1 Santa's Workshop: Objektorientiertes Chaos retten

1.1 Einleitung

Es ist der 23. Dezember, spät abends. In Santas Werkstatt herrscht Hochbetrieb, doch hinter den Kulissen gibt es ein ernstes Problem: Das bisherige Verwaltungssystem ist dem Weihnachtsstress nicht mehr gewachsen. Zu viele verschiedene Geschenktypen, Sonderfälle und Lieferarten haben den Code unübersichtlich gemacht. Änderungen führen ständig zu neuen Fehlern, Verantwortlichkeiten sind unklar und Tests fehlen.

Santa braucht dringend ein neues System klar strukturiert, objektorientiert und erweiterbar. Eure Aufgabe ist es, die Abläufe in der Werkstatt zu analysieren und ein Programm zu entwickeln, das Geschenke, Elfen und Lieferungen sauber modelliert. Das System soll so aufgebaut sein, dass neue Geschenk- oder Lieferarten ohne grossen Umbau ergänzt werden können und zuverlässig getestet sind.

Ob Weihnachten pünktlich kommt, hängt dieses Jahr von eurem Design ab.

1.2 Teamarbeit

Ihr arbeitet in euren bereits zugeteilten Teams.
Jedes Teammitglied übernimmt dabei **eine feste Rolle**, die während des Projekts **sichtbar dokumentiert** wird.

Hinweis
Alle Inhalte sind **Stoff für die kommende Abschlussprüfung im M320**.
Ziel ist daher **Verständnis, Design und Begründung**, nicht nur lauffähiger Code.

1.3 Ziel des Projekts

Entwickelt gemeinsam ein **objektorientiertes C#-Programm**, das Santas Werkstatt modelliert.

Dabei müsst ihr:

- objektorientiert **denken**
- UML als **Analysewerkzeug** einsetzen
- **Interfaces**, **Vererbung**, **Polymorphismus** und **Dependency Injection** korrekt anwenden
- zeigen, dass euer Design **testbar** ist

1.4 Rollen im Team

Zwingend mindestens einmal, dürfen aber auch mehrmals im Team vorkommen.

1.4.1 Rolle: Analyst / Designer

Verantwortung

- Fachliches Verständnis der Aufgabenstellung
- Entwurf des UML-Klassendiagramms
- Entscheidungen zu:
    - Klassen
    - Verantwortlichkeiten

    - Vererbung vs. Assoziation
- Erklärung der OOP-Prinzipien anhand des Designs

Muss abgeben:

- UML-Klassendiagramm
- kurze Designbegründung (½ Seite)

1.4.2 Rolle: Implementierer / Entwickler

Verantwortung:

- Umsetzung des UML-Designs in C#
- saubere Objektorientierung:
    - Kapselung
    - Abstraktion
    - Vererbung
    - Polymorphismus
- Einsatz von:
    - Interfaces
    - Collections (List<T>)
    - Constructor Injection

Muss abgeben:

- lauffähiges C#-Projekt
- Code-Kommentare zu wichtigen Designentscheidungen

1.4.3 Rolle: Tester / Qualitätsverantwortliche:r

Verantwortung:

- Überprüfung des Designs auf Testbarkeit
- Schreiben von Unit Tests
- Anwendung des **AAA-Prinzips**
- Erklärung:
    - warum Dependency Injection Tests vereinfacht
    - warum stark gekoppelte Klassen problematisch sind

Muss abgeben:

- mindestens 2 Unit Tests
- kurze Test-Reflexion

2 Teilabschnitt A: Analyse und UML

2.1 Gemeinsame Analyse

Als Team klärt ihr:

- Welche **Objekte** gibt es in Santas Werkstatt?
- Welche **Aufgaben** haben sie?
- Welche Beziehungen bestehen?

Beispielbereiche:

- Geschenke
- Elfen
- Lieferung
- Sonderverhalten von Geschenken

2.2 UML-Klassendiagramm

Der/die **Analyst:in** führt, das Team diskutiert.

Pflicht im Diagramm:

- mindestens **eine abstrakte Klasse**
- mindestens **ein Interface**
- **Vererbung (ist-ein)**
- **1:n-Beziehung mit Collection**
- Sichtbarkeiten (+ / − ggf. #)

Einige Beispiel-Ideen:

- `Gift «abstract»`
- `ToyGift, FoodGift, MagicGift`
- `IShippable`
- `Elf`
- `Workshop`
- `Reindeer`

2.3 Reflexion (Teamantworten)

Schriftlich beantworten:

1. Warum ist die gewählte Vererbung eine **ist-ein-Beziehung**?
2. Wo wäre Vererbung **falsch** oder ungeeignet?
3. Warum sind Collections in Klassen **private**?

3 Teilabschnitt B: Implementierung und Polymorphismus

3.1 Umsetzung in C#

Der/die **Implementierer:in** codet, das Team reviewt.

Pflicht:

- Attribute sind private
- Zugriff nur über Methoden / Properties
- Polymorpher Methodenaufruf über:
  - Interface **oder**
  - Basisklasse
- mindestens eine **Constructor Injection**

3.2 Interfaces und Dependency Injection

Als Team:

- definiert ein Interface (z. B. Lieferung, Verpackung, Kontrolle)
- verwendet es **polymorph**
- erklärt schriftlich:
  - Unterschied Interface / abstrakte Klasse / normale Klasse
  - warum Abhängigkeiten **von aussen** übergeben werden

4 Teilabschnitt C: Sequenzdiagramm, Tests und Qualität

4.1 Sequenzdiagramm

Der/die **Tester:in** führt, das Team unterstützt.

Erstellt ein Sequenzdiagramm für:
«Ein Elf erstellt ein Geschenk und übergibt es zur Lieferung»

Darzustellen:

- beteiligte Objekte
- Methodenaufrufe
- Objekterzeugung
- Verantwortlichkeiten

4.2 Unit Tests

- mindestens **2 Unit Tests**
- AAA-Prinzip klar sichtbar
- Teste **eine kleine Einheit**

Zusatzfrage (schriftlich)
«Warum wäre dieser Test ohne Dependency Injection schwieriger?»

5 Teilabschnitt D: Zusammensetzen des Systems

In `Program.cs` wird **nicht Logik implementiert**, sondern das **System zusammengesetzt**.

Hier zeigt ihr:

- Polymorphismus
- Dependency Injection
- Trennung von Verantwortung
- sauberes Design

5.1 `Program.cs` richtig verwenden

Hier dürft ihr:

- Objekte **erzeugen**
- Abhängigkeiten **zusammenstecken**
- Interfaces mit konkreten Implementierungen **verdrahten**
- Beispielabläufe **starten**

Hier dürft ihr **NICHT**:

- Fachlogik implementieren
- Entscheidungen treffen, die in Klassen gehören
- grosse if-/switch-Blöcke schreiben

5.1.1 Pflichtinhalte

Euer `Program.cs` muss:

1. **Mindestens ein Interface polymorph verwenden, z. B.**
   `IDeliveryService deliveryService = new ReindeerDelivery();`
2. **Constructor Injection sichtbar machen, z. B.**
   `Workshop workshop = new Workshop(deliveryService);`
3. **Mit Basisklassen oder Interfaces arbeiten, z. B.**
   `List<Gift> gifts = new List<Gift>();`
4. **Einen klaren Ablauf starten**
   z. B.:
   Geschenke erzeugen
   Elf erstellt / bearbeitet Geschenke
   Übergabe an Lieferung

5.1.2 Reflexionsfragen zu Program.cs

Diese Fragen beantwortet ihr **schriftlich**:

1. Warum wird in Program.cs eine **konkrete Implementierung** gewählt?
2. Warum kennt Workshop **nur das Interface**, nicht die konkrete Klasse?
3. Was müsste man ändern, wenn morgen ein
   DroneDelivery statt ReindeerDelivery verwendet wird?

6 Abgabe

Abgabeinhalt (1 ZIP-Archiv pro Team)

1. UML-Klassendiagramm
2. C#-Projekt
3. Sequenzdiagramm
4. Rollen-Dokumentation
5. Reflexionsdokument (ca. 1-1½ Seiten insgesamt über alle Teammitglieder)

6.1 Kriterien Schwerpunkte

- Objektorientiertes Denken
- saubere Verantwortlichkeiten
- korrekter Einsatz von:
    - OOP-Prinzipien
    - UML
    - Interfaces
    - Dependency Injection
- Testbarkeit und Begründungen
- **Teamarbeit sichtbar**

7 Hinweis

KI darf verwendet werden aber muss explizit gekennzeichnet werden und **überpüft wird** ob ihr:

- Entscheidungen versteht
- Design erklären könnt
- Zusammenhänge erkennt