Design Patterns Documentatie

Contents

[1. Factory Method Pattern 2](#_Toc199502849)

[2. Singleton Pattern 2](#_Toc199502850)

[3. Prototype Pattern 3](#_Toc199502851)

[4. Bridge Pattern 3](#_Toc199502852)

[5. Composite Pattern 4](#_Toc199502853)

[6. Strategy Pattern 5](#_Toc199502854)

[7. Observer Pattern 6](#_Toc199502855)

[8. Chain of Responsibility 6](#_Toc199502856)

[9. Command Pattern 7](#_Toc199502857)

[10. Thread Pool Pattern 7](#_Toc199502858)

[11. Producer-Consumer Pattern 8](#_Toc199502859)

|  |  |  |
| --- | --- | --- |
| Pattern soort | Pattern | Implementatie locatie |
| Creational | Factory | NoteFactory |
| Creational | Singleton | NoteManager |
| Creational | Prototype | Note.Clone |
| Structural | Bridge | Abstractnote, NoteStorageStrategy |
| Structural | Composite | INoteComponent,Notefolder |
| Behavioral | Strategy | NoteStorageStategy |
| Behavioral | Observer | INoteObserver |
| Behavioral | Chain of resposibility | INoteHandler |
| Behavioral | Command | NoteQueueProcessor |
| Concurrenccy | Tread pool | NoteQueueProcessor |
| Concurrenccy | Producer-consumer | NoteQueueProcessor |

A black and white diagram

AI-generated content may be incorrect.

# 1. Factory Method Pattern

Functie: Centraliseert objectcreatie met standaardwaarden

Waarom geschikt:

Vermijdt `new` operaties verspreid door de code

Garandeert consistente initialisatie

Vereenvoudigt testen via mockable factory

Codevoorbeeld:

|  |
| --- |
| public static class NoteFactory  {  public static Note Create(string title, string content)   {  return new Note   {  Title = title ?? "Nieuwe notitie",  Content = content ?? string.Empty,  Created = DateTime.Now,  LastModified = DateTime.Now  };  }   public static Note CreateEmpty()   {  return Create("Nieuwe notitie", "Typ hier...");  } } |

# 2. Singleton Pattern

Functie: Garandeert globale toegang tot NoteManager

Waarom geschikt:

Thread-safe instantiatie via `Lazy<T>`

Centraliseert business logic

Eenvoudige dependency management

Codevoorbeeld:

|  |
| --- |
| public sealed class NoteManager {  private static readonly Lazy<NoteManager> \_instance =   new Lazy<NoteManager>(() => new NoteManager());    public static NoteManager Instance => \_instance.Value;    private NoteManager()   {  \_storage = new SQLSizeNotes();  }    private readonly NoteStorageStrategy \_storage; } |

# 3. Prototype Pattern

Functie: clonefunctionaliteit voor kopieën

Waarom geschikt:

Vermijdt complexe re-initiatie

Behoudt referentiële integriteit

Ondersteunt undo/redo

Codevoorbeeld:

|  |
| --- |
| public class Note : ICloneable {  public List<Attachment> Attachments { get; set; }    public object Clone()  {  var clone = (Note)this.MemberwiseClone();  clone.Attachments = new List<Attachment>(this.Attachments);  clone.Created = DateTime.Now;  return clone;  } } |

# 4. Bridge Pattern

Functie: Ontkoppelt abstractie van implementatie

Waarom geschikt:

Wijzigingen aan opslag beïnvloeden niet de note-logica

Runtime switching van strategie mogelijk

Eenvoudig uit te breiden met nieuwe opslagtypes

Codevoorbeeld:

|  |
| --- |
| public abstract class AbstractNote {  protected readonly NoteStorageStrategy \_storage;    protected AbstractNote(NoteStorageStrategy storage)  {  \_storage = storage;  }    public abstract void Save(); }  public class TextNote : AbstractNote {  public override void Save()   {  \_storage.Save(new NoteDto(Title, Content));  } } |

# 5. Composite Pattern

Functie: behandeling van hiërarchische structuren

Waarom geschikt:

Recursieve compositie

Enkelvoudige en meervoudige objecten hetzelfde interface

Eenvoudig uit te breiden met nieuwe componenttypes

Codevoorbeeld:

|  |
| --- |
| public interface INoteComponent {  string Name { get; }  void Display(StringBuilder indentBuilder); }  public class NoteFolder : INoteComponent {  private readonly List<INoteComponent> \_children = new();    public void Add(INoteComponent component) => \_children.Add(component);    public void Display(StringBuilder indentBuilder)  {  Console.WriteLine($"{indentBuilder}+ {Name}");    var childIndent = new StringBuilder(indentBuilder.ToString())  .Append(" ");    foreach (var child in \_children)  {  child.Display(childIndent);  }  } } |

# 6. Strategy Pattern

Functie: Verwisselbare algoritmes voor opslag

Waarom geschikt:

Open/Closed principe

Eenvoudig testbaar

Runtime strategy switching

Codevoorbeeld:

|  |
| --- |
| public interface INoteStorageStrategy {  void Save(Note note);  Note Load(int id); }  public class CloudStorageStrategy : INoteStorageStrategy {  public void Save(Note note)  {  Console.WriteLine($"Uploading {note.Title} to cloud...");  } } |

# 7. Observer Pattern

Functie: Push-based state synchronisatie

Waarom geschikt:

Losse koppeling tussen subject en observers

Automatische propagatie van wijzigingen

Meerdere views opzelfde data

Codevoorbeeld:

A diagram of a company

AI-generated content may be incorrect.

# 8. Chain of Responsibility

Functie: Pipeline voor request verwerking

Waarom geschikt:

Dynamische handler ketens

Enkele verantwoordelijkheid per handler

Flexibele volgorde aanpassingen

Codevoorbeeld:

|  |
| --- |
| public class EncryptionHandler : AbstractNoteHandler {  public override Note Handle(Note note)  {  note.Content = Encrypt(note.Content);  return base.Handle(note);  }    private string Encrypt(string content)   => Convert.ToBase64String(Encoding.UTF8.GetBytes(content)); } |

# 9. Command Pattern

Functie: Encapsuleer operaties als objecten

Waarom geschikt:

Undo/redo ondersteuning

Uitstelbare executie

Transactioneel gedrag

Codevoorbeeld:

|  |
| --- |
| public interface INoteCommand {  void Execute();  void Undo(); }  public class DeleteNoteCommand : INoteCommand {  private readonly Note \_note;  private readonly NoteManager \_manager;    public void Execute() => \_manager.Delete(\_note);  public void Undo() => \_manager.Restore(\_note); } |

# 10. Thread Pool Pattern

Functie: Efficiënt thread hergebruik

Waarom geschikt:

Vermijdt thread creation overhead

Dynamic scaling

Work item queue

Codevoorbeeld:

|  |
| --- |
| ThreadPool.QueueUserWorkItem(state =>  {  var note = (Note)state;  NoteManager.Instance.Process(note); }, newNote); |

# 11. Producer-Consumer Pattern

Functie: Gecoördineerde concurrente verwerking

Waarom geschikt:

Buffered communicatie

Thread-safe queue

Rate limiting

Codevoorbeeld:

|  |
| --- |
| public class NoteProcessor {  private readonly BlockingCollection<Note> \_queue = new();    public void AddNote(Note note) => \_queue.Add(note);    private void ProcessQueue()  {  foreach (var note in \_queue.GetConsumingEnumerable())  {  try   {  \_manager.Save(note);  }  catch (Exception ex)  {  \_retryQueue.Add(note);  }  }  } } |