



AARHUS UNIVERSITET

Softwaredesign Eksamensforberedelse

| Deltagere | | |
|-----------------------|--------------------|--------|
| Navn | Tidspunkt | Lokale |
| Nickolai Lundby Ernst | 15. Juni kl. 13:00 | 423E |
| Rune Aagaard Keena | 15. Juni kl. 9:20 | 424E |

Indholdsfortegnelse

| | | |
|-----------|--|-----------|
| 1 | Strategy and Template Method Pattern | 4 |
| 1.1 | Template Method | 4 |
| 1.2 | Strategy Pattern | 5 |
| 1.3 | Comparison | 5 |
| 2 | Observer Pattern | 7 |
| 2.1 | The pattern | 8 |
| 2.2 | Push-Pull variants | 9 |
| 2.3 | Different types of subjects | 9 |
| 3 | Factory Patterns | 11 |
| 3.1 | Factory Method[2] | 11 |
| 3.2 | Abstract Factory[1] | 12 |
| 4 | State Machine Patterns | 14 |
| 4.1 | Nested States | 15 |
| 4.2 | Orthogonal States | 16 |
| 5 | GUI Design Pattern | 17 |
| 5.1 | MVC Pattern | 17 |
| 5.2 | MVP Pattern | 18 |
| 5.2.1 | Supervising Controller | 19 |
| 5.2.2 | Passive View | 20 |
| 5.3 | MVVM Pattern | 20 |
| 5.4 | N-Layer Architecture | 23 |
| 6 | Parallel Aggregation and MapReduce | 25 |
| 6.1 | Parallel Aggregation | 25 |
| 6.2 | MapReduce | 27 |
| 7 | Futures and Pipelines | 29 |
| 7.1 | Dependencies | 29 |
| 7.2 | Futures | 31 |
| 7.3 | Pipelines | 32 |
| 8 | Software Architecture | 35 |
| 9 | Mandatory Exercise: Chain of Responsibility | 38 |
| 9.1 | Implementation | 38 |
| 9.2 | Comparison | 39 |
| 9.3 | Conclusion | 39 |
| 10 | SOLID | 40 |

Information om eksamen i Software design (I4SWD)

1: Indledning og formalia

Denne note beskriver rammerne for og spørgsmålene til eksamen i faget Software Design (I4SWD). Eksamen i faget er en mundtlig eksamen af 20 minutters varighed med ekstern censur, og med på forhånd kendte spørgsmål. Præstationen vurderes efter 7-trinsskalaen.

2: Eksamensgennemførelsen

Der findes ikke to ens eksaminationer, så de vil variere lidt fra eksaminand til eksaminand. Eksamen afvikles imidlertid typisk som følger:

- 1. Planlæg at præsentere emnet i cirka (10 minutter)
- 2. Uddybende spørgsmål (5 minutter)
- 3. Formalia (træk spørgsmål, votering etc.) (5 minutter)

Under 1. ovenfor forventes den studerende på eget initiativ at kunne demonstrere sin dybdeforståelse af pensum indenfor det trukne spørgsmål (se afsnit 3).

Under 2. ovenfor vil eksaminator typisk stille spørgsmål til præsentationen og pensummet derudover.

3 Eksamensspørgsmål

Eksaminanden vil ved eksamens start trække netop ét af følgende spørgsmål:

- 1. Strategy Pattern + Template Method Pattern
- 2. Observer Pattern
- 3. Factory Patterns
- 4. State Machine Patterns
- 5. GUI Design Patterns
- 6. Parallel Aggregation + MapReduce
- 7. Futures + Pipelines
- 8. Software Architecture
- 9. Mandatory Exercise

1 Strategy and Template Method Pattern

The two patterns, GoF Strategy and GoF Template Method, are related because both are patterns for 'externalizing' their behavior, or parts of it, into other classes. They differ in that Template Method uses inheritance where Strategy uses delegation. Each pattern has its strengths and weaknesses, and whether to use one over the other depends on the situation. Template Method permits a superclass to decide the structure of a method call and thereby in detail controlling the behavior. Strategy permits to change an object's behavior at run-time.

1.1 Template Method

Type: Behavioral.

Intent: Define the skeleton of an algorithm in an operation, deferring some steps to client subclasses.

This pattern has two main parts:

- The "template method", generally implemented as a base class (possibly abstraction), which contains shared code and parts of the overall algorithm which are invariant. The template ensures that the overarching algorithm is always followed. In this class, "variant" portions are given a default implementation, or none at all.
- Concrete implementations of the abstract class, which fill in the empty or "variant" parts of the template with specific algorithms that vary from implementation to implementation.

At run time a concrete class is instantiated. A main method inherited from the base class is called, which then may call other methods defined by both the base class and the subclasses. This performs the overall algorithm in the same steps every time, but the details of some of the steps depend on which subclasses were instantiated. This pattern is an example of **inversion of control** because the high-level code no longer determines what algorithms run; a lower-level algorithm is instead selected at run-time.

| | |
|--|--|
| <pre>class TurnBasedGame { private List<Player> _players; // Template Method - defines "structure" public void PlayGame() { int i=0; InitGame(); while(!GameOver()) { TakeTurn(_players[i]); i = (i+1) % _players.Count; } AnnounceWinner(); } // Methods to be implemented by subclasses protected abstract void InitGame(); protected abstract bool GameOver(); protected abstract void TakeTurn(Player p); protected abstract void AnnounceWinner() } }</pre> | <pre>class Chess: TurnBasedGame { protected override void InitGame() { // Set up chess pieces } protected abstract bool GameOver() { // Check for check-mate } protected abstract void TakeTurn(Player p) { // Move one of p's pieces IAW chess rules } protected abstract void AnnounceWinner() { // Announce the winner of the game } }</pre> |
|--|--|

Figure 1: Example of the template method pattern, a turn-based game. The template here determines the order in which the "variant" parts are called.

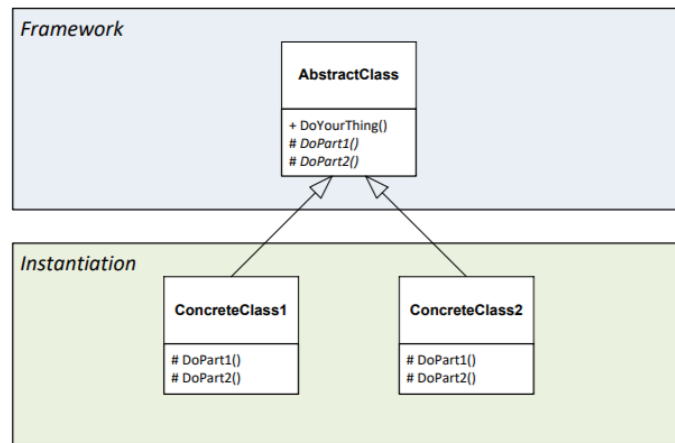


Figure 2: Class diagram for template method pattern.

1.2 Strategy Pattern

Type: Behavioral.

Intent: Define a family of algorithms, encapsulate each one, and make them interchangeable at run-time.

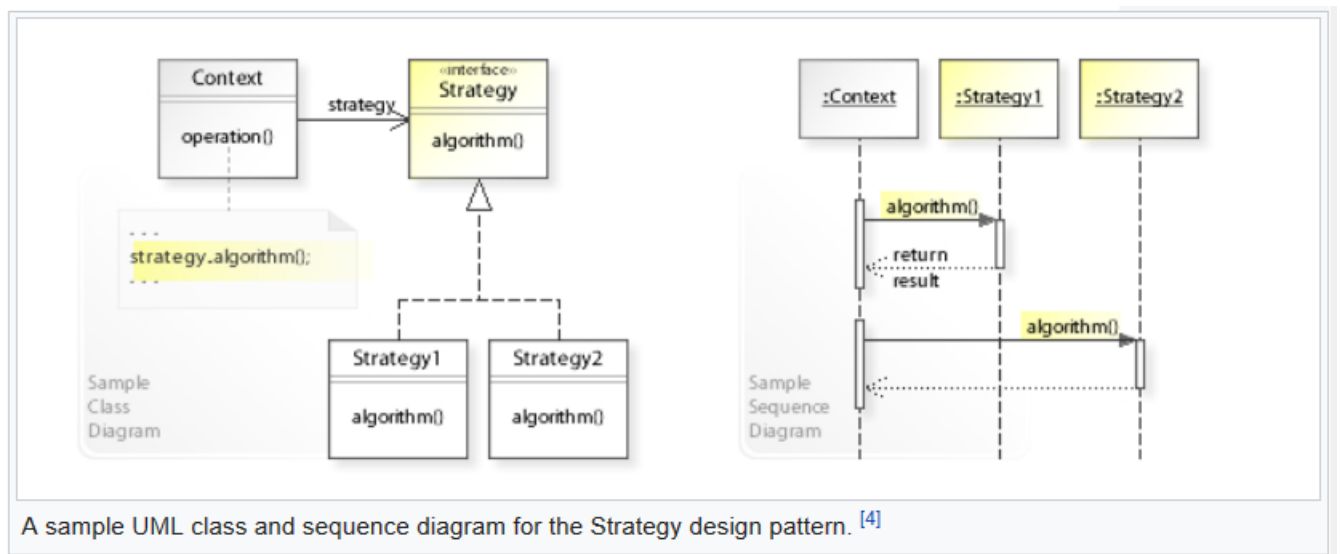


Figure 3: The Context class does not implement an algorithm directly. Instead Context refers to the Strategy interface for performing an algorithm, which makes Context independent of how an algorithm is implemented. The Strategy1 and Strategy2 classes implement the Strategy interface, that is, implement (encapsulate) an algorithm.

1.3 Comparison

Similarities: Both patterns are behavioral. Both are used to make the behavior of a system extensible.

Differences: Template Method uses inheritance: callback implementations, frameworks, behavior fixed at compile-time. Strategy uses delegation. Behavior can be changed at run-time.

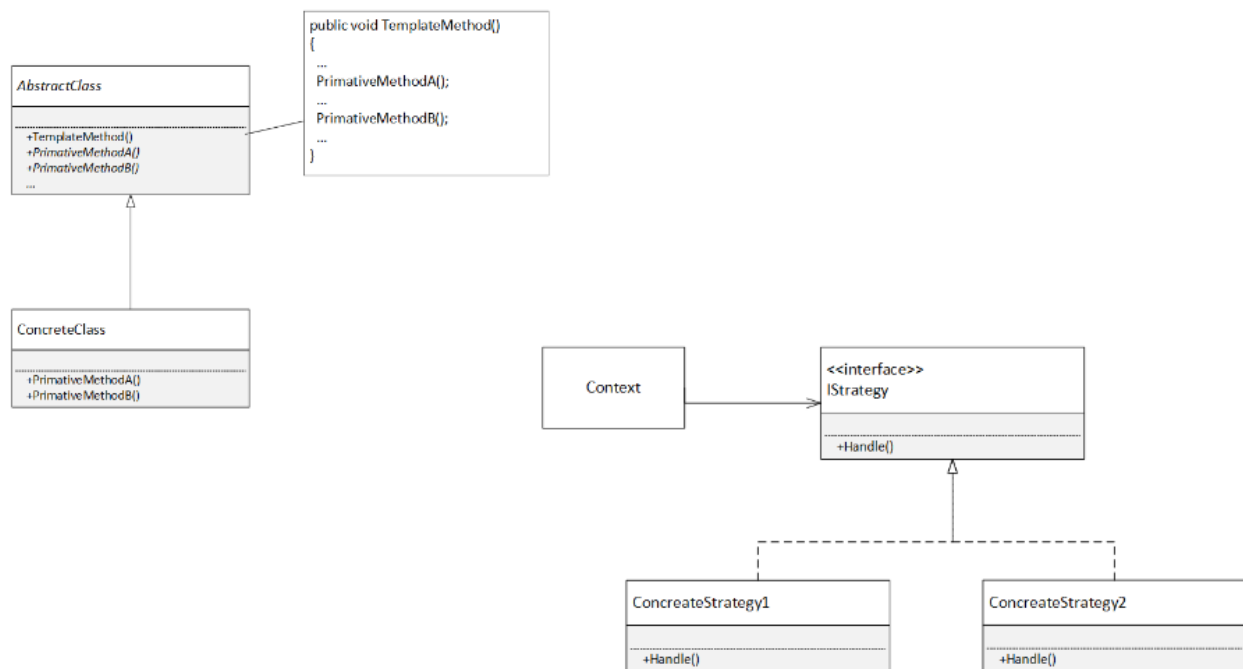
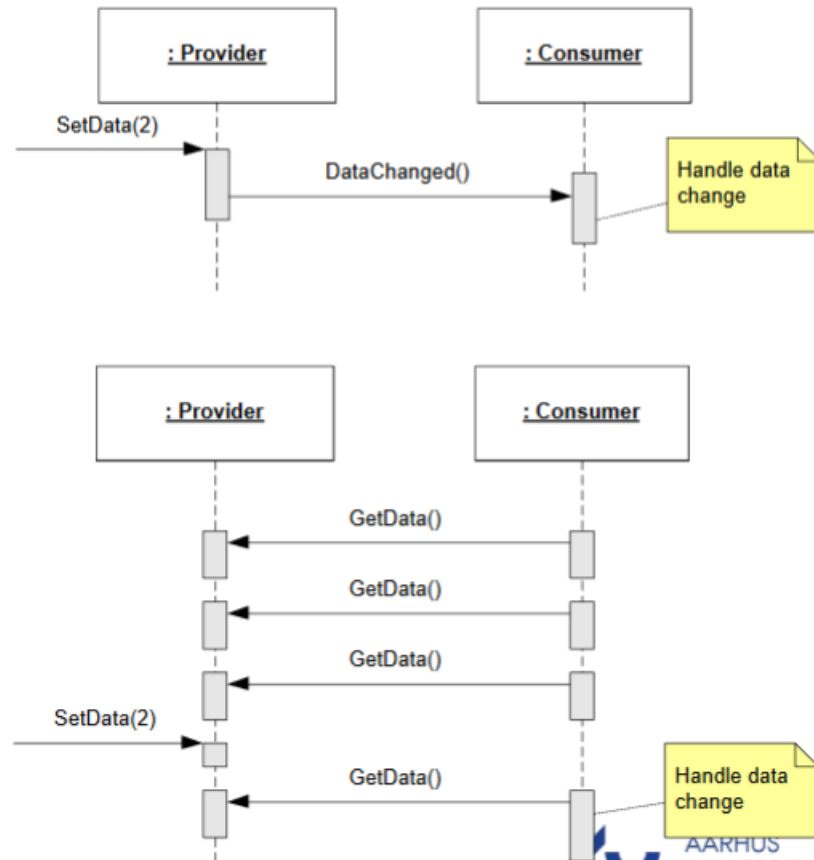


Figure 4: Comparison of inheritance(left) vs delegation(right)

2 Observer Pattern

Type: Behavioral.

The problem of updating data without the GoF Observer pattern, is too high coupling. Take figure 5 as an example. There are two variants, one where the providers pushes changes to the consumer, and another where the consumer pulls changes from the consumer.

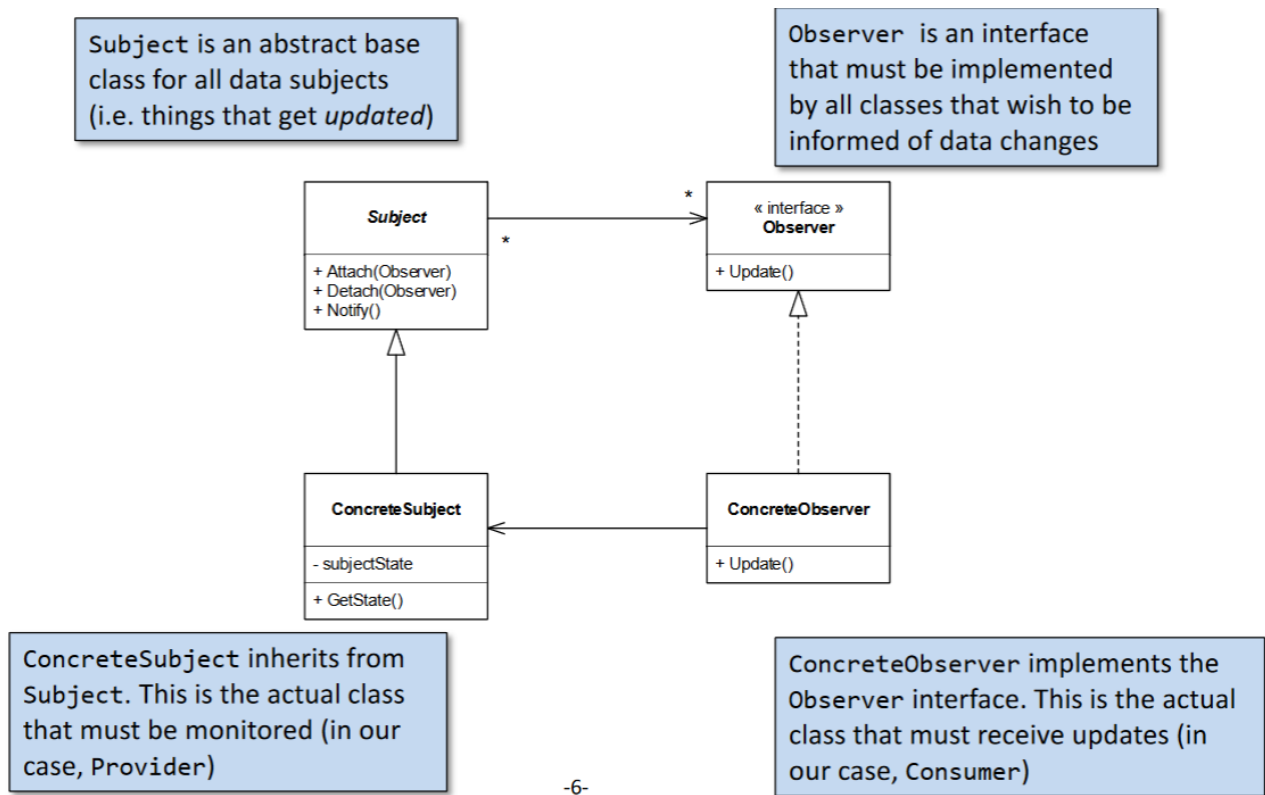


Figur 5: Top variant: pushing changes. Bottom variant: pulling changes. The problem with both of these, is that the objects are dependant upon one another. We need a pattern that allows for decoupling.

We need some mechanism (pattern) that:

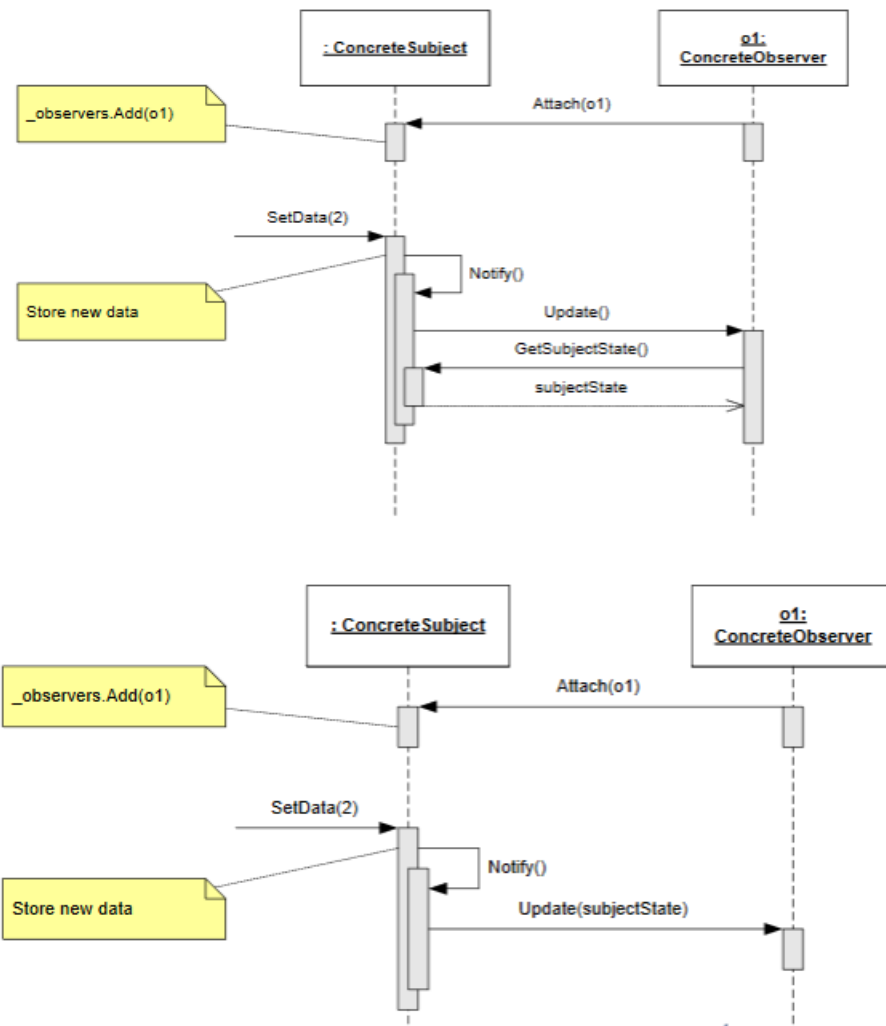
- Allows consumers to be added to provider without changing providers (adhering to the OCP).
- Allows provider to inform consumers of data changes (low coupling).
- Allows many consumers to be informed on updates of same data.

2.1 The pattern



Figur 6: Observer pattern: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

2.2 Push-Pull variants



Figur 7: Top: pull variant. Bottom: push variant

2.3 Different types of subjects

We also need a way, for one observer, to handle multiple subjects of different types.

- How can we handle observers that connect to subjects of *different* types?

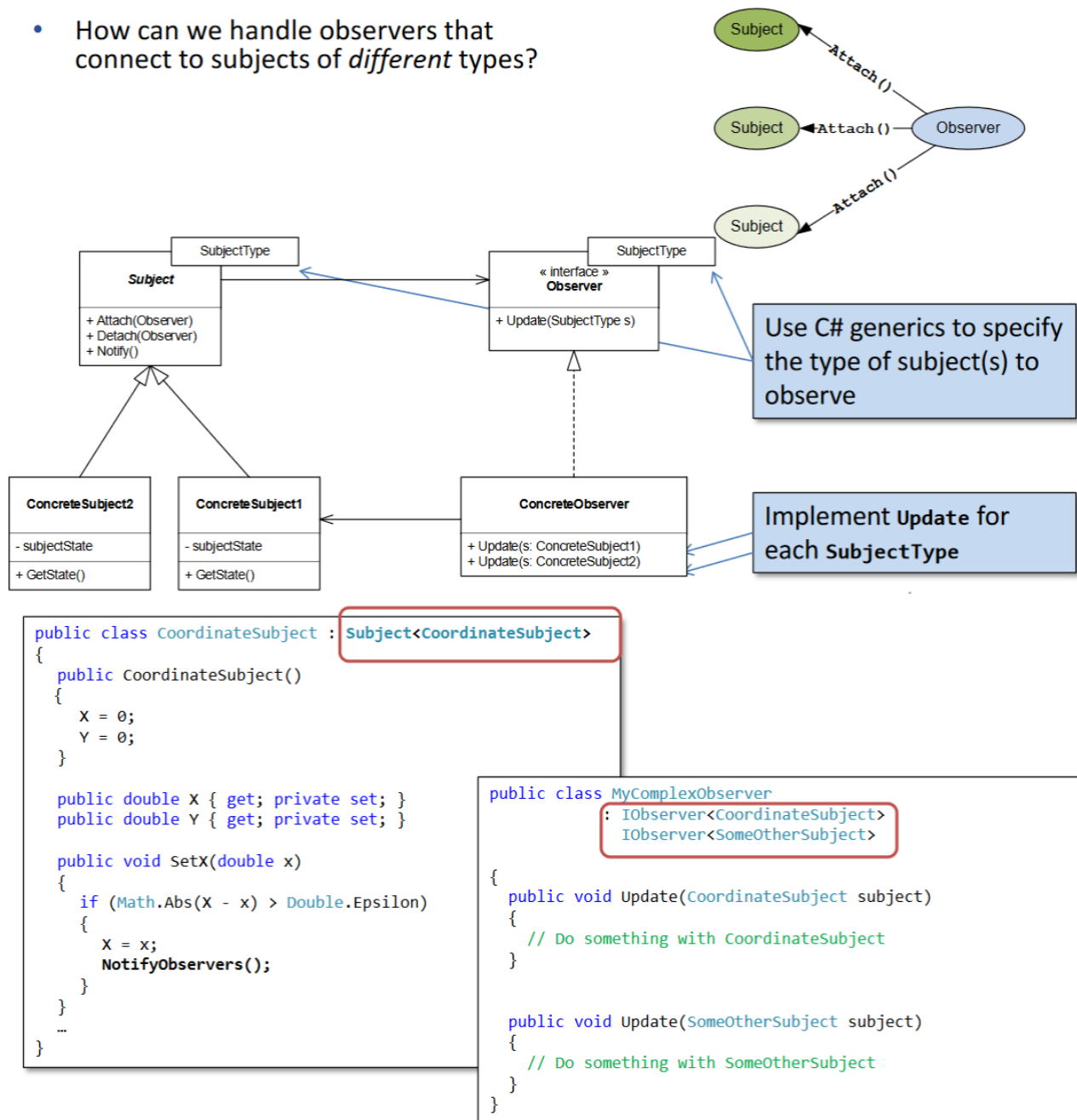


Figure 8: Handling subjects of different types

3 Factory Patterns

Factory Method and the related pattern **Abstract Factory**.

Type: Creational.

The factory patterns exist for one reason: to create objects for its clients. Factories covers a wide range of patterns, from simple methods you call to get an object, to abstract factories which puts together and initialize whole families of related objects. Common for all these implementations are that they carry the non-trivial responsibility to create (possibly families of) objects. A responsibility which otherwise should have been handled by the client and thereby break the SRP from SOLID.

3.1 Factory Method[2]

Define an interface for creating an object, but let the classes that implement the interface decide which object to instantiate.

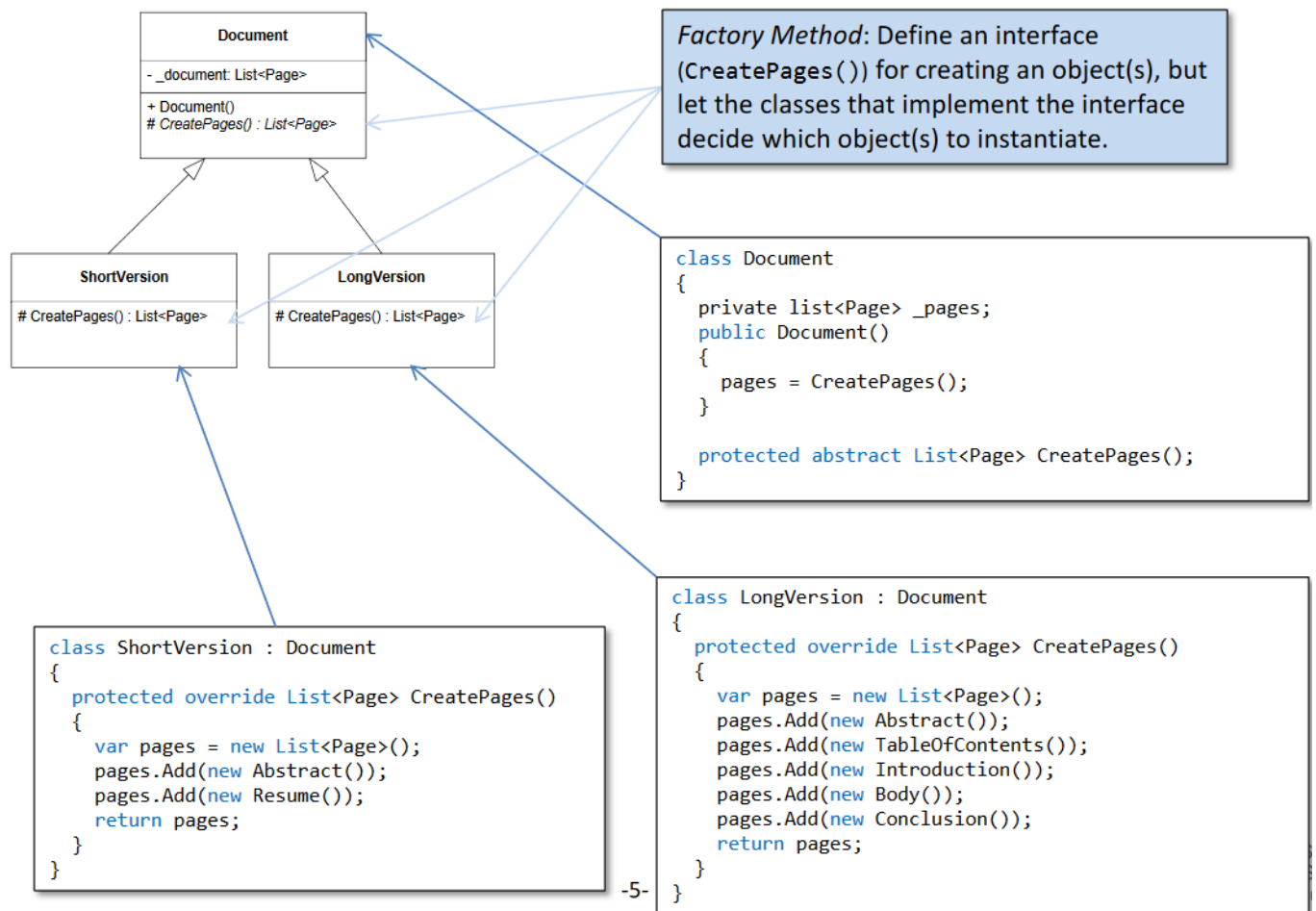


Figure 9: Factory Method for creating a document, implementing the `CreatePages()`.

3.2 Abstract Factory[1]

Define an interface for creating families of related or dependent objects without specifying their concrete classes.

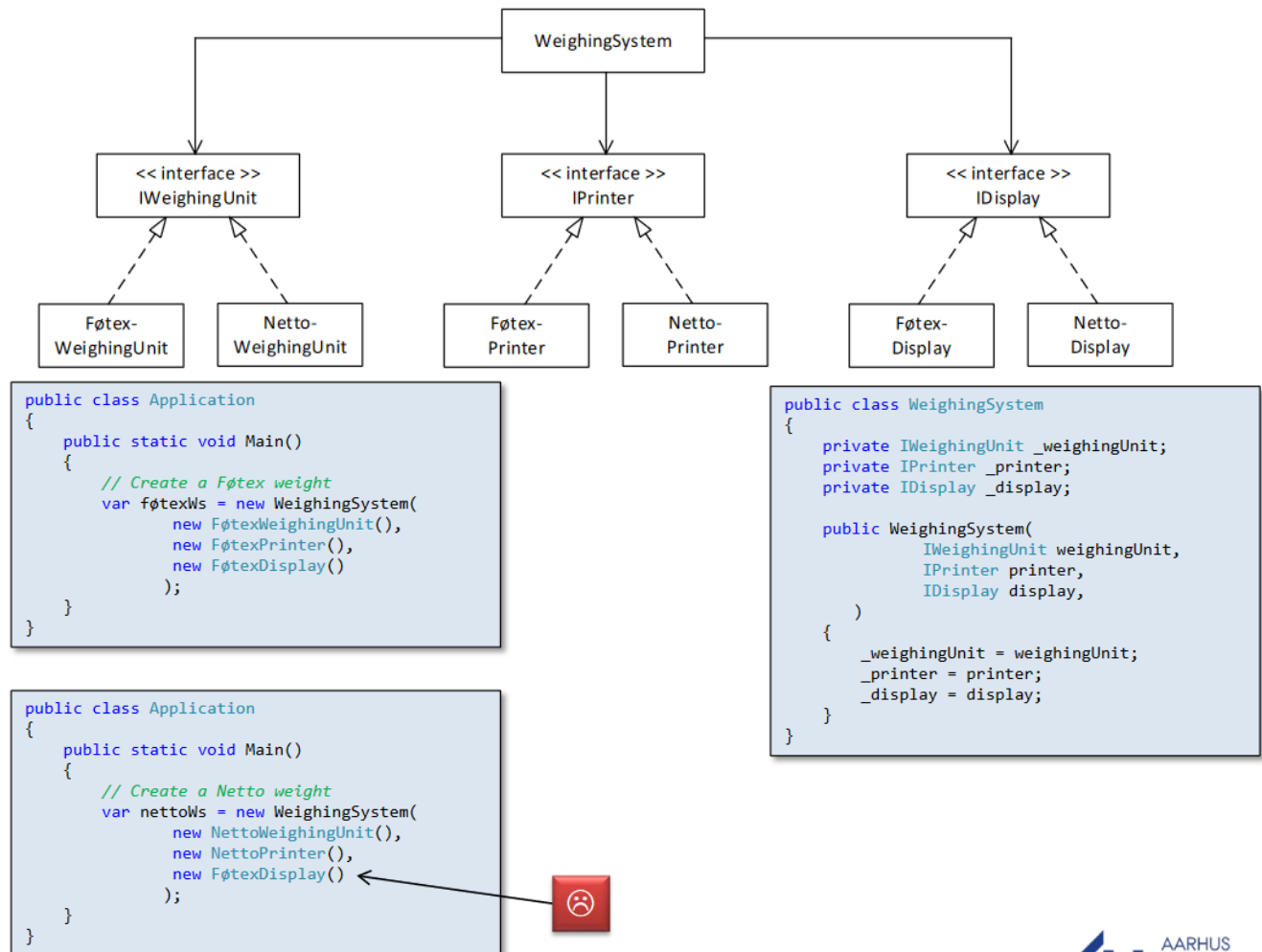


Figure 10: Creating the weighing system without a factory pattern. Complex and error prone!

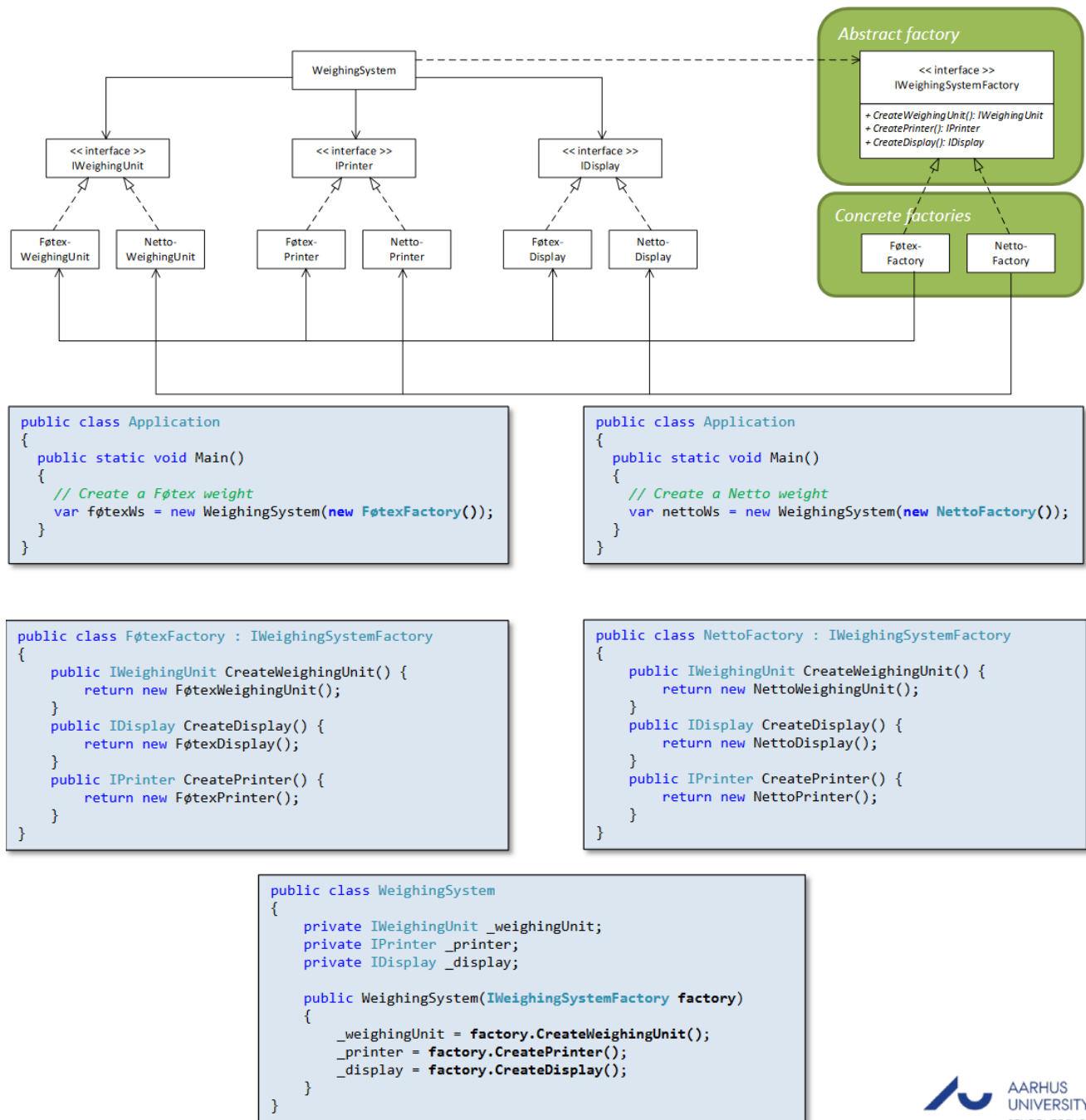


Figure 11: Creating the weighing system with an abstract factory.

4 State Machine Patterns

Type: Behavioral.

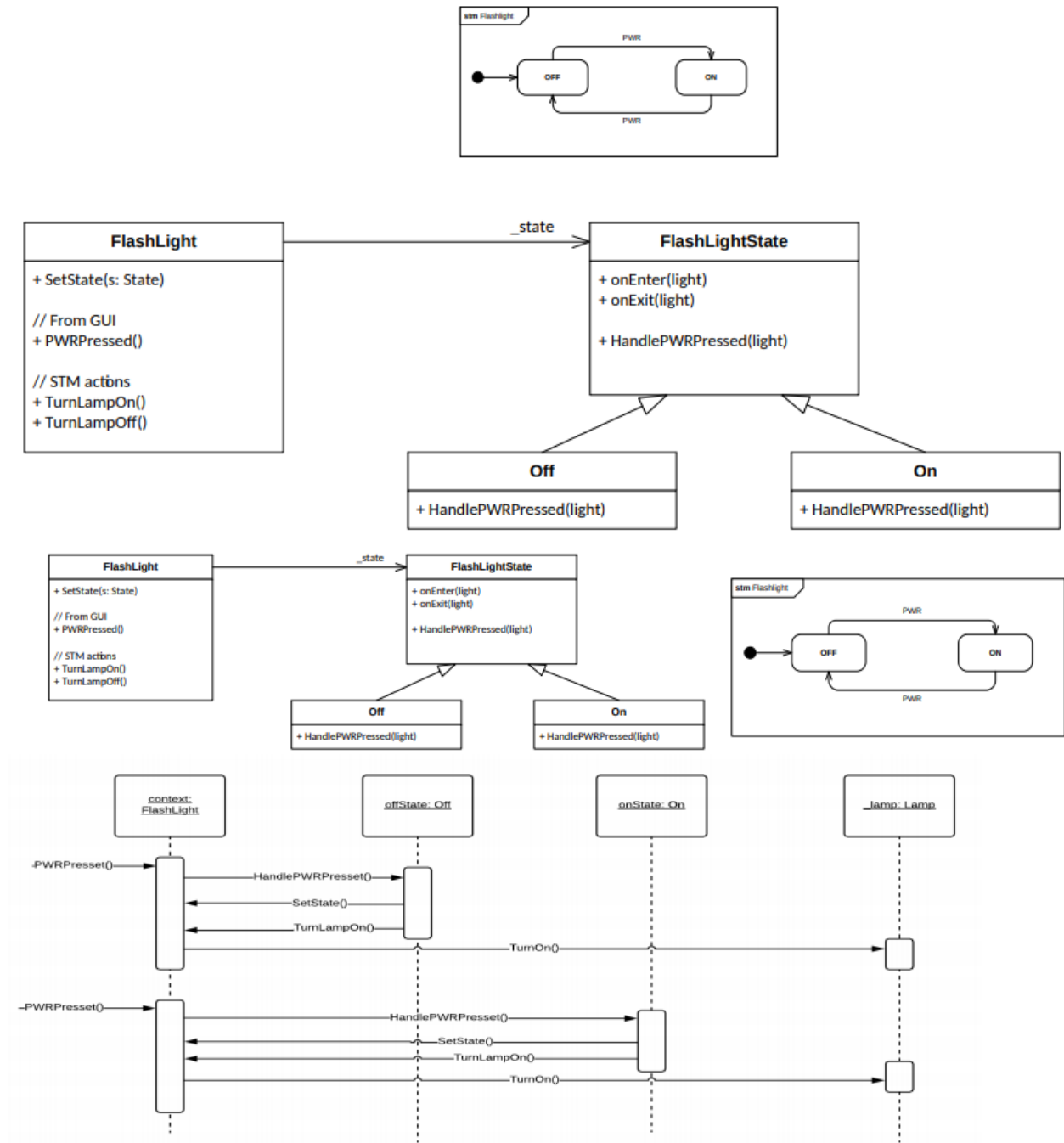
The state pattern is a design pattern that implements a state machine in an object-oriented way. With the state pattern, a state machine is implemented by implementing each individual state as a derived class of the state pattern interface, and implementing state transitions by invoking methods defined by the pattern's superclass.

The state pattern can be interpreted as a strategy pattern which is able to switch the current strategy through invocations of methods defined in the pattern's interface.

This pattern is used to encapsulate varying behavior for the same object based on its internal state. This can be a cleaner way for an object to change its behavior at runtime without resorting to large monolithic conditional statements, and thus improve maintainability.

We will see from the example on figure 12, that:

- Each state is implemented as a subclass of a common abstract state class.
- At any time, the context references exactly one state object - the current state.
- All state-dependent behavior is delegated to this state object - the behavior is externalized.
- The state classes should be kept stateless so they can be singleton implementations.
- When context receives an event, it immediately forwards it to its state object.
- If action shall be perform, the state object calls back into the context to perform the action.
- If a state change shall be performed, the state object will call back to the context, to set the contexts new state object - **Important:** The context is not responsible for setting the new state, the current state is!



Figur 12: The statemachine pattern for a context with two different states

4.1 Nested States

This pattern is especially neat when used for complex (nested, orthogonal) state machines.

Important: Correct order of OnEnter() / OnExit() calls, a bit like constructors/destructors.

- Superstate OnEnter() before substate OnEnter().

- Substate OnExit() before superstate OnExit().

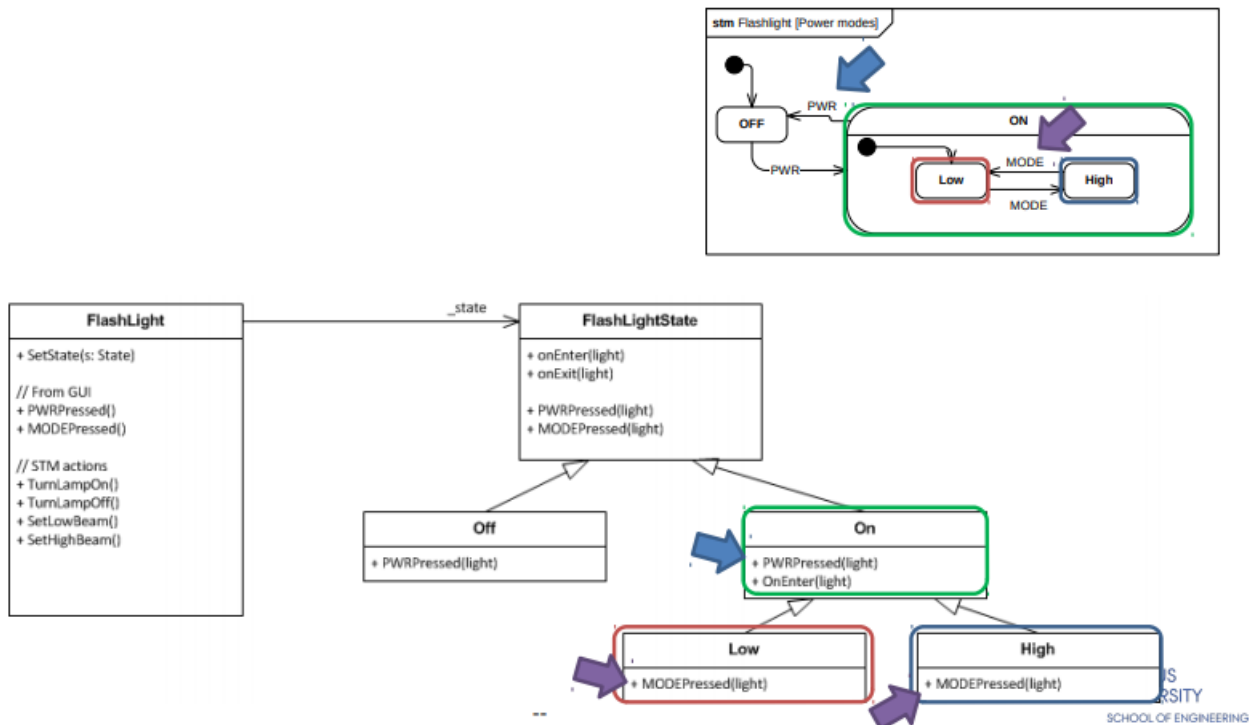


Figure 13: Nested states.

4.2 Orthogonal States

- Two implementation strategies:
 - 1) Collapse into a single state machine: Low-White, Low-Green, Low-Red, High-White, High-Green, High-Red.
 - 2) Create two separate state machines and let the context hold a reference to both.

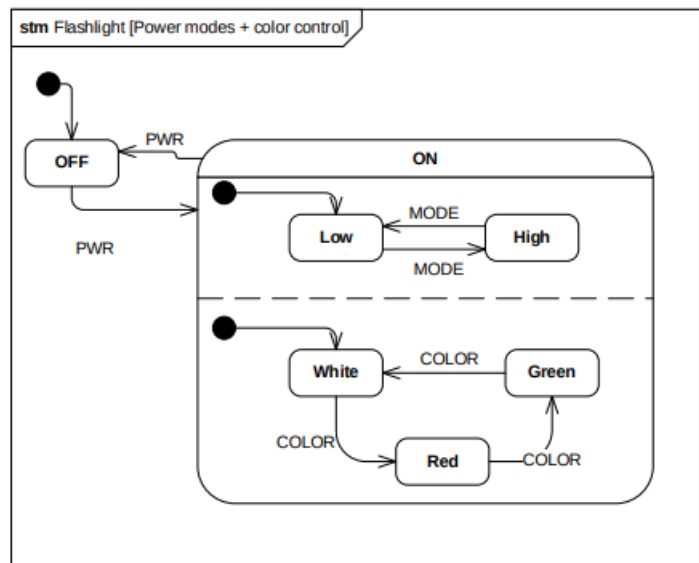


Figure 14: Orthogonal States.

5 GUI Design Pattern

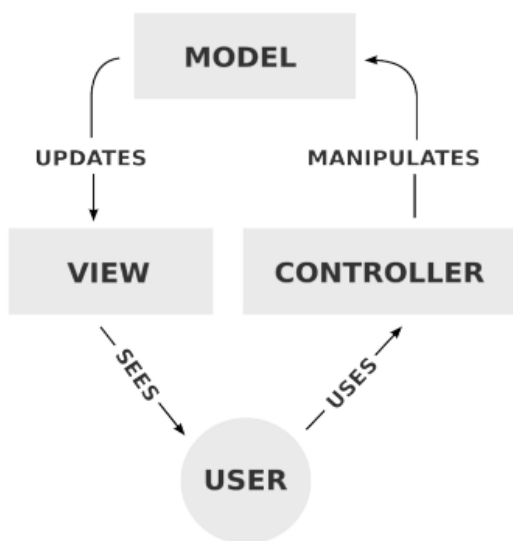
Three design patterns: Model-View-Controller (MVC), Model-View-Presenter (MVP) and Model-View-ViewModel (MVVM).

Why use GUI Design Patterns?

- Dealing with complexities of GUI that work on/with data.
- Managing data/state at various levels: "GUI state", "Session state", "Record state".
- Better separation of concerns
- Better software solutions: Manageable, Testable & Reuseable.

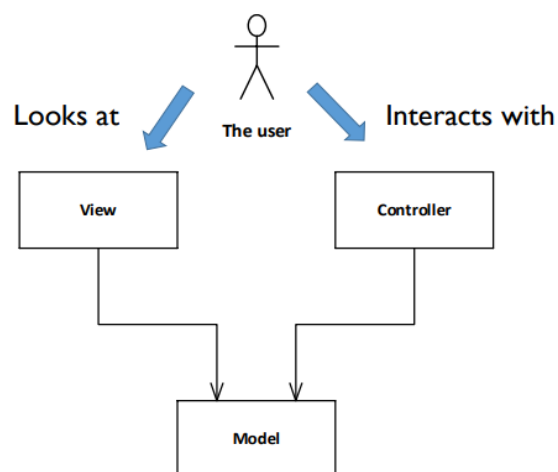
5.1 MVC Pattern

Based on the patterns: Observer, Strategy, Composite.



(a) Model-View-Control

Original MVC



(b) Class diagram showing dependencies, and how a user interacts.

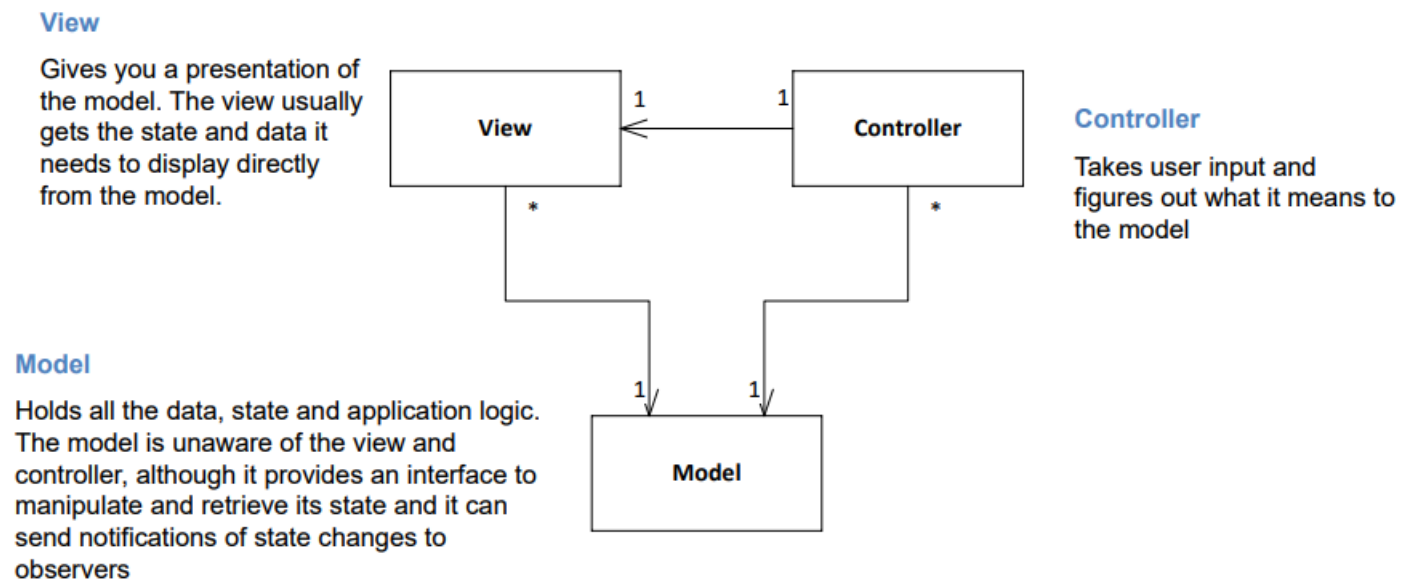
Model holds the data when the application is running, and also contains the business logic. It is mandatory that the model does not depend on the View or the Controller. If the Model has to update the View or Controller this must be done by use of the Observer Pattern.

View is responsible for showing the Model to the user. A View typically only has a minor area of the screen real estate as its output area, but it may own the entire screen. Several Views can be connected to the same model - each View will visualize different properties of the Model. The View is dependent on both the Model and its associated Controller (see figure 16). The View does not send a lot of messages to its Controller, but the View creates and releases its Controller.

Controller gets all input from the user, such as keyboard and mouse strokes. Depending on input, it will

update the Model or send commands to its associated View. When the Controller updates the Model, the View will be notified from the Model through the observer design pattern. In an application there will be one Controller for each View, but only one Controller will receive input at any one time.

MVC Class Diagram



Figur 16: MVC

Summary of the MVC Pattern:

- Strong separation between presentation (V & C) and domain (M).
- Divide GUI widgets into a controller (for reacting to input) and view (for displaying the state of the model). C&V should (mostly) not communicate directly, but through the model.
- Have views (and controllers) observe the model to allow multiple widgets to update without needed to communication directly - observer synchronization.

Advantages / Disadvantages:

- **Advantage:** A strong separation of model and UI makes the application easier to update and maintain.
- **Disadvantage:** Requires more code than forms & controls.

5.2 MVP Pattern

The evolution of operating systems like Microsoft Windows had resulted in generic GUI widgets (controls) that contained both representation and initial user input handling making the original MVC pattern awkward to use, but just collapsing the View and Controller (Document-View) can result in huge classes with too much responsibilities. To solve this: Enter MVP!

Two variants: Supervising Controller and Passive View.

5.2.1 Supervising Controller

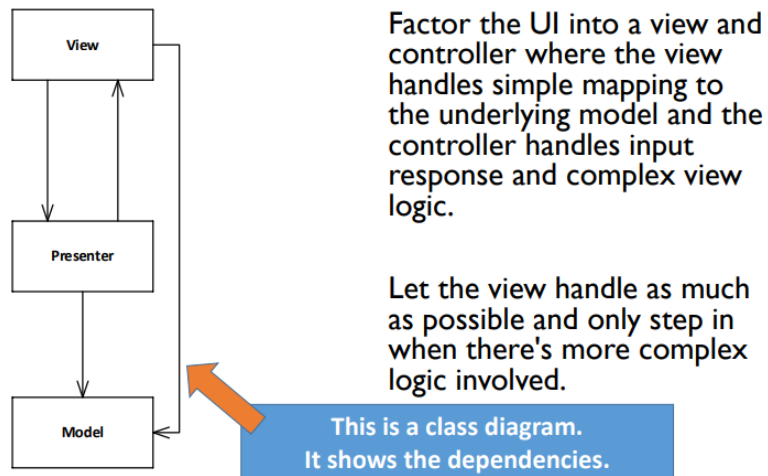
This MVP version is the version that Poul Ejnar sees as the one that best represent the fundamentals of the MVP-pattern.

Model holds the data while the application is running, and it contains the business logic. It is mandatory that the Model does not depend on the View or Presenter, and if it has to update this must be done by use of the Observer Design Pattern.

View is responsible for both showing the model on the screen and the initial handling of user input. Several Views can be connected to the same Model, each view will visualize different properties of the Model. The View can handle some user input itself, but often delegates the handling to the Presenter. The View is dependent on both the Model and its associated Presenter.

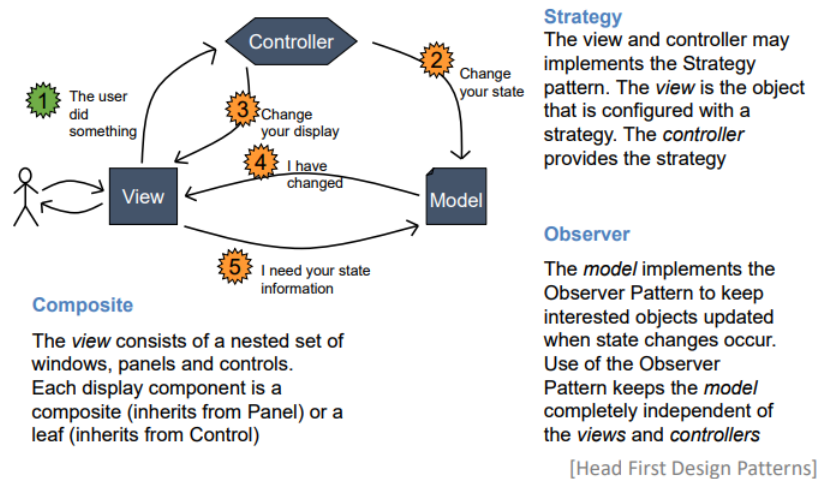
Presenter is responsible for the main part of updates to the Model, as the View only handle the simple cases itself. This is where most of the application behavior is placed.

MVP – Supervising Controller



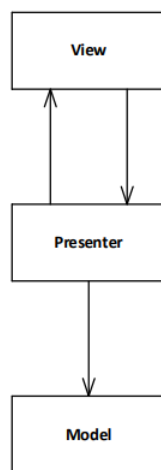
Figur 17: Model-View-Presenter - Supervising

A compound Pattern



Figur 18: Model-View-Presenter - Supervising

5.2.2 Passive View



The View has no direct association to the Model.

The View only contains the widgets that shows the data on the screen, and it will hand over all user input to the Presenter (Fowler calls it "Controller")

The Presenter contains all the GUI logic and is responsible for both updating the Model and updating the View

the Presenter populates the Widgets in the View with data from the Model.

The Presenter can be notified of changes to the Model.

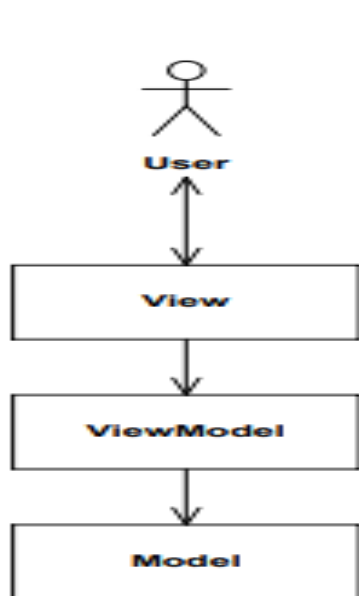
Figur 19: Model-View-Presenter - Passive

5.3 MVVM Pattern

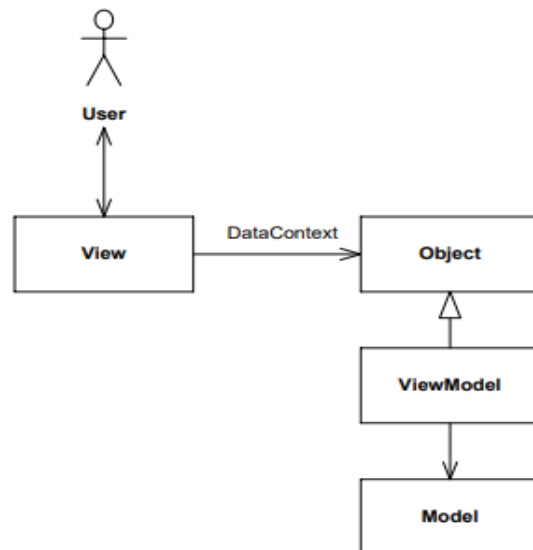
The Model-View-ViewModel Design Pattern[9], is a variation of the MVC Pattern, or the famous MVP Pattern, that is tailored for modern UI development, where the View is the responsibility of a designer rather than a classic developer. The designer is generally a more graphical, artistic focused person, and does less classic coding than a traditional developer. MVVM relies on one more thing: a general mechanism for data binding.

Reasons to use the MVVM Pattern:

- **Separation of concerns:** This is the main motive. The designer is responsible for the view, the developer is responsible for the presentation logic (ViewModel) and business logic (Model).
- **Testability:** The ViewModel has no dependencies on the View which makes it easy to substitute the View with Unit Tests. A clever developer can often place all presentation logic in the ViewModel which makes it possible to get 100% code coverage.
- **Low coupling:** This pattern set the stage for an application with very coupling between the classes that make up the application. And if you add the proper use of interfaces and patterns, you can get a modular application that can evolve over the years to come.
- **Smooth designer/developer workflow:** There is a very loose connection between View and ViewModel, so a designer can work on the View the same time as a developer works on the ViewModel.
- **Portability:** The pattern is not restricted to WPF. The View has to be built specific to the platform you are targeting, but the ViewModel and Model can be built to run on all platforms if you put them in a Portable Class Library.



(a) Class Diagram



(b) Because of data binding, the View does not need an association to the ViewModel - it can retrieve the ViewModel by use of a Locator or dependency injection. This is the MVVM pattern with no ViewModel association

View is responsible for both showing the Model on the screen and the initial handling of user input. The View has a relation to a ViewModel that holds the state of the View, this relation may be implemented as shown on figure 20a or as on figure 20b. The communication between a View and its ViewModel is mainly done by use of Data Bindings, where the View's DataContext property holds the reference to the ViewModel. **ViewModel** holds the View's state and contains the presentation logic - it acts as a middle man between the View and the Model. The ViewModel is used to convert the data from the Model to a format suitable for the View to bind to. The ViewModel also contains Commands the View can use to interact with the Model. RelayCommand etc.

Model is like in the MVC and MVP Pattern. It holds the data while the application is running and it also

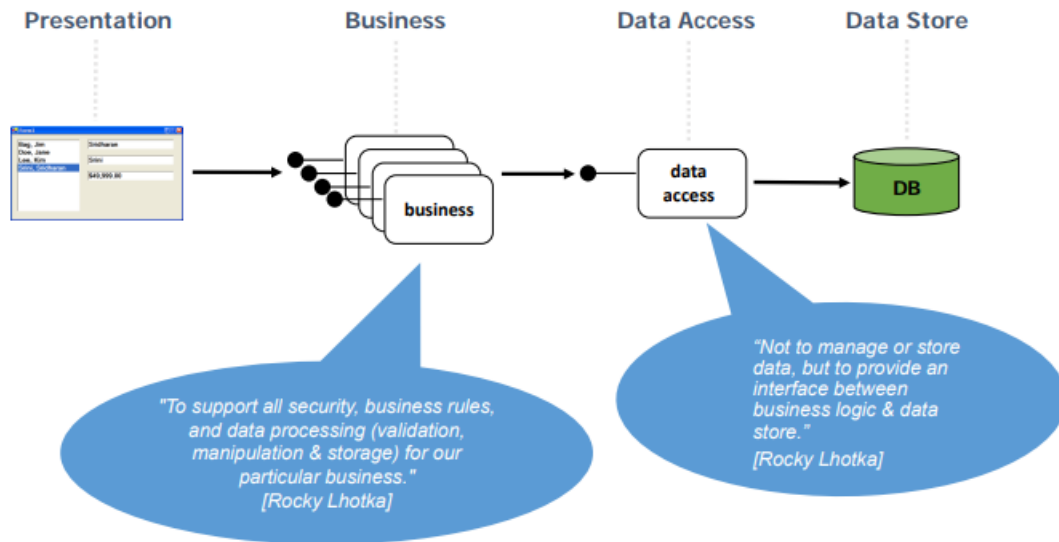
contains all the business logic. It is mandatory that the Model is unaware of the used GUI framework and the application classes in the presentation layer: View and ViewModel.

Basics: For data binding to work optimally the ViewModel must implement the `INotifyPropertyChanged` interface and the ViewModel must hold a reference to the Model. Then the ViewModel must expose all the relevant properties for the View to bind to.

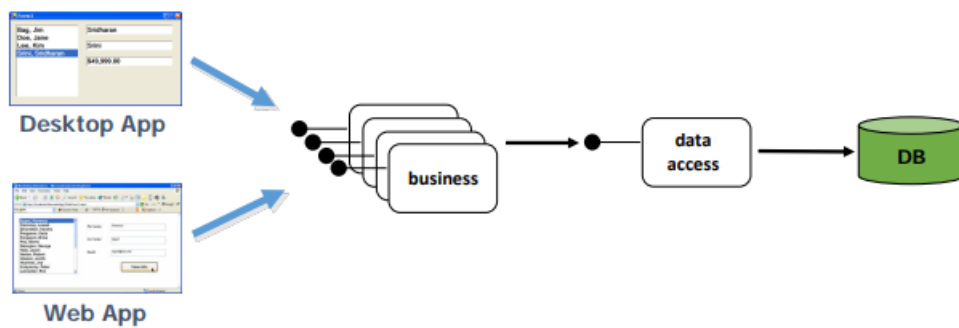
```
public double Height
{
    get { return bmiModel.Height; }
    set
    {
        if (value != bmiModel.Height)
        {
            bmiModel.Height = value;
            NotifyPropertyChanged();
        }
    }
}
```

Figur 21: ViewModel exposing relevant property for the View to bind to.

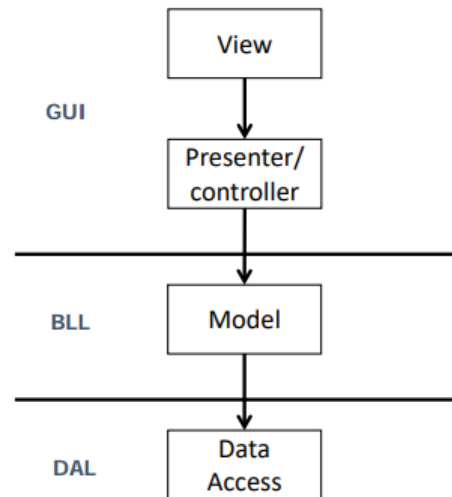
5.4 N-Layer Architecture



Reusable Model



Figur 22: N-Layer



Figur 23: Mapping MVC/MVP to 3-Layer Architecture

6 Parallel Aggregation and MapReduce

6.1 Parallel Aggregation

Aggregation is the action of collecting items to form a total quantity. In the scope of concurrent programming, aggregation is the collection of sub-results to one total result. Divide-and-combine.

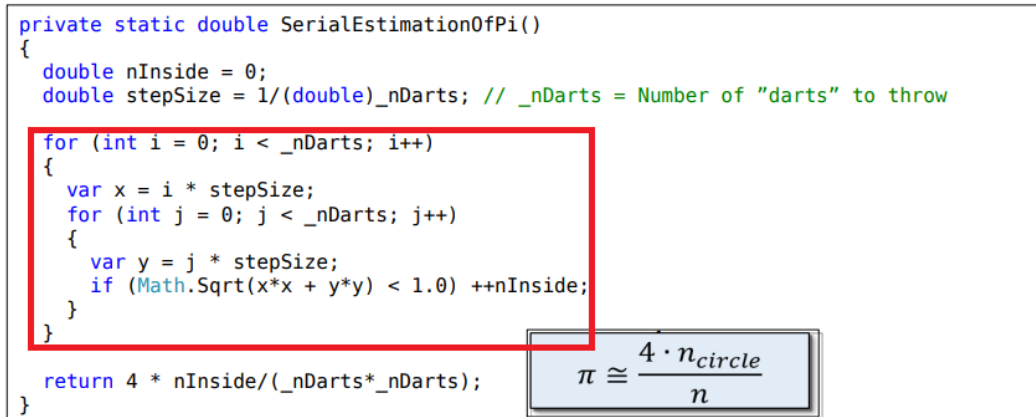


Figure 24: Aggregation, doing serial estimation of PI

Looking at this simple example on figure 24, first question is **where is the aggregation?** Its the part marked in red. Next question is obviously, can we parallelize this calculation and how? We can, and we do it by parallelizing the aggregation part, seen on figure 25.

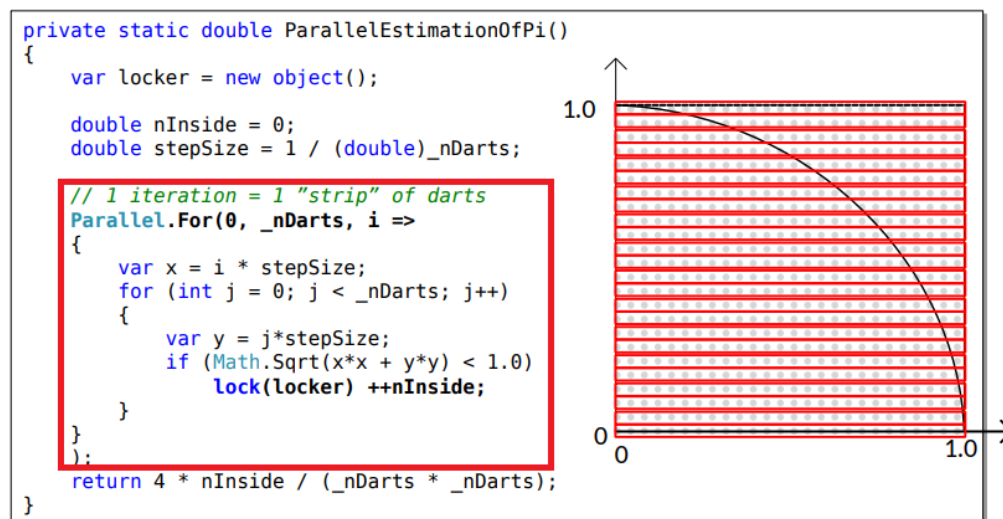


Figure 25: The aggregation part of the estimation of PI has been parallelized.

The solution on figure 25 is correct, but naive. Every iteration of the parallel loop does only a few real cycles worth of work, made up of a few additions, multiplications and divisions, and then takes a lock to accumulate that iterations result into the overall result. The cost of that lock will dominate all of the other work happening in the parallel loop, largely serializing it. To fix this, we need to minimize the amount of synchronization

necessary. That can be achieved by maintaining local sums. We know that certain iterations will never be in conflict with each other, namely those running on the same underlying thread (since a thread can only do one thing at a time), and thus we can maintain a local sum per thread. In addition to passing to **Parallel.For** a delegate for the body, you can also pass in a delegate that represents an initialization routine to be run on each task used by the loop, and a delegate that represents a finalization routine that will be run at the end of each task.

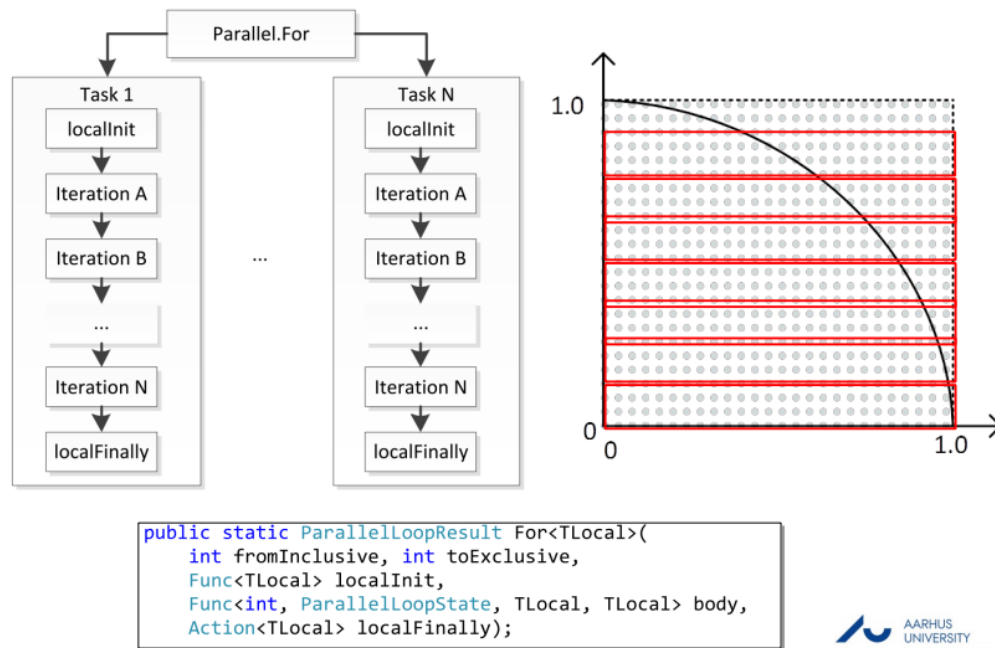


Figure 26: A delegate passed to **Parallel.For** that represents the initialization and finalization routine to be run on each task.

```
private static double ParallelEstimationOfPi()
{
    var locker = new object();
    double nInsideCircle = 0;
    double stepSize = 1 / (double)_nDarts;
    Parallel.For(0, _nDarts,
        () => 0, // localInit: Initialize nInside (passed to first iteration)
        (i, dummyState, nInside) =>
        {
            var x = i * stepSize;
            for (int j = 0; j < _nDarts; j++)
            {
                var y = j * stepSize;
                if (Math.Sqrt(x * x + y * y) < 1.0) ++nInside;
            }
            return nInside; // Handed over to next task executing on thread
        },
        // localFinally: lock and aggregate local result to global result
        inside => { lock (locker) nInsideCircle += inside; });

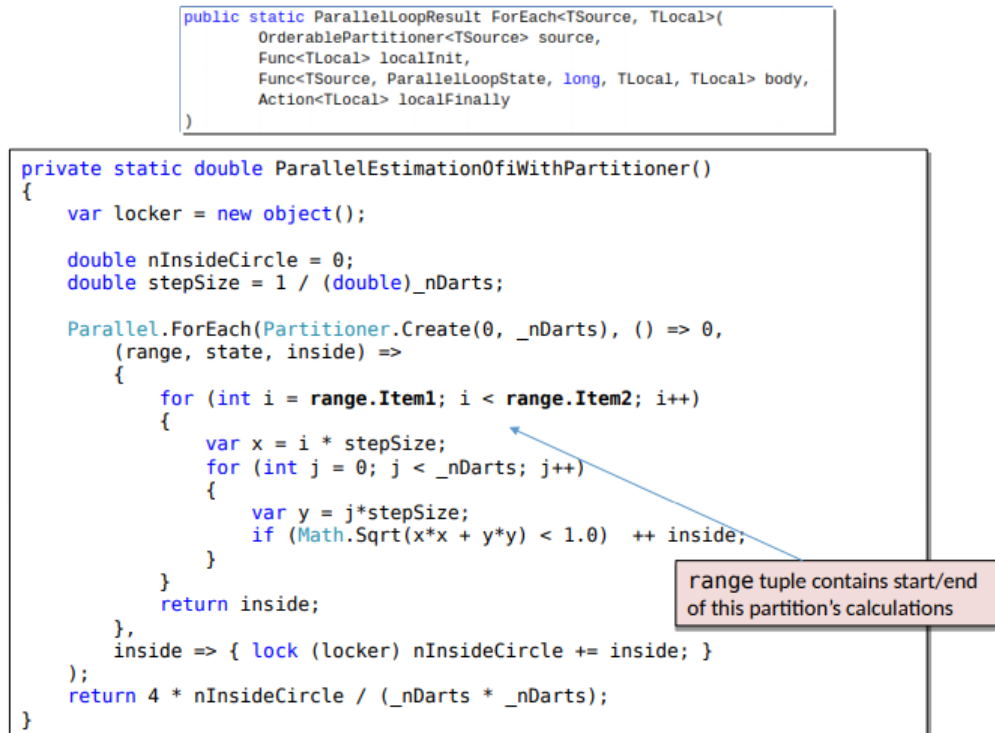
    return 4 * nInsideCircle / (_nDarts * _nDarts);
}
```

Requires no locking - runs in same thread

"Final" operations - requires locking

Figure 27: **localInit** at the beginning of the loop. The loop is all the iterations, no need for locking here as it runs in same thread. For the **localFinally** locking is required.

While the function on figure 27 is more efficient than the serial version, the work done in the delegate is still very limited. We can use a similarly overloaded `Parallel.ForEach()` with a `Partitioner` to create "optimal chunks" of work for each task.



```
public static ParallelLoopResult ForEach<TSource, TLocal>(
    OrderablePartitioner<TSource> source,
    Func<TLocal> localInit,
    Func<TSource, ParallelLoopState, long, TLocal, TLocal> body,
    Action<TLocal> localFinally
)

private static double ParallelEstimationOfPiWithPartitioner()
{
    var locker = new object();
    double nInsideCircle = 0;
    double stepSize = 1 / (double)_nDarts;

    Parallel.ForEach(Partitioner.Create(0, _nDarts), () => 0,
        (range, state, inside) =>
        {
            for (int i = range.Item1; i < range.Item2; i++)
            {
                var x = i * stepSize;
                for (int j = 0; j < _nDarts; j++)
                {
                    var y = j * stepSize;
                    if (Math.Sqrt(x*x + y*y) < 1.0) ++ inside;
                }
            }
            return inside;
        },
        inside => { lock (locker) nInsideCircle += inside; }
    );
    return 4 * nInsideCircle / (_nDarts * _nDarts);
}
```

range tuple contains start/end of this partition's calculations

Figur 28

6.2 MapReduce

MapReduce is a pattern that allows parallel computation on huge data sets.

MapReduce takes a set of input key/value pairs, and produces a set of output key/value pairs. The user of the MapReduce library expresses the computation as two functions: Map and Reduce.

The key to the strategy is to parallelize data processing on many nodes to allow speed-up and thus answer queries quickly.

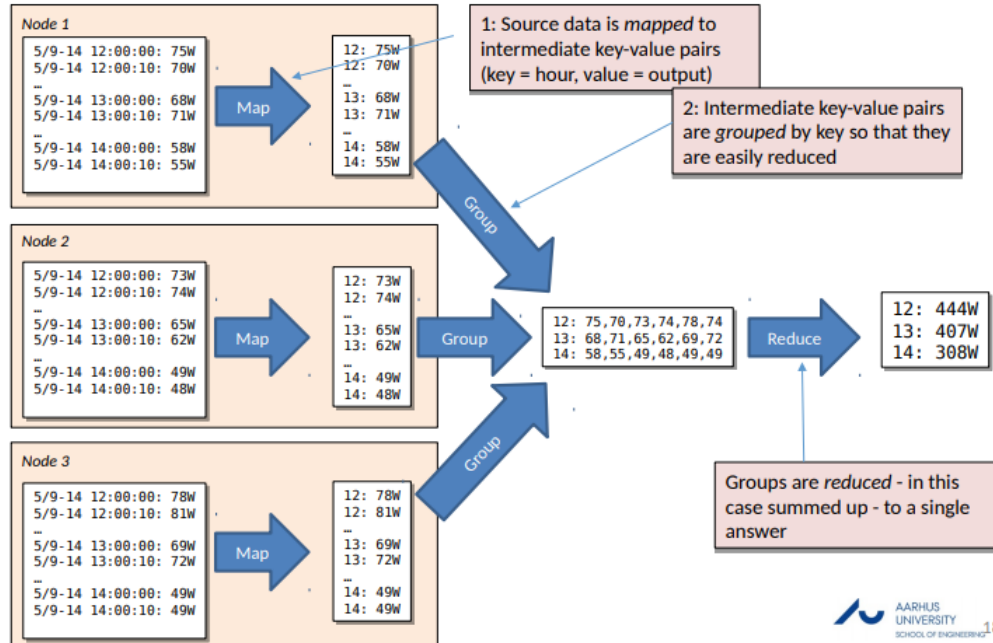
The four steps in MapReduce:

- **1. Distribute:** partitions and distributes source data to different nodes to allow parallel work.
- **2. Map:** transforms source data representation on each node to (a large number of simple) intermediate key-value pairs.
- **3. Group:** groups the intermediate key-value by keys for ease of reduction (a "group-by" operation).
- **4. Reduce:** merges/aggregates/interprets reduced data into an answer to the original query.

Distribute & Group do not vary, usually provided by a framework. **Map & Reduce** vary, provided by user.

Solar Panel, fig 29. Query: whats the total hourly production from all panels? Answer form: MW produced in hourly intervals.

Example: Solar panel



Figur 29: MapReduce example

```
var files = Directory.EnumerateFiles(@"..\..\..\Books", "*.txt").AsParallel();

var wordCount = files.MapReduce(
    path => Source(path),
    map => Map(map),
    group => Reduce(group));

var wc = wordCount.ToList();
wc.Sort(AscendingComparison);

foreach (var pair in wc)
{
    Console.WriteLine("{0}: {1}", pair.Key, pair.Value);
}
```

Listing 1: MapReduce implementation example.

7 Futures and Pipelines

7.1 Dependencies

A dependency is the Achilles heel of parallelism. A dependency between two operations implies that one operation can't run until the other operation has completed, inhibiting parallelism. Many real-world problems have implicit dependencies, and thus it's important to be able to accommodate them and extract as much parallelism as is possible. It's very common in real-world patterns where dependencies between components form a directed acyclic graph. Consider compiling a solution of eight code projects. Some have references to others, and thus depend on those projects being built first. The dependencies are:

- Components 1, 2 and 3 depend on nothing else.
- Component 4 depends on 1.
- Component 5 depends on 1, 2 and 3.
- Component 6 depends on 3 and 4.
- Component 7 depends on 5 and 6, and has no dependencies on it.
- Component 8 depends on 5, and has no dependencies on it.

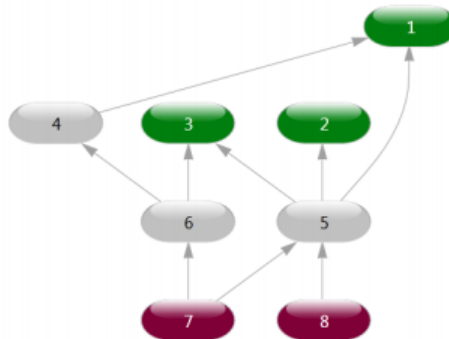


Figure 30: DAG of the solution to be compiled

```
C#
var f = Task.Factory;
var build1 = f.StartNew(() => Build(project1));
var build2 = f.StartNew(() => Build(project2));
var build3 = f.StartNew(() => Build(project3));
var build4 = f.ContinueWhenAll(new[] { build1 },
    _ => Build(project4));
var build5 = f.ContinueWhenAll(new[] { build1, build2, build3 },
    _ => Build(project5));
var build6 = f.ContinueWhenAll(new[] { build3, build4 },
    _ => Build(project6));
var build7 = f.ContinueWhenAll(new[] { build5, build6 },
    _ => Build(project7));
var build8 = f.ContinueWhenAll(new[] { build5 },
    _ => Build(project8));
Task.WaitAll(build1, build2, build3, build4, build5, build6, build7, build8);
```

Figure 31: If building each component is represented as a **Task**, we can take advantage of continuations to express as much parallelism as possible

Iterating in lock step

A common algorithm pattern is to have iterations, each consisting of many calculations, where iteration $i+1$ depends on calculations in iteration i . In these cases we can't parallelize the iterations, as they are dependent, but we can parallelize the calculations within each calculation. We must ensure **lock step**: all parallel calculations of one iteration have completed before calculations in next iteration start.

```
// Run simulation
for (int step = 0; step < timeSteps; step++)
{
    for (int y = 1; y < plateSize - 1; y++)
    {
        for (int x = 1; x < plateSize - 1; x++)
        {
            currIter[y, x] =
                ((prevIter[y, x - 1] +
                 prevIter[y, x + 1] +
                 prevIter[y - 1, x] +
                 prevIter[y + 1, x]) * 0.25f);
        }
    }
    Swap(ref prevIter, ref currIter);
}
return prevIter;
```

```
// Run simulation
for (int step = 0; step < timeSteps; step++)
{
    Parallel.For(1, plateSize - 1, y =>
    {
        for (int x = 1; x < plateSize - 1; x++)
        {
            currIter[y, x] =
                ((prevIter[y, x - 1] +
                 prevIter[y, x + 1] +
                 prevIter[y - 1, x] +
                 prevIter[y + 1, x]) * 0.25f);
        }
    });
    Swap(ref prevIter, ref currIter);
}
return prevIter;
```

Figure 32: Top we see the iterations done sequentially, and below they are done in parallel

Typically this approach is sufficient, it can be more efficient (largely for reasons of cache), however, to ensure that the same thread processes the same sections of iteration space on each time step. For this example, we spin up one **Task** per processor and assign each portion of the plate's size: each Task is responsible for processing that portion at each time step. Now we need to ensure that each Task does not go on to process its portion of the plate at iteration $i+1$ until all tasks have completed processing iteration i . For that we use a barrier.

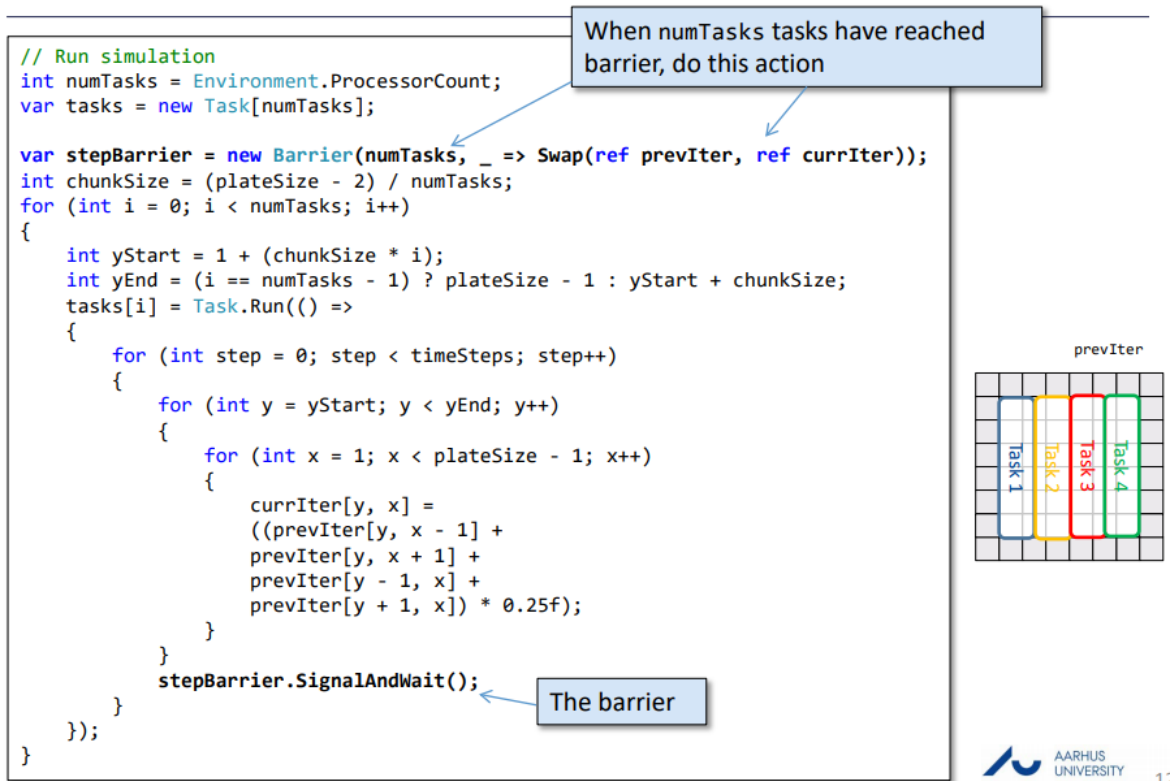


Figure 33: Task for iterating, and barrier to divide portions.

7.2 Futures

A **Future** is a stand-in for a computational result that is initially unknown but becomes available at a later time. The process of calculating the result can occur in parallel with other computations - a future task is a task that returns a value (`Task<TResult>`). Futures as asynchronous functions returning a value, parallel tasks are asynchronous actions that does not return a value. Futures are used when we want to parallelize code with data dependencies.

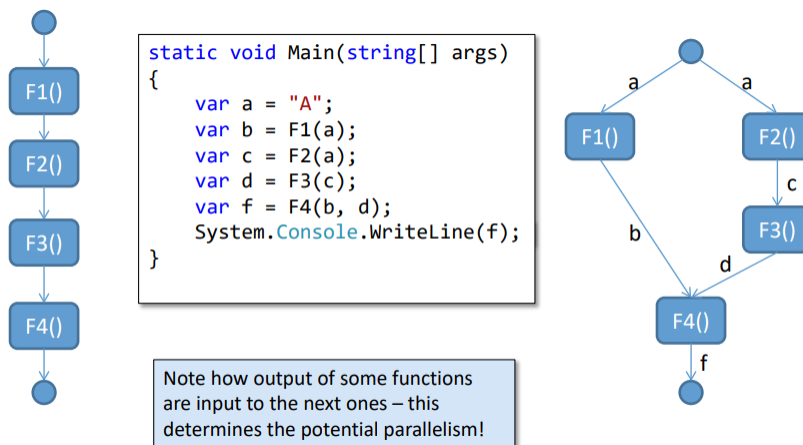


Figure 34: Example code is a body of a sequential method

This can be parallelized using futures, as shown on figure 35. On a multicore system **F1** will be able to run in parallel with the current thread. This means that **F2** can begin executing without waiting for **F1**. The function **F4** will execute as soon as the data it needs becomes available. It doesn't matter whether **F1** or **F3** finishes first, because the results of both functions are required before **F4** can be invoked. It doesn't matter which branch of the task graph shown in figure 34 runs asynchronously. Could have put **F2** and **F3** inside a future.

```
static void Main()
{
    var a = "A";
    Task<string> futureB = Task.Run(() => F1(a));
    var c = F2(a);
    var d = F3(c);
    var f = F4(futureB.Result, d);
    Console.WriteLine(f);
}
```

- Main task runs right part of tree
- futureB is "return value" from F1() and queried for result as F4() is called

Figur 35

7.3 Pipelines

The Pipeline pattern uses parallel tasks and concurrent queues to process a sequence of input values. Each task implements a stage of the pipeline, and the queues act as buffers that allow the stages of the pipeline to execute concurrently, even though the values are processed in order. You can think of software pipelines as analogous to assembly lines in a factory, where each item in the assembly line is constructed in stages.

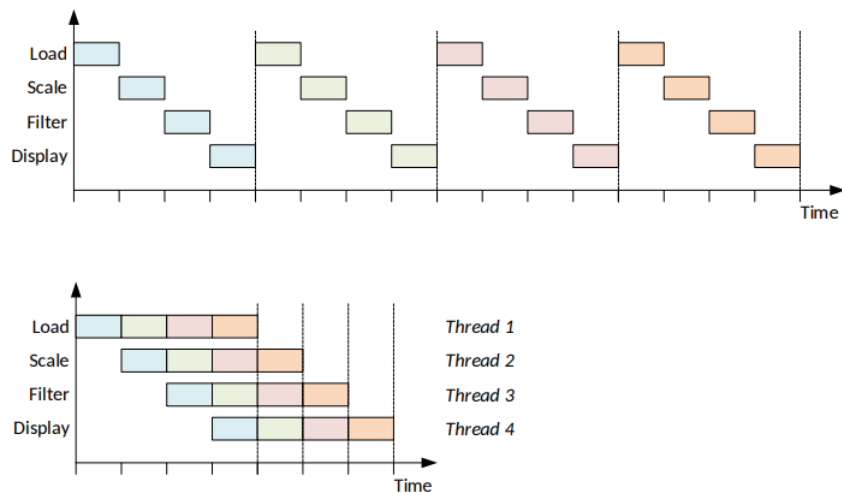


Figure 36: Processing images sequentially. Top is without pipeline, and bottom is pipelined.

Synchronization:

BlockingCollection<T> Class, lets us signal "end of file" condition with the **CompleteAdding** method. The collection also has another important method **GetConsumingEnumerable**, which returns an enumeration a consumer can use to "take" the values. There may be many consumers of a single producer. When a consumer "takes" a value, no other consumer will see it. A blocking collection's **GetConsumingEnumerable** method can be called by more than one consumer. This allows values from the producer to be divided among multiple recipients. If a recipient gets a value from the blocking collection, no other consumer will also get that value.

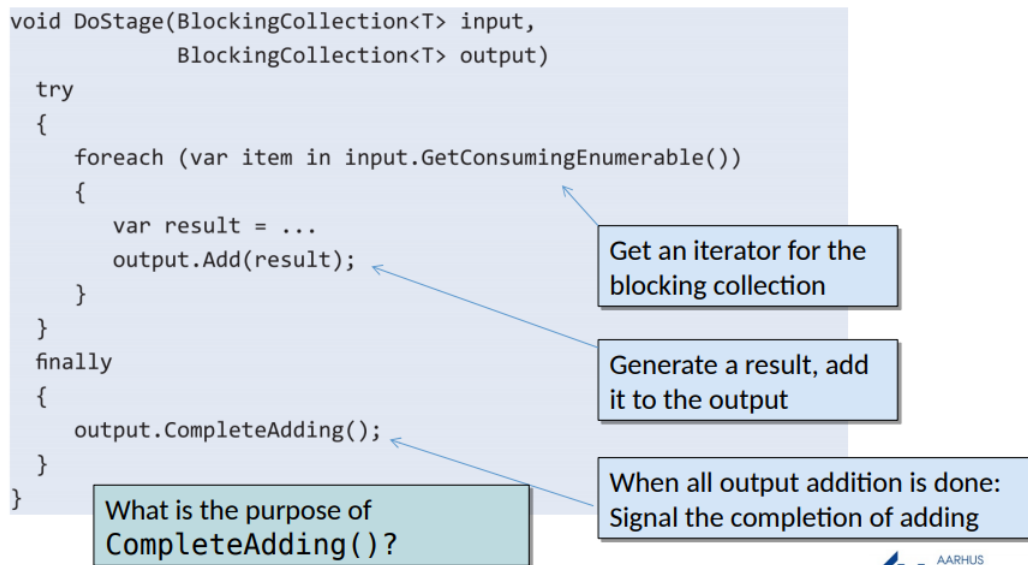


Figure 37: C# pipeline stage implementation, using tasks and concurrent queues.

Bottleneck when steps are of unequal duration. A solution is to add another filter stage to feed the display stage.

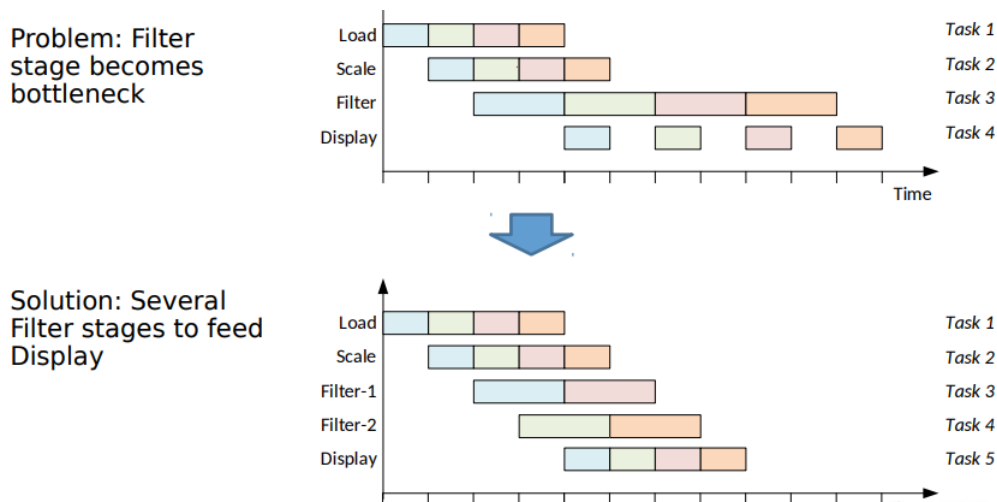
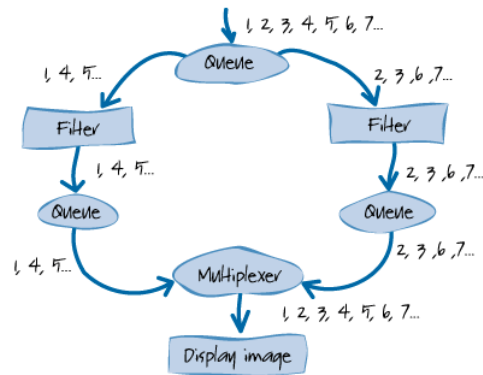


Figure 38

Adding another Filter stage, and then before sending the filtered images to the display stage, pass them by a multiplexer. The Multiplexer utilizes the fact that the `BlockCollection<T>` class allows you to read values

from multiple producers - the method **TakeFromAny**.



Figur 39

The images don't need to be sorted or reordered. Instead, the fact that each producer queue is locally ordered allows the multiplexer to look for the next value in the sequence by simultaneously monitoring the heads of all of the producer queues. **TakeFromAny** allows the multiplexer to block until any of the producer queues has a value to read.

Cancelling a pipeline with a cancellation token, passed from a higher layer of the application, such as the UI. In case of a pipeline stage, we need to observe this cancellation token in two places, shown in figure 40. The first point is at the beginning of the loop, and here we simply break. The second point is an overloaded version of **Add** accepting a token. This prevents the program from possible deadlock.

```
void DoStage(BlockingCollection<T> input,
             BlockingCollection<T> output,
             CancellationToken token)
{
    try
    {
        foreach (var item in input.GetConsumingEnumerable())
        {
            if (token.IsCancellationRequested) break;
            var result = ...
            output.Add(result, token);
        }
    }
    catch (OperationCanceledException) { }
    finally
    {
        output.CompleteAdding();
    }
}
```

Figur 40

8 Software Architecture

What? Why? How? When?

Structure, Process, Documentation, Styles

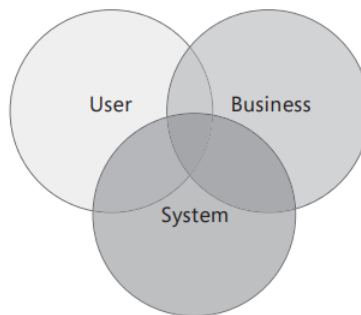
Software application architecture is the process of defining a structured solution that meets all of the technical and operational requirements, while optimizing common quality attributes such as performance, security, and manageability. It involves a series of decisions based on a wide range of factors, and each of these decisions can have considerable impact on the quality, performance, maintainability, and overall success of the application.

The decisions that are hard to change.

The expert developers shared understanding of how the system works.

Software Architecture is the important stuff.

Why is architecture important? Like any other complex structure, software must be built on a solid foundation. Poor software can lead to unstable software, that can't support existing or future business requirements, or is difficult to deploy or manage in a production environment.



Figur 41: Architecture should cover not only the system needs, but also the user and business needs.

Goals of architecture: seeks to build a bridge between business requirements and technical requirements.

- Expose the structure of the system, but hide the implementation details
- Realize all of the use cases and scenarios
- Try to address the requirements of various stakeholders
- Handle both functional and quality requirements

Architectural Landscape: It is important to understand the key forces that shape architectural decisions today, and will change architectural decisions in the future.

- **User empowerment** A design that supports this is flexible, configurable, and focused on the user experience. Do not overload users with unnecessary options; understand the key scenarios and make them as simple as possible

- **Market maturity** Take advantage of existing platform and technology options. build on higher level application frameworks where it makes sense, so you can focus on what is uniquely valuable in your application.
- **Flexible design** Take advantage of loose coupling to allow reuse and improve maintainability.
- **Future trends** When building your architecture, understand that future trends will affect your design eventually.

Key architecture principles

- **Build to change instead of to last.**
- **Model to analyze and reduce risk.** Use design tools, modeling system (UML) and visualize where appropriate to help you capture requirements and architectural and design decisions.
- **Use models as a communication and collaboration tool.** Efficient communication of the design, the decisions you make, and ongoing changes to the design, is critical to good architecture.
- **Identify key engineering decisions.** Understand the key decisions, and the areas where mistakes are most often made.

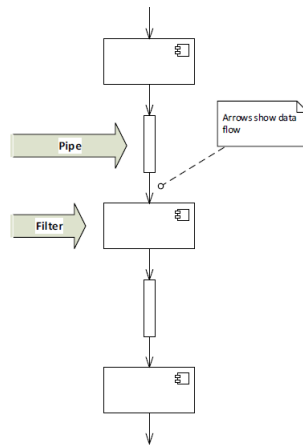
Key design principles

- **Separation of concerns.** Divide your application into distinct features with as little overlap in functionality as possible. The important factor is minimization of interaction points to achieve high cohesion and low coupling.
- **Single Responsibility principle**
- **Principle of Least Knowledge.** (also known as the Law of Demeter or LoD). A component or object should not know about internal details of other components or objects
- **Don't repeat yourself (DRY).** You should only need to specify intent in one place. For example, in terms of application design, specific functionality should be implemented in only one component.
- **Minimize upfront design.** Only design what is necessary. In some cases, you may require upfront comprehensive design and testing if the cost of development or a failure in the design is very high

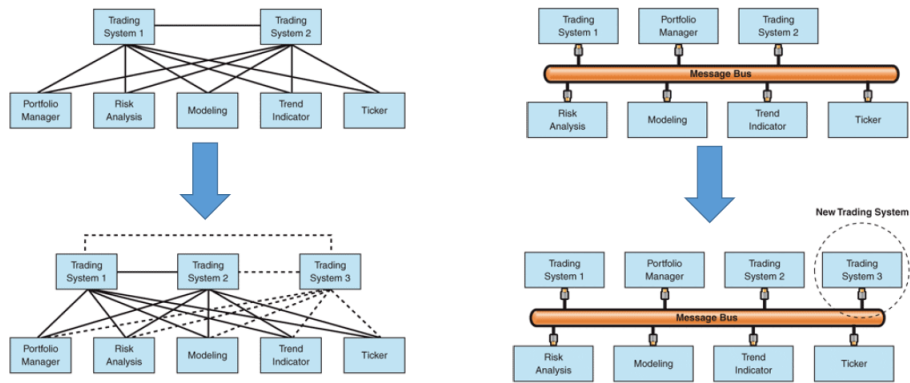
Design Practices

Design Styles

Pipes and filters and message bus.



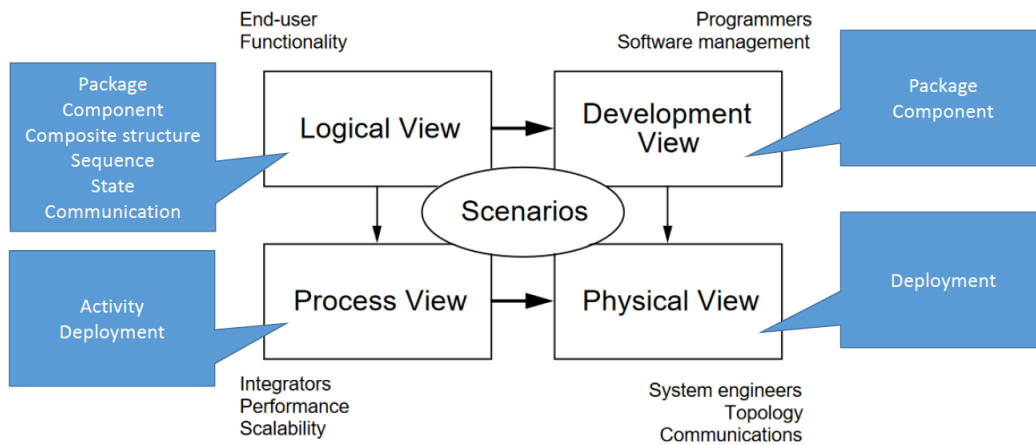
Figur 42: Pipes and filters architectural style.



Figur 43: Message bus architectural style:

Design Views - 4+1 view

AKA N+1 View. Der kan tilføjes flere views, fx. data view.



Figur 44: 4+1 View

9 Mandatory Exercise: Chain of Responsibility

Type: Behavioral

The purpose of using this pattern specifically, is if your program can be viewed as a chain, where each link of the chain, handles an event or passes the responsibility on.

A typical scenario involves one handler, and a varying amount of receivers. There should only be loose coupling between the sender and receiver. Once the handler is utilized in handling a message, it will pass the message, or request, on to the first link of the chain.

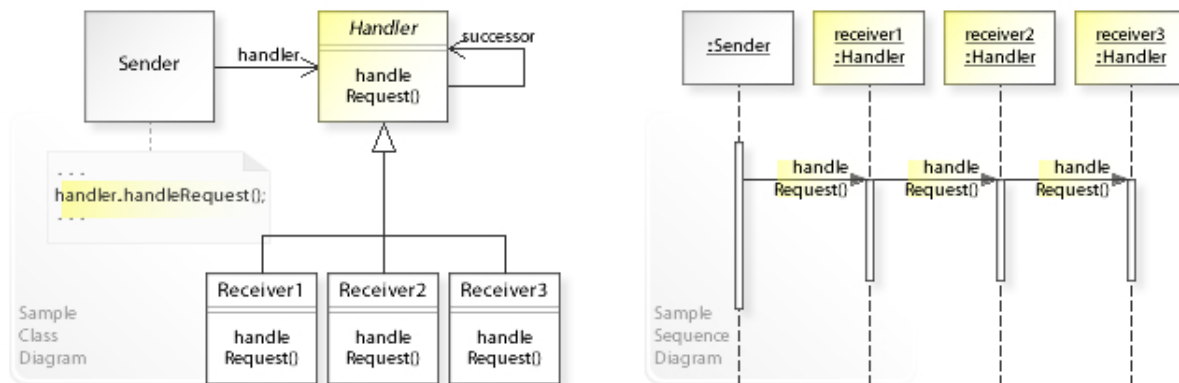


Figure 45: Chain of Responsibility

What we see, is that the client (sender) ask for a handler to handle the request. The `handleRequest` is then passed down the chain. On figure 47 we can see an implementation with sub-handlers.

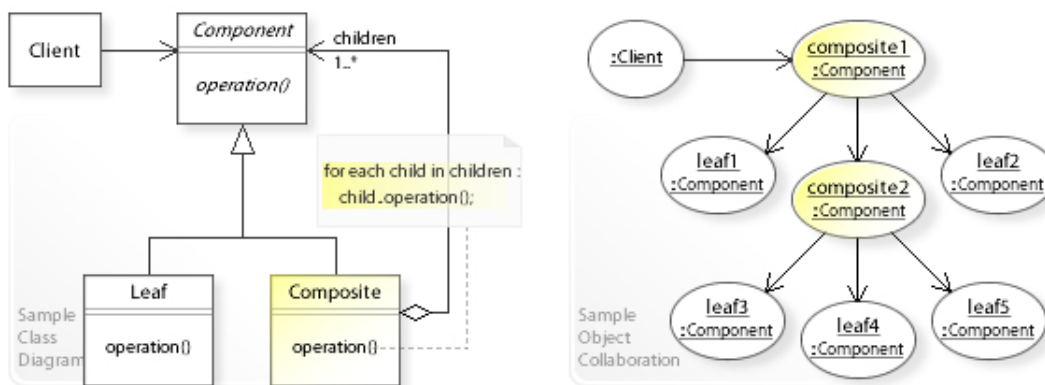
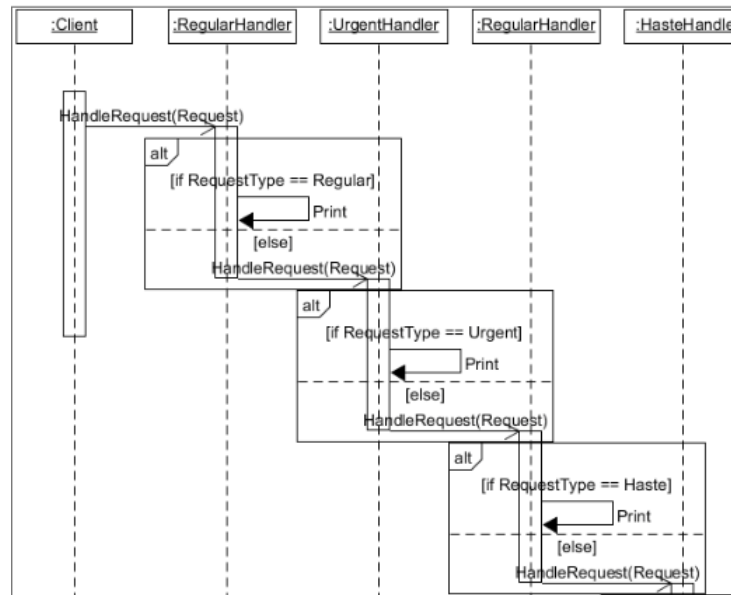


Figure 46: Chain of Responsibility

9.1 Implementation

Next lets look at the way this pattern is implemented. The **abstract class Handler** must have a `SetNextHandler` function, which is used to bind the chain together. Then it must also have an abstract function

HandleRequest. The derived Handle functions must provide the logic for the HandleRequest function. Typically in the type of an **if/else-statement**. If this type of request can be handled by this specific handler, do that. Else call the HandleRequest function of the next handler.



Figur 47: Chain of Responsibility

9.2 Comparison

- **Chain of Responsibility:** sends a request sequentially through a chain, until a recipient can handle that request.
- **Observer:** sends a request simultaneously to all relevant recipients. This pattern further allows all recipients to subscribe/unsubscribe further reception of requests.
- **Command:** Is different in that the handler has a one-way communication to the receiver. The handler locates the concrete object that can handle the request, and then sends it there.
- **Decorator:** has some similarities, but where CoR is behavioral, Decorator is structural. Decorator adds extra functionality to the objects and extra responsibility, whereas CoR seeks low coupling.

9.3 Conclusion

Chain of Responsibility is applicable in situations where one or more of certain criteria are fulfilled:

- The handler is determined automatically upon the nature of the request.
- The handler is not known in advance, so the request can not be sent directly
- More than one object would be able to handle the request.
- The chain of objects that may handle the request, should be specified in a certain way.

10 SOLID

There are four symptoms which are tell-tale signs of poor architecture. Any application that exhibits them is suffering from a design that is rotting from the inside out. But what causes that rot to take place. Before looking at what makes a design SOLID, let's look at what makes a design rotten.

Design- and software principles[8]

There are four primary symptoms that tell us that our designs are rotting:

- **Rigidity:** The tendency for software to be difficult to change, even in simple ways. Every change causes a cascade of subsequent changes in dependent modules.
- **Fragility:** Closely related to rigidity is fragility. Fragility is the tendency of the software to break in many places every time it is changed. Often the breakage occurs in areas that have no conceptual relationship with the area that was changed.
- **Immobility:** The inability to reuse software from other projects or from parts of the same project. When immobile, software usually gets rewritten rather than reused.
- **Viscosity:** comes in two forms: of the design, and of the environment. Faced with a change, engineers usually find more than one way to make the change. Some of the ways preserve the design, others do not (hacks). When the design preserving methods are harder to employ than the hacks, then the **viscosity of the design** is high. **Viscosity of environment** comes about when the development environment is slow and inefficient. For example, if compile times are very long, engineers will be tempted to make changes that don't force large recompiles, even though those changes are not optimal from a design point of view

What kind of changes cause designs to rot? Changes that introduce new and unplanned dependencies. Each of the four symptoms mentioned above is either directly, or indirectly caused by improper dependencies between the modules of the software. It is the dependency architecture that is degrading, and with it the ability of the software to be maintained.

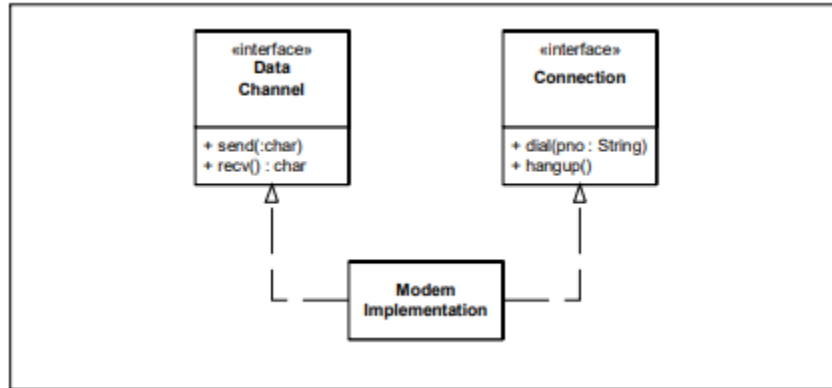
S - Single responsibility principle[7]

There should never be more than one reason for a class to change. Each responsibility is an axis of change. When the requirements change, that change will be manifest through a change in responsibility amongst the classes. If a class assumes more than one responsibility, then there will be more than one reason for it to change. If a class has more than one responsibility, then the responsibilities become coupled. Changes to one responsibility may impair or inhibit the class' ability to meet the others. This kind of coupling leads to fragile designs that break in unexpected ways when changed.

What is responsibility? **A reason for change.**

Listing 9-1**Modem.java -- SRP Violation**

```
interface Modem
{
    public void dial(String pno);
    public void hangup();
    public void send(char c);
    public char recv();
}
```

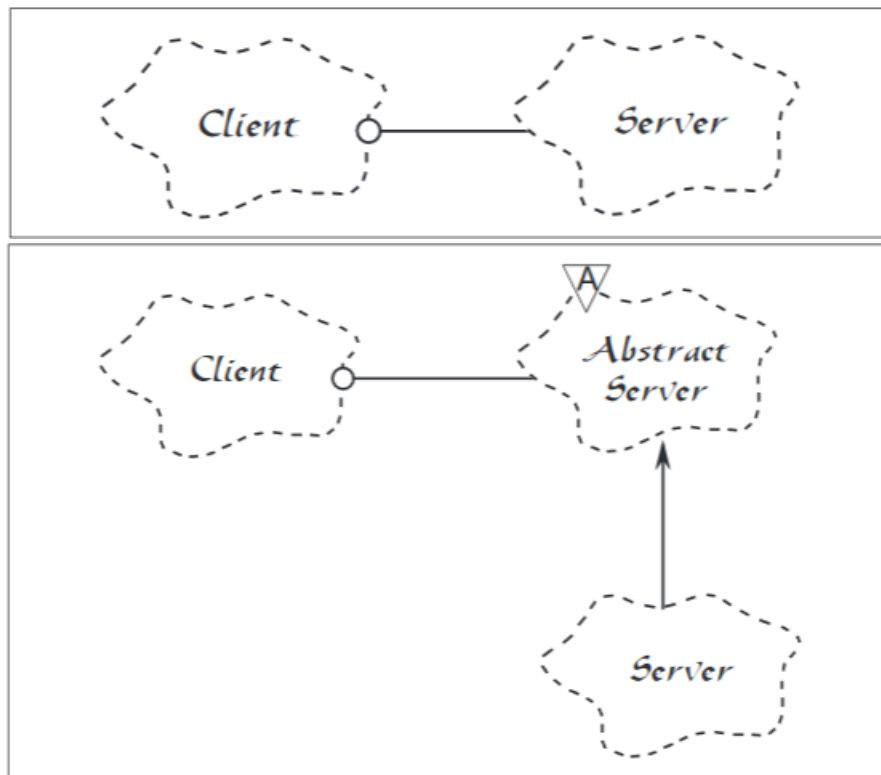


Figur 48: SRP violation

The Open-Closed Principle[6]

When a single change to a program results in a cascade of changes to dependent modules, that program exhibits the undesirable attributes that we have come to associate with "bad" design. The program becomes fragile, rigid, unpredictable and not reusable. The open-closed principle attacks this in a very straightforward way. It says that you should design modules that never change. When requirements change, you extend the behavior of such modules by adding new code, not by changing old code that already works. The **OCP** have two primary attributes:

- **1: They are "Open For Extension".** This means that the behavior of the module can be extended. That we can make the module behave in new and different ways as the requirements of the application change, or to meet the needs of new applications.
- **2: They are "Closed for Modification".** The source code of such a module is inviolate. No one is allowed to make source code changes to it.



Figur 49: OCP - Closed for extension at the top, open for extension at the bottom.

The arguments against public variables, when they could have been private: Bad modules can access them, and thus corrupt the behaviour of other modules. This violates the OCP. The same argument goes against global variables.

Liskov Substitution Principle[5]

In a computer program, if S is a subtype of T, then objects of type T may be replaced with objects of type S without altering any of the desirable properties of that program.

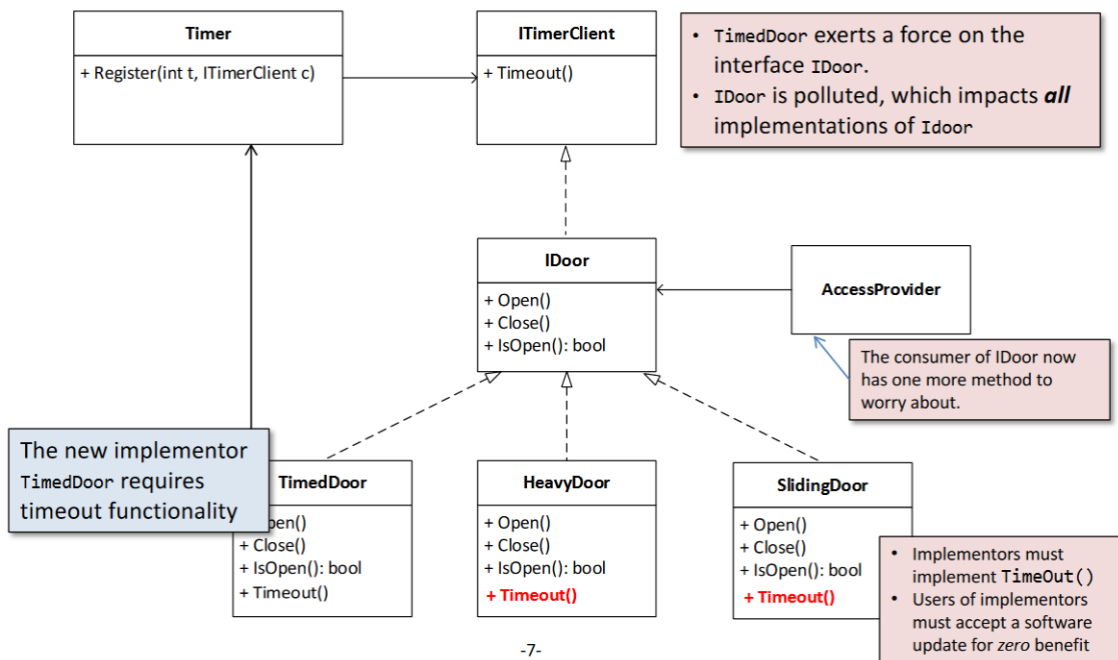
Any client of a class should be able to use subclasses of that class with no problems.

Interface Segregation Principle[4]

Interface pollution: The original interface has been polluted with an interface that it does not require. It has been forced to incorporate this interface solely for the benefit of one of its sub-classes. If this practice is pursued, then every time a derivative needs a new interface, that interface will be added to the base class. This will further pollute the interface of the base class, making it "fat".

CLIENTS SHOULD NOT BE FORCED TO DEPEND UPON INTERFACES THAT THEY DO NOT USE.

ISP VIOLATION (IMPLEMENTORS)



INTERFACE SEGREGATION (IMPLEMENTORS)

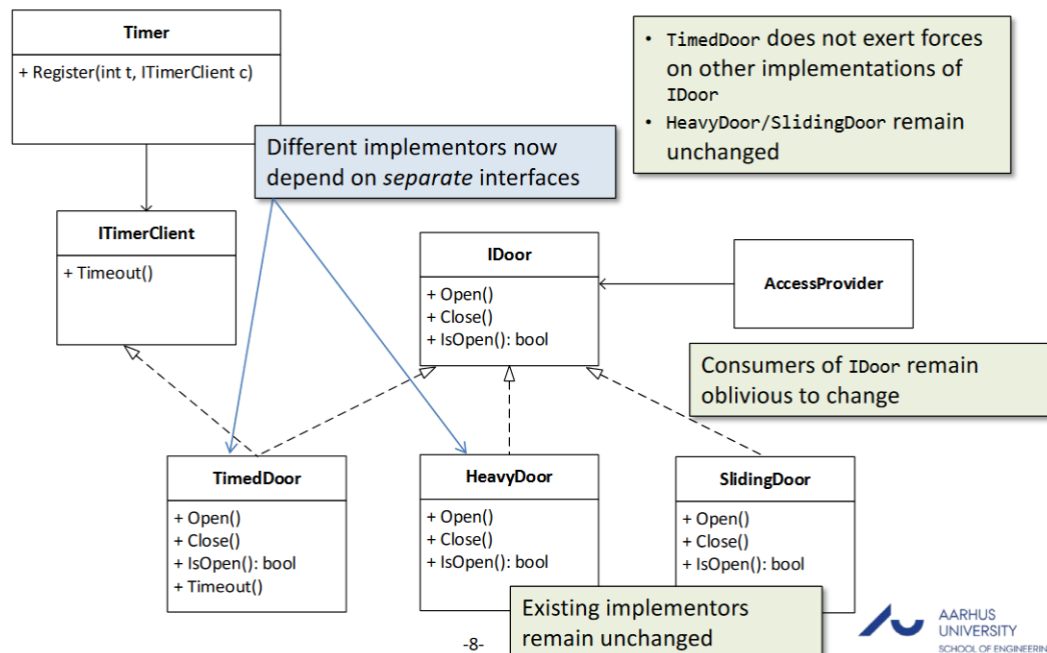
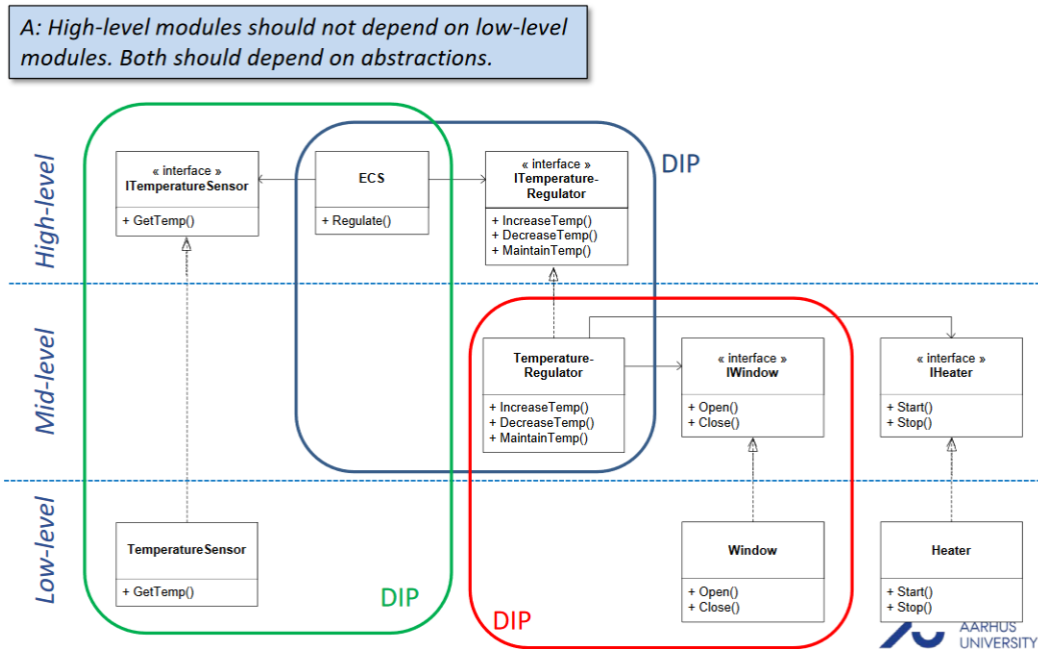


Figure 50: ISP Violations first, then ISP correct.

Dependency Inversion Principle[3]

- **1:** High-level modules should not depend on low-level modules. Both should depend on abstractions.
- **2:** Abstractions should not depend on details. Details should depend on abstractions.



Figur 51: DIP.

Figurer

| | | |
|----|--|----|
| 1 | Example of the template method pattern, a turn-based game. The template here determines the order in which the "variant" parts are called. | 4 |
| 2 | Class diagram for template method pattern. | 5 |
| 3 | The Context class does not implement an algorithm directly. Instead Context refers to the Strategy interface for performing an algorithm, which makes Context independent of how an algorithm is implemented. The Strategy1 and Strategy2 classes implement the Strategy interface, that is, implement (encapsulate) an algorithm. | 5 |
| 4 | Comparison of inheritance(left) vs delegation(right) | 6 |
| 5 | Top variant: pushing changes. Bottom variant: pulling changes. The problem with both of these, is that the objects are dependant upon one another. We need a pattern that allows for decoupling. | 7 |
| 6 | Observer pattern: Defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. | 8 |
| 7 | Top: pull variant. Bottom: push variant | 9 |
| 8 | Handling subjects of different types | 10 |
| 9 | Factory Method for creating a document, implementing the CreatePages(). | 11 |
| 10 | Creating the weighing system without a factory pattern. Complex and error prone! | 12 |
| 11 | Creating the weighing system with an abstract factory. | 13 |
| 12 | The statemachine pattern for a context with two different states | 15 |
| 13 | Nested states. | 16 |
| 14 | Orthogonal States. | 16 |
| 16 | MVC | 18 |
| 17 | Model-View-Presenter - Supervising | 19 |
| 18 | Model-View-Presenter - Supervising | 20 |
| 19 | Model-View-Presenter - Passive | 20 |
| 21 | ViewModel exposing relevant property for the View to bind to. | 22 |
| 22 | N-Layer | 23 |
| 23 | Mapping MVC/MVP to 3-Layer Architecture | 24 |
| 24 | Aggregation, doing serial estimation of PI | 25 |
| 25 | The aggregation part of the estimation of PI has been parallelized. | 25 |
| 26 | A delegate passed to Parallel.For that represents the initialization and finalization routine to be run on each task. | 26 |
| 27 | localInit at the beginning of the loop. The loop is all the iterations, no need for locking here as it runs in same thread. For the localFinally locking is required. | 26 |
| 28 | | 27 |
| 29 | MapReduce example | 28 |
| 30 | DAG of the solution to be compiled | 29 |
| 31 | If building each component is represented as a Task , we can take advantage of continuations to express as much parallelism as possible | 29 |
| 32 | Top we see the iterations done sequentially, and below they are done in parallel | 30 |
| 33 | Task for iterating, and barrier to divide portions. | 31 |
| 34 | Example code is a body of a sequential method | 31 |
| 35 | | 32 |

| | | |
|----|--|----|
| 36 | Processing images sequentially. Top is without pipeline, and bottom is pipelined. | 32 |
| 37 | C# pipeline stage implementation, using tasks and concurrent queues. | 33 |
| 38 | | 33 |
| 39 | | 34 |
| 40 | | 34 |
| 41 | Architecture should cover not only the system needs, but also the user and business needs. . | 35 |
| 42 | Pipes and filters architectural style. | 37 |
| 43 | Message bus architectural style: | 37 |
| 44 | 4+1 View | 37 |
| 45 | Chain of Responsibility | 38 |
| 46 | Chain of Responsibility | 38 |
| 47 | Chain of Responsibility | 39 |
| 48 | SRP violation | 41 |
| 49 | OCP - Closed for extension at the top, open for extension at the bottom. | 42 |
| 50 | ISP Violations first, then ISP correct. | 43 |
| 51 | DIP. | 44 |

Tabeller

Referencer

- [1] Dofactory. Abstract Factory. URL: <http://www.dofactory.com/net/abstract-factory-design-pattern> (sidst set 28.05.2018).
- [2] Dofactory. Factory Method. URL: <http://www.dofactory.com/net/factory-method-design-pattern> (sidst set 28.05.2018).
- [3] Michael Fowler. Dependency Inversion Principle. URL: https://blackboard.au.dk/bbcswebdav/pid-1637610-dt-content-rid-3502801_1/courses/BB-Cou-UUVA-73293/dip.pdf (sidst set 28.05.2018).
- [4] Michael Fowler. Interface Segregation Principle. URL: https://blackboard.au.dk/bbcswebdav/pid-1637610-dt-content-rid-3502800_1/courses/BB-Cou-UUVA-73293/isp.pdf (sidst set 28.05.2018).
- [5] Michael Fowler. Liskov Substitution Principle. URL: https://blackboard.au.dk/bbcswebdav/pid-1637610-dt-content-rid-3502799_1/courses/BB-Cou-UUVA-73293/lsp.pdf (sidst set 28.05.2018).
- [6] Michael Fowler. The Open-Closed Principle. URL: https://blackboard.au.dk/bbcswebdav/pid-1637628-dt-content-rid-3502816_1/courses/BB-Cou-UUVA-73293/ocp.pdf (sidst set 28.05.2018).
- [7] Michael Fowler. The Single Responsibility Principle. URL: https://blackboard.au.dk/bbcswebdav/pid-1637628-dt-content-rid-3502814_1/courses/BB-Cou-UUVA-73293/srp.pdf (sidst set 28.05.2018).
- [8] Robert C Martin. Design Principles and Design Patterns. URL: https://blackboard.au.dk/bbcswebdav/pid-1637628-dt-content-rid-3502815_1/courses/BB-Cou-UUVA-73293/Principles_and_Patterns.pdf (sidst set 28.05.2018).
- [9] Poul Ejnar Røvsing. The Model-View-ViewModel Design Pattern. URL: https://blackboard.au.dk/bbcswebdav/pid-1637667-dt-content-rid-3502758_1/courses/BB-Cou-UUVA-73293/BB-Cou-UUVA-69277_ImportedContent_20170609021151/BB-Cou-UUVA-61888_ImportedContent_20160707090143/BB-Cou-STADS-UUVA-52348_ImportedContent_20151221103641/Papers/Model-View-ViewModel-Note.pdf (sidst set 28.05.2018).