# Factory method & Abstract Factory



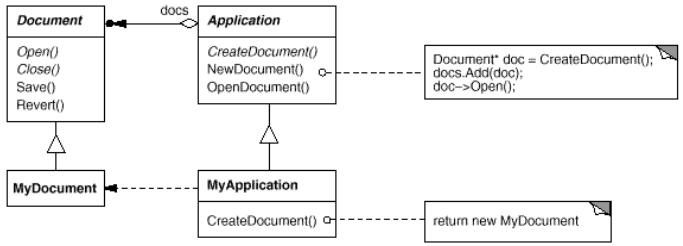
# Factory Method

- Applicabilità
  - In un algoritmo, so quando deve essere creato un oggetto di servizio, ma non so o non voglio stabilire (e/o voglio poter cambiare) quale classe istanziare.
    - Primo principio: programma usando *interfacce*: so cosa devo fare e che devo farlo adesso ma non voglio ancora decidere come
    - Creo le condizioni affinché questa decisione sia fatta altrove



# Factory Method

- Incapsulo la creazione dentro un metodo (dà l'*oggetto come valore di ritorno*): saranno le sottoclassi a decidere la classe da istanziare
- Il mio algoritmo (astratto) può essere scritto conoscendo (e/o usando) solo l'interfaccia dell'oggetto di servizio: *primo principio*

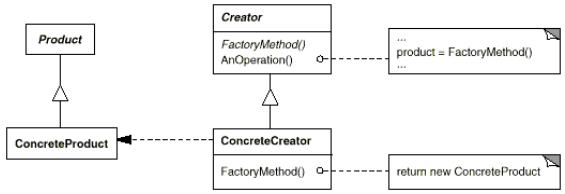



NB: questo non è UML



# Factory Method

- Schema del pattern
  - **Product** è l'interfaccia comune degli oggetti creati da factoryMethod()
  - **ConcreteProduct** è un'implementazione di Product
  - **Creator** dichiara il factoryMethod(), quest'ultimo ritorna un oggetto di tipo Product. Usa l'oggetto Product nel suo codice
  - **ConcreteCreator** implementa il factoryMethod(), o ne fa override, sceglie quale ConcreteProduct istanziare e ritorna tale istanza




NB: questo non è UML

```java
public interface Product {
    void request();
}

class CProduct implements Product {
    public void request() {
        System.out.println("CProduct");
    }
}

public class Client { //client application
    public static void main(String[] args) {
        Creator myApp = new CCreator(); // incapsulation!
        myApp.AnOperation(); //use the product service indirectly
         ...

        Product myProduct = myapp.getProduct(); // incapsulation!
        myProduct.request();     // use the product service directly
    }
}
```

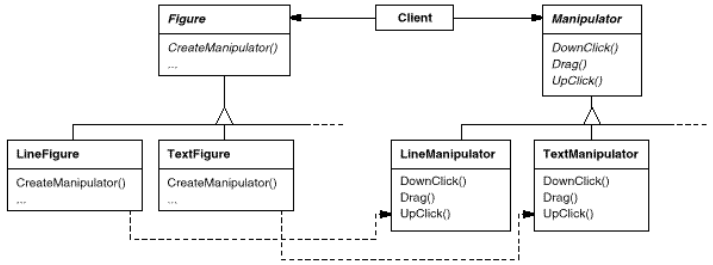
```java
abstract class  Creator {
    abstract public Product getProduct();

    public void AnOperation(){
          Product p = this.getProduct();
          p.request();
    }
}

public class CCreator extends Creator {
    @Override
    public Product getProduct() {
     return new CProduct();
    }
}
```

Sottoclassando Creator fornisco versioni alternative di AnOperation/request



# Gerarchie di classi parallele

- Il factory method non è chiamato per forza solamente dai Creators.
- Altri clients possono trovare i factory methods utili, specialmente nei casi di gerarchie di classi parallele

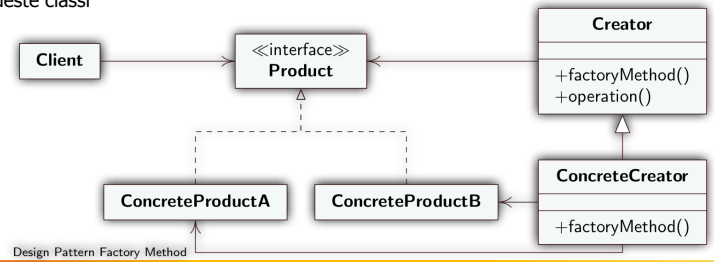



NB: questo non è UML



# Factory Method

- Applicabilità:
  - Una classe non è in grado di sapere in anticipo la classe dell'oggetto che deve creare
  - Una classe vuole delegare alle sottoclassi future la scelta di quale oggetto creare e utilizzare
  - Una classe (creator/client) delega responsabilità (metodi) a una (o più) classe di supporto (product) e vuole localizzare in un punto (il factory method) la scelta/conoscenza di quali siano queste classi

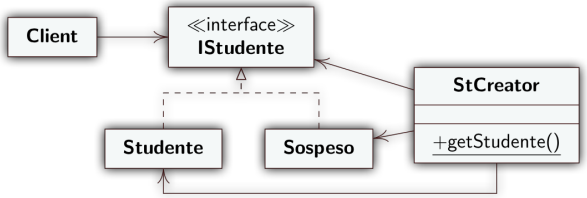



Design Pattern Factory Method



## Variante: Il `factoryMethod()` come metodo `static`

- `Se devo solo di incapsulare una scelta di classe`
- `Ho un solo il concreteCreator`
- `Non devo istanziare il creator`
- `Non estensibile`



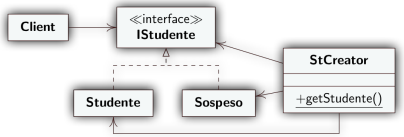
# FM vs Polimorfismo

```
public class Sospeso extends BasicStud  {
    public void nuovoEsame(String m, int v) {
           System.out.println("Non possibile");
       }
}

public class Studente extends BasicStud{
  private List<Esame> esami = new ArrayList<>();
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
```

```
public class Client {
     BasicStud s = new Studente();
    public void registra(){
    s.nuovoEsame("Maths", 8);
     }
}
```



```
public interface IStudente {
  public void nuovoEsame(String m, int v);
  public float getMedia();
}

public class Studente implements IStudente {
  private List<Esame> esami = new ArrayList<>();
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
  public float getMedia() {
    if (esami.isEmpty()) return 0;
    float sum = 0;
    for (Esame e : esami) sum += e.getVoto();
    return sum / esami.size();
  }
}

public class Sospeso implements IStudente {
  private float media;
  public Sospeso(float m) {
    media = m;
  }
  public void nuovoEsame(String m, int v) {
    System.out.println("Non e' possibile sostenere
esami");
  }
  public float getMedia() {
    return media;
  }
```

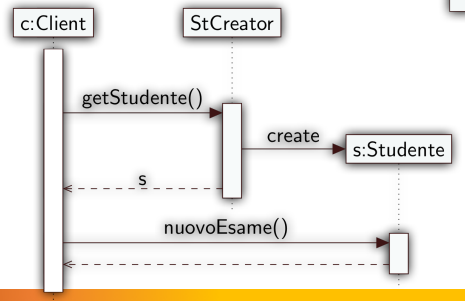


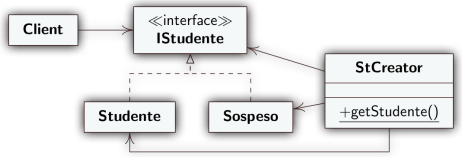

```
public class StCreator {
  private static boolean a = true;

   public static IStudente getStudente() {
     if (a) return new Studente();
     return new Sospeso(0);
   }
}

public class Client {

public void registra() {
    IStudente s = StCreator.getStudente();
    s.nuovoEsame("Maths", 8);
  }
}
```



# Interazione

- Nel precedente esempio di codice, l'interfaccia IStudente svolge il ruolo Product, le classi Studente e Sospeso svolgono il ruolo ConcreteProduct, e la classe StCreator svolge il ruolo ConcreteCreator






# Variante: creator non astratto

```
public class Creator {
  public IStudente getStudente() {
    return new Studente();
  }
}

public class SoCreator extends Creator{
  public IStudente getStudente() {
    return new Sospeso();
  }
}
```

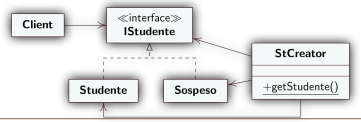
```
public class Client {
  public void registra( Creator sc) {
    if (sc==null) sc = new SoCreator();
    . . .
    IStudente s = sc.getStudente();
    s.nuovoEsame("Maths", 8);
  }
}
```

# Variante: factory method parametrico

- Istanzia una tra piu' classi possibili
- Scelta dal client, direttamente o indirettamente
- Due versioni
  - Creator non astratto, il client seleziona il prodotto
  - Creator astratto, ConcreteCreator seleziona il prodotto
    - il client seleziona il ConcreteCreator da istanziare



```
public interface IStudente {
  public void nuovoEsame(String m, int v);
  public float getMedia();
}

public class Studente implements IStudente {
  private List<Esame> esami = new ArrayList<>();
  public void nuovoEsame(String m, int v) {
    Esame e = new Esame(m, v);
    esami.add(e);
  }
  public float getMedia() {
    if (esami.isEmpty()) return 0;
    float sum = 0;
    for (Esame e : esami) sum += e.getVoto();
    return sum / esami.size();
  }
}

public class Sospeso implements IStudente {
  private float media;
  public Sospeso(float m) {
    media = m;
  }
  public void nuovoEsame(String m, int v) {
    System.out.println("Non e' possibile
                        sostenere esami");
  }
  public float getMedia() {
    return media;
  }
```




```
public class StCreator {

   public static IStudente getStudente(boolean a) {
     if (a) return new Studente();
     return new Sospeso(0);
   }
}

public class Client {

public void registra() {
    IStudente s = StCreator.getStudente(true);
    s.nuovoEsame("Maths", 8);
  }
}
```

Vediamo altro esempio (appEditor) su vscode...



# Variante: creator generico

- Derivo classi ConcreteCreator collegate al prodotto giusto
- il client seleziona indirettamente il prodotto giusto da usare
- Non c'è dipendenza diretta del client col prodotto
- Esempio su vscode

```
public interface Product {
    void request();
}
class CProduct implements Product {
    public void request() {
        System.out.println("CProduct");
    }
}

abstract class Creator <P extends Product>{
    abstract public Product getProduct();

    public void doSomething(){
          P p = this.getProduct();
          p.request();
    }
}

public class ConcreteCreator extends Creator<CProduct> {
    @Override
    public CProduct getProduct() { return new CProduct();}
}
```

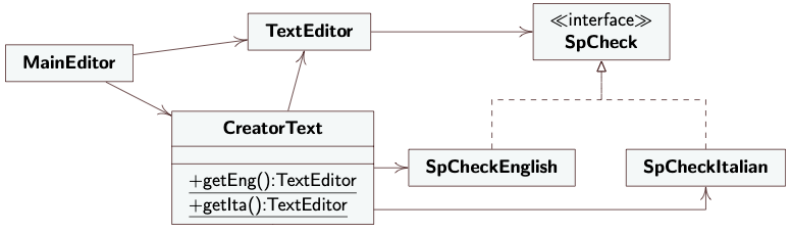
```
public class Client { //client application
  public static void main(String[] args){
     Creator myApp = new ConcreteCreator();

     myApp.doSomething();
     Product p = myapp.getProduct()
     p.request();
  }
}
```

# Dependency Injection

- Il design pattern Factory Method può essere usato per inserire le dipendenze necessarie ad altri oggetti (istanze di ConcreteProduct) : Dependency injection
  - Una classe C usa un servizio fornito da S (ovvero C dipende da S)
  - Esistono varie implementazioni di S (siano esse: S1, S2), la classe C non deve dipendere dalle implementazioni S1, S2
  - In un terzo contesto, creo l'istanza di C e la configuro con l'implementazione di S con cui desidero che operi: gli passo una istanza già creata di S1 o S2. Ho legato C ad S1 o S2
- Il pattern classico vorrebbe due concreteCreator C1 e C2:
- La dipendenza tra i due paralleli non è statica, viene "iniettata" attraverso un parametro, ad es. il costruttore al momento della creazione, per creare un legame tra due classi concrete di tipo composizione



# Esempio: Applicazione Editor





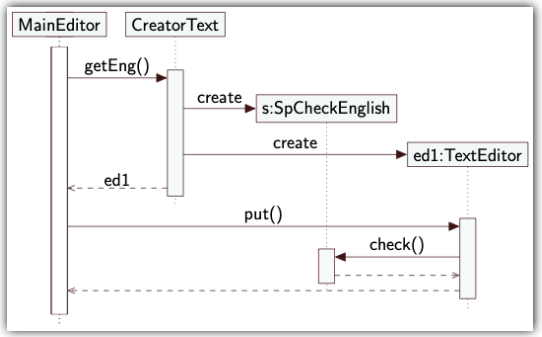

# Esempio Dep. Injection

```
public class TextEditorA implements TextEditor { //TextEditorA è un ConcreteProduct
   private SpellingCheck speller;
   public TextEditorA(SpellingCheck sp) {  // inserisco la dipendenza fornendo al
      speller = sp;                        // costruttore l'istanza del servizio sp
   }
   public void put(String s) {
      if (speller.check(s)) ...
      else ...
   }
}
```

```
public class CreatorText {  // CreatorText è un ConcreteCreator
   public static TextEditor getEnglishEditor() {
      return new TextEditorA(new SpCheckEnglish());
   }
   public static TextEditor getItalianEditor() {
      return new TextEditorB(new SpCheckItalian());
   }
}
```



# Sequence diagram



# Object Pool

- Un object pool è un deposito di istanze già create, una istanza sarà estratta dal pool quando una classe client ne fa richiesta
  - Il pool può crescere o può avere dimensioni fisse
    - Dimensioni fisse: se non ci sono oggetti disponibili al momento della richiesta, non ne creo di nuovi
  - **Il client restituisce al pool l'istanza usata quando non più utile**

- Il design pattern Factory Method può implementare un object pool
  - I client fanno richieste, come visto prima per il Factory Method
  - I client dovranno dire quando l'istanza non è più in uso, quindi riusabile
  - Lo stato dell'istanza da riusare potrebbe dover essere riscritto
  - L'object pool dovrebbe essere unico -> uso un Singleton

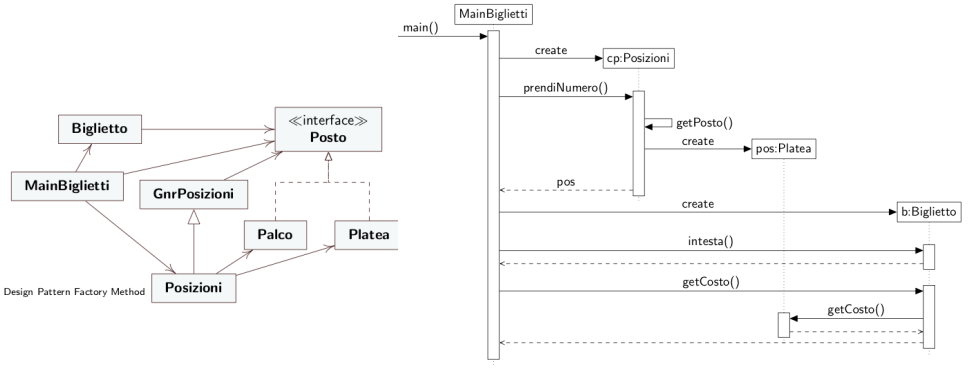


# Esempio di Object Pool

```
import java.util.LinkedList;
// CreatorPool è un ConcreteCreator e implementa un Object Pool

public class CirclePool extends ShapeCreator {
 private LinkedList<Shape> pool = new LinkedList<Shape>();

 // getShape() è un metodo factory che ritorna un oggetto prelevato dal pool
 public Shape getShape() {
      Shape s;
      if (pool.size() > 0) s = pool.removeFirst();
      else s = new Circle();  //concrete product
      return s;
 }

 // releaseShape() inserisce un oggetto nel pool
 public void releaseShape(Shape s) {
      pool.addLast(s);
 }
 }
```



# Esempio app Biglietti: Factory Method con Object Pool






# Variante: Creator con riflessione

- Il Client specifica come parametro il prodotto
- Non serve derivare sottoclassi di Creator: un solo creator "generico"
- Si dipendere <**per nome**> dai concrete product

```
Creator myApp = new Creator<CProduct>();
```

Vedi esempio su vscode

```java
import java.lang.reflect.InvocationTargetException;
class  Creator {
 public Product getProduct(Class<?> cp){
   try {
     return (Product)
          cp.getDeclaredConstructor().newInstance();
   } catch (InstantiationException |
       OMISSIS … SecurityException e) {
       e.printStackTrace();
      return null;
   }
 }

  public void AnOperation(String cn)
            throws ClassNotFoundException{
     Class<?> c = Class.forName(cn);
     Product p = this.getProduct(c);
     p.request();
  }
  public void AnOperation(Class<?> cp){
     Product p = this.getProduct(cp);
     p.request();
  }
}
```

```java
public class Client {
    public static void main(String[] args) {
        Product p;
        Creator myApp = new Creator();


        p = myApp.getProduct(CProduct.class);
        p.request();

        myApp.AnOperation(CProduct.class);

        try {
            myApp.AnOperation("CProduct");
        } catch (ClassNotFoundException e) {
            e.printStackTrace();
        }
    }
}
```

NB: Possibile anche parametrizzare direttamente il costruttore

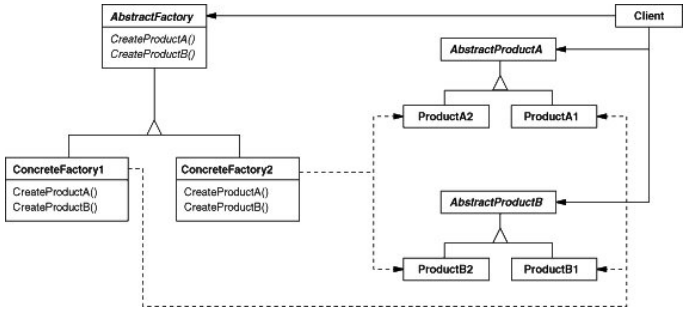
```
new Creator(CProduct.class);
new Creator("Cproduct");
new Creator<CProduct>();
```

Ricordiamo che passare nel costruttore istanze già create è una INIEZIONE di DIPENDENZA: `new Creator(new Cproduct());`



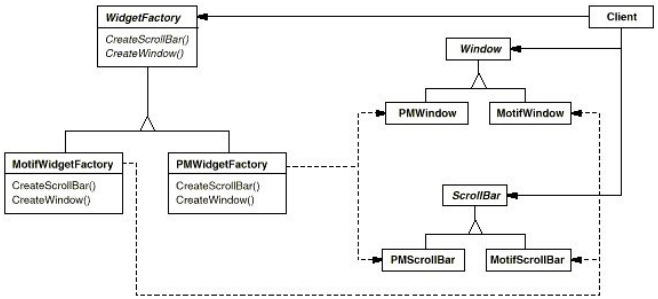
# Abstract Factory (Kit)

- 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

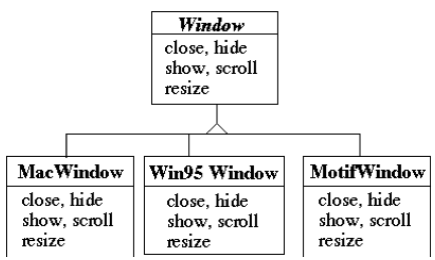
Devo disegnare Menu, Finestre e Pulsanti

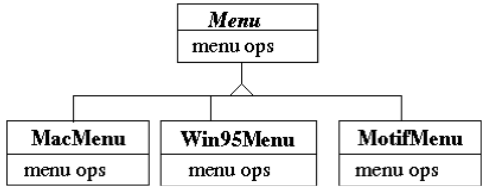
Devono apparire nello stile della piattaforma usata



# Esempio di applicazione

- 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…
  - Ridurre al minimo il numero di posti in cui rivelo la scelta di tipo?



# Nel client

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

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



# Implementazione classica

```
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 }
}
```



# Abstract Factory: applicabilità

- 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)

# Varianti di implementazione

- 1) classica: come set di factory methods: devo sottoclassare e fare ovverride dell'intera interfaccia. Le factory sono interne…
- 2) composition (inclusione) dei prodotti (2° principio): devo sottoclassare, ma non serve override; Le factory sono esterne (ad es. se preesistenti)
- 2.5) come set di factory methods overridden dalla sottoclasse, ma che ritornano Class meta-objects da istanziare



# Classica: 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 WinWidgetFactory extends WidgetFactory {
    public Window createWindow() { code to create a Windows window }.
    public Menu createMenu() { code to create a Windows Menu }
    public Button createButton() { code to create a Windows button }
}
```



# Variante 2: set di Factory Methods esterni

```
abstract class WidgetFactory {
        Window windowFactory;
        Menu menuFactory;
        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 MacWindowFactory();
                menuFactory = new MacMenuFactory();
                buttonFactory = new MacButtonFactory();
        }
}
```

```
//prodotto concreto (delegato) preesistente
class MacWindowFactory extends WindowFactory {
     public MacWindow() { code to build a MacWindow…}
      public Window createWindow() { blah }
      public Window createFancyWindow() { blah }
      public Window createPlainWindow() {blah }
      etc.
}
```

Vediamo esempio su vscode…



# Variante 2.5: sottoclassi con uso di reflection

```
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; }
}
```

Inserito in precedente esempio su vscode…