# Aufgabenstellung zur SmartHome-Simulation

## Ziel:

Im Rahmen der Module m320 und v320 wurde eine SmartHome-Simulation entwickelt. Diese dient als Ausgangspunkt für das Code-Review:

* Die Lösung soll analysiert und auf mögliche Verbesserungen untersucht werden.
* Schwerpunkt: **Clean Code Richtlinien**, **SOLID-Prinzipien**, und das **Erkennen von Code Smells**.

# Projektdaten zur SmartHome

## Wichtige Inhalte aus der Lösung:

### Kernkomponenten:

* BadWC.cs: Verwaltung von Bad/WC-Daten im SmartHome-System.
* Heizungsventil.cs: Steuerung und Verwaltung der Heizungsventile (z.B., Öffnungs- oder Temperaturzustand).

### Tests (Qualitätssversicherung):

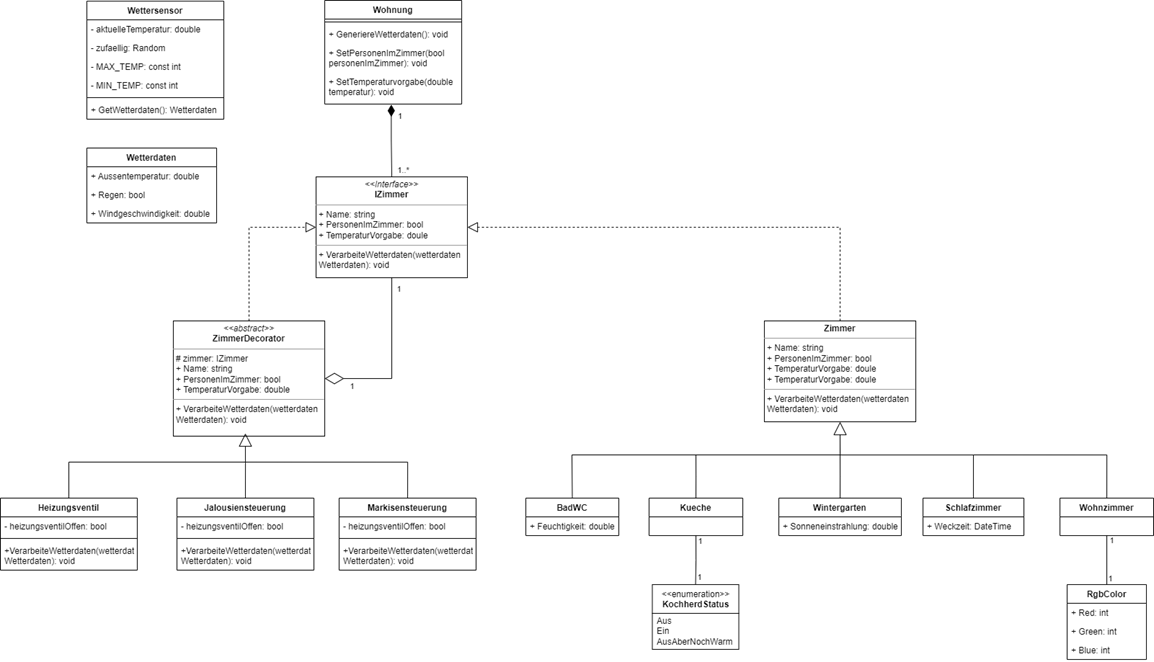
* HeizungsventilTests.cs: Testen der Heizungsventil-Funktionalitäten.
* ZimmerTests.cs: Überprüfung der Zimmer-Logik und zugehöriger Methoden.

## Struktur der Lösung:

**Visual Studio** (Projekt und Testprojekte vorhanden):

* Enthält das Hauptprojekt **"M320\_SmartHome"**.
* Testprojekt: **"M320\_SmartHomeTests"** mit Unit-Tests.

# Klassendiagramm



# Testkonzept:

**Testkonzept für SmartHome-Projekt**

## 1. Testziel:

Das Ziel des Testens in diesem Projekt ist es, sicherzustellen, dass die Logik des **Heizungsventils** gemäss den festgelegten Regeln funktioniert. Insbesondere soll geprüft werden, ob das Heizungsventil je nach den **aussentemperatur** spezifisch öffnet oder geschlossen bleibt.

## 2. Testmethoden:

Die Hauptmethode für das Testen ist der **Unit-Test**, da wir die Funktionalität der Klassen (Heizungsventil) und ihre Methoden direkt aufrufen und testen.

* Der **Unit-Test** wird verwendet, um die Funktion der Methode VerarbeiteWetterdaten() zu prüfen, die bestimmt, ob das Heizungsventil öffnet oder nicht, basierend auf den Wetterdaten.
* In den Tests werden spezifische Temperaturen getestet, die das Öffnen oder Schliessen des Ventils beeinflussen sollen.

## 3. Testarten:

Die Tests beinhalten hauptsächlich **Unit-Tests** und einige Tests mit gezielten Eingabewerten, die simulieren, wie das **Heizungsventil** auf unterschiedliche Temperaturen reagieren soll.

## 4. Teststruktur:

Die zu testenden Methoden sind in die Klasse **Heizungsventil** integriert. Der Testcode überprüft, ob **bei bestimmten Temperaturen** das **Heizungsventil** wie erwartet auf **geschlossen** oder **offen** setzt.

Die folgenden Tests werden durch die TestMethod-Annotation in der Klasse **ZimmerTests** vorgenommen:

## Tests:

1. **TestHeizungsventilBei20GradNichtÖffnet**
   * **Ziel:** Sicherstellen, dass das Heizungsventil nicht geöffnet wird, wenn die Aussentemperatur bei **20°C** liegt.
   * **Vorgehensweise:** Der Test simuliert das Wetter mit einer Aussentemperatur von **20°C** und prüft, ob das Heizungsventil geschlossen bleibt.
   * **Erwartetes Ergebnis:** zimmer.HeizungsventilOffen muss **false** zurückgeben.
2. **TestHeizungsventilBei19GradNichtÖffnet**
   * **Ziel:** Sicherstellen, dass das Heizungsventil nicht geöffnet wird, wenn die Aussentemperatur bei **19°C** liegt.
   * **Vorgehensweise:** Der Test simuliert das Wetter mit einer Aussentemperatur von **19°C** und prüft, ob das Heizungsventil geschlossen bleibt.
   * **Erwartetes Ergebnis:** zimmer.HeizungsventilOffen muss **false** zurückgeben.
3. **TestHeizungsventilBeiMinus25GradÖffnet**
   * **Ziel:** Sicherstellen, dass das Heizungsventil bei **-25°C** geöffnet wird.
   * **Vorgehensweise:** Der Test simuliert das Wetter mit einer Aussentemperatur von **-25°C** und prüft, ob das Heizungsventil öffnet.
   * **Erwartetes Ergebnis:** zimmer.HeizungsventilOffen muss **true** zurückgeben.
4. **TestHeizungsventilBei35GradNichtÖffnet**
   * **Ziel:** Sicherstellen, dass das Heizungsventil nicht geöffnet wird, wenn die Aussentemperatur bei **35°C** liegt.
   * **Vorgehensweise:** Der Test simuliert das Wetter mit einer Aussentemperatur von **35°C** und prüft, ob das Heizungsventil geschlossen bleibt.
   * **Erwartetes Ergebnis:** zimmer.HeizungsventilOffen muss **false** zurückgeben.

**5. Testressourcen und Tools:**

* **Testframework:** MSTest
* **Testdaten:** Verschiedene Aussentemperaturen als Eingabedaten.
* **Tools:** Microsoft Visual Studio mit integriertem Test-Runner.

**6. Testablauf:**

* **Testvorbereitung:** Zunächst müssen die entsprechenden Objekte (wie **Heizungsventil**, **Zimmer**, **Wetterdaten**) instanziiert werden.
* **Testdurchführung:** Die Methode VerarbeiteWetterdaten() wird für verschiedene Temperaturwerte aufgerufen.
* **Testüberprüfung:** Anschliessend wird die Eigenschaft HeizungsventilOffen überprüft, um sicherzustellen, dass das Ventil korrekt geöffnet oder geschlossen wird.

## Testbeschreibung:

* **Test 1: TestHeizungsventilBei20GradNichtÖffnet**
  + **Ziel:** Überprüfen, dass das Heizungsventil nicht öffnet, wenn die Aussentemperatur 20°C beträgt.
  + **Vorbedingungen:** Ein Heizungsventil wird instanziiert und mit Wetterdaten von 20°C versehen.
  + **Test:** Wir rufen zimmer.VerarbeiteWetterdaten(wetterdaten) auf.
  + **Erwartetes Ergebnis:** HeizungsventilOffen bleibt **false**.
* **Test 2: TestHeizungsventilBei19GradNichtÖffnet**
  + **Ziel:** Überprüfen, dass das Heizungsventil bei 19°C nicht öffnet.
  + **Vorbedingungen:** Ein Heizungsventil wird instanziiert und mit Wetterdaten von 19°C versehen.
  + **Test:** Wir rufen zimmer.VerarbeiteWetterdaten(wetterdaten) auf.
  + **Erwartetes Ergebnis:** HeizungsventilOffen bleibt **false**.
* **Test 3: TestHeizungsventilBeiMinus25GradÖffnet**
  + **Ziel:** Sicherstellen, dass das Heizungsventil bei -25°C öffnet.
  + **Vorbedingungen:** Ein Heizungsventil wird instanziiert und mit Wetterdaten von -25°C versehen.
  + **Test:** Wir rufen zimmer.VerarbeiteWetterdaten(wetterdaten) auf.
  + **Erwartetes Ergebnis:** HeizungsventilOffen wird **true** zurückgeben.
* **Test 4: TestHeizungsventilBei35GradNichtÖffnet**
  + **Ziel:** Sicherstellen, dass das Heizungsventil bei 35°C nicht geöffnet wird.
  + **Vorbedingungen:** Ein Heizungsventil wird instanziiert und mit Wetterdaten von 35°C versehen.
  + **Test:** Wir rufen zimmer.VerarbeiteWetterdaten(wetterdaten) auf.
  + **Erwartetes Ergebnis:** HeizungsventilOffen bleibt **false**.

# Code-Review-Checkliste nach Clean Code und SOLID

## Code-Style & Clean Code

1. **Einheitlichkeit:**
   * **Namenskonventionen:**
     + Werden einheitliche Namen für Klassen, Methoden und Variablen verwendet?
     + **Empfohlene Praktiken:** CamelCase für lokale Variablen und Methoden, PascalCase für Klassen und Schnittstellen.
     + Werden gängige Benennungsrichtlinien beachtet (z.B., **I** vor Schnittstellen, methoden als Verben)?
     + **Prüfung:**
       - Beispiel: Eine Klasse wie Heizungsventil und HeizungsventilManager sollte konsistent benannt sein.
   * **Einrückungen und Klammer-Positionierung:**
     + Wird der Code konsistent eingerückt (normalerweise 2 oder 4 Leerzeichen)?
     + Werden Klammern für Methoden und Kontrollstrukturen in der gleichen Art und Weise verwendet?
     + **Prüfung:**
       - Jede neue Code-Blockebene soll eine konsequente Einrückung haben.
2. **Verwendung von sprechenden Namen:**
   * Werden Variablen, Klassen und Methodennamen so gewählt, dass der Zweck sofort verständlich ist?
     + **Beispiel:** Anstelle von var x könnte man temperaturStellen verwenden.
   * **Prüfung:**
     + Jede Klasse oder Methode sollte durch ihren Namen ihre Verantwortung ausdrücken. Z.B. eine Klasse wie Heizungsventil beschreibt ihr Ziel und ihre Funktion, jedoch sollte auch VerbrauchsRechner bei Bedarf klar die Funktionsweise vermitteln.
3. **Redundanz vermeiden:**
   * Gibt es redundante Codeabschnitte, die zu wiederholen sind, z.B. doppelte Berechnungen oder inkonsistente Implementierungen von Funktionen?
   * **Prüfung:**
     + **Example:** Wenn an mehreren Stellen ähnliche Berechnungen auftauchen, ziehe in Erwägung, sie in einer Methode zu kapseln oder in eine Base-Klasse auszulagern.
   * **Vermeidung von Copy-Paste:** Keine gleichartigen Codeabschnitte ohne generische Implementierungen.
4. **Kommentierung:**
   * Wird der Code so geschrieben, dass keine unnötigen Kommentare nötig sind?
   * **Empfohlene Vorgehensweise:**
     + Schreibe Code in einer Art und Weise, dass er "selbstsprechend" ist.
     + Verwende Kommentare nur dort, wo sie zur Erklärung von nicht eindeutig verständlichen Teilen des Codes oder von Entscheidungen nötig sind.
     + **Beispiel:** Vermeide Kommentare wie // Funktioniert wie erwartet.
   * **Prüfung:**
     + Teste, ob du nach der Lesung eines Teils des Codes ohne Kommentar noch alles nachvollziehen kannst.
     + **Korrektur:** Zu viele Kommentare im Code können auf einen schlecht strukturierten Code hinweisen.
5. **Single Responsibility (Funktion/Methode):**
   * Erfüllt jede Methode **nur eine einzige Aufgabe**?
     + Eine Methode sollte eine einzige Aufgabe ausführen, eine Eingabe in ein Ausgabe-Verhältnis.
     + **Prüfung:**
       - Jede Methode sollte kürzer als 20 Zeilen sein und eine einzige Funktionalität bieten.
       - Beispiele:
         * Eine Methode, die sowohl das Einlesen einer Datei als auch die Verarbeitung von Daten macht, verletzt das Prinzip der **Single Responsibility** und sollte aufgeteilt werden.
6. **Magische Zahlen/Strings vermeiden:**
   * Werden keine magischen Zahlen oder String-Konstanten ohne Erklärung verwendet?
     + **Beispiel:**

csharp

Code kopieren

int maxTemperatur = 100;

* + - Magische Zahlen sollten durch eine benannte Konstante ersetzt werden:

csharp

Code kopieren

const int MaxTemperatur = 100;

* + **Prüfung:**
    - Ersetze hart codierte Werte durch benannte Konfigurationen oder Konstanten, um den Code leichter wartbar und verständlich zu machen.

## Weitere Prüfungspunkte für Code-Review

## Prüfung auf SOLID-Prinzipien:

1. **Single Responsibility Principle (SRP):**
   * Überprüfe, ob Klassen nur **eine Verantwortung** haben.
   * Beispiel: Eine Klasse, die sowohl Logik zur Temperaturregelung als auch Logik zur Kommunikation mit anderen Komponenten wie einem Heizsystem enthält, sollte getrennt werden.
2. **Open/Closed Principle (OCP):**
   * Der Code sollte erweiterbar sein, ohne dass bestehende Codeblöcke geändert werden.
     + Wird eine Klasse durch **Vererbung oder Interfaces** erweiterbar, ohne dass Änderungen an bestehenden Codes notwendig sind?
3. **Liskov Substitution Principle (LSP):**
   * Funktionieren Programme, wenn untergeordnete Klassen zur Laufzeit ersetzt werden, ohne das Verhalten zu verändern?
4. **Interface Segregation Principle (ISP):**
   * Wurde ein Interface nicht zu gross oder universell gestaltet, sodass es zu breiten Abhängigkeiten führt?
5. **Dependency Inversion Principle (DIP):**
   * Wird vom Code eine **Abstraktion** statt einer konkreten Klasse abhängig gemacht?