creational Design-Patterns

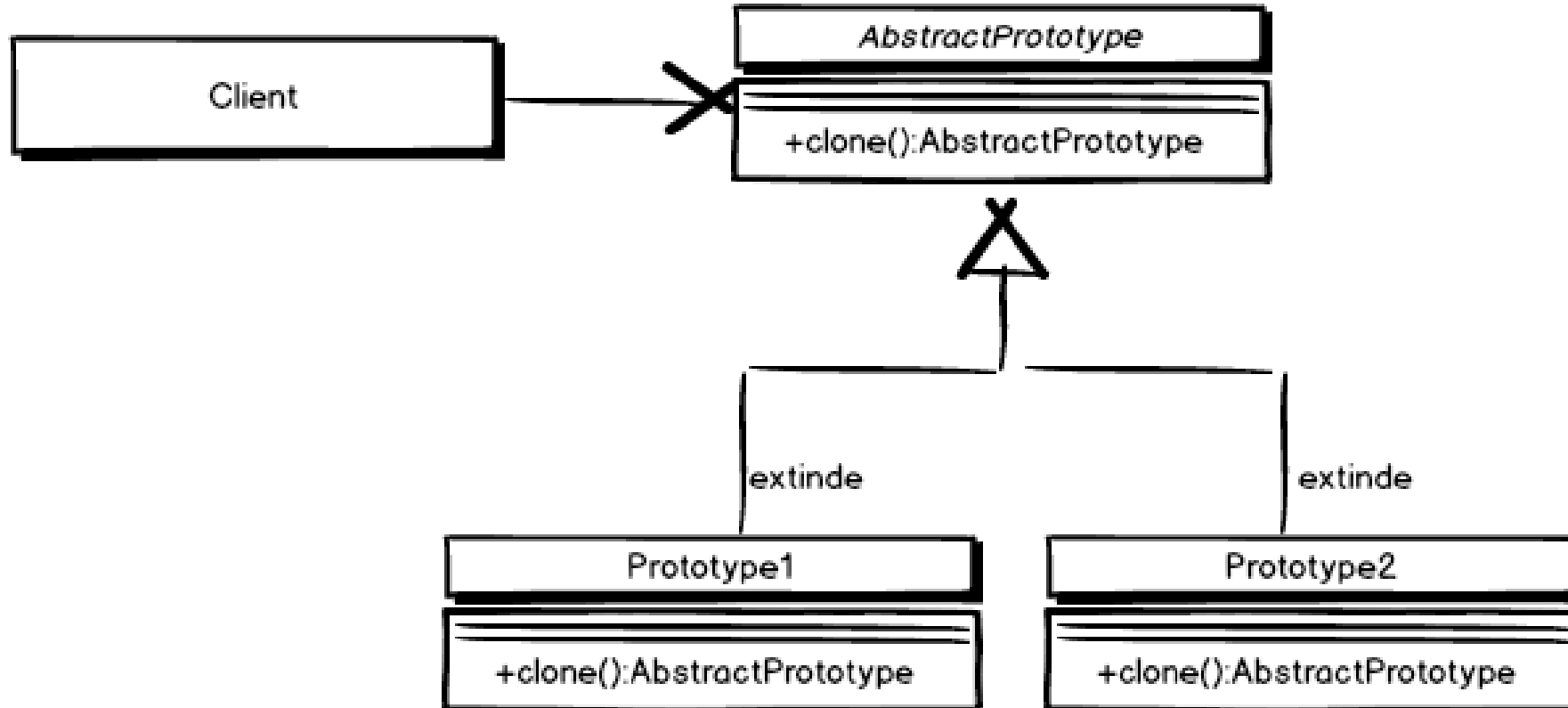
PROTOTYPE – Scenario

ACME Inc. wants to develop a 3D game for Android devices using its own engine. The two 3D models for characters are quite complex and their generation has an impact on the processing time and implicitly on the battery life. The same pattern is used several times to populate a scene with characters. An effective solution must be found to load the scenes quickly.

PROTOTYPE - Problem

- The solution generates expensive objects (creation time and busy memory) with long lifespan
- For efficiency, the solution reuses the object by cloning it (a new instant of the object is created)
- Implemented by a *clone* method ()

PROTOTYPE - Diagram



PROTOTYPE - Components

- *AbstractPrototype* – abstract class that defines the interface of a prototype; the class can implement a generic version of the clone () method that is inherited from concrete prototypes; in Java, this class extends the *Cloneable* interface to allow cloning of the object;
- *Prototype1*, *Prototype2* – concrete classes that represent the implementation of the *Prototype* pattern giving the possibility to clone existing objects;
- *Client* – sequence of code or API that uses classes that extend *AbstractPrototype*

PROTOTYPE - Implementation



SHALLOW COPY VS. DEEP COPY

In Java what is the default implementation for clones ()?

PROTOTYPE - Advantages and Disadvantages

Advantages:

- Quickly create identical objects (values) by cloning
- Avoid the explicit call of the constructor
- A collection of prototypes can be built to be used to generate new objects

Disadvantages:

- **Be careful when creating objects that share the same resources - shallow copy**

Structural Design-Patterns

Adapter, Facade, Decorator, Composite, Flyweight, Proxy

ADAPTER Pattern (Wrapper)

Structural Design-Patterns

ADAPTER – Scenario

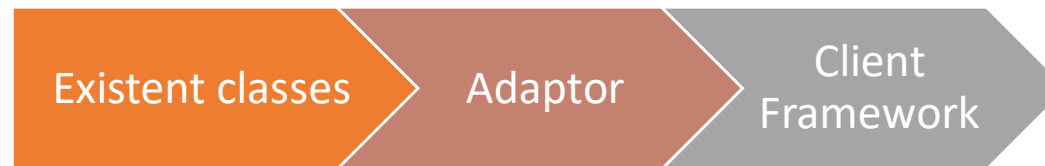
ACME Inc. wants to buy a new framework for back-end services. The interface for these services manages data through ACME objects, and the new framework processes data through MICRO objects. The company's developers need to find a solution to integrate the two frameworks without modifying them.

ADAPTER – Problem

- Using classes together that do not have a common interface
- The classes **do not change** but an interface is built that allows their use in another context
- The classes are adapted to a new context
- Calls to the class interface are masked by the adapter interface
- Transforming data from one format to another



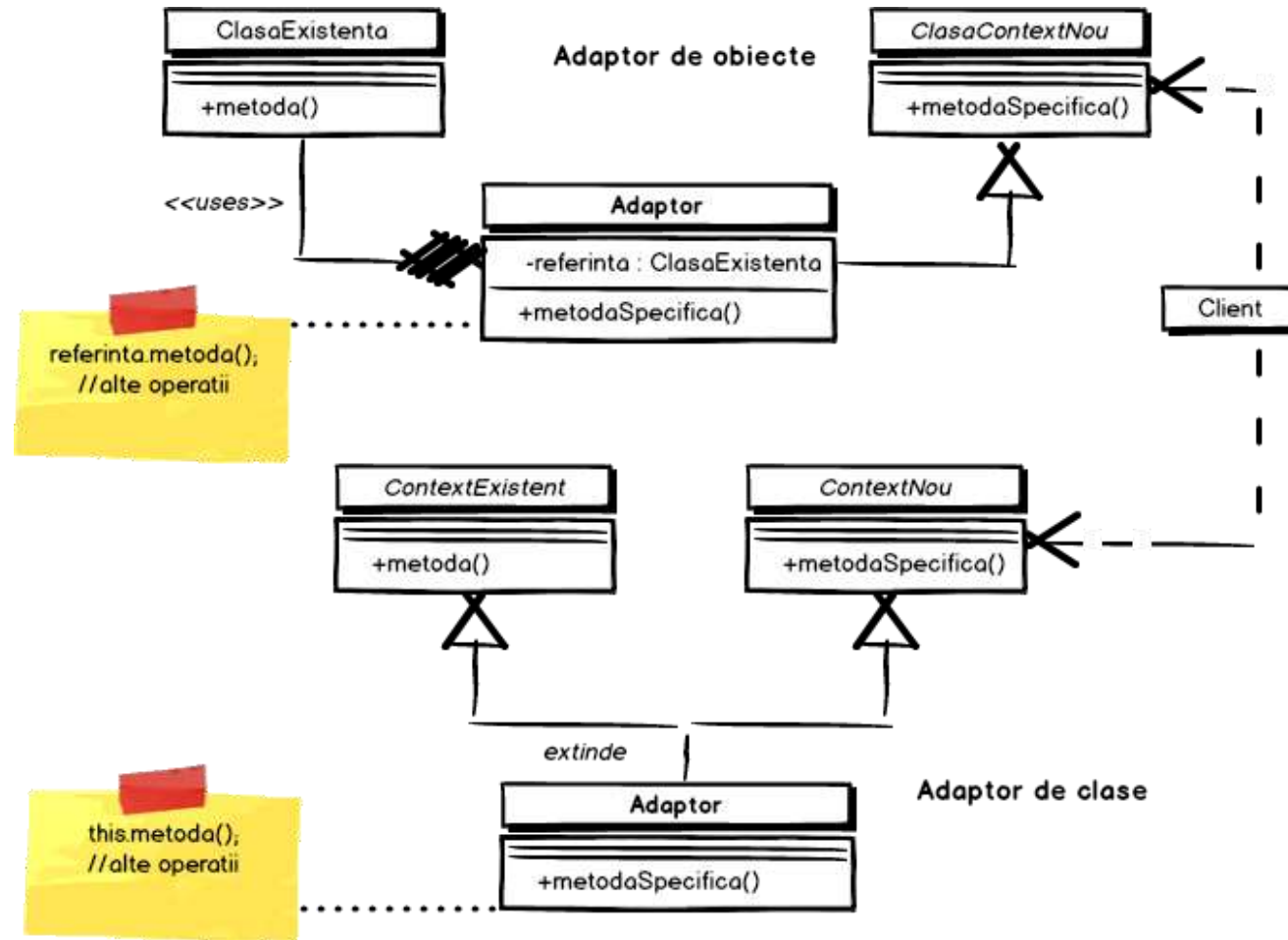
ADAPTER - Scenario



Codul nu se schimbă

Codul nu se schimbă

ADAPTER - Diagram



ADAPTER – Components

- **ClasaExistenta**
 - class to be adapted to a new interface (the source type);
- **ClasaContextNou**
 - defines the interface specific to the new domain (the destination type);
- **Adaptor**
 - adapts the interface of the existing class to that of the class in the new context;
 - contains (composition) a reference to the class / object to be adapted
- **Client**
 - represents the framework that calls the interface specific to the new domain

ADAPTER – Advantages and Disadvantages

Advantages:

- Existing classes (client and vendor) are not modified for use in another context
- Only an intermediate layer is added
- Adapters for any context can be easily defined

Disadvantages:

The class adapter is based on multiple derivation, which is not possible in Java. The alternative is through interfaces and composition

FAÇADE Pattern (Wrapper)

Structural Design-Patterns

FAÇADE - Scenario

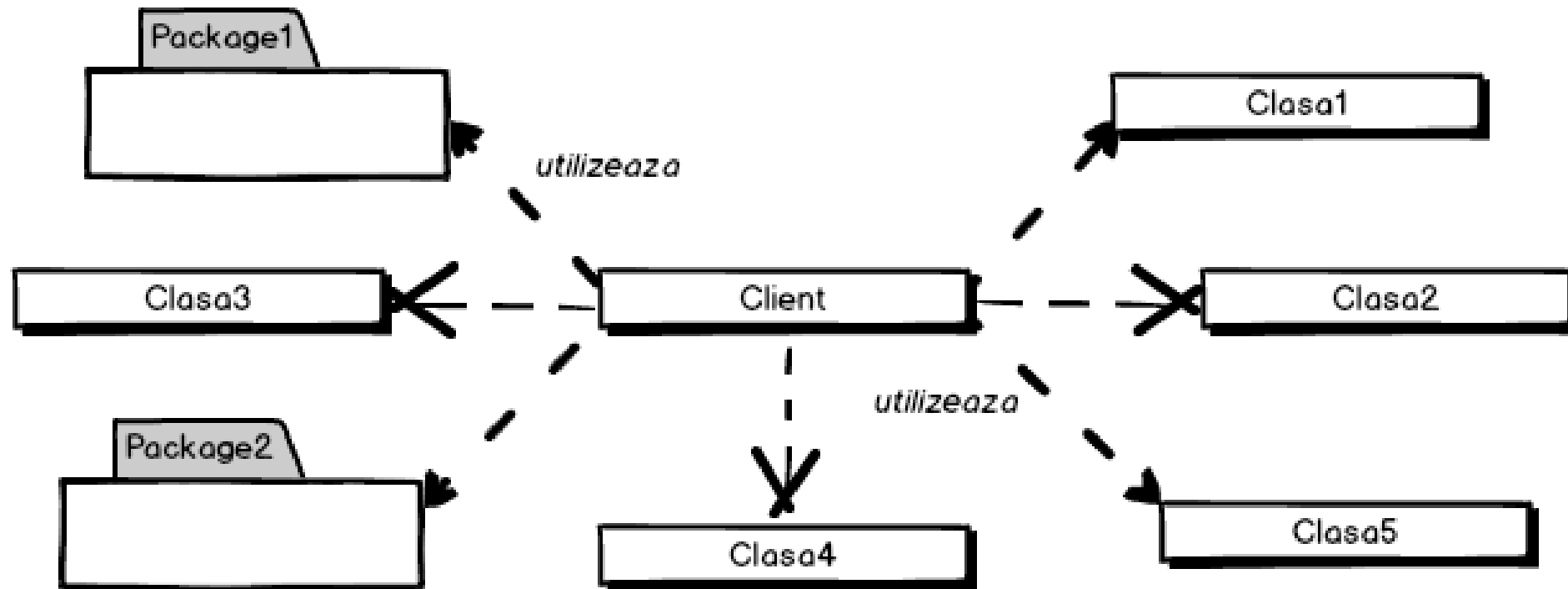
ACME Inc. develops a software solution for the management of a smart home. The inclusion in the framework of all the controllable components in such a house (windows, heating, alarm, etc.) generated a large number of classes. The department that develops the web interface of the solution offers a minimum set of functions that can be controlled remotely. Although the functionality is simple, the large number of classes that are instantiated and the methods used make it difficult to develop and test. In this sense, a simpler interface would help this department.

FAÇADE - Problem

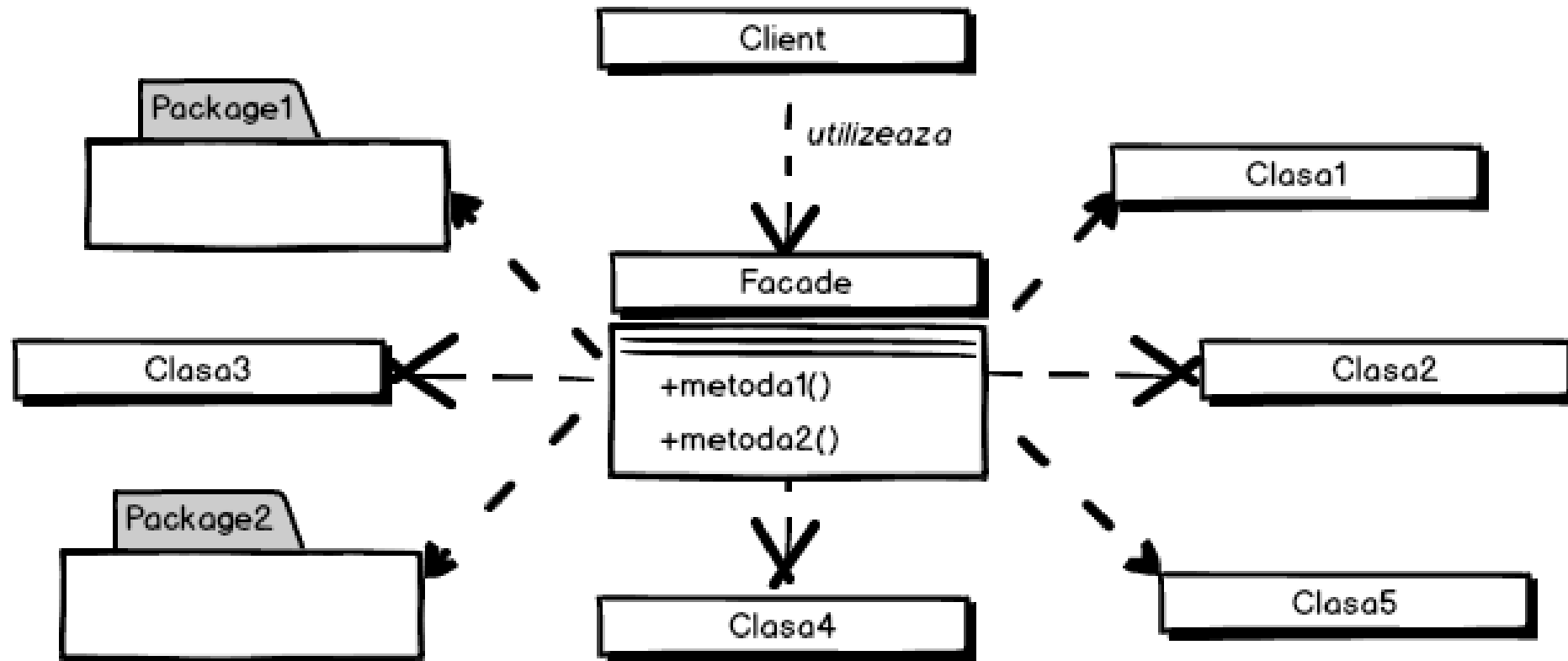
- The solution contains a lot of classes and the execution of a function involves multiple calls of methods in these classes.
- The classes do not change but an intermediate layer is built that allows easy calling / management of methods from several interfaces
- Useful in case the framework grows in complexity and it is not possible to rewrite it for simplification
- Calls to multiple interfaces are masked by this common interface



FAÇADE - Scenario



FAÇADE - Diagram



FAÇADE - Components

- **Clasa1, Clasa2, ..., Package1, Package2, ...**
 - existing classes that provide different interfaces;
- **Facade**
 - defines a simplified interface for the existing context;
- **Client**
 - Represents the framework that calls the interface specific to the new domain

FAÇADE - Advantages and Disadvantages

Advantages:

- The framework is not rewritten
- Only an intermediate layer is added that hides the complexity of the back framework
- Methods to simplify any situation can be easily defined
- Implement the ***Least Knowledge*** principle - reducing interactions between the object at the level of "close friends"

Disadvantages:

- Increases the number of wrapper classes
- Increases the complexity of the code by hiding some methods
- Negative impact on application performance

DECORATOR Pattern (Wrapper)

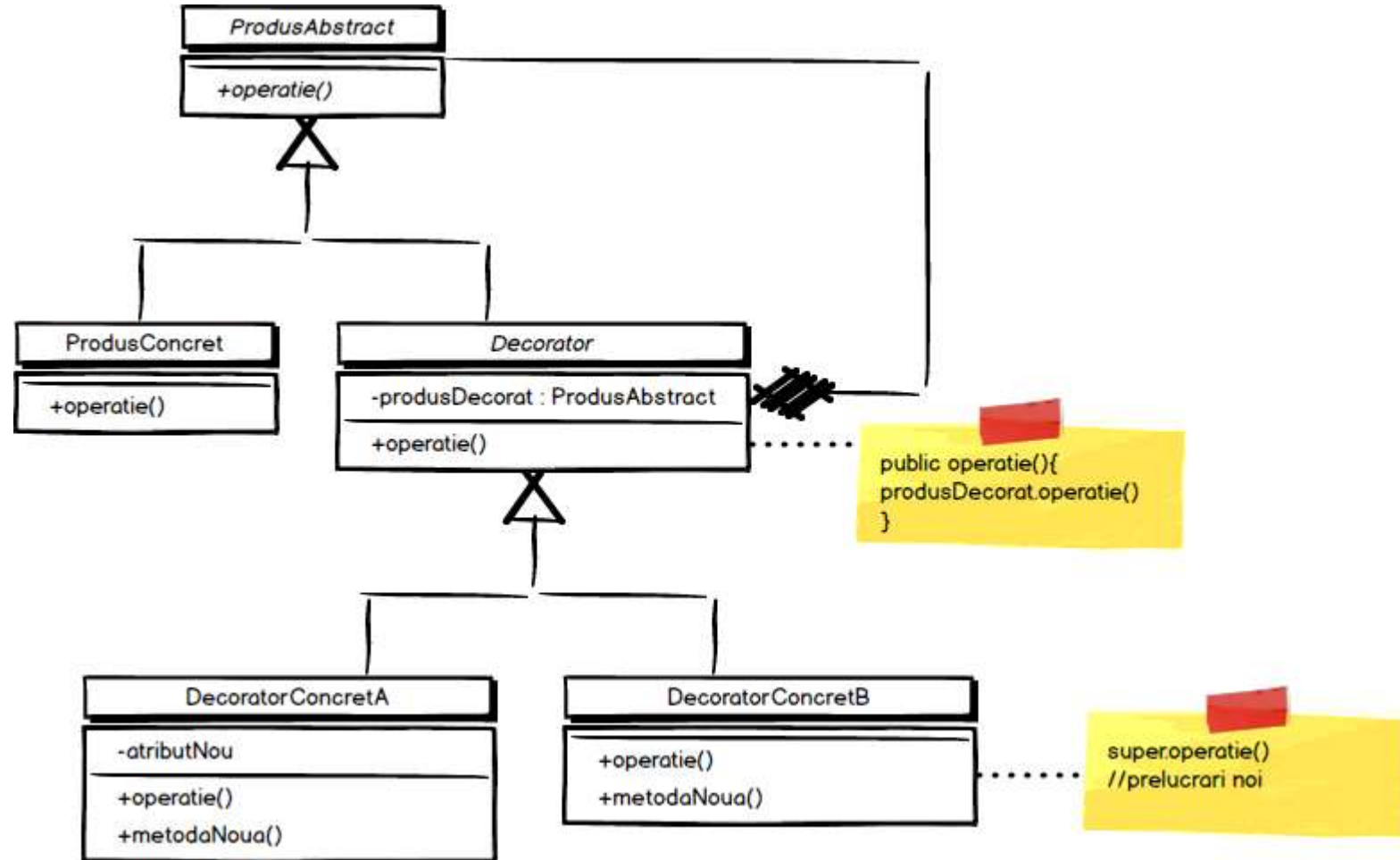
Structural Design-Patterns

DECORATOR - Problem

- Static or run-time extension (decoration) of the functionality or condition of some objects, independently of other instances of the same class
- The object can be extended by applying several decorators
- The existing class must not be modified
- Using a traditional approach, by deriving the class, leads to complex hierarchies that are difficult to manage. Derivation adds new behavior only to compilation



DECORATOR - Diagram



DECORATOR - Components

- **AbstractProduct**

- the abstract class that defines the interface of objects that can be decorated with new functions;

- **ConcreteProduct**

- defines objects that can be decorated;

- **Decorator**

- manages an *AbstractProduct* reference to the decorated object;
- the methods inherited from *AbstractProduct* call specific implementations from the class of the referred object;
- Can define a common interface to decorator classes;

- **ConcreteDecorator**

- Adds new functions to the referred object;

DECORATOR - Advantages

- Extending the functionality of a particular object is done dynamically, *at run-time*
- The decoration is transparent to the user because the class inherits the object-specific interface
- The decoration is made on several levels, but transparent to the user
- It does not impose limits on a maximum number of decorations

DECORATOR - Disadvantages

- A decorator is a wrapper for the original object. It is not identical to the encapsulated object
- Excessive use generates a lot of objects that look the same but behave differently -> difficult to understand and verify the code
- The situation must be analyzed carefully because in some situations the *Strategy* pattern is more appropriate

Adapter vs. Facade vs. Decorator

- **Adapter** – provides a different interface to the object
- **Facade** – provides a simplified interface to the object
- **Decorator** – provides an improved interface to the object

COMPOSITE Pattern

Structural Design-Patterns

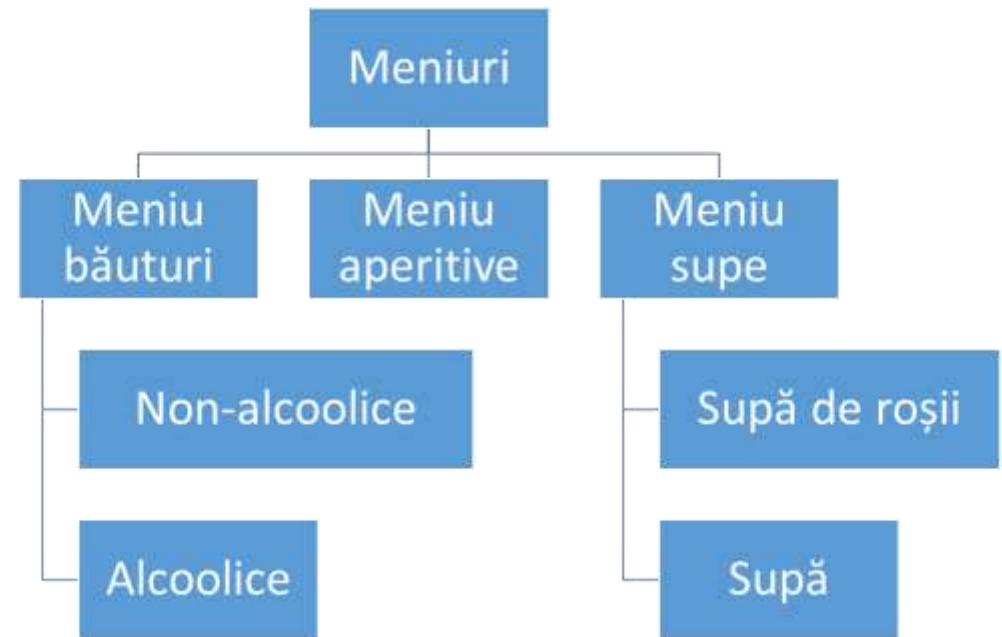
COMPOSITE - Scenario

ACME Inc. develops a software solution for human resources management in a company. The solution must provide a unitary mechanism that centralizes the company's employees and that takes into account:

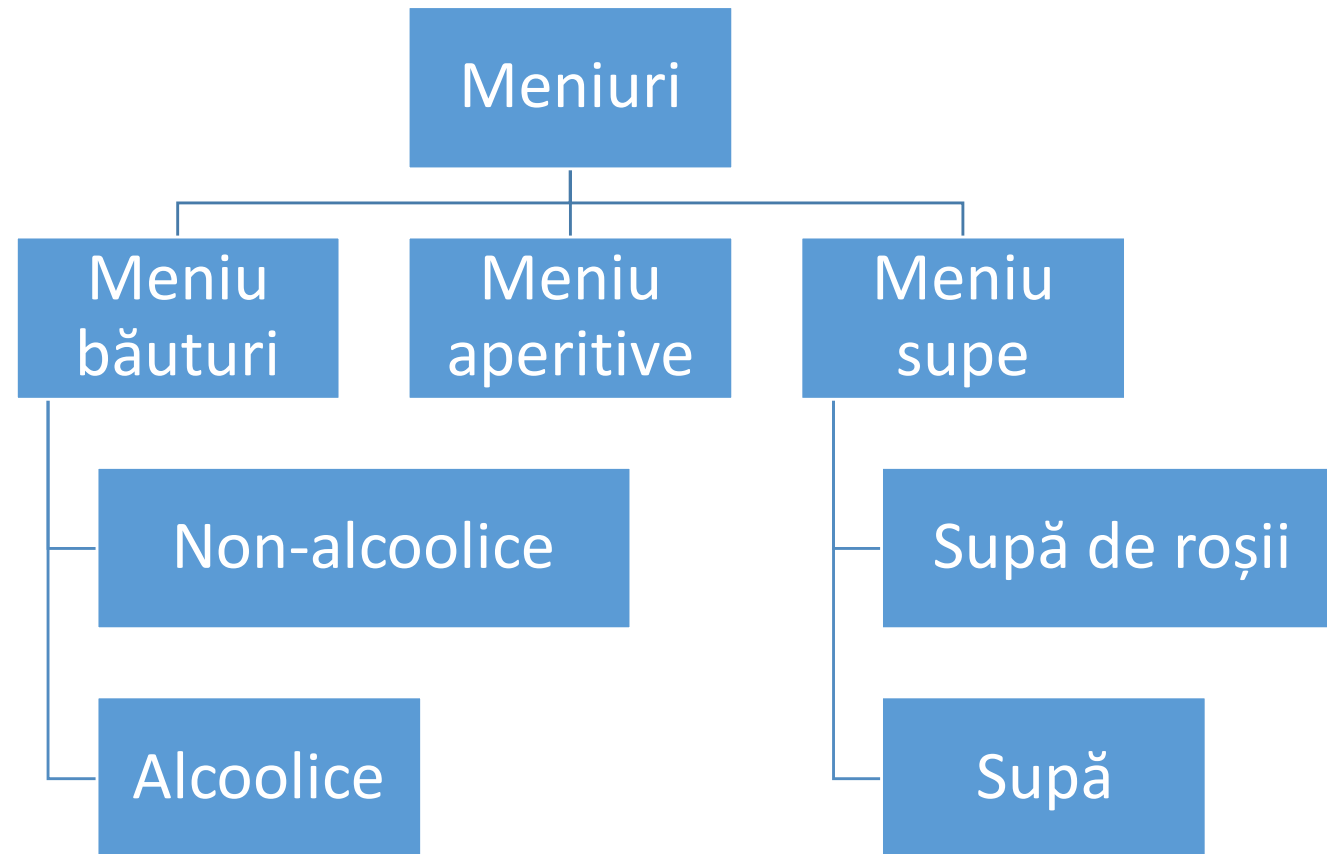
- hierarchical relationships
- employees belonging to a department
- different roles of employees
- the common set of functions that an employee can perform

COMPOSITE - Problem

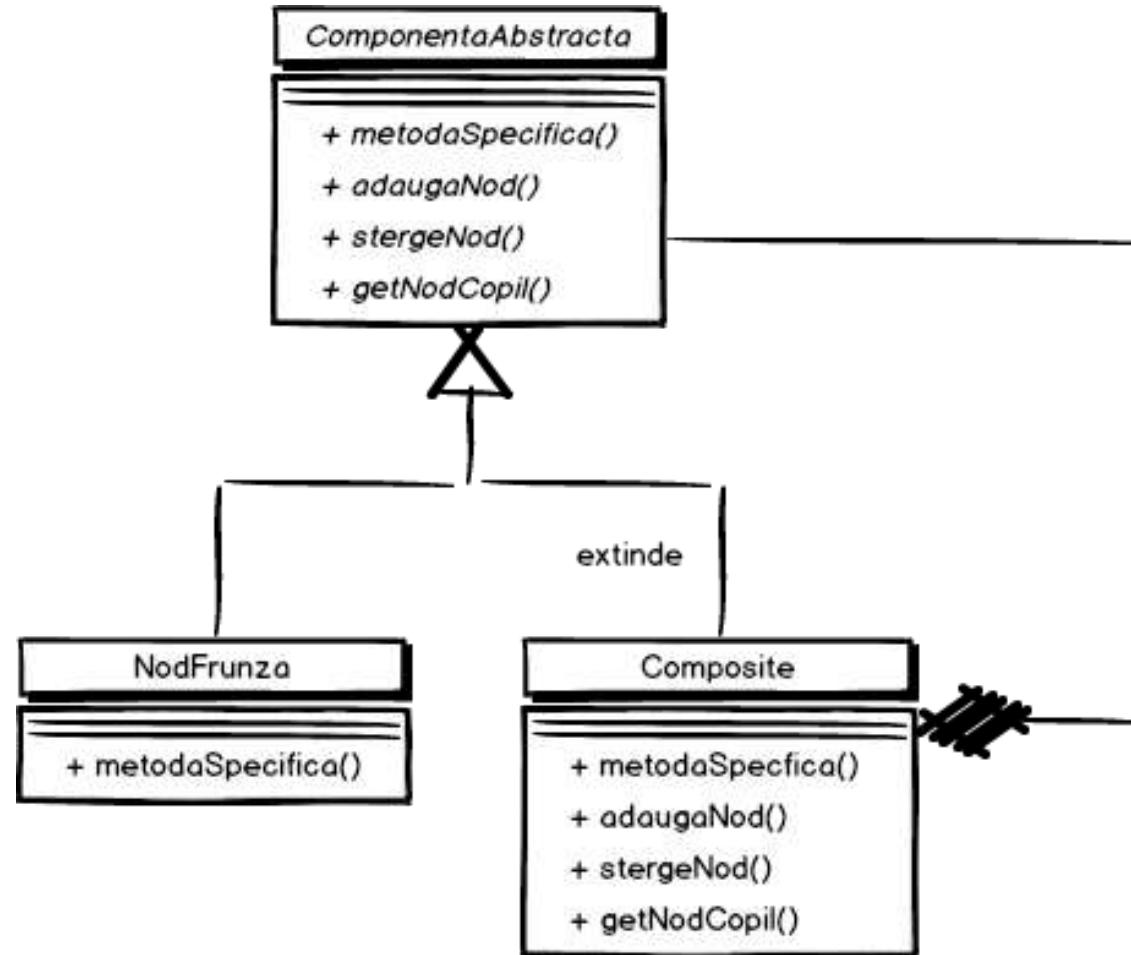
- The solution contains a lot of classes in a hierarchical relationship that must be treated as a unit
- Tree structures are built in which the intermediate nodes and the leaf nodes are treated as a unit



COMPOSITE - Scenario



COMPOSITE - Diagram



COMPOSITE - Components

- **Componenta**

- contains the abstract description of all components in the hierarchy
- Describes the interface of the objects in the composition

- **NodFrunza**

- Represents the leaf nodes in the composition
- Implement all methods

- **Composite**

- Represents a composite component - has son nodes
- Implements methods by which child nodes are managed

COMPOSITE – Advantages

- The framework is not rewritten
- Allows easy management of class hierarchies that contain both primitives and composite objects
- The code becomes simpler because the objects in the hierarchy are treated as a unit
- The addition of new components that respect the common interface does not raise additional issues

FLYWEIGHT Pattern

Structural Design-Patterns

FLYWEIGHT – Scenario

ACME Inc. develop a text editor as an alternative to known solutions. In the testing phase it was observed that as the text size increases, so does the memory occupied by this application. The growth rate is abnormal, quite fast, and finally generates delays between typing a character and the display. Tests on this area have shown that there is a link between the number of characters typed and the number of objects.

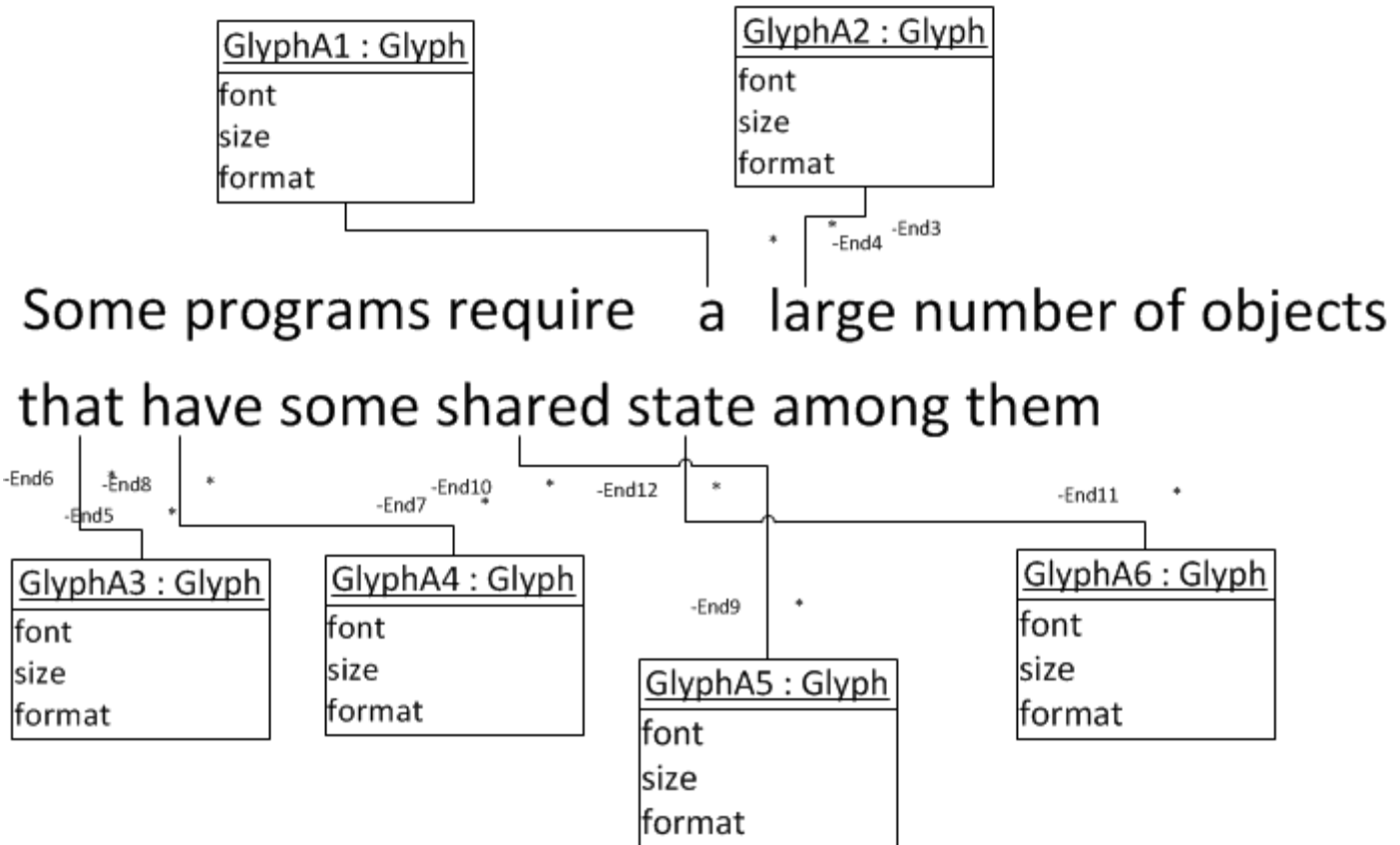
FLYWEIGHT - Problem

- The solution generates a lot of objects with a complex internal structure and occupying a large volume of memory
- Objects have common attributes but some of their state varies; the memory occupied by them can be minimized by sharing the fixed state between them
- The state of the objects can be managed through external structures and the number of objects actually created can be minimized
- Using an object means reloading its variable state to an existing object



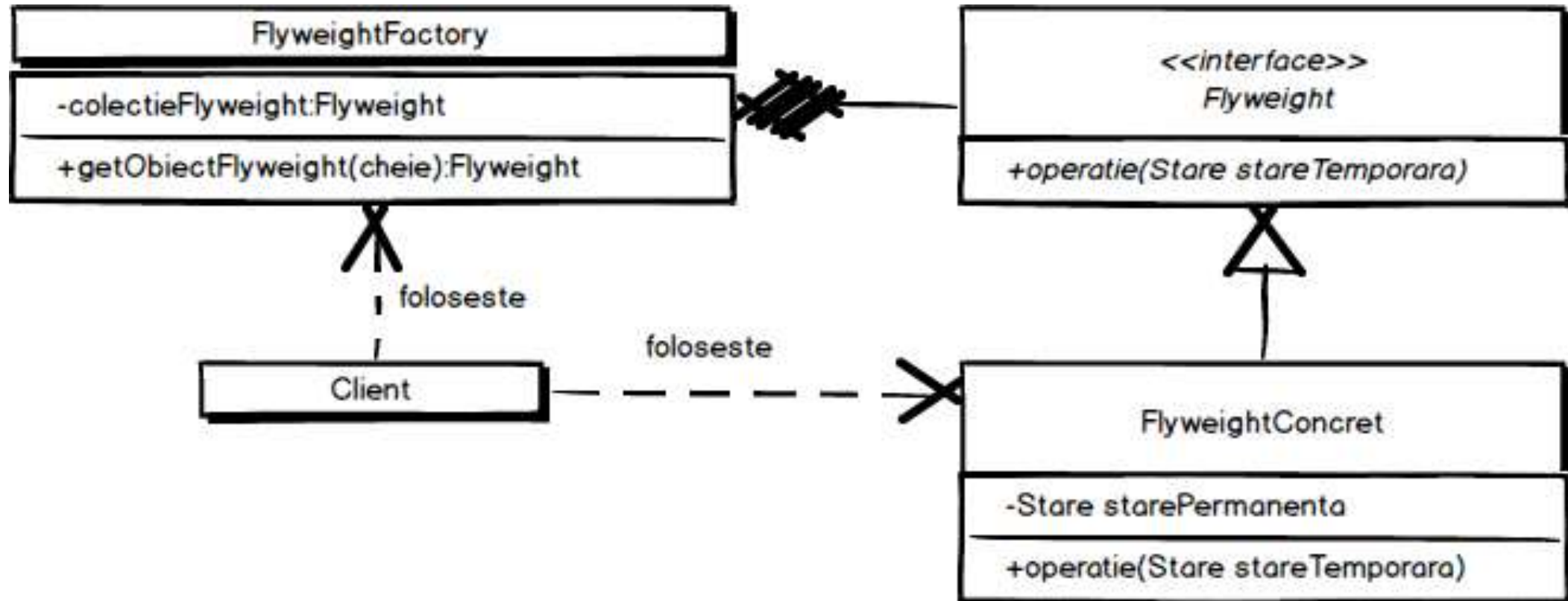
Attack of Clones

FLYWEIGHT - Scenario



Not known author source

FLYWEIGHT - Diagram



FLYWEIGHT - Components

- **Flyweight**

- Interface that allows objects to receive values that are part of their state and to make different processing based on it;

- **FlyweightFactory**

- A *factory* pattern that builds and manages flyweight objects;
- Maintain a collection of different objects so that they are created only once

FLYWEIGHT - Components

- **FlyweightConcret**

- The class implements the *Flyweight* interface and allows the storage of the ***permanent state*** (which cannot be shared) of objects
- Values that represent a temporary state shared between objects are received and processed through the methods in the interface

- **Client**

- Manage *flyweight* objects and their ***temporary state***

FLYWEIGHT – Advantages and Disadvantages

Advantages:

- It reduces the memory occupied by objects by sharing them between clients or their state between objects of the same type
- To properly manage the sharing of Flyweight objects between clients and threads, they must be *immutable*

Disadvantages:

- Classes and context must be analyzed to determine what is a variable state that can be internalized
- The effects are visible for solutions where the number of objects is large
- The low memory level depends on the number of Flyweight object categories

FLYWEIGHT vs DECORATOR vs PROTOTYPE

- **Prototype** creates identical objects by cloning an existing object. Created objects are identical (memory space, attributes)
- **Decorator** allows you to change the functionality of the object at runtime without changing its definition. Objects are created normally and then decorated on execution
- **Flyweight** allows you to store and create objects by sharing a partial object that contains only the common part (attributes and methods). Flyweight objects share the common part but manage their specific part

FLYWEIGHT vs DECORATOR vs PROTOTYPE

PROTOTYPE



Initial prototype

Decorated at run-time with a different color

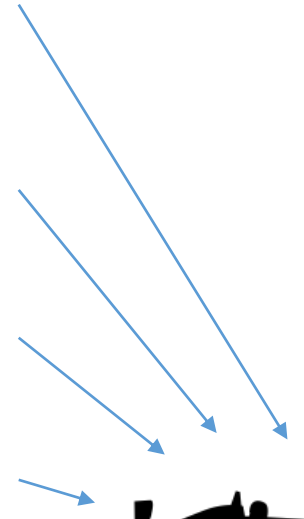
DECORATOR



Object default

Particular settings for instances

FLYWEIGHT



Flyweight

FLYWEIGHT – Similar templates

- ***Adapter*** – modify the interface of the object
- ***Decorator*** – maintains the initial interface but dynamically adds new functions to object behavior or changes existent ones
- ***Façade*** – provide a simplified interface

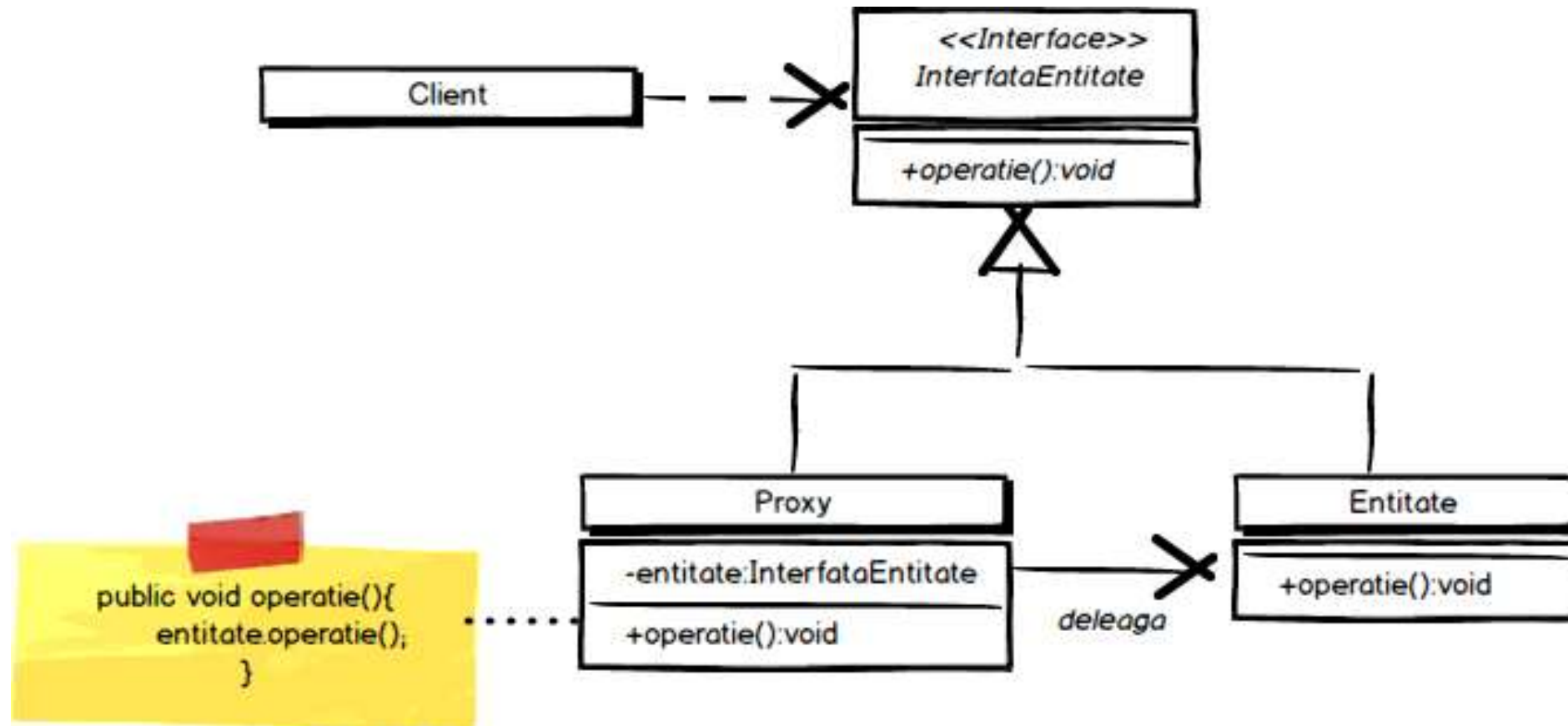
PROXY Pattern

Structural Design-Patterns

PROXY – Problem

- Interconnecting different APIs on the same machine or network
- Defining an interface between different frameworks
- The accessed object is external to the system
- The accessed objects are created on request
- Object access validation
- Add functionality when accessing the object

PROXY - Diagram



PROXY – Components

- **InterfataEntitate**

- Defines the interface of the real object to which the connection is made
- The interface is also implemented by the proxy so that it can connect to objects

- **Proxy**

- Manages the reference to the real object
- Implements the interface of the real object
- Controls access to the real object

- **Entitate**

- The actual object to which the proxy is linked

PROXY – Proxy types

- **Virtual Proxies:** manages the creation and initialization of expensive objects; they can only be created when a single instance is needed or shared between several clients;
- **Remote Proxies:** provide a local virtual instance for a remote object – Java RMI.
- **Protection Proxies:** controls access to an object's methods or to certain objects.
- **Smart References:** manages references to an object

Similar templates

- ***Adapter*** – modify the interface of the object
- ***Decorator*** – maintains the initial interface but dynamically adds new functions to object behavior or changes existent ones
- ***Strategy (Behavioral)*** – changes the behavior of the object by removing the dependency on the inner function (decouples the function implementation)
- ***Façade*** – provide a simplified interface
- ***Composite*** – aggregates several similar objects for a centralized hierarchical management

Behavioral Design-Patterns

Strategy, Observer, Chain of Responsibility, Template, State, Command,
Iterator, Memento

STRATEGY Pattern

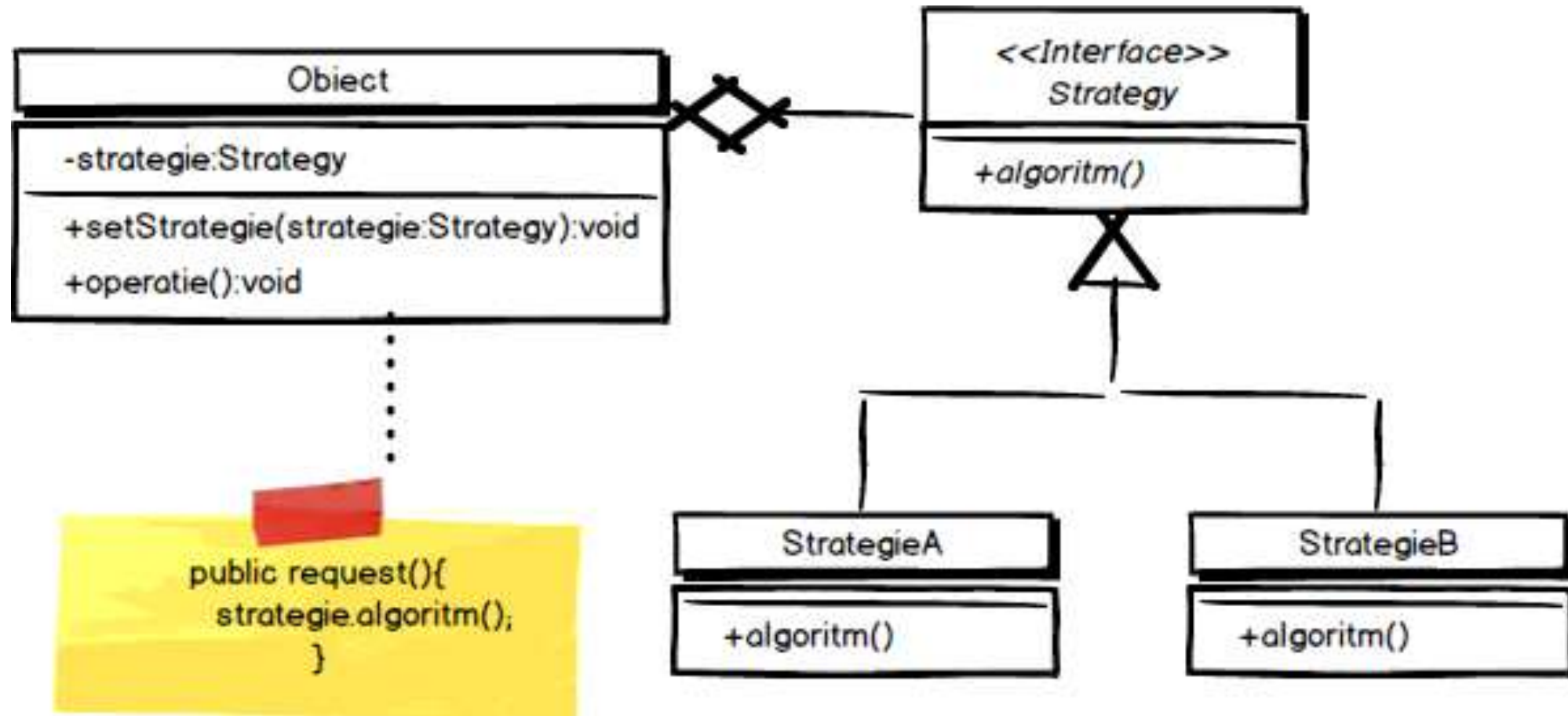
Behavioral Design-Patterns

STRATEGY - Problem

- *Run-time* choice of the algorithm / function to be used for data processing;
- The algorithm can be chosen based on conditions described at execution depending on the context of the input data
- The existing class must not be modified
- Using a traditional approach, by including in the classroom all possible methods, leads to complex hierarchies that are difficult to manage. Derivation adds new behavior only to compilation



STRATEGY – Diagram



STRATEGY - Components

- **Strategy**
 - the abstract class that defines the interface of objects that can provide new functions / processing algorithms;
- **StrategieA, StrategieB**
 - defines objects that provide solutions for data processing;
- **Object**
 - manages an *IStrategy* reference to the object that will provide the function / algorithm;
 - manages data / context that requires processing;

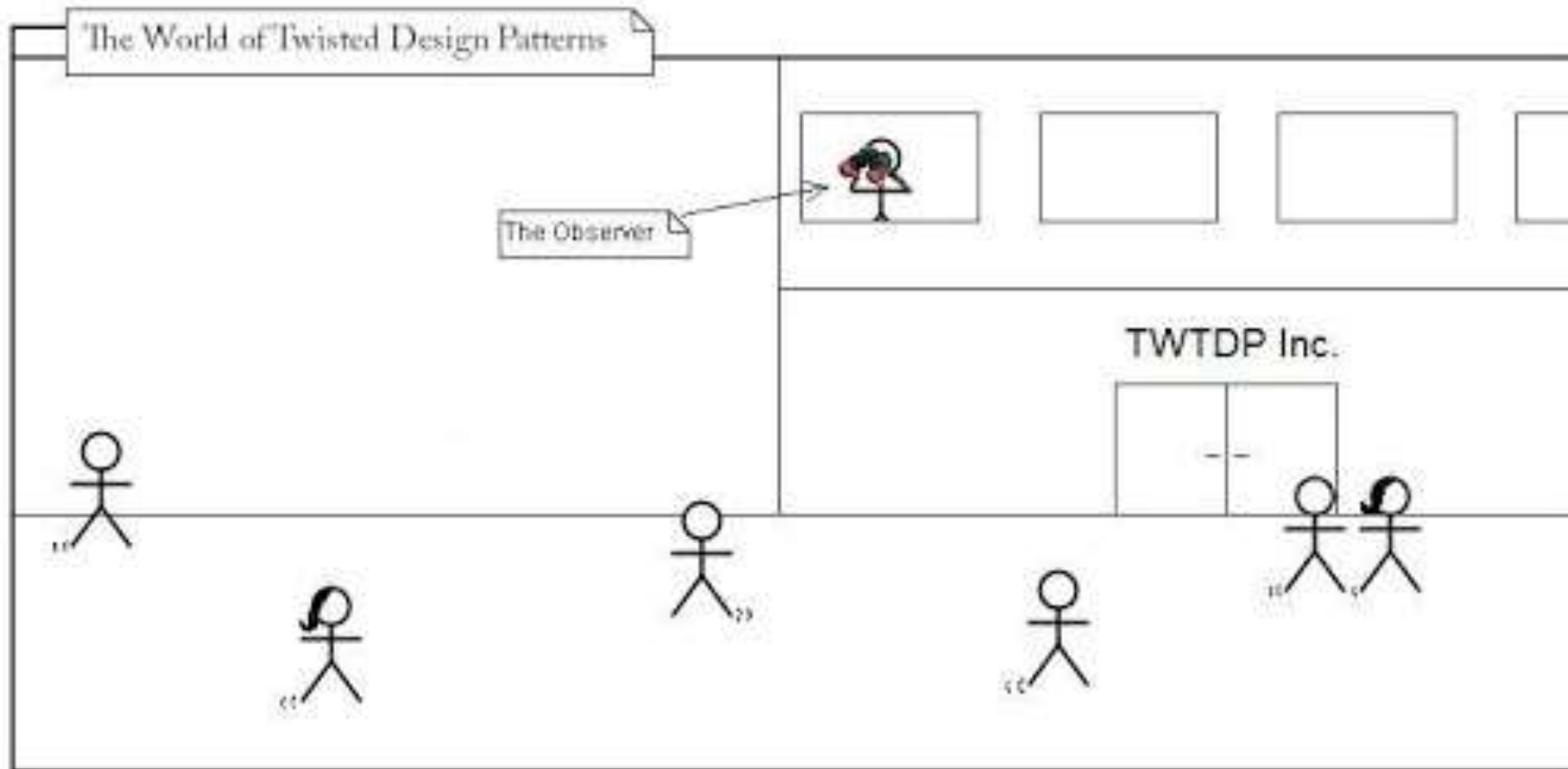
STRATEGY – Advantages

- The choice of the data processing method is made dynamically, at run-time
- It is allowed to define new algorithms independent of the modification of the class that manages the data
- It does not impose limits on a maximum number of functions / algorithms that can be used

OBSERVER Pattern

Behavioral Design-Patterns

OBSERVER



<http://umlcomics.blogspot.ro/2010/03/world-of-twisted-design-patterns.html>

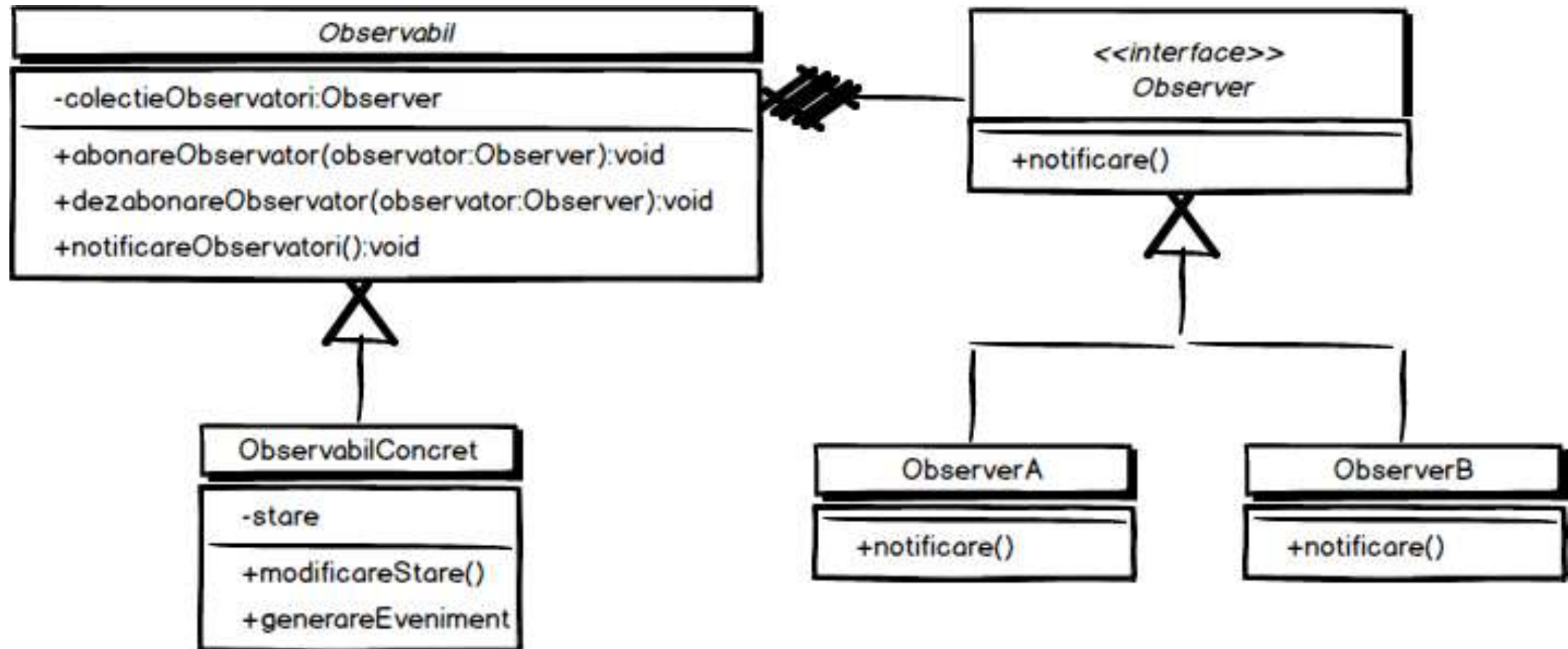
OBSERVER – Problem

- There are components that must be notified when an event occurs
- Interface level event management
- The components subscribe / register to that event - change of state / action
- Several components can be notified when an event occurs

OBSERVER – Advantages

- Outsourcing / delegation of functions to “observer” components that provide solutions to certain events independently of the event owner
- Concept integrated into the Model View Controller (MVC) architectural pattern
- Implements the concept of loose coupling OOP - objects are interconnected through notifications and not through class instantiations and method calls

OBSERVER – Diagram



OBSERVER - Componente

- **Observabil**

- clasa abstracta ce definește interfața obiectelor gestionează evenimente si care sunt observabile;

- **ObservabilConcret**

- defines objects that allow subscribers to subscribe;

- **Observator**

- Interface that defines the way in which observers are notified;
- Allows the management of several observers

- **ConcreteDecorator**

- Implements concrete functions that are executed after notification;

OBSERVER – Communication methods

2 models for notifying the observer of the status change:

- **Push** - the object sends all the details to the observer
- **Pull** - the object only notifies the observer and he requests the data when he needs them

CHAIN OF RESPONSIBILITY Pattern

Behavioral Design-Patterns

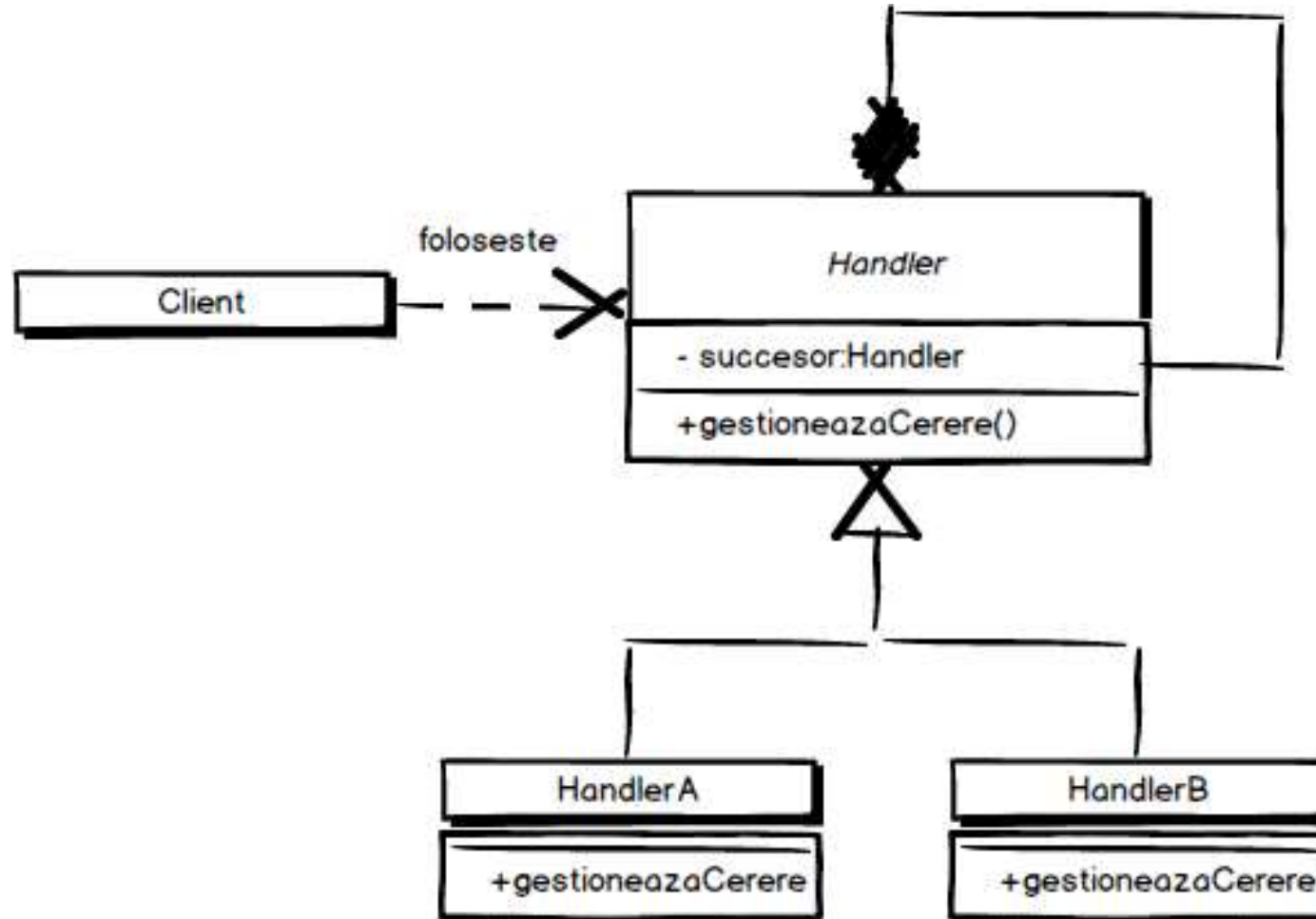
CHAIN OF RESPONSIBILITY – Problem

- The treatment of an event or an object is done differently depending on its condition
- The management of all cases would involve a complex structure that verifies all the particular cases
- There are links of dependence between use cases: the execution of one case may involve ignoring the others or treating the next case



<http://www.leahy.com.au/>

CHAIN OF RESPONSIBILITY - Diagram



CHAIN OF RESPONSIBILITY - Components

- **Handler**
 - the abstract class that defines the interface of the objects that manage the event processing request;
- **HandlerA**
 - defines concrete objects that form the notification processing sequence;
- **Client**
 - generates the event or notifies the first object in the object sequence;

STATE Pattern

Behavioral Design-Patterns

STATE - Problema

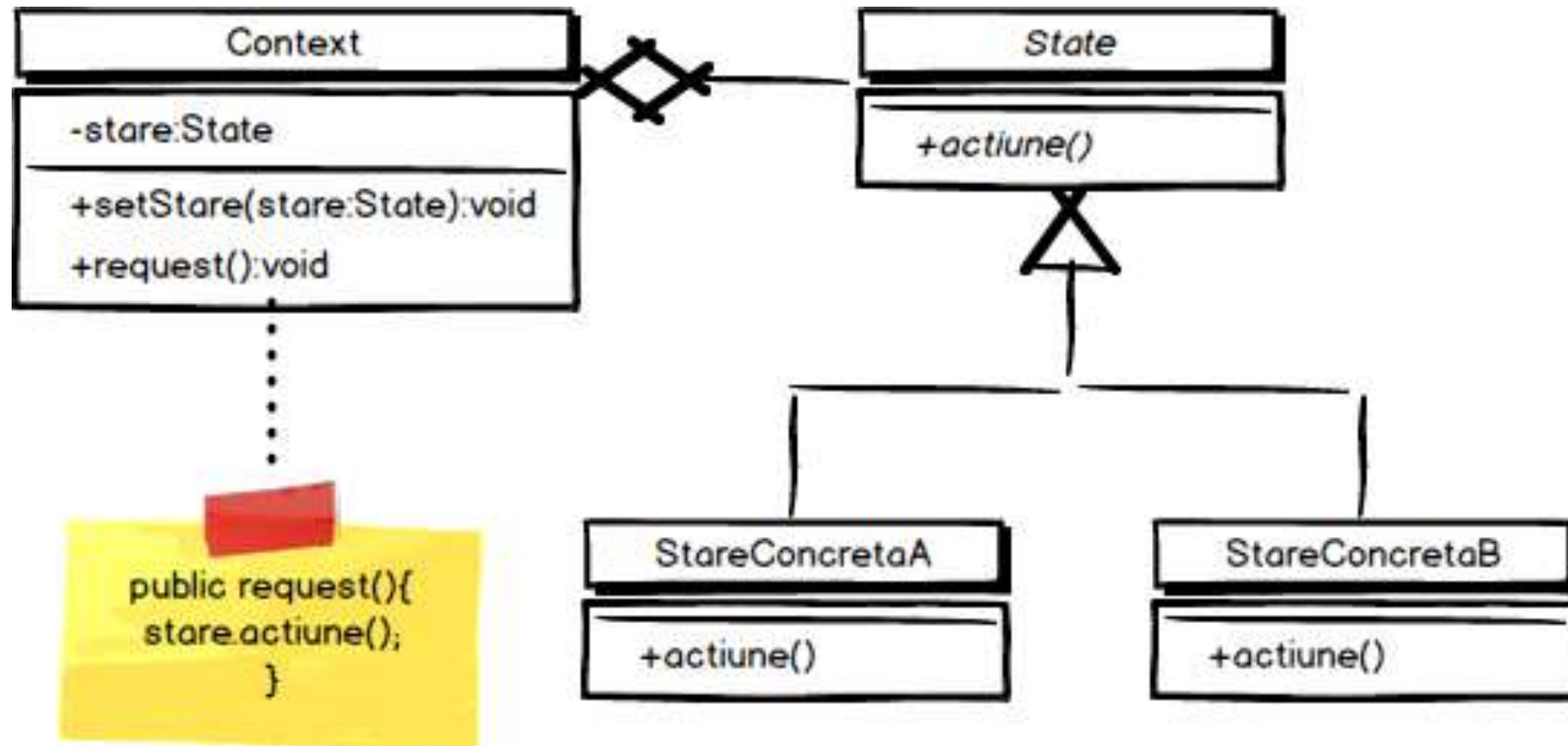
- Aplicația tratează un anumit eveniment diferit în funcție de starea unui obiect
- Numărul de stări posibile poate să crească și tratarea unitară a acestora poate să influențeze complexitatea soluției
- Modul de tratare a acțiunii este asociat unei anumite stări și este încapsulat într-un obiect de stare



R10-3e

Sign image from the Manual of Traffic Signs <<http://www.traffic-sign.us/>>
This sign image copyright Richard C. Moeur. All rights reserved.

STATE - Diagrama



STATE - Componente

- **State**

- clasa abstracta ce definește interfața obiectelor ce gestionează implementarea unor acțiuni în funcție de stare;

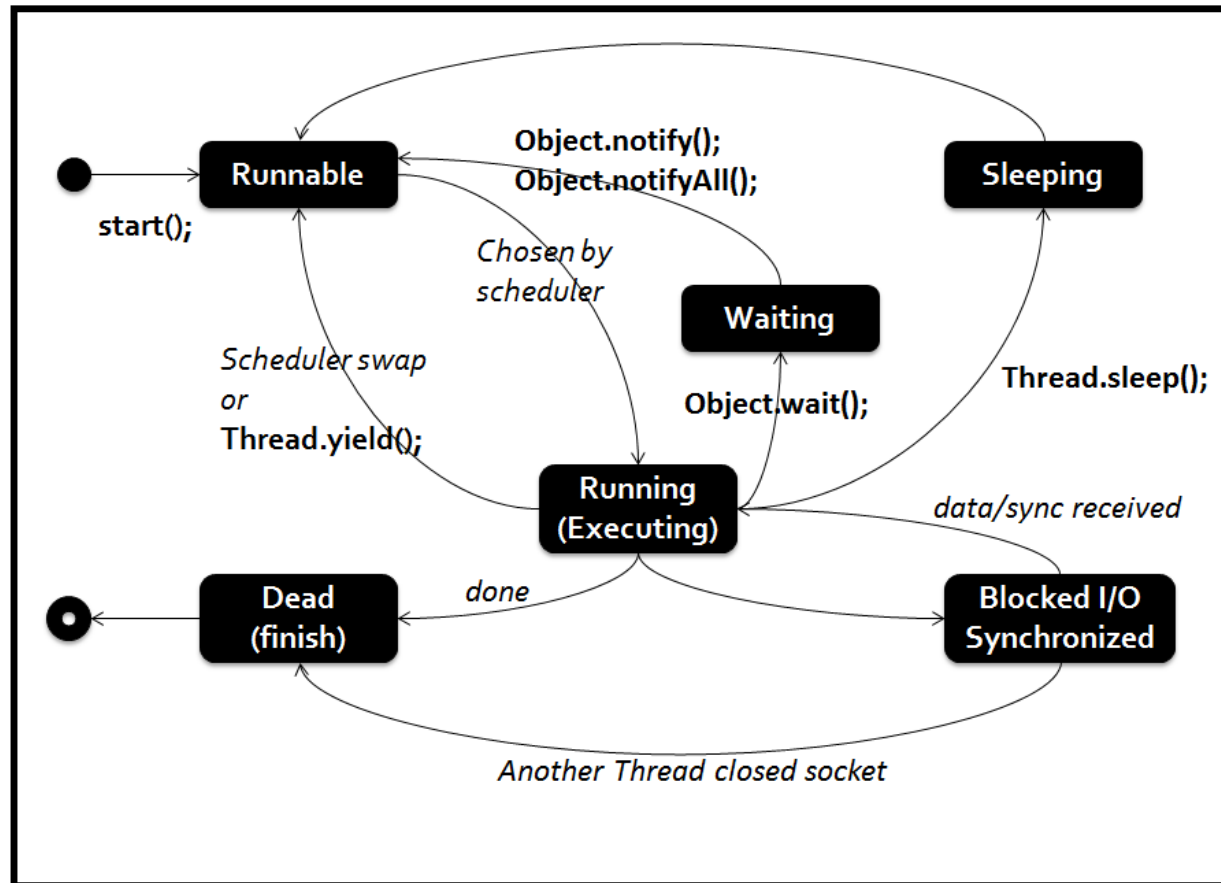
- **StareConcretaA, StareConcretaB**

- definește obiecte concrete ce implementează metodele de acțiuni aferente stării respective;

- **Context**

- gestionează referința de tip State și folosește metodele acesteia pentru a implementa diferite prelucrări în funcție de stare;

STATE Machine



Ciclu de viata Java Thread

<https://codexplo.wordpress.com/2012/10/20/state-diagram-of-java-thread/>

Modelul COMMAND

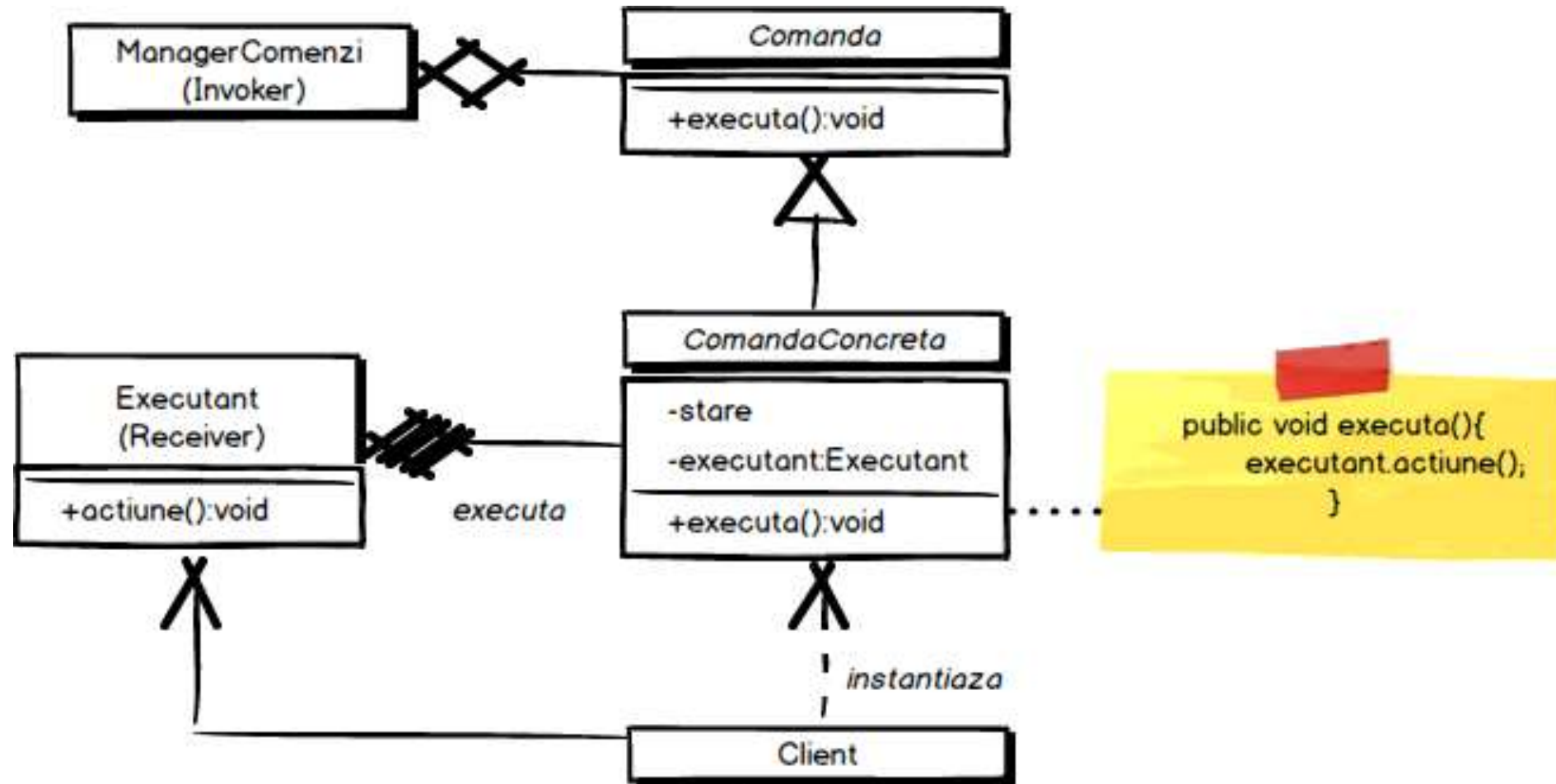
Modele comportamentale

COMMAND - Problema

- Aplicația definește acțiuni parametrizabile ce pot fi executate mai târziu fără a solicita clientului cunoașterea detaliilor interne necesare execuției.
- Pentru a nu bloca clientul, se dorește ca aceste acțiuni să fie definite și trimise spre execuție fără a mai fi gestionate de client
- Se decuplează execuția întârziată (ulterioară) a unei acțiuni de proprietar. Din punctul de vedere, acțiunea a fost deja trimisă spre execuție.
- Concept echivalent cu macro-urile. Obiectul de tip *command* încapsulează toate informațiile necesare execuției acțiunii mai târziu de către responsabil
- Clientul este decuplat de cel ce execută acțiunea



COMMAND - Diagrama



COMMAND - Componente

- **Comanda**
 - Definește interfața necesară execuției acțiunii + alte detalii;
- **ComandaConcreta**
 - Extinde interfața comenzii si implementează metoda prin care este controlat *Receiver-ul*
 - Reprezintă legătura dintre *Receiver* si acțiune
- **Client**
 - Creează un obiect *ComandaConcreta* si setează *Receiver-ul* acestuia;
- **Invoker (ManagerComenzi)**
 - Creează comanda și cere îndeplinirea acțiunii;
- **Receiver (Executant)**
 - Obiectul care este responsabil cu execuția acțiunii;

COMMAND - Scenariu

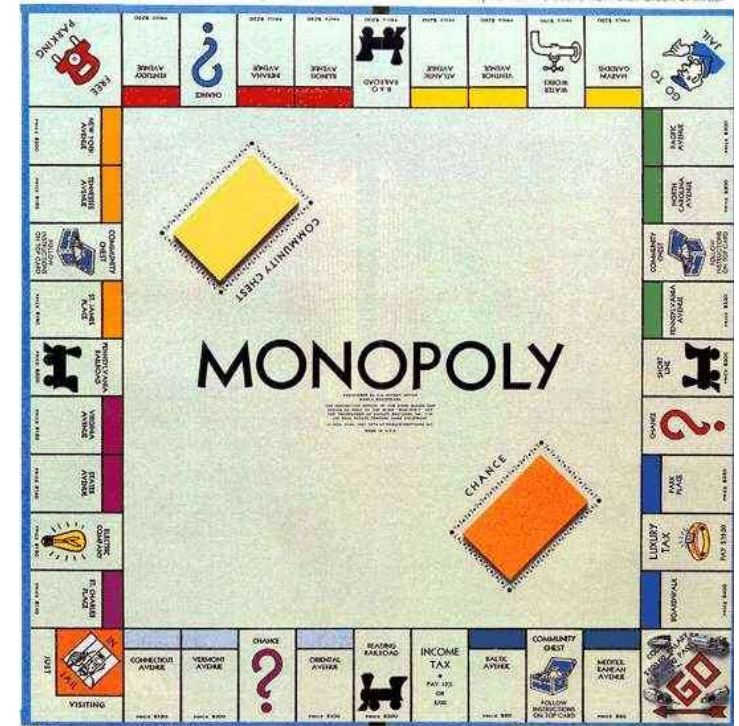
ACME Inc. dezvoltă o soluție software pentru un restaurant, astfel încât chelnerul să poată prelua comenzile direct pe telefonul mobil. Comenzile sunt preluate de la client și ele sunt create pe loc, fiind automat alocat bucătarul specializat pe acel fel de mâncare, ingredientele folosite și alte cerințe speciale ale clientului. Aceste detalii sunt puse de aplicație, fără a fi necesară intervenția chelnerului care doar selectează felul de mâncare solicitat. Comenzile sunt trimise bucătarilor la finalizarea comenzii pentru masa respectivă, urmând să fie executate în funcție de gradul de încărcare al fiecărui bucătar.

Modelul TEMPLATE METHOD

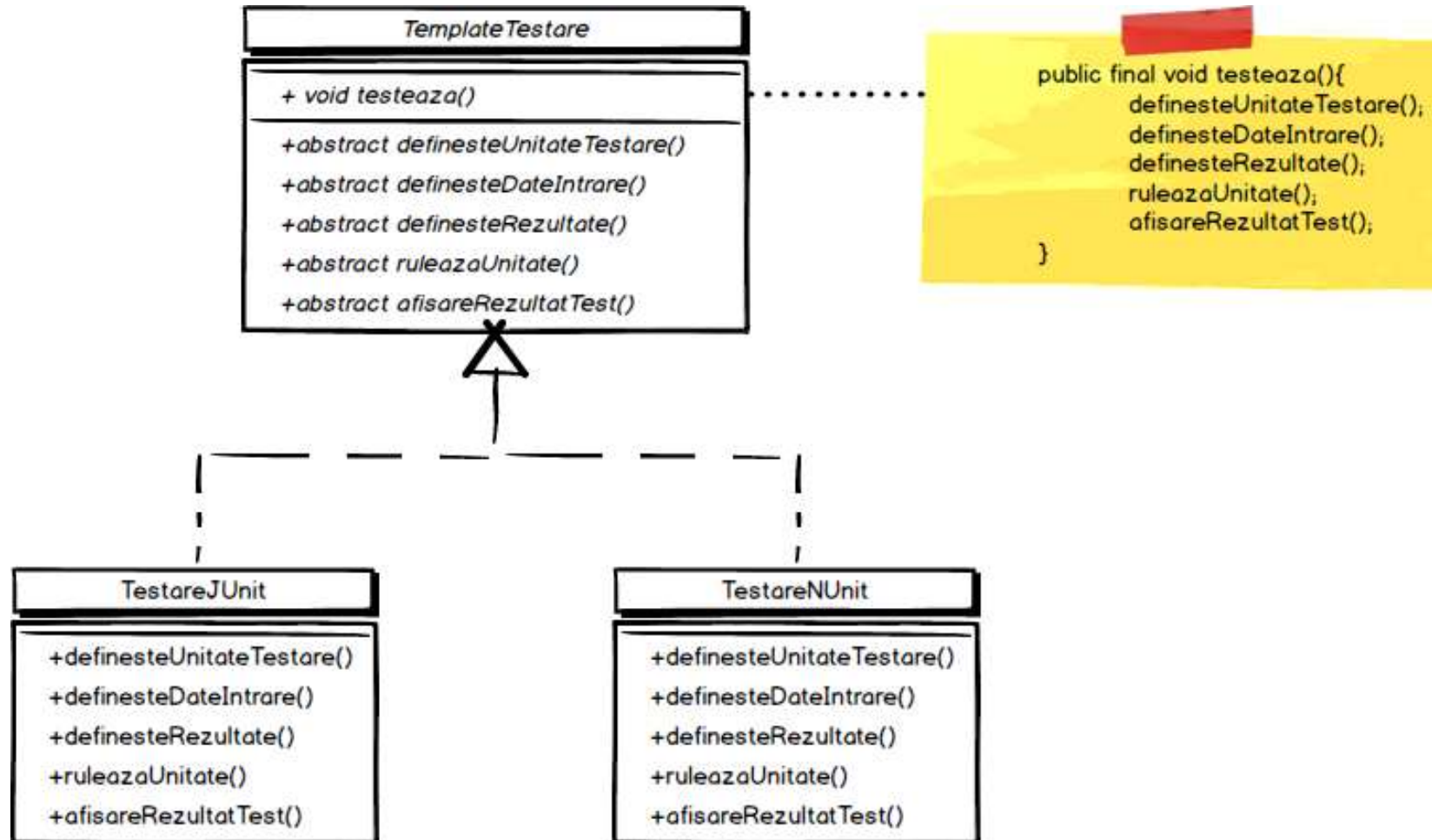
Modele comportamentale

TEMPLATE METHOD - Problema

- Implementarea unui algoritm presupune o secvență predefinită și fixă de pași
- Metoda ce definește schema algoritmului – metoda template
- Pot fi extinse/modificate metodele care implementează fiecare pas însă schema algoritmului nu este modificabilă
- Implementează principiul Hollywood: "*Don't call us, we'll call you.*"
- Metodele concrete de definesc pașii algoritmului sunt apelate de metoda template



TEMPLATE METHOD - Diagrama



TEMPLATE - Componente

- **TemplateTestare**

- Clasa abstracta ce definește interfața unui obiect de tip TemplateTestare si modalitatea în care sunt executate metodele
- Conține o metod template ce implementează șablonul de execuție a interfeței și care nu se supradefinește

- **TestareJUnit**

- Definește interfața obiectului într-o situație concretă

- **TestareJUnit**

- Definește interfața obiectului într-o situație concretă

Modelul MEMENTO

Modele comportamentale

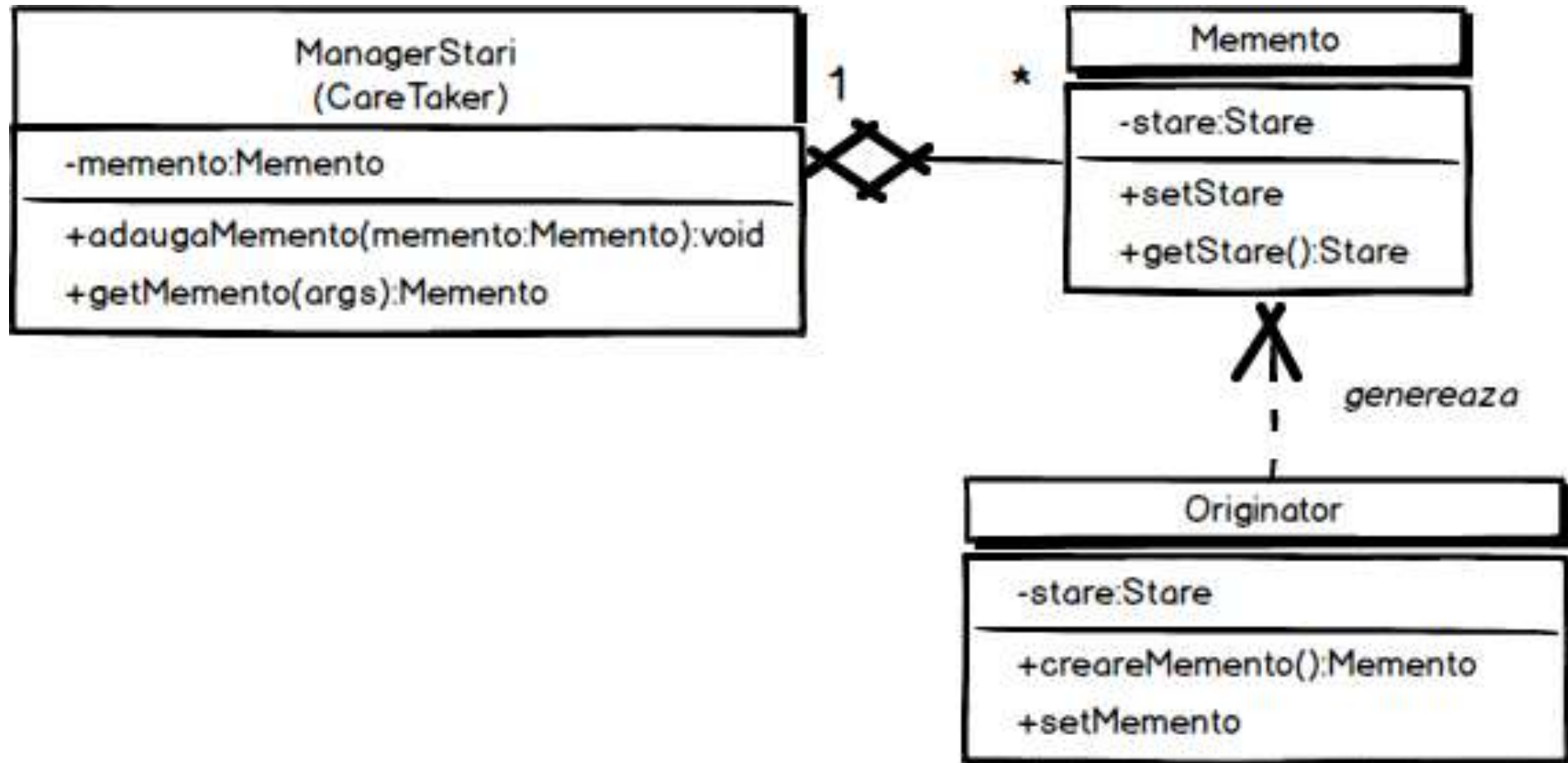
MEMENTO - Problema

- Aplicația trebuie să permită salvarea stării unui obiect
- Imaginile stării obiectului pentru diferite momente sunt gestionate separat
- Obiectul își poate restaura starea pe baza unei imagini anterioare

CTRL + Z

if only it worked for everything...

MEMENTO - Diagrama



MEMENTO - Componente

- **Memento**

- Gestionează starea internă a obiectului *Originator* pentru un anumit moment; Este creat de *Originator* și este gestionat de *Caretaker*

- **Originator**

- Obiectul a cărui stare este urmărită; Poate genera un Memento cu starea lui la momentul respectiv. Poate să își refacă starea pe baza unui Memento

- **ManagerStari (Caretaker)**

- Gestionează obiectele de tip Memento fără a avea acces pe conținutul acestora;

Recapitulare

- **Creăţionale:**

- *Factory* – creează obiecte dintr-o familie
- *Builder* – creează obiecte setând anumite atribute
- *Singleton* – creează o unică instanţă
- *Prototype* – creează clone ale unui obiect fără a se baza pe constructor

- **Structurale:**

- *Adapter* – adaptează un API (o interfaţă) la altul
- *Composite* – gestionează o ierarhie de obiecte
- *Decorator* – atribuie la run-time funcţionalitate nouă unui obiect existent
- *Façade* – simplifică execuţia (apelarea) unui scenariu complex
- *Flyweight* – gestionează eficient mai multe instanţe (clone) ale unui set redus de modele

Recapitulare

- **Comportamentale:**

- *Strategy* – schimbă la run-time funcția executată
- *Observer* – execută o acțiune când are loc un eveniment sau un observabil își schimbă starea
- *Chain of Responsibility* – gestionează o secvență de acțiuni ce pot procesa un eveniment sau un obiect
- *Command* – gestionează realizarea întârziată a unei acțiuni
- *Memento* – gestionează stările anterioare ale unui obiect
- *State* – stabilește tipul acțiunii în funcție de starea obiectului
- *Template* – gestionează un șablon fix de acțiuni