Universitatea Tehnică a Moldovei

Facultatea Calculatoare Informatică și Microelectronică

Departamentul Inginerie Software și Automatică

RAPORT

Proiect de AN la TMPS



A efectuat: Plesca Alexandru st. gr. TI-204

A verificat: Gaidau Mihai

Chişinău – 2023

**Cuprins**

[**Introducere** 2](#_Toc136940193)

[**Design Pattern** 3](#_Toc136940194)

[**1.1 Modele de design creațional** 4](#_Toc136940195)

[**1.2 Modele de proiectare structurală** 5](#_Toc136940196)

[**1.3 Modele de design comportamental** 6](#_Toc136940197)

[**Diagrame UML** 8](#_Toc136940198)

[**2.1 Modele de design creațional** 9](#_Toc136940199)

[**2.2 Modele de proiectare structurală** 11](#_Toc136940200)

[**2.3 Modele de design comportamental** 13](#_Toc136940201)

[**Concluzie :** 15](#_Toc136940202)

[**Bibliografie :** 16](#_Toc136940203)

# **Introducere**

Aplicația este un sistem simplu de management al magazinului, conceput pentru a gestiona categorii și produse. Oferă funcționalități pentru crearea, preluarea, actualizarea și ștergerea categoriilor și a produselor printr-un API RESTful. Aplicația a fost construită pentru a facilita gestionarea inventarului unui magazin online, permițând utilizatorilor să organizeze produsele pe categorii și să efectueze diverse operațiuni asupra acestora.

Scop și caracteristici:

Scopul aplicației este de a eficientiza gestionarea categoriilor și a produselor din cadrul unui magazin. Oferă o modalitate convenabilă de a organiza și menține un catalog de articole, facilitând administratorii magazinelor să adauge, să modifice și să ștergă produse, precum și să gestioneze categoriile cărora le aparțin.

Caracteristici cheie:

1. Gestionarea categoriilor: utilizatorii pot crea categorii noi, pot prelua categorii existente, pot actualiza detaliile categoriei, cum ar fi titlul și descrierea, și pot șterge categorii dacă este necesar.
2. Managementul produselor: Produsele pot fi create în anumite categorii, preluate în funcție de categorie, actualizate cu informații noi și șterse din sistem.
3. Preluare date: aplicația oferă puncte finale pentru a prelua liste de categorii, categorii individuale prin identificatorii lor unici și produse din anumite categorii.
4. API RESTful: aplicația urmează principiile transferului de stat reprezentativ (REST) și oferă o interfață API curată și standardizată pentru interacțiunea cu sistemul de management al magazinului.
5. Metode HTTP: Metodele HTTP acceptate includ GET (preluarea datelor), POST (crearea de date noi), PATCH (actualizarea datelor existente) și DELETE (eliminarea datelor).

Aplicația servește drept fundație pentru construirea infrastructurii de backend a unui magazin online, permițând integrarea perfectă cu interfețe front-end, aplicații mobile sau alte sisteme care necesită acces la inventarul magazinului.

În general, aplicația de gestionare a magazinului simplifică procesul de organizare și gestionare a produselor, oferind o soluție eficientă pentru menținerea unui inventar actualizat și bine structurat pentru orice magazin online.

**Design Pattern**

Design Patterns (sau modele de proiectare) sunt soluții reutilizabile la probleme comune care apar în dezvoltarea software-ului. Acestea sunt abstracții și concepte generale care pot fi aplicate în diferite situații pentru a rezolva anumite provocări de proiectare.

Design Patterns își au rădăcinile în cartea "Design Patterns: Elements of Reusable Object-Oriented Software" scrisă de Erich Gamma, Richard Helm, Ralph Johnson și John Vlissides, cunoscută și sub numele de Cartea Gang of Four. Această carte descrie 23 de modele de proiectare clasice care au fost dezvoltate și rafinate de-a lungul anilor.

Un Design Pattern oferă o soluție elegantă la o problemă comună, stabilind o abordare recomandată pentru structurarea și organizarea codului. Ele pot acoperi diferite aspecte ale proiectării software-ului, inclusiv structura claselor, relațiile dintre obiecte, gestionarea fluxului de date și comunicarea între componente.

Folosirea Design Patterns poate aduce numeroase beneficii în dezvoltarea software-ului, precum creșterea reutilizabilității, ușurința înțelegerii și menținerea codului, reducerea complexității și creșterea flexibilității. Cu toate acestea, este important să înțelegi în mod corespunzător fiecare Design Pattern și să-l aplici în mod adecvat la problema ta specifică pentru a beneficia de avantajele sale.

## **1.1 Modele de design creațional**

* Acestea sunt cele 6 Creational Design Patterns clasice. Fiecare dintre acestea abordează diferite aspecte ale creării obiectelor și oferă o soluție optimizată pentru situații specifice. Este important să înțelegi contextul și nevoile tale pentru a alege cel mai potrivit Design Pattern creational în dezvoltarea ta de software.

1. Singleton (Singleton): Singleton este un Design Pattern creational care asigură că o clasă are o singură instanță și oferă un punct global de acces la această instanță. Este util în cazurile în care trebuie să existe o singură instanță a unei clase care să ofere acces global la acea instanță.
2. Factory Method (Metoda Fabrică): Factory Method este un Design Pattern creational care furnizează o interfață pentru crearea de obiecte, dar permite subclaselor să decidă ce clasă să instantiate. În esență, acesta abstractizează procesul de creare a obiectelor și oferă o flexibilitate mai mare în alegerea clasei concrete pe care să o instanțiezi.
3. Abstract Factory (Fabrică Abstractă): Abstract Factory este un Design Pattern creational care furnizează o interfață pentru crearea familiei de obiecte legate sau dependente, fără a specifica clasele concrete. Prin intermediul acestuia, poți crea obiecte care sunt concepute să funcționeze împreună și să fie coerente în cadrul unui context.
4. Builder (Constructor): Builder este un Design Pattern creational care separă construcția unui obiect complex de reprezentarea sa. Acesta permite crearea obiectului pas cu pas, oferind o metodă flexibilă de construire a unui obiect cu opțiuni diferite.
5. Prototype (Prototip): Prototype este un Design Pattern creational care permite crearea de obiecte noi prin clonarea unui obiect existent, numit prototip. Aceasta elimină dependența de crearea de obiecte folosind constructori și oferă o modalitate mai flexibilă de generare a obiectelor.
6. Object Pool (Pool de Obiecte): Object Pool este un Design Pattern creational care pune la dispoziție un set predefinit de obiecte inițializate și gata de utilizare, numite "pool", pentru a fi reutilizate. În loc să creezi și să distrugi obiecte în mod repetat, poți recicla obiectele din pool, economisind astfel timp și resurse.

## **1.2 Modele de proiectare structurală**

* Acestea sunt cele 7 Structural Design Patterns clasice. Fiecare dintre acestea abordează diferite aspecte ale organizării și interacțiunii între obiecte și oferă soluții pentru probleme specifice. Este important să înțelegi nevoile tale de proiectare și să alegi Design Pattern-ul structural adecvat pentru a obține o arhitectură software flexibilă și ușor de întreținut.

1. Adapter (Adaptor): Adapter este un Design Pattern structural care permite comunicarea între două clase care au interfețe incompatibile. Funcționează ca un intermediar între aceste clase, convertind solicitările de la una într-un format acceptat de cealaltă.
2. Bridge (Pod): Bridge este un Design Pattern structural care separă abstractizarea de implementare, permițându-le să evolueze independent. Este util atunci când există multiple abstracții și multiple implementări, și trebuie să le combinăm într-un mod flexibil.
3. Composite (Compus): Composite este un Design Pattern structural care permite tratamentul uniform al obiectelor individuale și a compozițiilor acestora. În esență, tratează un obiect individual și o colecție de obiecte în același mod, permițând astfel manipularea lor într-un mod transparent.
4. Decorator (Decorator): Decorator este un Design Pattern structural care permite adăugarea de funcționalități suplimentare la un obiect într-un mod dinamic, fără a modifica clasa de bază. Acesta utilizează încapsularea și compunerea pentru a extinde funcționalitatea unui obiect la runtime.
5. Facade (Fațadă): Facade este un Design Pattern structural care oferă o interfață simplificată pentru un subsistem complex de clase. Ascunde complexitatea din spatele unei interfețe unificate, permițând astfel o utilizare mai ușoară și mai intuitivă a subsistemului.
6. Flyweight (Obiect Ușor): Flyweight este un Design Pattern structural care optimizează utilizarea memoriei prin partajarea eficientă a obiectelor. În loc să creați un nou obiect pentru fiecare instanță, acesta reutilizează obiecte existente similare, economisind astfel resurse.
7. Proxy (Procuror): Proxy este un Design Pattern structural care furnizează o înlocuire sau o reprezentare a unui obiect real. Poate fi util în situații în care se dorește controlul suplimentar asupra accesului la obiectul real, precum accesul la distanță sau implementarea unui nivel de securitate.

## **1.3 Modele de design comportamental**

* Acestea sunt cele 9 Behavioral Design Patterns clasice. Fiecare dintre acestea abordează diferite aspecte ale comportamentului și interacțiunii între obiecte în cadrul unei aplicații software. Alegerea corectă a Design Pattern-urilor behaviorale potrivite poate duce la un design mai flexibil, modular și ușor de extins.

1. Observer (Observator): Observer este un Design Pattern behavioral care stabilește o relație de tip unu-la-mulți între obiecte. Atunci când un obiect se schimbă, toate obiectele dependente de acesta sunt notificate automat și actualizate.
2. Strategy (Strategie): Strategy este un Design Pattern behavioral care permite schimbarea comportamentului unei clase prin înlocuirea unei strategii. Aceasta separă algoritmul de implementarea specifică, permițând clasei să fie mai flexibilă și ușor de adaptat la diferite scenarii.
3. Chain of Responsibility (Lanț de Responsabilitate): Chain of Responsibility este un Design Pattern behavioral care permite transmiterea unei solicitări de-a lungul unei lanț de obiecte până când aceasta este gestionată de unul dintre aceste obiecte. Acesta descrește cuplarea între obiectul care solicită și obiectul care o prelucrează, oferind o flexibilitate mai mare.
4. Command (Comandă): Command este un Design Pattern behavioral care încapsulează o solicitare ca un obiect, permițând astfel parametrizarea clientului cu diferite solicitări, coadă sau înregistrare de solicitări și posibilitatea de a anula o solicitare. Acesta separă solicitarea de execuția acesteia.
5. Iterator (Iterare): Iterator este un Design Pattern behavioral care oferă o modalitate de a accesa elementele unui obiect de colecție într-un mod secvențial, fără a expune detaliile de implementare ale acestuia. Acesta permite parcurgerea eficientă a unei colecții fără a cunoaște structura internă.
6. Memento (Memento): Memento este un Design Pattern behavioral care capturează și salvează starea internă a unui obiect fără a încălca încapsularea. Acesta permite restaurarea obiectului la o stare anterioară și oferă un mecanism de revenire la stări anterioare.
7. State (Stare): State este un Design Pattern behavioral care permite unui obiect să-și schimbe comportamentul în funcție de starea internă. Acesta abstrage stările posibile ale unui obiect în clase separate și îi permite să treacă de la o stare la alta într-un mod controlat.
8. Template Method (Metodă Șablon): Template Method este un Design Pattern behavioral care definește scheletul unei algoritme într-o clasă de bază, lăsând implementarea detaliilor specifice subclaselor. Acesta oferă un mod convenabil de a defini o structură comună pentru mai multe clase.
9. Visitor (Vizitator): Visitor este un Design Pattern behavioral care separă algoritmul de obiectele pe care le operează. Permite adăugarea de noi operații la o ierarhie de obiecte fără a le modifica.

# **Diagrame UML**

O diagramă UML (Unified Modeling Language) este o reprezentare vizuală a unui sistem sau a unei aplicații software folosind simboluri și notații standardizate. Diagramele UML sunt utilizate pe scară largă în dezvoltarea de software și proiectarea sistemului pentru a comunica și documenta diferite aspecte ale structurii, comportamentului și interacțiunilor unui sistem.

Diagramele UML oferă o modalitate de a vizualiza diferite perspective ale unui sistem și de a capta elementele și relațiile sale cheie. Acestea servesc ca limbaj comun pentru dezvoltatori, designeri și părțile interesate pentru a comunica și a înțelege arhitectura și designul sistemului.

Există mai multe tipuri de diagrame UML, fiecare cu scopul și obiectivul său specific. Iată câteva tipuri comune de diagrame UML:

1. Diagrama de clasă: reprezintă structura statică a unui sistem, inclusiv clasele, atributele, metodele și relațiile acestora.
2. Diagrama de caz de utilizare: ilustrează interacțiunile dintre actori (utilizatori sau sisteme externe) și sistem, concentrându-se pe funcționalitatea sistemului și pe obiectivele utilizatorului.
3. Diagrama secvenței: arată interacțiunile dintre obiecte de-a lungul timpului, captând ordinea mesajelor schimbate între ele.
4. Diagrama activității: Prezintă fluxul activităților sau proceselor dintr-un sistem, prezentând secvența acțiunilor și deciziilor.
5. Diagrama mașinii de stări: reprezintă diferitele stări și tranziții ale unui obiect sau sistem, captând comportamentul acestuia ca răspuns la evenimente.
6. Diagrama componentelor: Vizualizează componentele fizice sau logice ale unui sistem și dependențele acestora.
7. Diagrama de implementare: arată componentele hardware și software dintr-un sistem și aranjarea lor fizică într-o rețea sau într-un mediu de calcul.

Aceste diagrame, împreună cu altele precum Diagramele pachetelor, Diagramele de comunicare și Diagramele de colaborare, oferă o vedere cuprinzătoare a arhitecturii, designului și interacțiunilor unui sistem.

Prin utilizarea diagramelor UML, profesioniștii în software își pot comunica și împărtăși ideile în mod eficient, pot analiza și valida proiectele de sistem și pot facilita colaborarea între membrii echipei implicați în dezvoltarea și întreținerea sistemului.

## **2.1 Modele de design creațional**

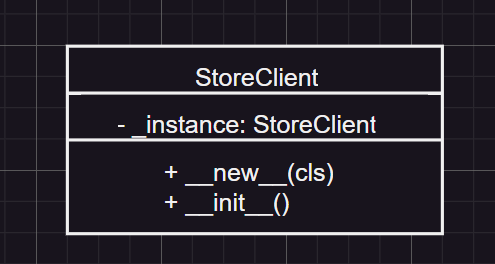


Figura 1.1 – Diagrama Singleton

În diagramă:

* Clasa StoreClient este reprezentată cu numele său.
* Atributul \_instance este notat ca o variabilă privată.
* Metodele \_\_new\_\_ și \_\_init\_\_ sunt indicate ca metode publice.

Modelul Singleton asigură crearea unei singure instanțe a clasei StoreClient în întregul program. Variabila privată \_instance deține o singură instanță a clasei. Metoda \_\_new\_\_ este suprascrisă pentru a verifica dacă o instanță există deja și, dacă nu, creează o instanță nouă. Metoda \_\_init\_\_ inițializează obiectul.

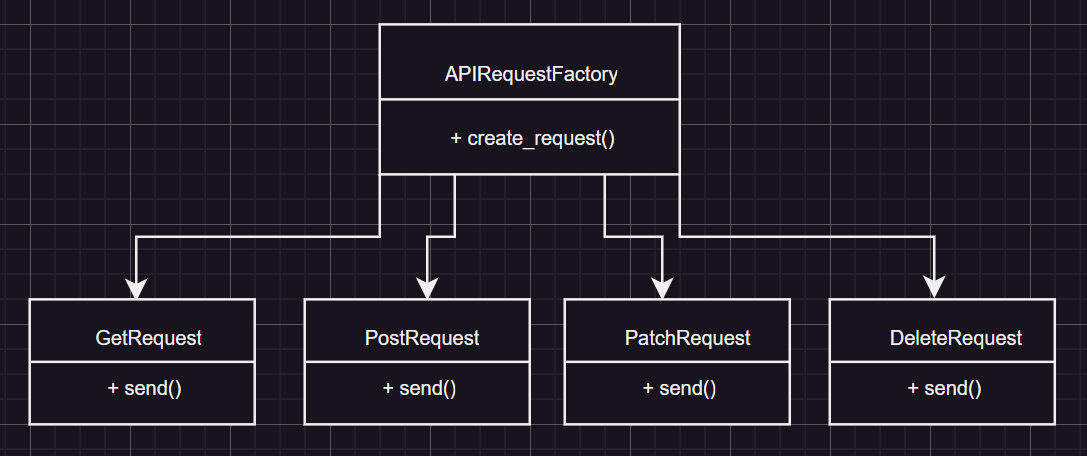


Figura 1.2 – Diagrama Factory

În diagramă:

* Clasa APIRequestFactory este reprezentată ca clasa din fabrică responsabilă pentru crearea diferitelor tipuri de solicitări.
* Metoda create\_request() este metoda din fabrică responsabilă pentru crearea și returnarea anumitor obiecte de solicitare.
* Clasele de cereri concrete (GetRequest, PostRequest, PatchRequest, DeleteRequest) sunt clasele de produs care implementează interfața comună (metoda send()) definită de fabrică.

Modelul Factory Method permite clasei APIRequestFactory să delege crearea diferitelor tipuri de obiecte de solicitare către clasele de solicitare concrete. Metoda din fabrică create\_request() este responsabilă pentru crearea obiectului de cerere adecvat pe baza parametrilor dați, cum ar fi metoda de solicitare. Această abordare permite flexibilitate și extensibilitate ușoară atunci când se introduc noi tipuri de solicitări în viitor.

## **2.2 Modele de proiectare structurală**

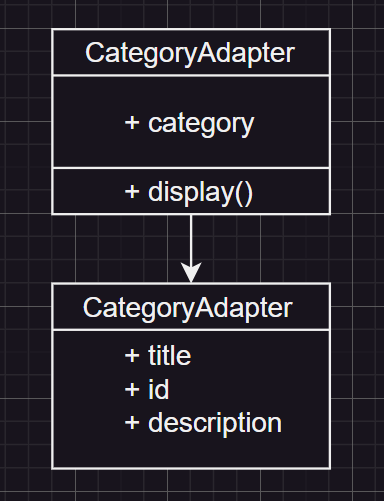


Figura 1.3 – Diagrama Adapter

În diagramă:

* Clasa CategoryAdapter reprezintă adaptorul. Adaptă obiectul Category la o interfață diferită.
* Atributul categorie din CategoryAdapter reprezintă obiectul Category împachetat.
* Metoda display() din CategoryAdapter adaptează comportamentul obiectului Category la interfața dorită.

Modelul Adapter permite clasei CategoryAdapter să adapteze obiectul Category existent la o interfață diferită. Clasa CategoryAdapter include obiectul Category și oferă un set diferit de metode sau comportament prin metoda display(). Acest lucru permite ca obiectul Category să fie utilizat în contexte care așteaptă interfața adaptată.

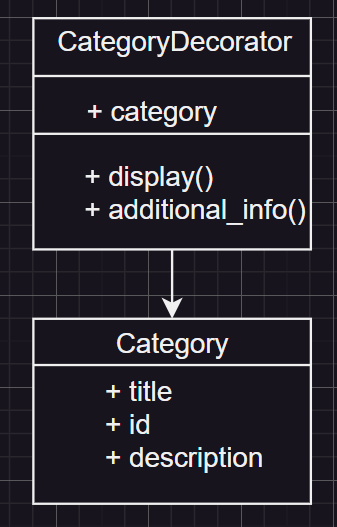


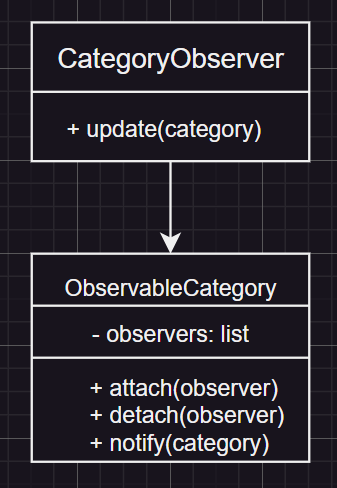
Figura 1.4 – Diagrama Adapter

În diagramă:

* Clasa CategoryDecorator reprezintă Decoratorul. Acesta include obiectul Category și adaugă un comportament suplimentar.
* Atributul categorie din CategoryDecorator reprezintă obiectul Category împachetat.
* Metoda display() din CategoryDecorator suprascrie comportamentul metodei display() a obiectului Category.
* Metoda additional\_info() din CategoryDecorator adaugă un comportament suplimentar metodei display().

Modelul Decorator permite clasei CategoryDecorator să adauge noi funcționalități sau să modifice comportamentul obiectului Category fără a-l modifica direct. Prin împachetarea obiectului Category, CategoryDecorator își poate îmbunătăți metoda display() prin adăugarea de informații suplimentare folosind metoda additional\_info().

## **2.3 Modele de design comportamental**



Figuta 1.5 – Diagrama Observer

În diagramă:

* Clasa CategoryObserver reprezintă Observatorul. Acesta definește metoda update() pentru a primi actualizări de la observabil.
* Clasa ObservableCategory reprezintă Observable. Menține o listă de observatori și oferă metode de gestionare a acestora (attach(), detach(), notify()).
* Alte clase (OtherClass, AnotherClass etc.) pot acționa ca observatori prin implementarea metodei update() și înregistrându-se cu observabilul.

Modelul Observer permite ObservableCategory să notifice observatorii săi înregistrați (cum ar fi OtherClass, AnotherClass etc.) când există actualizări. Observatorii implementează metoda update(), care este apelată de observabil pentru a furniza informațiile actualizate de categorie.

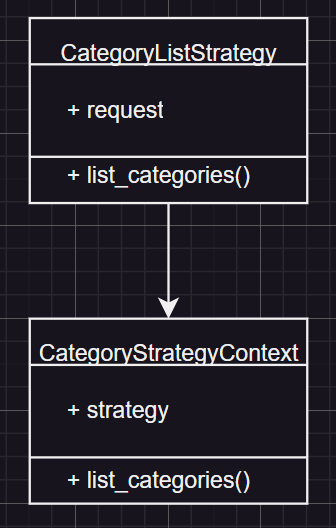


Figura 1.6 – Diagrama Strategy

În diagramă:

* Clasa CategoryListStrategy reprezintă Strategia. Acesta încapsulează algoritmul pentru listarea categoriilor pe baza unei cereri specifice.
* Atributul de solicitare din CategoryListStrategy reprezintă obiectul de solicitare utilizat pentru preluarea categoriilor.
* Metoda list\_categories() din CategoryListStrategy execută algoritmul pentru listarea categoriilor pe baza cererii date.
* Clasa CategoryStrategyContext reprezintă Contextul. Menține o referință la strategia curentă și oferă o metodă pentru a executa strategia (list\_categories()).

Modelul Strategy permite CategoryStrategyContext să delege algoritmul pentru listarea categoriilor către diferite strategii, cum ar fi CategoryListStrategy. CategoryListStrategy încapsulează algoritmul specific pentru listarea categoriilor pe baza solicitării furnizate. CategoryStrategyContext oferă o modalitate de a seta strategia curentă și de a o executa, făcând abstracție de client din detaliile specifice implementării strategiei.

# **Concluzie :**

Aplicația de management al magazinului este un instrument valoros pentru organizarea și gestionarea eficientă a produselor din cadrul unui magazin online. Cu API-ul ușor de utilizat și funcționalitatea cuprinzătoare, aplicația simplifică procesul de creare, actualizare și ștergere a categoriilor și a produselor. Oferă o interfață convenabilă pentru administratori pentru a menține un inventar precis și bine structurat.

Utilizând funcțiile aplicației, proprietarii de magazine pot adăuga cu ușurință noi categorii, pot atribui produse unor categorii specifice, pot actualiza detaliile despre produse și pot elimina articolele învechite sau întrerupte. Designul API RESTful al aplicației asigură compatibilitatea cu diverse interfețe front-end, aplicații mobile și sisteme externe, permițând integrarea perfectă în ecosistemul magazinului.

Aplicația de gestionare a magazinului îmbunătățește productivitatea prin eficientizarea sarcinilor de gestionare a stocurilor, reducerea efortului manual și furnizarea unui sistem centralizat pentru gestionarea operațiunilor legate de produs. Îmbunătățește eficiența generală, facilitând administratorilor să mențină un catalog actualizat și bine organizat.

În concluzie, aplicația de management al magazinului oferă o soluție fiabilă și eficientă pentru gestionarea eficientă a categoriilor și produselor dintr-un magazin online. Cu funcționalitatea sa robustă, API-ul ușor de utilizat și suportul pentru operațiuni esențiale, aplicația dă putere proprietarilor de magazine să mențină o experiență de cumpărături organizată și perfectă pentru clienții lor.

# **Bibliografie :**

<https://app.diagrams.net/>

<https://refactoring.guru/design-patterns>

<https://www.tutorialspoint.com/design_pattern/design_pattern_overview.htm>

<https://www.freecodecamp.org/news/the-basic-design-patterns-all-developers-need-to-know/>