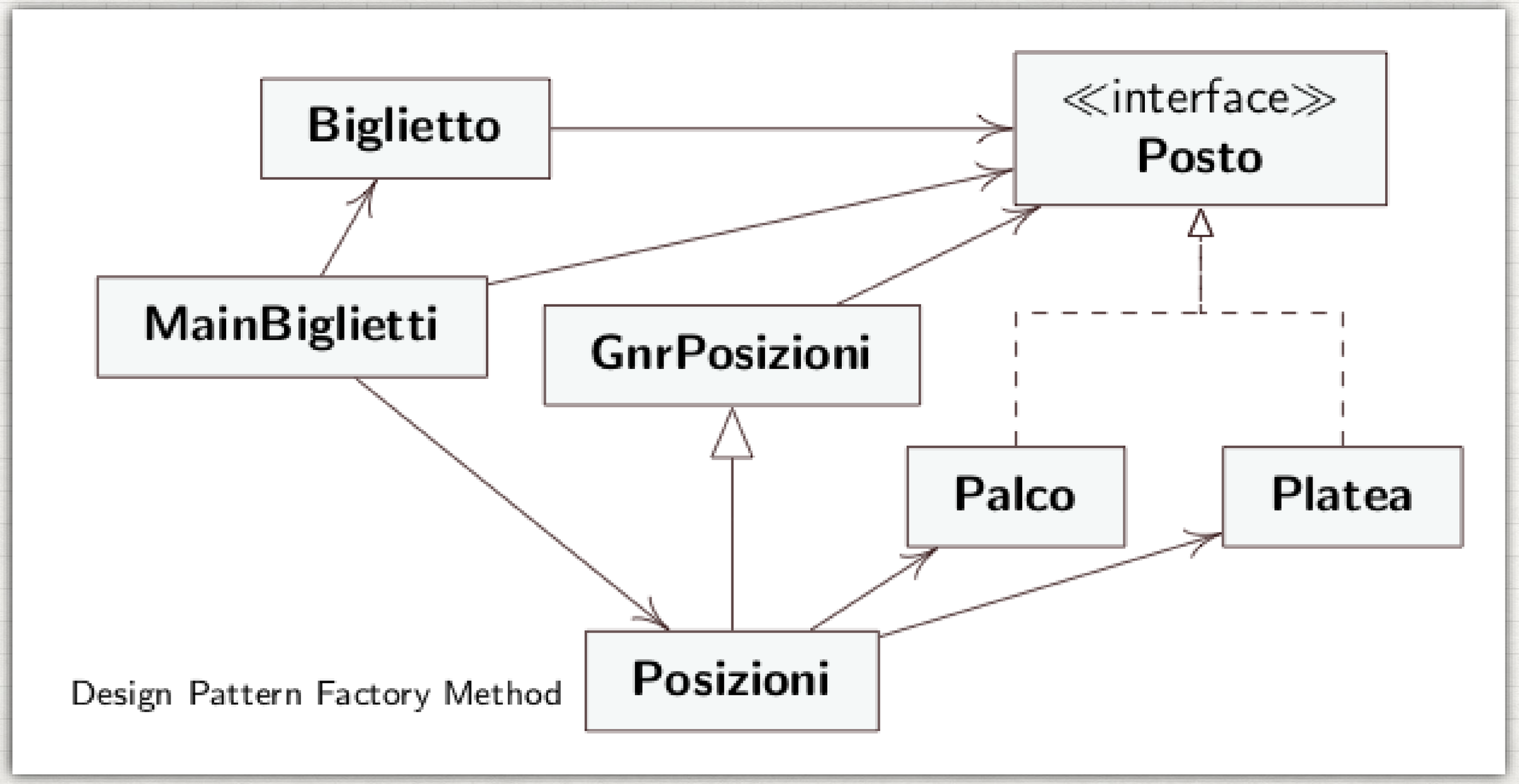
Lezione 2

- In questa lezione:
- Esempi di Applicazione con **Factory method**
  - **Versione standard**
  - **Con dependency injection e object pool**
- **Prototype** design pattern ed applicazione
- **Abstract factory** design pattern

prof. Andrea Calvagna
Corso di Ingegneria del Software
Universtità di Catania – DMI – 21 Aprile 2022

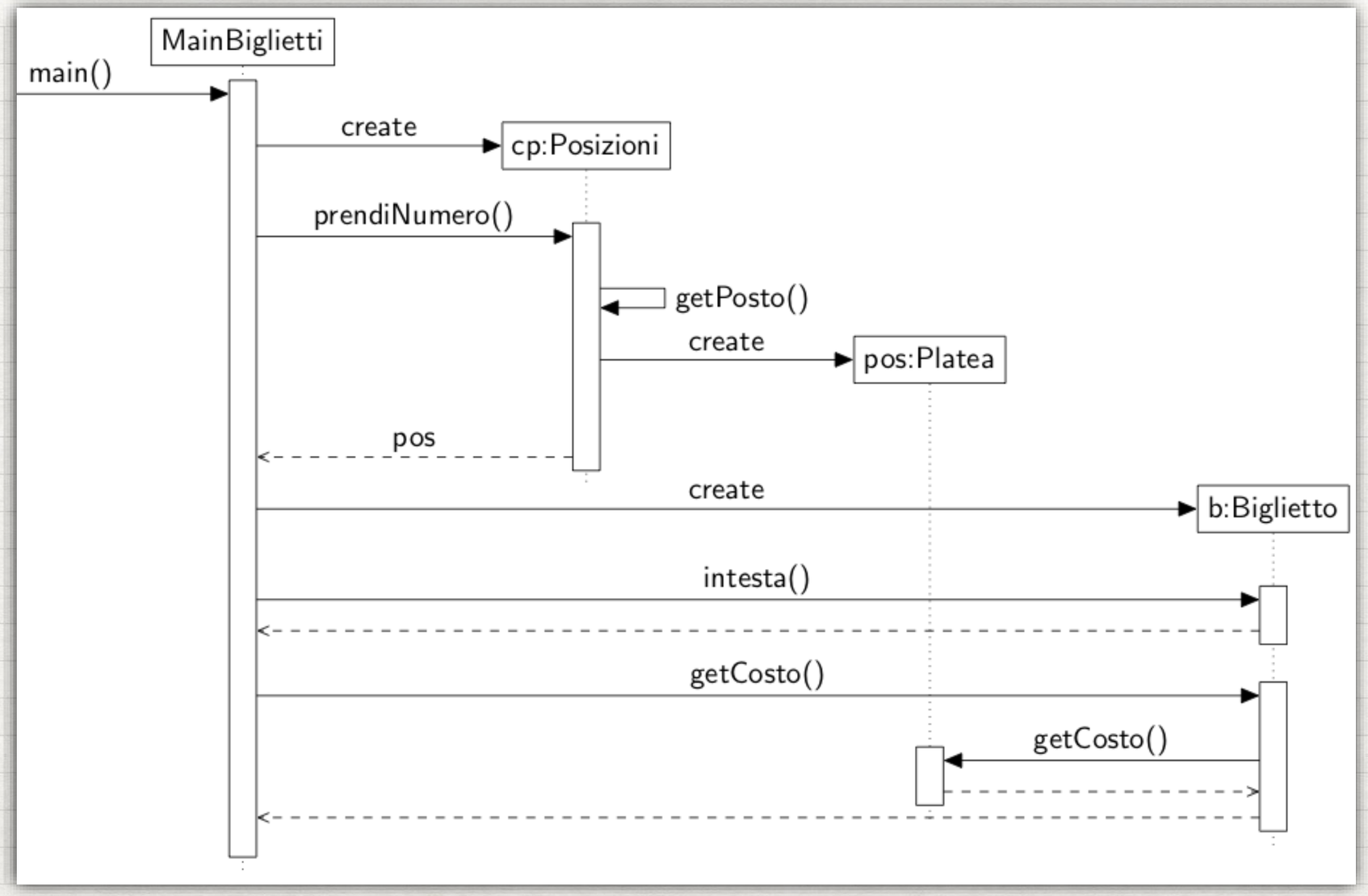
APPLICAZIONE BIGLIETTERIA : CLASS DIAGRAM



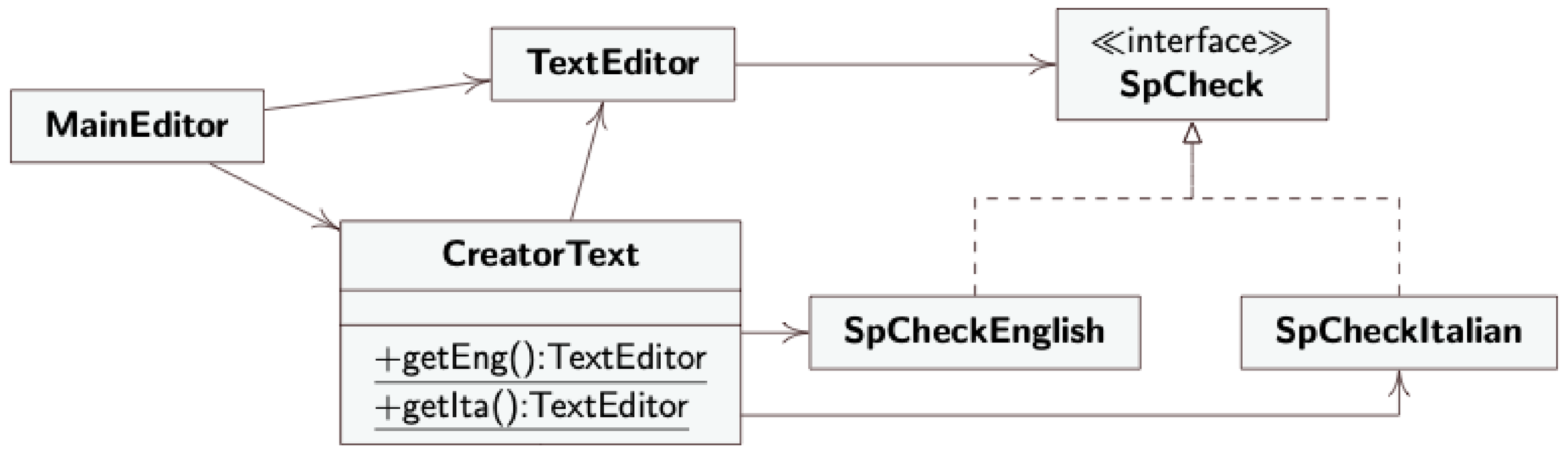
Visual Studio Code

- Codice sul sito dmi al seguente link: https://www.dmi.unict.it/tramonta/se/oop/appBiglietti.html

DIAGRAMMA DI SEQUENZA



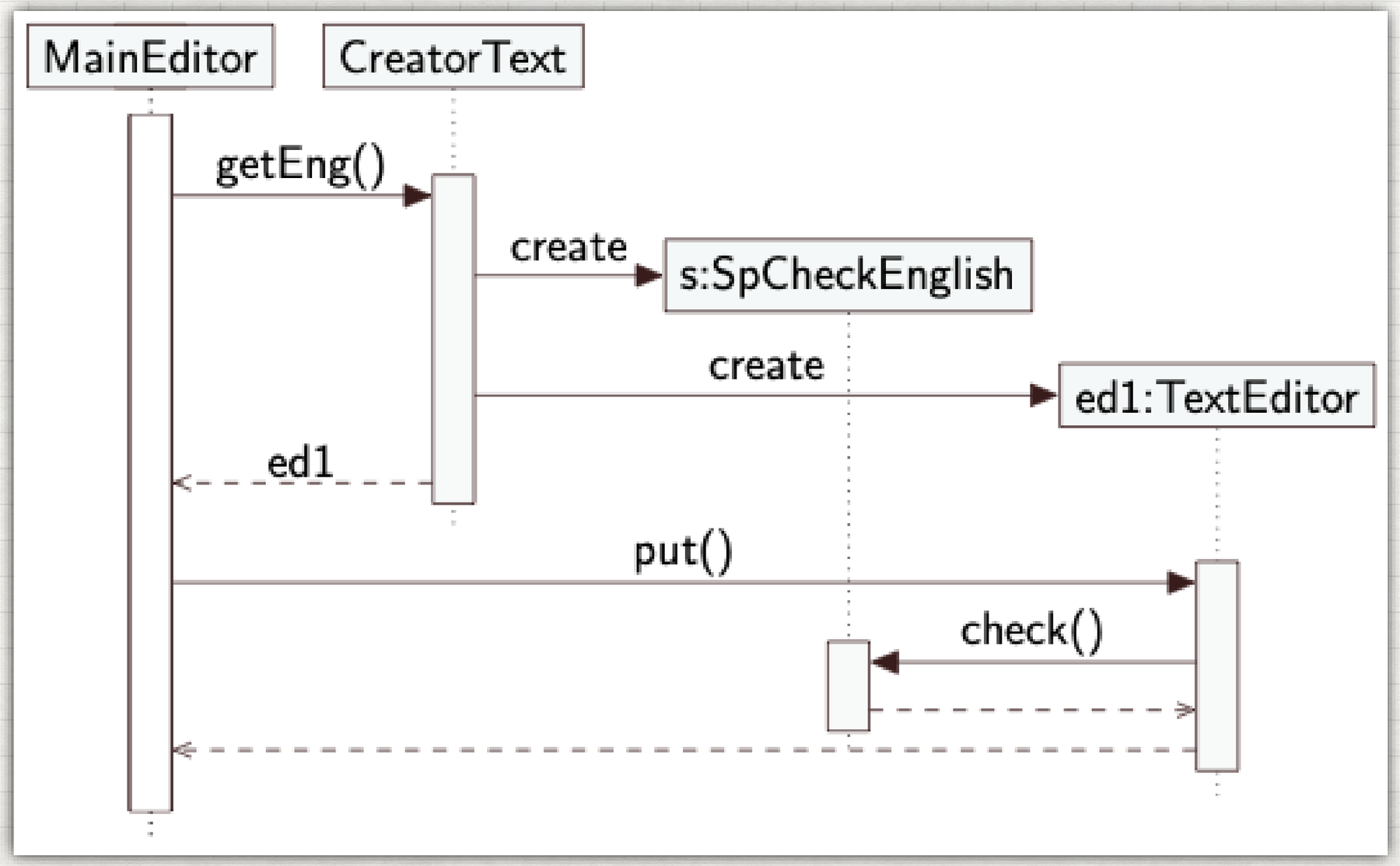
Applicazione Editor: Class diagram



Visual Studio Code

- Codice dell'applicazione sul sito dmi al seguente link: https://www.dmi.unict.it/tramonta/se/oop/appEditor.html

SEQUENCE DIAGRAM



GOF PATTERN CATALOG

		Purpose		
		Creational	**Structural**	**Behavioral**
Scope	**Class**	Factory Method (107)	Adapter (139)	Interpreter (243) Template Method (325)
	Object	Abstract Factory(87) Builder (97) Prototype (117) Singleton (127)	Adapter (139) Bridge (151) Composite (163) Decorator (175) Facade (185) Proxy (207)	Chain of Responsibility (223) Command (233) Iterator (257) Mediator (273) Memento (283) Flyweight (195) Observer (293) State (305) Strategy (315) Visitor (331)

Prototype

Intento:

Specificare il tipo di oggetti da usare usando istanze prototipali

Nuove Istanze sono clonate dal client a partire da tali istanze di riferimento già esistenti

- Il clients non crea mai istanze con “new”: non conosce il tipo esatto del prodotto che usa. Riceve un riferimento al clone di un prototipo

invece di **AbstractClass obj = new ConcreteSubClass();**
avrò **AbstractClass obj = prototype->clone();**

Nel FM (std) derivavo una sottoclasse per ridefinire il metodo creazionale

AbstractClass obj = getConcreteSubClass();

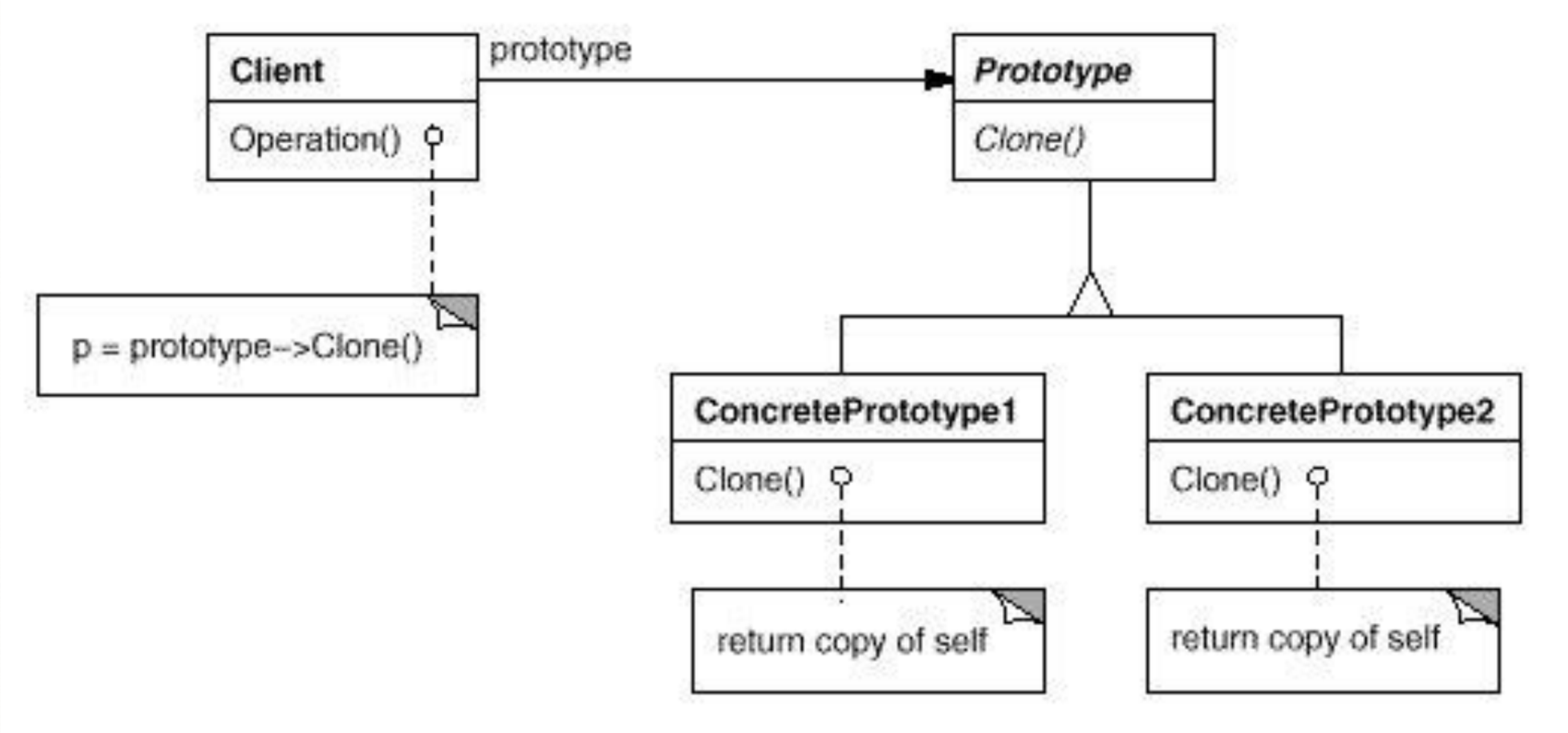
il prototipo invece viene passato come oggetto gia’ istanziato, da *clonare*

esempio

- Polizza Assicurativa
  - Un agente assicurativo ha un modello di polizza standard (prototipo)
  - lo clona e lo usa come base di partenza
  - Lo edita per customizzarlo al bisogno specifico

  La clonazione è una **operazione di duplicazione di un oggetto: una copia**

Prototype



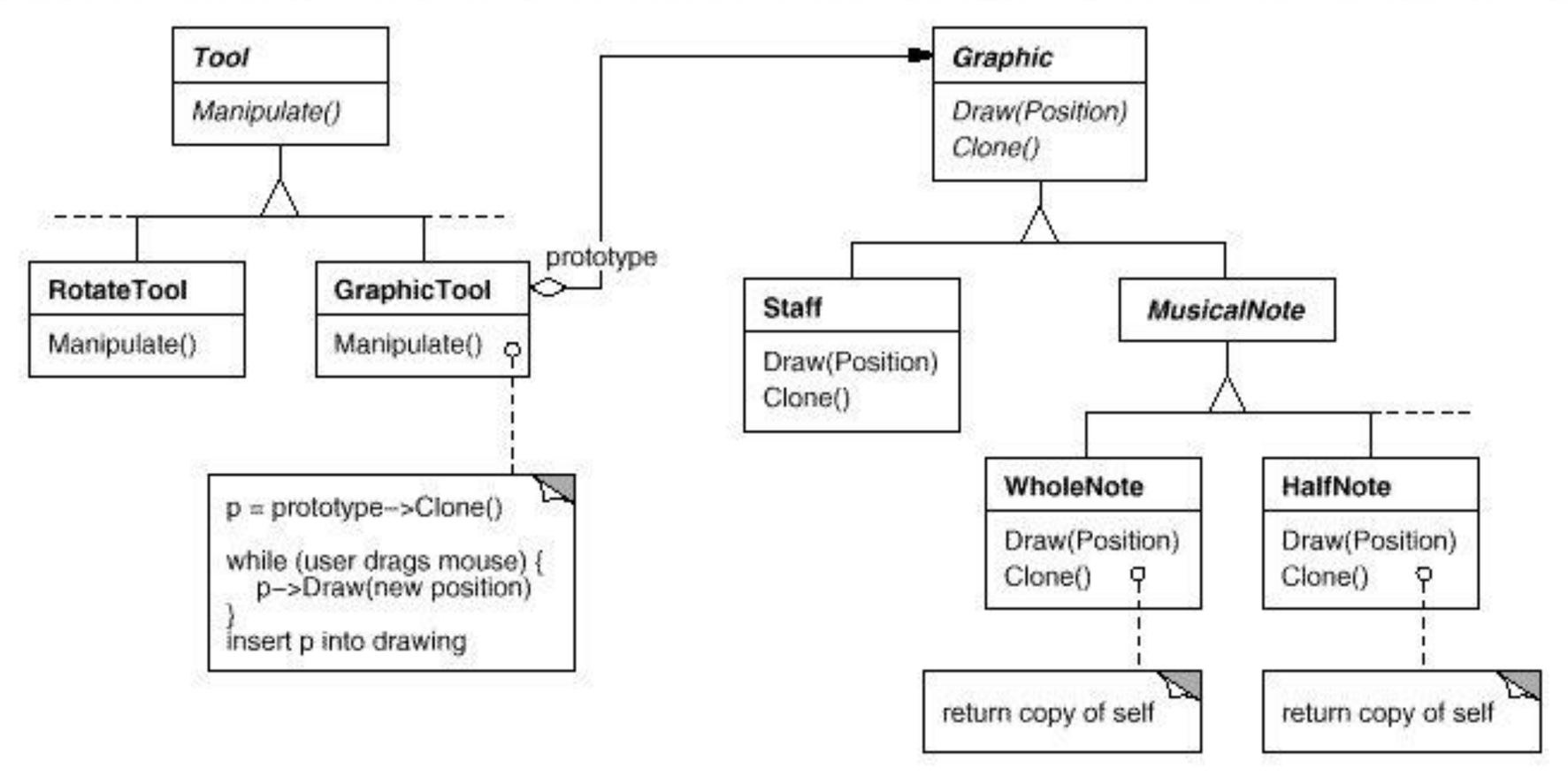
- Unico requisito è il metodo Clone() : ulteriori interfacce funzionali specifiche sono a parte
- ConcreteProto1 e CconcreteProto2 potrebbero anche essere molto diverse tra loro
- il client usa un clone di un oggetto esistente, ignorando quale sia la sua classe reale

Prototype

- **Applicabilità**
  - Quando un sistema dovrebbe essere indipendente dal prodotto concreto che usa
    - **e in più**…
  - la classe concreta da usare è nota solo a run-time: impossibile scrivere a priori sottoclassi (in stile FM) per incapsularne la creazione
    - Ricevo in un parametro un oggetto (prototipo) già istanziato, da clonare
  - **Oppure**, una classe ha solo in pochi stati possibili, quindi ho poche versioni di istanze tutte uguali
  - **Oppure** Se la creazione di nuove istanze di classe è costosa

    es.: il costruttore fa varie query a DB. Copiare una istanza già pronta è più performante

Classi con poche istanze distinte



WholeNote, HalfNote, etc… cambiano solo nello stato (durata) e nel glifo

potrei avere la sola classe **MusicalNote** e alcune sue istanze configurate diversamente (prototipi) da clonare a volontà

Si riduce il numero di classi nel sistema

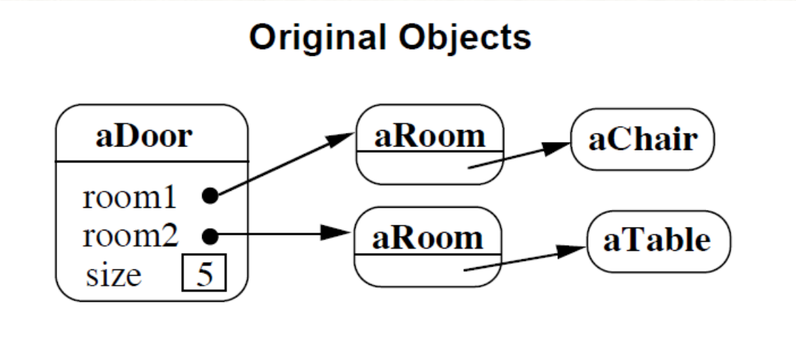
Relazione di aggregazione: riferimento ad un oggetto con ciclo di vita autonomo

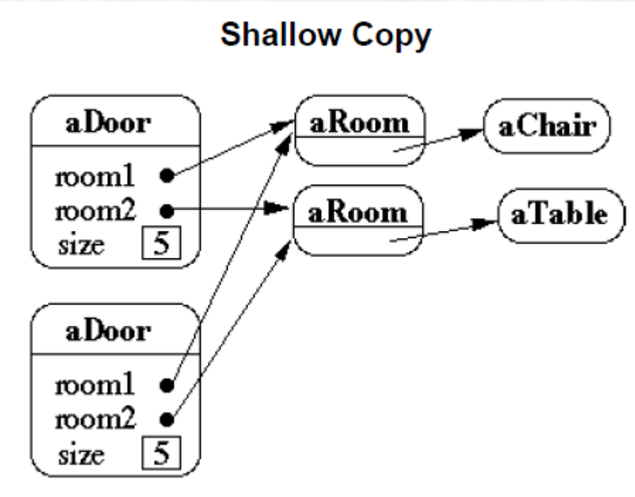
Prototipi in Java

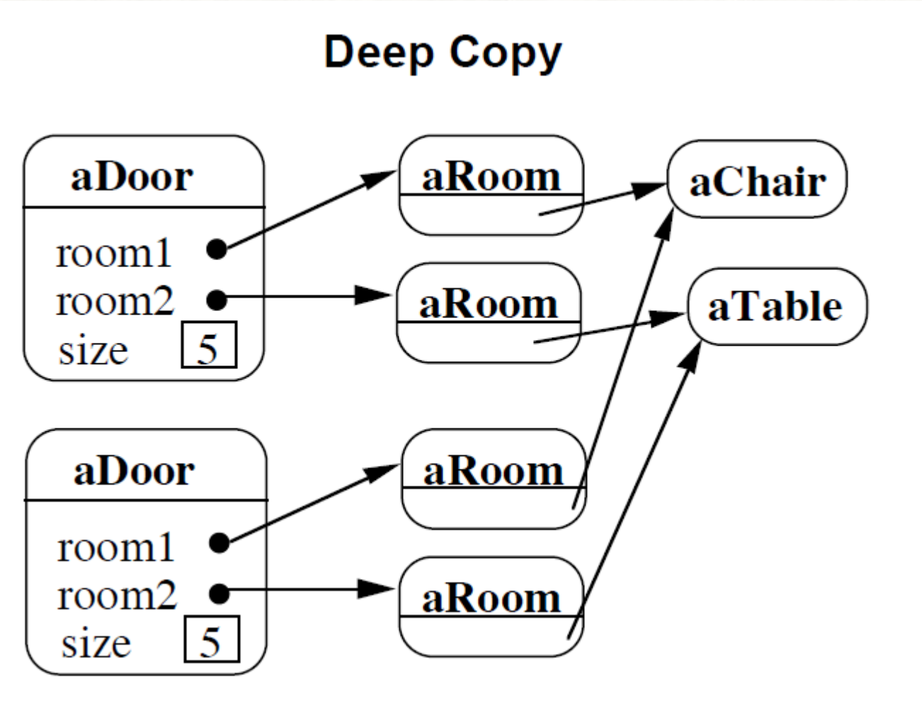
- **public interface Cloneable** (dal JDK 1.0)
  - È vuota
  - Una classe implementa **Cloneable** per indicare al metodo **Object.clone()** che la sua esecuzione è (non sempre) permessa:
    - copia valore per valore tutte le variabili di istanza
  - Altrimenti exception **CloneNotSupportedException**
  - metodo **Clone()** predefinito nella classe Object, non qui
  - override per **deep copy**: attenzione ai riferimenti circolari!

Shallow vs Deep copy

Per oggetti con struttura complessa personalizzo il Clone() per assicurare che la copia possa cambiare il suo stato in modo indipendente dal prototipo originale







Possibile non duplicare gli oggetti senza stato modificabile (attributi di istanza)

In Java

```
class Door implements Cloneable {
    Room room1;
    Room room2;

    public void Initialize( Room a, Room b){
        room1 = a; room2 = b;
    }

    public Object clone() throws CloneNotSupportedException {
        // method for deep copy
        Door cloned = super.clone();
        cloned.Initialize(room1.clone(), room2.clone());
        return cloned;

        // no need to implement this method for shallow copy
        //return super.clone();
    }
}
```

Conseguenze

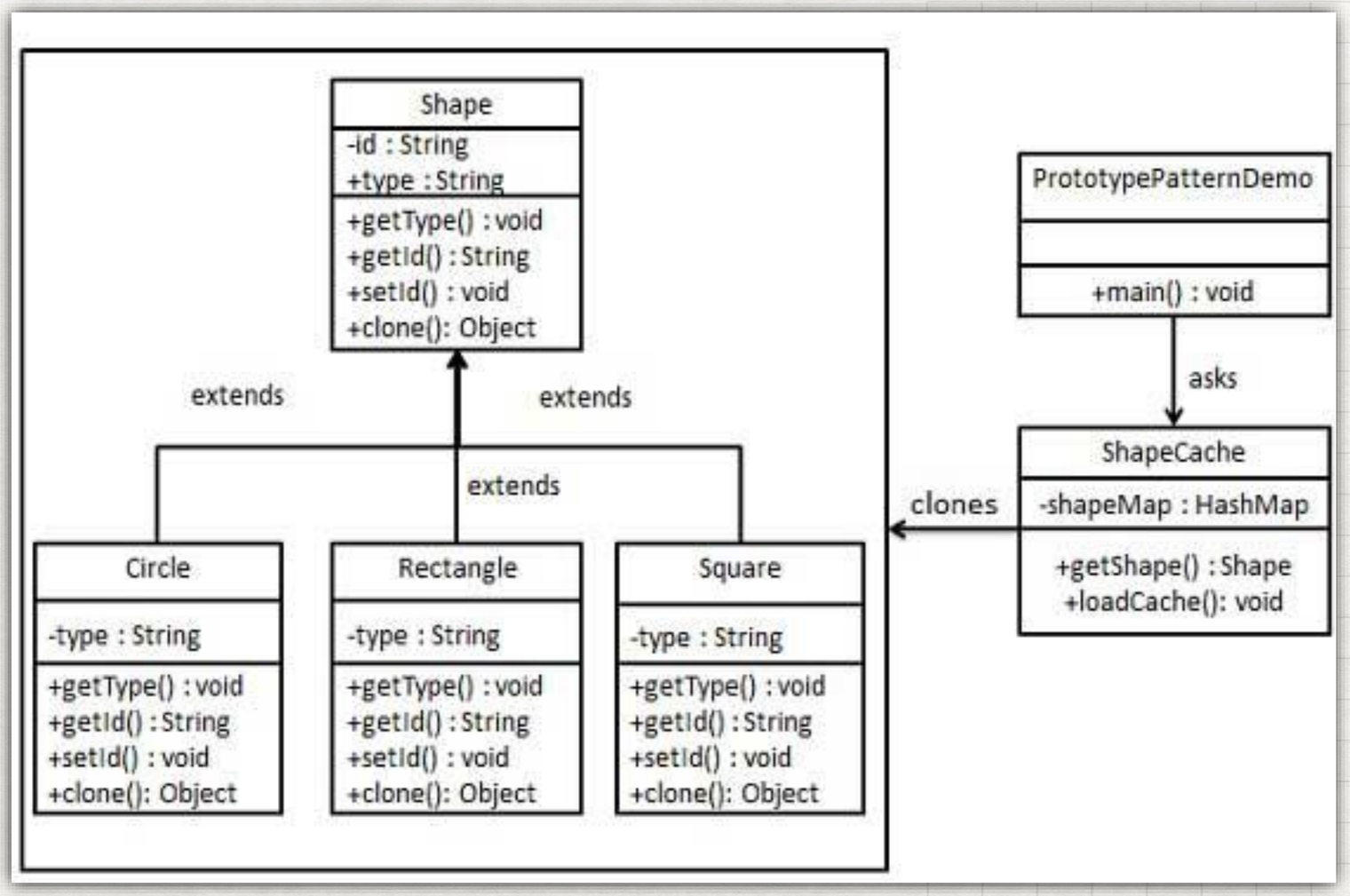
- - posso sostituire I prototipi facilmente (cambio l’oggetto o cambio la sua struttura interna) a run-time senza modifiche nel client
- - tengono celata la struttura (eventualmente) complessa al loro interno
- - ho meno bisogno di classi derivate, rispetto al factory method
- - devo implementare il metodo Clone()
- - devo poterli inizializzare, se necessario
- - a volte comodo un **prototype manager**

Prototype Manager

- In alcuni casi il numero di prototipi non è noto o fisso
- Uso un Catalogo (o registro) di prototipi clonabili
- E’ una memoria associativa dove posso registrare (o cancellare) nuovi prototipi sotto un etichetta o un codice simbolico
- Restituisce il riferimento ad un prototipo associato ad una data chiave
    - prototypeManager.get(“ProtoXY123”).clone()
- La chiave potrebbe essere il nome della classe cui appartiene

Esempio Prototype Manager

CLASS DIAGRAM



- Uso ShapeCache per ottenere cloni di varie forme
- Shape è una classe astratta che implementa *Clonable*
- ShapeCache usa una HashTable per mantenere gli originali dei vari prototipi
- Che sono in questo caso tutti di tipo Shape

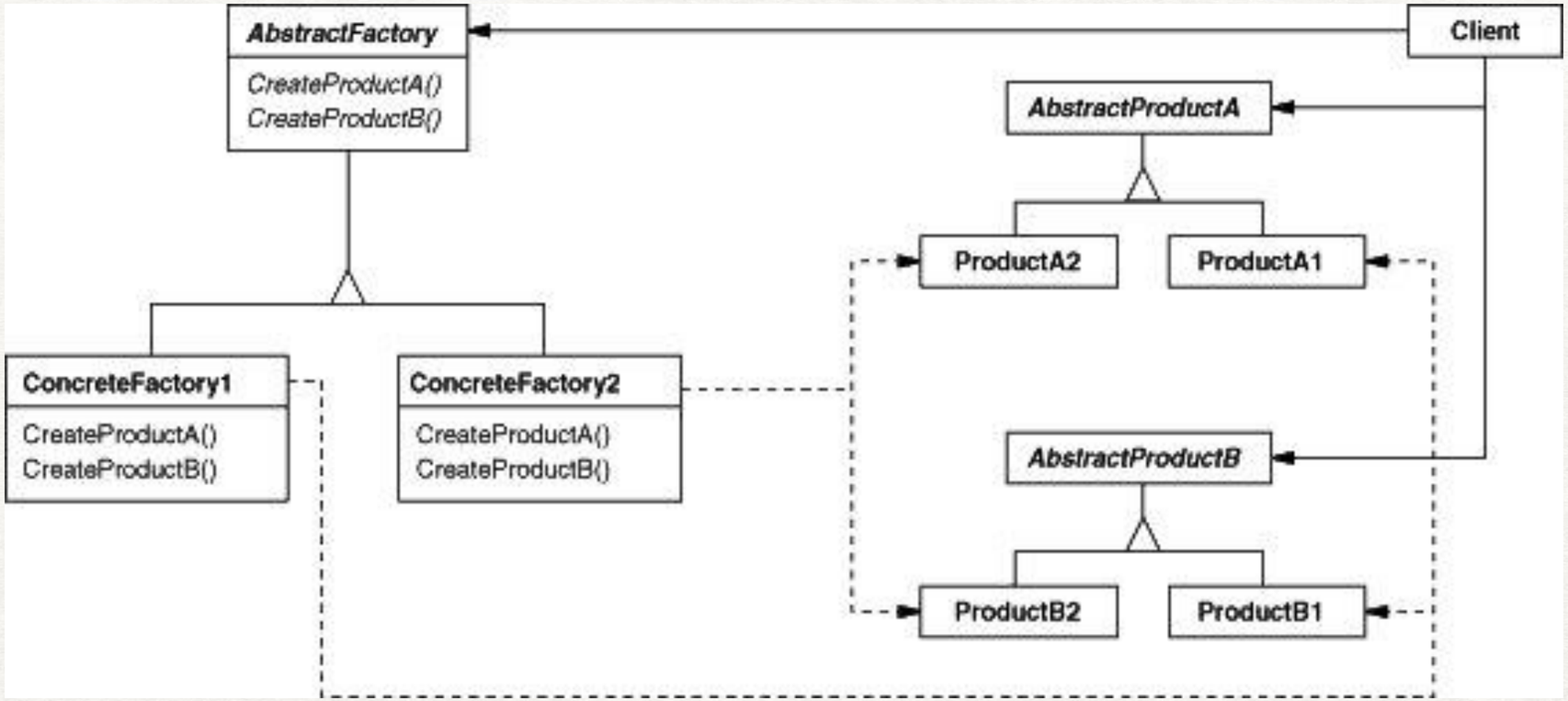
Visual Studio Code

Codice su internet al link:

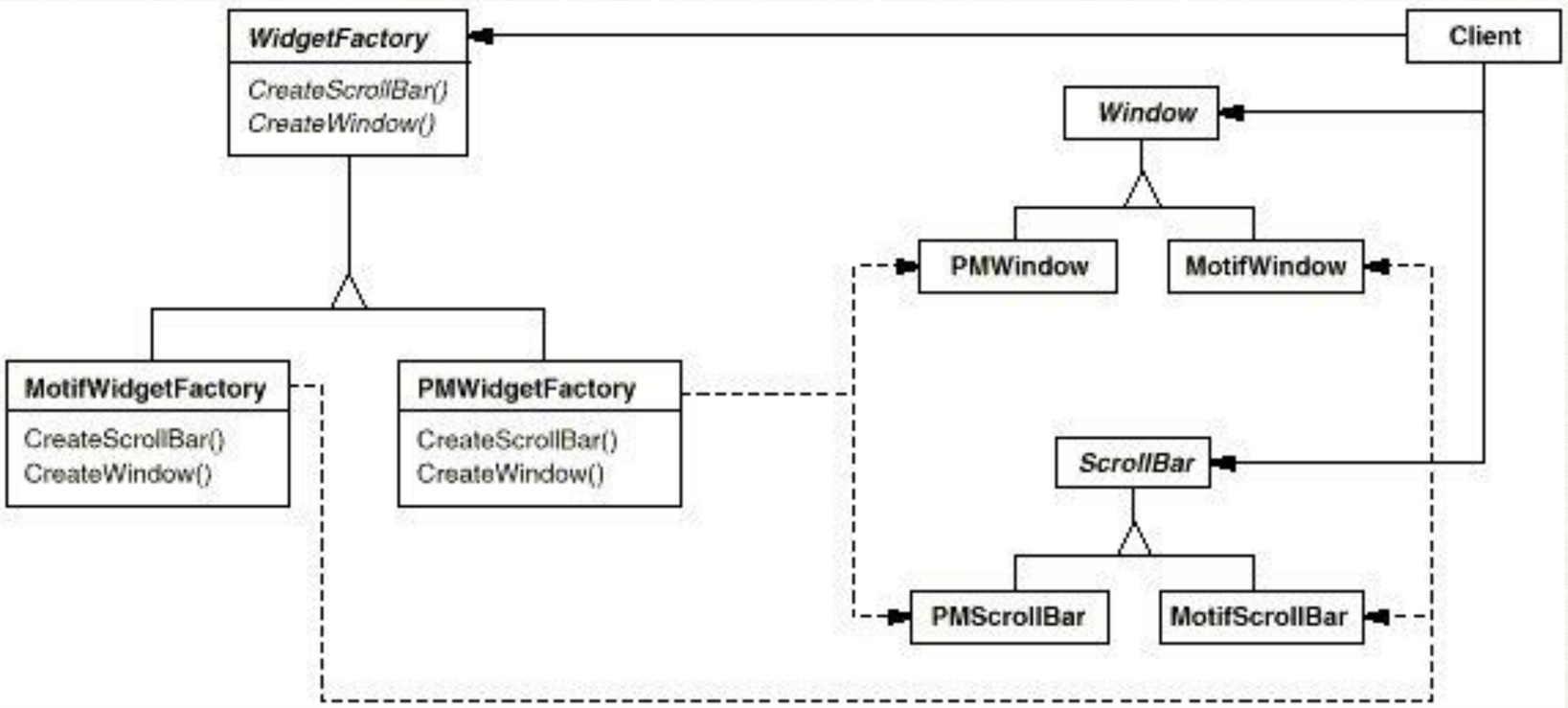
https://www.tutorialspoint.com/design_pattern/prototype_pattern.htm

Abstract factory (Kit)

- **Intento:**
- Fornire una interfaccia per la creazione di *famiglie di oggetti correlati* o dipendenti senza specificare le loro classi concrete



Esempio



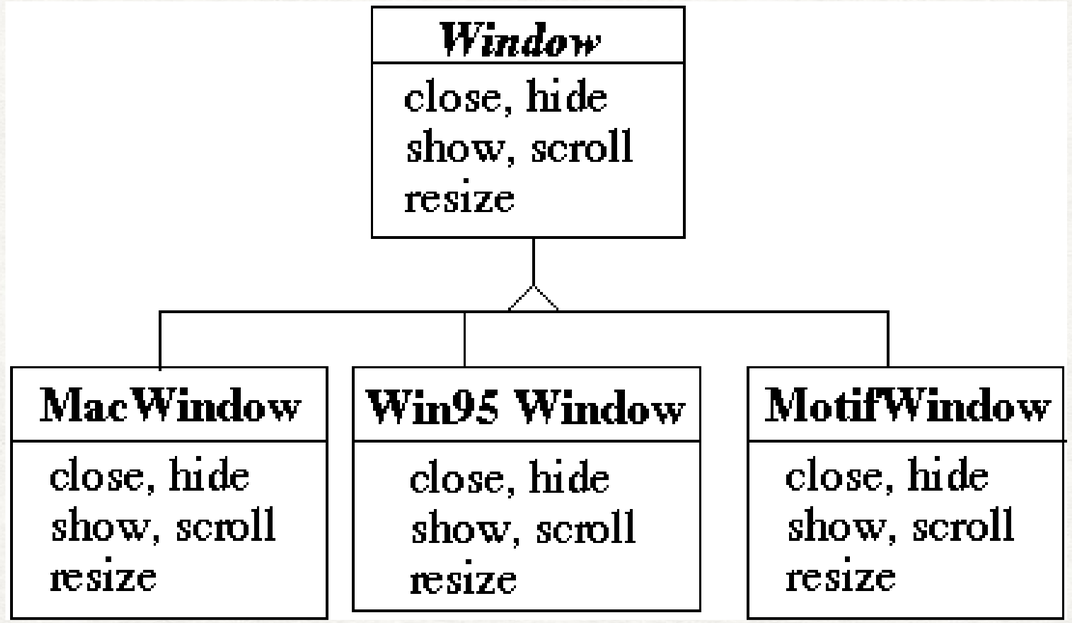
Voglio un applicazione grafica multipiattaforma: Mac, PC e Unix

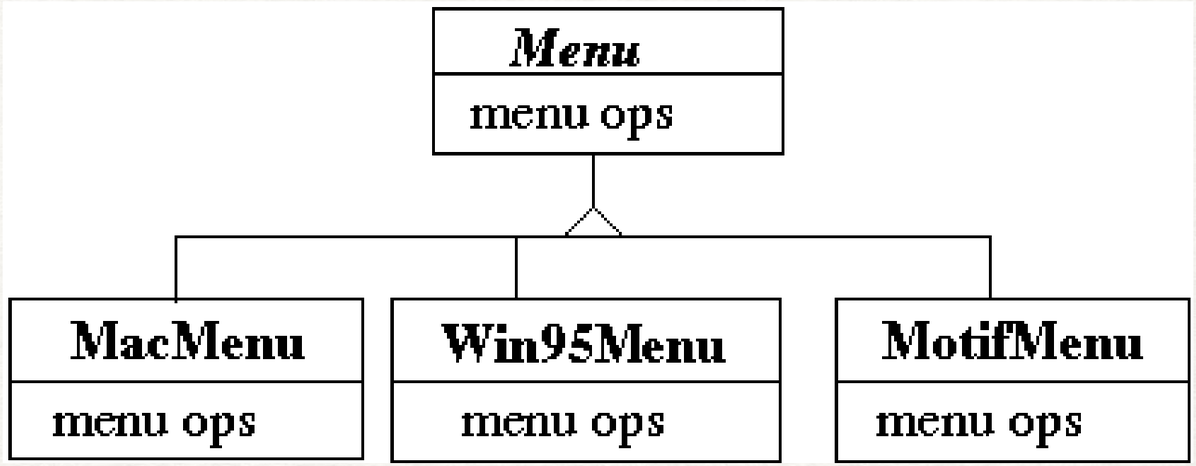
Devo disegnare Menu, Finestre e Pulsanti

Devono apparire nello stile della piattaforma usata

- Garantisce consistenza: uso nel client di prodotti tutti di una stessa famiglia
- E Flessibilità: Consentire facilmente di cambiare l'intera famiglia di prodotti

- Creo
  - Una interfaccia (o classe astratta) per ogni widget
  - Una classe concreta per ogni piattaforma





- L’applicazione può rivolgersi all’interfaccia
- Ad es.:

```
public void installDisneyMenu()
{
    Menu disney =  Crea un menu’ in qualche modo
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    disney.addItem( "Mickey Mouse" );
    disney.addGrayBar( );
    disney.addItem( "Minnie Mouse" );
    disney.addItem( "Pluto" );
    etc.
}
```

  - Come creare il menù in modo che: sia del tipo giusto e…
  - Riduco al minimo il numero di posti in cui rivelo la piattaforma

Uso l'Abstract Factory:

```
abstract class WidgetFactory {
    public Window createWindow();
    public Menu createMenu();
    public Button createButton();
}
class MacWidgetFactory extends WidgetFactory {
    public Window createWindow() { code to create a mac window }
    public Menu createMenu() { code to create a mac Menu }
    public Button createButton() { code to create a mac button }
}
class Win95WidgetFactory extends WidgetFactory {
    public Window createWindow() { code to create a Win95 window }
    public Menu createMenu() { code to create a Win95 Menu }
    public Button createButton() { code to create a Win95 button }
}
```

Nel client

```
public void installDisneyMenu(WidgetFactory myFactory) {
    Menu disney = myFactory.createMenu();
    disney.addItem( "Disney World" );
    disney.addItem( "Donald Duck" );
    ....
}
```

- Dobbiamo solo assicurarci che ogni applicazione istanzi la *factory* appropriata per la sua piattaforma e la passo come oggetto al resto del codice

- Appilcabilità:
  - Voglio indipendenza dal tipo concreto di prodotti che creo e uso
  - Possibilità di configurare il sistema con una tra varie famiglie di prodotti
  - famiglie di prodotti correlati sono state progettate per essere usati insieme e si vuole imporre questo vincolo di coerenza
  - fornire librerie di classi intercambiabili rivelando solo le interfacce (API)

- **Conseguenze**:
- AbstractFactory rimanda la creazione della versione di prodotti che devo usare **ad una istanza** di una sua sottoclasse ConcreteFactory.
- Rispetto al factory method, usa un oggetto e vi incapsula la crezione di molteplici prodotti.
- Consente di cambiare facilmente la famiglia di prodotti
- Difficile aggiungere in seguito nuovi prodotti (l'interfaccia è ciò su cui si basano i client)
  - In tal caso meglio un *prototype manager*

Varianti di implementazione

- 1) come set di factory methods: devo sottoclassare e fare ovverride dell'intera interfaccia
- 2) composition (inclusione) dei prodotti (2° principio): devo sottoclassare, ma non serve override; I factory methods sono nei prodotti
- 2.5) come set di factory methods overridden dalla sottoclasse, ma che ritornano Class meta-objects da istanziare
- 3) composizione con prototipi (delega ad altre istanze): non derivo sottoclassi, non faccio override (la factory è concreta)

Metodo 1) set di Factory Methods interni

abstract class **WidgetFactory** {
public Window createWindow();
public Menu createMenu();
public Button createButton();
}
class **MacWidgetFactory** extends WidgetFactory {
public Window createWindow() { code to create a mac window }
public Menu createMenu() { code to create a mac Menu }
public Button createButton() { code to create a mac button }
}
class **Win95WidgetFactory** extends WidgetFactory {
public Window createWindow() { code to create a Win95 window }
public Menu createMenu() { code to create a Win95 Menu }
public Button createButton() { code to create a Win95 button }
}

Metodo 2) set di Factory Methods esterni

```
abstract class WidgetFactory {
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public Window createWindow(){ return windowFactory.createWindow() }   //delega
    public Menu createMenu(){ return menuFactory.createMenu() }
    public Button createButton(){ return buttonFactory.createMenu() }
}

//sottoclasso per selezionare il delegato
class MacWidgetFactory extends WidgetFactory {
    public MacWidgetFactory() {
        windowFactory = new MacWindow();
        menuFactory = new MacMenu();
        buttonFactory = new MacButton();
    }
}
```

```
//prodotto concreto (delegato)
class MacWindow extends Window {
    public Window createWindow() { blah }
    public Window createFancyWindow() { blah }
    public Window createPlainWindow() {blah }
    etc.
}
```

Comodo se il codice per la creazione dei prodotti concreti si trova in altre classi (magari preesistenti)

Metodo 2.5) sottoclasso ed uso la reflection invece della delega

```
abstract class WidgetFactory {
    public Class windowClass();
    public Class menuClass();
    public Class buttonClass();

    public Window createWindow() { return windowClass().newInstance() }
    public Menu createMenu() { return menuClass().newInstance() }
    public Button createButton() { return buttonClass().newInstance() }
}

class MacWidgetFactory extends WidgetFactory {
    public Class windowClass() { return MacWindow.class; }
    public Class menuClass() { return MacMenu.class; }
    public Class buttonClass() { return MacButton.class; }
}
```

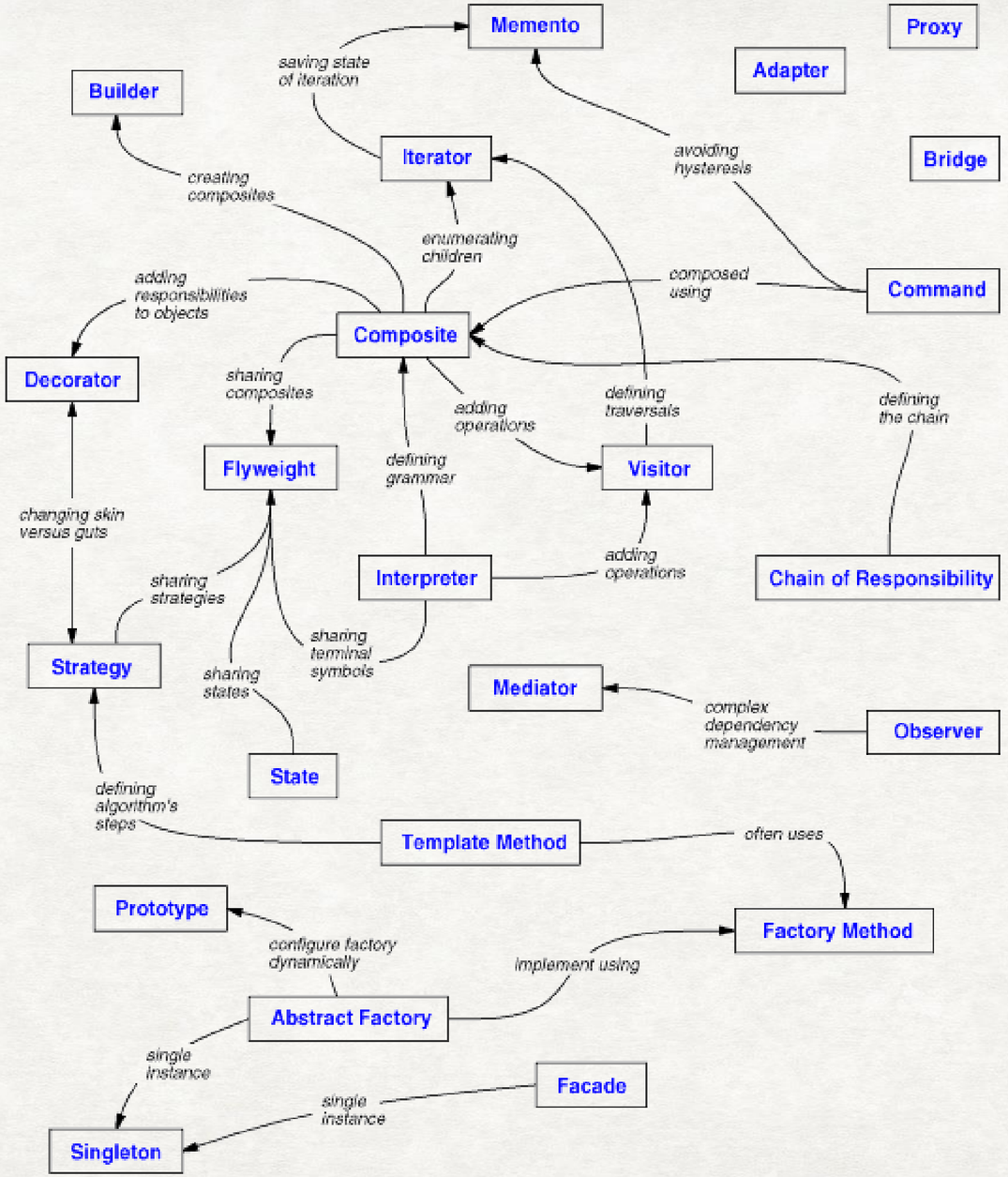
con programmazione generica e reflection elimino la sottoclasse (vedi FM)

Metodo 3) delega diretta a Prototipi

```
class WidgetFactory{ //non più abstract
    private Window windowFactory;
    private Menu menuFactory;
    private Button buttonFactory;

    public WidgetFactory( Window windowPrototype, Menu menuPrototype, Button buttonPrototype){
        this.windowFactory= windowPrototype;
        this.menuFactory= menuPrototype;
        this.buttonFactory= buttonPrototype;
    }
    public Window createWindow() { return windowFactory.createWindow() }
    public Window createWindow( Rectangle size){ return windowFactory.createWindow(size) }
    public Window createFancyWindow(){ return windowFactory.createFancyWindow() }
    etc.
}
```

Non serve sottoclassare WidgetFactory ma inizializzarla con le Giuste istanze cui delegare (prototipi di Factory)


Memento
Proxy
saving state of iteration
Builder
Adapter
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
adding operations
sharing strategies
Interpreter
Chain of Responsibility
sharing terminal symbols
Strategy
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton