Design Patterns en hun Toepassing in het Temperatuur Monitoring Systeem

Inhoudsopgave

Inleiding

Het Temperatuur Monitoring Systeem is ontworpen om sensorgegevens van verschillende temperatuur- en vochtigheidssensoren te verwerken, analyseren en visualiseren. Het systeem leest sensorgegevens van tekstbestanden (die realtime feeds van een Raspberry Pi Pico simuleren), verwerkt de gegevens via verschillende componenten en biedt analyses, visualisaties en waarschuwingen op basis van sensormetingen.

Om een flexibele, onderhoudbare en schaalbare architectuur te creëren, zijn meerdere design patterns geïmplementeerd. Deze documentatie beschrijft de gekozen design patterns en waarom ze geschikt zijn voor deze specifieke toepassing.

Creational Patterns

Singleton Pattern

Beschrijving:
Het Singleton pattern zorgt ervoor dat een klasse slechts één instantie heeft en biedt een globaal toegangspunt tot deze instantie.

Implementatie:
In ons systeem is `SensorTypeManager` geïmplementeerd als Singleton om een centrale registratie van sensortypen bij te houden.

Waarom geschikt:

- **Centrale registratie:** We hebben één consistente bron van sensortype-informatie nodig in het hele systeem.
- **Vermijden van duplicatie:** Zonder Singleton zou elke component zijn eigen lijst van sensortypen moeten bijhouden, wat leidt tot inconsistenties.
- **Globale toegang:** Verschillende componenten moeten de sensortype-informatie kunnen raadplegen en bijwerken.
- **Thread-safety:** De implementatie met locking mechanismen zorgt voor veilig gebruik in een multithreaded omgeving.

```
public class SensorTypeManager
{
    private static SensorTypeManager _instance;
    private static readonly object _lock = new object();
    private readonly Dictionary<string, string> _sensorTypes = new
Dictionary<string, string>();

    private SensorTypeManager() { }

    public static SensorTypeManager Instance
    {
        get
        {
            lock (_lock)
            {
                if (_instance == null)
                {
                    _instance = new SensorTypeManager();
                }
                return _instance;
            }
        }
    }

    // Methoden om sensortypen te registreren en op te vragen
}
```

Factory Method Pattern

Beschrijving:
Het Factory Method pattern definieert een interface voor het creëren van objecten, maar laat subklassen beslissen welke klassen geïnstantieerd worden.

Implementatie:
We gebruiken het Factory Method pattern om verschillende typen sensordata-analyzers te creëren.

Waarom geschikt:

- **Flexibele objectcreatie:** We hebben verschillende analyzers nodig voor temperatuur, vochtigheid en batterij met specifieke configuraties.
- **Encapsulatie van creatie-logica:** De complexiteit van het instantiëren van analyzers wordt verborgen achter de factory interface.
- **Uitbreidbaarheid:** Nieuwe typen analyzers kunnen worden toegevoegd zonder bestaande code te wijzigen.
- **Configuratie:** Factories kunnen parameters accepteren om analyzers met verschillende drempelwaarden te configureren.

```
public abstract class AnalyzerFactory
{
    public abstract ISensorDataAnalyzer CreateAnalyzer();
}

public class TemperatureAnalyzerFactory : AnalyzerFactory
{
    private readonly double _warningThreshold;
    private readonly double _criticalThreshold;

    public TemperatureAnalyzerFactory(double warningThreshold = 25.0, double
criticalThreshold = 30.0)
    {
        _warningThreshold = warningThreshold;
        _criticalThreshold = criticalThreshold;
    }

    public override ISensorDataAnalyzer CreateAnalyzer()
    {
        return new TemperatureAnalyzer(_warningThreshold, _criticalThreshold);
    }
}
```

Structural Patterns

Composite Pattern

Beschrijving:
Het Composite pattern stelt je in staat om objecten in boomstructuren samen te stellen en individuele objecten en composities uniform te behandelen.

Implementatie:
In ons systeem worden sensoren georganiseerd in een hiërarchische structuur met `SensorLeaf` (individuele sensoren) en `SensorGroup` (groepen van sensoren).

Waarom geschikt:

- **Hiërarchische organisatie:** Sensoren kunnen logisch worden gegroepeerd op basis van type, fabrikant of locatie.

- **Uniforme behandeling:** Code kan zowel individuele sensoren als groepen van sensoren op dezelfde manier behandelen.
- **Geaggregeerde statistieken:** Statistieken kunnen op elk niveau van de hiërarchie worden berekend.
- **Flexibele structuur:** De structuur kan dynamisch worden aangepast zonder client code te wijzigen.

```
public interface ISensorComponent
{
    string Name { get; }
    string Type { get; }
    void AddData(SensorData data);
    Dictionary<string, double> GetAggregatedData();
    void DisplayInfo(int depth = 0);
    int GetSensorCount();
}

public class SensorLeaf : ISensorComponent { /* ... */ }
public class SensorGroup : ISensorComponent { /* ... */ }
```

Facade Pattern

Beschrijving:
Het Facade pattern biedt een vereenvoudigde interface voor een complex subsysteem.

Implementatie:
`SensorSystemFacade` verbergt de complexiteit van verschillende subsystemen (parsers, observers, analyzers, etc.).

Waarom geschikt:

- **Complexiteitsbeheersing:** Het temperatuurmonitoring-systeem bevat veel samenhangende componenten.
- **Ontkoppeling:** Client code hoeft niet direct te interacteren met subsystemen.
- **Gebruiksgemak:** Eenvoudige interface voor het verwerken van sensorgegevens.
- **Integratie:** De facade integreert alle andere patterns in een coherent systeem.

```
public class SensorSystemFacade : IDisposable
{
    private readonly SensorDataProcessor _processor;
    private readonly SensorDataSubject _subject;
    private readonly SensorTypeManager _typeManager;
    private readonly List<ISensorDataParser> _parsers;
    private readonly AnalyzerManager _analyzerManager;
    private readonly SensorCompositeManager _compositeManager;
    private readonly ActorSystemManager _actorSystem;
    private readonly DisplayManager _displayManager;

    public SensorSystemFacade()
    {
        // Initialisatie van alle subsystemen
```

```
    }

    public void ProcessSensorData(string rawData)
    {
        // Verwerk sensorgegevens via alle subsystemen
    }

    // Andere methoden
}
```

Bridge Pattern

Beschrijving:
Het Bridge pattern scheidt een abstractie van zijn implementatie, zodat beide onafhankelijk kunnen variëren.

Implementatie:
In ons systeem wordt het Bridge pattern gebruikt om de formattering van gegevens (implementatie) te scheiden van de weergavemechanismen (abstractie).

Waarom geschikt:

- **Formattering vs. weergave:** We hebben verschillende manieren nodig om gegevens te formatteren (tekst, JSON) en weer te geven (console, bestand).
- **Combinaties vermijden:** Zonder Bridge zouden we aparte klassen nodig hebben voor elke combinatie (TextConsoleDisplay, JsonFileDisplay, etc.).
- **Onafhankelijke variatie:** Nieuwe formatters of displays kunnen worden toegevoegd zonder bestaande code te wijzigen.
- **Samenhang:** Gerelateerde weergave-functionaliteit blijft gegroepeerd in weergaveklassen.

```
// Implementatie interface
public interface ISensorDataFormatter
{
    string FormatSensorData(SensorData data);
    string FormatSensorList(IEnumerable<SensorData> dataList);
    string FormatStatistics(Dictionary<string, double> statistics);
    string FormatAlert(string alertMessage, string severity);
}

// Abstractie interface
public interface ISensorDataDisplay
{
    void DisplaySensorData(SensorData data);
    void DisplaySensorList(IEnumerable<SensorData> dataList);
    void DisplayStatistics(Dictionary<string, double> statistics, string title);
    void DisplayAlert(string alertMessage, string severity);
}

// Concrete formatter: TextFormatter, JsonFormatter
// Concrete display: ConsoleDisplay, FileDisplay
```

Behavioral Patterns

Strategy Pattern

Beschrijving:
Het Strategy pattern definieert een familie van algoritmen, kapselteert ze elk in en maakt ze uitwisselbaar.

Implementatie:
We gebruiken het Strategy pattern voor verschillende algoritmen om sensorgegevens te parsen, met implementaties voor verschillende dataformats.

Waarom geschikt:

- **Verschillende dataformaten:** Sensorgegevens komen in verschillende formaten, elk met eigen parseermethoden.
- **Runtime selectie:** Het juiste parsealgoritme kan worden geselecteerd op basis van het formaat van de ontvangen gegevens.
- **Vermijden van conditional logica:** Complexe if/else of switch statements worden vermeden.
- **Uitbreidbaarheid:** Nieuwe parsestrategieën kunnen eenvoudig worden toegevoegd voor nieuwe dataformaten.

```
public interface ISensorDataParser
{
    bool CanParse(string rawData);
    SensorData Parse(string rawData);
}

public class StandardFormatParser : ISensorDataParser { /* ... */ }
public class ManufacturerFirstFormatParser : ISensorDataParser { /* ... */ }

public class SensorDataProcessor
{
    private ISensorDataParser _parser;

    public void SetParser(ISensorDataParser parser)
    {
        _parser = parser;
    }

    public SensorData ProcessData(string rawData)
    {
        return _parser.Parse(rawData);
    }
}
```

Observer Pattern

Beschrijving:
Het Observer pattern definieert een één-op-veel-afhankelijkheid tussen objecten, zodat wanneer één object verandert, alle afhankelijke objecten automatisch worden geïnformeerd.

Implementatie:
We gebruiken het Observer pattern om verschillende componenten te notificeren wanneer nieuwe sensorgegevens binnenkomen.

Waarom geschikt:

- **Real-time monitoring:** Meerdere componenten moeten reageren op nieuwe sensorgegevens.
- **Losse koppeling:** De databron hoeft niet te weten welke componenten de data gebruiken.
- **Flexibele notificaties:** Observers kunnen dynamisch worden toegevoegd of verwijderd.
- **Gespecialiseerde verwerking:** Elke observer kan de data op zijn eigen manier verwerken.

```
public interface ISensorDataObserver
{
    void Update(SensorData data);
}

public class SensorDataSubject
{
    private readonly List<ISensorDataObserver> _observers = new
List<ISensorDataObserver>();

    public void Attach(ISensorDataObserver observer) { /* ... */ }
    public void Detach(ISensorDataObserver observer) { /* ... */ }
    public void Notify(SensorData data) { /* ... */ }
}

public class TemperatureMonitor : ISensorDataObserver { /* ... */ }
public class BatteryMonitor : ISensorDataObserver { /* ... */ }
```

Visitor Pattern

Beschrijving:
Het Visitor pattern scheidt een algoritme van de objectstructuur waarop het werkt, zodat je nieuwe operaties kunt toevoegen zonder de klassen van de objecten te wijzigen.

Implementatie:
We gebruiken het Visitor pattern om operaties (zoals gezondheidscontroles, anomaliedetectie) uit te voeren op de sensor-hiërarchie.

Waarom geschikt:

- **Nieuwe operaties:** We kunnen nieuwe diagnostische operaties toevoegen zonder de sensorklassen te wijzigen.
- **Statusverzameling:** Bezoekers kunnen statusgegevens verzamelen terwijl ze door de hiërarchie navigeren.
- **Dubbele verzending:** Verschillende gedrag voor individuele sensoren versus groepen.
- **Rapportage:** Statistieken en waarschuwingen kunnen worden gegenereerd voor de gehele sensorstructuur.

```
public interface ISensorVisitor
{
    void Visit(SensorLeaf sensor);
    void Visit(SensorGroup group);
    void Reset();
    string GetResult();
}

public class SensorHealthVisitor : ISensorVisitor { /* ... */ }
public class AnomalyDetectionVisitor : ISensorVisitor { /* ... */ }

// Toevoeging aan ISensorComponent interface
public interface ISensorComponent
{
    // Bestaande methoden
    void Accept(ISensorVisitor visitor);
}
```

Concurrency Patterns

Thread Pool Pattern

Beschrijving:
Het Thread Pool pattern beheert een verzameling worker threads die kunnen worden hergebruikt voor het uitvoeren van taken.

Implementatie:
`ThreadPoolManager` beheert een pool van threads voor het parallel verwerken van sensorgegevens.

Waarom geschikt:

- **Efficiënte verwerking:** Bij veel sensoren is parallelle verwerking noodzakelijk voor goede prestaties.
- **Beperking van resources:** Het thread pool voorkomt de creatie van te veel threads.
- **Beheerste parallellisme:** De mate van parallellisme kan worden beperkt om systeemoverbelasting te voorkomen.
- **Taakplanning:** Taken kunnen worden gepland en geprioriteerd voor optimale doorvoer.

```
public class ThreadPoolManager
{
    private static readonly Lazy<ThreadPoolManager> _instance =
        new Lazy<ThreadPoolManager>(() => new ThreadPoolManager());

    public static ThreadPoolManager Instance => _instance.Value;

    private readonly int _maxDegreeOfParallelism;
    private readonly SemaphoreSlim _semaphore;

    // Methoden voor het plannen en uitvoeren van taken
    public async Task<TResult> QueueTaskAsync<TResult>(Func<TResult> function) {
/* ... */ }
```

```
    public async Task QueueTaskAsync(Action action) { /* ... */ }
    public async Task ProcessBatchAsync<T>(IEnumerable<T> items, Func<T, Task>
processFunction) { /* ... */ }
}
```

Producer-Consumer Pattern

Beschrijving:
Het Producer-Consumer pattern scheidt de productie van data van de consumptie ervan, waardoor ze met verschillende snelheden kunnen werken.

Implementatie:
`SensorDataQueue` implementeert een buffer tussen de productie van sensorgegevens en de verwerking ervan.

Waarom geschikt:

- **Variabele snelheden:** Sensorgegevens kunnen sneller binnenkomen dan ze kunnen worden verwerkt.
- **Ontlasten van producent:** De producent hoeft niet te wachten tot de verwerking is voltooid.
- **Bufferwerking:** Pieken in dataproductie kunnen worden opgevangen.
- **Asynchrone verwerking:** Dataproductie en -consumptie kunnen onafhankelijk van elkaar werken.

```
public class SensorDataQueue
{
    private readonly BlockingCollection<string> _dataQueue;
    private CancellationTokenSource _cancellationTokenSource;
    private Task _consumerTask;

    // Methoden voor starten/stoppen en produceren/consumeren
    public void Start(Action<string> processAction) { /* ... */ }
    public void Stop() { /* ... */ }
    public void Produce(string rawData) { /* ... */ }
    private void ConsumeData(Action<string> processAction, CancellationToken
token) { /* ... */ }
}
```

Actor Model Pattern

Beschrijving:
Het Actor Model pattern behandelt actoren als fundamentele eenheden van berekening die communiceren via berichtgeving.

Implementatie:
We gebruiken actoren voor de verwerking van sensorgegevens en alarmen, waarbij elke actor zijn eigen staat beheert.

Waarom geschikt:

- **Isolatie:** Actoren hebben geen gedeelde staat, waardoor race conditions worden verminderd.

- **Asynchrone berichtgeving:** Natuurlijk model voor asynchrone communicatie tussen componenten.
- **Fouttolerantie:** Actoren kunnen fouten isoleren en herstellen.
- **Schaalbaarheid:** Het model schaalt natuurlijk over meerdere cores of zelfs machines.

```
public class SensorDataActor : ReceiveActor
{
    private readonly Dictionary<string, List<SensorData>> _sensorDataStore =
        new Dictionary<string, List<SensorData>>();

    public SensorDataActor()
    {
        // Definitie van berichthandlers
        Receive<SensorDataMessage>(message => {
            ProcessSensorData(message.Data);
        });

        Receive<AnalyzeDataMessage>(message => {
            var result = AnalyzeSensorData(message.SensorType);
            Sender.Tell(new DataAnalysisResult(message.SensorType, result));
        });
    }

    // Verwerkingsmethoden
}
```

Integratie van Patterns

Een belangrijke sterkte van deze implementatie is hoe de verschillende design patterns samenwerken:

1. **Data Inname & Parsing (Strategy):** Gegevens worden geparseerd met behulp van de juiste strategie, geselecteerd op basis van het formaat.
2. **Centrale Registratie (Singleton):** Geparseerde gegevens worden geregistreerd in de centrale SensorTypeManager.
3. **Hiërarchische Organisatie (Composite):** Sensorgegevens worden toegevoegd aan de juiste componenten in de sensorhiërarchie.
4. **Diagnostische Operaties (Visitor):** Visitors kunnen de hiërarchie doorlopen voor gezondheidscontroles en anomaliedetectie.
5. **Parallelle Verwerking (Thread Pool, Actor Model):** Gegevens worden parallel verwerkt voor efficiëntie.
6. **Continue Verwerking (Producer-Consumer):** De datatoevoer wordt gedecoupled van de verwerking voor stabiele performance.
7. **Flexibele Weergave (Bridge):** De resultaten worden geformatteerd en weergegeven via verschillende displays.
8. **Eenvoudige API (Facade):** Client code interacteert met alle subsystemen via een eenvoudige facade.

Deze integratie zorgt voor een systeem dat flexibel, onderhoudbaar en schaalbaar is.