WORKSHOP

# REAKTIVE PROGRAMMIERUNG

# AGENDA

- Wer sind wir? Was sind eure Erwartungen?
- Code Review / Übungs-App
- Warum reaktive Programmierung? Herausforderungen sowie Vor- und Nachteile.
- Architektur, Konzepte und Best Practices Rx.NET
- Grundkonzepte der reaktiven Programmierung
  - Vergleich mit imperativer Programmierung
  - Einführung in Rx.NET und ReactiveUI
  - Praxisbeispiele, Übungen und Live-Coding
- Reactive UI
  - Architektur, Konzepte und Best Practices Reactive UI
  - Testing und Performance
- Dynamic Data
  - Architektur, Konzepte und Best Practices Dynamic Data
  - Testing und Performance
- Übungs-App Präsentation
- Fazit und Diskussion

# WARUM REAKTIVE PROGRAMMIERUNG?

- Herausforderungen in der modernen Softwareentwicklung:
  - Skalierbarkeit
  - Performance
  - Asynchrone Verarbeitung
- Anwendungsfälle:
  - Echtzeitanwendungen
  - Datenintensive Prozesse
  - Responsive User Interfaces

# STREAMS

- **Streams**: Datenflüsse und Ereignisse
- **Observer Pattern**: Beobachter-Reaktionsmuster
- **Push vs. Pull**: Unterschied zu herkömmlichen Methoden
- **Reaktive Operatoren**: Mapping, Filtering, Combining

# WAS SIND DATENSTRÖME? (STREAMS)

- **IObservable<T>** als Kernkonzept von Rx.NET (https://introtorx.com/)
- Daten fließen kontinuierlich, nicht als einzelne Aufrufe
- Wie werden Observables erstellt?
  - Observable.Create<T>()
  - Observable.Range<T>()
  - Observable.Return<T>()
  - Observable.Interval()
  - Observable.FromEventPattern()
- Beispiel Code / Live Demo

# OBSERVER PATTERN

- **Observable**
  - Subscribe - Anhängen an den Stream
- **Observer**
  - OnNext - Wert Signal
  - OnError - Fehler Signal
  - OnCompleted - Fehler Signal

# COLD OBSERVABLE

- Ein Cold Observable startet jedes Mal von vorne, wenn sich ein Observer abonniert.
- Beispielhafte Analogie:
  - Stell dir einen Film vor, den du dir auf Netflix ansiehst – jeder Zuschauer sieht den Film von Anfang an, unabhängig davon, wann er einschaltet.
- Datenquelle:
  - Die Datenquelle wird pro Subscription neu gestartet.
- Beispiele:
  - *Observable.Return()*
  - *Observable.Create()*
  - *Observable.Interval()* (wenn neu erstellt)
- Web-Requests, Datenbankabfragen etc.

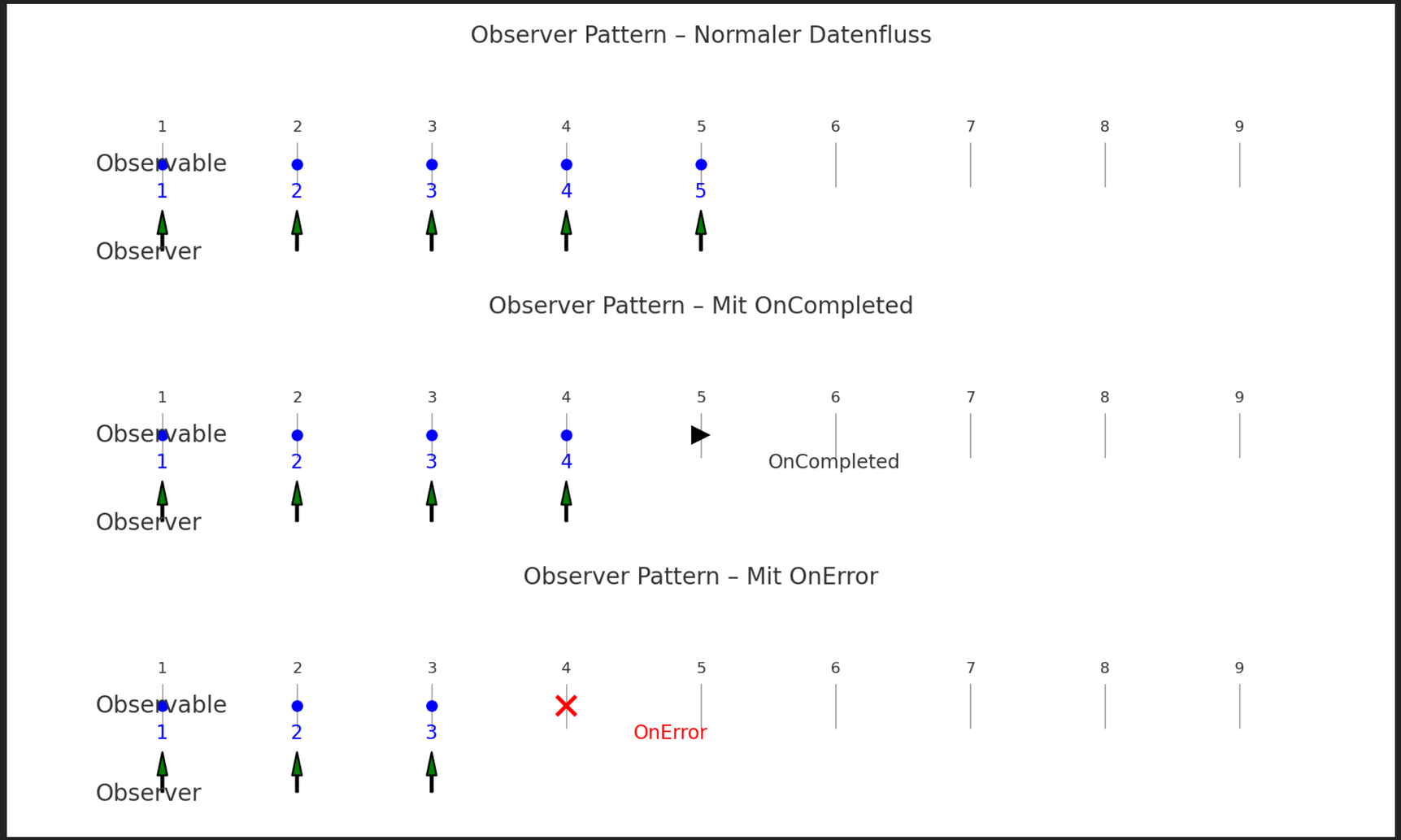
# HOT OBSERVABLES

- Ein Hot Observable produziert Daten unabhängig davon, ob es Abonnenten gibt oder nicht. Neue Observer bekommen nur zukünftige Werte.
- Beispielhafte Analogie:
  - Eine Live-Sendung im Fernsehen – du siehst nur den Teil, der läuft, ab dem Zeitpunkt, an dem du einschaltest.
- Datenquelle:
  - Die Datenquelle läuft unabhängig von Abonnenten.
- Beispiele:
  - Subject<T> (z. B. PublishSubject, BehaviorSubject)
  - Konvertierte Cold Observables mit .Publish().RefCount()
  - UI-Events, Timer, Sensoren etc.

# OBSERVER PATTERN

- **Beobachter-Reaktionsmuster erklärt:**
  - Ein Observable sendet Daten/Ereignisse
  - Ein Observer reagiert darauf (z.B. mit OnNext, OnError, OnCompleted)
- **Beziehung zwischen Publisher und Subscriber**
  - Lose gekoppelte Architektur
  - Reagieren statt Abfragen (Push-basiert)
- **Vorteile:**
  - Entkopplung von Sender und Empfänger
  - Einfache Zusammensetzung komplexer Abläufe

# OBSERVER PATTERN IN RX



# RESSOURCEN BEREINIGEN (DISPOSABLES)

- Warum Disposables?
  - **Ressourcenverwaltung** bei Subscriptions (z. B. Timer, Event-Handler, Netzwerkstreams)
  - **Vermeidung** von Speicherlecks durch nicht abgemeldete Observer
  - Kontrolle über **Lebensdauer** von Datenströmen
- Wie funktionieren Disposables in Rx.NET?
  - **IDisposable**-Instanz wird bei Subscribe() zurückgegeben
  - Dispose() **beendet** die Subscription und gibt Ressourcen frei
- **Tipp**:
  - In komplexeren Szenarien CompositeDisposable verwenden, um mehrere Subscriptions zentral zu verwalten

# PUSH VS. PULL – UNTERSCHIEDLICHE DENKWEISEN

- **Pull-Modell (Imperativ):**
  - Der Konsument fragt aktiv nach Daten (z.B. mittels Schleifen oder Methodenaufrufen)
  - Kontrolle liegt beim Konsumenten
  - Beispiel: var item = list[0];
- **Push-Modell (Reaktiv):**
  - Der Produzent sendet Daten/Ereignisse an den Konsumenten
  - Kontrolle liegt beim Produzenten

# PUSH VS. PULL – UNTERSCHIEDLICHE DENKWEISEN

- **Fazit:**
  - **Push** eignet sich ideal für Ereignisse, Streams und asynchrone Datenflüsse (potentiell unendliche Datenmenge)
  - **Pull** eignet sich ideal für synchrone Datenflüsse, Datenanzeigen und Abfragen

| Merkmal | Pull (Imperativ) | Push (Reaktiv) |
|---|---|---|
| Datenfluss | Konsument gesteuert | Produzent gesteuert |
| Fehlerverhalten | try-catch | OnError-Kanal |
| Synchronität | meist synchron | meist asynchron |

# REAKTIVE OPERATOREN

- **Was sind reaktive Operatoren?**
  - Funktionen, die auf IObservable<T>-Sequenzen angewendet werden
  - Transformation, Filterung, Kombination, Aggregation von Datenströmen
- **Anwendungsszenarien:**
  - Reaktive Filterung von UI-Events
  - Verarbeitung von Benutzereingaben (z. B. Autosuggest)
  - Kombinieren mehrerer Datenquellen in Echtzeit
- Tipp:
  - Marble Diagramme https://rxmarbles.com/

# REACTIVE OPERATOREN

- **Wichtige Operatoren:**
  - Select – Transformation von Werten (analog zu LINQ Select)
  - Where – Filtern von Werten
  - Merge – Zusammenführen mehrerer Streams
  - CombineLatest – Kombinieren aktueller Werte mehrerer Observables
  - Throttle / Debounce – Ereignisbegrenzung bei hoher Frequenz

# VERGLEICH MIT IMPERATIVER PROGRAMMIERUNG

- Synchrone vs. Asynchrone Verarbeitung
  - enumerable.ToObservable()
  - Observable.ToEnumerable()
- LINQ arbeitet synchron mit vorhandenen Datenquellen
- Rx.NET arbeitet non-blocking mit Datenströmen
- Beide nutzen ähnliche Operatoren, aber mit unterschiedlichem Paradigma
- Beispiel mit Code-Snippets:
  - ToObservable / ToEnumerable
  - Traditionelle Callbacks vs. Observable (Integration vorhandener APIs in reaktive Abläufe)

# ASYNCHRONE PROGRAMMIERUNG

- Ziele:
  - Nicht-blockierende und performante Anwendungen
  - Bessere Nutzererfahrung durch reaktive UI (Single vs. Multi-Threaded Applications)
- Grundbegriffe:
  - async / await
  - Task<T> und ValueTask<T>
  - **UI Context-Switch!** ConfigureAwait(false)
- Typische Anwendungsfälle:
  - Webzugriffe (HTTP / WCF)
  - Datenbankoperationen
  - Datei- und Netzwerkzugriffe

# INTEGRATION ASYNCHRONER METHODEN IN RX.NET

- Problem:
  - async-basierte Methoden lassen sich nicht direkt mit IObservable<T> kombinieren
- Lösungen:
  - *Observable.FromAsync(() => LoadDataAsync())*
  - *SelectMany* für das Verketten mehrerer async-Aufrufe
- Vorteil:
  - Bestehende async-APIs können in reaktive Workflows eingebunden werden
  - Einhaltung des asynchronen Programmiermodells
- Hinweise:
  - Observable.FromAsync erzeugt einen reaktiven Stream aus einer async Methode
  - Nützlich für Web-APIs, Microservices, etc.
- Beispiel Code / Demos

# WARUM „USING“?

- HttpClient implementiert IDisposable - Ressourcen, wie Netzwerkverbindungen, werden verwaltet und explizit freigegeben
- Mit „using var“ wird sichergestellt, dass client.Dispose() automatisch aufgerufen wird, sobald die Methode (LoadAsync) beendet ist
- Dies verhindert Ressourcenlecks, insbesondere bei kurzfristiger Verwendung (z. B. in einem einzelnen Request)
- Vorteile von “using var“ (alt using(var…) oder async using(var …))
  - Kürzere und übersichtlichere Syntax (seit C# 8)
  - Keine explizite Dispose()-Anweisung notwendig
  - Ressourcen wie Sockets oder Dateihandles werden sicher freigegeben
- Hinweis zu HttpClient
  - Für wiederholte/parallele HTTP-Anfragen sollte ein HttpClient **nicht** pro Anfrage instanziert werden, da das zu Socket-Erschöpfung führen kann
  - In solchen Fällen ist eine Singleton-Instanz (z. B. über HttpClientFactory oder Dependency Injection) besser geeignet

# ASYNC-AUFRUFE MIT ABBRUCH (CANCELLATIONTOKEN)

- Warum Cancellation?
  - Ermöglicht das Abbrechen von Anfragen bei Zeitüberschreitung, Benutzeraktion oder Navigation
  - Vermeidet unnötige Serverlast oder UI-Updates nach Abbruch
- Integration in HttpClient:
  - Die Methode *GetStringAsync(string, CancellationToken)* unterstützt Abbruchtoken
- Tipp:
  - In UI-Anwendungen mit CompositeDisposable kombinieren
  - Auch *Timeout()* Operator in Rx.NET kann nützlich sein

# HTTP MIT RX.NET – SELECTMANY, TIMEOUT & CANCELLATION

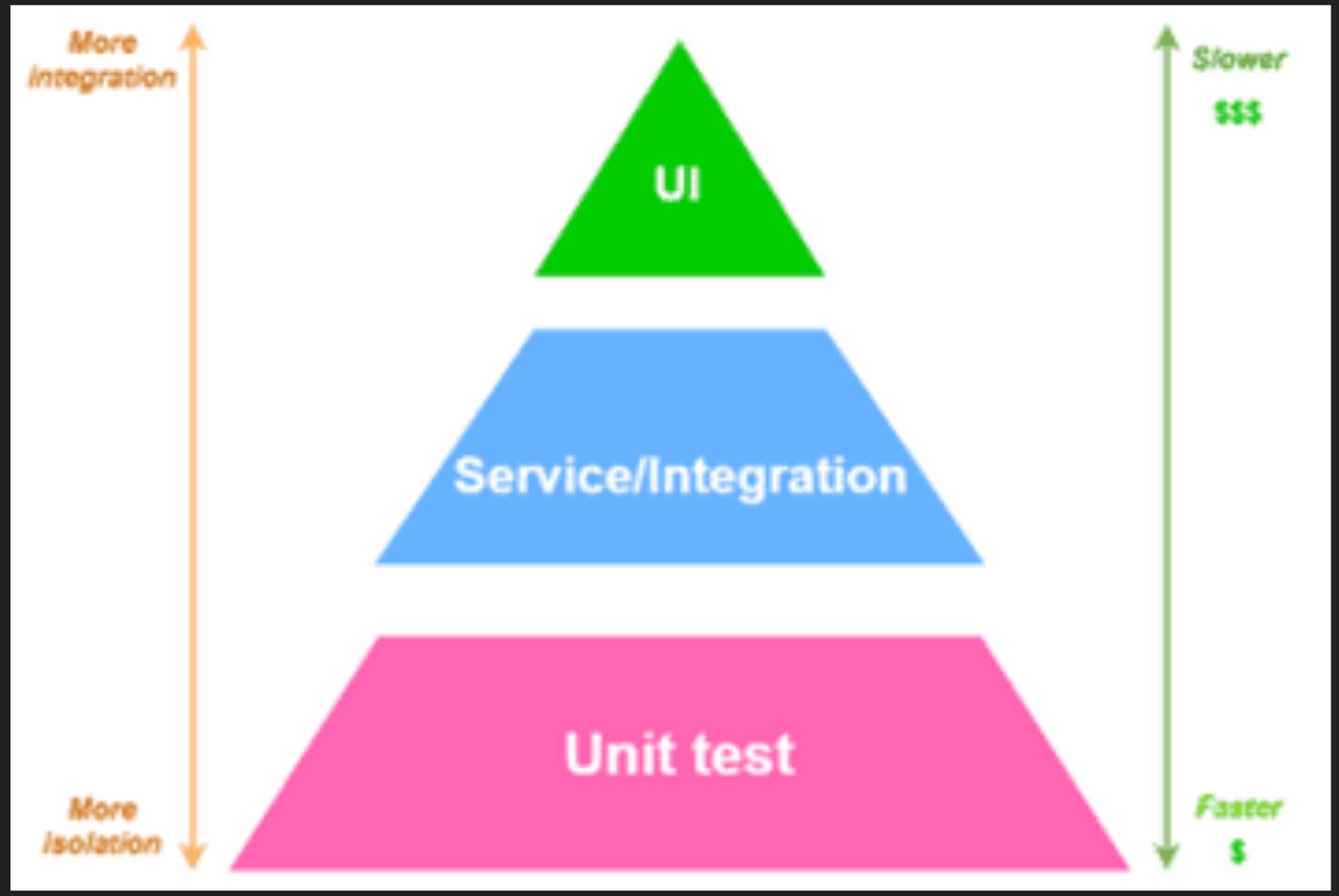
- Ziel:
  - Mehrere HTTP-Requests aus einem Event-Stream starten, mit Timeout und Abbruchmöglichkeit
- Erklärung:
  - *SelectMany* startet pro URL eine neue Anfrage (flach gemappt)
  - *Timeout* bricht Anfragen ab, die zu lange dauern
  - *CancellationToken* ermöglicht manuelles Abbrechen
  - *Catch* verhindert, dass ein Fehler den Stream vollständig beendet
- Szenarien:
  - UI-Interaktion (z. B. Sucheingaben, Scrollen)
  - Serienanfragen an Microservices
- Beispiel Code / Demos

# TESTING

- Grundlagen / Struktur
  - xUnit
- Dependency Injection / Mocking
  - Moq
- Testing in Rx.NET
  - Async / Timing
  - Schedulers / TestSchedulers
  - Marble-Tests https://github.com/alexvictoor/MarbleTest.Net

# TESTING GRUNDLAGEN MIT XUNIT

- Test-Pyramide
- AAA- Struktur
  (Arrange, Act, Assert)
- xUnit
  - [Fact] vs. [Theory]
  - [Fact] → ein einzelner Test
  - [Theory] → parametrisierte Tests mit [InlineData(...)]




(Quelle: anymindgroup.com/news/tech-blog/15053)

# XUNIT

- Test Setup = Contructor
- Test TeadDown = implement IDisposable
- [Fact] = simpler Test
- [Theory] = Data Driven Test verwendet [InlineData(func params)]

# WAS IST DEPENDENCY INJECTION (DI)?

- Abhängigkeiten (z. B. Services, Repositories, Logger) werden nicht selbst innerhalb einer Klasse erzeugt, sondern von außen bereitgestellt
- Ziel:
  - Lockere Kopplung (Loose Coupling)
  - Bessere Testbarkeit
  - Austauschbarkeit von Implementierungen

# WARUM DEPENDENCY INJECTION?

- Ohne DI
  - nicht (unit) testbar
  - nicht erweiterbar
  - starr
- Mit DI
  - Klare Trennung von Logik und Infrastruktur
  - Testbarkeit durch einfache Mocks oder Stubs
  - Konfiguration zentral steuerbar

# MOCK MIT XUNIT UND MOQ

- *dotnet add package Moq*

| Methode | Bedeutung |
| --- | --- |
| `new Mock<IInterface>()` | Erzeugt ein Mock-Objekt |
| `mock.Object` | Gibt die IInterface-Instanz zurück |
| `mock.Verify(...)` | Prüft, ob eine Methode mit bestimmten Parametern aufgerufen wurde |
| `mock.Setup(...)` | (Optional) Verhalten vorgeben / Rückgabewert festlegen |

# RX.NET IOBSERVABLE<T> + MOQ

- *Ziel:*
  - *Komponente, die einen Datenstrom (IObservable<string>) von einem Service erhält*
  - *Im Test simulieren wir diesen Service mit Moq*
  - *Was kannst Du testen?*
    - *Erfolgreiche Verarbeitung*
    - *Timeout-Verhalten*
    - *Fehler-Handling*
    - *Stream-Abbruch*
    - *Logging / Retry (mit Catch / RetryWhen)*

# WOFÜR BRAUCHE ICH SCHEDULER IN RX.NET?

| Anwendungsfall | Beispiel |
| --- | --- |
| Testbarkeit | `TestScheduler` – virtuelle Zeit |
| Multithreading | `NewThreadScheduler`, `TaskPoolScheduler` |
| UI-Anwendungen | `DispatcherScheduler`, `SynchronizationContextScheduler` |
| Steuerung von Zeit | `Interval`, `Delay`, `Timer`, `Timeout`, etc. |

# RX.NET SCHEDULERS

- Scheduler in Rx.NET sind ein zentrales Konzept
- Testbarkeit, Parallelität und Kontrolle über die Ausführung
- Ein Scheduler ist eine Abstraktion über „Wann und wo" ein Operator oder Observer-Code ausgeführt wird. Er kontrolliert:
  - Zeitpunkt der Ausführung (Now, Schedule(...))
  - Thread/Context der Ausführung (ThreadPool, UI, TestScheduler, etc.)

# WICHTIGE SCHEDULER IN RX.NET

| Scheduler | Beschreibung |
| --- | --- |
| `ImmediateScheduler` | Führt Code sofort synchron aus |
| `CurrentThreadScheduler` | FIFO auf aktuellem Thread |
| `NewThreadScheduler` | Führt auf neuem Thread aus |
| `ThreadPoolScheduler` | Nutzt ThreadPool |
| `TaskPoolScheduler` | Nutzt `Task.Run()` |
| `EventLoopScheduler` | Serialisiert Events auf dediziertem Thread |
| `DispatcherScheduler` (WPF/WinUI) | Für UI-Threads (WPF, Avalonia, WinForms) |
| `TestScheduler` | Simuliert Zeit für Unit Tests (sehr präzise!) |

# WAS MACHT DER TESTSCHEDULER BESONDERS?

- Nutzt virtuelle Zeit (du kontrollierst die Zeit manuell)
- Ermöglicht deterministische Tests von Streams, Timern und Delays
- *Zeit in TestScheduler*
  - *Zeit wird in Ticks gemessen (1 Tick = 100 Nanosekunden)*
  - *1 Sekunde = TimeSpan.FromSeconds(1).Ticks = 10_000_000*
- TestScheduler unterstützt:
  - ***AdvanceTo()*** *- setzt virtuelle Zeit auf die angegebene Anzahl von Ticks. Dadurch werden alle bis zu diesem absoluten Zeitpunkt geplanten Aktionen ausgeführt.*
  - ***AdvanceBy()*** *- verwendet um die Uhr um eine bestimmte Zeit vorstellen. Im Gegensatz zu AdvanceTo ist das Argument hier relativ zur aktuellen virtuellen Zeit. Auch hier erfolgt die Messung in Ticks.*
  - ***Start()*** *- führt alles aus, was geplant wurde, und verlängert die virtuelle Zeit nach Bedarf für Arbeitselemente, die für eine bestimmte Zeit in die Warteschlange gestellt wurden.*

# WANN SOLLTE ICH OBSERVE-ON ODER SUBSCRIBE-ON VERWENDEN?

| Operator | Beschreibung |
| --- | --- |
| `SubscribeOn(scheduler)` | Beeinflusst **wo** die Subscription beginnt (Upstream) |
| `ObserveOn(scheduler)` | Beeinflusst **wo** die Observer (Callbacks) ausgeführt werden (Downstream) |

```
observable
    .SubscribeOn(TaskPoolScheduler.Default)
    .ObserveOn(DispatcherScheduler.Current)
    .Subscribe(x => label.Text = x.ToString());
```

# WANN SOLLTE ICH OBSERVE-ON ODER SUBSCRIBE-ON VERWENDEN?

- SubscribeOn
  - Bestimmt den Thread, auf dem die Subscription (Datenquelle) beginnt
  - Also: Wo startet der Datenstrom?
  - Betrifft z. B. Netzwerkaufrufe, Datenbankzugriffe usw.
  - Hat nur beim ersten Aufruf eine Wirkung – spätere SubscribeOns werden ignoriert
- ObserveOn
  - Bestimmt den Thread, auf dem die weiteren Operatoren nach diesem Punkt ausgeführt werden
  - Also: Wo werden die empfangenen Daten weiterverarbeitet oder angezeigt?
  - Kann mehrmals in der Kette verwendet werden

| Operator | Wirkung auf ... | Kann mehrfach verwendet werden? | Typischer Einsatz |
|---|---|---|---|
| `SubscribeOn` | **Wo** beginnt die Datenquelle? | ❌ Nur einmal wirksam | Daten holen im Hintergrund |
| `ObserveOn` | **Wo** läuft die Weiterverarbeitung? | ✅ Ja | UI-Update oder verschiedene Phasen trennen |

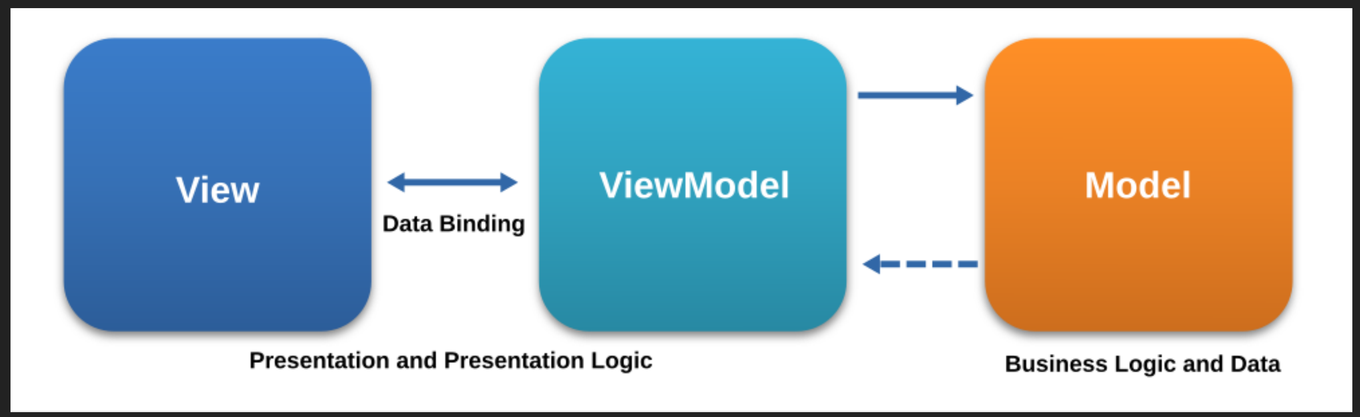
# MARBLE-SCHEDULER

- Testen mithilfe von Marble-Diagrammen im ASCII-Format
- https://github.com/alexvictoor/MarbleTest.Net

# REACTIVE UI

- Open-Source MVVM Framework für .NET
- Ideal für UI-Logik mit dynamischen Zuständen
- Reaktive Programmierung mit Observables (Rx.NET)
  - Testbarkeit durch Trennung von Logik und UI
  - Weniger Boilerplate-Code durch Bindings
  - Reaktive Datenflüsse vereinfachen komplexe UI-Zustände

# ARCHITEKTUR – MODEL-VIEW-VIEWMODEL (MVVM)




- Macht größere Apps wart- und testbarer sowie skalierbar
- Saubere Trennung von Zuständigkeiten
  - Model: Daten/Datenmodelle und Geschäftslogik
  - View:  Anzeige - UI (XAML, etc.)
  - ViewModel: Logik und Vermittler zwischen Model & View
    - Datenbindung (Binding)
    - Commands (ReactiveCommand)

# ARCHITEKTUR

- ReactiveObject - ermöglicht PropertyChange-Notifications
- ReactiveCommand - deklarative Commands mit Ausführungsstatus und Fehlerbehandlung
- WhenAnyValue, ObservableAsPropertyHelper - automatische Bindung & reaktive Props

# DATENFLUSS

- View schickt User-Input an ViewModel
- ViewModel verarbeitet & ändert Model
- Model sendet neue Daten zurück
- View wird automatisch aktualisiert (durch Binding)

# ZUSTANDSVERWALTUNG

- Du kannst Änderungen im Datenfluss beobachten (Observables)
- Ideal für komplexe UI-Logik mit dynamischen Zuständen (z. B. Live-Suche, Ladeindikatoren, Validierungen)
- Statt „wenn dies passiert, dann tu das“ → du reagierst auf Zustandsänderungen
- ObservableAsPropertyHelper verwenden, um State (.*ToProperty(…)*) aus Observables zu erzeugen

# VORTEILE

- Testbarkeit durch Trennung von Logik und UI
  - Durch die Trennung von View und Logik (MVVM) kannst du das ViewModel einfach unit-testen, ganz ohne UI
  - Commands und Properties lassen sich einfach simulieren und prüfen
- Weniger Boilerplate-Code durch Bindings
  - Keine manuelle Event-Registrierung nötig (PropertyChanged, Click-Handler, usw.)
  - ReactiveCommand & Observables ersetzen viele klassische C#-Eventmuster
- Reaktive Datenflüsse vereinfachen komplexe UI-Zustände
- Änderungen im ViewModel wirken sich automatisch auf die UI aus – und umgekehrt.

# IN PRACTICE!

- ObservableAsPropertyHelper verwenden, um State (.ToProperty()) aus Observables zu erzeugen
- ReactiveCommand verwenden, statt normalen Commands oder EventHandler
- Fehlerbehandlung integrieren (*AddItemCommand.ThrowExceptions.Subscribe(ex => Logger.Log(ex))*)
- Trenne ViewModel von UI-Logik (keine UI-Komponenten in ViewModel)
- Vermeide Side-Effects in Subscriptions - InvokeCommand / .Do() verwenden
- Vermeide die „unsaubere“ Verwendung von Rx - „Memory-Leaks“ drohen
- Vermeide komplexe Bindungen im View (XAML, etc.) - Logik gehört ins ViewModel
- *Observable* ohne Subscribe() tut nichts!

# TESTING

- Verwende Rx im Testing (vm.command.Execute.Subscribe(), vm.command.CanExecute().FirstAsync().Wait())
- Immer View-Model per Unit-Tests - Trenne ViewModel von UI-Logik!
- Auch Fehlerbehandlung in die Tests integrieren (*Record.Exception(() => {})*)
- Vermeide Side-Effects, um „gut“ testen zu können
- Verwende TestScheduler für komplexe „timing“ Tests

# NACHTEILE

- Rx.NET (Reactive Extensions) ist komplex – besonders für Einsteiger
- Reaktive Denkweise unterscheidet sich stark von klassischem imperativen C#-Code
- Man muss verstehen, wie Observables, Subscriptions und Schedulers funktionieren
- Fehler können schnell passieren, wenn man z. B. vergisst, Subscriptions zu disposen → Memory Leaks
- MVVM + Rx bringt zusätzliche Komplexität, die sich nicht immer lohnt
- Fehler in Observable-Chains sind manchmal nicht intuitiv nachvollziehbar
- Stack Traces bei Rx.NET sind manchmal nicht hilfreich ohne spezielle Tools (z. B. Rx Spy, DynamicData Debugging)
- Dispose-Handling ist Pflicht
  - Du musst aktiv auf Speicher- und Ressourcenmanagement achten:
    - DisposeWith(...)
    - CompositeDisposable
  - Sonst → Speicherlecks durch nicht abgemeldete Observables.

# FORTGESCHRITTENE FEATURES

- *WhenAnyValue, ObservableAsPropertyHelper, Interaction<TInput, TOutput>* etc.
- Für komplexe Szenarien wie:
  - Bedingte Commands
  - Validierung in Echtzeit
  - Zustandsabhängige Anzeige/Verhalten

# DYNAMIC DATA

- DynamicData ist eine Reactive Collection Library für .NET
- Sie wurde entwickelt, um reagierende (Observable) Datenlisten einfach, effizient und sauber zu verwalten
- Arbeitet perfekt mit ReactiveUI zusammen
  - Vermeidet UI-Refreshes bei jedem Mini-Update → sehr performant
  - Macht komplexe Listenlogik extrem elegant und deklarativ
- DynamicData ist für Listen, was ReactiveUI für einzelne Properties ist

# VORTEILE

- Normale ObservableCollection<T> reicht oft nicht aus, wenn du:
  - mit großen Datenmengen arbeitest
  - komplexe Filter, Sortierung, Suche, Gruppierung oder Live-Updates brauchst
  - die UI automatisch mit Änderungen synchron halten willst

# NACHTEILE

- Hohe Lernkurve und Overkill für die Anzeige von „einfachen“ Listen
- Komplexe Fehlersuche, da Events asynchron & verzweigt sein können
- Gefahr von Memory Leaks, wenn Subscriptions nicht korrekt entsorgt werden
- Begriffe wie *ChangeSet, AutoRefresh, Connect(), Transform()* etc. sind anfangs ungewohnt
- Performance kann leiden bei falscher Nutzung
  - Wenn du zu viele .AutoRefresh()-Abonnements oder .Transform()-Operationen stapelst, kann das schnell ineffizient werden
  - Besonders bei sehr großen Listen musst du bewusst mit Slicing, Paging und Caching arbeiten
- Fehlerquellen bei AutoRefresh & Binding
  - AutoRefresh() reagiert nur auf PropertyChanged – funktioniert nicht bei normalen Feldern oder Events
  - Bei falscher Konfiguration „verhält sich die Liste nicht wie erwartet“ – Debugging kann mühsam sein

# WIE FUNKTIONIERT DYNAMIC DATA?

- Beobachtbare Datenquellen (z. B. ObservableCollection, SourceList<T>)
- Erzeugen eines Change Sets, das Änderungen (Add, Remove, Replace…) verfolgt
- Transformation in ReadOnlyObservableCollection<T> für die UI-Bindung

# BEST PRACTICES

- Dispose nie vergessen – z. B. CompositeDisposable
- Kette klar strukturieren – Transformationen trennen
- Kombiniere nicht zu viel in einer Pipeline
- Nutze .AutoRefresh() bei Property-Änderungen
- Vermeide "Hot Observables", wenn nicht nötig

# WORAUF SOLLTE MAN ACHTEN?

- Threading – ObserveOn() nicht vergessen
- Fehlerbehandlung – Catch, Retry, Logging
- Zu viele Subscriptions? – Unübersichtlicher Code & potenzielle Leaks
- Performance bei großen Datenmengen – Benchmarks sinnvoll

# FAZIT UND DISKUSSION

- Q & A
- Code Review