



UNIVERSITÀ
DEGLI STUDI
FIRENZE

**Scuola di
Ingegneria**

Ingegneria Del Software: Ferrari Factory

PROGETTO PER IL CORSO DI INGEGNERIA DEL SOFTWARE INERENTE AL
PRIMO TIPO DI ELABORATO, CONSISTENTE NELLA COMPOSIZIONE TRA I
DESIGN PATTERNS OBSERVER, ABSTRACT FACTORY, STRATEGY E SINGLETON

Andrea Rossini | 7004777 | 21/04/2022

Sommario

1) Introduzione e presentazione dell'elaborato	2
2) Excursus teorico sui design patterns usati e contesto d'uso	2
2.1) Observer	2
2.2) Abstract Factory	4
2.3) Strategy	5
2.4) Singleton	6
3) Use Case Diagram	6
4) Progettazione	7
4.1) Class Diagram	7
4.2) Sequence Diagram	8
5) Implementazione	9
5.1) Catalogo	9
5.2) Observer	10
5.3) Subject	10
5.4) Acquirente	10
5.5) Rivenditore	11
5.6) MostraAuto	12
5.7) Classi per Abstract Factory	14
5.7.1) AbstractFactory	14
5.7.2) DeluxeFactory e StandardFactory	14
5.7.3) LaFerrari, SF90Stradale e Testarossa	15
5.7.4) Classi rappresentanti i prodotti concreti	15
5.8) Classi per Strategy	16
5.8.1) PaymentStrategy	16
5.8.2) CreditCardStrategy	17
5.8.3) TransferStrategy	17
5.9) ControlloBudget	17
5.10) main	18
5.11) Output del programma	19

6)	Unit tests svolti	22
6.1)	<i>AcquirenteTest</i>	22
6.2)	<i>ControlloBudgetTest</i>	23
6.3)	<i>RivenditoreTest</i>	24
6.4)	<i>LaFerrariDeluxeTest, SF90StradaleTest e TestarossaDeluxeTest</i>	25
7)	Fonti	26

1) INTRODUZIONE E PRESENTAZIONE DELL'ELABORATO

Per questo elaborato (modo #1) ho scelto di realizzare uno scenario nel quale troviamo un acquirente interessato all'acquisto di una Ferrari; questo verrà convinto o meno da uno spot pubblicitario realizzato dal rivenditore che andrà ad aggiornare il suo indice di gradimento (percentuale tra 0 e 100). Se questo è superiore al 50%, l'acquirente può scegliere il metodo di pagamento che preferisce (tra carta di credito e bonifico) e, successivamente, il modello di Ferrari (LaFerrari, SF90 Stradale o Testarossa) e la versione (Standard o Deluxe). Se il modello scelto è disponibile, viene notificato al rivenditore di inviare l'ordine alla fabbrica, non prima però di aver controllato che il budget dell'acquirente sia sufficiente e che il pagamento sia andato a buon fine; a questo punto l'acquirente calcola quanto dovrà spendere in totale tra assicurazione e super bollo all'anno e, se questa spesa dovesse superare il 40% del budget, deciderà di annullare l'ordine. Se questo non accade, il rivenditore aggiorna il numero di auto vendute e l'acquirente userà l'auto per un certo numero di giorni e di km (random); ad un certo punto il rivenditore controllerà se è il momento di effettuare la revisione (in base proprio al numero di giorni passati dall'acquisto e ai km percorsi) e, in caso affermativo, la effettua azzerando entrambi i parametri. Inoltre l'acquirente potrà vedere a schermo un'immagine raffigurante l'auto scelta ogni volta che avrà deciso quale acquistare. Tutto ciò viene eseguito 3 volte.

1) EXCURSUS TEORICO SUI DESIGN PATTERNS USATI E CONTESTO D'USO

Nella realizzazione di questo elaborato, come già accennato in copertina, ho utilizzato i patterns Observer, Abstract Factory, Strategy e Singleton; di seguito fornisco una breve introduzione teorica per ognuno di essi ed il contesto nel quale sono stati usati.

2.1) Observer

Il pattern Observer (o Publish-Subscribe) permette di creare una dipendenza tra oggetti in modo tale che, se un oggetto cambia il suo stato interno, ciascuno degli oggetti dipendenti da esso viene notificato e aggiornato automaticamente. Esso trova applicazione quando siamo nel caso nel quale uno o più oggetti (**Observer**) devono conoscere lo stato di un oggetto (**Subject**): dunque il “**subject**” è l’oggetto osservato e gli “**observers**” sono gli oggetti che osservano. Di seguito vediamo un diagramma UML che esemplifica meglio i componenti in gioco:

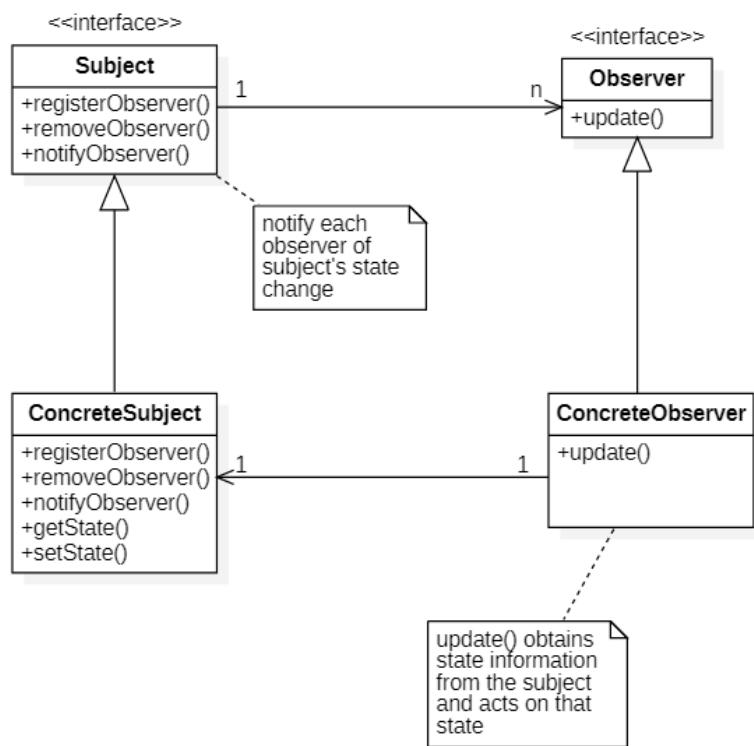


Figura 1: Diagramma UML di classe del pattern Observer pull

- **Subject**: classe Observable. Ha conoscenza dei propri observer, che possono essere illimitati, fornisce operazioni per l’aggiunta e la cancellazione di Observer e fornisce un metodo per la notifica agli Observer.
- **Observer**: interfaccia Observer. Specifica un’interfaccia per la notifica di eventi agli oggetti interessati in un Subject.

- **ConcreteSubject:** classe ObservedSubject. Essa mantiene lo stato dell'oggetto osservato e notifica gli observer in caso di cambio di stato; inoltre invoca le operazioni di notifica ereditate dal Subject, quando i ConcreteObserver devono essere informati.
- **ConcreteObserver:** mantiene un riferimento ad un oggetto di tipo ConcreteSubject e implementa l'interfaccia dell'Observer definendo il comportamento in caso di cambio di stato del soggetto osservato.

Nel mio progetto, ho usato tale pattern nell'ambito della comunicazione tra acquirente e rivenditore e tra rivenditore e la funzionalità di mostrare a schermo l'immagine dell'auto (Observer "pull", con il metodo update() che non ha bisogno di parametri): in particolare l'acquirente ricopre il ruolo di Concrete Subject, dunque si occupa di notificare il rivenditore (che quindi ricopre il ruolo di Concrete Observer) quando esso ha scelto l'auto da acquistare, e il rivenditore, di conseguenza, in base alla scelta dell'acquirente creerà l'ordine specifico. Il rivenditore ricopre anche il ruolo di Concrete Subject, in quanto deve notificare alla classe MostraAuto (Concrete Observer) che l'acquirente ha scelto la macchina, in modo che questa mostri a schermo l'auto corrispondente al momento giusto.

I vantaggi nell'uso di questo pattern consistono nell'accoppiamento "meno stretto possibile" tra oggetti, la facilità di inviare dati ad altri oggetti in modo efficace senza modificare il Subject o l'Observer, l'indipendenza tra Subject e Observer (quindi una modifica di uno non implica dover modificare l'altro) e infine anche il fatto che è possibile aggiungere/rimuovere gli observers in qualsiasi momento.

2.2) Abstract Factory

L'intento di questo pattern è quello di fornire un'interfaccia per creare famiglie di prodotti tra loro collegati o dipendenti, dunque si crea un'interfaccia con metodi astratti che le classi derivate vanno a specializzare per creare le versioni specializzate delle classi collegate: in pratica abbiamo una fabbrica che rende diverse fabbriche ognuna delle quali produce diversi prodotti. Si tratta di un pattern creazionale, dal momento che rende più semplice creare oggetti complessi per i quali l'uso del solo costruttore non è sufficiente. Di seguito viene riportato il class diagram di tale pattern:

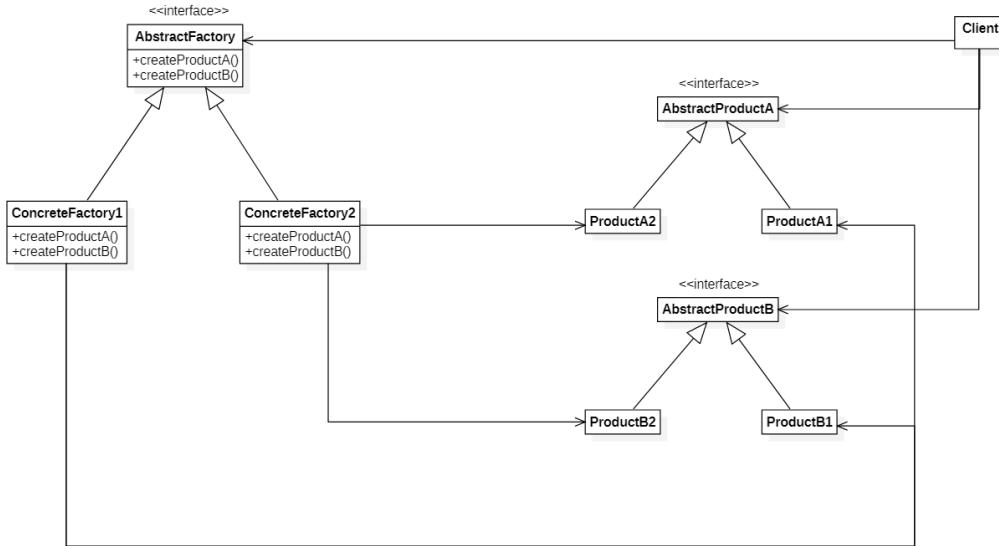


Figura 2: Diagramma UML di classe del pattern Abstract Factory

AbstractFactory definisce l’interfaccia per la creazione dei vari prodotti tra loro collegati, mentre le **ConcreteFactory1** e **ConcreteFactory2** hanno la conoscenza su quali tipi sono tra loro collegati e devono essere istanziati assieme. Per ogni tipologia di prodotto si ha una classe che ne definisce l’interfaccia (**AbstractProductA** e **AbstractProductB**) e per ogni tipo di prodotto se ne hanno le varianti (**ProductA1** e **ProductA2**, **ProductB1** e **ProductB2**).

Nel mio progetto, ho usato tale pattern nell’ambito della creazione dei diversi tipi di Ferrari, quindi troviamo l’interfaccia **AbstractFactory** che dichiara i metodi che verranno specializzati nelle varie factory concrete (**DeluxeFactory** e **StandardFactory**). Gli **AbstractProduct** sono rappresentati dalle interfacce **LaFerrari**, **SF90Stradale** e **Testarossa**, le quali dichiarano il metodo **create()** che verrà definito e specializzato per ogni prodotto (**LaFerrariStandard** e **LaFerrariDeluxe**, **SF90StradaleStandard** e **SF90StradaleDeluxe**, **TestarossaStandard** e **TestarossaDeluxe**).

2.3) Strategy

Lo Strategy pattern è uno dei pattern fondamentali originariamente definiti dalla Gang of Four. Il suo scopo è quello di isolare un algoritmo all’interno di un oggetto, in modo da poter modificare dinamicamente gli algoritmi utilizzati da un’applicazione. Gli algoritmi devono essere intercambiabili tra loro, cioè devono esporre sempre la stessa interfaccia, cosicché il client non debba fare nessuna assunzione su quale particolare strategia viene istanziata. Di seguito ne riporto il class diagram:

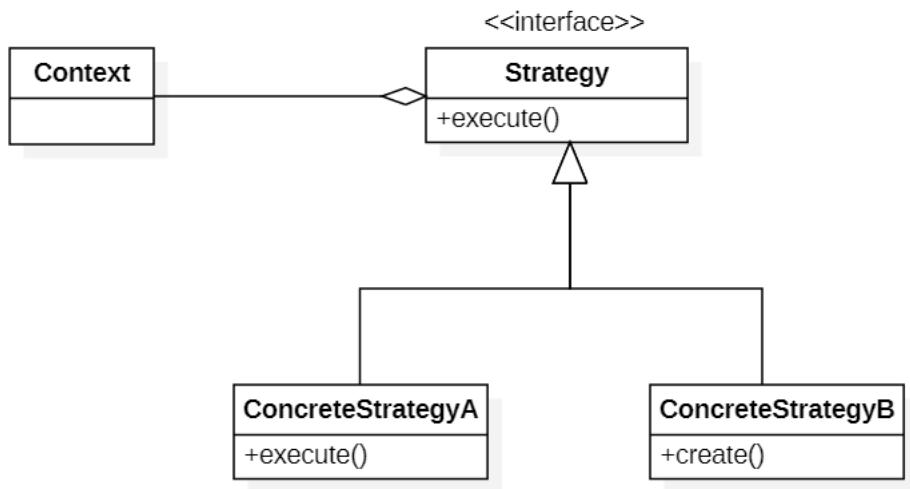


Figura 3: Diagramma UML di classe del pattern Strategy

L'interfaccia **Strategy** è comune a tutte le strategy concrete: essa dichiara un metodo che viene usato dal contesto per eseguire una strategia. Il **Context** mantiene un riferimento ad una delle strategy concrete e comunica con essa attraverso l'interfaccia strategy; le **ConcreteStrategyA** e **ConcreteStrategyB** implementano in modo differente un algoritmo che viene usato dal contesto. Nel mio progetto, ho ricorso a tale pattern nell'ambito di quale tipo di pagamento l'acquirente preferisce usare: abbiamo quindi l'interfaccia **PaymentStrategy** che dichiara il metodo *pay(float amount)*, il quale verrà implementato dalle due strategy concrete **CreditCardStrategy** e **TransferStrategy**.

2.4) Singleton

La GoF afferma che “ *Il Singleton pattern ha lo scopo di assicurarsi che una classe abbia solo un’istanza e fornire un punto di accesso globale ad essa* ” ; gli elementi che caratterizzano questo pattern sono il **costruttore privato** (per evitare la creazione di oggetti da classi esterne) e un **metodo statico** (per accedere all'unica istanza dell'oggetto). Nel mio elaborato ho creato due singleton, l'**Acquirente** e il **Rivenditore**, per fare in modo che la comunicazione avvenisse tra un solo acquirente e un solo rivenditore.

3) USE CASE DIAGRAM

Di seguito riporto il diagramma dei casi d'uso relativo al mio progetto:

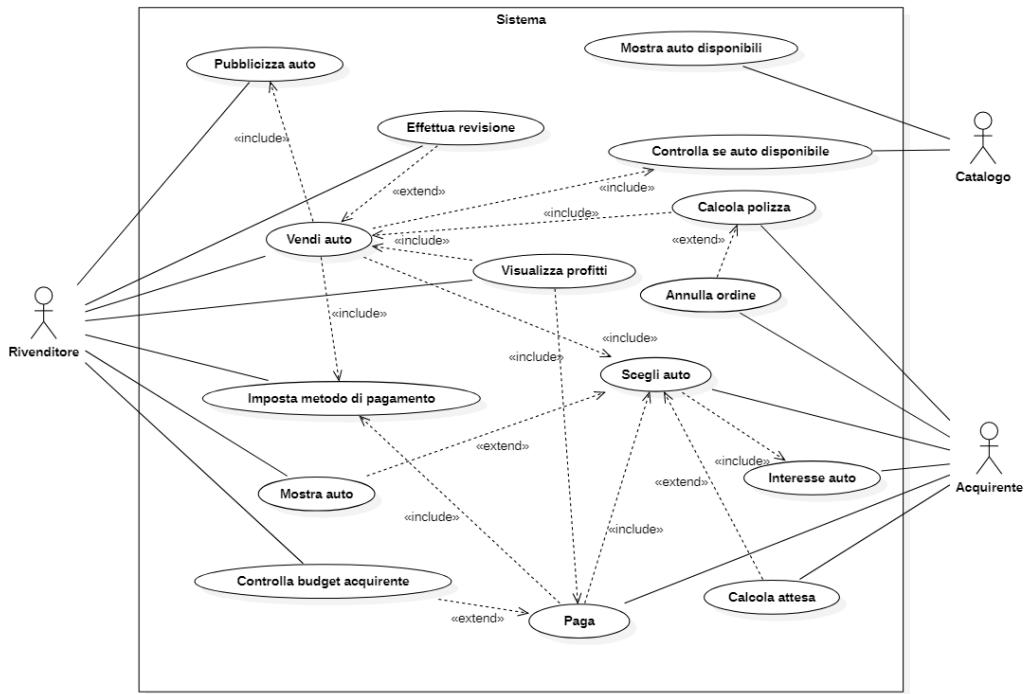


Figura 4: Use Case Diagram dell'elaborato

Abbiamo quindi 3 attori: Il **rividitore**, l'**acquirente** e il **catalogo**.

4) PROGETTAZIONE

In questa sezione vengono riportati il Class Diagram e due Sequence Diagrams relativi a due particolari casi d'uso.

4.1) Class Diagram

Riporto anche il link alla [repository GitHub](#) del progetto, che contiene anche i diagrammi UML (utile per il class diagram, che è troppo grande per una pagina di documento).

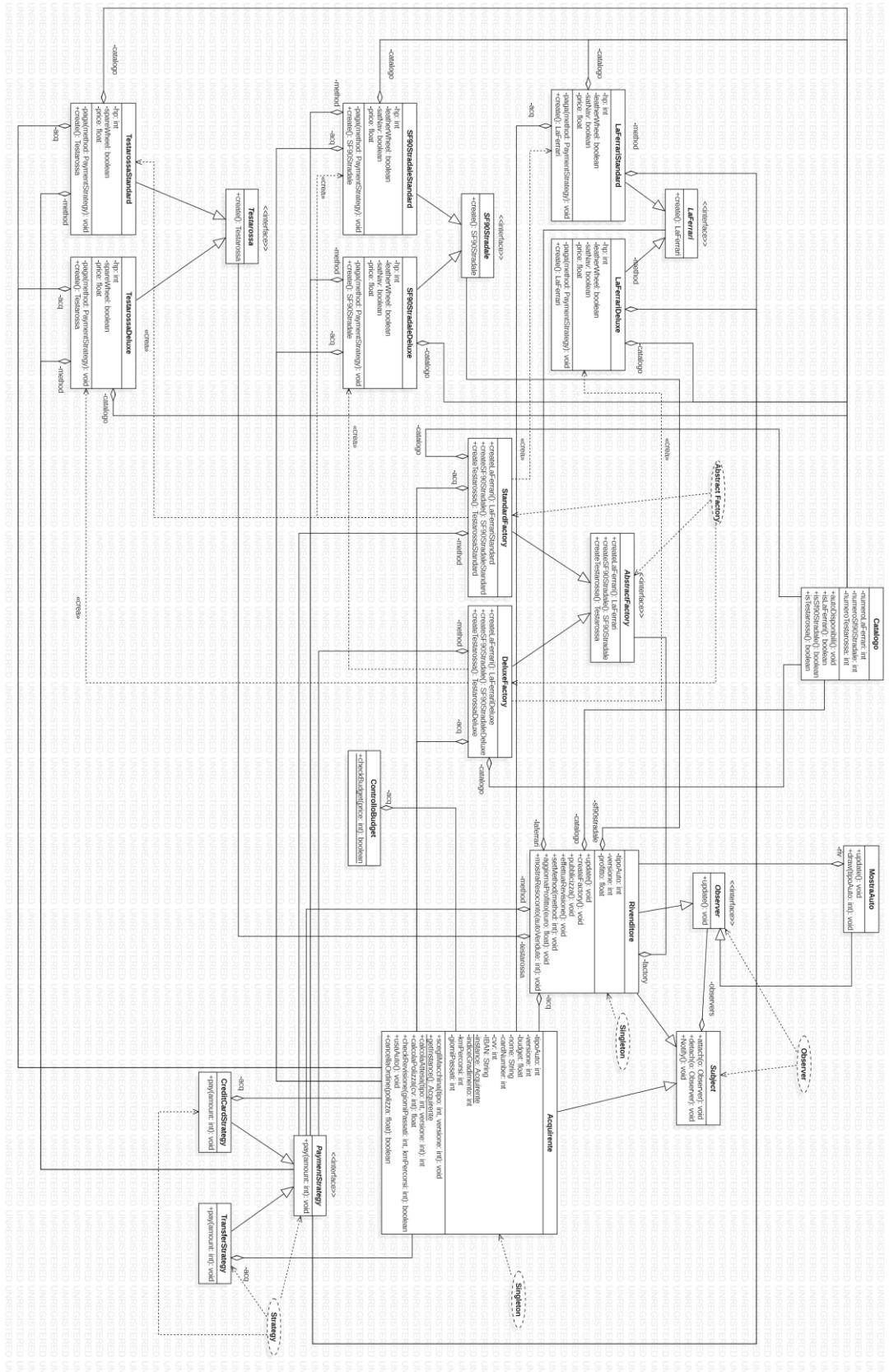


Figura 5: Class Diagram dell'elaborato

4.2) Sequence Diagram

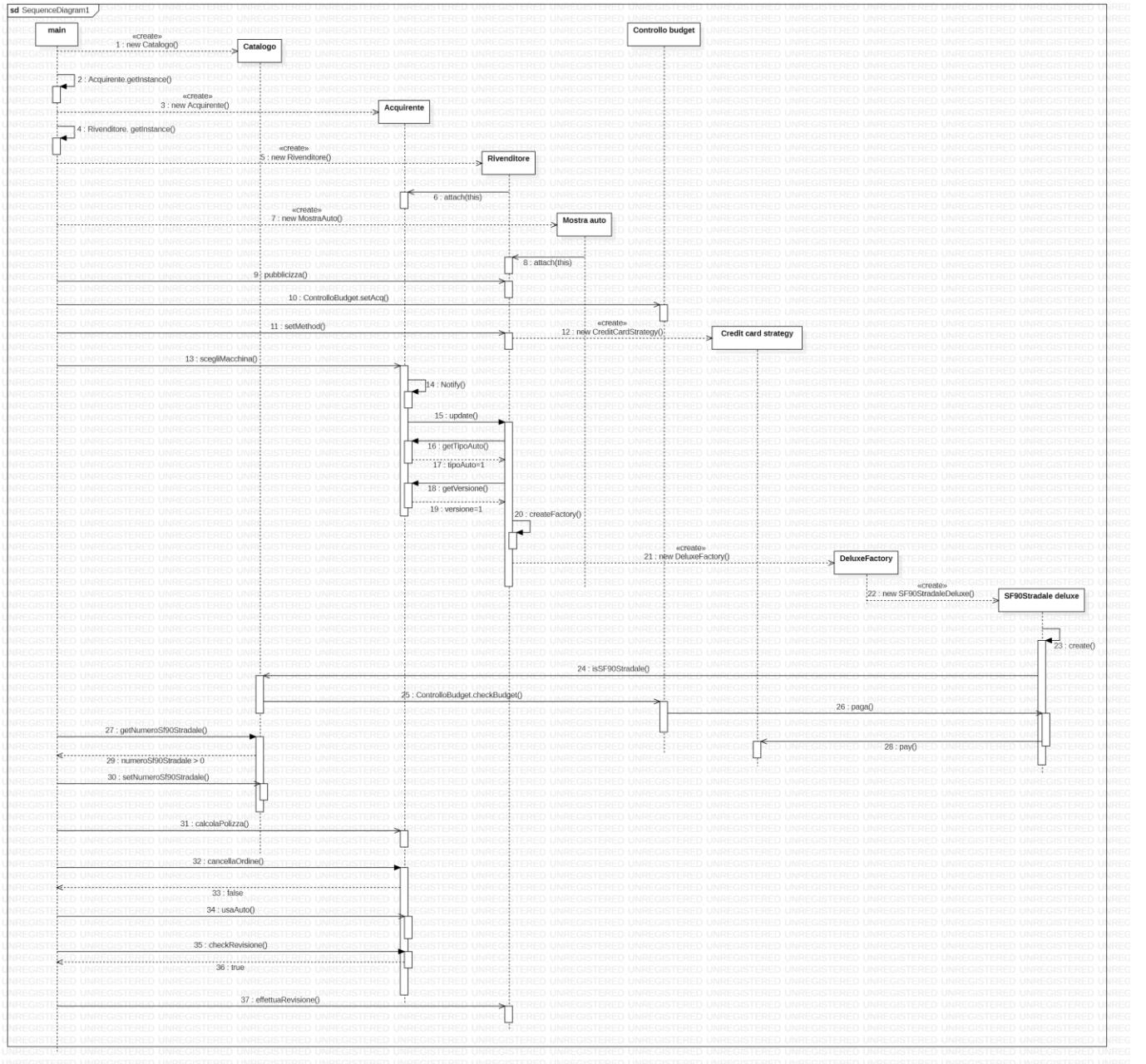


Figura 6: Sequence Diagram relativo ad un acquirente che sceglie di acquistare una SF90 Stradale Deluxe, presente nel catalogo del rivenditore; l'acquirente ha un budget sufficiente, la polizza rientra nei criteri stabiliti, quindi usa l'auto per un tot di giorni e km, dopo i quali la porterà ad effettuare la revisione dal rivenditore stesso.

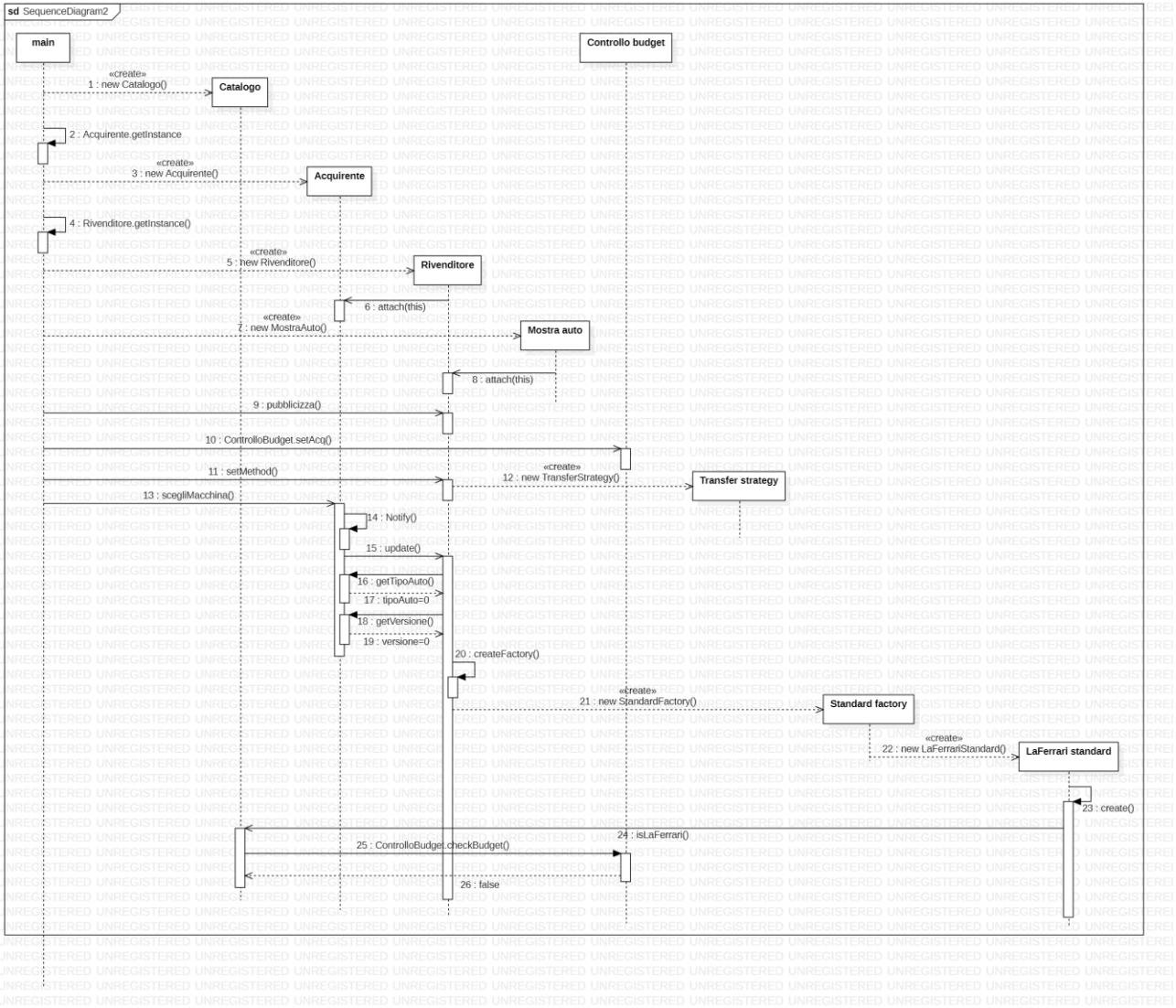


Figura 7: Sequence Diagram relativo ad un acquirente che è interessato all'acquisto di una LaFerrari Standard, ma non ha budget a sufficienza per completare l'acquisto.

5) IMPLEMENTAZIONE

Tutte le classi, tranne la classe astratta **Subject** e le varie interfacce, sono state etichettate come **final** al fine di evitare il problema della classe base fragile.

5.1) Catalogo

Questa classe imposta (nel costruttore) il numero di auto disponibili per ogni tipo e fornisce metodi per mostrare il numero di auto disponibili e per controllare se un particolare tipo è disponibile (*numeroLaFerrari*, *numeroSF90Stradale*, *numeroTestarossa* sono 3 interi inizializzati a 0)

```

public Catalogo(){
    numeroLaFerrari = (int)(Math.random() * 6);
    numeroSf90Stradale = (int)(Math.random() * 6);
    numeroTestarossa = (int)(Math.random() * 6);
    this.autoDisponibili();
}

4 usages  ↳ andreareset88
public void autoDisponibili(){
    System.out.println("Attualmente sono disponibili: ");
    System.out.println(this.getNumeroLaFerrari()+" LaFerrari");
    System.out.println(this.getNumeroSf90Stradale()+" SF90Stradale");
    System.out.println(this.getNumeroTestarossa()+" Testarossa");
}

}

public boolean isLaFerrari(){
    if(this.getNumeroLaFerrari() > 0){
        System.out.println("LaFerrari disponibile all'acquisto!");
        return true;
    } else {
        System.out.println("LaFerrari non è attualmente disponibile...");
        return false;
    }
}

```

5.2) Observer

Interfaccia che dichiara il metodo *update()*.

5.3) Subject

Classe astratta che dichiara e definisce i metodi *attach()*, *detach()*, *Notify()*.

5.4) Acquirente

```

private Acquirente(int tipoAuto, int versione, float budget, String nome, int cardNumber, int cvv, String IBAN) {
    this.tipoAuto = tipoAuto;
    this.versione = versione;
    this.budget = budget;
    this.nome = nome;
    this.cardNumber = cardNumber;
    this.cvv = cvv;
    this.IBAN = IBAN;
}

// SINGLETON
7 usages  ↳ andreareset88
public static Acquirente getInstance(int tipoAuto, int versione, float budget, String nome, int cardNumber, int cvv, String IBAN){
    if(instance == null)
        instance = new Acquirente(tipoAuto, versione, budget, nome, cardNumber, cvv, IBAN);
    return instance;
}

3 usages  ↳ andreareset88
public void scegliMacchina(int tipo, int versione) throws InterruptedException, IOException {
    tipoAuto = tipo;
    this.versione = versione;
    Notify();
}

```

Voglio una sola istanza dell'acquirente, quindi uso il pattern Singleton al momento della creazione dell'oggetto. Quando questo invoca *scegliMacchina(tipo, versione)*, si innesca il meccanismo del pattern Observer che andrà ad invocare il *Notify()* del Subject e quindi l'*update()* dell'Observer.

Il metodo *calcolaAttesa(tipo, versione)* si occupa di restituire quanti giorni l'acquirente dovrà aspettare prima che la macchina da lui acquistata gli venga consegnata: imposto 3 valori di base (uno per ogni tipo di auto), dunque la funzione restituirà tali valori nel caso di versione standard, mentre aggiungerà 16,

Il metodo *calcolaPolizza(cv)* si occupa di calcolare, in base alla potenza del motore della macchina scelta, il totale annuo e mensile da pagare tra assicurazione e super bollo.

```

public float calcolaPolizza(int cv){
    float assicurazione = 0;
    float superBollo = 0;
    if(cv >= 0 && cv < 700){
        assicurazione = 8000;
        superBollo = 9000;
    } if(cv >= 700 && cv < 1000){
        assicurazione = 10000;
        superBollo = 11000;
    } else if(cv >= 1000){
        assicurazione = 11500;
        superBollo = 12000;
    }
    float totaleAnnuo = assicurazione + superBollo;
    System.out.println("Costo annuo: assicurazione = "+assicurazione+" €, super bollo = "+superBollo+" €, per un totale di "+totaleAnnuo+" €");
    System.out.println("Costo mensile: assicurazione = "+assicurazione/12+" €, super bollo = "+superBollo/12+" €, per un totale di "+totaleAnnuo/12+" €/mese");
    return totaleAnnuo;
}

```

Continuando, *checkRevisione(giorniPassati, kmPercorsi)* dice se è o meno il momento di portare l'auto ad effettuare la revisione, *usaAuto()* imposta su base casuale quanti giorni sono passati dalla consegna e quanti km sono stati percorsi e *cancellaOrdine(polizza)* calcola qual è la percentuale della polizza rispetto al budget dell'acquirente e, se superiore al 40%, comanda di annullare l'ordine.

```
public boolean checkRevisione(int giorniPassati, int kmPercorsi) throws InterruptedException {
    if(giorniPassati >= 1095 || kmPercorsi >= 20000){
        System.out.println("E' necessario portare la vettura in officina per la revisione");
        return true;
    } else {
        System.out.println("Non è ancora il momento per la revisione");
        return false;
    }
}

3 usages  ± andreareset88
public void usaAuto(){
    int giorni = (int)(Math.random() * 2001);
    this.setGiorniPassati(giorni);
    int km = (int)(Math.random() * 40001);
    this.setKmPercorsi(km);
}

// Se il costo della polizza supera il 40% del budget, annulla l'ordine
5 usages  ± andreareset88
public boolean cancellaOrdine(float polizza){
    float perc = (polizza * 100) / (this.getBudget());
    System.out.println("La polizza annua rappresenta il "+perc+" % del budget");
    return perc >= 40;
}
```

5.5) Rivenditore

La classe Rivenditore ha una duplice funzione: infatti essa funge sia da Observer in attesa della scelta da parte dell'acquirente sia da Subject che comunica alla classe MostraAuto quando mostrare l'immagine dell'auto scelta. Anche in questo caso voglio una sola istanza della classe, quindi uso il Singleton al momento della creazione dell'oggetto.

Nel costruttore, il rivenditore si inserisce nella lista degli observers interessati allo stato dell'acquirente. Il metodo *update()*, dopo aver ottenuto il tipo e la versione dell'auto scelta dall'acquirente, invoca *createFactory()* e anche *Notify()*, che andrà a notificare all'osservatore che è il momento di mostrare a schermo l'immagine dell'auto.

```
private Rivenditore(Acquirente acq, PaymentStrategy method, Catalogo catalogo) {
    this.acq = acq;
    this.acq.attach( this );
    this.method = method;
    this.catalogo = catalogo;
}

// SINGLETON
5 usages  ± andreareset88
public static Rivenditore getInstance(Acquirente acq, PaymentStrategy method, Catalogo catalogo) {
    if(instance == null)
        instance = new Rivenditore(acq, method, catalogo);
    return instance;
}

1 usage  ± andreareset88
@Override
public void update() throws InterruptedException, IOException {
    this.tipoAuto = acq.getTipoAuto();
    this.versione = acq.getVersione();
    createFactory();
    Notify();
}
```

Il metodo *createFactory()* viene usato nel contesto del pattern Abstract Factory, ed è il responsabile dell'istanziazione della factory concreta, prima, e del prodotto concreto, poi (sulla base del tipo e della versione di auto scelta). Per capire se l'acquirente è o meno interessato all'acquisto di una vettura uso l'indice di gradimento (un intero che rappresenta una percentuale tra 0 e 100) : in particolare esso indica un interesse a procedere se superiore al 50%. Questo parametro viene modificato dal metodo *pubblicizza()*, che simula la messa in onda di uno spot pubblicitario (con un semplice messaggio a schermo) che andrà ad aggiornare l'indice di gradimento dell'acquirente su base casuale:

```

private void createFactory() throws InterruptedException {
    if(versione == 0)
        factory = new StandardFactory(acq, method, catalogo);
    else if(versione == 1)
        factory = new DeluxeFactory(acq, method, catalogo);
    if(tipoAuto == 0) {
        laferrari = factory.createLaFerrari();
        laferrari.create();
    }
    if(tipoAuto == 1) {
        sf90stradale = factory.createSF90Stradale();
        sf90stradale.create();
    }
    if(tipoAuto == 2) {
        testarossa = factory.createTestarossa();
        testarossa.create();
    }
}

5 usages  andreamer88
public int pubblicizza() throws InterruptedException {
    System.out.println("SPOT PUBBLICITARIO IN TRASMISSIONE...");
    int gradimento = (int)(Math.random() * 101);
    acq.updateIndiceGradimento(gradimento);
    Thread.sleep( millis: 5000 );
    System.out.println("SPOT TERMINATO");
    System.out.println(acq.getNome()+" ha un indice di gradimento del "+gradimento+" %");
    return gradimento;
}

```

Troviamo poi un metodo, *effettuaRevisione()*, che viene invocato o meno sulla base dei giorni passati dalla consegna dell'auto e dei km percorsi, il quale, semplicemente, azzera entrambi i parametri citati; *aggiornaProfitto(euro)* viene usato dal rivenditore per tenere conto dei profitti in seguito ad ogni auto venduta. Infine *mostraResoconto(autoVendute)* mostra un breve riepilogo dell'affare con l'acquirente, ossia quante auto sono state vendute ed il guadagno totale.

5.6) MostraAuto

Questa è la classe responsabile di mostrare, dopo che l'acquirente ha scelto quale auto acquistare e il rivenditore ha effettuato l'invio dell'ordine alla fabbrica, l'immagine a schermo della macchina. Per implementare tale funzionalità ho preso spunto da quanto esemplificato su <https://www.delftstack.com/howto/java/display-an-image-in-java/>. Vogliamo quindi usare la classe **JLabel** della libreria **Swing**; **JLabel** estende **JComponent**, e possiamo allegare questo componente ad un **JFrame**. Per leggere il file immagine, usiamo la classe **File** e passiamo il percorso dell'immagine, poi convertiamo questa in un oggetto **BufferedImage** usando *ImageIO.read()*; ora creiamo un'icona che deve essere mostrata nel **JLabel**. Per fare ciò, abbiamo bisogno di un oggetto **JFrame** con un **FlowLayout** e una certa dimensione; adesso creiamo un oggetto **JLabel** e ne impostiamo la sua icona usando *JLabel.setIcon()*, dopodichè aggiungiamo il componente **JLabel** al **JFrame** e impostiamo la visibilità del frame a true.

Questa classe riveste il ruolo di Observer dello stato del rivenditore. Per fare in modo che quest'ultimo notifichi quando è il momento giusto per mostrare l'immagine dell'auto, invocherà il metodo *Notify()* come ultima istruzione del metodo *update()*: in questo modo siamo certi che avviene la corretta sequenza degli eventi (**acquirente** sceglie auto → **rivenditore** istanzia la factory corretta e invia l'ordine alla fabbrica → la classe **MostraAuto** mostra a schermo l'auto).

```

@Override
public void update() throws IOException, InterruptedException {
    System.out.println("L'auto scelta verrà mostrata in una nuova scheda");
    Thread.sleep( millis: 3000 );
    this.draw(riv.getAutoScelta());
}

1 usage  ↗ andreareset88
public void draw(int tipoAuto) throws IOException {
    File laferrari = new File( pathname: "C:\\\\Users\\\\acer\\\\OneDrive\\\\Immagini\\\\La-ferrari.jpg");
    File sf90 = new File( pathname: "C:\\\\Users\\\\acer\\\\OneDrive\\\\Immagini\\\\sf90-stradale.jpg");
    File testarossa = new File( pathname: "C:\\\\Users\\\\acer\\\\OneDrive\\\\Immagini\\\\testarossa.jpg");
    BufferedImage image;
    if(tipoAuto == 0)
        image = ImageIO.read(laferrari);
    else if(tipoAuto == 1)
        image = ImageIO.read(sf90);
    else image = ImageIO.read(testarossa);
    ImageIcon ImageIcon = new ImageIcon(image);
    JFrame jFrame = new JFrame();
    jFrame.setLayout(new FlowLayout());
    jFrame.setSize( width: 824, height: 494 );
    JLabel jLabel = new JLabel();
    jLabel.setIcon(ImageIcon);
    jFrame.add(jLabel);
    jFrame.setVisible(true);
    jFrame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
}

```

5.7) Classi per Abstract Factory

5.7.1) Abstract Factory

Quest'interfaccia dichiara i 3 metodi che dovranno essere implementati dalle factory concrete **StandardFactory** e **DeluxeFactory** per la creazione dei prodotti concreti.

5.7.2) DeluxeFactory e StandardFactory

Queste classi svolgono il ruolo di factory concrete che implementano i 3 metodi *createLaFerrari()*, *createSf90Stradale()* e *createTestarossa()* definiti in **AbstractFactory**; l'unica differenza tra le due classi risiede nei parametri passati al costruttore dei prodotti concreti creati (il prezzo sarà più alto nella versione deluxe, così come saranno presenti o meno gli optional, e cambia la potenza del motore).

Qui sotto riporto quindi solo il codice della classe **DeluxeFactory**, essendo appunto le differenze minime

```
1 usage  ± andreareset88
public DeluxeFactory(Acquirente acq, PaymentStrategy method, Catalogo catalogo) {
    this.acq = acq;
    this.method = method;
    this.catalogo = catalogo;
}

1 usage  ± andreareset88
@Override
public LaFerrari createLaFerrari() {
    return new LaFerrariDeluxe( hp: 963, satNav: true, leatherWheel: true, price: 3000000, acq, method, catalogo);
}

1 usage  ± andreareset88
@Override
public SF90Stradale createSF90Stradale() {
    return new SF90StradaleDeluxe( hp: 1050, satNav: true, leatherWheel: true, price: 5000000, acq, method, catalogo);
}

1 usage  ± andreareset88
@Override
public Testarossa createTestarossa() { return new TestarossaDeluxe( hp: 470, spareWheel: true, price: 800000, acq, method, catalogo); }
```

5.7.3) LaFerrari, SF90Stradale e Testarossa

Si tratta di 3 interfacce (Abstract products) che dichiarano il metodo *create()*; questo verrà implementato dai prodotti concreti per effettuare correttamente la procedura di acquisto della rispettiva auto.

5.7.4) Classi rappresentanti i prodotti concreti

Trattasi di 6 classi: **LaFerrariStandard** e **LaFerrariDeluxe**, **SF90StradaleStandard** e **SF90StradaleDeluxe**, **TestarossaStandard** e **TestarossaDeluxe**; queste svolgono il ruolo di prodotti concreti e tutte implementano la rispettiva interfaccia tra le 3 citate sopra.

Gli attributi caratteristici di ogni prodotto concreto, ossia la potenza del motore, se presenti o meno gli optional e il prezzo, sono stati marcati come **final** e sono stati sprovvisti di metodi **setter** per evitare che possano accidentalmente venire modificati.

Tutte comprendono il metodo *paga(method)*, il quale verrà invocato da *create()* per effettuare il pagamento; esso riceve un oggetto di tipo **PaymentStrategy** per indicare quale metodo di pagamento è stato scelto dall'acquirente.

```

public LaFerrariDeluxe(int hp, boolean satNav, boolean leatherWheel, float price, Acquirente acq, PaymentStrategy method, Catalogo catalogo) {
    this.hp = hp;
    this.satNav = satNav;
    this.leatherWheel = leatherWheel;
    this.price = price;
    this.acq = acq;
    this.method = method;
    this.catalogo = catalogo;
}

1 usage  ± andreamer88
public void paga(PaymentStrategy paymentMethod) throws InterruptedException {
    float amount = this.getPrice();
    paymentMethod.pay(amount);
}

```

Il “cuore” di queste classi è l’implementazione del metodo *create()*, che controlla se l’auto scelta è disponibile ed effettua le operazioni necessarie per effettuare il pagamento per poi creare l’ordine ed inviarlo alla fabbrica.

Sotto riporto solo l’implementazione fornita in **LaFerrariDeluxe**, essendo identica per tutti i prodotti concreti.

```

@Override
public LaFerrari create() throws InterruptedException {
    if(catalogo.isLaFerrari()) {
        if (ControlloBudget.checkBudget(this.getPrice())) {
            paga(method);
            System.out.println("E' in corso l'invio dell'ordine alla fabbrica per la LaFerrari...");
            try {
                Thread.sleep( 5000 );
            } catch (InterruptedException e) {
                e.printStackTrace();
            }
            System.out.println("INVIO EFFETTUATO!");
            System.out.println("E' stata scelta la versione DELUXE, che include un motore ibrido da " + hp + " cv");
            if (satNav)
                System.out.print(" , il sistema di navigazione");
            if (leatherWheel)
                System.out.println(" , il volante in pelle");
            int giorni = acq.calcolaAttesa(acq.getTipoAuto(), acq.getVersione());
            System.out.println(acq.getNome() + " , l'auto Le verrà consegnata tra " + giorni + " giorni");
            return this;
        } else {
            System.out.println("Spiacenti, l'invio non è andato a buon fine");
            return null;
        }
    } else return null;
}

```

5.8) Classi per Strategy

Adesso vengono mostrate le classi che permettono di differenziare il metodo di pagamento in base a quale viene scelto dall’acquirente; per fare ciò uso il pattern Strategy.

5.8.1) PaymentStrategy

Trattasi dell’interfaccia Strategy: essa dichiara il metodo *void pay(amount)* che verrà specializzato nei 2 Concrete Strategies

5.8.2) CreditCardStrategy

Questa classe, avente il ruolo di Concrete Strategy, implementa l'interfaccia **PaymentStrategy** e vuole associati l'acquirente e il rivenditore. Nell'implementazione del metodo *void pay(amount)*, viene permesso all'acquirente di inserire il codice di sicurezza cvv della carta di credito fino ad un massimo di 3 tentativi, dopo i quali l'operazione viene annullata e il sistema viene chiuso; se il cvv inserito è corretto, procede con il pagamento diminuendo il budget dell'acquirente e aggiornando i profitti del rivenditore.

```
@Override
public void pay(float amount) throws InterruptedException {
    int tentativiRimanenti = 3;
    boolean correct = false;
    while(!correct) {
        System.out.println("Inserire cvv per la verifica (" + tentativiRimanenti + " tentativi rimanenti)");
        Scanner scanner = new Scanner(System.in);
        int code = scanner.nextInt();
        if (acq.getCVV() == code) {
            correct = true;
        }
        System.out.println("Carta " + acq.getCardNumber() + " accettata, tentativo di pagamento in corso...");
        Thread.sleep( millis: 3000 );
        float budget = acq.getBudget();
        budget -= amount;
        acq.setBudget(budget);
        riv.aggiornaProfitto(amount);
        System.out.println("Pagamento di " + amount + " €, effettuato dal Sig. " + acq.getName() + " riuscito, il budget rimanente è di " + budget + " €");
    } else {
        System.out.println("Il codice cvv non è corretto");
        tentativiRimanenti--;
        if(tentativiRimanenti == 0){
            System.out.println("Attenzione, tentativi esauriti!");
            System.exit( status: 0 );
        }
    }
}
```

5.8.3) TransferStrategy

Seconda classe avente il ruolo di Concrete Strategy; anch'essa dunque implementa l'interfaccia **PaymentStrategy** e vuole l'acquirente e il rivenditore. L'implementazione di *void pay(amount)* è più semplice rispetto a quella fornita per il pagamento tramite carta di credito: infatti, se l'acquirente decide di pagare tramite bonifico, dato che il budget è già stato controllato ed è sufficiente, l'unica cosa che viene fatta è diminuire quest'ultimo e aumentare i profitti del rivenditore.

```
@Override
public void pay(float amount) throws InterruptedException {
    System.out.println("Tentativo di pagamento tramite bonifico in corso...");
    float budget = acq.getBudget();
    budget -= amount;
    Thread.sleep( millis: 3000 );
    acq.setBudget(budget);
    riv.aggiornaProfitto(amount);
    System.out.println(amount + " pagato con bonifico effettuato da " + acq.getName() + " all'IBAN " + acq.getIBAN() + ", rimangono " + budget + " €");
}
```

5.9) ControlloBudget

Questa classe vuole associato l'acquirente e contiene solo il metodo *boolean checkBudget(price)*, che verrà invocato dal metodo *create()* del prodotto concreto che è stato istanziato; tale metodo è stato dichiarato **static** per poter essere invocato senza avere alcuna istanza della classe.

```

public static boolean checkBudget(float price){
    float budget = acq.getBudget();
    System.out.println("Controllo budget acquirente in corso...");
    try {
        Thread.sleep( millis: 2000 );
    } catch (InterruptedException e) {
        e.printStackTrace();
    }
    if(budget >= price){
        System.out.println(acq.getNome()+" può procedere all'acquisto!");
        return true;
    } else {
        System.out.println("ERRORE, budget troppo basso, "+acq.getNome()+" non può procedere all'acquisto..");
        return false;
    }
}

```

5.10) main

Nel **main** del programma, dopo aver inizializzato il **Catalogo**, l'**Acquirente**, il **Rivenditore** e il gestore responsabile di mostrare l'immagine dell'auto a schermo e impostati alcuni parametri (compresa la chiamata *gestore.setRiv()*, che comanda al gestore osservatore di allegarsi al soggetto rivenditore), comincia la vera e propria esecuzione: il rivenditore pubblicizza le auto e, se l'indice di gradimento dell'acquirente è sufficientemente alto e l'auto scelta è disponibile all'acquisto, viene fatto scegliere a quest'ultimo il metodo di pagamento che preferisce (sfruttando il pattern Strategy come già discusso). A questo punto viene eseguito il metodo *acq.scegliMacchina(acq.getTipoAuto, acq.getVersione)* che darà il via alla collaborazione dei pattern Observer e Abstract Factory per la creazione e l'invio dell'ordine, nonché per mostrare (a operazioni terminate) l'immagine raffigurante l'auto a schermo. Ovviamente, se l'indice di gradimento dell'acquirente non dovesse superare la soglia prefissata, tutte le operazioni elencate non verranno eseguite, così come verrà annullato l'ordine se il costo della polizza annua (somma di assicurazione e super bollo) risultasse troppo elevato. Queste operazioni verranno eseguite 3 volte, per fare in modo di impostare tutti e 3 i tipi di auto. Alla fine l'osservatore acquirente si "stacca" dal soggetto acquirente, e lo stesso viene fatto per l'osservatore gestore che si disaccoppia dal soggetto rivenditore; come ultima istruzione viene mostrato il resoconto dell'affare tra rivenditore e acquirente.

Nel frammento di codice che riporto qua sotto vengono eseguite le operazioni per la creazione della SF90 Stradale deluxe

```

public static void main(String[] args) throws InterruptedException, IOException {
    Catalogo cat = new Catalogo();
    // Acquirente inizializzato con SF90 Stradale deluxe
    Acquirente acq = Acquirente.getInstance( tipoAuto: 1, versione: 1, budget: 7500000, nome: "Charles", cardNumber: 789543, cvc: 997, IBAN: "MN6546H");
    Rivenditore riv = Rivenditore.getInstance(acq, method: null, cat);
    MostraAuto gestore = new MostraAuto(riv);
    gestore.setRiv();
    int autoVendute = 0;
    ControlloBudget.setAcq(acq);
    riv.pubblicizza();
    if(acq.getIndiceGradimento() > 50) {
        // Qui entrano in gioco Abstract Factory e Observer
        if(cat.getNumeroSf90Stradale() > 0) {
            System.out.println("Scegliere metodo di pagamento (0 per carta di credito, 1 per bonifico)");
            Scanner scanner = new Scanner(System.in);
            int metodo = scanner.nextInt();
            riv.setMethod(metodo);
            acq.scegliMacchina(acq.getTipoAuto(), acq.getVersione());
            cat.setNumeroSf90Stradale(cat.getNumeroSf90Stradale() - 1);
            float polizza = acq.calcolaPolizza( cv: 1050);
            if (!acq.cancellaOrdine(polizza)) {
                autoVendute++;
                acq.usaAuto();
                Thread.sleep( millis: 5000);
                System.out.println("Sono passati " + acq.getGiorniPassati() + " giorni e sono stati percorsi " + acq.getKmPercorsi() + " km dalla data dell'acquisto");
                if (acq.checkRevisione(acq.getGiorniPassati(), acq.getKmPercorsi()))
                    riv.effettuaRevisione();
            } else {
                System.out.println("Polizza troppo alta, ordine annullato...");
                cat.setNumeroSf90Stradale(cat.getNumeroSf90Stradale() + 1);
            }
        }
    } else System.out.println(acq.getNome()+" non è interessato all'acquisto della vettura scelta.");
    cat.autoDisponibili();
    Thread.sleep( millis: 5000);
    riv.pubblicizza();
}

```

5.11) Output del programma

Riporto qui sotto come appare un output del mio programma:

Attualmente sono disponibili:
1 LaFerrari
5 SF90Stradale
5 Testarossa
SPOT PUBBLICITARIO IN TRASMISSIONE...
SPOT TERMINATO
Charles ha un indice di gradimento del 53 %
Scegliere metodo di pagamento (0 per carta di credito, 1 per bonifico)
0
SF90 Stradale disponibile all'acquisto!
Controllo budget acquirente in corso...
Charles può procedere all'acquisto!
Inserire cvv per la verifica (3 tentativi rimanenti)
888
Il codice cvv non è corretto
Inserire cvv per la verifica (2 tentativi rimanenti)
997
Carta 789543 accettata, tentativo di pagamento in corso...
Pagamento di 5000000.0 €, effettuato dal Sig. Charles riuscito, il budget rimanente è di 5000000.0 €
Invio dell'ordine alla fabbrica per la SF90 Stradale in corso...
INVIO COMPLETATO!
E' stata scelta la versione DELUXE, che include un motore ibrido da 1050 cv , il sistema di navigazione , il volante in pelle
Charles , l'auto Le verrà consegnata tra 95 giorni
L'auto scelta verrà mostrata in una nuova scheda
Numero di SF90 Stradale impostato a 4
Costo annuo: assicurazione = 11500.0 €, super bollo = 12000.0 €, per un totale di 23500.0 €
Costo mensile: assicurazione = 958.3333 €, super bollo = 1000.0 €, per un totale di 1958.3334 €/mese
La polizza annua rappresenta il 0.47 % del budget
Sono passati 1294 giorni e sono stati percorsi 9803 km dalla data dell'acquisto
E' necessario portare la vettura in officina per la revisione
Revisione in corso...
Revisione completata!
Attualmente sono disponibili:
1 LaFerrari
4 SF90Stradale
5 Testarossa
SPOT PUBBLICITARIO IN TRASMISSIONE...
SPOT TERMINATO
Charles ha un indice di gradimento del 93 %
Scegliere metodo di pagamento (0 per carta di credito, 1 per bonifico)
1
LaFerrari disponibile all'acquisto!
Controllo budget acquirente in corso...
Charles può procedere all'acquisto!
Tentativo di pagamento tramite bonifico in corso...
2500000.0 pagato con bonifico effettuato da Charles all'IBAN MN654GH, rimangono 2500000.0 €
E' in corso l'invio dell'ordine alla fabbrica per la LaFerrari...
INVIO COMPLETATO!
E' stata scelta la versione BASE, che comprende un motore Ferrari a benzina da 900 cv
Charles , l'auto Le verrà consegnata tra 60 giorni
L'auto scelta verrà mostrata in una nuova scheda
Numero di LaFerrari impostato a 0
Costo annuo: assicurazione = 10000.0 €, super bollo = 11000.0 €, per un totale di 21000.0 €
Costo mensile: assicurazione = 833.3333 €, super bollo = 916.6667 €, per un totale di 1750.0 €/mese
La polizza annua rappresenta il 0.84 % del budget
Sono passati 841 giorni e sono stati percorsi 20076 km dalla data dell'acquisto
E' necessario portare la vettura in officina per la revisione
Revisione in corso...
Revisione completata!
Attualmente sono disponibili:
0 LaFerrari
4 SF90Stradale
5 Testarossa
SPOT PUBBLICITARIO IN TRASMISSIONE...
SPOT TERMINATO
Charles ha un indice di gradimento del 11 %
Charles non è interessato all'acquisto della vettura scelta.
Attualmente sono disponibili:
0 LaFerrari
4 SF90Stradale
5 Testarossa
RIEPILOGO DELL'AFFARE CON IL SIG.Charles :
Numero di auto vendute: 2, guadagno totale: 7500000.0 €.
Process finished with exit code 0

L'immagine dell'auto scelta viene mostrata in una nuova finestra dopo il messaggio “*L'auto scelta verrà mostrata in una nuova scheda*”



Questa è la finestra pop-up che si apre quando viene visualizzata l'immagine della SF90 Stradale



Questa invece è la finestra che si apre quando viene visualizzata l'immagine della LaFerrari.



Nell'esecuzione del mio programma, l'acquirente risulta non interessato all'acquisto di una Testarossa, ma ne riporto comunque la finestra che si è aperta durante un'altra esecuzione.

6) UNIT TESTS SVOLTI

In questa sezione vengono analizzati gli **Unit tests** svolti. I metodi marcati come `@BeforeClass` vengono eseguiti prima di tutte le altre istruzioni, `@Before` indica che quello specifico metodo verrà ripetuto prima di ogni test, `@Test` identifica un metodo come un caso di test, `@After` indica che quel metodo verrà ripetuto dopo ogni test, mentre `@AfterClass` indica un metodo che viene eseguito come ultima istruzione all'interno della classe.

6.1) AcquirenteTest

Per testare la classe **Acquirente**, utilizzo innanzitutto un metodo statico marcato come

```
public class AcquirenteTest {
    28 usages
    private static Acquirente acq;

    ▲ andreameritaBB *
    @BeforeClass
    public static void initialize() {
        // Acquirente inizializzato con sf90 deluxe
        acq = Acquirente.getInstance( tipoAuto: 1, versione: 1, budget: 10000000, nome: "Charles", cardNumber: 789543, cvc: 997, IBAN: "MN654GH");
        assertNotNull( message: "Acquirente non inizializzato", acq);
    }

    ▲ andreameritaBB *
    @Before
    public void setUp() throws Exception {
        acq.setTipoAuto(1);
        acq.setVersione(1);
        assertNotNull( message: "Acquirente non inizializzato", acq);
    }

    ▲ andreameritaBB *
    @Test
    public void attesaSF90DeluxeMaggioreUguale75(){
        assertEquals( message: "SF90 deluxe richiede 95", expected: 95, acq.calcolaAttesa(acq.getTipoAuto(), acq.getVersione()));
        acq.setVersione(0);
        assertEquals( message: "SF90 standard richiede 75", expected: 75, acq.calcolaAttesa(acq.getTipoAuto(), acq.getVersione()));
    }
}
```

`@BeforeClass` chiamato `initialize()`, che ha il compito di inizializzare l'acquirente come prima istruzione; all'interno di questo metodo troviamo anche l'asserzione `assertNotNull(acq)`, la quale, eventualmente, ci informerà con il messaggio fornito di un'errata inizializzazione dell'acquirente.

Il metodo `setUp()`, marcato come `@Before`, verrà eseguito prima di ogni caso di test: esso si occupa di reimpostare il tipo e la versione dell'auto a SF90 Stradale Deluxe e di controllare che l'istanza dell'acquirente sia correttamente inizializzata.

Il primo caso di `@Test` si ha con il metodo `attesaSF90DeluxeMaggioreUguale75()`: come facilmente intuibile, servirà a verificare che il tempo di attesa che l'acquirente dovrà aspettare dopo aver ordinato una SF90 Stradale Deluxe sia ≥ 75 giorni (infatti, nel metodo `acq.calcolaAttesa(tipo,versione)`, nel caso di versione deluxe il tempo di attesa viene aumentato di 20 giorni per la SF90 Stradale, mentre risulta di 75 giorni in caso di versione standard). Come seconda istruzione viene impostata la versione standard, quindi con l'asserzione `assertEquals()` andiamo a controllare che il tempo di attesa risulti di esattamente 75 giorni.

Proseguendo, troviamo il `@Test polizzaSF90uguale23500LaFerrari21000()`: anche qui, dal nome si capisce che l'intento è verificare che la polizza totale per la SF90 Stradale sia di 23500 €, mentre quella per la LaFerrari di 21000 €.

Nel `@Test revisioneTest()`, imposto a 1100 i giorni passati dalla consegna dell'auto e a 5000 i km percorsi: voglio verificare che, poiché siamo sopra i 1095 giorni, è il momento di effettuare la

revisione dell'auto (anche se non sono stati superati i 20000 km); successivamente imposto a 1000 i giorni passati per controllare che il metodo `acq.checkRevisione(giorniPassati, kmPercorsi)` restituisca `false`, dato che nessuna delle due condizioni viene verificata (`giorniPassati >= 1095 || kmPercorsi >= 20000`).

L'ultimo `@Test` della classe è `cancellaOrdineSeMaggiore40PercentoBudget()`: nella prima asserzione vogliamo controllare che l'ordine non viene cancellato se l'acquirente, con budget di 10.000.000 €, sceglie come prima auto la SF90 Stradale deluxe, mentre nella seconda, invece, ci aspettiamo che il metodo `acq.cancellaOrdine(polizza)` restituisca `true`, cioè che l'ordine venga cancellato, dal momento che abbiamo passato una polizza di 5.000.000 € rappresentante più del 40% del budget.

L'ultimo metodo, etichettato come `@AfterClass`, è `deallocate()`: esso viene eseguito come ultima istruzione della classe e non fa altro che deallocare l'acquirente.

```

@Text
public void polizzaSF90uguale23500LaFerrari121000(){
    SF90StradaleDeluxe sf90StradaleDeluxe = new SF90StradaleDeluxe( hp: 1050, satNav: true, leatherWheel: true, price: 5000000, acq, method: null, catalogo: null);
    assertEquals( message: "Polizza SF90 sbagliata", expected: 23500, acq.calcolaPolizza(sf90StradaleDeluxe.getHp()), delta: 0.0);
    LaFerrariDeluxe ferrariDeluxe = new LaFerrariDeluxe( hp: 963, satNav: true, leatherWheel: true, price: 3000000, acq, method: null, catalogo: null);
    assertEquals( message: "Polizza LaFerrari sbagliata", expected: 21000, acq.calcolaPolizza(ferrariDeluxe.getHp()), delta: 0.0);
}

@Text
public void revisioneTest() throws InterruptedException {
    acq.setGiorniPassati(1100);
    acq.setKmPercorsi(5000);
    assertTrue( message: "Richiamo revisione errato", acq.checkRevisione(acq.getGiorniPassati(), acq.getKmPercorsi()));
    acq.setGiorniPassati(1000); // Con 1000 giorni e 5000 km non è tempo di revisione
    assertFalse( message: "Richiamo revisione errato", acq.checkRevisione(acq.getGiorniPassati(), acq.getKmPercorsi()));
}

@Text
public void cancellaOrdineSeMaggiore40PercentoBudget(){
    SF90StradaleDeluxe sf90StradaleDeluxe = new SF90StradaleDeluxe( hp: 1050, satNav: true, leatherWheel: true, price: 5000000, acq, method: null, catalogo: null);
    assertFalse( message: "Ordine cancellato anche se costo rientra nella soglia", acq.cancellaOrdine(acq.calcolaPolizza(sf90StradaleDeluxe.getHp())));
    assertTrue( message: "Ordine non cancellato se costo troppo alto", acq.cancellaOrdine( polizza: 5000000));
}

@Text
@AfterClass
public static void deallocate() { acq = null; }
}

```

6.2) ControlloBudgetTest

Per testare la classe **ControlloBudget**, definisco inanzitutto il metodo statico `@BeforeClass initialize()`, che inizializza l'acquirente e lo assegna alla classe che stiamo testando.

Abbiamo poi il metodo `@Before setUp()` che con un'asserzione `assertNotNull()` controlla che l'acquirente sia correttamente inizializzato prima di ogni test case.

L'unico `@Test` di questa classe è rappresentato da `cancellaSeBudgetMinorePrezzo()`: l'obiettivo è verificare che l'acquirente possa o meno procedere all'acquisto in base al proprio budget. In particolare, le prime due `assertTrue()` indicheranno che l'acquirente, con budget di 5.000.000 €, può acquistare l'auto scelta che costerà 4.000.000 € nel primo caso e 5.000.000 € nel secondo (infatti, la condizione nel metodo `ControlloBudget.checkBudget(price)` è che `budget >= price`), mentre l'`assertFalse()` indicherà che il budget è inferiore, anche se solo di 1 €, al prezzo e che quindi l'acquisto non può essere completato.

```

public class ControlloBudgetTest {
    4 usages
    private static Acquirente acq;

    ▲ andreareset88
    @BeforeClass
    public static void initialize() {
        acq = Acquirente.getInstance( tipoAuto: 0, versione: 1, budget: 5000000, nome: "Charles", cardNumber: 12344, cvv: 444, IBAN: "NBV678UI");
        ControlloBudget.setAcq(acq);
    }

    ▲ andreareset88
    @Before
    public void setUp() { assertNotNull( message: "Acquirente non inizializzato", acq); }

    ▲ andreareset88 *
    @Test
    public void cancellaSeBudgetMinorePrezzo(){
        assertTrue( message: "Acquisto non consentito con prezzo giusto", ControlloBudget.checkBudget( price: 4000000));
        assertTrue( message: "Acquisto non consentito con prezzo giusto", ControlloBudget.checkBudget( price: 5000000));
        assertFalse( message: "Acquisto consentito anche con budget piccolo", ControlloBudget.checkBudget( price: 5000001));
    }

    ▲ andreareset88
    @After
    public void tearDown() throws Exception {
        acq = null;
    }
}

```

Infine troviamo il metodo `@After` `tearDown()` che dealloca l'acquirente (avrei potuto etichettarlo come `@AfterClass`, ma essendoci un solo test case risulta indifferente)

6.3) RivenditoreTest

Nella classe **RivenditoreTest** testeremo, oltre alla classe **Rivenditore**, anche i metodi della classe **Catalogo** (dato che nell'inizializzazione del rivenditore è necessario fornire il catalogo, a questo punto ho scelto di testare anche quest'ultimo qui e risparmiare un'ulteriore classe).

Troviamo quindi il metodo `@Before` `setUp()` che inizializza l'acquirente, il catalogo e il rivenditore, seguito dal primo `@Test` riguardante il catalogo, `numeroAutoDisponibiliMinoreUguale5()`: qui andreamo semplicemente a verificare , tramite 3 `assertTrue()`, che dopo l'inizializzazione del catalogo il numero di ciascuna macchina sia compreso nell'intervallo [0,5].

```

@Before
public void setUp() throws Exception {
    acq = Acquirente.getInstance( tipoAuto: 2, versione: 0, budget: 6000000, nome: "Charles", cardNumber: 987654, cvv: 777, IBAN: "MNFR567");
    cat = new Catalogo();
    riv = Rivenditore.getInstance(acq, method: null, cat);
}

▲ andreareset88
@Test
public void numeroAutoDisponibiliMinoreUguale5(){
    assertTrue( message: "Numero auto non corretto", condition: cat.getNumeroLaFerrari() <= 5 && cat.getNumeroLaFerrari() >= 0);
    assertTrue( message: "Numero auto non corretto", condition: cat.getNumeroSF90Stradale() <= 5 && cat.getNumeroSF90Stradale() >= 0);
    assertTrue( message: "Numero auto non corretto", condition: cat.getNumeroTestarossa() <= 5 && cat.getNumeroTestarossa() >= 0);
}

▲ andreareset88
@Test
public void checkAutoDisponibili(){
    cat.setNumeroLaFerrari(3);
    assertTrue( message: "LaFerrari non disponibile", cat.isLaFerrari());
    cat.setNumeroSF90Stradale(0);
    assertFalse( message: "SF90 Stradale non disponibile", cat.isSF90Stradale());
}

```

aspetto che `cat.isSF90Stradale()` renda `false`.

Il secondo test case riguardante il catalogo è rappresentato da `checkAutoDisponibili()`: impostiamo a 3 il numero di LaFerrari e verifichiamo, tramite `assertTrue()`, che il metodo `cat.isLaFerrari()` renda `true`. Dopo, setto a 0 il numero di SF90 Stradale e tramite `assertFalse()` mi

Il primo `@Test` riguardante il rivenditore è `profittoTest()`: imposto il profitto a 450 €, poi tramite un `assertTrue()` controllo che il metodo `riv.aggiornaProfitto(2000000)` restituisca correttamente un guadagno di 2000450 €.

Troviamo quindi il secondo e ultimo test case per il rivenditore, `pubblicaEIndiceGradimentoTrae100()`, che non fa altro che chiamare `riv.pubblica()` e controllare che l'indice di gradimento risultante stia in [0,100].

Troviammo in fondo il metodo **@After** `tearDown()` che dealloca l'acquirente, il catalogo e il rivenditore.

```
@Test
public void profittoTest(){
    riv.setProfitto(450);
    assertEquals( message: "Calcolo profitto non corretto", riv.aggiornaProfitto( euro: 2000000), actual: 2000450, delta: 0.0);
}

andreaset88
@Test
public void pubblicizzaEIndiceGradimentoTra0e100() throws InterruptedException {
    assertTrue( message: "Percentuale di gradimento incorretta", condition: riv.pubblicizza() >= 0 && riv.pubblicizza() <= 100);
}

andreaset88
@After
public void tearDown() throws Exception {
    acq = null;
    cat = null;
    riv = null;
}
```

6.4) LaFerrariDeluxeTest, SF90StradaleTest e TestarossaDeluxeTest

Ho riunito queste 3 classi di test poiché, anche se su oggetti diversi, vado ad eseguire gli stessi identici test; riporto quindi, per comodità, solo gli snippet del codice riguardante la prima classe, **LaFerrariDeluxeTest**.

Abbiamo quindi il metodo **@BeforeClass** `initialize()` che inizializza l'acquirente, il catalogo, il rivenditore e la LaFerrari Deluxe e setta l'acquirente per la classe **ControlloBudget**.

```
@BeforeClass
public static void initialize() {
    acq = Acquirente.getInstance( tipoAuto: 0, versione: 1, budget: 6000000, nome: "Charles", cardNumber: 98765, cvc: 889, IBAN: "MN789GH");
    ControlloBudget.setAcq(acq);
    cat = new Catalogo();
    riv = Rivenditore.getInstance(acq, method: null, cat);
    laferrari = new LaFerrariDeluxe( hp: 963, satNav: true, leatherWheel: true, price: 3000000, acq, new TransferStrategy(acq, riv), cat);
}

andreaset88
@Test
public void creaLaFerrariDeluxeTest() throws InterruptedException {
    cot.setNumeroLaFerrari(1);
    assertNotNull( message: "LaFerrari Deluxe non creata", laferrari.create());
}

andreaset88
@Test
public void nonCreareSeBudgetTroppoBasso() throws InterruptedException {
    acq.setBudget(1000000);
    assertNull( message: "Oggetto creato anche se budget troppo basso", laferrari.create());
}

andreaset88
@AfterClass
public static void deallocate() {
    acq = null;
    cat = null;
    riv = null;
    laferrari = null;
}
```

Il primo **@Test** è `createLaFerrariDeluxeTest()`: qui imposto il numero di LaFerrari a 1 per essere sicuro che almeno una sia disponibile, poi tramite un' `assertNotNull()` eseguo il metodo `laferrari.create()` e verifico che l'oggetto restituito non sia **null**.

L'ultimo test è `nonCreareSeBudgetTroppoBasso()`, dove imposto inizialmente il budget dell'acquirente a 1.000.000 €, per controllare quindi che eseguendo `laferrari.create()` si riceva un **null**, poiché il budget risulta troppo basso per procedere all'acquisto.

In fondo troviamo il metodo **@AfterClass** `deallocate()` che dealloca gli oggetti utilizzati.

Quest'ultimo snippet mostra che tutti i test sono stati eseguiti con successo, dunque i metodi scritti fanno effettivamente quello che ci si aspetta che facciano!

The screenshot shows the JUnit test results for the `FerrariFactory` package. All 15 tests have passed, taking 1 minute and 2 seconds. The output window displays the test names and their execution times, along with the console logs for each test case.

```

All in FerrariFactory[SWE] ×
Tests passed: 15 of 15 tests – 1 min 2 sec
<default package> 1 min 2 sec "C:\Program Files\Java\jdk-17\bin\java.exe" ...
Attualmente sono disponibili:
2 LaFerrari
0 SF90Stradale
4 Testarossa
Numero di SF90 Stradale impostato a 3
SF90 Stradale disponibile all'acquisto!
Controllo budget acquirente in corso...
Charles può procedere all'acquisto!
Tentativo di pagamento tramite bonifico in corso...
4500000.0 pagato con bonifico effettuato da Charles all'IBAN MNG654HJ, rimangono 0.0 €
Invio dell'ordine alla fabbrica per la SF90 Stradale in corso...
INVIO COMPLETATO!
E' stata scelta la versione BASE, che include un motore ibrido da 1000 cvCharles , l'auto Le verrà consegnata tra 75 giorni
SF90 Stradale disponibile all'acquisto!
Controllo budget acquirente in corso...
ERRORE, budget troppo basso, Charles non può procedere all'acquisto..
Siamo spiacenti, l'invio non è andato a buon fine
Attualmente sono disponibili:
4 LaFerrari
4 SF90Stradale
1 Testarossa
SPOT PUBBLICITARIO IN TRASMISSIONE...
SPOT TERMINATO
Charles ha un indice di gradimento del 26 %
SPOT PUBBLICITARIO IN TRASMISSIONE...
SPOT TERMINATO

```

7) FONTI

- Immagine SF90 Stradale:
<https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.hdmotori.it%2F2019%2Fo5%2F29%2Fferrari-ibrida-sf90-stradale-foto-interni-velocita%2F&psig=AovVaw1RslWAFK5UmAHgUgbF2GMW&ust=1651829725607000&source=images&cd=vfe&ved=oCAwQjRxqFwoTCICg5LqHyPcCFQAAAAAdAAAAABAD>
- Immagine LaFerrari:
<https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.gommeblog.it%2Ftuning%2Fsuperca%2F42569%2Fferrari-laferrari-video%2F&psig=AOvVawopWHp7ZxHbqqcBu5XSUaHY&ust=1651829946876000&source=images&cd=vfe&ved=oCAwQjRxqFwoTCLjyyJ6IyPcCFQAAAAAdAAAAABAD>
- Immagine Testarossa: https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.classic-trader.com%2Fit%2Fautomobili%2Fsearch%2Fferrari%2Ftestarossa&psig=AOvVaw2YQ46u3ZPulyL_sLuwom5S&ust=1651830035779000&source=images&cd=vfe&ved=oCAwQjRxqFwoTCIjq4MulyPcCFQAAAAAdAAAAABAD
- Teoria sui design patterns: https://e-l.unifi.it/pluginfile.php/1829136/mod_folder/content/o/3.%20Design%20Patterns/DesignPatternsJava20.pdf?forcedownload=1 (Slide fornite dal professor Vicario per il corso di Ingegneria Del Software dell'Università degli Studi di Firenze) e alcuni cenni dal libro del professor Marco Bertini, “*Programmazione Object-Oriented in C++ : Design Pattern e introduzione alle buone pratiche di programmazione*”, Società Editrice Esculapio.

- Sotware usato per la creazione degli artefatti UML: **StarUML** ([link download](#))
- IDE usato per la scrittura del codice JAVA: **IntelliJ IDEA Ultimate 2022.1** ([link download](#))

GRAZIE PER L'ATTENZIONE!