Relazione Tecnica e Architetturale del Sistema di Bacheca Annunci

Pietro Mezzatesta

Settembre 2025

Sommario

1 Introduzione

Il progetto Sistema di Bacheca Annunci è stato sviluppato adottando i principi fondamentali della **Programmazione Orientata agli Oggetti** e seguendo un'architettura di tipo **Model-View-Controller (MVC)**. La programmazione a oggetti organizza il software attorno a "oggetti" che rappresentano entità reali o concettuali, legando dati e metodi che li manipolano. In particolare, il paradigma OOP enfatizza quattro pilastri: incapsulamento (i dati interni sono protetti da accesso diretto), ereditarietà (le classi figlie estendono classi genitori utilizzando comportamenti comuni), polimorfismo (oggetti diversi possono essere trattati attraverso interfacce comuni) e astrazione (si modellano concetti generici mediante classi astratte o interfacce). Nel nostro caso, queste scelte garantiscono modularità e riuso: ad esempio, le classi AnnuncioAcquisto e AnnuncioVendita estendono l'astratta Annuncio, ereditando attributi e metodi generici ma personalizzando il comportamento specifico (es. la scadenza per le vendite).

L'architettura MVC separa il sistema in tre componenti logiche interconnesse: il **Modello**, che contiene la logica di business e le entità principali (classi Annuncio , Utente , Bacheca , ecc.); la **Vista**, che gestisce l'interfaccia utente (nel progetto, rappresentata sia dalla CLI che dalla GUI); e il **Controllore**, che media l'interazione tra Vista e Modello richiamando le operazioni del Modello in risposta agli input

dell'utente. Questa suddivisione consente di isolare i cambiamenti in un ambito (ad es. la GUI) senza impattare il cuore logico, migliorando manutenibilità e estendibilità del codice. Ad esempio, mentre il Modello è del tutto indipendente dall'interfaccia, le componenti di Vista/Controller (classi BachecaCLI , BachecaGUI) interagiscono con il modello invocando i metodi di Bacheca e mostrando i risultati.

Il design del sistema si propone inoltre di utilizzare **eccezioni personalizzate** per la gestione degli errori di dominio, anziché far abortire il programma inaspettatamente. In tal modo, gli errori semantici (utente non trovato, operazione non autorizzata, data di scadenza invalida, ecc.) vengono segnalati chiaramente e possono essere gestiti in modo controllato, migliorando la robustezza complessiva. Infine, la persistenza dei dati (salvataggio/caricamento di annunci e utenti su file) è realizzata tramite la serializzazione degli oggetti Java, utilizzando l'interfaccia marker Serializable. Questo permette di salvare su disco lo stato completo del modello e ripristinarlo all'avvio successivo.

2 Architettura Generale

Il progetto è organizzato in più **package** separati per responsabilità. Il *Modello* risiede nel package *dominio* , che contiene le entità di dominio e la logica principale. In dettaglio:
- **dominio**: classi Annuncio (astratta), AnnuncioAcquisto, AnnuncioVendita, Bacheca e Utente. Queste classi incapsulano i dati (campi) e forniscono i metodi di manipolazione necessari.
- **eccezioni**: package per le classi di eccezione personalizzate (AnnuncioNonTrovatoException, UtenteNonTrovatoException, UtenteNonAutorizzatoException, UtenteGiaRegistratoException, DataScadenzaNonValidaException), utilizzate dal modello per segnalare condizioni d'errore di dominio.
- **persistence**: package dedicato alla persistenza e ai servizi di supporto. Include ad esempio BachecaPersistence (metodi statici per caricare/salvare la Bacheca e la mappa utenti), Autenticazione (gestisce login e registrazione interagendo con Bacheca), e Validator (utility per convalida di email e password).
- **interfaccia**: package per la logica di interazione con l'utente. Comprende BachecaCLI (interfaccia a riga di comando) e BachecaGUI (interfaccia grafica Swing) che presentano menu e form, e mediano le richieste verso il modello Bacheca.
- **test**: package contenente le classi di test JUnit, che verificano il corretto funzionamento delle componenti del modello e della persistenza.

Questa suddivisione in package riflette i livelli architetturali: il modello di dominio è del tutto separato dalle viste e dalla persistenza. Il Main avvia il sistema caricando dati (package persistence) e poi l'interfaccia (CLI/GUI). Ciò segue il principio "Separation of Concerns" (separazione delle responsabilità), tipico dell'architettura MVC, che migliora la manutenibilità e permette eventuali estensioni o modifiche limitate. Ad esempio, si potrebbe aggiungere una nuova vista (una web app) senza toccare la logica di business, oppure cambiare la modalità di salvataggio (passare a un database) restando nel package persistence .

3 Classi del Modello

Le classi di dominio sono nel package *dominio*. Di seguito le descrizioni dettagliate.

3.1 Classe Annuncio

La classe astratta Annuncio rappresenta un'inserzione generica nella bacheca. È marcata abstract e implementa Serializable, permettendo la sua persistenza su file. Annuncio incapsula i dati base comuni a qualsiasi annuncio:
- *id* (String, final): identificatore univoco dell'annuncio. Viene generato automaticamente (tramite UUID) e non può cambiare, garantendo l'unicità assoluta in bacheca.
- *autore* (Utente, final): l'utente che ha creato l'annuncio. Anch'esso è immutabile una volta creato, in modo da assicurare che l'autore non possa essere alterato (coerenza semantica).
- *nomeAnnuncio* (String): titolo/nome dell'annuncio.
- *prezzo* (double): prezzo offerta/richiesta.
- *paroleChiave* (Set): insieme di parole chiave assocciate all'annuncio per la ricerca.

Questi campi sono dichiarati private, applicando **l'incapsulamento**: possono essere letti/modificati solo tramite metodi pubblici controllati (getter/setter). Ad esempio, getNomeAnnuncio() , getPrezzo() , getParoleChiave() forniscono un accesso di sola lettura, mentre setNomeAnnuncio(), setPrezzo(), setParoleChiave() consentono modifiche con eventuale validazione. In questo modo si proteggono gli invarianti interni e si garantisce che il modello rimanga consistente (ad es. impedendo nome o prezzo nulli/vuoti). Tutti i metodi setter di Annuncio lanciano IllegalArgumentException se i nuovi valori non sono validi, mantenendo il controllo interno.

Metodi speciali: toString() è sovrascritto per restituire una rappresentazione leggibile dell'annuncio (utile per debug o visualizzazione). Inoltre Annuncio definisce l' isScaduto() come **metodo astratto**: le sottoclassi concrete (AnnuncioAcquisto, AnnuncioVendita) devono implementarlo diversamente. Questo riflette l'astrazione: la logica generale di un annuncio è definita qui, mentre i dettagli (es. "come stabilire se è scaduto") sono specifici per tipo.

Non sono definite eccezioni personalizzate nella classe Annuncio stessa, in quanto ogni validazione sui campi viene gestita mediante eccezioni standard (IllegalArgumentException o simili). Eventuali operazioni di dominio più complesse (come l'aggiunta o rimozione dalla bacheca) sollevano eccezioni definite in altri componenti.

3.2 Classi AnnuncioAcquisto e AnnuncioVendita

AnnuncioAcquisto e AnnuncioVendita sono sottoclassi di Annuncio che incarnano il concetto di annuncio di "richiesta" e di "offerta". Entrambe ereditano tutti i campi e metodi di Annuncio. La principale differenza è semantica e funzionale:

- AnnuncioAcquisto: rappresenta l'intenzione di acquistare un oggetto/servizio. Non introduce nuovi campi o metodi: la sua esistenza serve solo a distinguere semanticamente dalle vendite. Il metodo isScaduto() in questa classe è banalmente implementato per restituire sempre false, poiché gli annunci di acquisto non hanno scadenza.
- AnnuncioVendita: rappresenta l'offerta di vendita di qualcosa. Introduce un campo aggiuntivo essenziale:
- dataScadenza (LocalDate): data oltre la quale l'annuncio non è più valido. Nel costruttore AnnuncioVendita(...) , oltre a impostare tutti i campi ereditati, viene richiesto un parametro LocalDate dataScadenza. Se la data non è strettamente futura (ad esempio odierna o passata), viene lanciata un'eccezione personalizzata DataScadenzaNonValidaException, garantendo coerenza temporale. Vengono forniti getter e setter per la scadenza; anche il setter verifica che la nuova data sia valida (altrimenti solleva DataScadenzaNonValidaException). Il metodo isScaduto() è **override** per restituire true se la dataScadenza è precedente o uguale alla data corrente, altrimenti false. In questo modo il sistema può riconoscere quando una vendita è terminata. Inoltre, toString() è sovrascritto per includere le informazioni di scadenza nel testo.

Dal punto di vista OOP, queste subclassi implementano il principio di **ereditarietà** e **polimorfismo**: mantengono le caratteristiche comuni (nome, prezzo, paroleChiave, autore) e aggiungono solo ciò che serve. AnnuncioVendita e AnnuncioAcquisto possono essere usati in modo intercambiabile quando si tratta di collezioni generiche di Annuncio, sebbene operino diversamente su alcuni metodi (polimorfismo).

3.3 Classe Bacheca

La classe Bacheca è il cuore logico del sistema, gestendo l'insieme di annunci e utenti. Implementa Serializable per consentire di salvarne facilmente lo stato, e anche Iterable<Annuncio> per poter iterare comodamente sugli annunci interni. Internamente, la bacheca utilizza due mappe:
- Map<String, Annuncio> annunci: mappa gli ID univoci a oggetti Annuncio. Grazie alla struttura Map, operazioni come aggiunta, rimozione e ricerca per ID sono efficienti (tipicamente O(1)).
- Map <String, Utente> utentiRegistrati : mappa email normalizzate (chiavi univoche) a oggetti Utente. L'email è usata come chiave perché è identificatore unico per ogni utente.

Principali metodi e responsabilità della classe Bacheca (che agisce quasi da "controller" di logica di business):

- **Registrazione utente**: registraUtente(Utente utente) registra un nuovo utente. Controlla che l'oggetto utente non sia nullo, che l'email non sia già presente, e in caso contrario inserisce il mapping. Se c'è un conflitto (email già registrata) viene lanciata UtenteGiaRegistratoException .
- **Autenticazione**: la bacheca fornisce getUtente(email) per recuperare un utente data l'email, e isUtenteRegistrato(email) per verificare rapidamente l'esistenza di un utente. Questi metodi sono usati dalla classe Autenticazione (nel package persistence) per gestire login/registrazione.
- **Aggiunta annuncio**: aggiungiAnnuncio(Annuncio a) inserisce un annuncio in bacheca. Verifica che l'annuncio non sia nullo, che abbia un ID valido, e che non esista già un annuncio con lo stesso ID (per evitare duplicati). In caso di errore lancia IllegalArgumentException. Altrimenti fa semplicemente annunci.put(a.getId(), a).
- **Rimozione annuncio**: rimuoviAnnuncio(Annuncio a, Utente u) rimuove un annuncio esistente. Controlla che l'annuncio non sia nullo e che effettivamente esista nella mappa (AnnuncioNonTrovatoException se non

c'è). Inoltre verifica che l'utente corrente sia lo stesso autore dell'annuncio: se no, lancia UtenteNonAutorizzatoException. Solo l'autore può eliminare il proprio annuncio. Su successo, l'annuncio viene rimosso dalla mappa.

- **Modifica annuncio**: modificaAnnuncio(id, utenteCorrente, nuovoNome, nuovoPrezzo, nuoveParoleChiave, nuovaDataScadenza) modifica i dati di un annuncio esistente. Controlla che l'ID e l'utente siano validi, che l'annuncio esista e che l'utente sia l'autore. Poi applica in modo facoltativo gli aggiornamenti (se i parametri non sono nulli/vuoti): nome, prezzo, parole chiave (normalizzandole), e, nel caso di AnnuncioVendita, anche la nuova data di scadenza (controllando la validità). Solleva eccezioni apposite (AnnuncioNonTrovatoException, UtenteNonAutorizzatoException, DataScadenzaNonValidaException) se falliscono i controlli.
- **Ricerca annunci**: cercaAnnunci(Set parole, Utente utenteCorrente) restituisce una lista di annunci di tipo AnnuncioVendita che contengono almeno una delle parole chiave indicate. Le parole chiave degli annunci vengono normalizzate (trim, lowercase); la ricerca esclude automaticamente gli annunci il cui autore è l'utente corrente (per non mostrare i propri annunci di vendita nella ricerca) e confronta le parole chiave delle richieste con quelle salvate. In presenza di set di parole null o vuoti, la lista risultante è vuota.
- **Pulisci bacheca**: pulisciBacheca() scorre tutti gli annunci e rimuove quelli di tipo AnnuncioVendita per i quali isScaduto() risulta true. Conta e restituisce quanti annunci sono stati eliminati, Gli AnnuncioAcquisto non vengono mai rimossi qui (non hanno scadenza).
- **Elenchi e helper**: metodi come getAnnunciPerAutore(Utente u) restituiscono tutti gli annunci pubblicati da un certo utente, filtrando la mappa annunci. Metodi getter come getAnnunci() e getUtentiRegistrati() restituiscono copie delle strutture interne (per non esporre direttamente il campo privato). setUtenti(Map<...>) serve a ricaricare la mappa utenti (ad es. dopo la lettura da file) normalizzando le chiavi. getNumeroAnnunci() e getNumeroUtentiRegistrati() restituiscono i conteggi attuali. Infine iterator() permette di iterare sugli annunci, rendendo la bacheca iterabile.

La classe Bacheca non è serializzabile "automaticamente" dal package dominio; invece la persistenza è gestita tramite la classe BachecaPersistence. Tuttavia, implementando Serializable, ogni istanza di Bacheca (con tutte le sue mappe) può essere salvata e ripristinata. In sintesi, Bacheca coordina tutte le operazioni sugli

annunci e utenti, gestendo anche la logica di autorizzazione e validazione, servendosi delle eccezioni definite.

3.4 Classe Utente

La classe Utente modella un utente registrato nella bacheca. Ogni utente ha i seguenti campi chiave:

- **email** (String, final): identificatore unico e immutabile dell'utente, normalizzato in minuscolo senza spazi. Viene usato come chiave nella mappa utenti di Bacheca
- **nome, cognome** (String): dati anagrafici, modificabili con controlli (non possono essere nulli o vuoti).
- **passwordHash, salt** (String, entrambi final): per la sicurezza, la password in chiaro non viene mai salvata. Al posto di questa, l'oggetto conserva un **hash crittografico** della password. In fase di costruzione (new Utente(email, nome, cognome, password)), la password viene validata (controllando criteri di complessità tramite la classe Validator), quindi viene generato un salt casuale e calcolato l'hash (SHA-256) della password combinata con il salt. L'hash e il salt sono memorizzati, mentre la password in chiaro viene abbandonata. Questo approccio garantisce che, anche in caso di compromissione dei dati, le password non siano immediatamente recuperabili. L'utilizzo del salt (un valore unico casuale per ogni utente) rende gli hash unici: **due utenti con la stessa password** otterranno hash diversi, contrastando possibili attacchi.

I metodi pubblici di Utente includono getter per nome, cognome, email, e setter che accettano solo valori validi (altrimenti IllegalArgumentException). Non esistono setter né passwordHash per né per salt, che sono final. Per verificare la password, è presente il metodo checkPassword(String pass): questo calcola l'hash della password fornita con lo stesso salt e lo confronta con l'hash memorizzato. Se coincidono, restituisce true, altrimenti false. In questo modo l'utente può autenticarsi senza mai gestire la password originale.

Utente ridefinisce inoltre equals() e hashCode() basandosi **solo sull'email**, in quanto essa costituisce l'identificatore univoco. Due oggetti Utente con stessa email sono considerati uguali. Infine, il metodo toString() fornisce una descrizione testuale formattata dell'utente, utile per log e debug.

L'implementazione della classe Utente segue le buone pratiche di sicurezza per lo storage delle password: nessuna password in chiaro è mai conservata, e l'uso di salt randomici aumenta la robustezza contro attacchi o matching di hash. Il framework prevede quindi di confrontare sempre gli hash tramite checkPassword, proteggendo la password in ogni fase.

4 Gestione delle Eccezioni Personalizzate

Il sistema definisce diverse eccezioni personalizzate nel package eccezioni, tutte estendono Exception. Esse sono usate per segnalare condizioni di errore semantiche in fase di esecuzione, rendendo chiaro il motivo del fallimento. Le principali eccezioni sono:

- **AnnuncioNonTrovatoException**: sollevata quando un'operazione su un annuncio fallisce perché l'annuncio richiesto non è presente nella bacheca (ad esempio durante rimuoviAnnuncio o modificaAnnuncio).
- **UtenteNonTrovatoException**: indica che non esiste un utente registrato con l'email specificata, ad esempio durante il login.
- **UtenteNonAutorizzatoException**: sollevata quando un utente tenta un'operazione non consentita, ad esempio cercare di modificare o rimuovere l'annuncio di un altro. Bacheca lancia questa eccezione se l'utente corrente non è l'autore dell'annuncio target.
- **UtenteGiaRegistratoException**: scatenata quando si cerca di registrare un nuovo utente con un'email già presente nel sistema (registraUtente).
- **DataScadenzaNonValidaException**: emessa da AnnuncioVendita quando la data di scadenza fornita (nel costruttore o nel setter) non è valida (ad es. non futura). Questo previene la creazione di vendite già scadute o con data odierna.

Ogni eccezione personalizzata dispone di un costruttore che accetta un messaggio descrittivo. Queste eccezioni non sono "unchecked" ma richiedono la gestione (o la dichiarazione con throws) nei metodi di alto livello (come nel controller CLI/GUI). Ad esempio, i metodi del controller catchano queste eccezioni per mostrare all'utente un messaggio di errore chiaro invece di terminare l'applicazione. L'uso di eccezioni di dominio rende più leggibile il flusso di controllo: è immediatamente chiaro se un fallimento è dovuto a un problema di dato mancante o a un'operazione non permessa.

5 Persistenza

Per preservare i dati tra esecuzioni, il progetto utilizza l**a serializzazione degli oggetti Java**. In pratica, le istanze di Bacheca (che contengono tutte le mappe di utenti e annunci) e la mappa degli utenti vengono scritte su file come flussi di byte. Questo è possibile perché Bacheca e Utente implementano Serializable; inoltre tutte le classi annuncio (e le collezioni usate) sono anch'esse serializzabili.

La classe di supporto BachecaPersistence (nel package persistence) offre metodi statici: salvaBacheca(Bacheca b, String file) e caricaBacheca(String file), oltre a salvaUtenti(Map) e caricaUtenti(). Internamente, questi usano ObjectOutputStream e ObjectInputStream per scrivere/leggere. Ad esempio, SalvaBacheca() apre un FileOutputStream e un ObjectOutputStream e chiama writeObject(bacheca). Nel caricamento si fa readObject() dentro un ObjectInputStream e si castano i byte letti all'oggetto Bacheca. Se il file non esiste, caricaBacheca restituisce null (il codice chiamante inizializza allora una nuova bacheca), mentre caricaUtenti restituisce una mappa vuota se il file non è presente.

Questa strategia rende la persistenza **trasparente**: l'intero stato del modello viene salvato in un colpo solo, senza dover gestire manualmente formati di testo o DB. Naturalmente, è responsabilità dell'applicazione serializzare quando necessario. Nel codice, la GUI prevede un'operazione "Esci e salva" che chiama BachecaPersistence.salvaBacheca e salvaUtenti. All'avvio, il Main prova a leggere bacheca.ser e utenti.ser; se il file della bacheca esiste, lo carica insieme agli utenti, altrimenti crea una nuova istanza vuota. I messaggi debug informano l'esito del caricamento.

In sintesi, l'uso di **Serializable** consente di convertire lo stato degli oggetti del modello in un flusso di byte e viceversa. L'interfaccia Serializable è "marker" e non definisce metodi, ma la classe deve dichiararla per poter essere serializzata. In questo progetto non sono stati definiti metodi di scrittura/ lettura personalizzati (writeObject / readObject), si utilizza la serializzazione standard Java. Eventuali eccezioni I/O (IOException, ClassNotFoundException) sono gestite e spesso mostrano errori in console se il file è corrotto o mancante (nel test incluso si verifica il caricamento di file corrotti o vuoti).

6 Interfacce Utente

Il sistema fornisce sia un'interfaccia testuale (CLI) che una grafica (GUI) per interagire con l'utente. Entrambe si connettono al modello Bacheca e seguono la logica MVC/Separation: la vista raccoglie input, il controller (parte delle classi stesse) invoca il modello, poi la vista mostra l'output.

- **BachecaCLI** (package interfaccia): gestisce il menu e i comandi in riga di comando. Al lancio, mostra opzioni numerate (es. inserisci annuncio, rimuovi annuncio, cerca per parole chiave, pulisci, visualizza miei annunci, modifica annuncio). Per ogni scelta, viene chiamato un metodo interno che acquisisce dati tramite la console (richiama i metodi di Input.readString()), li valida, e richiama i metodi Bacheca corrispondenti (ad es. aggiungiAnnuncio, cercaAnnunci , ecc.). Ad esempio, per inserire un annuncio chiede il tipo (A/V), nome, prezzo, data scadenza e parole chiave; poi crea l'oggetto AnnuncioAcquisto o AnnuncioVendita e lo passa a bacheca.aggiungiAnnuncio() . Eventuali eccezioni lanciate (es. DataScadenzaNonValidaException) sono catturate e stampate come messaggi d'errore all'utente. La CLI aggiorna anche lo stato dei file: in alcune opzioni di uscita richiama BachecaPersistence per salvare dati. Il codice di BachecaCLI mostra come vengono tradotte le scelte utente in chiamate al modello, e come la logica di autorizzazione viene applicata.
- **BachecaGUI** (package interfaccia): implementa la finestra grafica con Swing. All'avvio, crea un JFrame principale con titolo personalizzato (saluto all'utente). L'interfaccia utilizza un CardLayout con diversi pannelli: uno di menu principale con bottoni, e altri pannelli per ciascuna funzione (inserisci annuncio, cerca, rimuovi, visualizza, modifica). Ciascun pannello contiene campi di input (JTextField , JTextArea , ecc.) e bottoni. Ad esempio, il pannello d'inserimento ha caselle per tipo, nome, prezzo, scadenza, parole chiave, e un pulsante "Salva Annuncio" che chiama gestisciNuovoAnnuncio(). Questo metodo legge i valori dai campi, crea l'oggetto annuncio e invoca bacheca.aggiungiAnnuncio, gestendo eventuali eccezioni con JOptionPane di errore. La GUI visualizza i risultati (liste di annunci, conferme, errori) in un'area di testo centrale scrollabile. Azioni come "Esci" salvano i dati (richiamando BachecaPersistence e poi chiudendo l'applicazione). In sostanza, BachecaGUI funge sia da **vista** (mostra finestre, campi, bottoni) sia da **controller** (gestisce i listener dei bottoni, chiama il modello). La connessione con il modello avviene tramite l'oggetto Bacheca passato al costruttore di BachecaGUI insieme all' Utente corrente.

In entrambi i casi (CLI/GUI), l'interfaccia ricava l'utente autenticato tramite la classe Autenticazione e agisce su quella Utente. Poiché il modello (Bacheca) è indipendente dall'interfaccia, il codice utenteCorrente passa sempre come argomento quando serve controllare permessi (ad es. in rimuoviAnnuncio o modificaAnnuncio). Non ci sono componenti dedicate solo a "Vista" separate da controller: piuttosto ogni interfaccia fa anche le funzioni di controllo locale. Resta comunque rispettata la separazione logica: le classi UI non contengono la logica di business (come la ricerca vera e propria); questa è tutta delegata al modello Bacheca. Le interfacce si limitano a presentare menu, leggere input, mostrare risultati e a gestire i flussi di dialogo (es. chiedere conferma, feedback all'utente).

7 Testing con JUnit

Il sistema è stato sottoposto a test unitari estesi con **JUnit 5** (framework standard per testing Java). Le classi di test coprono le funzionalità critiche del modello e della persistenza, verificando che ogni metodo pubbico si comporti come previsto. In particolare sono state create classi di test per:

- **AnnuncioAcquisto** (AnnuncioAcquistoTest): verifica che il costruttore crei correttamente l'oggetto, che getter e setter funzionino, e che i parametri invalidi (null, stringhe vuote, prezzo negativo, set di parole chiave nullo) generino IllegalArgumentException . Si controlla anche che le parole chiave siano normalizzate (spazi rimossi) e che isScaduto() ritorni sempre false.
- **AnnuncioVendita** (AnnuncioVenditaTest): analogo ad Acquisto, ma include test specifici per la data di scadenza. Ad esempio, costruire con una data nel passato deve fallire con DataScadenzaNonValidaException, e isScaduto() deve riflettere correttamente il confronto con la data corrente.
- **Utente** (UtenteTest): testa il costruttore (validazione di email, password, nome/cognome) e il metodo checkPassword. Verifica che le password vengano accettate solo se corrette (grazie all'hash con salt) e rifiutate se sbagliate.
- **Bacheca** (BachecaTest): copre tutte le operazioni di bacheca. Crea una bacheca di test con utenti e annunci preimpostati e verifica metodi come aggiungiAnnuncio (e che rimuovere con utente sbagliato lanci UtenteNonAutorizzatoException), cercaAnnunci (compresi casi con parole chiave multiple o annuncio proprio escluso), pulisciBacheca (rimuove i soli annunci scaduti), registraUtente (con email duplicata dovrebbe generare UtenteGiaRegistratoException), ecc. Controlla inoltre i metodi di accesso (getAnnunciPerAutore , isUtenteRegistrato , ecc.).

- **BachecaPersistence** (BachecaPersistenceTest): verifica la serializzazione. Ad esempio, salva una bacheca di test su file e la ricarica, poi confronta che i dati coincidano (getNumeroAnnunci, getNumeroUtenti, esistenza degli stessi annunci tramite ID). Testa anche i casi di file inesistente (dovrebbe ritornare null o mappa vuota senza eccezione) e di file corrotto/vuoto (il caricamento deve restituire null). Per gli utenti salva una mappa di utenti e verifica che, dopo il caricamento, tutti gli utenti sono presenti e le loro password funzionano con checkPassword. Viene inoltre testata un'integrazione completa: salvare bacheca+utenti e ricaricarli combinando i dati (impostando con setUtenti) per verificarne l'integrità.

Gli unit test utilizzano le **asserzioni** di JUnit (assertEquals, assertThrows, assertTrue, ecc.) per controllare il comportamento atteso. Grazie ai @BeforeEach preparano scenari noti, e ogni test singolo controlla un frammento di funzionalità. L'insieme dei test garantisce un'ampia copertura di casi: costruttori validi/invalidi, eccezioni lanciate correttamente, risultati delle ricerche, persistenza funzionante. I test automatizzati sono integrati in un ciclo di sviluppo guidato dai test (TDD): prima si definisce il test e poi si fa sì che il codice lo passi. Questo approccio riduce i bug ed aiuta a mantenere la qualità del codice. In generale, JUnit consente di rilevare immediatamente regressioni se del codice viene modificato: le suite di test esistenti falliscono in caso di comportamento inatteso. Le classi testate coprono così le parti più rilevanti del modello (annunci, bacheca, utenti) e della persistenza.

8 Conformità ai Requisiti della Consegna

Il progetto soddisfa puntualmente tutti i requisiti richiesti:

1. **Principi OOP e MVC**: il codice segue i paradigmi OOP descritti (incapsulamento, ereditarietà tra annunci, astrazione) e l'architettura rispetta lo schema MVC evidenziato (Modello indipendente, interfacce separate, persistenza come servizio).
2. **Architettura a livelli**: come visto, il modello è nel package package dominio, l'interfaccia utente nell'interfaccia, e i servizi trasversali (Autenticazione, Validator, BachecaPersistence) in persistence. Le responsabilità sono ben suddivise.
3. **Classi di dominio dettagliate**: Per ciascuna classe, tutti i campi private, metodi pubblici e costruttori sono stati definiti con le dovute validazioni (null, vuoti) e i commenti Javadoc ne descrivono lo scopo. In particolare,

Annuncio è abstract con campi autore, nomeAnnuncio, id, prezzo, paroleChiave; le sottoclassi dettagliano l'unica differenza (dataScadenza per Vendita). La classe Bacheca gestisce mappe e logica come richiesto, e Utente ha email, nome, cognome, gestione sicura della password con hash e salt.

4. **Eccezioni personalizzate**: implementate tutte le eccezioni indicate dalle specifiche. Esse vengono lanciate nei punti giusti (ad es. UtenteGiaRegistratoException in registraUtente ...ecc.).
   Questo garantisce che ogni errore venga comunicato chiaramente.
5. **Persistenza**: la consegna richiedeva l'uso di Serializable e la struttura di file per salvataggio/caricamento. Il progetto lo fa attraverso BachecaPersistence: i file bacheca.ser e utenti.ser contengono i dati serializzati. Le operazioni caricaBacheca, salvaUtenti, salvaBacheca, caricaUtenti gestiscono il flusso di dati. L'architettura dei file è semplice (due file .ser nella directory). Il caricamento iniziale (Main) verifica e agisce di conseguenza.
6. **Interfacce CLI e GUI**: entrambe sono implementate e funzionanti.
   La CLI (BachecaCLI) mostra menu testuali e chiede input, la GUI (BachecaGUI) utilizza Swing con menu a pulsanti e campi di input. Entrambe invocano i metodi di Bacheca per realizzare le funzioni (inserimento, rimozione, ricerca, ecc.) e aggiornano l'utente tramite testo a console o finestre di dialogo. La logica è coerente (es. protezione dell'autore, visualizzazione messaggi di errore se le eccezioni vengono sollevate). Le classi gestiscono correttamente il collegamento con il modello passato al costruttore.
7. **Testing JUnit**: come specificato, sono inclusi test per le classi principali del modello (AnnuncioAcquistoTest, AnnuncioVenditaTest, UtenteTest , BachecaTest , BachecaPersistenceTest). Coprono sia casi normali che di errore, con asserzioni complete. L'approccio garantisce un'alta copertura delle funzionalità del modello (creazione, lettura, modifica, cancellazione) e della persistenza. I risultati del caricamento/salvataggio sono verificati tramite test file. Questo rispetta la richiesta di test funzionali e copertura dati.

In sintesi, ogni punto della consegna è stato implementato e documentato sistematicamente. Eventuali differenze minori (ad esempio l'uso di IllegalArgumentException anziché eccezioni personalizzate per campi nulli) sono scelte di design già viste in classi analoghe. Nel complesso, il progetto rispetta i requisiti specificati.

9 Estensioni Future Possibili

Per migliorare ulteriormente il sistema, si potrebbero considerare diverse estensioni:

- **Database relazionale**: sostituire la serializzazione file con un database SQL (ad es. SQLite, MySQL) per la persistenza.
- **Interfaccia Web o Mobile**: sviluppare una nuova vista (front-end web con HTML/JS o un'app mobile) collegata via REST al back-end. Grazie all'architettura MVC, basterebbe aggiungere servizi e controller web che usino il modello esistente, senza alterare la logica di business.
- **Ruoli utente**: introdurre ruoli diversi (es. amministratore) con permessi aggiuntivi (es. moderare annunci o utenti). Ciò richiederebbe un'estensione di Utente con campo ruolo e controlli extra in Bacheca .
- **Notifiche**: aggiungere un meccanismo di notifiche (es. avvisi di scadenza imminente o nuovi annunci che corrispondono a chiavi salvate).
- **Miglioramento sicurezza**: usare algoritmi di hashing più robusti anziché SHA-256. Anche la gestione delle password potrebbe essere migliorata con salvataggio sicuro dei salt separatamente.
- **Ricerca avanzata**: implementare ricerca full-text o filtri avanzati (fascia di prezzo, data).
- **UI migliorate**: rifinire la GUI grafica (layout responsivo, immagini).
- **Internazionalizzazione**: definire l'interfaccia in altre lingue, mettendo testi su file di risorse.
- **Logging e configurazione**: aggiungere logging dettagliato e parametri di configurazione per rendere il sistema più configurabile senza ricompilare.

Queste estensioni amplierebbero le funzionalità e la robustezza del progetto, permettendo all'applicazione di scalare o adattarsi a scenari più complessi mantenendo comunque la struttura a oggetti già definita.

10 Conclusioni

Il **Sistema di Bacheca Annunci** realizzato applica efficacemente i principi della programmazione orientata agli oggetti e l'architettura Model-View-Controller. L'analisi dettagliata ha mostrato come ogni classe del modello (Annuncio, Utente, Bacheca) sia progettata attraverso i principi fondamentali della OOP, e come le eccezioni personalizzate migliorino la gestione degli errori. La persistenza basata sulla serializzazione di Java semplifica il salvataggio dei dati, mentre le interfacce CLI e GUI offrono modalità d'uso diverse, entrambe collegate correttamente al modello. I test JUnit forniscono una copertura completa delle funzionalità, garantendo affidabilità. Nel complesso, il sistema soddisfa i requisiti della consegna e costituisce una base solida per futuri sviluppi. L'adozione di pratiche OOP e la chiara separazione dei livelli rendono il codice manutenibile e pronto a estensioni, come l'integrazione con un database o interfacce aggiuntive. In definitiva, questa implementazione dimostra un'architettura coerente e ben strutturata, facilitando la lettura, il test e la manutenzione futura del software.