UI Patterns and Frameworks | UIP

Zusammenfassung

INHALTSVERZEICHNIS

1. EINLEITUNG

1.1. VORTEILE VON GUIS

Dank GUIs kann auch die breite Öffentlichkeit Computer verwenden.

- ***Einfacher Einstieg:*** Wenig Lernaufwand, muss keine Befehle merken
- ***Übersicht:*** Ein GUI zeigt ganz einfach grafisch eine Übersicht über das Programm und seinen Funktionen an. Dadurch können auch Funktionen entdeckt werden, die man als Benutzer noch nicht kannte
- ***Keine Tippfehler:*** Im Vergleich zu CLIs kann man Klicken statt Tippen, was Fehler durch Vertippen ausschliesst
- ***Visuelles Feedback:*** Status, Fortschritt, Warnungen etc. sind sofort erkennbar

1.2. RAD / LOW-CODE ENTWICKLUNG

Mit Low-Code Entwicklung und Rapid Application Development *(RAD)* können sehr schnell GUIs zusammengeklickt werden. Dieser Ansatz führt jedoch langfristig zu vielen Problemen wie ***Abhängigkeit*** *(Vendor Lock-in)*, ***Wartbarkeit*** *(Fehlende Vorgabe von Separation of Concerns)*, ***Nachvollziehbarkeit*** *(Riesige Vererbungshierarchien der UI Controls)* und ***Testing*** *(Gibt es nicht)*. **RAD ist eine Zeitbombe:** Weiterentwicklung wird unmöglich und ein Neuanfang ist finanziell unrealistisch.

1.3. SCHLECHTER CODE

Lesen und Verstehen von Code ist der grosse Teil der Arbeit. Das lokalisieren von Bugs ist bei schwierig lesbarem Code sehr aufwändig. Das führt zu ***hohen Kosten für die Weiterentwicklung***. Es ist bei schlechtem Code fast unmöglich, kleine Einheiten zu ändern ohne dabei anderen, funktionierenden Code nicht anzutasten. Durch ***schlechte Testbarkeit*** schleichen sich bei der Weiterentwicklung neue Fehler ein.

1.4. LANGLEBIGKEIT VON USER INTERFACES

Die Langlebigkeit von User Interfaces wird beeinflusst durch:

- ***Nachvollziehbarkeit & Wartbarkeit:*** Durch Unabhängigkeit der Geschäftslogik von der Darstellung, Testbarkeit und Lesbarkeit wird die Nachvollziehbarkeit und Wartbarkeit sichergestellt.
- ***Langfristige Unabhängigkeit:*** Abhängigkeit von Herstellern vermindern, Plattformunabhängigkeit, Abtrennung von UI Technologien
- ***Austauschbarkeit des UIs:*** Verschiedene Darstellungen anbieten *(Mobile/Desktop UIs)*, verschiedene Arten von Benutzerinteraktion anbieten, Darstellung und Interaktion austauschen, ohne die Geschäftslogik anzutasten.
- ***Nicht alles neu erfinden:*** Es gibt Muster (Patterns) und Prinzipien, die sich bewährt haben, Konzepte, die vieles vereinfachen und Gerüste (Frameworks) für unsere Applikationen.

2. DESIGN PRINZIPIEN

Software Design Prinzipien sind ***Regeln*** und ***Grundsätze***, die bei der Entwicklung bedacht werden sollten und helfen, systematische Probleme zu lösen. Sie unterstützen dabei, ***Qualitätseigenschaften*** der Software wie Wartbarkeit, Erweiterbarkeit und Stabilität zu erreichen. Dabei müssen sie aber nicht stur eingesetzt werden, sondern nur wo sinnvoll.

2.1. KISS (KEEP IT SIMPLE, STUPID)

Systeme und Lösungen sollten ***so einfach wie möglich*** gehalten werden, da Komplexität zu schwer wartbarem und fehleranfälligem Code führt. Sollte aber nie ein Vorwand sein, ein UI nicht von Geschäftslogik zu trennen.
Ziel: Einfachheit eines Designs beibehalten, um die Verständlichkeit und Wartbarkeit zu fördern. ***Vorgehen:*** Unnötige Funktionen oder komplizierte Architektur vermeiden. Klare, nachvollziehbare Lösungen verwenden.

2.2. YAGNI (YOU AIN'T GONNA NEED IT)

Funktionen und Features, die aktuell ***nicht benötigt*** werden, sollten ***nicht implementiert*** werden. Die Entwickler sollten sich auf die ***unmittelbaren Anforderungen*** konzentrieren. Saubere Architektur ist aber trotzdem ein Muss.
Ziel: Ressourcenverschwendung und unnötige Komplexität vermeiden.
Vorgehen: Nur das implementieren, was aktuell benötigt wird. Zusätzliche Features auf den Zeitpunkt verschieben, an dem sie gebraucht werden.

2.3. DRY (DON'T REPEAT YOURSELF)

Jede Information innerhalb eines Systems sollte ***nur einmal existieren***. Duplikation führt zu Inkonsistenzen und erschweren die Wartung, da Änderungen an mehreren Stellen durchgeführt werden müssen. ***Ziel:*** Reduzierung von Redundanz und Erhöhung der Wartbarkeit. ***Vorgehen:*** Gemeinsame Logik oder Daten an einem zentralen Ort speichern und wiederverwenden, anstatt zu duplizieren.

2.4. LAW OF DEMETER (DON'T TALK TO STRANGERS)

Objekte sollten nur mit ***eng verwandten*** Objekten kommunizieren. Ein Objekt sollte nicht direkt auf die inneren Details anderer Objekte zugreifen, sondern nur mit den unmittelbaren Nachbarn interagieren.
Ziel: Reduzierung der Abhängigkeiten zwischen Klassen und Verbesserung der Kapselung. ***Vorgehen:*** Verkettete Methodenaufrufe vermeiden *(`car.getMotor().start()`)*. Jede Klasse sollte nur Methoden von Objekten aufrufen, die sie direkt kennt *(z.B. Wrapper-Methode `car.start()`, welche `motor.start()` callt)*.

2.5. KOMPOSITION VOR VERERBUNG

Vererbungen kreieren ***statische Abhängigkeiten***: Die Child-Klasse ist komplett abhängig vom Parent, sie kann ohne sie nicht kompiliert werden. Änderungen im Parent betreffen auch alle Children. Häufig benötigt eine Klasse ***nicht alle Eigenschaften*** seines Parents. Deswegen wird Komposition *(«hat»-Beziehung)* vorgezogen: Entweder durch Interfaces oder Kombination von Objekten zur Laufzeit *(eine Instanz der «Child-Klasse» ist eine Klassenvariabel im Parent)*.
Ziel: Vermeidung starrer, hierarchischer Beziehungen und Förderung von lose gekoppelten, flexiblen Systemen.
Vorgehen: Anstatt eine Unterklasse von einer Basisklasse abzuleiten, sollten Interfaces definiert werden, welche sich auf die Fähigkeiten eines Objekts beziehen. Sehr oft ist eine «hat»-Beziehung (Komposition) die richtige Lösung.

2.6. S.O.L.I.D. PRINZIPIEN

Single Responsibility (SRP): Es sollte nie mehr als einen Grund dafür geben, eine Einheit zu ändern. Jede Klasse oder Komponente sollte also nur ***eine Verantwortung*** haben. ***Ziel:*** Verbesserung der Verständlichkeit und Wartbarkeit durch klare Trennung der Verantwortlichkeiten.

Open / Closed Principle (OCP): Einheiten sollten sowohl ***offen für Erweiterungen*** als auch ***geschlossen für Modifikationen*** sein. Das bedeutet, dass bestehender Code nicht verändert, sondern ergänzt werden sollte, wenn neue Funktionen hinzugefügt werden. ***Ziel:*** Reduzierung der Änderungen an bestehendem Code und Erleichterung der Erweiterbarkeit des Systems.

Liskov Substitution Principle (LSP): ***Subtypen*** sollten ohne Probleme überall ***eingesetzt*** werden können, wo der ***Basistyp*** erwartet wird. Ein Objekt einer abgeleiteten Klasse muss sich also wie ein Objekt der Basisklasse verhalten.
Ziel: Sicherstellen, dass der Code mit Unterklassen genauso funktioniert wie mit der Basisklasse. ***Beispiel:*** Eine `Pinguin`-Klasse, die von einer `Vogel`-Klasse erbt, sollte keine `fly()`-Methode haben.

Interface Segregation Principle (ISP): Clients sollten nicht dazu gezwungen werden, von Interfaces abzuhängen, die sie nicht verwenden. Statt grosse, umfassende Interfaces zu erstellen, sollten diese in kleinere, spezifischere Interfaces aufgeteilt werden. ***Ziel:*** Erhöht Flexibilität und Lesbarkeit, da jede Klasse nur die Methoden implementieren muss, die sie tatsächlich benötigt.

Dependency Inversion Principle (DIP): High-Level-Module sollen nicht von Modulen niedriger Ebene abhängen. Beide sollen von Abstraktionen (Interfaces) abhängen. ***Ziel:*** Reduzierung der Kopplung zwischen Modulen und Erhöhung der Flexibilität.

ISP und DIP treten selten alleine auf, sie sind oft Ursache oder Folge anderer Prinzipien. ISP tritt oft mit LSP auf.

2.6.1. Übersicht

	Weg	***Ziel***
SRP	Trennung und Fokus der Komponenten auf jeweils eine Aufgabe.	***Klarheit:*** Aufgabe der Komponente, ***Lokalisierbarkeit*** eines Fehlers
OCP	Kernlogik kapseln und Erweiterungen durch Schnittstellen ermöglichen.	***Stabilität:*** keine Änderung bestehender Logik, ***Erweiterbarkeit*** durch passende neue Komponenten
LSP	Erwartungen an Schnittstellen vollständig erfüllen.	***Austauschbarkeit:*** Implementationen, ***Robustheit:*** keine unerwarteten Ereignisse
ISP	Schnittstellen auf den minimalen Verwendungszweck reduzieren.	***Spezifität*** für die verwendete Komponente, ***Wiederverwendbarkeit*** der kleinen Schnittstelle
DIP	Feste Kopplung vermeiden.	***Unabhängigkeit:*** Abhängigkeiten austauschbar, ***Testbarkeit:*** Abhängigkeiten durch «Testdummies» ersetzbar

3. DESIGN PATTERNS

Design Patterns sind Beschreibungen ***erfolgreicher Strukturen*** von Software für ***bestimmte Problemstellungen***. Sie adressieren ***wiederkehrende Probleme*** und beschreiben ***generische Lösungen***, die funktionieren. Sie bieten eine gemeinsame Sprache für Experten.

3.1. ARCHITECTURAL PATTERNS

Architekturmuster sind ***wiederverwendbare*** Lösungen für ***häufige Probleme*** der Softwarearchitektur. Sie behandeln verschiedene Themen der Softwaretechnik, wie zum Beispiel die Leistungsgrenze von Computerhardware, hohe Verfügbarkeit und die Minimierung von Geschäftsrisiken. Oft bauen Architekturpattern auf einer ***Kombination von mehreren Design Patterns*** auf.

3.2. GOF DESIGN PATTERNS

Gang of Four (GoF): Buch über Design Patterns von 1994 von Erich Gamma, Richard Helm, Ralph Johnson und John Vlissides.

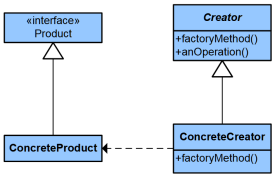
Creational Patterns (blau): Patterns für die ***Erstellung von Objekten***, welche die Wiederverwendbarkeit des Codes verbessern.

Structural Patterns (orange): Patterns, um Objekte in grösseren Strukturen flexibel und effizient anzuordnen.

Behavioral Patterns (grün): Patterns für Algorithmen und Zuteilung der Zuständigkeiten zwischen Objekten. Sie sind ***Micro Frameworks*** und damit Grundlage für Frameworks.

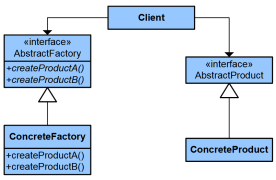
Factory Method (nicht verwenden)

Absicht: Definiert eine Erzeugungsschnittstelle, aber Unterklassen entscheiden, welche konkrete Klasse instanziiert wird. ***Problem:*** `new()` koppelt Code hart an konkrete Klassen.
Lösung: Objekterzeugung in eine Factory Method auslagern. ***Motivation:*** Flexible Instanziierung *(z. B. anhand Laufzeitdaten)*.



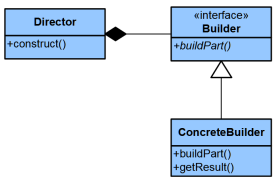
Abstract Factory

Absicht: Erzeugt passende Objektfamilien ohne konkrete Klassen. ***Problem:*** Konkrete Klassen binden den Code; Kombinationen werden inkonsistent. ***Lösung:*** Abstract Factory + pro Familie eine Concrete Factory. ***Motivation:*** Produktfamilien per Factory-Austausch wechseln.



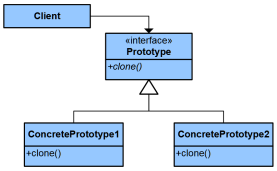
Builder

Absicht: Baut Objekte schrittweise; gleicher Ablauf erzeugt Varianten. ***Problem:*** Viele Optionen führen zu fetten Konstruktoren oder zu vielen Subklassen. ***Lösung:*** Bauprozess in Builder mit klaren Schritten auslagern. ***Motivation:*** Praktisch fürs Bootstrapping eines DI-Containers.



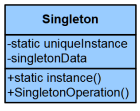
Prototype

Absicht: Kopiert Objekte ohne Abhängigkeit vom konkreten Typ. ***Problem:*** Neuerstellung ist teuer oder konkrete Klassen sollen nicht genannt werden. ***Lösung:*** `clone()`-Methode bereitstellen. ***Motivation:*** Ähnliche Objekte klonen und anpassen statt neu bauen. Meist durch Programmiersprache schon nativ implementiert.



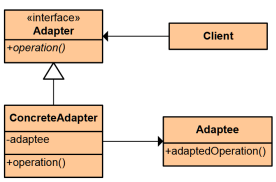
Singleton (nicht verwenden)

Absicht: Genau eine Instanz + globaler Zugriff. ***Problem:*** Mehrere Instanzen machen Shared-Resources unübersichtlich. ***Lösung:*** Konstruktor privat, `getInstance()` liefert immer dieselbe Instanz. ***Motivation:*** Zentraler, kontrollierter Zugriff; aber schlecht testbar wegen globalem Zustand, deshalb besser via DI.



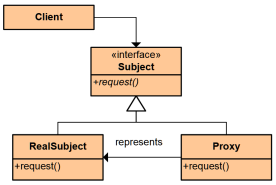
Adapter

Absicht: Macht inkompatible Schnittstellen kompatibel. ***Problem:*** Bestehende Klasse passt nicht zur erwarteten API. ***Lösung:*** Adapter kapselt das Objekt und übersetzt Aufrufe.
Motivation: Interface-Anpassung & Entkopplung von Frameworkcode.



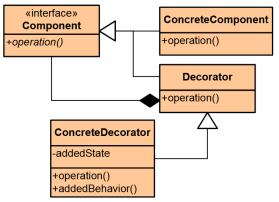
Proxy

Absicht: Stellvertreter kontrolliert Zugriff und ergänzt Logik. ***Problem:*** Objekt ist teuer/langsam oder Zugriff muss geregelt werden.
Lösung: Proxy mit gleicher Schnittstelle *(z.B. Lazy Loasding, Cache, Checks)* delegiert ans echte Objekt.
Motivation: z.B. für Benachrichtigungen.



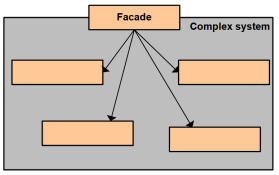
Decorator

Absicht: Fügt einem Objekt zusätzliches Verhalten hinzu, ohne die Klasse zu ändern.
Problem: Funktion soll erweitert werden, ohne die Klasse zu ändern. ***Lösung:*** Anlegen eines Decorator um ein Objekt um Aufrufe weiterzuleiten und zu ergänzen. ***Motivation:*** Anordnung von UI Elementen.



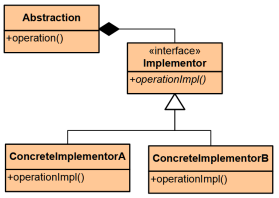
Facade

Absicht: Vereinfacht den Zugriff auf ein komplexes Subsystem. ***Problem:*** Client kennt zu viele Details/Abhängigkeiten, wird unübersichtlich, stark gekoppelt. ***Lösung:*** Facade kapselt Komplexität und bietet nur nötige Methoden.
Motivation: Unterstützt ISP durch klare, schlanke Schnittstellen.



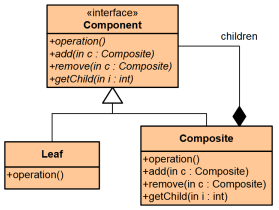
Bridge

Absicht: Trennt Abstraktion von Implementierung, beide variieren unabhängig.
Problem: Viele Variantendimensionen führen zu Klassenexplosion. ***Lösung:*** Abstraktion hält Implementor-Interface und delegiert.
Motivation: Abstraktion/Implementierung erweitern ohne gegenseitige Änderungen.



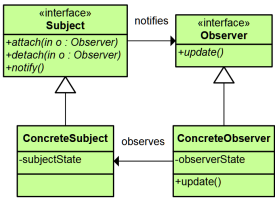
Composite

Absicht: Baut Objektbäume und behandelt Gruppen wie Einzelobjekte. ***Problem:*** Client muss ständig zwischen Element und Gruppe unterscheiden, wird komplex. ***Lösung:*** Gemeinsames Component-Interface: `Leaf` und `Composite` implementieren es, Composite verwaltet Kinder und delegiert. ***Motivation:*** Anordnung von UI Elementen.



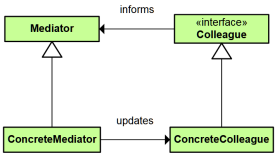
Observer

Absicht: Benachrichtigt mehrere Abonnenten bei Änderungen. ***Problem:*** Polling oder starke Kopplung der Abhängigen. ***Lösung:*** Observer registrieren und bei Änderungen automatisch informieren *(1:n Beziehung)*. ***Motivation:*** Zentrale Verteilung; neue Beobachter ohne Logikänderung. *(Subject: Sich ändernde Quelle, Observer: der Zuhörer)*



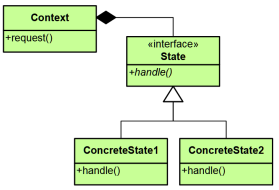
Mediator

Absicht: Reduziert direkte, chaotische Abhängigkeiten durch Kommunikation über einen Vermittler. ***Problem:*** Viele Direktverbindungen führen zu Spaghetti-Kopplung, schwer wartbar.
Lösung: Kommunikation über Mediator bündeln statt direkt *(m:1:n Beziehung)*. ***Motivation:*** Messaging-/Benachrichtigungsmechanismen.



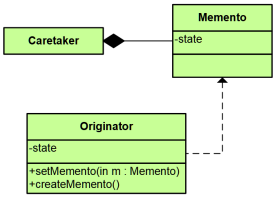
State

Absicht: Verhalten wechselt je nach internem Zustand. ***Problem:*** Viele Zustände führen zu langen `if/else`-Ketten im Objekt. ***Lösung:*** Zustände werden als eigene State-Klassen modelliert.
Motivation: Zustandslogik wird getrennt. So können Zustände ergänzt werden, ohne die bestehenden Bedingungen aufzublähen.



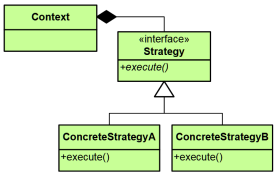
Memento

Absicht: Ein Objektzustand wird gespeichert und wiederhergestellt, ohne Implementierungsdetails offenzulegen. ***Problem:*** Undo/Restore sollte möglich sein, ohne die Kapselung zu verletzen.
Lösung: Ein Memento *(Snapshot)* hält den Zustand, ein Caretaker verwaltet die Mementos. ***Motivation:*** Rücksprünge werden ermöglicht, während interne Details verborgen bleiben.



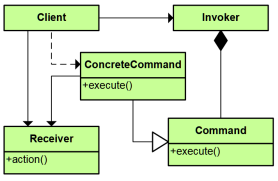
Strategy

Absicht: Kapselt austauschbare Algorithmen.
Problem: Algorithmus-Varianten führen zu verzweigtem Code und starker Koppelung.
Lösung: Jede Variante als Strategy, zur Laufzeit wählen. ***Motivation:*** für Kapselung und Austauschbarkeit von Logik.



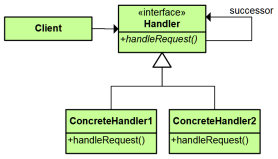
Command

Absicht: Eine Anfrage wird als eigenständiges Objekt mit allen nötigen Infos verpackt.
Problem: Aktionen sollen später, in einer Queue oder mit Undo ausführbar sein.
Lösung: Ein Command kapselt Aktionen und wird vom Aufrufer entkoppelt. ***Motivation:*** für Kapselung und Austauschbarkeit von Logik.



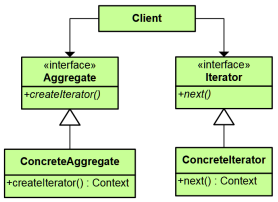
Chain of Responsibility

Absicht: Anfragen werden durch eine Kette von Handlern weitergereicht, bis einer sie verarbeitet. ***Problem:*** Ein fester Empfänger erzeugt starre Logik und Sonderfälle. ***Lösung:*** Jeder Handler entscheidet, ob er die Anfrage bearbeitet oder weiterleitet. ***Motivation:*** Für Middleware



Iterator

Absicht: Durchläuft eine Sammlung ohne interne Details offenzulegen. ***Problem:*** Traversierung hängt stark von der konkreten Datenstruktur ab. ***Lösung:*** Ein Iterator-Interface kapselt das Durchlaufen. ***Motivation:*** Einheitliche Traversierung verschiedener Collections, Implementierungsdetails bleiben verborgen *(Wie in C++)*.



4. FRAMEWORKS

Ein Framework nimmt dem Applikationsentwickler ***Standardaufgaben*** ab *(Instanziierung, Auswahl der Komponenten, Request Handling, Event Handling, ...)*. Es ist ein ***Gerüst*** für den Applikationscode und funktioniert nach dem ***Hollywood Prinzip/ Inversion of Control***. Es baut auf ***Structural Design Patterns*** und / oder ***Meta Programmierung*** *(z.B Reflection, Code Generation, Interpreter)* auf.
Ein Framework bietet Strukturierung durch ***Modularität***, ***Kapselung***, ***Layering*** und ***Abstraktionen***. Es bietet dafür ***Control Flow***, ***Hooks*** und ***Extension Points***.

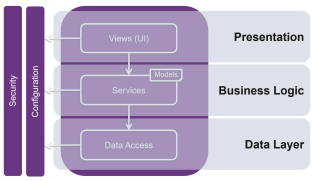
Gefahren von Frameworks: Starke Kopplung durch eigene Subklassen von Framework-Klassen, ***Schulungsaufwand***, ***Verminderte Testbarkeit*** *(Adapter verwenden, DI und Integration Tests nutzen)*, ***Angst vor Updates***, ***Vendor Lock-In / Abhängigkeit*** *(offene Standards nutzen, restriktive Vertragsklauseln meiden, eigene Interfaces nutzen, Exit-Plan dokumentieren)*, schwierige Migration einzelner Komponenten.
Ein Framework ist gut, wenn: Der eigene Code möglichst ***wenig Abhängigkeiten*** zum Framework hat, das Framework so viel ***Standardaufgaben*** wie möglich ***übernimmt***, ohne dass die Konfiguration zu ***aufwändig*** wird.
Framework Entwicklungs-Schwierigkeiten: Verbesserungen müssen vorgenommen werden, ***ohne*** dass sich die ***bestehende API ändert***. Es sollte keine ***Breaking Changes*** geben. ***Web Standards*** können ein Segen oder eine Bremse für Frameworks sein: Passe ich mich an oder mache ich etwas eigenes? Wann zu einem neuen Standard wechseln? Frameworks haben oft viele ***Abhängigkeiten***, was alles komplizierter macht. ***Dilemma:*** Flexibilität vs. Einfachheit, Innovation vs. Stabilität, Anpassbarkeit vs. Meinungsvorgaben, Rückwärtskompabilität vs. Modernisierung, Funktionsumfang vs. Leistung.
Inversion of Control (Hollywood Prinzip): Anfragen können durch das Framework bearbeitet und an den richtigen Applikationscode weitergeleitet werden. Das Framework ruft also Applikationscode auf. Dies nennt man «Inversion of Control». ***Hollywood Prinzip:*** Don't Call Us - We will call you! Hollywood ruft dich an, um dich zu engagieren, nicht umgekehrt.

Unterschied Framework zu Library und Runtime:
Library: Die Funktionalität der Bibliothek wird aus dem Applikationscode aufgerufen (Kein Hollywood Prinzip). Beim Verwenden einer Library wird der Applikationscode ***abhängig*** von der Library. Businessrelevante Libraries sollten wie ***Infrastruktur Code*** behandelt werden und in eine Komponente ausserhalb der Business Layer ***abgekoppelt*** werden. Dies gelingt mit ***Interfaces***, welche die Library kapseln.
Runtime: Ist ein Abstraktionslayer über das darunter liegende Betriebssystem. Kümmert sich um Interpretation des Codes, Speicherverwaltung und Schnittstellen. Oft liefert eine Runtime gleich auch ein Framework mit, welches die API der Runtime nutzt und ein Gerüst für den Applikationscode bietet. Frameworks können Inversion of Control und Dependency Injection für bessere Seperation of Concerns und Testing nutzen.

4.1. INSTANZIIERUNG VON OBJEKTEN

Objekte sollten nur erzeugt werden, wenn sie auch gebraucht werden. Die gleichen Instanzen müssen an verschiedenen Orten verwendet werden können. Hier sind ***Dependency Inversion Principle*** und ***Interface Segregation*** zu beachten. **Aufteilung nach Dependency Inversion Principle:** ***Geschäftslogik*** sollte so weit wie möglich von ***Technologie*** getrennt sein. Es werden also ***Abstraktionen / Interfaces*** eingeführt, welche von den Komponenten abhängen.

Layer Architektur: Im einfachsten Fall entsteht durch die Separierung eine Layer-Architektur. Die obere Schicht kommuniziert nur über Interfaces mit der unteren.
Crosscutting Concerns: In einer Layer Architektur kommt es oft zu Crosscutting Concerns *(Security Context, Konfiguration)*, die in mehreren Layern gebraucht werden.



Wie können diese Objekte in einer ***ganzen Applikation verfügbar*** gemacht werden, ***ohne*** die Implementation ***global*** zugänglich zu machen? Diese ***Vertical Layer*** werden beim ***Bootstrapping*** in den Application Context enkapsuliert, welcher den einzelnen Layern übergeben wird.
Bootstrapping: Der Prozess, in dem während des Systemstarts alle nötigen Komponenten und Abhängigkeiten geladen werden. Einstiegspunkt zwischen Framework und Applikationscode. Stellt Service Builder & Provider bereit.

4.1.1. Service Provider

Das Framework definiert applikationsweite ***Handler*** *(z.B. Router, Navigator, Command oder Strategy Pattern)*. Dieser Handler kann ***Anfragen*** entgegennehmen oder auf ***Zustandsänderungen*** reagieren. Über eine ***Konfiguration*** wird definiert, bei welcher Anfrage welches Applikationsobjekt für die Bearbeitung des Befehls zuständig sein soll. Diese Anfrage wird durch den Handler an einen ***Service Provider*** oder einen ***Dependency Injection Container*** *(«Dependency Injection Container» (Seite 5))* weitergeleitet. Der ***Service Provider*** ist eine konfigurierbare ***Factory***, die Applikationsobjekte und Abhängigkeiten dynamisch auflöst. Er ist eine ***zentrale Komponente*** in einem Framework und für die Verwaltung und Bereitstellung von Diensten zuständig. Wird im Bootstrapper ***separiert*** vom restlichen Applikationscode konfiguriert und ist dem Applikationscode möglichst ***unbekannt***.

4.1.2. Service Builder

Ist ein ***dynamischer Builder*** *(GOF Pattern)* für einen Service Provider oder DI-Container. Er arbeitet eng mit dem Service Provider zusammen, um die benötigten Dienste zu konfigurieren. Er ist für das ***Bootstrapping*** des Service Providers zuständig. In TypeScript gibt es zwei Service-Arten: ***Transient*** *(returnt jedes Mal neue Instanz des Service)* und ***Singleton*** *(returnt diesselbe Instanz beim Aufruf)*

4.2. MIDDLEWARE

Middleware ist eine ***konfigurierbare*** Komponente in einer Pipeline. Sie kann ***Anfragen*** und Antworten vor und nach der Weitergabe ***verändern***, ***erweitern*** oder ***blockieren***.
- ***Anfragebearbeitung*** *(Authentifizierung, Autorisierung, Validierung, Fehlerbehandl.)*
- ***Protokollierung, Monitoring:*** *(Erfassen von Request/Response, Monitoring)*
- ***Datenumwandlung:*** *(Konvertierung von Formaten, (De-)serialisierung)*

Wird oft ***zwischen*** Handler des Frameworks und Applikationscode eingeschleust *(Data Modification)* oder um einen Aufruf ***herum gestülpt*** *(Error Handling)*. Damit wird das ***Framework*** und nicht unser Code ***erweitert***. Damit fungiert es als ***Extension Point*** eines Frameworks *(Abwandlung von Chain of Responsibility Pattern)*. Es können darin ***Decorators*** und ***Proxies*** aneinandergekettet werden, um Anfragen und Antworten zu modifizieren.

4.2.1. Middleware-Pipeline

Pipeline-Struktur: Jede Middleware kann ***vor*** und ***nach*** dem nächsten Schritt eingreifen. Durch `next()` wird entschieden, ob der Request ***weitergeleitet*** wird. Die ***Reihenfolge*** in der Pipeline beeinflusst die ***Verarbeitung***. Durch die ***Kettenreihenfolge*** wird hinzufügen, entfernen und umordnen sehr ***flexibel***. Jede ***Middleware*** übernimmt ***eine spezifische Aufgabe***. Z.B.:
`Logging → Auth. → Autorisierung → Fehlerbehandlung → Endpunkt`

```
export const log = <TServices>(s: IServiceProvider<TServices>) ⇒
  async (input: RenderResult, next: (result: RenderResult) ⇒
  Promise<RenderResult> ) ⇒ {
    console.log(`logging middleware received`, input);
    return await next(input); };
```

4.3. META PROGRAMMING & REFLECTION

Introspection: Fähigkeit eines Programms, seinen ***eigenen Zustand*** zu ***beobachten*** und ***analysieren*** *(Abfragen von Objekteigenschaften, Liste von Methoden)*.
Intercession: Fähigkeit eines Programms, seinen eigenen ***Ausführungszustand*** zu ***verändern*** oder seine ***Interpretation anzupassen*** *(Attribut hinzufügen)*.
Type Reflection: Fähigkeit, ***zur Laufzeit*** Informationen über die ***Typen*** von Variablen, Objekten etc. zu erhalten *(`typeof`, `instanceof`, `constructor`)*.
Structural Reflection: Fähigkeit, ***zur Laufzeit*** Informationen über die ***Struktur*** von Objekten und Funktionen zur erhalten und möglicherweise zu verändern *(`Object.keys()`, `Object.getOwnPropertyNames()`)*.

Annotation / Attributierung: Code wird ***zur Kompilierzeit*** mit Metadaten angereichert, welche zur Laufzeit über ***Reflections*** ausgelesen werden können.

```
class MyClass {                          new MyClass().logMe("test", 42);
  @Log // ← Annotation                   // Output:
  logMe(a1: string, a2: number) {        // logme Args: ["test",42]
    console.log("Running..."); }}        // Running...

function Log(target: any, key: string, descr: PropertyDescriptor) {
  const originalFunction = descr.value;
  descr.value = function (...args: any[]) {
    console.log(`${key} Args: ${JSON.stringify(args)}`);
    return originalFunction.apply(this, args); }; }
```

Nachteile Annotation: Müssen ***direkt auf Anwendungscode*** platziert werden → ***statische Abhängigkeit zum Framework***. Die ***Nachvollziehbarkeit*** leidet, weil «Magie»/«Framework-Fuckery» passiert.

Type Reflection in C#: ***Abfragen von Informationen*** zu einer Klasse, ***ohne*** eine ***Instanz*** zu erzeugen. ***Dynamisches Nachladen*** von Assemblies, ***Suche*** nach Typen, Interfaces usw., ***Erzeugen von generischen*** Typen.
Code Reflection in C#: ***LINQ*** Expressions, ***dynamische*** Kompilierung, ***dynamische*** Erstellung von Lambda-Ausdrücken.

4.4. DEPENDENCY INJECTION CONTAINER

Ist ein ***Service Provider*** mit einem «intelligenten» ***Service Builder***, der Abhängigkeiten selbst auflöst. Dies wird meist mit ***Reflection*** gelöst. Die ***manuelle Konfiguration*** mit dem Service Builder wird dadurch ***minimiert***.
Weitere Fähigkeiten: ***Ressourcenverwaltung*** *(Post-Construct, Pre-Destroy)*, ***Fehlerbehandlung***, Behandlung ***zirkulärer*** Referenzen, komplexe Scopes, Container Hierarchien, Configuration-/Parameterinjection, Lazy Injection, Interception/Middleware, Multiinjection *(Collections von Objekten als Konstruktionsparameter)*.

In ***C# oder Java*** sind Type Reflections mächtiger, was dem DI-Container erlaubt, ***Typen ohne Attribute zu registrieren***. C# kann Typen nach bestimmten Kriterien ***suchen*** und Objekte alleine anhand der Typeninformationen instanziieren *(Constructor enthält Interfaces, welche .NET automatisch ergänzt)*. Der Anwendungscode ist somit ***nicht vom Framework abhängig***.

5. SEPARATED PRESENTATION

Code, der die Darstellung manipuliert, sollte ***nur*** das tun. Oft wird ein UI weiter in ***Präsentation*** *(Inhalt, verschiedene Ausgabegeräte, Medien, Layouts, Animationen)*, ***Applikationslogik*** *(Applikations-Zustand, Prozesse, die durch das UI ausgelöst werden. Sollte nichts über das UI wissen.)* und ***Interaktionslogik*** *(verschiedene Eingabegeräte, Eventhandling, Navigation)* geteilt. ***Diese Abtrennung ist die Grundlage aller UI-Architekturpatterns***.

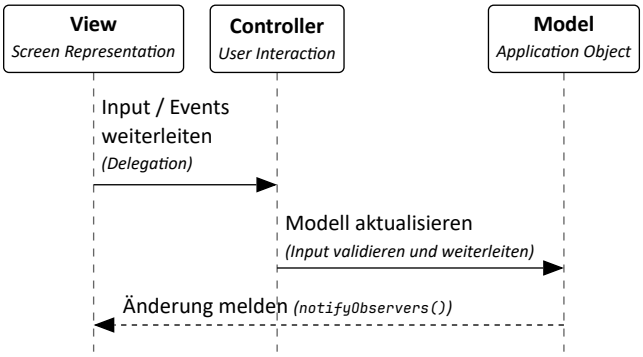
Vorteile: ***Nachvollziehbarkeit*** und Testbarkeit, ***Wiederverwendbarkeit***, ***Migrationsfähigkeit*** & ***Austauschbarkeit*** der Präsentation, alternative Eingabe wird erleichtert, Fokus der Entwickler im Team.

Für UI-Updates wird oft das ***Observer Pattern*** angewandt. **Vorteil:** Stellt View & Model-Synchronisationi sicher, keine statische Abhängigkeit von Model & View. **Nachteil:** Kontrollfluss wird verborgen.

Imperative Programmierung: Beschreibt die ***Umsetzungsschritte***, ist in Business-Logik und komplexen Prozessen verbreitet *(Bspw. Template Engines und Frameworks)*.
Deklarative Programmierung: Beschreibt das gewünschte ***Resultat***, ermöglicht intuitivere UI-Definitionen *(Bspw. HTML/CSS statt JS)*. ***Trennung*** von ***Präsentations- und Geschäftslogik*** wird einfacher.
Markup Sprache: Daten werden mit ***Semantik*** angereichert, damit ***Interpreter*** Inhalt auf seine Art anzeigen kann *(Bspw. HTML, XML, Markdown)*
Content vs. Semantik: HTML: `<h1>` gibt nur an, dass Tag als Titel präsentiert werden soll, aber nicht, wie er dargestellt wird.
Template Engine: Erlaubt in Kombination mit Markup ***dynamische Daten zu rendern*** *(Conditional Rendering, Loops)*.
Template Extensions: Erweiterung für dekl. UI Programmierung, z.T. direkt von Laufzeitumgebung angeboten *(z.B. Razor, ASP, PHP)*.
Deklarative View Engine: Teile eines UI Frameworks, ermöglichen ***viel mehr*** als traditionelle Server-side Template Engines. Mehrfaches Rendering, Events, Dateninitialisierung, Data-Binding, Komponenten, Komposition. Praktisch alle ***modernen JS UI Frameworks*** verwenden solche Konzepte. Im Hintergrund oft Abwandlung von ***Observer Pattern*** mit ***Data-Binding***.
Vorteile Deklaratives UI: ***Klarheit***, Design-Tools für ***visuelle Bearbeitung***, ***Wiederverwendbarkeit***, Beschränkung der Möglichkeiten.

5.1. MVC

View / Observer: Rendert die ***Darstellung***, ***reagiert*** auf ***Änderungen*** des Models.
Controller: Reagiert auf ***Benutzereingaben***, ***empfängt*** und ***validiert*** Eingabe, leitet Manipulation des Models ein.
Model / Subject: Verwaltet ***Daten*** und ***Zustand*** der Anwendung.



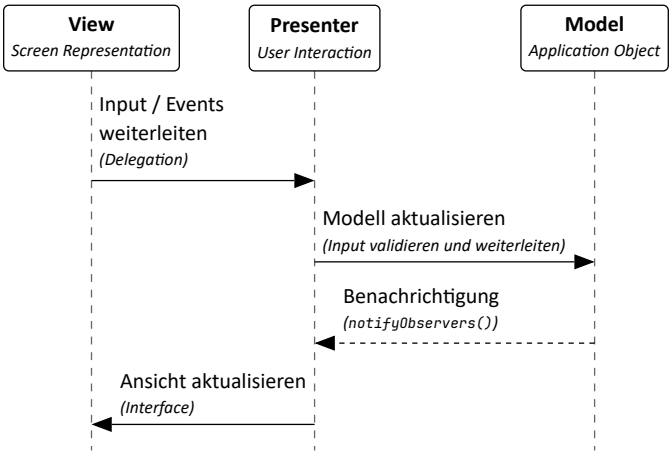
In MVC ist nur die ***logische Zuordnung*** zu ***Interaktion*** *(Controller)*, ***Präsentation*** *(View)* und ***Applikationslogik*** *(Model)* klar definiert.
Schwächen von MVC: Ermöglicht es nur schwer, Controls in Applikationen mit einem ***anderen Datenmodel*** wieder zu verwenden. Es ist schwierig, ***visuelle UI Editoren*** für MVC Architektur zu bauen.

5.1.1. Forms and Controls

In den 90ern beliebt. ***Sammlungen von wiederverwendbaren UI Elementen*** mit definierten Interfaces. ***WYSIWYG*** Entwicklungstool, Controls wurden per ***Drag and Drop*** eingefügt. ***EventHandler*** für die Controls wurden geschrieben. Zusammen gab das ein ***Formular***, welches ***Applikations-Logik*** enthielt.
Probleme: Enge Kopplung, wenig Wiederverwendbarkeit, schwer test- und wartbare Views, Vendor Lock-In, Toolabhängigkeit, verletzte SOLID-Prinzipien. ***RAD und Low-Code verursachen Forms & Controls.***

5.2. MVP PASSIVE VIEW

View: Keine ***statische Abhängigkeit*** oder ***Observer-Kommunikation*** zum Model. ***Interface***, um Austauschbarkeit zur erhöhen. Die View ***beobachtet nicht***, sie wird ***aktiv vom Presenter manipuliert***.
Presenter / Observer: Observer des Models. Ändert die View aktiv über ein Interface und konvertiert die Daten.
Model / Subject: Verwaltet ***Daten*** und ***Zustand*** der Anwendung.



In MVP ist die ***Aufgabentrennung***, die ***Kommunikation*** zwischen den Komponenten und die ***statische Abhängigkeit*** zwischen den Komponenten klar definiert.

Vorteile: Stark in ***Separation of Concern*** und in der Programmierung mit einem Domain Model. Stark in ***Wiederverwendung*** von Widgets und Controls und bei WYSIWYG. Gute ***Testbarkeit*** und ***Nachvollziehbarkeit***.
Nachteile: ***Komplexität*** ist stark erhöht.
Vergleich zu MVC: In MVP beobachtet der Presenter das Model. Die View beobachtet nicht, sondern wird aktiv vom Presenter manipuliert. ***Presenter*** ist eher auf ***Widget-Ebene*** implementiert, verwaltet ***mehrere Controls***. Die ***View*** wird als eine ***Struktur von Contols*** angesehen, enthält ***kein Verhalten*** bzgl. Reaktion der Control auf ***Benutzereingaben***. Änderungen am Model werden oft als Befehle in einem ***Command Pattern*** implementiert. Grundlage für ***Undo/Redo Funktionen***.
Wann verwenden: Architektur soll ***unabhängig*** von UI Technologie sein. Es sind mehrere ***technologisch unterschiedliche UIs*** möglich. M, V und P können auf ***verschiedene Computer*** verteilt werden. ***Thin Clients*** möglich.

5.2.1. MVP und Mediator Pattern (Mediated MVP)

Hilft, den ***Presenter*** aufzuteilen, ***schlanker*** und ***testbarer*** zu machen. Durch Events, welche durch die Views auf dem Mediator *(Presenter)* aufgerufen werden, wird ***Kommunikation erleichtert*** und ***Abhängigkeit verringert***. Models können Kommunikation ebenfalls über Mediator regeln und benötigen ***keinen Observer*** mehr. Befehle können in Commands gepackt werden, was Funktionen wie ***Undo/Redo*** einfach implementierbar macht. *(Siehe «Mediator» (Seite 3))*.

Vorteile: ***Reduzierte Komplexität*** durch zentrale Steuerung, verbesserte ***Warbarkeit***, erleichterte ***Erweiterbarkeit***.
Nachteile: ***Single Point of Failure***, pot. ***Performance-Einbussen***, erhöhte ***Abhängigkeit*** vom Mediator.

MVP mit Commands/Selections/Interactions

Events können durch Mediated MVP in Gruppen unterteilt werden:
- ***Commands:*** Kapseln Aktionen, die auf dem Model ausgeführt werden
- ***Selections:*** Bestimmen, welche Teile des Models von Aktionen betroffen sind. Views können so gezielt aktualisiert werden.
- ***Interactors*** entscheiden und koordinieren *(Passender Command wählen, Parameter von Selections ableiten, über Presenter anstossen)*.

Diese Aufteilung ist ein ***Grundbaustein moderner Architekturen*** *(Flux/Redux, Signals, State Machines)*. Damit und mit einem Pub/Sub Pattern können auch verteilte MVPs umgesetzt werden.

5.3. MVP SUPERVISING CONTROLLER

Supervising Controller: Presenter formt das Domain-Model in eine ***View-nahe Struktur*** *(Presentation Model)* um. Die View reagiert mit ***Observer Pattern*** auf Änderungen im ***Presentation Model*** *(Durch Data Binding an State gebunden)*. Der Presenter betrachtet das ***Presentation Model*** als Interface für die View und ***manipuliert die View nicht direkt*** *(verarbeitet Input und aktualisiert State)*.

MVP mit ***UI State***, ***Deklarativem UI*** und ***Data Binding***. Ist eine Mischung aus ***MVC***, ***MVP Passive View*** und ***Forms & Controls*** mit klaren Regeln und ohne deren Nachteile. UI Zustand ist nicht mehr im Model angesiedelt, sondern fungiert als State zwischen Presenter und View.

Vorteile: Presenter wird ***noch weniger Abhängig*** von Controls oder Widgets und State-Manipulation kann direkt mit dem Presenter getestet werden. Komposition wird ***einfacher*** und kann rein ***deklarativ*** erfolgen.
Nachteil: ***Abhängigkeit*** des State von der Data Binding Technologie.

5.3.1. State

Umfasst alle ***Zustandsinformationen*** des UI *(Sichtbarkeit, Aktivierung, Auswahl, Fokus, Formularwerte und temporäre Zustände)*. State ist ***flüchtig***, nicht permanent abgespeichert. Der State kann mit Übergängen über ***Zustandsautomaten*** modelliert werden und ***beeinflusst die Presentation***.

MVP beinhaltet zwingend Separation of Concerns, Wiederverwendbarkeit und Austauschbarkeit der View, Separates Testen von View, Presenter und Model.

6. DATA BINDING

Data Binding stellt sicher, dass jede ***Änderung in einem UI-Element*** automatisch auf die zugrunde liegende Abstraktion ***übertragen*** wird, und umgekehrt. Data Binding ist ein ***automatisiertes Observer Pattern*** mit Fokus auf ***Synchronisation*** zwischen ***State*** und ***View***.

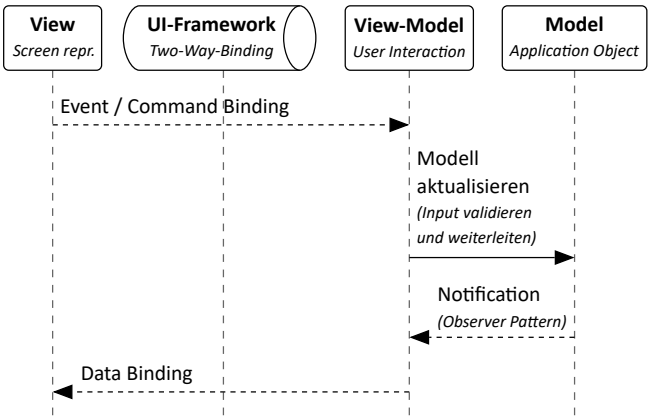
Benachrichtigungsmechanismus: Das beobachtbare Objekt löst bei einer Änderung einen ***Event*** mit dem Namen des geänderten Properties aus. Die View-Engine muss nur diesen `PropertyChanged`-Event ***abonnieren***, um über Änderungen informiert zu werden. Änderungen an ***Collections*** sind komplexer und müssen gezielt erkannt *(`add`, `remove`, `clear`)* und behandelt werden.

One-Way Binding: Erlaubt die Darstellung von Daten in der View ***ohne Rückfluss*** zum ***Model***.
Two-Way Binding: Synchronisiert Änderungen durch ***deklarative Anweisungen*** zwischen View und Model in ***beide Richtungen*** und ist eine ***zentrale Komponente*** in modernen UI-Frameworks.
View Engine: Framework, welches ***Standardaufgaben*** der Programmierung von ***Präsentation*** und ***Benutzerinteraktion*** abnimmt. Oft erweiterte deklarative ***Template Engines***, welche die Views automatisch und dynamisch an Änderungen des State anpassen. Diese automatische Synchronisation zwischen View und State nennt man ***Data Binding***.
Deklarative Anweisungen: Werden von der View Engine als ***Informationen für Automatisierung*** der «Abonnierung» *(Observer Pattern)* von Datenänderungen verwendet. Findet durch Kompilierung, Reflection oder Zugriff über Propertynamen statt *(`obj[propName]`)*.

Data Binding Hooks: Bei einer Datenänderung *(before, on change, after change, on error)* kann z.B. ein ***Validator*** zum Zug kommen, eine ***Datenkonvertierung*** stattfinden oder ein ***visueller Effekt*** ausgelöst werden
Vergessene Updates beim Data Binding: Dev vergisst, `PropertyChanged`-Event bei Änderungen einzurichten, Änderungen ausserhalb der getrackten Wege, Race Conditions.

7. MVVM

Basiert auf MVP Supervising Controller, aber einfacherer ***Kontrollfluss*** *(bei MVP entweder über Presenter oder Data Binding – wo eingreifen?)*. State der View & Presenter verschmelzen zum View Model. Synchronisation zur View über ***Data Binding mittels UI-Framework***.



Vorteile: Weniger ***Boilerplate-Code*** als MVP, Automatische ***Synchronisierung*** der View über ***Data Binding***, klare ***Trennung der Verantwortlichkeiten*** durch Commands und Data Binding, ***Testbarkeit*** des ViewModels ohne die View, ***Wartbarkeit*** wegen Kapselung.
Nachteile: Starke ***Abhängigkeit*** von der View Engine. ViewModel muss sich an ***Konventionen*** des Frameworks halten. Data Binding erschwert ***Debugging***. Two Way Binding kann ***Performance*** negativ beeinflussen.
MVVM implementiert meist ***Data Binding ohne automatische Auslösung*** von `PropertyChanged`. Einfacher, ***Zyklische Updates zu erkennen*** und bei explizit ausgelösten Benachrichtungungen die ***Performance zu optimieren***. Aber es können ***Updates vergessen*** werden.

7.1. INTERAKTIONSLOGIK

Commands: Abstraktion der Benutzerinteraktion. ***Enkoppelt*** UI von der Interaktionslogik. Prüft `CanExecute()` und delegieren Workload weiter an ViewModel, welches wiederum das Model aufruft, welches die Modifikation der Daten durchführt. Können ***mehrere*** VMs manipulieren.
`CanExecute`: Kondition, ob Befehl ausgeführt werden kann *(z.B. $\neq$ null)*
Vorteile von Command Interfaces: ***Konsistenz*** und ***Wiederverwendbarkeit*** in der View, ***leicht testbar***, da Interaktionslogik von UI entkoppelt ist. ***Klare Trennung*** von Benutzeroberfläche, Interaktionslogik und Geschäftslogik. Möglichkeit der ***Unterstützung einer Undo/Redo-Funktionalität***.

	Eigenschaft	***MVP (passiv)***	***MVVM***
Abhängigkeit	**View von Framework**	keine	stark an Framework gekoppelt
	Presenter/VM → View	über Interface	Framework-Mechanismus
	View → Presenter/VM	über Mediator Interface oder Event Binding	direkte Referenzierung auf Widget Ebene
Informationsfluss	**Presenter/VM → View**	Manipulation der View über Interface	Data Binding
	View → Presenter/VM	Events, Eventhandler oder direkter Methodenaufruf auf Presenter	Two Way Binding, Commands

7.2. DYNAMISCHE PROXIES

Für ***Automatisierung des Observer Patterns***. Mit einem Proxy werden Änderungen am Zustand ***abgefangen*** und ***automatisch*** an alle Abonnenten *(z.B. die View)* weitergeleitet *(«Proxy» (Seite 3))*. Die State-Logik kennt den Proxy nicht. Proxy-basierte Observable State Pattern bieten eine ***flexiblere Alternative*** zu klassischem MVVM Pattern, da es den Zustand ***unabhängig*** von einem ***spezifischen Framework*** hält und so die ***Abhängigkeit*** des ViewModels vom Framework ***reduziert***.

7.3. KOMPONENTEN-BASIERTE ARCHITEKTUR

Architekturtyp, der auf einem ***anderen Abstraktionslevel*** ist als MVC, MVP oder MVVM. Es ist eine ***vertikale Trennung***: Jede Komponente kapselt Darstellung, Logik und Daten für einen bestimmten UI-Bereich. Die einzelnen Komponenten können ***zusätzlich*** nach einem MV(X) Pattern implementiert werden. ***Deklarative UIs*** sind Komponentenbasiert. Angular, vueJS und React haben eine ausgeprägte Komponentenbasierung mit vorgegebener innerer Kompontentenarchitektur. ***Slots:*** Argumente für Parameter.

Vorteile: ***Wiederverwendbarkeit***, ***Komposition*** *(Komponenten können für grössere Features kombiniert werden)*, ***Kapselung*** *(einfachere Umsetzung von OCP)* und eine klare ***Trennung*** von Anliegen. Komponenten sind in der Regel ***unabhängig*** und können in verschiedenen Kontexten eingesetzt werden.
Nachteile: ***Schlechte Separated Presentation***, jede Komponente verwaltet ihre eigene Logik, Darstellung und Daten. ***Aufwendige Kommunikation/Synchronisation***; ohne weitere Tools ist Kommunikation zwischen Komponenten aufwendig.
Eliminieren der Nachteile:

- ***Passive Komponenten:*** State wird nur von übergeordneten Komponenten kontrolliert. Verbessert Wiederverwendbarkeit und Testbarkeit.
- ***Isolated Stateful Components:*** Komponenten, die einen State verwalten/manipulieren sollen nichts anderes tun und als Childkomponenten nur passive Komponenten verwenden.
- ***State Container:*** Erleichtern die Synchronisation zwischen Komponenten & fördern Separated Presentation. Abstrakte View ist ohne View testbar.

7.3.1. Moderne Komponentenbasierte Architektur

Lokaler State: State *(ohne State Container)* ist lokal in Komponente gekapselt.
View: ***Deklarative View*** als Teil der Komponente.
Interaktionslogik: Oft in der Komponente gekapselt, ***ohne View testbar***.
Interne Kommunikation zur View: Durch Data Binding/Events oder Hooks
Wiederverwendbarkeit: Komponenten sind ***eigenständig*** und für verschiedene Teile der App verwendbar.
Interne Pattern: Angular oder vueJS basieren stark auf ***Data Binding***. MVVM, Observable State oder Reactive Konzepte deshalb naheliegend. Bei ***funktionalem React*** *(Ohne State Container)* entsteht eine Art ***MVC***. Controller und View sind ***verschmolzen***. Erster Teil der Funktion ist der Controller, der zweite Teil die View.
Sharing von State: ***State Container*** erlauben ***State Sharing*** zwischen Komponenten und Trennung von State und View. Manipulation des States ist stark von ***MVP*** beeinflusst. ***Zyklische Pattern*** *(MVU/MVI)* werden oft unterstützt.

8. PATTERNS IN REACT

React arbeitet nach dem Hollywood Prinzip, ist also ein Framework.

8.1. JAVASCRIPT XML (JSX)

Deklarativer Syntax für HTML in JS. Ermöglicht Implementation von ***wiederverwendbaren Komponenten*** und mächtigen ***Kompositionen*** *(Reacht nutzt das Composite Pattern «Composite» (Seite 3))*. Wird von vielen JS-UI-Frameworks verwendet und muss nicht unbedingt HTML ausgeben. Funktionen müssen JSX returnen, nur ***Props*** *(simple JS-Objekte)* als Parameter. CSS in «{{ ... }}».
TypeScript XML: typisiertes JSX für TypeScript.

8.1.1. JSX zu JavaScript kompilieren

Der ***TypeScript Compiler*** oder ***Babel*** *(ein JS Compiler)* können JSX/TSX kompilieren. Die gängigen Tools erwarten `*.jsx`/`*.tsx` als Dateiendung.

```
{ // tsconfig.json
  "compilerOptions": {
    "jsx": "react", // gibt JSX Standard an, mit dem gearbeitet wird
    "jsxFactory": "React.createElement", // Factory-Funktion, die JSX
generiert. Benötigte Signatur: createElement(type, props, ...childern)
    "jsxFragmentFactory": "Fragment" }
```

eigenes JSX Framework: Obige ***Angaben*** in Compiler-Optionen setzen, ***`createElement(type, props, ..children)`*** bereitstellen *(oder Adapter um JS `document.createElement()` schreiben)*.

```
createElement("div", {id: "root"}); → <div id=root></div>
```

VDom: React baut vereinfacht gesagt aus den JSX Funktionen einen ***Virtual DOM*** mit den Elementen, ihren Attributen und ihren Kindern.
Render Funktion: Damit die Inhalte im HTML angezeigt werden können, braucht es eine `render`-Funktion, die den ***VDom in HTML umwandelt***.

8.2. FUNCTIONAL COMPONENTS

Functional Components *(FC)* werden als Funktion in JSX geschrieben. Sind ***Komponenten***, die als JSX Tags wiederverwendet werden können. Können ***Props*** und ***Children*** entegegennehmen und einen ***Zustand*** kapseln.

Beispiel einer Funktionalen Komponente und deren Aufruf

```
const HelloMessage = (props: { name: string }) => {
  return <div style={{ fontSize: 20px }}> // ← CSS in camelCase
    Hello {props.name}</div>;}

const HelloWorld = () => { return <div><HelloMessage name="OST" /></div>; }
```

Im Beispiel oben wird das ***Dependency Inversion Principle*** verletzt, weil `HelloWorld` direkt von `HelloMessage` abhängig ist. `HelloMessage` müsste als Prop an `HelloWorld` übergeben werden:

```
const HelloWorld = (props: { children?: React.ReactNode }) => {
  return <div>{props.children}</div>; };
const App = () => {
  return (<HelloWorld><HelloMessage name="OST" /></HelloWorld>);};
```

8.2.1. React Hooks

Hooks ermöglichen das ***Eingreifen*** in den Rendering Prozess einer Funktionalen Komponente. Die wichtigsten sidn `useState()` und `useEffect()`. Sie erlauben es, Komponenten mit ***Zustand*** zu implementieren und auf Änderungen des Zustands zu ***reagieren***.

React verwaltet für jeden FC einen ***Component Context*** im Hintergrund, in welchem Informationen zum ***letzten Rendern*** der Komponente gespeichert werden *(Fiber-Architektur)*. Die Hooks werden ***der Reihe nach registriert*** und die Informationen mithilfe eines ***aufsteigenden Indexes*** in einer Struktur auf dem Component Context abgespeichert und somit immer in der gleichen Reihenfolge aufgerufen. Deshalb darf ein Hook nicht mit einer ***Conditional Anweisung*** *(`if`)* zugewiesen werden, weil er dann nur manchmal aufgerufen wird und somit die ***Reihenfolge*** durcheinander gebracht wird.

Die nachfolgenden Beispiele sind ***vereinfacht***, da React ***asynchron*** rendert und Zustandsänderungen zusammenfasst *(«Batching»)*. React verwaltet eine eigene ***Update Queue*** und kann wartende Updates abbrechen *(interrupting)*

`useState(initialState)`

Ist ein ***Observer Pattern*** *(Subject: count, Observer: Counter)*. Damit lässt sich ein Zustand ***definieren***, ***lesen*** und ***ändern***. Erstellt die Variable und dazugehörigen Setter.

```
import { useState } from 'react';
export const Counter = () => {
  const [count, setCount] = useState(0);
  return ( <div>
      <p>Aktueller Zählerstand: {count}</p>
      <button className="button scale08"
        onClick={() => setCount(count + 1)}>+</button></div> ); };
```

`useEffect(setup, dependencies)`

Ist ein ***Observer Pattern*** *(Subject: setInterval, Observer: console.log)*. Führt ***asynchron*** Seiteneffekte wie Datenabrufe oder DOM-Manipulationen nach dem Rendern aus. Wird immer ausgeführt, wenn sich Abhängigkeiten ändern. Mehrere Aufrufe registrieren mehrere Handler. Blockiert ***nicht*** das UI.

```
const url = "http://localhost:5093"; const player = new Player{...};
useEffect(() => {
  const connection = createConnection(url, player)
  connection.connect();
  return () => connection.disconnect();
}, [url, player]); //dependency-array, values used in `setup` lambda
```

Ist das Dependencies-Array leer, wird der Effect nur beim ersten Render ausgeführt. Ansonsten bei jedem Rerender durch Änderungen an den Dependencies.

Sollte ***nicht verwendet*** werden, um den ***abgeleiteten Zustand zu verwalten***, der direkt aus Props oder anderem Zustand berechnet werden kann, da das zu ***unnötiger Komplexität*** führt *(wird nach dem Rendern ausgeführt, somit muss einmal mehr gerendert werden)*. `useMemo()` ist die synchrone Variante von `useEffect()`: «`useMemo(calculateValue, dependencies)`» (Seite 8).

8.3. RECONCILIATION UND FIBERS

React verwendet einen ***vDOM***, weil direkte DOM-Manipulationen ***langsam*** und ***ineffizient*** sind. React nutzt ***Reconciliation*** für effiziente Updates: nur ***betroffene*** Komponenten werden neu ausgeführt *(unbetroffene Child-Komponenten werden ebenfalls nicht ausgeführt)*.

Vor React v16 wurden ***Position im Array*** und ***Name*** der Komponente verglichen. Der State kann so jedoch in die falsche Komponente rutschen, wenn keine ***stabilen und eindeutigen Keys*** verwendet werden.
Seit React v16 werden ***Fibers*** verwendet. Interne Datenstruktur für vDOM. Speichern ***State***, ***Props*** und Referenzen zu HTML-Elementen. Navigation via ***child***, ***sibling***, ***return*** *(Parent)* und ***stateNode*** *(Verbindung zum echten DOM-Element)*.

Current vs. Work-in-Progress: ***Current Fiber*** ist der aktuelle UI-Zustand. ***WIP Fiber*** entsteht während der Reconciliation. ***WIP*** wird nach Reconciliation zum neuen ***Current Fiber***.

8.4. CONCURRENT MODE

JS hat im Browser nur ***einen einzigen Thread***. Asynchronität bedeutet hier, dass die Behandlung einer Anfrage auf einen ***späteren Zeitpunkt*** verschoben werden kann, ohne den ***Hauptthread*** zu blockieren. Dies wird durch das ***Event-Loop-Modell*** erreicht. Asynchrone Aufgaben werden in eine ***Warteschlange*** gestellt und nacheinander ***abgearbeitet***.

Der ***Concurrent Mode*** in React verbessert die ***Reaktionsfähigkeit*** durch das Priorisieren und Unterbrechen des Renderings. So bleibt das UI reaktionsfähig.

Fiber-basierte Aufteilung: React teilt das Rendering in ***kleine Einheiten*** auf. Jede ***Fiber*** ist eine ***Arbeitseinheit***. Nach jedem Fiber wird überprüft, ob es etwas ***wichtigeres*** gibt, ***falls ja:*** Unterbrechen und wichtigere Aufgabe zuerst. ***Falls nein:*** weiter mit dem nächsten Fiber.

Zwei Render Phasen: ***Render Phase*** ist unterbrechbar und beinhaltet vDOM erstellen, Komponenten ausführen & Änderungen berechnen. Die ***Commit Phase*** ist nicht unterbrechbar und beinhaltet DOM-Änderungn anwenden, Refs aktualisieren & `useEffect()` ausführen. Muss atomic erfolgen.

Prioritäten in React: ***Immediate*** sind Benutzer-Inputs. Dafür wird das aktuelle Rendering unterbrochen. ***Normal*** sind Netzwerk-Responses oder Animations-Updates. ***Low*** sind Hintergrund Tasks wie Analytics.

`useMemo(calculateValue, dependencies)`

Im Vergleich zu ***`useEffect()`*** gibt es noch ***`useMemo()`***, welches ***synchron*** ist und das Rendering ***blockiert***. Wird ***während*** dem Rendering ausgeführt und für ***pure functions*** verwendet. Es speichert Ergebnisse im Component Context. So werden teure Berechnungen nicht unnötig wiederholt. ***`useCallback()`*** wird für das gleiche verwendet, nur speichert es Funktionen.

9. REACTIVITY

Bietet einen ***effizienten*** und ***sicheren*** Ansatz zur Gestaltung moderner, ***ereignisgesteuerter*** UIs und ***vermeidet*** typische ***Probleme***, die im Observer Pattern auftreten.

9.1. PROBLEME DES OBSERVER PATTERN

- **Kontrollfluss-Chaos:** Kontrollfluss ist ***nicht direkt im Code erkennbar***, da dynamisch aufgebaut. ***Unübersichtlich***, erschwert Lesen, Testen und Debugging. ***Unnötige Aufrufe*** von Handlern, wenn ein Observer bei mehreren Subjects «subscribed» ist.
- **Verpasste Events:** Wird ein Handler erst ***nach dem Auslösen registriert***, geht das erste Event ***verloren***. Automatischer Aufruf beim ***`subscribe()`*** hilft dagegen.
- **Memory Leaks:** Entwickler müssen alle Callbacks wieder ***deregistrieren***.
- **State-Chaos:** In Callbacks wird oft der ***State verändert***. Wird schnell ***unübersichtlich***. Dagegen hilft entweder das Aufgeben der Kapselung oder die Verwendung eines anderen Synchronisationsmechanismus.
- **Endlosloops:** ***Zirkuläre Abhängigkeiten*** werden oft übersehen.
- **Race Conditions:** ***Ordering-, Reentrancy- und Scheduling-Effekte*** können ähnliche Probleme verursachen. Wird schnell sehr ***unübersichtlich***.
- **SOLID Verstösse:** ***SRP*** *(Subject verwaltet Zustand und Benachrichtigungen, Observer empfangen Benachrichtigungen und verarbeiten diese)*, ***OCP*** *(Für verschieden benachrichtigte Observer muss eine Änderung am Subject vorgenommen werden)* und ***ISP*** *(Je nach Interface müssen Observer nicht benötigte Methoden implementieren, da das Interface annimmt, dass Subjects unterschiedlich benachrichtigt werden)*.

```
let path: Path | null = null;  // ⚠ 1. State Chaos: escaped scope
const moveObserver = (event: MouseEvent) => {
  path?.lineTo(event.position);  // ⚠ 2. Komplexität Kontrollfluss
  draw(path); };
control.addMouseDownObserver((event) => {
  path = new Path(event.position);  // Schritt 1
  // ⚠ 3. Verpassen des ersten Events möglich
  control.addMouseMoveObserver(moveObserver);
  // ⚠ 4. Memory Leak: manuelles Cleanup nötig (dispose nötig)
});
control.addMouseUpObserver((event) => {
  control.removeMouseMoveObserver(moveObserver);  // Schritt 3
  // ⚠ 5. Race Condition / Fehlende Separation of Concerns
  path?.close();
  draw(path);
  // ⚠ 6. Endlosloop falls draw auch Events triggert
});
```

9.2. SIGNALS / EFFECT / COMPUTED

Damit lassen sich die typischen Probleme des Observer Patterns reduzieren.

Reactivity: ***Automatische*** Reaktion auf lokale Zustandsänderungen. Beinhaltet die ***Automatisierung*** der Mechanismen und des Lifecycle Management des Observer Patterns. Meist für Datensynchronisation mit UI verwendet.

Signal: Ein Signal hat ***einen Wert***, der sich ***ändern*** kann. Es kann andere Komponenten über Änderungen ***informieren*** und beinhaltet ein ***verstecktes Tracking***, das erfassen kann, wann es gelesen wird. Ist ein ***Observer Pattern*** für nur einen primitiven Wert.

```
const mySignal = new Signal(0); // Ein Signal mit Initialwert 0
mySignal.subscribe((value) => console.log(value));
// Benachrichtigung bei Wertänderung. Nur versteckt aufgerufen
mySignal.value = 5; // Ändert den Wert und benachrichtigt Abonnenten
```

Effect: Wird ***automatisch*** neu ausgeführt, wenn sich ein Wert eines Signals ***ändert***, welches in der letzten Ausführung des Effects verwendet wurde. Eignet sich für UI Updates. Kann ***mehrere*** Signals enthalten und abonnieren.

```
// der Effect wird in einem UI Framework automatisch erstellt
const counterUpdate = new Effect(() => {
  counterElement.textContent = `Counter: ${counterSignal.value}`; });
counterUpdate.run(); // wird durch das UI Framework aufgerufen.
counterSignal.value++; // automatisches Update
```

Computed: ***Kombination von Signal und Effekt***. Verwendet einen Effect, um ein gekapseltes Signal zu ***modifizieren*** und ermöglicht das ***Lesen*** und die ***Subscription*** von diesem Signal. Wird oft ***lazy*** umgesetzt *(werden bei Abhängigkeit durch den Dependency Tracker «dirty» gesetzt und bei Bedarf neu berechnet)*.

```
// der Effect wird in einem UI Framework automatisch erstellt
const A = new Signal(8); const B = new Signal(12);
const sum = new Computed(() => { return A.value + B.value; });
sum.subscribe((value) => console.log(value));
A.value = 2; // Ausgabe 14 (durch Änderung von A)
```

Vorteile: Kontrollfluss durch Dependency Container kontrolliert, Effekt liest immer Wert beim ersten Zugriff, Zentrale Verwaltung von Signals, Race Conditions unwahrscheinlich, keine Memory Leaks oder Endlosloops.

Dependency Tracker: Arbeitet im Hintergrund und ***verbindet Signal und Effect***. Ansonst ***unsichtbar***. Zuständig für das Verfolgen von ***reaktiven Abhängigkeiten*** und ***Aufrufen von Effect*** bei Änderungen.

Signals vs. `useState()`: Mit `useState()` wird der State der Komponente gehalten. Mit Signals kann dieser ***herausgelöst***, ***wiederverwendet*** und ***separat getestet*** werden: Damit wird nur der betroffene Teil, nicht die ganze Komponente aktualisiert. Signals sollten in der ***Programmiersprache*** und nicht im Framework integriert sein. Dies führt zu kompletter ***Trennung*** von non-View-Code von View und Framework. Signals ermöglichen ***fein granulare Reaktivität*** durch präzise Abhängigkeiten und automatische Aktualisierungen.

9.3. REACTIVE PROGRAMMING

Reaktives Programmieren ist ein ***asynchrones Programmierparadigma***, das sich auf die Reaktion auf Datenströme *(Menge von Events)* und Ereignisse über ***Zeit*** konzentriert. Klassisches Observer Pattern ist imperativ, Reactive Programming hingegen ***deklarativ*** aus dem Blickwinkel der ***Veränderungen***. So entsteht das Konzept von ***Streams***. Observer agieren als ***Consumer*** dieser Streams.

Promises: Versprechen, dass in Zukunft ein Resultat kommen wird. Handler wird im Vergleich zum Observer Pattern nur ***einmal*** aufgerufen. `then()` wrappt den Rückgabewert automatisch in ein Promise, das macht ***Chaining*** möglich *(Im OP nicht möglich, weil `setValue()` `void` zurückgibt)*.

Stream: Eine ***Sequenz von Ereignissen***, die über eine ***Zeitspanne*** auftreten.

Observable: Ein ***Stream***, dessen Ereignisse *(Wert, Fehler, Endsignal)* von einem ***Abonnenten*** *(Observer)* verarbeitet werden können. Bietet Unterstützung für ***Nachrichtenaustausch*** zwischen Publisher und Subscriber. Vereinfacht ***Ereignisverarbeitung***, asynchrone Programmierung und Handhabung mehrer Werte. Ist ***lazy*** *(Code wird erst bei Aufruf von `subscribe()` ausgeführt)*, vereinfacht ***Abmelden*** durch direktes Zurückgeben von `unsubscribe()` nach `subscribe()`.

Observer: ***Implementiert*** `next()`, `error()` und `complete()`. Das Observable ***ruft diese Methoden auf***, um Werte, Fehler oder das Ende des Streams zu signalisieren.

Operatoren: ***transformieren*** Observables und geben neue zurück. ***Filter*** z.B. filtert Ereignisse und lässt nur bestimmte weiter.

Pipe: wird verwendet, um mehrere Operatoren auf ein Observable anzuwenden und den Datenstrom zu ***transformieren***. Erlaubt ***sauberes Verketten*** von Operatoren wie `map()`, `filter()` etc. Macht den Code ***modularer*** und ***übersichtlicher***.

Rate Limiting: ***Debounce*** wartet nach jedem Ereignis eine bestimmte Zeit und emittiert letzten Wert nach einer Inaktivitätsperiode *(z.B. zum Aktionen erst auszuführen, wenn Benutzer Eingabe beendet hat)*. ***Throttle*** emittiert sofort und ignoriert dann weitere Werte für eine bestimmte Zeitspanne *(z.B. um Verarbeitung von Ereignissen zu reduzieren, die in hoher Frequenz auftreten, wie Scroll, um Ressourcen zu schonen)*.

Subject: Spezielles Observable, das Werte an ***viele Observer*** verteilen kann *(Multicast)*. Es kann sowohl als ***Observable*** als auch als ***Observer*** agieren. Bei Multicast teilen sich alle Abonnenten den selben Datenfluss, sodass alle die ***gleichen*** Werte ***gleichzeitig*** erhalten. Subject ist Beispiel für ***Hot Observable***, da es bereits aktiv ist und Daten sendet, egal ob Abonnenten vorhanden sind oder nicht.

10. STATE CONTAINER

State Container wie Redux ***speichern den Zustand*** an einem zentralen Ort, bieten Methoden ***zur Änderung*** und ermöglichen einheitlichen Datenfluss. Vorteile sind die konsistente ***«Single Source of Truth»*** und ein klarer, unidirektionaler Datenfluss, was ***Wartung*** und ***Debugging*** erleichtert. **Reacts Context API** ist ein primitiver State Container *(kein Time-Travel-Debugging oder autom. Dependency-Tracking)*.

Lokaler State von Komponenten ist schwer zu teilen und führt zu Problemen bei Trennung von State und UI-Logik.
State Sharing: Notwendig, um Zustände in komplexen Anwendungen konsistent und effizient zu verwalten.

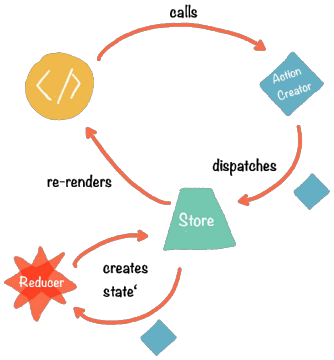
10.1. LÖSUNGEN FÜR STATE SHARING

State Liftung & Props Drilling: Die ***einfachste*** Lösung für State Sharing. Mit ***State Lifting*** wird der State ***eine Hierarchie weiter nach oben*** verschoben und mit ***Props Drilling*** wird er an ***Childkomponenten*** weitergegeben. Wird jedoch schnell ***Performance-Intensiv*** weil der gesamte vDOM neu gerendert werden muss. Führt auch zu ***unübersichtlichem*** und ***fehleranfälligem*** Code. ***Verletzt SRP*** und ***ISP***.
`useContext()`: Der State wird ***indirekt hierarchisch*** ohne Props Drilling ***geteilt***. Verwendete Konponente definert ***Kontext*** *(`createContext()`)* alle Komponenten weiter unten in Hierarchie können ihn verwenden *(`useContext()`)*. Provider = Container, Context = State, = Consumer
Dependency Injection: MVVM Frameworks *(WPF, MAUI)* können einen DI-Container verwenden, um den State zu teilen und das gleiche ViewModel in verschiedenen Komponenten zu verwenden oder Observable Proxies zu teilen.

10.2. REDUX

Ist ein ***State Container***, der oft in React-Anwendungen verwendet wird. Bietet eine ***einheitliche*** und ***vorhersagbare*** Datenverwaltung und schafft eine ***zentrale Quelle*** für den gesamten Anwendungszustand. Das macht es einfacher, Änderungen zu verfolgen und Fehler zu minimieren. Debugging durch ***Time Traveling***. Redux implementiert das ***Observer Pattern*** & ***Command Pattern***.

Store: Zentraler ***Speicherort*** für ***gesamter Zustand*** der Anwendung mit Benachrichtigungsmechanismus für Änderungen.
Action: Beschreibt, ***was*** in der Anwendung ***geschehen*** soll. Enthält einen ***Type*** für die Art der Änderung *(z.B. `ADD_ITEM`)* und zusätzliche Daten als ***Payload***. **Action Creator:** Funktion, die eine ***Action*** mit einem ***Payload*** aus einem Event erstellt und das ***Resultat*** mit `dispatch()` an den ***Store*** sendet. **Reducer:** Pure Function *(Ohne Side Effects)*, die den ***aktuellen State*** und eine ***Action*** annimmt und einen ***neuen Zustand*** zurück gibt. ***Verarbeitet die Action***. Vereinfacht Testing. Datenfluss läuft ***zirkulär*** und ***unidirektional***.



Vorteile: ***Vorhersagbarkeit:*** jede State-Änderung läuft durch den gleichen Pfad. ***Debuggbarkeit:*** Action-Log speichert Veränderungen und ***Time-Travel-Debugging*** ist möglich.

Redux ist mehr oder weniger ***MVP*** mit ***Commands/Selections/Interactors***: Commands → Actions, Selections → `useSelector()`, Interactors → Reducers, Model → Store, Passive Views → «Stupid Views». Redux bringt aber einige Verbesserungen: Unidirectional Flow, Immutable Updates, Pure Functions, Time Travel Debugging, Predictable State Changes. **Redux Toolkit (RTK):** Offizielle, moderne Art, Redux-Logik zu schreiben. ***RTK Query*** ist eine fortgeschrittene Daten-Fetching-Lösung, die darauf aufbaut *(beinhaltet automatisches Caching, Background Updates, Optimistic Updates, Error Handling, Loading States, Tag-basierte Invlidierung, ...)*.

10.2.1. Thunks (Async)

Middleware erweitert Redux um zusätzliche Funktionalität wie Logging, Caching oder ***Asynchronität***. Ein ***Thunk*** ist eine Funktion, die eine bestimmte Berechnung oder Aktion ***verzögert*** ausführt und erst aufruft, wenn sie wirklich benötigt wird *(In React z.B. um Code nach dem Abschluss von asynchronem Code auszuführen)*. ***Thunk Middleware*** ist eine spezielle Middleware für Redux und ermöglicht ***asynchrone Logik*** durch Dispatching von Thunks. ***Typische Thunk Patterns sind:***
Loading States: `START`, `SUCCESS`, `ERROR` ermöglichen UI-Feedback während asynchroner Operation.
Conditional Dispatching: Zugriff auf aktuellen State via `getState()` *(z.B. User nur fetchen, wenn nicht geladen oder Daten veraltet sind)*
Chain Thunks: Ein Thunk kann andere Thunks dispatchen
Error Handling: Zentrale Fehlerbehandlung in Thunks

10.3. REAKTIVE STATE CONTAINER (Z.B. MOBX)

MobX vs. Redux: Weniger ***Boilerplate***, kein explizites ***Dispatching***, einfacher bei ***komplexen*** State-Änderungen. ***Automatische Synchronisierung*** von UI und State durch ***Dependency-Tracking***. Schlechtere Nachvollziehbarkeit, schwierigeres Debugging. Grössere direkte Abhängigkeiten vom State Container.

10.3.1. MobX Kernkonzepte

MobX verwendet ein ***Observable State Proxy*** Pattern *(«Dynamische Proxies» (Seite 6))* für Zugriffs- und Änderungs-Detection, kombiniert mit ***Signal Read-Tracking*** *(«Signals / Effect / Computed» (Seite 8))* für automatische Dependency Registration *(Reaktivität)* ohne zentralen Store.

Observable (State): Datenstruktur, die auf Änderungen reagiert. MobX verwendet ***Observable State Proxies***, um Zustand zu verwalten.
Actions: Definieren von State-Änderungen. Actions ***modifizieren*** Observable State. Die Updates lösen erst zum Schluss ein Rendering aus.
Computed: Abgeleitete Werte, reactive Funktionen die automatische Updates auslösen können. ***Identisch zu lazy Signal***.
Reactions: Automatische Reaktionen *(effect / autorun)*. Änderungen werden automatisch getrackt.

makeAutoObservable(): ermöglicht das ***automatische Konvertieren*** einer Instanz einer Klasse in Observable Werte, Computed und Actions. Besser als @Decorators, welche deprecated sind. ***`autorun()`:*** MobX-Version von Effect.
Performance-Realität: Mit `observer()` von MobX in React wird die ***ganze Component neu gerendert*** und der ***vDOM komplett neu*** erstellt. Ist also ***nur automatisches Component-Tracking***, keine ***feingranularen*** DOM-Updates möglich. Dafür braucht es andere Frameworks.

TC39: Erweiterter zukünftiger Signal Standard, der MobX ***obsolet*** macht. Alle MobX-Features sind damit nativ verfügbar.
Hauptunterschiede zu MobX: ***Explizite Signal API*** *(Mit Decorators ist TC39 nicht von Proxies abhängig)* und ***Native Browser Support*** *(bald ohne Library verfügbar)*.
Deep Reactivity: Viele Frameworks brauchen Deep Reactivity *(Reaktive Objektbäume)*, das ist ***performancetechnisch*** nicht optimal. TC39 bietet bessere Alternativen für ***«Flat Reactivity»*** mit `SignalObject` oder über Decorators.

10.4. STATE MACHINES ALS STATE CONTAINER

Probleme vom State Pattern: If-Else ***Chaos***, Diverse verletzte ***SOLID*** Prinzipien *(LSP, DIP, OCP, ISP)*. Viel ***Boilerplate***, schwer visualisierbar, unklares ***Memory Management***, Keine Hierarchie, keine Guards *(Transitions jederzeit möglich)*, Kein Event System. Für ***komplexe*** State Logik ***unzureichend***. *(«State» (Seite 3))*

State Machines hingegen sind klar überblickbar:

```
const playerMachine = { initial: 'stopped', states: {
    stopped: { on: { PLAY: 'playing' } },
    playing: { on: { PAUSE: 'paused' } },
    paused: { on: { PLAY: 'playing' } } }};
```

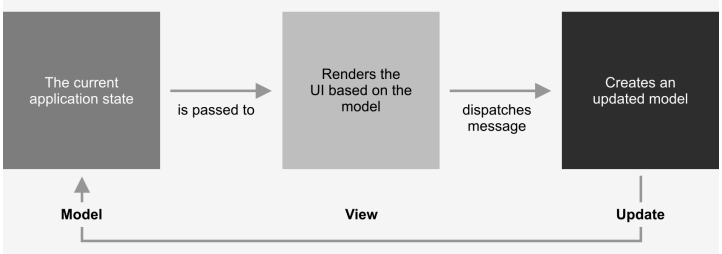
Verbesserungen von modernen State Machine Frameworks: ***Weniger Code:*** Konfiguration statt Implementation, ***Typ-Sicherheit:*** Auto-generierte TypeScript Types, ***Visualisierung:*** State Charts sind selbstdokumentierend, ***Testing:*** States und Transitions sind explizit testbar, ***Hierarchical States:*** Nestet States und parallele States möglich.
XState kombiniert ***State Machine*** mit ***State Container***. Bietet kein Boilerplate dank ***deklarativer Config***, ***Guards*** für bedingte Transitions, hierarchisch & parallele ***States***, ***Event-based*** System.
Hierarchical State Machines: Erlauben die Modellierung komplexer Zustände mit ***Unterzuständen***, was die ***Übersichtlichkeit*** bei komplexer Logik erhöht.
Guards und Actions: Guards ermöglichen ***bedingte Transitions***, Actions führen ***Side Effects*** aus. Das macht State Logik ***testbar*** und ***vorhersagbar***.

Vorteile: ***Robustheit*** *(Impossible States Prevention, deterministic)*, ***Developer Experience***, ***Skalierbarkeit***
Nachteile: ***Learning***, ***Verbosity*** *(Mehr Boilerplate für simple Logic)*, ***Integration*** *(Zusätzliche Dependency, Codebase Migration)*
Verwenden bei: ***Komplexer UI-Logik***, Business Prozesse *(Workflows)*, Bug-prone Areas, Games, Echtzeit-Kommunikationsprotokolle.
Nicht verwenden bei: ***Einfachen Use-Cases***, Datenintensiven Applikationen.

10.5. MODERNE MV* PATTERN

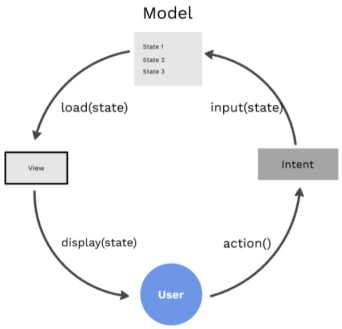
Probleme klassischer MVP/MVVM: ***Bidirectional Data Binding*** *(Schwer nachvollziehbare Updates)*, ***schwierige State Synchronization*** zwischen State und View.
MVVM Two-Way Binding Probleme: Der Setter muss selber implementiert und mit Benachrichtigungslogik befüllt werden. Daraus entsteht ein ***unkontollierter Kontrollfluss*** *(Cascading Updates, Side Effects, Zyklen, Performance)*.
Reactive MVVM: Bietet automatische Reaktivität, Transactional Updates, Computed Values mit Memoization, Automatisches Cleanup, Unidirectional Data Flow, einfachere Testbarkeit.

MVU (Model View Update): Einfachere, ***funktionale Architektur*** für synchrone, deterministische ***Zustandsänderungen***. Spezifische Implementierung von ***MobX***.

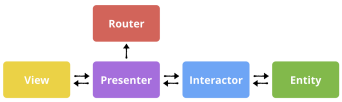


The Elm Architecture (TEA): ***Model:*** Immutable State/Data Structure der Anwendung. ***View:*** Pure Function, die Model zu HTML/UI rendert, sendet Messages bei User-Interaktionen. ***Update:*** Pure Function, die Message und Model zu neuem Model berechnet. ***Geschlossener Kreis:*** Model zu View *(rendering)* zu Update *(bei Messages)* zu neuem Model. ***Eigenschaften:*** Alle Functions sind pure *(testbar)*, State ist immutable *(time-travel)*, unidirectional flow *(predictable)*. ***Einfluss:*** Redux, moderne React Patterns basieren auf TEA Prinzipien.

MVI (Model View Intent): Vorhersage von ***Zustandsänderungen***, ***Single Source of Truth***, ***Unidirektionaler*** zirkulärer Datenfluss. Spezifische Implementierung ***Redux***.



VIPER: iOS Clean Architecture



11. RENDERING STRATEGIEN

Rendering Strategien ***wählen*** heisst, die untenstehenden Concerns der jeweiligen Situation entsprechend in ***Balance*** zu bringen.
- ***Performance:*** Statisch *(Sofort verfügbar)*, dynamisch *(Berechnung erforderlich)*
- ***SEO & Auffindbarkeit:*** Bestimmt Business-Erfolg
- ***Kosten:*** Serverlast *(Infrastruktur-Kosten)*, Entwicklung *(Implemetation, Wartung)*
- ***User Experience:*** Time to interactivity, white screen, Reaktionen
- ***Sicherheit:*** Code-Offenlegung *(Remote Presentation Risk)*, Attack Surface
- ***Wartbarkeit & Testing:*** Logik, visuelle Validierung, Nachvollziehbarkeit

Server Side Rendering (SSR): Server rendert den gesamten HTML-Inhalt und sendet ihn an den Client. Seite sofort sichtbar und SEO-freundlich. Interaktivität benötigt weitere Technologien. ***Geeignet für:*** SEO-relevante Seiten mit wenig Traffic, Seiten mit personalisiertem Zugriff *(z.B. eCommerce)*.
Client Side Rendering (CSR): HTML und JS werden an den Browser gesendet, der die Seite dann rendert. Initial-Ladevorgang langsamer, dafür Seitenübergägne reaktiv und flüssig. ***Geeignet für:*** Hochinteraktive Anwendungen, die dynamische Inhalte unterstützen *(z.B. Dashboards)*.
Static Site Generation (SSG): HTML wird zur Build-Zeit vorab generiert und als statische Dateien bereitgestellt. Client erhält sofort fertige HTML-Seite. Extrem schnelle Ladezeit. ***Geeignet für:*** wenig dynamische Sites die schnelles Laden erfordern *(z.B. Doku-Seiten, Unternehmenswebseiten)*.
Deferred Static Generation / Incremental Static Regeneration (DSG, ISR): Seiten werden on-demand bei der ersten Anfrage generiert und als statische Dateien gecacht. Bereits generierte Seiten werden zeitgesteuert und ergebnisbasiert neu generiert, um Inhalte aktuell zu halten. ***Geeignet für:*** Grosse Content-Webseiten mit Mischung aus dynamischen und statischen Inhalten *(z.B. Nachrichtenplattformen, Produktseiten)*.
Edge-Side Rendering (ESR): HTML wird an Edge-Servern des CDN generiert, die geografisch näher zum Nutzer stehen. Kombination aus Server-Performance und globaler Verteilung für optimale Ladezeiten. ***Geeignet für:*** Globale Anwendungen mit lokalisiertem Content *(Internationale Shops, SaaS)*.

Core Web Vitals: ***Largest Contentful Paint*** *(LCP, grösstes Element geladen)*, ***First Input Delay*** *(FID, Erste Eingabe-Reaktion)*, ***Cumulative Layout Shift*** *(CLS, visuelle Stabilität)*.
Performance Metriken: ***Time to First Byte*** *(TTFB, Server-Antwortzeit)*, ***First Contentful Paint*** *(FCP, erstes sichtbares Element)*, ***Time to Interactive*** *(TTI, voll interaktionsfähig)*.
Erfolgs-Prinzipien: Anforderungen vor Technologie, Team-Expertise berücksichtigen, einfach starten und dann erweitern, Validierungs-Metriken verwenden

11.1. HYBRIDE RENDERING STRATEGIEN

Kombiniert Strategien. ***Performance:*** SSG für statische Inhalte, ***SEO:*** SSR für dynamische Inhalte, ***Interaktivität:*** CSR für User Interfaces, ***Flexibilität:*** Per-Page Strategien. Mit Frameworks wie Next.js ist das möglich. **Vorteile:** Einfaches Wählen der Rendering-Strategie, Performance *(Code Splitting, Image Optimization)*, file-based Routing, API Routes, Zero-Config.

11.1.1. Hydration

Schnell sichtbare Inhalte durch SSR, anschliessend Interaktivität auf dem Client durch JS und State. Dabei werden Event Listeners attached und State rekonstruiert. ***Hydration Mismatch:*** Server hat andere Daten als Client. ***Lösung:*** Server-Daten werden serialisiert und an Client übertragen. ***Vorteile:*** Schnellere Ladezeiten, SEO-freundlich, Benutzerinteraktion.
Full Hydration: Server rendert komplettes HTML, Client lädt alles, alles neu rendern *(Re-execution)*, dann Event Listeners attachen. ***Vorteile:*** Einfachste Implementierung, keine Architektur-Komplexität, einfaches Debugging, Bewährte Technologie. ***Nachteile:*** Performance bei Scale, grosse TTI, Verschwendete CPU durch doppelte Arbeit, Mobile Probleme.
Selective Hydration: Einteilung in ***Suspense Boundaries***. User Interaction triggern Hydration. Viewport-basiert: Sichtbare Bereiche zuerst. ***Vorteile:*** Schnellere Performance, intelligente Priorisierung, Mobile Performance. ***Nachteile:*** Komplexere Architektur, schwieriges Debugging *(non-deterministic)*, gleiche Bundle-Size. ***Time Slicing:*** Hydration wird in 5ms Chunks vorgenommen, Browser bleibt responsiv. ***Priority Lanes:*** zuerst UserBlocking *(klicks)*, dann sichtbarer Inhalt, dann versteckte Inhalte. ***Ergebnis:*** Keine Konfiguration nötig, Intelligente Unterbrechung, Smooth User Experience.
Concurrent Hydration: Mit React Fiber. ***Interruptible:*** Hydration pausiert für User-Interaktionen, ***Prioritized:*** Wichtige UI-Elemente werden zuerst hydriert, ***Progressive:*** Schrittweise Hydration ohne Blocking des UI.
Streaming Hydration: Server Stream Start, Parallel Processing, Chunk-wise Hydration & Progressive Enhancement. ***Vorteile:*** Schnellster First Paint, Parallel Processing, Mobile optimiert, Progressive UX, Bessere perceived Performance. ***Nachteile:*** Server-Komplexität, Framework-Abhängigkeit, Debugging schwierig wegen asynchroner Streams, Monitoring komplex. ***Verwenden bei:*** Slow Datenquellen, Content-Heavy Pages, Mobile First
Prefetching (Re-)Hydration: Link Hover Detection *(Prefetch bei Hover)*, Viewport Intersection, User Pattern Analysis, Background Rehydration. ***Vorteile:*** Instant Navigation, Smart Prediction, Mobile Performance, SPA-ähnlich, Cache Optimierung. ***Nachteile:*** Bandwidth Overhead *(ungenutzte Prefetches)*, Server Load, Mobile Battery, Prediction Accuracy. ***Optimierungsstrategien:*** Connection-aware, Data-saver respect, Priority-based, Analytics-driven. ***Verwenden bei:*** Multi-Page apps, E-Commerce, Browse Heavy Sites.
Partial Hydration *(Island Architecture)*: Statische Basis, Interaktive Inseln. ***Vorteile:*** Extreme Performance, minimal Bundle Size, Instant First Paint, Battery Life, Bessere SEO. ***Nachteile:*** Intensive Planung, Framework-Limits, State Management, Lernkurve. ***Verwenden bei:*** Landing Pages, Produkt Site
Progressive Hydration: Enterprise Pattern. Multi-Stage nach Priorität, Framework-gesteuert, Performance Bugdets, Business-Priorität. ***Vorteile:*** Optimales UX, Business-Orientiert, Framework-Support, Flexible Balance, Adaptivität ***Nachteile:*** Höchste Komplexität, Framework-Abhängigkeit, Monitoring komplex, Debugging schwierig. ***Use-Cases:*** E-Commerce, Financial, SaaS.

11.1.2. Resumability (Qwik)

Normale Hydration: Server rendert Component Tree, Client lädt JS Bundle, erstellt identischen Component Tree, und vereint JS mit DOM → Verursacht Performance Impact durch Double Work.
Resumability: Server rendert HTML & serialisiert State. Client lädt nur einen Loader *(z.B. QwikLoader)*, deserialisiert State aus HTML und ist sofort interaktiv. ***23x schnellere Time to Interactive, 200x kleineres Bundle.***
Qwik hat zum Ziel: 0ms to interactive, ***Instant Loading***, Serializable Applications, Dev Experience wie React. Setzt auf ***globalen Event Listener***. Der Optimizer ***transformiert Event Handler in URLs*** *(z.B. `./chunk-abc.js#handleClick`)*. Diese URLs werden als Strings im HTML serialisiert und der Code wird ***lazy*** beim Klick geladen. **Qwik-Serialisierung:** References als IDs statt Kopien bedeutet, dass ***circular References*** möglich sind. ***Ideal für:*** Performance-kritische Apps, Hohe Effizienz, perfekter Server/Client Sync. ***Herausforderungen:*** Entwickler Experience, Technische limits *(Nicht alle Datentypen serialisierbar, Build Complexity)*.

11.2. MICRO FRONTENDS

Warnsignale bei wachsenden Applikationen: ***Trägheit*** der Weiterentwicklung durch hohe Komplexität, ***Technologie-Stagnation*** durch Abhängigkeit, ***Team-Koordination*** schwierig, ***Inkonsistentes Design*** durch zu grosse Teams, ***Langsame Ladezeiten*** wegen zu grossen Bundles.

Micro Frontends: Teile grosse Anwendungen in ***unabhängige*** Frontend-Module, die separat entwickelt und deployed werden können. Ermöglichen modularisiete, flexible Frontend-Architektur.

Vorteile: Incremental Upgrades, ***Unabhängige*** Entwicklung, einfache Codebasis, unabhängige Teams, hohe ***Skalierbarkeit***, Nutzung ***unterschiedlicher*** Technologien möglich. **Nachteile:** ***Erhöhte Komplexität*** bei Integration, Testing und Monitoring und ***Performance Overhead*** durch duplicated Dependencies, mehr Netzwerk-Requests und grössere Bundle Sizes durch mehrere Frameworks.

11.2.1. Kernprinzipien

- ***Technology Agnostic:*** Jedes Team wählt seinen Stack *(Keine Abhängigkeiten, Experimentieren möglich. Shared Dependencies z.B. müssen aber koordiniert werden)*
- ***Isolated Team Code:*** Keine geteilte Runtime *(Fehler-Isolation, Independent Updates, Keine Konflikte. Isolation auf verschiedenen Ebenen wie Build-Zeit und Runtime)*
- ***Team Prefixes:*** Namespacing für CSS und Events *(Vermeidet Konflikte, macht Ownership klar ersichtlich, erleichtert Debugging. z.B. `[team]-[component]-[element]`)*
- ***Native Browser Features:*** Browser APIs nutzen *(weniger Dependencies, mehr Stabilität und Performance, Framework-agnostisch)*
- ***Build Resilient Sites:*** Progressive Enhancement *(Resilienz-Strategien wie Graceful Degradation, Fallback Components, Errr Boundaries. z.B. zeige Skeleton UI wenn Service langsam ist, Basic HTML funktioniert auch ohne JS)*

11.2.2. Integration Strategien

- ***Server-side:*** Templates & SSI *(Server Side Includes)*. Fragments zur Build-Zeit zusammenfügen, Edge Side Includes, Backend rendert komplette Seite. *(Perfektes SEO, Schnelles Laden, einfache Implelentation. Aber: Schwierig bei dynamischen Inhalten, limitierte Interaktivität, gemeinsame Deployment-Pipeline)*
- ***Build-Time:*** Nicht empfohlen, zerstört Hauptvorteile von Micro Frontends wie Autonomie und Unabhängigkeit. Shared Repository, Monorepo. *(Coordination Hell, Shared Dependencies, Build Failures, testing Bottleneck)*
- ***iFrames:*** Einfach, aber limitiert. *(Starke Isolation, Legacy Integration, Sicherheit)*
- ***JavaScript:*** Flexibelster Ansatz. *(Runtime-Integration ohne Build-Coupling, Shared State Management möglich, Error Isolation & Fallbacks)*
- ***Web Components:*** Browser-Nativ *(Keine Framework-Abhängigkeit)*

11.3. ENTITY COMPONENT SYSTEM (ECS)

Observer Pattern ist für ***komplexe Interaktionen*** wegen Memory Leaks, Event Listener Explosionen, Event Storm, Event Chain Debugging, unklarer State Synchronisation und durch Tight Coupling ***nicht geeignet***.

ECS bietet zentrale Koordination *(keine Event Listener nötig)*, Performance Optimierung durch Spatial Partitioning, Memory Effizienz *(Keine Event Listener, keine Memory leaks)*, einfaches Debugging, Skalierbarkeit und direkte Manipulation.

Time-Based Architecture: Gut für Animationen & Physik, real-time Simulationen, 1000+ bewegende Objekte.

Drei Kernkomponenten: ***Entity:*** ID-Nummer, ***Component:*** Pure Daten wie Position, Velocity, Health. ***System:*** Logik. Verarbeitet Entities mit bestimmten Components. Objekte werden durch Komposition von Components definiert.

Ablauf Game Loop: Alle Systems sequentiell durchlaufen, Entity mit passenden Components ausgewählt, System modifiziert Component, nächstes System.

12. MULTIPLATTFORM-FRONTENDS & .NET MAUI

12.1. MULTIPLATTFORM-FRONTENDS

Multiplattform-Frameworks sollen ***nativ-anfühlendes UI*** auf verschiedene Formfaktoren und Hardware mit ***unterschiedlichen*** (Design-)Anforderungen und Ressourcen bringen. Auch sollten die Features und der Use Case der Plattform ***angepasst*** werden *(Mobile: Simpel, Desktop: Viele Features)*. Darum baut man besser ***spezialisierte Apps***, anstatt eine universelle App mit Kompromissen.

Konsequenzen für die Entwicklung:

- ***Dev Experience und Tooling:*** Hot Reload *(Live Preview auf verschiedenen Plattformen)*, Remote Debugging auf echten Geräten, Emulatoren *(Performance-Unterschied)*, Wechseln zwischen Targets, Build Times *(separate Builds für jede Plattform)*, Dependencies *(Platform-spezifisch managen)*
- ***Teststrategien und Herausforderungen:*** Plattformspezifische Integration Tests, End2End Tests *(echte Geräte vs. Emulatoren)*, Manuelles Testing jeder Platform, Device Fragmentation *(5000+ Android Devices)*, Support verschiedener OS Versionen, komplexe CI/CD *(Builds und Integration Tests für jede Plattform)*
- ***Architektur und Maintenance:*** 60-80% shared code vs. 20-40% plattformspezifisch, Abstraktion von Plattform APIs?, DI für Plattform-Services oder Conditionals *(`if "iOS"`)*?, Updates auf mehreren OS, Breaking Changes, Technical Debt exponentiell mit vielen Plattformen
- ***Performance:*** Native schnell, Multiplattform-FWs häufig langsam und gross

Bei Code Sharing gilt: Je näher am OS/UI, desto schwieriger die Abstraktion.
Gut teilbarer Code: Business Logik, Datenmodelle & DTOs, API-Clients, State Management *(ViewModels, Stores, Services)*, Utilities *(Formatierung, Parsing, Caching)*.
Schlecht teilbarer Code: Plattform-spezifische Features *(Kamera, Biometrie)*, Native UI Elemente, Betriebssystem APIs, UI/UX Patterns *(Navigation, Menüstruktur)*

Native UI Approach	***Unified UI Approach***
UI Komponenten werden ***abstrahiert*** und auf die nativen Controls der Plattform gemapped.	UI Komponenten werden in einer eigenen Rendering-Engine gerendert.
+ Echtes Plattform Look & Feel	+ Einheitliches Look & Feel überall
+ UI automatisch vom OS angepasst *(z.B. iOS Liquid Glass-Update)*	+ Volle Kontrolle über UI/UX
+ Kleinere App-Grösse *(20-40MB)*	+ Oft bessere Animations-Performance
+ Zugriff plattformspezifische APIs	– Grössere App *(10-200MB)*
– Unterschiedliches UI pro Plattform	– Plattform-Updates verzögert
– Weniger UI Kontrolle	– Fühlt sich nicht nativ an
– Starke Framework-Abhängigkeit	

Durch Frameworks wird ***Abstraktion*** der Plattform und des UX, eine ***gemeinsame Codebase*** und ***einheitliches*** Tooling versprochen. In der Realität hat man oft ***Framework-Abhängigkeiten***, ***Abstraktions-Grenzen*** *(nicht alle Features abbildbar)*, Bindung an Update-Zyklen und Debugging-Komplexität. Die Verwendung ist ein Trade-off zwischen ***Developer Convenience*** und ***Plattform-Flexibilität***.
Native UI Frameworks: .NET MAUI, React Native, Qt, Kotlin Multiplat., Swift UI
Unified UI Frameworks: Flutter, Avalonia UI, Uno Plattform, Electron, Tauri, .NET MAUI Blazor Hybrid

12.2. UI-PRINZIPIEN

Klassisches Imperatives UI

Beschreibt, wie UI ***aufgebaut*** wird, ***manuelle*** Objekterstellung und Konfiguration, ***explizite*** Event-Handler Registrierung, direkte ***State-Mutation*** erlaubt, Schritt-für-Schritt Konstruktion, keine automatische Reaktivität.
Beispiele: Windows Forms, Java Swing, iOS UIKit, GTK.

```
JPanel panel = new JPanel(); // Java Swing
panel.setLayout(new BoxLayout(panel, BoxLayout.Y_AXIS));
JLabel label = new JLabel("Hello World");
label.setFont(new Font("JetBrains Mono", Font.PLAIN, 24));
label.addActionListener(e → {/*...*/}); panel.add(label);
```

XML-basierendes Imperatives UI

Reine ***Markup-Sprache*** *(nicht alleine ausführbar)*, ***getrennt*** von Logik *(Code Behind)*, keine eingebetteten Expressions, ***statische Struktur*** zur Design-Zeit, Plattform-spezifische ***XML-Dialekte***, Tooling-Support *(VS Designer, IntelliSense)*.
Beispiele: XAML *(WPF/MAUI)*, Android XML, Java FX FXML, XUL, Glade

```
<LinearLayout android:orientation="vertical"> <!—Android XML—>
 <TextView android:text="Hello World" android:textSize="24sp" />
 <Button android:text="Start" android:onClick="onButtonClick" /></LinearLayout>
```

Template-basierendes Deklaratives UI

HTML-Templates mit ***Framework Directives***, Logik ***separiert***, Template-Expressions möglich, ***Two-Way Data Binding*** oft unterstützt, ***reaktive*** Updates *(Template re-rendert bei State-Änderung)*. ***Beispiele:*** Angular, Vue.js

```
@Component({ // Angular Component
 selector: 'app-example',
 template: `<div><h1 [style.fontSize.px]="24">Hello There</h1>
 <button (click)="onClick()">Start</button>
 <ul><li *ngFor="let i of items">{{ i }}</li></ul></div>`})
export class ExampleComponent { items = ['Nina', 'Jannis'];
  onClick() {/*...*/} }
```

Code-basierendes Deklaratives UI (modern)

In Programmiersprache ***eingebettet***, kann Expressions und Logik ***inline*** enthalten, ***reaktive*** Updates *(UI reflektiert State)*, ***Side Effects*** erlaubt, Stateful Components *(`useState, State<T>`)*, Component Composition, UI und Logik können ***gemischt*** werden. ***Beispiele:*** Flutter, Swift UI, Jetpack Compose, React, SolidJS, Svelte.

```
Column( // Flutter (Dart (JVM-basiert))
 children: const [Text('Hoi!', style: TextStyle(fontSize: 24)),
   ElevatedButton(onPressed: null, child: Text('Click Me')),],)
```

Rein Funktionales Deklaratives UI

Rein ***Funktionale*** Paradigmen, ***Immutable*** State *(keine Objektmutationen)*, Strikte ***Typsicherheit*** zur Compilezeit, ***Unidirectional*** Data Flow, ***Seiteneffekte*** im Typensystem *(Elm: Commands, Haskell: Monads)*, keine impliziten Seiteneffekte in View-Funktionen. Elm: Garantiert keine Runtime Exceptions.

Beispiele: Elm, Reflex-DOM *(Haskell)*

```
view :: MonadWidget t m  ⇒ Dynamic t Model → m () -- Haskell
view model = el "div" $ do
    el "h1" $ dynText
      $ fmap (const "Hello World") model
    button ← button "Click Me"
    pure ()
```

12.3. WAS IST .NET MAUI?

Microsoft .NET MAUI (Multi-platform App UI) wurde 2022 als Nachfolger von Xamarin.Forms mit .NET 6 eingeführt *(Aktuell ist .NET 10)*. ***Zweck:*** Vereinheitlichung der Cross-Plattform-Entwicklung mit einem einzigen Projekt für alle Plattformen *(Write once, run anywhere)*. Wie alle .NET-Sprachen verwendet es ***nuget*** als Paketmanager. MAUI basiert auf dem ***MVVM-Prinzip*** *(«MVVM» (Seite 6))*. ***Vorteile***: Abstrahierte UI-Komponenten, Hot Reload, Native OS-/Hardwarezugriffe. MAUI unterstützt nativ Android, iOS, Windows, macOS und Web *(Linux jedoch nicht)*. Vor nativem .NET-Port auf Linux war ***Mono*** die genutzte .NET Runtime auf Linux.

Das UI wird durch ***XAML*** definiert, eine deklarative Sprache basierend auf ***XML***, die zusammen mit C# zu UI Elementen kompiliert wird. Es ermöglicht die ***Trennung*** von Layout und Code. Je nach Framework gibt es andere Dialekte, MAUI verwendet ***MAUI XAML***. Die XAML-Baumstruktur ist auch über C# definierbar, aber XAML ist oft ***leichtgewichtiger*** und kürzer, insbesondere bei Verschachtelungen.

Neben dem klassischen .NET MAUI gibt es ***.NET MAUI Blazor Hybrid***, ein Unified UI Web-Framework, welches in WebView *(Chromium)* gerendert wird. Durch die Verwendung von Blazor kann C# anstatt JS für Client-Code verwendet werden. Es wird ***kein WebAssembly verwendet***, darum ähnliche Performance wie nativer .NET-Code. WebDev-Tooling *(Hot Reload, Browser Dev Tools)*.

12.4. AUFBAU

Startup eines MAUI-Programms

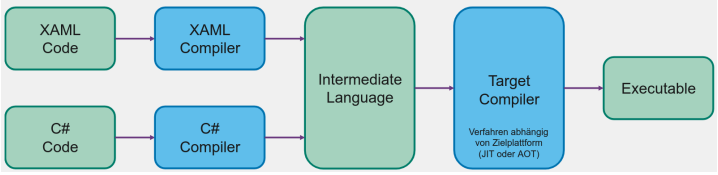
`Platforms` → `MAUI Program` → `App` → `App Shell` → `Main Page`

Dateien in einem MAUI-Projekt

- ***`Platforms`-Ordner:*** Plattform-spezifischer (Startup-)Code
- ***`Resources`-Ordner:*** Von allen Plattformen verwendete Ressourcen
- ***`App.xaml`:*** Einstiegspunkt in MAUI-Applikation
- ***`AppShell.xaml`:*** Definition der visuellen Hierarchie mittels Shell (optional) *(z.B. Pages, Flyoutmenüs, Tabs etc.)*
- ***`MainPage.xaml`:*** Inhalt des ersten Fensters der App
- ***`MauiProgram.cs`***: Bootstrapping der MAUI-Applikation *(Builder)*

Zusammensetzung einer Main Page (Drei `partial`-Klassen)

- ***XAML-Markup:*** Definiert die UI-Elemente als hierarchischer Baum mit Bindings und Ressourcen. Wird zu C# kompiliert.
- ***C# Code-Behind:*** Die Logik hinter den UI-Elementen *(z.B. Event-Handler)*
- ***Generierter C# Code:*** Verknüpft die XAML-Elemente mit dem Code-Behind, generiert entsprechende Felder



13. XAML

Die ***Extensible Application Markup Language (XAML)*** ist eine XML-basierende Beschreibungssprache zur ***Gestaltung*** grafischer Oberflächen. Sie ist ***hierarchisch*** als Baum strukturiert. Durch sie lässt sich Layout und Code voneinander trennen.

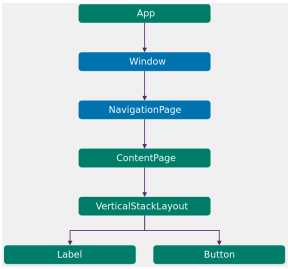
UI laden:

```
public partial class MainPage : ContentPage {
  public MainPage() {
    InitializeComponent(); // With XAML, you only need this line
    // If you want to use Imperative UI (code will override XAML):
    var button = new Button { Text = "OK", WidthRequest = 60, Margin = 5};
    var label = new Label { /*...*/ };
    var stackLayout = new VerticalStackLayout { label, button };
    this.Content = stackLayout; }}
```

13.1. XAML TREES

Das XAML kann als ***zwei verschiedene Trees*** angezeigt werden. Diese Trees funktionieren ähnlich wie der ***DOM*** auf Webseiten.

Ein ***`StackLayout`*** mit 2 Labels besteht aus 3 Logischen Elementen, hat aber je nach Platform 10+ native Views. Beispielsweise hat ein Button auf Android 3 native Views:
`ButtonHandler`→`MaterialButton`→`TextView`



Visual Tree (grün + blau): Vollständiger, visuell dargestellter Baum. Ist ***Plattformspezifisch*** *(abhängig von Handler & nativen Controls)* und enhält automatisch generierte Knoten *(Templates, Wrappers)*. ***Verwendung:*** Layout-Debugging *(Warum wird ein Element nicht angezeigt?)*, Performance-Analyse *(Wie viele native Views werden erstellt?)*, Handler-Probleme *(MAUI Control-native View-Mapping)*.

Logical Tree (grün): Nur explizit definierte XAML-Elemente. ***Plattformunabhängig***. Die Grundlage für Ressourcen, BindingContext und Navigation. ***Verwendung:*** Ressourcen-Lookup, BindingContext-Vererbung, Navigation & Hierarchie, Code-Operationen *(`FindByName`)*

13.2. XAML-GRUNDLAGEN

13.2.1. XML Namespaces

Mit dem ***`xmlns`-Attribut*** auf dem Root-Element werden ***Namespaces*** definiert.

Standard-Namespace: ohne Doppelpunkt *(Präfix optional)*

Benannter Namespace: mit Doppelpunkt *(nur mit Präfix verwendbar)*

```
<?xml version="1.0" encoding="utf-8" ?><Application
  xmlns="http://schemas.microsoft.com/dotnet/2021/maui"
  xmlns:x="http://schemas.microsoft.com/winfx/2009/xaml"
  xmlns:local="clr-namespace:UIP_Vorlesung_10"
  x:Class="Vorlesung_10.App"><!— ... —></Application>
```

Übliche Namespaces in MAUI:

`xmlns:` Standard-Namespace für MAUI-Control Library, ***`xmlns:x:`*** XAML-spezifische Elemente, ***`xmlns:local:`*** Eigene Controls/ViewModels aus dem Projekt, ***`x:Class`***: Name der Code-Behind-Klasse im Namespace `x` *(Üblicherweise immer x, kann aber auch anders benannt werden)*.

13.2.2. Named Elements

Elemente können ***benannt*** werden. Ermöglicht Zugriff im Code-Behind. Attribut führt zum Property in der generierten Klasse. MAUI kennt nur `x:Name`-Attribut.

```
<Label x:Name="MyLabel" /> bzw. this.MyLabel.Text = "El Töni";
```

13.2.3. Event Handler

Reaktion auf Ereignisse der UI Controls. Methode wird im XAML registriert und im Code Behind implementiert. Immer zwei Parameter: ***`object`***: Auslöser des Events, ***`EventArgs`***: Event-spezifische Argumente, abgeleitet von `EventArgs`.

Achtung: Gefahr von Top-Down Abhängigkeit!

```
<Button Clicked="OnClick" Text="Start" />
private void OnClick(object sender, EventArgs args) { /*...*/ }
```

13.2.4. Type Converter

Um Werte aus dem UI in andere Typen zu konvertieren, können eigene Typen-Konverter von `TypeConverter` abgeleitet werden.

```
<local:LocationControl Center="10, 20">

public partial class LocationControl : Label { // Code Behind
  public Location Center { set ⇒ this.Text = $"{value.Lat}/{value.Long}"; } }

[TypeConverter(typeof(LocationConverter))] // Data Model
public class Location {
  public double Lat { get; set; } public double Long { get; set; } }

public class LocationConverter : TypeConverter {
  public override object ConvertFrom(
    ITypeDescriptorContext ctx, CultureInfo culture, object value) {
    // Missing checks: Array contains 2 elems? Strings convertible to double?
    var valueAsString = (string)value;
    var valueArray = valueAsString.Split(',');
    return new Location { Lat = Convert.ToDouble(valueArray[0]),
      Long = Convert.ToDouble(valueArray[1]) } } }
```

13.2.5. Property Element vs. Attribute Syntax

XAML kann mit der ***Property Element Syntax*** oder der ***Attribute Syntax*** geschrieben werden. Die Attribute Syntax ist kompakter, benötigt aber Type Converter *(z.B. String `"Red"` zu `Color` konvertieren)*. Die Property Element Syntax wird für komplexe Objekte oder wenn kein Type Converter vorhanden ist verwendet.

```
<Label Text="Attribute Syntax"
TextColor="Red" Background="Blue"/>
```

```
<Label Text="Property Element Syntax">
  <Label.TextColor>
    <Color>Red</Color>
  </Label.TextColor>
  <Label.Background>
    <SolidColorBrush>
      <SolidColorBrush.Color>
        <Color>Blue</Color>
      </SolidColorBrush.Color>
    </SolidColorBrush>
  </Label.Background>
</Label>
```

13.2.6. Content Properties

Jede ***XAML-Klasse*** kann genau ***eine Eigenschaft*** mit der `[ContentProperty()]`-Annotation als ihren Inhalt definieren. Diese kann dann in verkürztem Syntax in das Element geschrieben werden.

```
<MyLabel                                    // Annotation sets "main field"
  Text="Normales Property" />               [ContentProperty(nameof(Text))]
<MyLabel>Content Property                   public class MyLabel : View {
</MyLabel>                                    public string Text { get; set; } }
```

Elemente, welche andere Elemente enthalten, können ebenfalls via Content Property gesetzt werden, um die Lesbarkeit von Beziehungen zu fördern.

```
<VerticalStackLayout>           <VerticalStackLayout>
  <Label Text="Inhalt" />         <VerticalStackLayout.Children>
  <Label>Inhalt</Label>             <Label Text="Inhalt" />
</VerticalStackLayout>              <Label>Inhalt</Label>
<!--Oben: Content Property S.     </VerticalStackLayout.Children>
Rechts: Property Element S.--> </VerticalStackLayout>
```

13.2.7. Attached Properties

Setzen einer Eigenschaft auf einem Element, welche zu einem anderen Element gehört – sie wird dem anderen Element ***angehängt***. Wird meist bei Layouts angewendet, diese müssen gewisse ***Werte*** für die Gestaltung ***kennen***. Die Kind-Elemente definieren diese Werte. Fördert Lesbarkeit des XAML. Im Beispiel wird `Grid.Row` anstatt direkt auf `Grid` auf den Children gesetzt. Equivalenter C# Code: `Grid.SetRow(R, 0); Grid.SeRow(G, 1); Grid.SetRow(B, 2)`

```
<Grid RowDefinitions="3*,2*,1*" >
  <Label Grid.Row="0" x:Name="R" Background="Red" />
  <Label Grid.Row="1" x:Name="G" Background="Green" />
  <Label Grid.Row="2" x:Name="B" Background="Blue" /></Grid>
```

13.2.8. Markup Extensions

Erlauben Logik zur ***Compile-Zeit*** in XAML einzubetten.

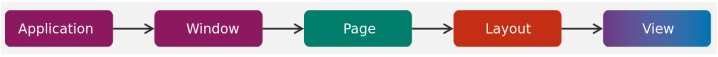
Syntax: `{Präfix:Klassenname Property=Value}`. **Spezialfälle:** `{x:Null}`: Null-Wert, `{Binding Name}`: Data Binding, `{StaticResource ButtonStyle}`: Ressourcen-Lookup. Eigene Extensions können durch Implementieren von `IMarkupExtension` erstellt werden. ***Verwendung:*** Translations, Formatierungen, Ressourcen-Lookup.

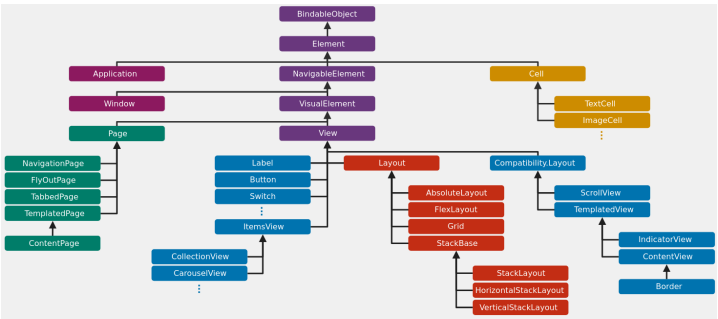
```
<Label Text="{local:Translate Key='WelcomeMessage'}" />
```

```
public class TranslateExtension : IMarkupExtension {
  public string Key { get; set; }
  public object ProvideValue(IServiceProvider sp) {
    // Look up translation in resources
    return AppResources.ResourceManager.GetString(Key); } }
```

14. GUI-GRUNDELEMENTE IN XAML

(→ bedeutet: «Hat 1 oder mehr davon»)





Violett: *Basisklassen,* ***Weinrot:*** *Application & Window,* ***Grün:*** *Pages,* ***Rot:*** *Layouts,* ***Blau:*** *Views & Controls,* ***Gelb:*** *Cells*

14.0.1. Basisklassen

Die abstrakten Basisklassen ergänzen schrittweise ***weitere Funktionalität***. Sie stehen allen ableitenden GUI-Elementen ***zur Verfügung***. Nicht alle Klassen leiten alle Basisklassen ab – nicht alle GUI-Elemente besitzen alle Attribute *(Alle `View`-Klassen haben Margin, aber nur `Layout`-Klassen auch Padding).*

Klassen: ***`BindableObject`:*** Implementation von Data-Binding, Validierung, Typenkonvertierung und Events. ***`Element`:*** Basisklasse für hierarchische Steuerelemente mit allen notwendigen Methoden und Eigenschaften. ***`NavigableElement`:*** Unterstützt Navigation. ***`VisualElement`:*** Bildschirm-Element, das Platz zur Anzeige erhält, visuelle Darstellung hat und Touch-Eingabe haben kann. ***`View`:*** Visuelles Element zur Platzierung von Layouts und Steuerelementen.

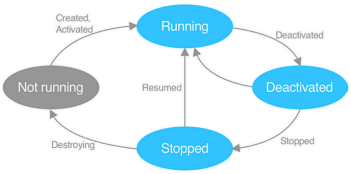
14.1. APPLICATION & WINDOW

Die ***`Application`***-Klasse legt über das ***`MainPage`-Property*** den ersten Screen fest. Ebenfalls erzeugt und verwaltet sie ***Fenster***. Das Hauptfenster wird via ***`CreateWindow()`***, weitere Fenster über ***`Application.Current.OpenWindow()`*** geöffnet *(iOS unterstützt kein Multiwindowing)*. Um auf ***Window-Lifecycle-Events*** *(Window erstellt/geschlossen)* zuzugreifen muss `CreateWindow()` überschrieben werden *(Application-Lifecycle-Methoden sind seit MAUI 9 deprecated)*. Die Application-Klasse ermöglicht die zentrale Verwaltung von App-weiten XAML-Ressourcen.

```
public partial class App : Application {
  public App() { InitializeComponent(); MainPage = new FirstPage();}
  // Zugriff auf Window für Konfiguration & Events
  protected override Window CreateWindow(IActivationState actSt) {
    var window = base.CreateWindow(actSt);
    window.Title = "Name des Fensters"; // Fenster konfigurieren
    // Window-Lifecycle-Events abonnieren
    window.Created += (s, e) => { /* ... */ };
    window.Activated += (s, e) => { /* ... */ };
    return window; } }
```

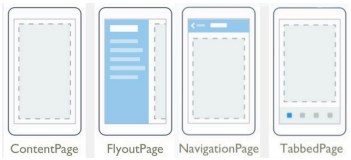
14.1.1. Lifecycles

MAUI leitet wichtige ***Lifecycle-Events*** der darunterliegenden Plattformen auf ***Window-Ebene weiter***. Im Multi-Window-Modus hat jedes Window seinen eigenen Lifecycle. Lifecycle-Events sind nur auf der Window-Ebene verfügbar.



14.2. PAGES

Elemente zur ***Strukturierung*** und ***Gestaltung*** ganzer Screens. Sie füllen normalerweise ihre Eltern-Windows vollständig aus. ***Verschachtelung*** von Pages ist üblich *(z.B. `ContentPage` innerhalb von `NavigationPage` oder `TabbedPage`).*



- ***`ContentPage`:*** Leerer Screen ohne Zusatzelemente
- ***`FlyoutPage`:*** Slide-in Menu von Links *(«Hamburger-Menu»)*
- ***`NavigationPage`:*** Hierarchische Navigation mit Toolbar
- ***`TabbedPage`:*** Wechsel zwischen Tabs

Hauptgrund Verwendung von Page Typen: Verschiedene Arten zu navigieren.

14.2.1. NavigationPage

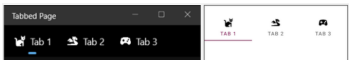
Die Navigation kann auf 3 verschiedene Arten eingerichtet werden:

1. ***`Application.MainPage`:*** Austausch der angezeigten Page, genügt für sehr einfache Apps
2. ***`NavigableElement.Navigation`:*** Erlaubt hierarchische Navigation *(Stack)*, Modale Navigation *(Benutzer kann nicht zur vorherigen Seite zurückkehren, ohne sie zu schliessen)* funktioniert immer, Mode-less nur mit `NavigationPage`
3. ***Shell:*** Navigation auf Basis von URIs, viele Eigenheiten und Spezialfälle

```
// Option 1: App.xaml.cs
public partial class App : Application {
  public App() { InitializeComponent();
    MainPage = new NavigationPage(new FirstPage());  } }
// Option 2: FirstPage.xaml.cs
private async void NextPage(object s, EventArgs e) {
  await Navigation.Push(new SecondPage()); }
// Option 2: SecondPage.xaml.cs
private async void PrevPage(object s, EventArgs e) {
  await Navigation.PopAsync(); }
```

14.2.2. Tabbed Page

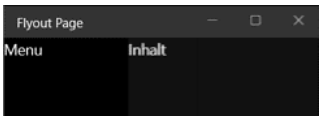
Erstellt standardmässig eine Tab-Leiste oben im Fenster.




```
<TabbedPage>
  <ContentPage Title="Tab 1" IconImageSource="i1.png" />
  <ContentPage Title="Tab 2" IconImageSource="i2.png" />
  <ContentPage Title="Tab 3" IconImageSource="i3.png" />
</TabbedPage>
```

14.2.3. Flyout Page

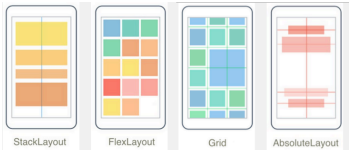
Die NavigationPage im XAML ist nötig, damit auf Android eine ***Navigation Bar*** inklusive Menü-Icon dargestellt wird. Passt sich nach OS grössenabhängig an. Bei wenig Platz erscheint oben links ein Menü-Icon fürs Ein-/Ausklappen.




```
<FlyoutPage>
  <FlyoutPage.Flyout>
    <ContentPage Title="Menu">
      <Label Text="Menu" />
    </ContentPage>
  </FlyoutPage.Flyout>
  <FlyoutPage.Detail>
    <NavigationPage>
      <x:Arguments>
        <ContentPage Title="Inhalt">
          <Label Text="Inhalt" />
        </ContentPage>
      </x:Arguments>
    </NavigationPage>
  </FlyoutPage.Detail>
</FlyoutPage>
```

14.3. LAYOUTS

Elemente zur ***Ausrichtung*** und ***Gruppierung*** von Views. Layouts sind Container für Kind-Elemente: Parent-Child Beziehung *(Composite Design-Pattern)*, Verschachtelung möglich.



- ***`StackLayout`:*** Horizontale oder Vertikale Anordnung
- ***`FlexLayout`:*** Ähnlich wie Stack, mit Wrapping & mehr Gestaltung
- ***`Grid`:*** Anordnung in Zeilen und Spalten
- ***`AbsoluteLayout`:*** Absolute oder proportionale Anordnung im Layout

14.3.1. Stack Layout

Das `StackLayout` existiert aus ***Kompatibilitätsgründen*** mit ***Xamarin***. In .NET MAUI sollten die optimierten Ableitungen verwendet werden. Abgesehen von fehlenden Orientation-Attribut funktionieren sie gleich.

```
<StackLayout Orientation="Horizontal"></StackLayout>
<StackLayout Orientation="Vertical"></StackLayout>
<VerticalStackLayout></VerticalStackLayout> <!— Optimierte Abl. —>
<HorizontalStackLayout></HorizontalStackLayout> <!— Opt. Abl. —>
```

14.3.2. Grössenangaben für Views

Layouts verwenden die ***`HorizontalOptions`/`VerticalOptions`-Attribute*** der gruppierten Kinder für deren Ausrichtung. Grössenangaben auf Views *(`WidthRequest`/`HeightRequest`)* sind optional, sonst so gross wie nötig dargestellt. Die Grössenangaben sind ***Device-independent Units*** *(50 doppelt so gross wie 25)*. Alle Grössenangaben sind Wünsche an die Rendering Engine *(`MinWidthRequest` ≤ `WidthRequest` ≤ `MaxWidth Request`)*

Gültige Werte für `HorizontalOptions`/`VerticalOptions`:
Start *(links oben)*, ***Center*** *(zentriert)*, ***End*** *(rechts unten)*, ***Fill*** *(Füllt den verfügbaren Platz)*.
Verfügbarkeit Padding/Margins: ***Pages*** *(nur Padding)*, ***Layouts*** *(Padding & Margin)*, ***Views*** *(Padding & Margin wo sinnvoll)*
Gültige Padding/Margins: ***`n`*** *(Gleicher Wert für alle Seiten)*, ***`x,y`*** *(horizontal/vertikal)*, ***`l,t,r,b`*** *(Left, Top, Right, Bottom – Achtung: anders als in CSS!)*

```
<VerticalStackLayout>
 <Label Text="K1" HorizontalOptions="Start"></VerticalStackLayout>
```

14.3.3. Flex Layout

Flexible Variante des Stack Layouts *(Komplizierter & Langsamer)*. Wichtige Eigenschaften: ***`Direction`*** *(Richtung der Kinder)*, ***`Wrap`*** *(Automatischer Umbruch)*, ***`JustifyContent`***: *(Verteilung in Hauptrichtung)*, ***`AlignItems`*** *(Verteilung in Nebenrichtung)* `<FlexLayout Direction="Row" Wrap="Wrap"></FlexLayout>`

14.3.4. Grid Layout

Erlaubt ***Stapelung*** von Elementen *(mehrere Element innerhalb einer Zelle)*. Abstand zwischen Zellen *(`ColumnSpacing`)* und Reihen *(`RowSpacing`)* sowie Verbindung zwischen Zellen *(`ColumnSpan`)* und Reihen *(`RowSpan`)* möglich.
Spezial-Werte: ***`*`*** *(Rest des Platzes)*, ***`N*`*** *(Proportionaler Anteil an Gesamtfläche)*, ***`Auto`*** *(Anpassen auf die Children, rechenaufwändig)*.

```
<Grid><Grid.RowDefinitions>
  <RowDefinition Height="1*" /><RowDefinition Height="2*" />
  <RowDefinition Height="3*" /></Grid.RowDefinitions>
<!—Same for ColumnDefinitions. "Grid.Row" = Attached Property—>
<Label Grid.Row="0" Grid.Column="0" /> <!—Same for others—></Grid>
```

Die Definition kann auch direkt in `Grid` erledigt werden:

```
<Grid RowDefinitions="50,Auto,*" ColumnDefinitions="*,*,/" ></Grid>
```

14.3.5. Absolute Layout

Oft verwendet, um ***Overlays*** zu gestalten. Zwei Varianten: ***1) Absolute Werte:*** Positionierung ab linker, oberer Ecke. Selten verwendet wegen Gerätevielfalt, ***2) Proportionale Werte:*** Bezug auf Grösse des Elternelements, kombinierbar mit absoluten Werten. ***LayoutBounds***-Property: Abstand linker Rand, Abstand rechter Rand, Element-Breite, Element-Höhe

```
<AbsoluteLayout LayoutBounds="20,10,60,20">Absolut</AbsoluteLayout>
<AbsoluteLayout LayoutBounds="0.5,0.5,0.5,0.5">Proport.</AbsoluteLayout>
```

14.4. VIEWS / CONTROLS

Elemente, die ***interagierbar*** sind. Alle Views erben von `VisualElement`.
Gemeinsame Eigenschaften: ***Aussehen*** *(`BackgroundColor`, `Opacity`)*, ***Zustand*** *(`IsVisual`, `IsEnabled`)*, ***Grösse*** *(`Margin`, `WidthRequest`, `HeightRequest`)*, ***Transformationen*** *(`Rotation`, `Scale`, `Translation`)*

- ***Display Views:*** Nur Anzeige von Infos, keine Interaktion *(Label, Image)*
- ***Interactive Views:*** Benutzereingaben, lösen Events aus *(Button, Entry)*
- ***Container Views:*** Enthalten andere Views, erweitern Funktionalitäten von Kind-Elementen *(ScrollView, Border)*

14.4.1. Events

Views unterstützen verschiedene Events: ***Tippen*** *(`Tapped`)*, ***Fokusänderungen*** *(`Focused`, `Unfocused`)*, ***Eigenschaft-Änderungen*** *(`PropertyChanged`)*, ***Grössen-Änderungen*** *(`SizeChanged`)*. Viele Views haben zusätzlich eigene Events: `Button` *(`Clicked`, `Pressed`, `Released`)*, `Entry` *(`TextChanged`, `Completed`)*, `Switch` *(`Toggled`)*.

14.4.2. Collection Views

Inhalte ***variablen Umfangs*** darstellen. ***`ListView`*** *(Einfache Listen)*, ***`TableView`*** *(Gruppierte Listen)*, ***`Picker`*** *(Auswahl: 1 von N)*, ***`CarouselView`*** *(Horizontales Swiping)*.
Wichtige Properties: ***`ItemsSource`*** *(darzustellende Collection)*, ***`ItemTemplate`*** *(Darstellung einzelner Items, meist via `ItemTemplate.DataTemplate`)*, ***`ItemsLayout`*** *(Linear/Grid Layout)*, ***`SelectionMode`*** *(Anzahl auswählbarer Elemente: `None`, `Single`, `Mutiple`)*

14.5. PLATTFORMSPEZIFISCHE ANPASSUNGEN

Unterschiedliche UIs nach Plattform mit ***`OnPlatform`*** *(nach OS)* und ***`OnIdiom`*** *(nach Gerätetyp)*. Default-Wert sollte immer gesetzt werden. **Anwendungsfall:** Verschiedene Controls je nach Plattform *(nur mit Property Element Syntax)*

- ***`OnPlatform`:*** `iOS`, `Android`, `MacCatalyst`, `WinUI` *(Compile-Time Definition)*
- ***`OnIdiom`:*** `Phone`, `Desktop`, `Tablet`, `TV`, `Watch` *(Runtime Definition)*

```
<Label Text="{OnIdiom 'Default', Phone='On a Phone'}" />
<OnPlatform x:TypeArguments="View">
  <On Platform="iOS"><Label Text="iOS gets Label"/></On>
  <On Desktop><Button Text="Desktop gets Button"</On></OnPlatform>
```

15. .NET MAUI DESIGN

Das Aussehen von Views wird über Attribute beeinflusst.

15.1. BILDER

In `Resources/Images` ***JPG***, ***PNG*** oder ***SVG*** ablegen. SVGs werden in PNG konvertiert *(schnelleres Rendering & bessere Grössenanpassungen für verschiedene DPIs)*. Image kann auch GIF-Animationen anzeigen, Web-Bilder *(inkl. Lade-Indikator & Caching)* und Bilder aus DLLs/Assemblies und Byte Arrays laden. Aus App Icon & Splash Screen werden ***plattformspezifische*** Elemente automatisch ***erzeugt***.
Aspect kontrolliert Skalierung: ***`AspectFit`*** *(Bildverhältnis beibehalten)*, ***`AspectFill`*** *(Fläche füllen, Verhältnis beibehalten)*, ***`Fill`*** *(Fläche füllen, Bildverhältnis ignorieren)*, ***`Center`*** *(gemäss Originalgrösse zentriert darstellen)*
Dateinamen: Lowercase, alphanumerisch und mit Underscores.
`<Image Source="ost_logo_v1.svg" Aspect="AspectFit" />`

15.2. SCHRIFTEN

In `Resources/Fonts` ***TTF/OTF-Datei*** ablegen. Muss im `Builder` registriert werden. Danach im XAML mit `FontFamily` setzbar. OS spezfische Schriftgrösse wird auf alle Views mit Text angewendet. ***Icons*** in Schriftarten *(FontAwesome)* mit `FontImageSource` verwenden.

```
var builder = MauiApp.CreateBuilder();
builder.UseMauiApp<App>().ConfigureFonts(fonts ⇒ {
 fonts.AddFont("JetBrains-Mono.ttf", "JetBrainsMono") });
<Label Text="print('Hello World!')" FontFamily="JetBrainsMono" />
```

15.3. FARBEN

Meist mehrere Varianten, wie Farbe zugewiesen werden kann *(Label: Background (Brush), BackgroundColor (Color))*. ***`Color`*** enthält Farbe, ***`Brush`*** Farbverläufe. `Color` wird intern zu `SolidColorBrush` umgewandelt. Brush hat Subtypen wie `LinearGradientBrush`.

```
<Label Text="Brush"><Label.Background>
  <LinearGradientBrush StartPoint="0,0" EndPoint="0,1">
  <GradientStop Offset="0.0" Color="#6E1C50" />
  <GradientStop Offset="0.5" Color="#AE5C90" />
</LinearGradientBrush></Label.Background></Label>
```

15.4. ANIMATIONEN

Alle ***`VisualElement`***-Objekte unterstützen: `Fading`, `Scale`, `Rotate`, `Translate`. Sie unterstützen variable Zeitdauern & Easing *(Default: 250ms & linear)*

15.5. RAHMEN & SCHATTEN

`Border` *(Container Control)* zeichnet Rahmen & Hintergründe. Ermöglicht Formen, Farbverläufe, Linienmuster, Padding. Schatten werden über ***`Shadow`*** gesetzt. ***`Shape`*** beschreibt geometrische Formen.

```
<Border StrokeShape="RoundRectangle 20,20,20,20"><Border.Shadow>
  <Shadow Brush="White"></Border.Shadow></Border>
```

15.6. RESSOURCEN

Beliebiges Objekt, das in XAML definiert werden kann *(Brush, Color etc.)*. Besitzt `x:Key` zur Identifikation. ***Ziel: Wiederverwendung***.
`VisualElement.Resources` = lokal, `Application.Resources` = global.

- ***Statische Resourcen `{StaticResource Key}`:*** Einmalige Auswertung beim Laden der Page, Dictionary-Lookup zur Runtime. Schneller als Dyn. Resourcen.
- ***Dynamische Resourcen `{DynamicResource Key}`:*** Wiederholte Auswertung, Behählt Link zum Dictionary Key zur Laufzeit. Reagiert auf Änderungen im Dict.

Die gleiche Resource kann sowohl mit `StaticResource` als auch mit `DynamicResource` referenziert werden.
Suchreihenfolge: Aktuelles Element → Parent-Elemente → `Application.Resources`. Suche bricht beim ersten Treffer ab.
Schlüssel ***nicht gefunden***: Ignoriert *(XAML)*, Exception *(C#)*.

```
<ContentPage><ContentPage.Resources>
  <SolidColorBrush x:Key="OSTBrush">
</ContentPage.Resources></ContentPage>
<Label Background="{StaticResource Key=OSTBrush}" />
```

```
var brush = Resources["OSTBrush"] as Brush;
```

15.6.1. Resource Dictionaries

Ressourcen können in einem ***Resource Dictionary*** gespeichert werden. Code-Behind optional. Nimmt alle XAML-Elemente auf. Basistypen können im XAML definiert werden, mit entsprechendem Import. Kann mit anderen Dictionaries gemerged werden. Wird entweder in `Application.Resources`, `VisualElement.Resources` oder als eigenständige `.xaml`-Datei definiert.
Priorität bei Schlüsselkollisionen: 1. Lokale Resources 2. Merged Resources *(Später definierte Merges überschreiben frühere Merges!)*

```
<!— MyDict.xaml, import namespaces for C# types —>
<ResourceDictionary xmlns:s="clr-namespace:System;assembly=System.Runtime">
  <s:Double x:Key="Margin">2</s:Double></ResourceDictionary>
<!— MainWindow.xaml —>
<ResourceDictionary><ResourceDictionary.MergedDictionary>
  <ResourceDictionary Source="MyDict.xaml">
</ResourceDictionary></ResourceDictionary.MergedDictionary>
<Label Margin="{StaticResource Margin}" />
```

15.6.2. Statische Werte

Mit ***`x:static`*** kann auf C#-Konstanten zugegriffen werden. ***Keine Ressourcen!***

```
public static class MyRes {
  public static SolidColorBrush B3 = new (Color.FromArgb("FFFF")); }
```

```
<ContentPage xmlns:local="clr-namespace:Resources.Examples">
  <Label Background="{x:Static local:MyRes.B3}"></ContentPage>
```

15.7. STYLES

Ressourcen können Werte ***zentral*** verwalten, sie müssen aber immer noch bei ***jedem*** Element ***referenziert*** werden. Styles könnnen Properties ***gruppieren*** und auf Elemente anwenden. **Zwei Arten der Zuweisung:**
1) Explizit: `Style="{StaticResource MyStyle}"` auf gewünschtem Element.
2) Implizit: `TargetType="Button"` gibt an, für welche Element-Typen der Style gilt. Ohne `x:Key` wirkt der Style für alle Elemente dieses Typs. ***Inline-Attribute*** können Styles ***überschreiben***. Styles können mit ***`BasedOn=""` vererbt*** werden.

15.7.1. MauiCSS

Auch mit ***CSS*** können Elemente gestylt werden. Einbinden über `<StyleSheet>` in `<Application.Resources>`, Zuweisung mit `StyleClass=""`. Die Build-Action der CSS-Datei muss auf ***`MauiCSS`*** gesetzt werden. Kann nicht alles, was XAML kann und auch nicht alles, was reguläres CSS kann *(keine Media-Queries oder px-Anweisungen)*. Kann auch direkt in XAML geschrieben werden:

```
<StyleSheet><![CDATA[^contentpage { color: red; }]]></StyleSheet>
```

15.8. THEMING

Mit der ***`AppThemeBinding` Markup-Extension*** kann nach Light- und Darkmode unterschieden werden. Auch kann ein Default-Wert *(OS gibt kein Theme vor)* gesetzt werden *(Erster Wert ist Default, wenn nicht gesetzt)*.

```
<Label TextColor="{AppThemeBinding Light=Black Dark=White}">Hi!</Label>
```

Mit ***`Application.Current.RequestedTheme`*** kann das aktuelle Theme ausgelesen werden, ***`Application.Current.RequestedThemeChanged`*** ist der Event für den Theme-Wechsel. ***`Value`*** gibt Ressource zurück, die derzeit verwendet wird.
Das ***eingebaute Theming*** ist ***unflexibel***: Kein Laden von Resource Dictionaries, kein OS-unabhängiges Theme ladbar. Muss also selber gebaut werden:

1. ***Mehrere Resource Dictionaries*** mit identischen Keys *(z.B. Light.xaml & Dark.xaml)*
2. Laden des Standard-Themes als ***Merged Dictionary***
3. Zugriff auf alle Ressourcen via ***`DynamicResource`***
4. Laden eines neuen Dictionaries beim Wechsel des Themes

Ergibt saubere Trennung der Ressourcen, bei Wechsel könnnen Controls kurz flackern, nur aktueller VisualTree wird aktualisiert. Es gibt auch eine Markup Extension ***`AppThemeBinding`***, welche das Theme autom. nach dem OS anpasst.

15.9. CUSTOM CONTROLS

Es gibt 4 verschieden tiefe Stufen zur eigenen Erstellung von Controls:

15.9.1. Stufe 1: Eigene Controls definieren

Leere Ableitung von MAUI-View, übernimmt alle Attribute der Basisklasse, eigene Attribute möglich. Verwendung via Klassennamen nach Import des Namespaces im XAML *(`xmlns:cc="clr-namespace:V_13.CustomControls"`)*. Präfix wie gewohnt frei wählbar. Auch als Typ für implizite Styles möglich.

```
public class AlertLabel : Label { /* Empty */ }
```

15.9.2. Stufe 2: Darstellung via Custom Templates verändern

Die ***visuelle Repräsentation*** der View soll ***kontrolliert*** werden um z.B. Labels abgerundete Ecken zu geben. Dafür gibt es ***Control Templates:*** Die View muss ***von `ContentView` ableiten*** und alle Inhalte im XAML in ***`<ContentView.ControlTemplate><ControlTemplate>`*** gepackt werden. Mit ***`ControlTemplates`*** ist die Struktur im XAML ohne Anpassung der gesamten Control-Klasse ***austauschbar***.

Bindable Property

Um eigene Daten auf der View zu definieren, muss ein ***Bindable Property*** verwendet werden, normale .NET Properties funktionieren nicht. Diese können im XAML mit ***`Text="{TemplateBinding Message}"`*** verwendet werden. Wird das Property geändert, werden auch die entsprechenden UI-Elemente angepasst. Können nur auf Control Templates verwendet werden und keine ganzen XAML-Elemente darstellen.

```
<ContentView><ContentView.ControlTemplate><ControlTemplate>
  <Border StrokeShape="RoundRectangle 10,10,10,10">
  <Label Text="{TemplateBinding Message}" /></Border>
</ControlTemplate></ContentView.ControlTemplate></ContentView>
```

```
public static readonly BindableProperty MessageProperty =
  BindableProperty.Create(
    nameof(Message),  // Name of property
    typeof(string),   // Type of property
    typeof(AlertBox), // Type the property should belong to
    string.Empty);    // Initial value
public String Message { get => (string)GetValue(MessageProperty);
  set => SetValue(MessageProperty, value); }
```

```
<ContentPage xmlns:cc="clr-namespace:V_13.CustomControls">
    <cc:AlertBox Message="Box 1" /> <!— Bindable Property used —>
    <cc:AlertBox Message="Box 2" /></ContentPage>
```

15.9.3. Stufe 3: Inhalt mit Content Presenter anzeigen

`ContentPresenter` ist ein ***Platzhalter*** für beliebige XAML-Elemente in Control Templates. Gibt den Inhalt von ***`Content`*** aus. Ermöglicht ***Verschachtelung*** in Custom Controls. Nötig, weil in `ControlTemplate` nicht auf Content zugegriffen werden kann. Bei Vererbung würde ***ganzer*** Content ***überschrieben***.

```
<!— Anstatt <Label> <ContentPresenter /> in Control Template —>
<ContentPage xmlns:cc="clr-namespace:V_13.CustomControls">
  <cc:AlertBox><Image Source=/><Label/></cc:AlertBox></ContentPage>
```

15.9.4. Stufe 4: Anpassung der nativen Views mit «Handlers»

Um native Views *(z.B. Buttons)* anzupassen oder MAUI-Controls auf native Platform-Controls abzubilden, werden ***Handler*** benötigt. Sind mächtiger als `OnPlatform`/`OnIdiom`. **Ablauf:** 1. Ableiten des gewünschten Handlers 2. Überschreiben von `ConnectHandler()`. 3. Mit `#if ANDROID` mehrere Versionen von `ConnectHandler()` erstellen. 4. Registrieren in Builder mit `ConfigureMauiHandlers(h => h.AddHandler<Entry, CustomHandl>())`

16. .NET MAUI ARCHITECTURE

16.1. DATA BINDING IN MAUI

Entkopplung von View *(XAML)* & Model *(C# Klassen)* durch Data Binding. Controller verbleibt im Code Behind – nicht ideal. UI-Elemente können mit dem ***BindingContext*** mit dem Model verbunden werden: `var u = new User(); this.BindingContext = u;` Danach können die Properties ***direkt im XAML verwendet*** werden: `<Label Text="{Binding FirstName}">`. Die Datenquelle kann mit der ***Path-Eigenschaft*** überschrieben werden. Quelle muss `INotifyPropertyChanged` implementieren. ***Mode-Eigenschaft:*** Richtung des Datenflusses. ***Converter-Eigenschaft:*** Datenumwandlung zwischen Quelle und Ziel.

16.1.1. Binding Mode

Standardwert abhängig von Ziel-Eigenschaft. **Bsp.:** `{Binding Mode=OneWay}`
– ***OneTime:*** Einmalige Aktualisierung des Ziels beim Setzen der Quelle
– ***OneWay:*** Ziel wird bei jeder Änderung der Quelle aktualisiert
– ***OneWayToSource:*** Quelle wird bei jeder Änderung des Ziels aktualisiert
– ***TwoWay:*** Quelle & Ziel werden gegenseitig synchronisiert

16.1.2. Value Converter

Hilfssobjekt zur ***Datenumwandlung***. Implementiert `IValueConverter` mit `Convert()` *(Quelle → Ziel)* und `ConvertBack()` *(Ziel → Quelle)*.

```
public class C : IValueConverter {
  public object Convert(object value) { /* ... */ }
  public object ConvertBack(object value) { /* ... */ } }
```

```
<ContentPage xmlns:con="clr-namespace:(...)">
<ContentPage.Resources><con:C x:Key="myCon"/><ContentPage.Resources>
<Entry Text="{Binding Path=Name Converter={StaticResource myCon}}"/>
```

16.1.3. Multi-Binding

Verwendung analog zu Binding `Path`, `Mode`, etc. ***Mehrere Quell-Eigenschaften***. Nur in ***Property Element Syntax***. Ist Zieleigenschaft kein String, muss Converter `IMultiValueConverter` implementieren. Ist erstes Zeichen im Format String «{«, muss es mit {} escapet werden.

```
<Label><Label.Text><MultiBinding StringFormat="{}{0}, {1} ({2} J.)">
  <Binding Path="LastName" /><Binding Path="FirstName" />
  <Binding Path="Age" /></MultiBinding></Label.Text></Label>
```

16.1.4. Binding Context

Enthält ***Standardquelle*** für Bindings. Property von `BindableObject`. Wenn undefiniert: Traversierung des Logical Trees nach oben bis zum ersten Treffer. Beliebige Quell-Objekte möglich. Meist pro Page ein BindingContext. **Datenquellen**: `BindingContext` *(Code Behind)* `{Binding Source=}` *(XAML-Default)*, `{RelativeSource=}` *(Visual Tree-Referenz)*, `{x:Reference}` *(Property von Element, z.B. `x:Name`)*

16.1.5. Compiled Bindings

Bindings werden erst zur ***Runtime*** aufgelöst *(kein IDE-Support)*. `x:DataType` vergibt explizit Typ. **Vorteile:** Compile-Time Fehler, bessere Performance.

16.2. OBSERVER PATTERN IN .NET

Damit Änderungen an Binding Properties aktualisiert werden, muss die Klasse ***`INotifyPropertyChanged` (INPC)*** implementieren.
Quelle: `Observable` mit INPC, **Ziel:** `Observer` mit Event Handler. Es gibt in MAUI keine Basisklasse, die INPC implementiert, um Framework-unabhängig zu bleiben *(INPC funktioniert in allen XAML-Frameworks)*

```
public class UserViewModel : INotifyPropertyChanged {
  public event PropertyChangedEventHandler PropertyChanged;
  protected virtual void OnPropertyChanged(string name) {
    var eventArgs = new PropertyChangedEventArgs(name);
    PropertyChanged?.Invoke(this, eventArgs);
  }
  private string _firstName = "Nina";
  public string FirstName {
    get => _firstName;
    set { if (_firstName == value) { return; }
      _firstName = value;
      OnPropertyChanged(nameof(FirstName)); } } }
```

16.3. OBSERVABLE COLLECTIONS

Collections können mit Collection Views dargestellt werden *(z.B: CarouselView, Eigenschaften: `ItemsSource`, `ItemTemplate`, `ItemsLayout`)*. Für Data Binding muss ***`INotifyCollectionChanged` (INCC)*** implementiert werden. ***`ObservableCollection<T>`*** implementiert INCC *(Add/Remove Element)* und INPC *(Änderung an Element in der Collection)* und sollte bevorzugt verwendet werden. Die enthaltenen Elemente müssen aber ihre Änderungen selbstständig via INPC kommunizieren!

16.3.1. Custom Item Templates

Für eigene Layouts wird ein ***`ContentView`*** erstellt. Bietet vollständige Kontrolle über Layout und Inhalt. ***Vorteile:*** Wiederverwendbarkeit, eigene Code-Behind Logik, bessere Organisation. ***`x:DataType`*** aktiviert Compiled Bindings. `ItemTemplate` ist eine Property, die ein `DataTemplate` enthält.

16.4. COMMANDS

Mit Data Binding können ***Properties verknüpft*** werden, aber keine Methoden. **Lösung:** Methoden in Objekte packen und `ICommand` implementieren. Parameter werden in View gebunden.

- `void Execute(object param)`: Code der Aktion *(z.B. Alter von Nutzer verringern)*. Sollte meist `CanExecuteChanged` invoken
- `bool CanExecute(object param)`: Kann Aktion ausgeführt werden? *(Alter $\geq$ 0?)*
- Event `CanExecuteChanged`: Auslösen wenn `CanExecute`-Bedingung ändert *(Alter ändert sich)*

```
<Button Text="Decrease Age" Command="{Binding DecreaseAge}"
  CommandParameter="{Binding SelectedUser}" />
```

16.5. MVVM IN MAUI

Die Applikation sollte vollständig über UI-Abstraktionen bedienbar sein *(für Tests)*. Die View soll dumm sein. ***Model:*** Businesslogik & Datenstrukturen *(C# Klassen, Interfaces)* ***View:*** Darstellung *(XAML & CodeBehind, MAUI Control Lib)*
ViewModel: Präsentationslogik, Model für View adaptieren *(C#-Klasse mit INPC)*
V $\Leftrightarrow$ VM: Lose Kopplung *(Data Binding)*, ***VM $\Leftrightarrow$ M:*** Starke Kopplung *(Methoden/Events)*

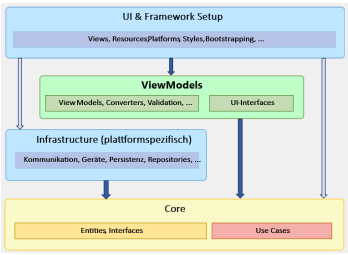
Wohin gehört welcher Code: ***Model:*** Logik ist Teil der Domäne oder wird mehrfach verwendet. ***ViewModel:*** Logik ist unabhängig vom verwendeten UI-Framework. ***View:*** Der Rest.

16.5.1. Bootstrapping und Dependency Injection (DI)

UI-Elemente werden mit dem .NET DI Container instanziert. Er löst Verkettungen automatisch auf *(TodoListPage $\rightarrow$ IManageTodos $\rightarrow$ ITodoRepository)*.

16.6. GESAMTARCHITEKTUR

Es werden ***4 Unterprojekte*** erstellt. Referenzen von Core auf Infrastruktur sind nicht kompilierbar $\rightarrow$ versehentliche ***Architekturverletzungen unmöglich***. ***Unittests*** sollten hauptsächlich Usecases & View Models testen.



17. MAUI ADVANCED

17.1. BACKGROUND THREADS

MAUI baut auf plattformspezifischen Mechanismen auf und hat einen ***Main Thread*** *(GUI-Aktualisierung)* und eventuelle ***Background Threads*** *(für langlaufende Aktionen)*. **Stolperfallen:** Langlaufende Operationen auf Main Thread ***blockieren*** GUI, GUI-Updates aus Background Threads führen zu ***Exceptions***. **Lösungen:**

- ***Eigene Threads:*** `Task.Run()` *(Thread-Pool)*, Parallel LINQ, `new Thread()` *(manuelle Thread-Erstellung – vermeiden!)*, `BackgroundWorker` *(veraltet!)*
- ***Ans OS delegieren:*** `async`/`await` *(Thread wird freigegeben, OS managed IO und benachrichtigt bei Completion)*

Über die ***`MainThread`***-Klasse lässt sich Code an den Main Thread zurückdelegieren: ***`BeginInvokeOnMainThread()`*** *(fire-and-forget, für einfache UI-Updates ohne Abhängigkeiten, keine Garantie wann Update passiert)*, ***`InvokeOnMainThreadAsync()`*** *(Abwarten mit `await`, wenn Reihenfolge wichtig, z.B. UI-Update, dann DB.)*

17.1.1. Best Thread Practices

Bei ***async-Aufrufen*** aus Main Thread ***ohne UI-Manipulation*** ist kein Aufruf von `MainThread`-Funktionen nötig. ***Background Threads*** für intensive Berechnungen oder IO nutzen. ***Keine UI Updates*** aus Background Threads $\rightarrow$ dafür `MainThread`-Funktionen verwenden. Vermeide Blockierungen durch ***synchronen*** Code *(`.Wait()`, `.Result`)*

17.2. DATEN SPEICHERN

- ***`FileSystem.Current.AppDataDirectory`*** *(User Data)*,
- ***`FileSystem.Current.CacheDirectory`*** *(Temp Data)*,
- ***`OpenAppPackageFileAsync(filename)`*** *(Read-only Stream für MAUI-Assets)*

Für kleinere Datenmengen: `Preferences.Set()`/`.Get()` *(Schnell, für nicht sensitive Daten)*, `SecureStorage.SetAsync()`/`.GetAsync()` *(Verschlüsselt, für Tokens/Passwörter)*. Bei Secure Storage ist ***Error Handling wichtig!*** Device Security Settings können sich ändern, dann Key entfernen und neu authentifizieren.

17.2.1. State Restauration & App Lifecycle

Service Pattern: Zentraler `AppStateService` kapselt Storage APIs mit type-safe Constants. Kombiniert Preferences & SecureStorage. **App Lifecycle Integration:** `OnStart()` *(State restore)*, `OnSleep()` *(State speichern vor Background)*, `OnResume()` *(State validieren)*

17.3. NAVIGATION & ROUTING

Shell Navigation: Bietet Navigation Stack Management und Deep Linking. Zwischen verschiedenen UI-Elementen kann mit URIs navigiert werden *(Navigation ohne feste Hierarchie)*. ***`IQueryAttributable`*** bietet eine saubere Trennung, ist trim-safe und mit modernen Deployment-Szenarien kompatibel. Sollte man bei Native AOT Deployment verwenden. **Navigation Lifecycle:** ***`OnNavigating`*** ist vor Navigation, ***`OnNavigated`*** nach Navigation.

17.4. PLATFORM INTEGRATION

MAUI enthält für viele Fälle bereits ***einheitliche APIs***, welche auf Plattform APIs gemapped werden. Genügt das nicht, können pro Plattform eigene Services erstellt werden, Wissen über APIs der Zielplattform nötig.
Varianten: ***Conditional Compilation*** *(Alles in einer Datei, `#if iOS`. Schlechte Lesbarkeit, unübersichtlich. Nur für einfache Features)*, ***Cross-Platform API*** *(partial class, jede Plattform in eigener Datei. Empfohlene Variante)*, ***Plattform-spezifische Registrierung*** *(Plattform registriert eigene Services beim Start)*

17.5. LOSLÖSUNG VOM FRAMEWORK

Input-Validierung ist oft Businesslastig, muss aber häufig in UI verwendet werden. Mit dem Dependency Inversion Principle lassen sich MAUI-Funktionen über Abstraktionen im ViewModel ansprechen: Interfaces definieren und per DI im Constructor injizieren. *GoF Pattern: Adapter*

17.6. ERROR HANDLING

Errorhandling ist stark Plattformabhängig, Fehler müssen je nachdem pro Plattform anders gehandhabt werden. Implementierung kann via Delegates oder Interfaces stattfinden. Mit `INotifyPropertyChanged` & `INotifyDataErrorInfo` können Fehlermeldungen implementiert werden.