UNIVERSITATEA TEHNICĂ DIN CLUJ-NAPOCA

Tema 3: Gestionarea Clădirilor folosind Design Patterns

Student: Tcaci Liviu

Inginerie Software

Universitatea Tehnică din Cluj-Napoca

10 Decembrie 2024

Cuprins

1 Introducere

În contextul dezvoltării aplicațiilor software complexe, utilizarea șabloanelor de proiectare (design patterns) este esențială pentru a asigura un cod modular, scalabil și ușor de întreținut. Acest document prezintă implementarea unei aplicații pentru gestionarea clădirilor într-o localitate, utilizând șabloane de proiectare creational și structural: **Abstract Factory** și **Bridge**.

2 Descrierea Proiectului

Aplicația dezvoltată are scopul de a gestiona diferite tipuri de clădiri dintr-o localitate, precum case, blocuri, spitale și școli. Fiecare tip de clădire are atribute specifice și poate fi afișată în diferite formate (JSON, CSV, XML).

2.1 Structura Proiectului

- **src/**: Codul sursă al aplicației.
  - `models.py`: Definiții ale claselor de bază (`Casa`, `Bloc`, `Spital`, `Scoala`).
  - `factory.py`: Implementarea Abstract Factory (`CladireFactory`, `ConcreteCladireFactory`
  - `program_principal.py`: Clasa `ProgramPrincipal` pentru gestionarea clădirilor.
  - `display.py`: Implementarea Bridge pentru afișarea clădirilor în diferite formate.
  - `command.py`: Implementarea șablonului de proiectare Command.
  - `invoker.py`: Clasa `Invoker` pentru gestionarea comenziilor.
  - `main.py`: Punctul de intrare al aplicației.
- **resources/**: Diagrame UML înainte și după optimizare.
- **docs/**: Documentație suplimentară și justificări.
  - `justificari.md`: Justificarea alegerii Abstract Factory.
  - `display_justificari.md`: Justificarea alegerii Bridge pentru afișare.
- **README.md**: Acest fișier.

3 Justificarea Alegerea Șabloanelor de Proiectare

În dezvoltarea aplicației noastre, am ales să utilizăm șabloane de proiectare creational și structural pentru a asigura o arhitectură robustă și scalabilă. În această secțiune, vom justifica alegerea șabloanelor **Abstract Factory** și **Bridge**.

3.1 Abstract Factory (Creational)

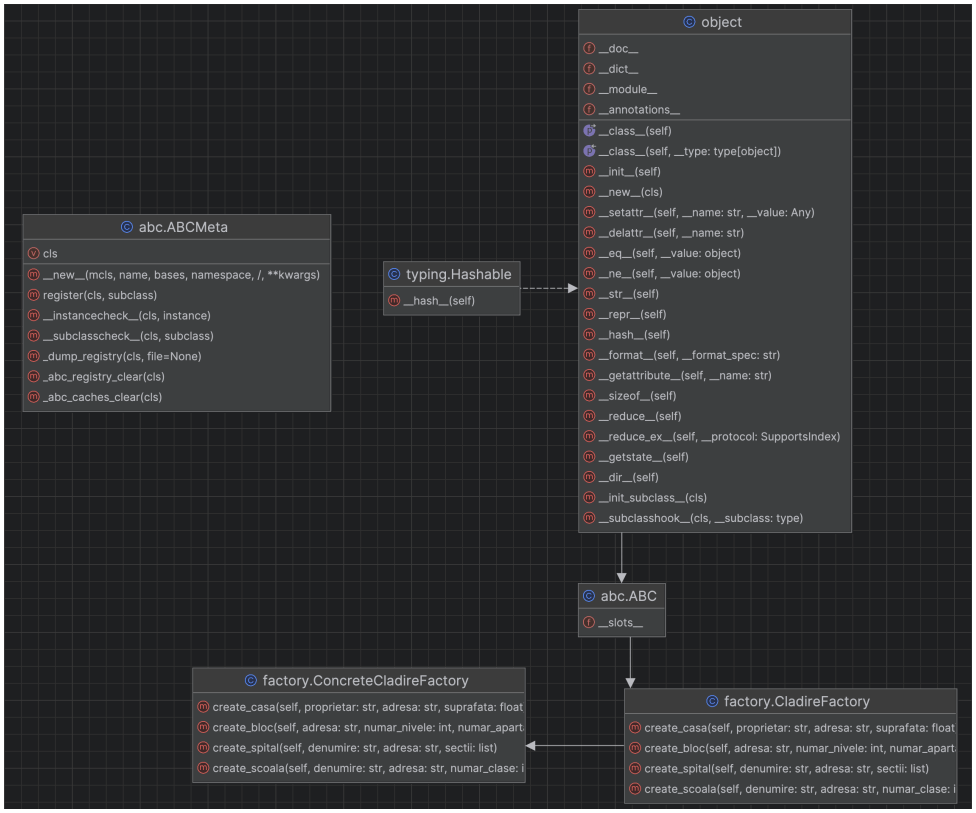

Figura 1: Diagrama UML pentru Abstract Factory.

3.1.1 Motivație

Gestionăm mai multe tipuri de clădiri: `Casa`, `Bloc`, `Spital`, și `Scoala`. Fiecare dintre acestea are propriile atribute și metode, dar toate fac parte dintr-o familie de produse (clădiri). Utilizarea șablonului **Abstract Factory** ne permite să creăm obiecte din diferite familii într-un mod organizat și consistent.

3.1.2 De ce Abstract Factory

- **Multiple Familii de Produse**: Având mai multe tipuri de clădiri, Abstract Factory facilitează crearea de obiecte din diverse familii fără a depinde de clasele concrete.
- **Scalabilitate**: Este ușor să adăugăm noi tipuri de clădiri fără a modifica codul existent. Trebuie doar să extindem factory-ul concret.
- **Independența de Implementare**: Clasa `ProgramPrincipal` nu cunoaște detaliile concrete ale instanțierii obiectelor, ceea ce reduce dependențele și crește modularitatea.

3.1.3 De ce nu Factory Method

- **Singură Familie de Produse**: Factory Method este mai potrivit când se lucrează cu o singură familie de produse, în timp ce noi gestionăm multiple tipuri de clădiri.

- **Complexitate**: Utilizarea Abstract Factory ne oferă o soluție mai potrivită pentru nevoile noastre, evitând complexitatea suplimentară care ar putea apărea cu Factory Method în contextul actual.

3.2 Bridge (Structural)

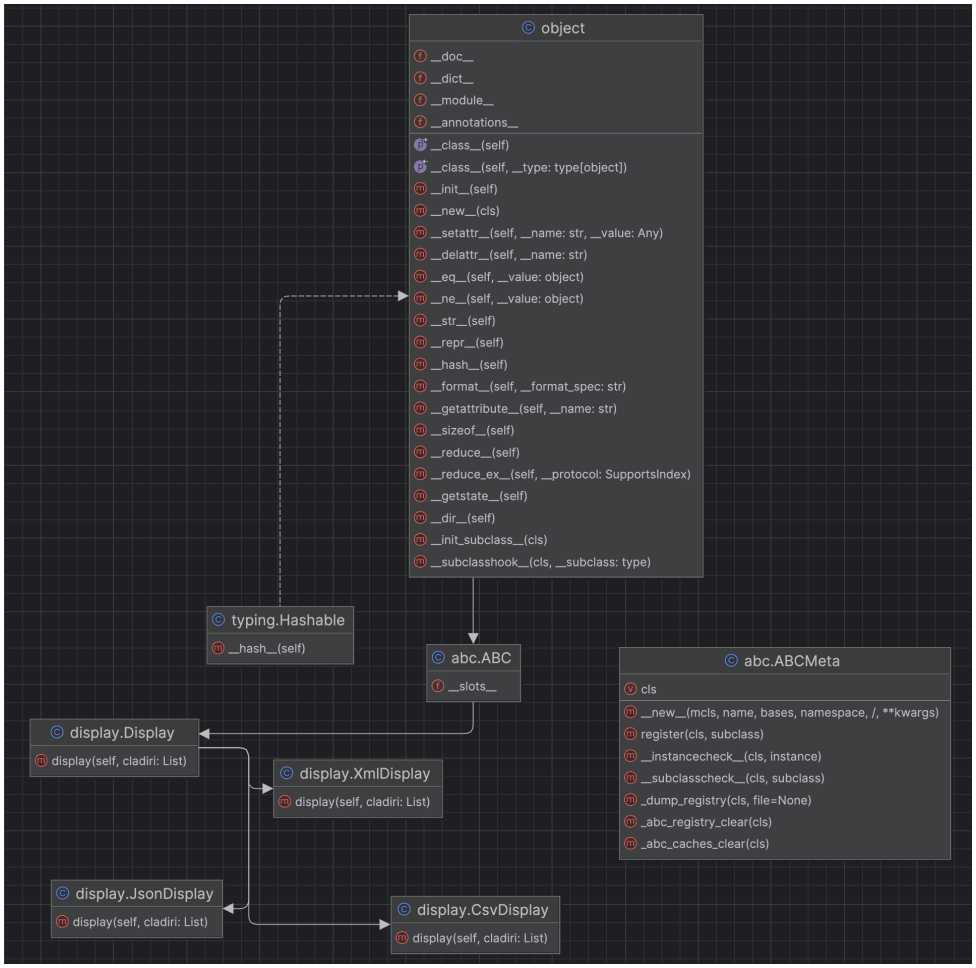

Figura 2: Diagrama UML pentru Bridge.

3.2.1 Motivație

Aplicația noastră necesită afișarea informațiilor despre clădiri în multiple formate: `JSON`, `CSV`, `XML`. Dorim să separăm logica de afișare de clasele de bază ale clădirilor pentru a facilita adăugarea de noi formate fără a modifica clasele existente.

3.2.2 De ce Bridge

- **Separarea Abstracției de Implementare**: Permite gestionarea independentă a logicii de afișare și a claselor de clădiri.
- **Scalabilitate**: Adăugarea unui nou format de afișare este simplă, doar prin crearea unei noi clase care implementează interfața `Display`.

- **Flexibilitate**: Permite combinarea diferitelor metode de afișare cu diversele tipuri de clădiri fără crearea de clase intermediare complexe.

3.2.3 De ce nu Adapter sau Decorator

- **Adapter**: Este mai potrivit pentru a face interoperabile două interfețe existente, ceea ce nu este cazul nostru.
- **Decorator**: Este util pentru adăugarea de comportamente suplimentare la obiecte, dar nu pentru separarea logicii de afișare de clasele de bază.

3.3 Command (Behavioral)

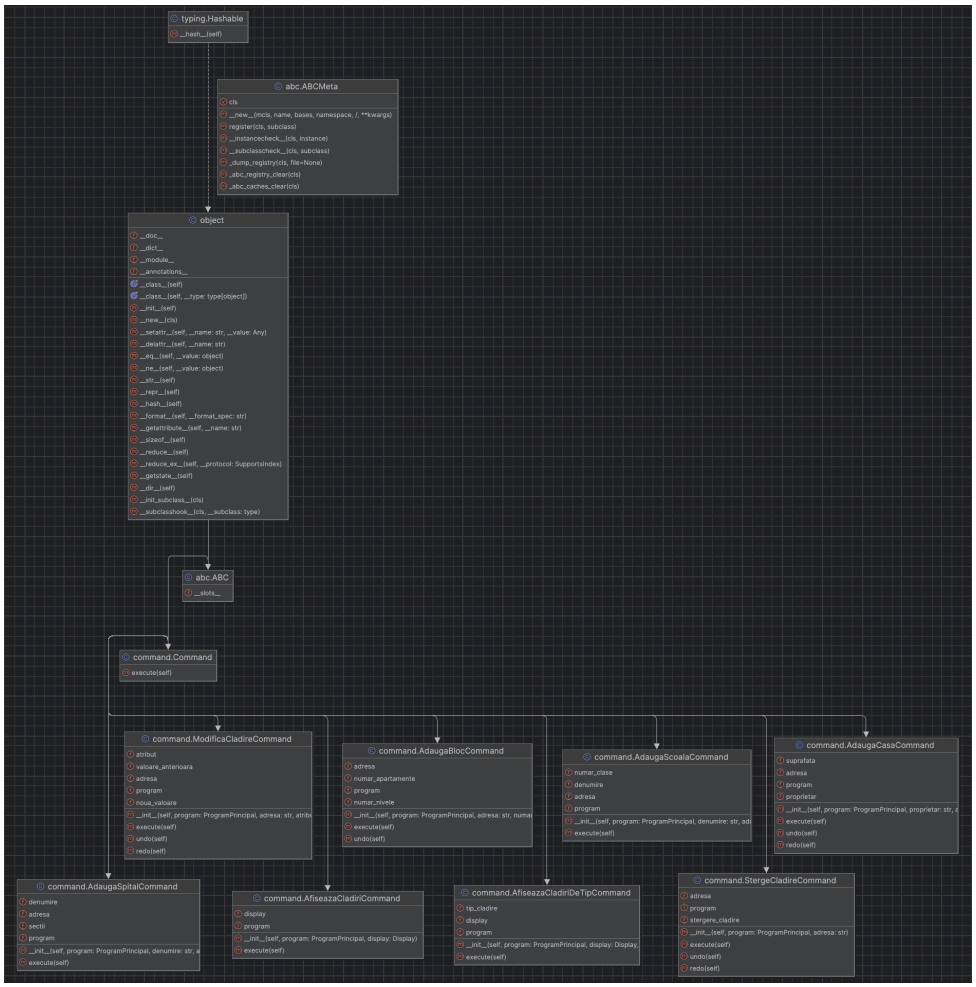

Figura 3: Diagrama UML pentru Command.

3.3.1 Motivație

Aplicația noastră necesită suport pentru operațiuni de **Undo** și **Redo**. Utilizarea șablonului **Command** ne permite să encapsulăm fiecare operațiune într-un obiect separat, facilitând gestionarea istoricului de comenzi și implementarea funcționalităților de **Undo/Redo**.

3.3.2 Beneficii

- **Encapsulare a Comenzilor**: Fiecare comandă este reprezentată printr-o clasă separată, facilitând extinderea și întreținerea.
- **Gestionarea Istoricului**: Obiectele de tip `Invoker` pot păstra istoricul comenzilor executate și pot gestiona operațiunile de **Undo/Redo**.
- **Flexibilitate**: Permite reordonarea, repetarea sau combinarea comenzilor într-un mod ușor de gestionat.

3.4 Invoker

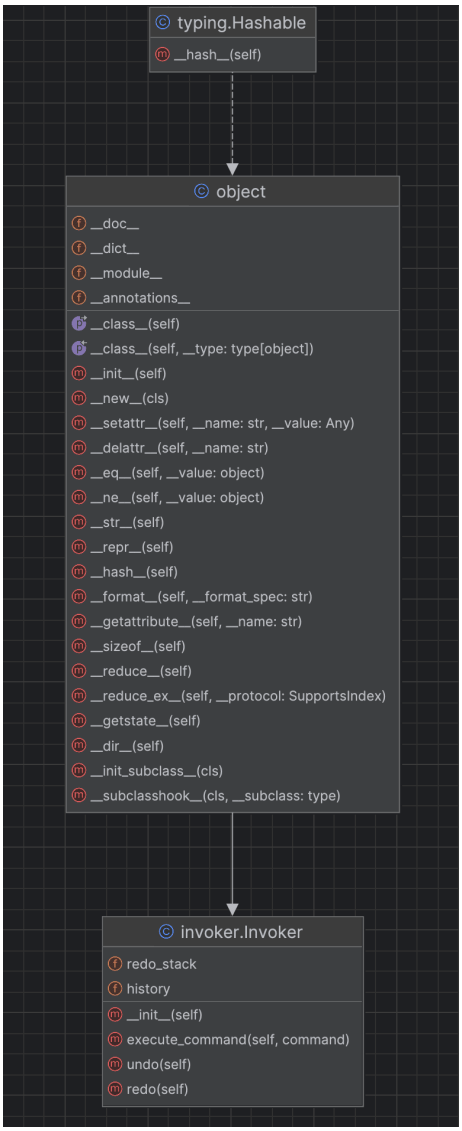

Figura 4: Diagrama UML pentru Invoker.

3.4.1 Rolul Invoker

Clasa `Invoker` gestionează executarea, anularea și refacerea comenzilor. Aceasta menține două stive: una pentru comenzi executate și alta pentru comenzi refăcute, facilitând astfel funcționalitățile de **Undo/Redo**.

4 Diagrame UML

4.1 Diagramă Clase

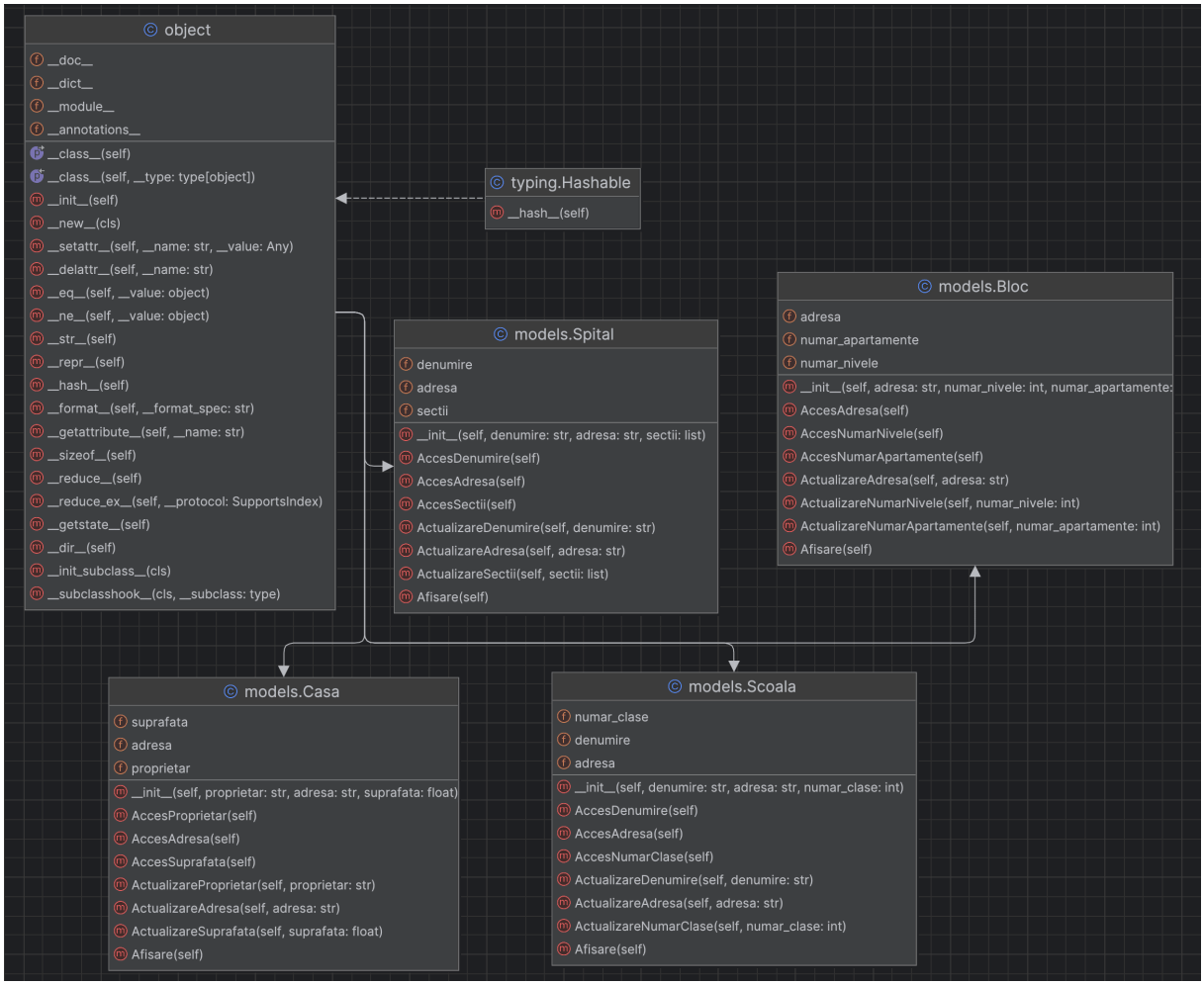

Figura 5: Diagrama UML pentru Clasele de Bază.

4.2 Diagramă Program Principal

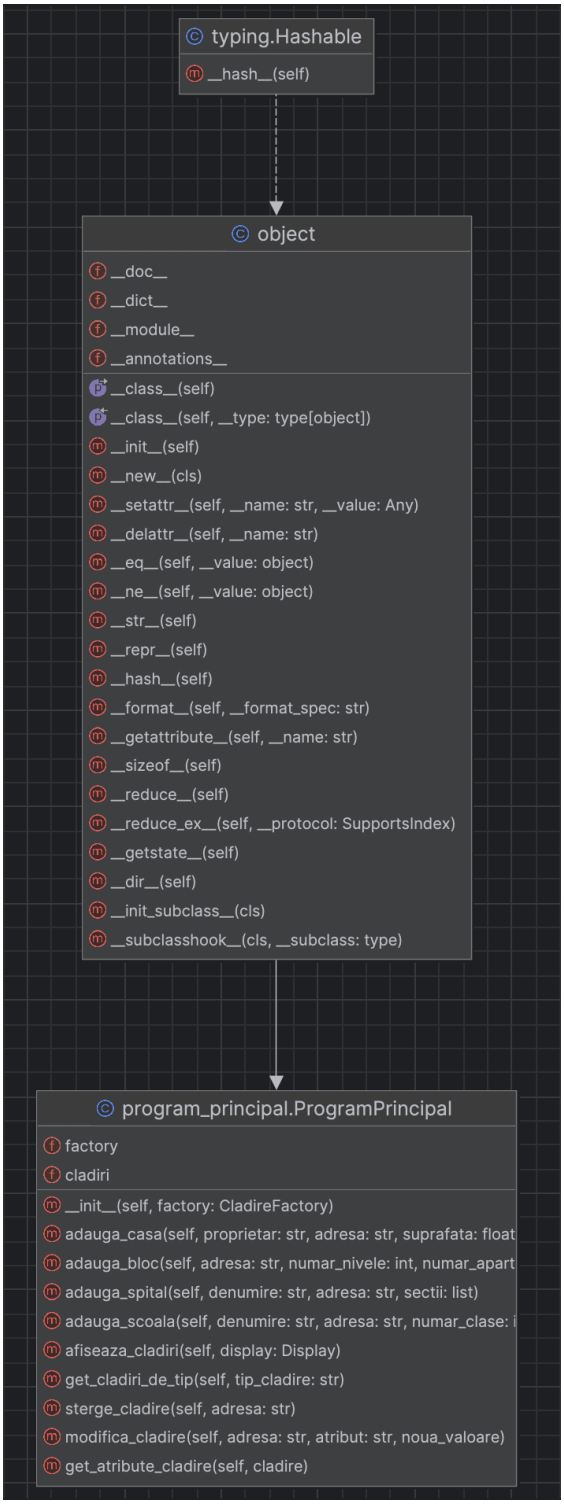

Figura 6: Diagrama UML pentru Clasa `ProgramPrincipal`.

5 Concluzii

Utilizarea șabloanelor de proiectare **Abstract Factory**, **Bridge** și **Command** a contribuit semnificativ la crearea unei aplicații modulare, scalabile și ușor de întreținut. Aceste șabloane au permis separarea clară a responsabilităților, facilitând adăugarea de noi funcționalități fără a afecta componentele existente.

6 Referințe

- Gamma, E., Helm, R., Johnson, R., & Vlissides, J. (1994). *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.
- Wikipedia contributors. (2023). *Abstract Factory Pattern*. `https://en.wikipedia.org/wiki/Abstract_factory_pattern`
- Wikipedia contributors. (2023). *Bridge Pattern*. `https://en.wikipedia.org/wiki/Bridge_pattern`
- Wikipedia contributors. (2023). *Command Pattern*. `https://en.wikipedia.org/wiki/Command_pattern`