**- Systementwurf -**

# QuantumCryptoCram

## Version: 1.6

| | | |
|---|---|---|
| Projektbezeichnung | QuantumCryptoCram | |
| Projektleiter | Johannes Sporrer | |
| Verantwortlich | SW-Architekt: C. Kennerknecht | |
| Erstellt am | 14.04.2021 | |
| Zuletzt geändert | 30.05.2021 17:30 | |
| Bearbeitungszustand | | in Bearbeitung |
| | | vorgelegt |
| | X | fertig gestellt |
| Dokumentablage | Quantumcryptocram\02_Entwurf\Systementwurf.docx | |

# Änderungsverzeichnis

| Änderung | | | Geänderte Kapitel | Beschreibung der Änderung | Autor | Zustand |
|---|---|---|---|---|---|---|
| Nr. | Datum | Version | | | | |
| 1 | 14.04.21 | 1.1 | 2,3 | Initiale Produkterstellung | C.K. | Fertig |
| 2 | 26.04.21 | 1.2 | 2,3 | Einpflegung Reviewkritik | C.K. | Fertig |
| 3 | 30.03.21 | 1.3 | 2.1 | Einpflegung Reviewkritik | C.K | Fertig |
| 4 | 07.04.21 | 1.4 | | Entfernung Vorlagetext und Erstellung Abbildungsverzeichnis | C.K., B.B., J.S. | Fertig |
| 5 | 16.05.21 | 1.5 | | Erstellung K. „Schnittstellen“ + Komponenten | C.K. | Fertig |
| 6 | 30.05.21 | 1.6 | | Erstellung K. “Entwicklungskonzept” + Designabsicherung und Verbesserungen | C.K | Fertig |

# Prüfverzeichnis

Die folgende Tabelle zeigt einen Überblick über alle Prüfungen – sowohl Eigenprüfungen wie auch Prüfungen durch eigenständige Qualitätssicherung – des vorliegenden Dokumentes.

| Datum | Geprüfte Version | Anmerkungen | Prüfer | Neuer Produktzustand |
|---|---|---|---|---|
| 20.04.21 | 1.1 | Siehe Mergerequest !15 | Benedikt Bartl | |
| 29.04.21 | 1.2 | Siehe Mergerequest !42 | Florian Hofmann | |
| 30.05.21 | 1.6 | Siehe Mergerequest !106 | Benedikt Bartl | |
| 30.05.21 | 1.6 | Siehe Mergerequest !106 | Elias Zell | |
| 31.05.21 | 1.6 | Siehe Mergerequest !106 | Jakob Götz | |

# Inhalt

# 1 Einleitung

Dieses Dokument soll ein Grundverständnis der Systemstruktur vermitteln, ohne den Entwurf in allen Einzelheiten darzulegen. Das Grundverständnis soll jedoch ausreichen, um sich ggf. anhand des Quellcodes in weitere Einzelheiten leicht einarbeiten zu können.

Kernthemen in diesem Dokument sind:

- Übersicht über die Zerlegung des Systems: Welche (größeren) Systemkomponenten gibt es? Wofür ist jede einzelne davon zuständig? Wie hängen diese Komponenten voneinander ab?
- Schnittstellenübersicht: Welche Schnittstellen stellt das System und jede Systemkomponente für seine/ihre Umgebung bereit?
- Systemkomponenten: Wie ist jede Systemkomponente aufgebaut?
- Designabsicherung: Zeigt für ausgewählte „architektur-relevante“ Use-Case-Szenarien, dass und wie diese mit dem gewählten Systementwurf realisierbar sind.

Der Systementwurf wird auf Grundlage der funktionalen und nicht-funktionalen Anforderungen sowie des konzeptuellen Datenmodells gewonnen, etwa indem man für ausgewählte „architektur-relevante“ Use-Case-Szenarien untersucht, welche Teile des Systems zur Realisierung in welcher Weise zusammenarbeiten müssen.

Die Gliederung dieses Dokuments orientiert sich grob am Aufbau der V-Modell-XT®[1]-Produkte „System-Architektur“ und „SW-Architektur“, ist jedoch zur Verwendung für die Veranstaltung **„Software-Projekte“** in Informatik-Curricula der **OTH-Amberg-Weiden** angepasst worden (und nicht konform zum V-Modell-XT).

# 2 Zentrale Architektur

Als zentrale Architektur für die Anwendung „QuantumCryptoCram“ wurde die sehr bekannten Architektur „Clean-Architecture“ verwendet, welche unter diesem Namen ursprünglich von Robert C. Martin 2012 bekannt gemacht worden ist (https://blog.cleancoder.com/uncle-bob/2012/08/13/the-clean-architecture.html). Dabei ist die Architektur keine „Neuerfindung“, sondern Martin versuchte die damals verschiedenen bekannten Architekturen zu vereinen, wobei er die Kerngedanken Aller identifiziert und diese eben unter dem Namen „Clean Architekture“ publiziert. Unter anderem bezieht er sich dabei auf der Hexagonalen Architektur“ von Alistar Cockburn (https://fideloper.com/hexagonal-architecture) bzw. der „Onion Architektur“ von Jeffry Palermo (https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/).
Die gemeinsamen Grundgedanken sind bzw. eine „Clean Architecture“ zeichnet aus:

- **Abstraktion der Benutzeroberfläche/Präsentation:**
  Die Technologie zur Darstellung der Anwendungsinhalte und Interaktion mit dem Benutzer kann leicht geändert werden, ohne die zentrale Logik der Anwendung ändern zu müssen.
  Genauso weiß die Anwendungslogik nichts über die Darstellung der eigenen Daten, sprich sie referenziert keine Oberflächen-Komponenten, und ist damit nicht an eine konkrete UI-Technologie (z.B. WPF, ASP .NET) gebunden.
- **Testbarkeit:**
  Die zentrale Anwendungslogik kann komplett ohne Benutzeroberfläche oder externen Abhängigkeiten getestet werden.
  **Unabhängigkeit von externen Abhängigkeiten bzw. Frameworks:**
- Die Anwendung bindet externe Abhängigkeiten (Datenbank, Logging, …) oder externe Frameworks (ORM, UI-Frameworks) streng entkoppelt über klar definierte Schnittstellen ein. Somit bleibt die Anwendung flexibel und kann leicht verwendete Technologien austauschen, falls diese veraltet sind oder sich Anforderungen geändert haben.

1 V-Modell® ist eine geschützte Marke der Bundesrepublik Deutschland.

Ein Vorteil einer solchen Architektur ist auch, dass Oberfläche und die Anwendungslogik größtenteils unabhängig voneinander entwickelt werden können, was vor allem bei großen Teamgrößen sehr wichtig ist.

Robert C. Martin und Jason Tylor stellen die „Clean Architecture“ grafisch als „Zwiebelmodell“ dar, wie auch schon Jeffry Palermo mit der „Onion-Architecture, wobei die einzelnen Schichten die logischen Abstraktionsebenen der Anwendung darstellen. Wir verwenden die konkrete Namensgebung der einzelnen Schichten von Tylor und gliedern nach dieser auch unsere Projektstruktur:

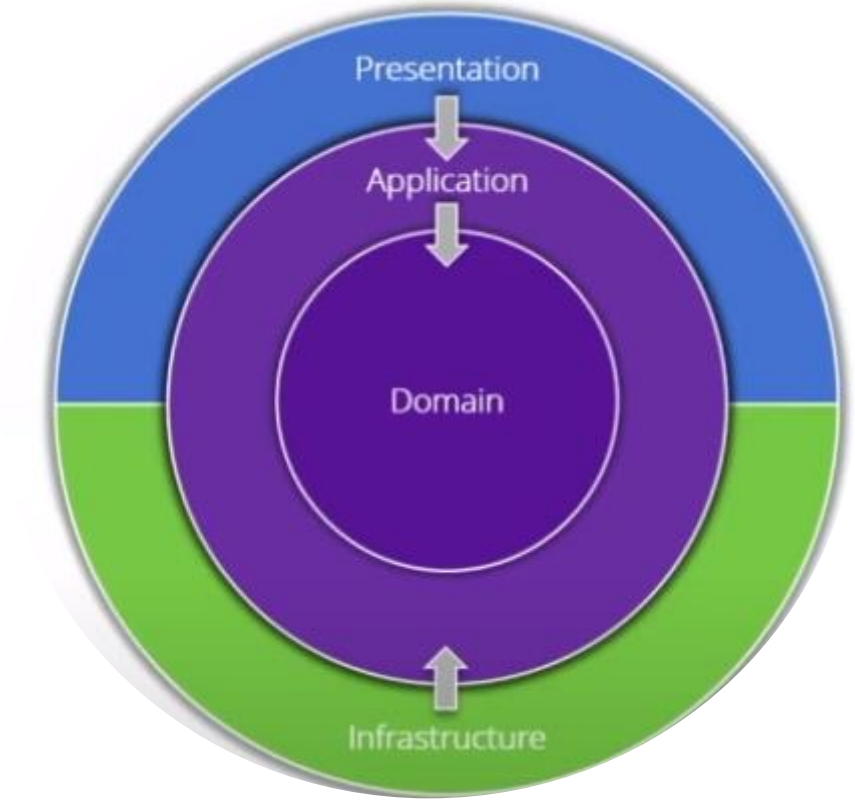


*Abbildung 1: Quelle: (https://youtu.be/5OtUm1BLmG0?t=13) "Clean Architecture with ASP.NET Core 3.0 - Jason Taylor - NDC Sydney 2019"*

## 2.1 Presentation

In dieser Schicht bzw. in diesem Projekt befindet sich das Frontend der Anwendung.
Durch die ausgewählte Architektur wurde zwar der Code für die Darstellung der Anwendung von der eigentlichen Anwendungslogik über Schnittstellen getrennt, allerdings fehlt noch eine saubere Trennung der Oberflächenlogik von dem konkreten Visualisierungsframework, in unserem Falle das „Windows Presentation Framework“.
Hierzu wurde das ebenfalls sehr bekannte Pattern „Model View ViewModel“ (MVVM) verwendet, welche gut mit der „Clean Architecture“ harmoniert und in Kombination mit dieser u.a. auch häufig bei der Web-/Mobile-Entwicklung verwendet wird. Die „View“-Dateien, in welchen mit Hilfe von XAML-Anweisungen das Aussehen der Anwendung definiert wird, werden in einem gleichnamiges Unterordner gegliedert. Gleiches gilt auch für die „ViewModel“-Klassen, welche Daten der Anwendungsschicht aggregieren und diese der Oberfläche zum Darstellen bereitstellen, bzw. Nutzeraktionen entgegennehmen und an die Anwendungsschicht weiterleiten.
Die Kopplung der Oberfläche und der „ViewModels“ findet dabei lose über das „Binding“-Prinzip, welches WPF nativ unterstützt, statt (Siehe: https://docs.microsoft.com/en-us/dotnet/desktop/wpf/data/).

## 2.2 Application

In dem Projekt „Anwendung“ befinden sich die Klassen, die die zentrale Logik der Anwendung enthalten. Hier werden die Nutzeraktionen verarbeitet, Modelklassen erstellt und modifiziert, Berechnungen vorgenommen und interne Datenstrukturen aktualisiert, welche dann wieder von den ViewModels erfasst

werden können. Dabei können auch Zugriffe auf externe Schnittstellen (Datenbank, Netzwerk, …) stattfinden. Außerdem werden in dieser Schicht die Schnittstellen (Interfaces) zu externen Diensten definiert. Die konkreten Implementierungen dieser Interfaces befinden sich meist im „Infrastructure“-Projekt.

## 2.3 Domain

In der „Domänen“-Schicht befinden sich hauptsächlich die „Model“-Klassen, also diejenigen Klassen die als schlichte Datencontainer für Objekte der realen Anwendungsdomäne dienen. Diese besitzen meist, bis auf evtl. Validierungen, keine eigene Logik. Außerdem können hier systemweite Enumerations, Exceptions und Events definiert werden.

## 2.4 Infrastructure

In dieser Schicht befinden sich konkrete, externe Ressourcen der Anwendung, welche dabei die Schnittstellen implementieren, die die Anwendung definiert. Diese können zum Beispiel ein File-System-Service, ein Datenbank-Service, oder ein Netzwerk-Server sein. Falls diese externen Ressourcen umfangreich sind, werden in der Praxis häufig anstatt einem „Infrastructure“-Projekte mehrere eigene Projekte erstellt, die z.B. für eine Datenbankanbindung den Namen „Persistance“ oder für einen TCP-Server den Namen „Networking“ tragen. Für das Project „QuantumCryptoCram“ wird zunächst nur ein „Infrastructure“-Projekt angelegt und bei Bedarf wie beschrieben untergliedert.

## 2.5 Common

Oft ergänzt man die vier zentralen Projekte noch durch ein „Common“-Projekt. Hier wird allgemeine grundlegende Logik untergebracht, die keine spezielle Anwendungslogik darstellt, und somit auch theoretisch leicht in anderen Anwendungen wiederverwendet werden kann.

## 2.6 Abhängigkeiten und „Dependency Injection“

Ein zentraler Gedanke der gewählten Architektur „Clean Architecture“ ist, dass Abhängigkeiten grundsätzlich nur von außen nach innen (bezogen auf die Zwiebeldarstellung) stattfinden dürfen. Das heißt Anwendungslogik darf nie Objekte/Methoden des konkreten UI-Frameworks referenzieren bzw. Modelklasse (Domain) darf nie von Anwendungslogik wissen. Eine sehr wichtige Rolle spielt hier auch das „Dependency Inversion“-Prinzip (Das „D“ der „SOLID“ Softwaredesigngrundregeln für objektorientierte Programmieren, welche ebenfalls von R. C. Martin gesammelt und publiziert worden sind). Dieses besagt ähnlich, dass zentrale Module der Anwendung nicht von „low-level“-Modulen abhängig sein dürfen. Stattdessen sollte beide Ebenen von gemeinsamen Abstraktionen (Interfaces) abhängen, sodass beide Ebenen lose gekoppelt sind und bei Bedarf leicht angepasst bzw. ersetzt werden können.

So besitzt zum Beispiel der Quellcode einer zentralen Anwendungsklasse keine Referenz auf eine „low-level“ „Logger“-Klasse, sondern verwendet lediglich ein Objekt vom Typ „ILogger“. Die konkrete Implementierung des „ILogger“-Interfaces, also die Logger-Klasse, könnte somit leicht ausgetauscht werden, ohne dass die Anwendungsklasse geändert werden müsste. Mit dieser Vorgehensweise kann es aber immer noch vorkommen, dass an einer Stelle der Klasse (meistens im Konstruktor) ein Objekt vom Typ „Logger“ instanziiert und für die interne Verwendung einer Member-Variablen vom Typ „ILogger“ zugewiesen werden muss. Solche internen „harten“ Referenzen machen vor allem beim Unit-Testen Probleme, denn für Unit-Tests möchte man oft für Interfaces eigene „Fake-Klassen“ verwenden, ohne dabei den Quellcode der Anwendungsklasse selbst ändern zu müssen.

Abhilfe schafft hier eine Vorgehensweise, das sogenannten „Dependency Injection Pattern“. Dabei werden alle externen Abhängigkeiten einer Anwendungsklasse nicht in der Klasse selbst instanziiert, stattdessen erhält diese eine Referenz des zum Beispiel „Logger“-Objektes über den Konstruktor (die gängigste Form), wobei die Instanziierung von der übergeordneten Klasse vorgenommen wird. Dieses Prinzip kann so lange fortgesetzt werden, bis gewissermaßen alle Abhängigkeiten, die in einer Anwendung auftreten, zentral in der Startroutine der Anwendung instanziiert werden und deren Instanzreferenz durch die gesamte Klassenhierachie „durchgereicht“ werden. Der klare Vorteil ist hier, dass an einer einzigen Stelle bestimmt werden kann, welche konkreten Implementierung für eine Abhängigkeit in der gesamten Anwendung verwendet werden soll. So kann z.B. für ein ILogger-Interface leicht von einem

„ConsoleLogger“ auf einen „FileLogger“ gewechselt werden, ohne dass andere Klassen angefasst werden müssten.
Ein Nachteil dieser Vorgehensweise ist jedoch, dass bei großen Klassenhierachien in einer höheren Ebene die Konstruktoren-Parametern einer Klasse schnell überfüllt werden und diese Interfaces listen, welche die Klasse selbst gar nicht benötigt.
Um dieses Problem in der Praxis zu lösen, entkoppelt man das eigentliche Instanziieren von Objekten und übergeben der Referenzen über sogenannten „Dependency-Injection-Container“. Ein solche Container besitzt Logik, um zum Start der Anwendung automatisch allen Klassen die jeweils im eigenen Konstruktor definierten Referenzen übergibt. In der Startroutine wird jetzt nur noch ein solcher Container konfiguriert.

Folgender Ausschnitt zeigt Konfigurationen des Dependency-Injection-Containers der Anwendung „QuantumCryptoCram“:

```
protected override void ConfigureIoC(IStyletIoCBuilder builder)
        {
            builder.Bind<IEncryptionService>().To<XorEncryptionService>();
            builder.Bind<IRandomGenerator>().To<RandomGenerator>();
            builder.Bind<SimulationOptions>().ToSelf().InSingletonScope();
            builder.Bind<IPhotonMeasurement>().ToFactory<IPhotonMeasurement>(container ⇒
            {
                if (container.Get<SimulationOptions>().IsPhotonCloningEnabled)
                {
                    return new PhotonMeasurementWithCloning(container.Get<IRandomGenerator>());
                }
                else
                {
                    return new RealPhotonMeasurement(container.Get<IRandomGenerator>());
                }
            });
        }
```

*Abbildung 2: Konfiguration eines DI-Containers bei Starten der Anwendung*

In der ersten Zeile wird einer Abstraktion „IEncryptionService“ in dem Fall die konkrete Implementierung „XorEncryptionService“ zugewiesen. Immer wenn jetzt in einem beliebigen Konstruktor ein „IEncrpytion-Service“ verlangt wird, wird automatisch eine neue Instanz der gewählten Implementierung erzeugt. Mann kann aber auch für die gesamte Anwendung eine einzige Instanz nach dem Singleton-Pattern verwenden. was wie hier bei Konfigurationsklassen sinnvoll ist. Dazu fügt man der Registrierung ein „InSingeltonScope()“ an. Sollte eine Klasse kein Interface implementieren, man aber dennoch in einem Konstruktor automatisch eine Instanz erhalten möchte, so kann man diese auf sich selbst mit „ToSelf()“ registrieren. Wenn für die Instanziierung von Implementierung eigene Logik benötigt wird, kann man mit einer inline „Factory“-Methode erzielen. So werden hier zum Beispiel, abhängig von einer Nutzereinstellungen, zwei verschiedene Implementierungen für das Messen von Photonen verwendet.
Sollte sich eine Implementierung durch möglicherweise andere Kundenanforderungen ändern, wäre hier z.B. der „XorEncryptionService“ leicht durch einen „AesEncryptionService“ austauschbar.

# 3 Übersicht über die Zerlegung des Systems

## 3.1 Projektstruktur

In der folgenden groben Übersicht, sind die gegenseitigen Referenzen der Projekte dargestellt. Diese folgen dabei der oben beschriebenen Architektur. Außerdem werden verwendete externen Pakete/Frameworks für jedes Projekt gelistet.

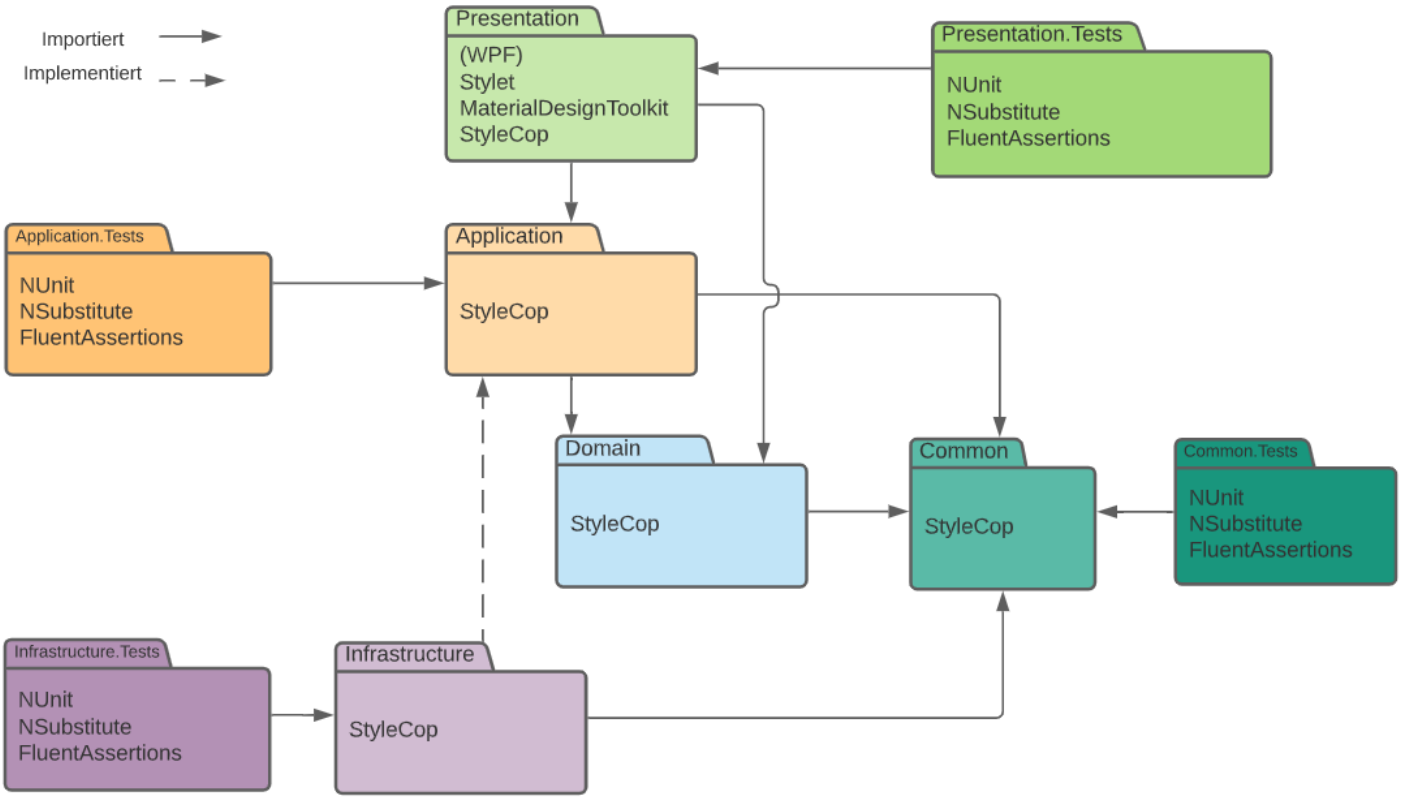


*Abbildung 3: Übersicht über die Projektstruktur und über verwendete externe Third-Party-Pakete*

## 3.2 Third-Party-Pakete

**Stylet**

- Webseite: https://github.com/canton7/Stylet/wiki
- GitHub: https://github.com/canton7/Stylet
- Kosten: Kostenlose Nutzung
- Lizenz: MIT License (Open Source)

Stellt grundlegende Funktionen für das effektive Nutzen des MVVM Musters zur Verfügung, wie zum Beispiel das Lifetime-Management von Views und das Navigieren zwischen Views. Außerdem besitzt es schon einen einfachen Dependency-Inversion-Container, welcher so oder so extern bezogen werden würde.

**Anmerkung:** Wir hätten zwar die Kompetenzen diese Grundlagen selbst zu implementieren, der Fokus der Entwicklung soll aber auf der eigentlichen Anwendungslogik liegen.

**Material Design in XAML Toolkit**

- Webseite: http://materialdesigninxaml.net/
- GitHub: https://github.com/MaterialDesignInXAML/MaterialDesignInXamlToolkitt
- Kosten: Kostenlose Nutzung
- Lizenz: MIT License (Open Source)

Grundlagen-Bibliothek zum Designen moderner WPF Oberflächen.

**NUnit3 – Unit Test Framework**

- Webseite: https://nunit.org/
- GitHub: https://github.com/nunit/nunit
- Kosten: Kostenlose Nutzung
- Lizenz: MIT License (Open Source)

Notwendig zum Erstellen und Ausführen von Unit-Test-Methoden.

**NSubstitute Mocking- Framework**

- Webseite: https://nsubstitute.github.io/help/getting-started/
- GitHub: https://github.com/nsubstitute/NSubstitute
- Kosten: Kostenlose Nutzung
- Lizenz: BSD 3-Clause (Open Source)

Notwendig, um den zu testenden Code von externen Abhängigkeiten (Logger, Config, Database, …) zu isolieren. Grundvoraussetzung für das Mocken ist, Abhängigkeiten nur über saubere Abstraktionen im Code einzubinden, sprich das „Dependency Inversion"-Prinzip zu beachten. NSubstitute hat eine leichtere und sauberere Syntax als das sehr populäre „Moq"-Framework, was vor allem Unit-Test-Einsteigern entgegenkommt.

**Fluent Assertions**

- Website: https://fluentassertions.com/
- GitHub: https://github.com/fluentassertions/fluentassertions
- Kosten: Kostenlose Nutzung
- Lizenz: Apache-2.0 (Open Source)

Vorteile

- Unit-Test-Assertions können leichter und intuitiver geschrieben werden.
- Assertions können in einer fließender (natürlicher Sprache) gelesen werden.
- Verwechseln vom „Actual" und „Expected" Parameter ist nicht möglich, was bei den Default-Assertions leicht passieren kann.
- Gibt ab Werk verständlichere/strukturierter Fehlermeldungen aus.

**StyleCop Analyzers**

- Webseite: https://github.com/DotNetAnalyzers/StyleCopAnalyzers/blob/master/DOCUMENTATION.md
- GitHub: https://github.com/DotNetAnalyzers/StyleCopAnalyzers
- Kosten: Kostenlose Nutzung
- Lizenz: MIT License (Open Source)

Statische Codeanalyse-Software, mit welcher man Coding-Style-Guidelines erzwingen kann.

# 4 Entwicklungskonzept

Die Domäne des BB84-Protokolles bietet Rollen, Objekte und Konzepte an, die sich sehr gut mit einer objektorientierten Programmierung modellieren lassen können. Daher stand dieses Paradigma mitsamt den gängigen Techniken wie Vererbung und Polymorphie bei der Entwicklung der Architektur bzw. des UML-Diagrammes im Fokus.
Im Fokus der Lernanwendung steht die Möglichkeit für den Anwender Protokollfehler machen zu dürfen. Um dies umzusetzen, muss die Anwendung Szenarien ermöglichen, die nicht dem Standard-Protokollablauf entsprechen. Das größte „eigenverschuldete“ Fehlerpotential bei dem Schlüsselaustauschprotokoll „BB84“ liegt im Allgemeinen darin, dass öffentliche Nachrichten entweder in der falschen Reihenfolge oder gänzlich unnötig gesendet werden. Deshalb erlaubt die Anwendung explizit, dass Alice und Bob jederzeit alle möglichen, öffentlichen Nachrichten senden dürfen. Für die Architektur der ViewModel-, Model- und Datenklassen bedeutet dies, dass gemeinsame Attribute oder Logik, die diese beiden Protokollpartner besitzen, immer in gemeinsamen Basisklassen mit dem Präfix „ProtocolPartner“ zusammengefasst werden. Gemeinsamkeiten die alle drei Rollen, also auch Eve, betrifft, werden in gemeinsamen Basisklassen mit dem Präfix „ProtocolRole“ implementiert.

Ebenfalls wurde die Architektur so ausgelegt, dass die Anwendung leicht mit einem Netzwerkmodus ausgestattet werden kann. Entsprechende Klassen besitzen das Präfix „Remote“. Primär wird jedoch der lokale Modus, bei dem die drei Protokollrollen zusammen an einem Rechner agieren, entwickelt. Hierfür erstellte Klassen verwenden das Präfix „Local“. „Local“ und „Remote“-Klassen implementieren immer ein gemeinsames Interface, sodass diese ja nach Modus-Wahl des Nutzers während der Laufzeit der Anwendung ausgetauscht werden können (siehe Kapitel: Abhängigkeiten und „Dependency Injection“).

Ein detailliertes UML-Diagramm liegt parallel zu diesem Dokument unter dem Namen „QuantumCryptoCram_UML.png“ bei. Das ausführliche UML-Diagramm hat zum einen sehr beim Planen der Abhängigkeiten bzw. für die richtige Verwendung des Dependency Injection Containers geholfen, zum anderen können darin die verschiedenen Schnittstellen, Implementierungen und Projektzugehörigkeiten leichter nachvollzogen werden. Des Weiteren soll das Diagramm als Entwicklungsanleitung für die initialen Implementierungsphase dienen.

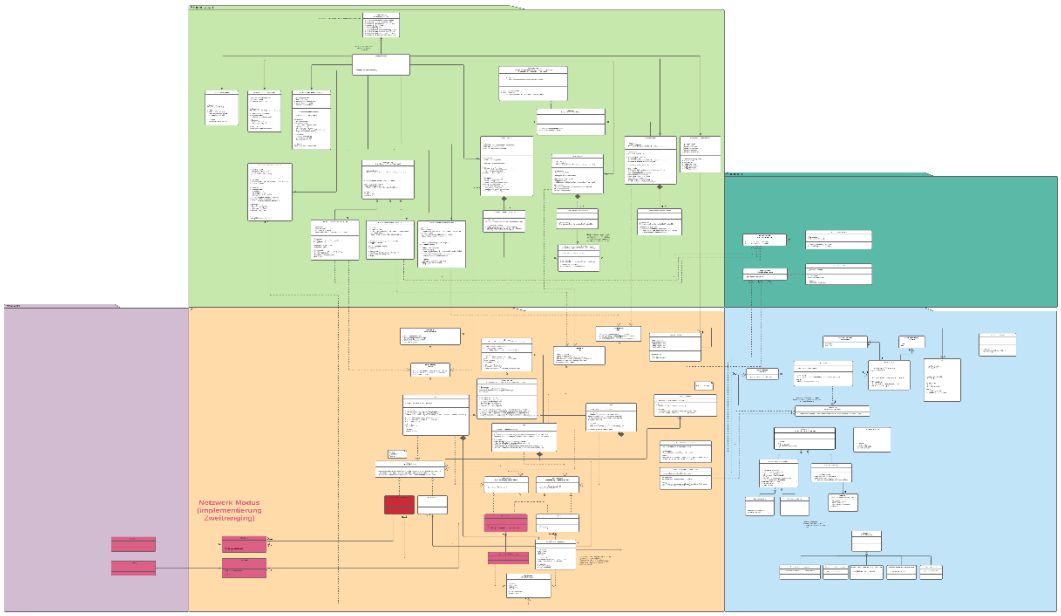

*Abbildung 4: Detailliertes UML Diagramm*

**Anmerkung:** Nachdem die Architektur und Abhängigkeiten feststanden und die erste Entwicklungsphase abgeschlossen war, wurden im UML nur noch neu eingeführte Klassen ergänzt. Klassenattribute und Methoden wurden meistens nicht aktualisiert, da der Synchronisationsaufwand schwer zu rechtfertigen war, solche Details wären stattdessen im versionierten Quellcode nachzusehen.

# 5 Schnittstellenübersicht

Im Folgenden werden die wichtigsten Schnittstellen der Anwendung betrachtet.

## 5.1 Abstraktion der Präsentations- und Applikationslogik

Die zentrale Anwendungslogik verbirgt sich hinter drei zentralen Schnittstellen, IAlice, IEve, IBob. In diesen wird genau definiert, welche Informationen die jeweiligen Klassen Alice, Eve, Bob (Modelklassen) veröffentlichen und entgegennehmen. Die entsprechenden ViewModels können damit alle Methoden bereits implementieren, ohne dass die echten Modelle vorhanden sind. Außerdem könnten die ViewModels damit leicht mit UnitTests getestet werden, denn Abhängigkeiten auf die echten Modelle können durch Interfaces für Tests gemockt werden und die Mocks anschließend validiert werden.

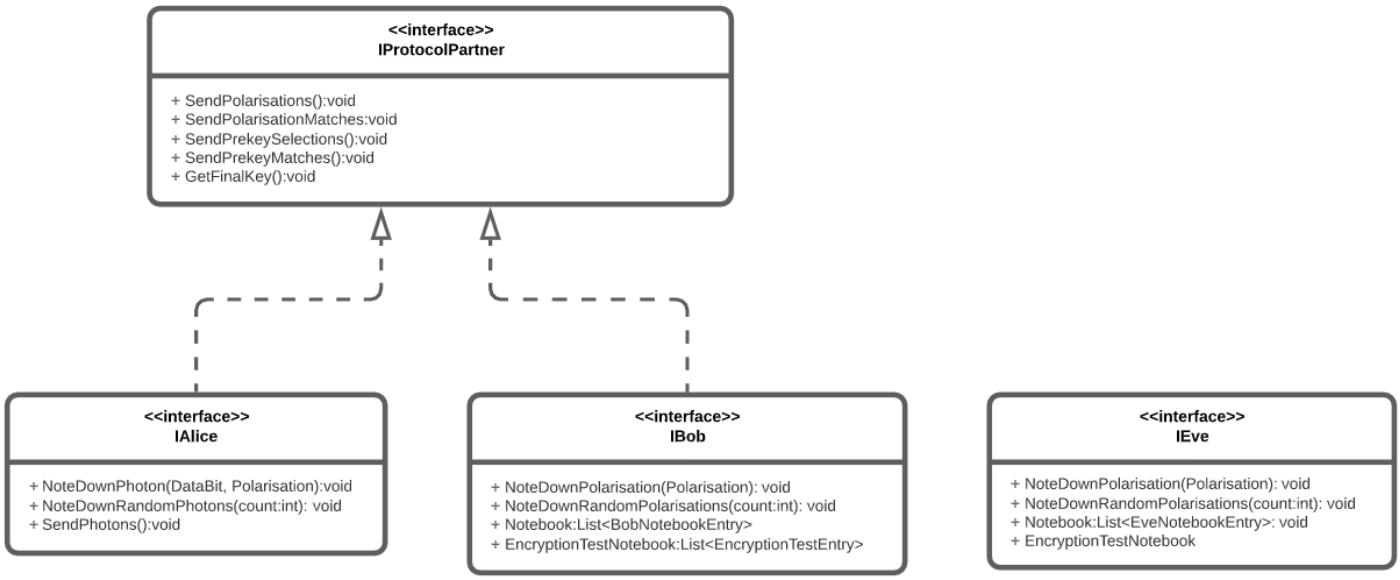


*Abbildung 5: Schnittstellen für die Hauptmodellklassen*

## 5.2 Abstraktion der Kommunikation (Lokal/Remote)

### 5.2.1 Konzept „Quanten-/Photonen-Kanal“

Um das Senden und Empfangen der Photonen unabhängig vom Lokalen oder Netzwerkmodus zu machen, hängen die Rollenklassen von entweder einem oder beiden der folgenden Interfaces ab:

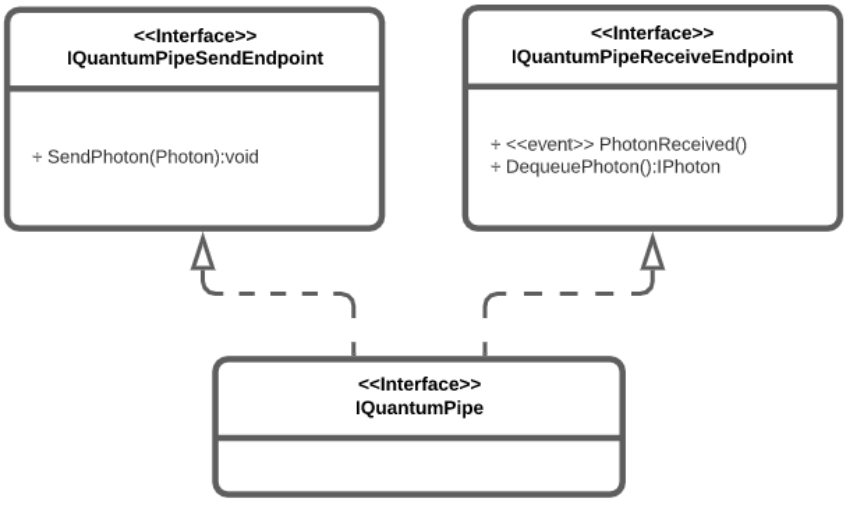


*Abbildung 6: Schnittstelle „IQuantumPipe"*

Im lokalen Modus werden beide Interfaces von einer „LocalPipe“ implementiert. Diese fungiert dabei nur als „Vermittler“, und ermöglicht der Sender-Klasse einen Delegaten der Empfänger-Klasse indirekt aufrufen zu können.

```
var pipeAE = new LocalQuantumPipe();
var pipeEB = new LocalQuantumPipe();
Alice = new Alice(pipeSendEndpoint: pipeAE);
Eve = new Eve(pipeSendEndpoint: pipeAE, pipeReceiveEndpoint: pipeEB);
Bob = new Bob(pipeReceiveEndpoint: pipeEB);
```

*Abbildung 7: Aufsetzten der unidirektionalen „QuantumPipe" zwischen den Hauptmodellklassen*

Für den Netzwerkmodus kann eine „RemotePipe“-Klasse erstellt werden. Diese ruft beim Senden (z.B. bei der Alice-Rolle) jetzt keinen Delegaten auf, sondern sendet über einen TCP-Client ein TCP-Telegramm (mit hartcodiertem Empfänger) an einen zentralen TCP-Server. Der Server vermittelt das Telegramm an die entsprechende Empfänger-Rolle, wobei hier zunächst über einen TCP-Client das Telegramm empfangen wird und an die eigene „RemotePipe“ weitergeleitet wird.

### 5.2.2 Konzept „Öffentliches Netzwerk“

Der zweite Kommunikationsweg, der für die Protokollteilnehmer relevant ist, ist das Konzept eines öffentlichen "ungesicherten" Kanals. Um dies umzusetzen, wird ein "Event Aggregator"- Pattern verwendet.

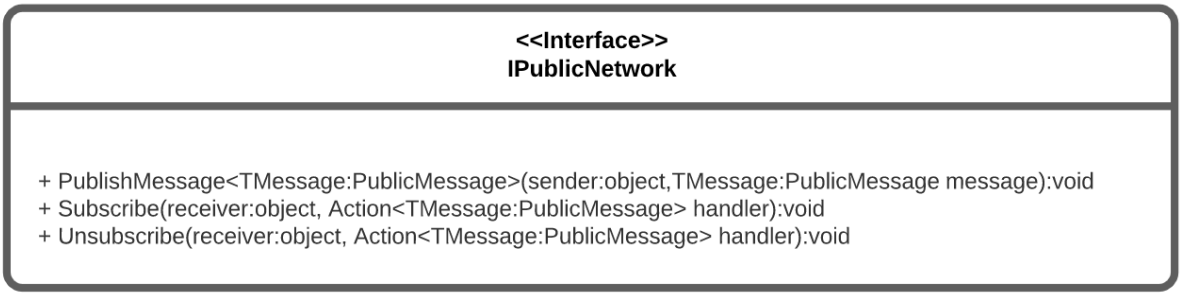


*Abbildung 8: Schnittstelle „IPublicNetwork"*

Dabei können beteiligte Klassen Rollen nach dem „Publish/Subscribe-Pattern“ (bzw. "Observer-Pattern") öffentliche Nachrichten absetzen und erhalten. Jedoch müssen hierbei die Klassen nichts voneinander wissen, denn das Vermitteln wird von einer zentralen Klasse, der "Aggregator" oder auch "Broker" genannt wird, gehandhabt. Im lokalen Modus wird durch das „LocalPublicNetwork“ z.B. bei einer eingehenden Nachricht (Event) ähnlich wie oben lediglich Delegaten aller Subscriber-Klassen aufgerufen. Im Netzwerk Modus werden Events durch das „RemotePublicNetwork“ über einen TCP-Client an den Server abgesendet. Im Gegensatz zur „RemotePipe“ werden die Nachrichten dann an alle anderen Netzwerkteilnehmer gesendet.

Der Event-Aggregator-Methoden wurde generisch gehalten, wobei Nachrichten die versendeten werden vom Typ „PublicMessage“ (siehe Kapitel „Datenmodell“) ableiten müssen.

## 5.3 Abstraktion des Konzeptes „Photonen Messen“

Es gibt zwar eine Domain-Klasse „Photon“ jedoch sollten Domänenklassen nur reine Datencontainer sein. Die Logik, wie ein Photon gemessen wird, wird daher durch ein Interface abstrahiert, welches in einer eigenen Klasse „PhotonMeasurement“ des Applikation-Projektes implementiert wird:

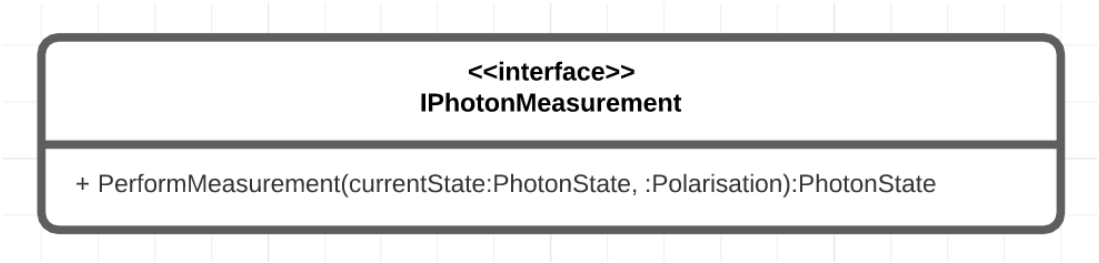


*Abbildung 9: Schnittstelle „IPhotonMeasurement“*

Je nachdem, ob für die Anwendung die Einstellung „Photonen Klonen“ aktiviert ist, wird das Interface mit einer anderen Implementierung erfüllt. Eine „RealPhotonMeasurement“-Klasse respektiert die quantenmechanischen Gesetzt und verändert den Zustand (bzw. die Polarisation) bei einer Messung.
In der Klasse „PhotonMeasurementWithCloning“ wird das Konzept des „Klonens“ eines Photons umgesetzt, indem sich hier bei einer Messung der innere Zustand (bzw. die Polarisation) eines Photons nicht ändert.
Das Abstrahieren dieser Logik erlaubt ebenfalls ein Mocken von dieser beim Unit-Testen, falls dies notwendig werden sollte.

## 5.4 Abstraktion des Konzeptes „Simulation Manager“

(Wurde im konzeptionellen Datenmodel noch „TopologyManger“ genannt.)

Da zum Instanziieren der Hauptklassen Alice, Eve, Bob diese richtig miteinander verknüpft („Quantum-Pipe“) werden müssen und dies vom Modus der Anwendung abhängig ist, wurde die entsprechende Logik durch eine Schnittstelle „ISimulationManager“ abstrahiert.

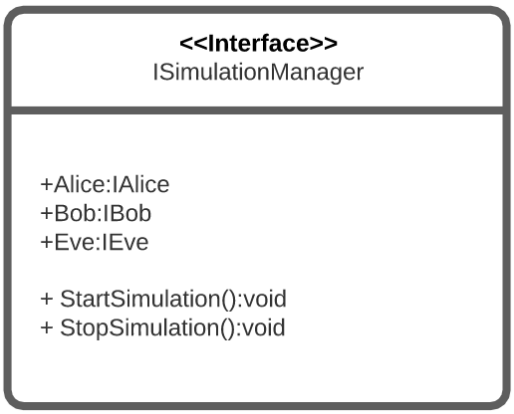


*Abbildung 10: Schnittstelle „ISimulationManager"*

Die entsprechenden Implementierungen („LocalSimulationManger“ bzw. „RemoteSimulationManager“) instanziieren zum Simulationsstart, nachdem die Simulationsoptionen von dem oder die Nutzer festgelegt sind, die Hauptklassen und veröffentlicht die Referenzen, welcher vom „Dependency Injection Container“ jeweils zum Erfüllen der Schnittstellen IAlice, IBob und IEve verwendet wird. Im Prinzip agiert eine „SimulationManager“-Klasse nach dem Factory-Pattern als „verlängerter Arm“ für den DI-Container. Dieses Muster wird häufig dann verwendet, wenn man beim Erstellen der Referenzen, wie in unserem Fall, zusätzliche Logik benötigt. Die Implementierung des „ISimulationManager“ selbst wird dabei ebenfalls als Singleton im DI-Container registriert, sodass über die gesamte Anwendung hinweg ein und dieselben Referenzen der Hauptklassen verwendet werden. Beim Beenden der Simulation werden die Objekte der Hauptklassen ordentlich aufgeräumt (Dispose-Pattern, siehe „https://docs.microsoft.com/en-us/dotnet/standard/garbage-collection/implementing-dispose“), sodass diese für eine neue Simulation wieder komplett neu ohne alte Daten instanziiert werden können.

# 6 Systemkomponenten

Die wichtigsten Systemkomponenten/-klassen wurde bereits im letzten Kapitel bereits über ihre Schnittstelle beschrieben. Ein weiteres zentrales Konzept der Andendung QuantumCryptoCram stellt das Datenmodell dar:

## 6.1 Datenmodell

Da WPF generell sehr gut für das Darstellen von Datenbanktabellen geeignet ist und unsere Protokoll-Rollen große Datenmengen verwalten müssen, hat sich angeboten, jeder Rolle eine eigene „In-Memory“-Tabelle zu geben. Eine echte Datenbankanbindung wäre zwar leicht mithilfe eines ORM-Systems, wie das „Microsoft-Entity-Framework“, ergänzbar. Dies ist jedoch nicht notwendig, da nach einer abgeschlossenen Simulation die Daten nicht persistent gespeichert werden müssen.

Jede Zeile einer Tabelle enthält dabei alle nützlichen Informationen, die eine Rolle im Verlauf des Protokolls über ein einziges Photon „sammelt“. Ein Großteil dieser Informationen wird auf der Oberfläche dargestellt, was dem Nutzer beim besseren Verständnis des BB84-Protokolls unterstützen soll, da dieser den gesamten Weg vom Erstellen eines Photons bis zum Verwenden derselben Photoneninformation als geheimen Schlüssel mitverfolgen kann. Gleichzeitig soll die Rolle des Angreifers erkennen, dass selbst mit allen nützlichen Informationen ein Angriff praktisch unmöglich ist.

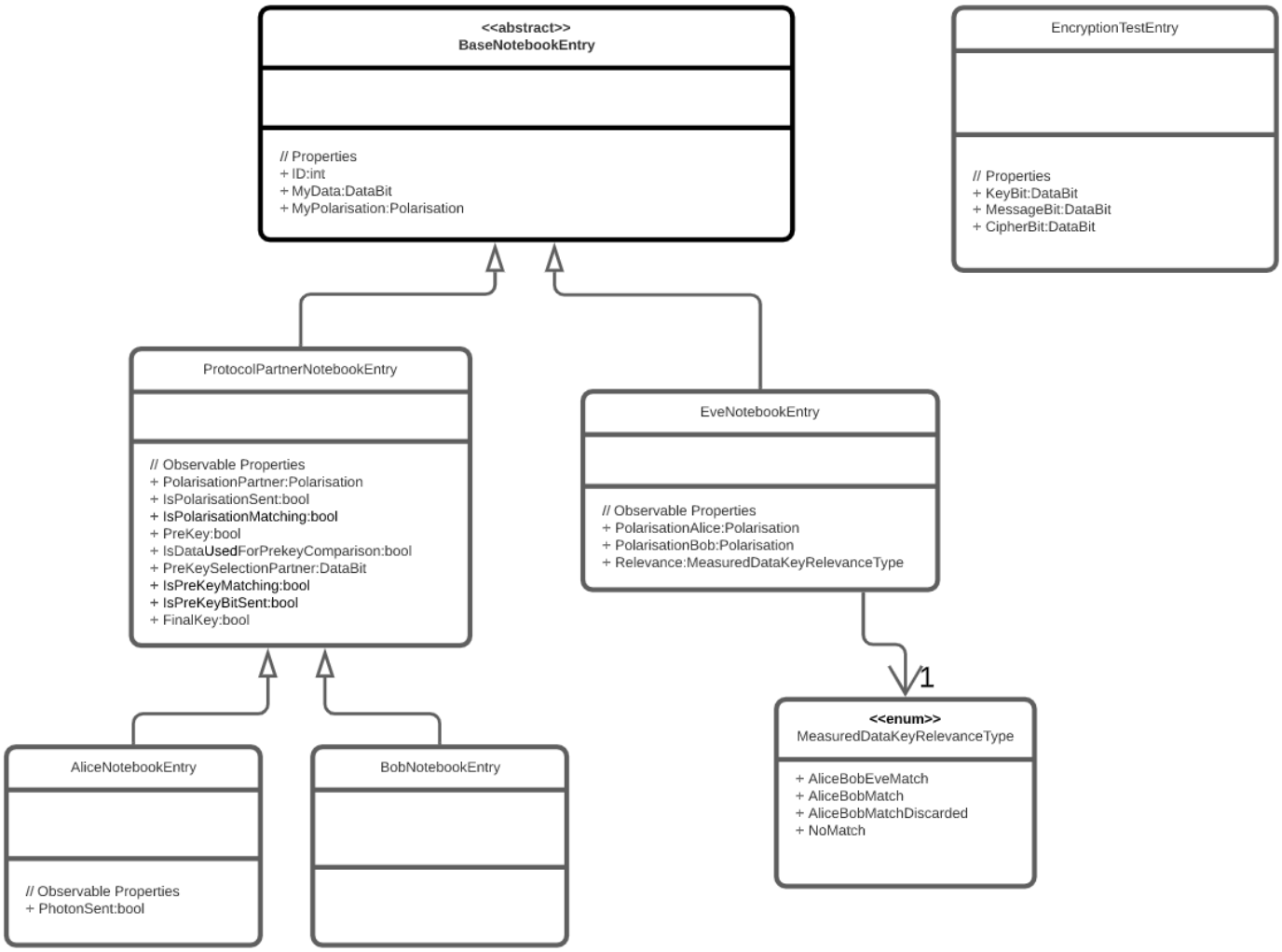


*Abbildung 11: Datenmodell zum Verwalten von BB84-Protokoll-Informationen*

Da Alice und Bob fast die gleichen Informationen verwalten müssen, erben diese, ähnlich wie bei den Schnittstellen, von einer gemeinsamen abstrakten Klasse „ProtocolPartnerNotebookEntry“.

Information, welche alle drei Rollen gemeinsam besitzen, werden in der gemeinsamen abstrakten Basisklasse „ProtocolRoleNotebookEntry“ definiert. Zum Beispiel besitzt jeder Eintrag eine eindeutige ID, wobei für jedes Photon aller drei Tabellen dieselbe ID (Primärschlüssel) erhalten soll, sodass sich Parteien beim öffentlichen Informationsaustausch auf ein und dasselbe Photon beziehen können. Außerdem besitzt jede Rolle eine eigene Vorstellung über den inneren Zustand eines Photons, welcher in unserer Simulation mithilfe der Enumerationen „Polarisation“ und „DataBit“ dargestellt wird.


*Abbildung 12: Repräsentation eines Photonenstatus im Datenmodell*

Für die öffentliche Kommunikation wurden einfache Datenklassen angelegt, die sich Protokollrollen als „Nachricht“ zuschicken können:

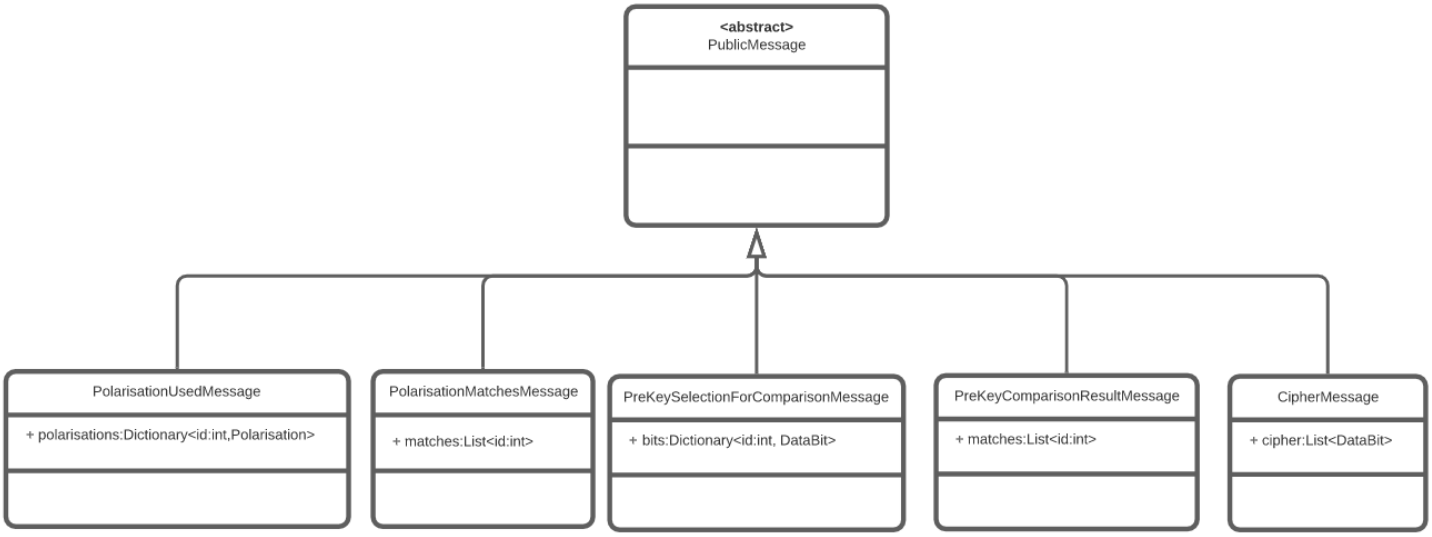


*Abbildung 13: Klassen, die öffentliche Nachrichten im BB84-Protokoll repräsentieren.*

Jede Klasse spiegelt dabei eine Art des öffentlichen Austausches im BB84-Protokoll wieder, die Namen sind dabei selbsterklärend gewählt worden.

# 7 Designabsicherung

Im Folgenden wird näher betrachtet, wie das MVVM-Muster zusammen mit der gewählten Architektur konkret funktioniert. Dazu wird das Szenario betrachtet, dass ein Nutzer die Anwendung startet im lokalen Modus startet (UC-2), eine Simulation startet (UC-3) in die Alice-Oberfläche navigiert (UC-5) und eine Photonenkonfiguration ins Notebook (UC-11) einträgt.

Für dieses Szenario sind der Reihenfolge nach die vier Oberflächen bzw. Views „MainMenuView“, LocalModeView“, „SimulationOverviewView“ und „AliceView“ relevant.

Um zwischen diesen Views zu navigieren, verwenden wird den vorgegebenen Mechanismus des MVVM-Frameworks „Stylet“. Stylet gibt eine sogenannte „ShellView“ vor, die als eine Art „leere Hülle“ immer die View anzeigt, die im entsprechenden „ShellViewModel“ als „ActiveItem“ gesetzt ist. Zum Ändern des „ActiveItem“, also zum Ändern der angezeigten View, veröffentlicht das „ShellViewModel“ die Methode „NavigateTo(IScreen screen)“. Wird diese Methode mit einer eigenen View als Parameter aufgerufen, dann schaltet Stylet automatisch die richtige View auf und instanziiert gleichzeitig das entsprechende ViewModel, welches Daten-Kontext für die View agiert.

Beim Start der Anwendung wird zunächst immer auf das „MainMenuView“ navigiert.
Von dort aus navigiert der Nutzer manuell weiter auf die „LocalModeView“. Das „LocalModeViewmodel“ wird automatisch instanziiert, wobei mithilfe des DI-Containers eine Singelton-Referenz auf „SimulationOptions“ erlangt wird. Der Nutzer kann jetzt Einstellungen über die Oberfläche vornehmen, wobei Textfelder und Buttons auf Eigenschaften des „LocalModeViewmodel“ gebunden sind. Beim setzten einer solchen Eigenschaften werden erneut Eigenschaften der „SimulationOptions“ gesetzt, welche damit persistent über die Simulationslaufzeit gespeichert bleiben und für andere Klassen verfügbar sind.

Durch das eigentliche Starten der Simulation über einen Button navigiert der Nutzer in die „SimulationOverviewView“. Wiederum wird das „SimulationOverviewViewModel“ inszanziiert. Dieses erhält für die Abhängigkeit „ISimulationManager“ eine Referenz auf den „LocalSimulationManager“, da dies so für den lokalen Modus im DI-Container registriert ist. Im Konstruktor des „SimulationOverviewViewModel“ wird vom „LocalSimulationManager“ die „StartSimulation()“-Methode gerufen. Jetzt erzeugt der „LocalSimulationManager“ die Instanzen der Modelklassen „Alice“, „Eve“ und „Bob“ und verbindet diese über eine „LocalQuantumPipe“. Für den gesamten Verlauf einer Simulation hält der „LocalSimulationManager“ diese Instanzreferenzen also „am Leben“. Dies ist notwendig, da die Modelobjekte ständig im Hintergrund öffentliche Nachrichten bzw. Photonen erhalten müssen, auch wenn diese selbst gerade nicht von deren entsprechenden Viewmodel-Objekten „geladen“ sind. Außerdem werden die Modelklassen-Instanzen von diesem Moment an vom DI-Container verwendet, wenn in Konstruktoren nach „IAlice“, „IEve“ oder „IBob“-Referenzen „gefragt“ wird.

Dies ist beim nächsten Schritt des Szenarios der Fall, wenn der Nutzer auf die „AliceView“ navigiert und das „AliceViewModel“ von Stylet instanziiert wird. Im Konstruktor des „AliceViewModel“ wird zu Beginn gleich der der aktuelle Datenstand bzw. das Notebook der Alice-Klasse geladen. (Der Datenstand kann sich ändern, wenn z.B. zwischen der „AliceView“ und „BobView“ mehrmals navigiert, bzw. gearbeitet wird.) Beim Laden des Notebooks des „Alice“-Objektes wird jedes einzelne, darin enthaltene „AliceNotebookEntry“-Objekt mit einem entsprechenden „AliceNotebookEntryViewModel“-Objekt gewrappt. Diese Technik ist zwar in dem MVVM-Muster nicht zwingend notwendig, allerdings wird sie immer dann verwendet, wenn man den Datenobjekten zusätzlich oberflächenspezifische Logik oder Attribute geben möchte (siehe: https://nathan.alner.net/2010/02/09/mvvm-to-wrap-or-not-to-wrap-how-much-should-the-viewmodel-wrap-the-model-part-1/).

Im letzten Schritt des Szenarios wird vom Nutzer „NoteDownPhotonConfiguration“ betätigt. Im XAML-Code von Buttons einer View kann man definieren, welche Methode Stylet bei einem Button-Klick von dem entsprechenden ViewModel aufrufen soll. In diesem Fall wird die Methode „NoteDownPhotonConfigurationCommand“ des „AliceViewModel“ aufgerufen. In diesen „…Command“-Methoden können Nutzereingaben validiert und konvertiert werden. Anschließend werden eine oder mehrere Methoden der Modelobjekte aufgerufen. In diesem Fall wir die Methode „NoteDownPhotonConfiguration(DataBit dataBit, Polarisation polarisation)“ des Alice-Objekts gerufen, wobei das aktuell vom Nutzer gewählte Datenbit und die Polarisation übergeben wird.

In der Methode des Alice-Objektes wird jetzt ein neuer Eintrag vom Typ „AliceNotebookEntry“ im eigenen Notebook (Datentabelle) angelegt und Datenbit und Polarisation eingetragen. Bildlich wird hier der

Prozess modelliert, wie Alice vor dem Senden der Photonen deren Konfigurationen festlegt und diese in einer Tabelle „niederschreibt“. Das Notebook ist dabei eine „ObservableCollection“ von „AliceNotebookEntry“-Objekten.

Eine „ObservableCollection“ wird nativ vom .NET-Framework zur Verfügung gestellt und löst immer dann ein „CollectionChanged“-Event aus, wenn die Sammlung größer wird (oder kleiner wird, was aber für diese Anwendung nicht relevant ist). Dabei wird das neu hinzugefügte Element als Event-Parameter übergeben. Das „AliceViewModel“ abonniert dieses Event und erfährt so, wenn es neue „AliceNotebookEntry“-Elemente gibt, sodass diese, wie bereits schon beschrieben, erneut mit „AliceNotebookEntryViewModel“-Objekten gewrappt werden können. Die „AliceNotebookEntryViewModel“-Elemente werden von dem „AliceViewModel“ wiederum als „ObservableCollection“ gehalten, welche jetzt von Komponenten der „AliceView“, z.B. von einem „WPF DataGrid“, als „ItemSource“ verwendet werden kann.
Eine „WPF GridColumn“ kann jetzt auf ein einzelnes Attribut bzw. eine Property des „AliceNotebookEntry“- bzw. „AliceNotebookEntryViewModel“-Objektes gebunden werden.

Die „Datenobjekte implementieren ebenfalls ein sogenanntes „INotifyPropertyChanged“-Interface, sodass immer, wenn sich ein eine Eigenschaft ändert, ein “PropertyChanged“-Event ausgelöst wird und somit die View aktiv die Anzeige für die Eigenschaft aktualisieren kann. Insgesamt ist dies das gängige Vorgehen im MVVM-Muster in WPF, wenn Sammlungen an Datenobjekten in der Oberfläche dargestellt werden müssen. In diesem Beispiel-Szenario würde jetzt in der „AliceView“ ein neuer Eintrag in der Notebook-Tabelle erscheinen und die beiden Spalteneinträge „DataBit“ und „Polarisation“ einen Wert anzeigen. Die restlichen Spalteneinträge sind noch leer und werden dann im Laufe der Simulation durch andere UseCases gefüllt.

# 8 Abkürzungsverzeichnis

| Abkürzung | Erklärung |
|---|---|
| SOLID | Akronym für wichtige Prinzipen des objektorientierten Programmierens: „Single-responsibility principle“, “Open-closed principle”, “Liskov substitution principle”, “Interface segregation principle” und “Dependency inversion principle” |
| AES | Advanced Encryption Standard |
| ORM | Ein “Object Relational Mapper“-System abstrahier den Zugriff auf eine Datenbank, indem automatisch „In Memory“-Objektlisten mit den Tabellen einer echten Datenbank synchronisiert werden |
| TCP | Transport Control Protocol |
| MVVM | Bekanntes „Model View ViewModel“ Programmierungsmuster für die Oberflächenprogrammierung. |

# 9 Abbildungsverzeichnis