|  |
| --- |
| Testing Konzept FortressForge |

Das Ziel dieses Testkonzepts ist es, eine fehlerfreie und benutzerfreundliche Nutzungserfahrung des FortressForge Programmes zu gewährleisten. Das Konzept bestimmt die wesentlichen Bereiche, die überprüft werden müssen, um die reibungslose Funktionalität sicherzustellen.  
Wir möchten uns dabei integrierte / automatisierte Tests zu hilfe nehmen welche das refactoring vereinfachen sollten aber auch einen Teil auf Manuellem Testung basieren aufgrund der Architektur und der gewählten Frameworks / Unity etc.

Des Weiteren Haben wir uns dazu entschieden, dass wir keine Test Coverage von 100% anstreben aus mehreren Gründen. Durch unsere Verwendung von Unity und Fish.net als Framework haben wir eine exponentiell wachsende Komplexität in unser Programm eingeführt. Dies führt dazu, dass es Bereiche gibt in denen Sich das Testen…

* … weniger lohnt (Weil ein Teil des zu behandelnden Codes bereits von Unity oder Fish.net abgedeckt wird)
* ... als zu komplex herausstellt (Weil Unity und Fish.net eine Lernkurve ausweisen welche grösser ist als die uns zur verfügbar stehenden Projektzeit)
* … beinahe unmöglich automatisiert umzusetzen ist (Weill es mit unserem Wissenstand und Zeitvorgabe nicht möglich wäre diese sinngemäss zu implementieren, als Beispiele wäre das grossflächige testen des Multiplayers mit mehreren Clients über externe Server oder Hosts)
* … erübrigt, weil die Funktionen so zentral integriert sind, dass sie direkt beim Starten des Spiels auffallen würden.

Wie Oben vermerkt bereiteten uns das Testen der Online Synchronisierung, automatisierte Tests mit Unity, Spiel Performance Test und UI-Test die grössten Probleme. Da es sich nach wie vor um ein Spiel handelt und mit zukünftigem Wachstum der Gamelogik sowieso eine exponentielle tiefe und Test Cases erwartet wird ist der Ansatz zum Manuellen End-testing sinnvoll eingesetzt. Demnach werden wir was wir können und als sinnvoll erachten mit automatischen Tests abdecken und den Rest manuell durch das einfache Spielen des Spiels evaluieren.

**Testarten:**

|  |  |
| --- | --- |
| Unit - Tests | Durch Unit-Tests wird ein einzelner abgegrenzter Teil einer Applikation getestet, bspw. eine Methode. Dabei wird ein klarer Input- und Output-Wert definiert und verglichen. |
| Unity - Tests | Stellt die Verknüpfung der Unit Tests und der Entwicklerplattform Unity dar. |
| Manuelle - Tests | Beinhalten manuelle Nutzer Tests in Form von dem einfachen Spielen des Spiels. |

**Testumgebung:**

Wir nutzen JetBrains Rider als DIE neben der Entwicklungsplattform Unity für die Entwicklung und Tests. Die Plattform ermöglicht effizientes Schreiben und Debuggen von zuverlässigen Tests.

Nachfolgend folgt eine Auflistung unserer Test Cases und Anwendungen die wir testen zuzüglich einer kleinen Beschreibung.

**FortressForge Test-Cases**:

* **CameraControllerTests.cs**  
  Der CameraController.cs ist für die Steuerung der Kamera Bewegungen in game zuständig. Dank dem in Unity integriertem Test Framework war es uns möglich eine grosse Anzahl Test für den Kamera Controller zu erstellen welche die Bewegungsabläufe in verschiedenen Situationen der Kamera Evaluiert. Da dies die ersten Test waren die von uns implementiert wurden, dientes sie uns als Lernmaterial für zukünftige.
* **EconomySystemTest.cs**  
  Die EconomySystemTest.cs Klasse validiert die Kernlogik des Wirtschaftssystems in FortressForge. Ziel war es zu prüfen, ob Ressourcen korrekt berechnet werden und ob Gebäude bei einem negativen Ressourcenhaushalt korrekt deaktiviert werden. Durch gezielte Testszenarien mit simulierten Akteuren (Mocks) konnten verschiedene Zustände abgebildet und zuverlässig überprüft werden. Dabei lag der Fokus auf dem Zusammenspiel zwischen Ressourcenkonsum und -produktion sowie der Reaktion des Systems auf kritische Ressourcenengpässe. Diese Tests stellen sicher, dass das Wirtschaftssystem auch bei komplexeren Strukturen stabil und nachvollziehbar funktioniert.
* **ResourceTests.cs**  
  Die ResourceTests.cs konzentriert sich auf die Überprüfung der Methoden zur Ressourcenüberprüfung und -zahlung innerhalb des EconomySystems. Hierbei wird getestet, ob die Funktion CheckForSufficientResources korrekt erkennt, ob genügend Ressourcen für einen Bauvorgang vorhanden sind – und ob PayResource diese dann ordnungsgemäss abzieht. Die Tests decken verschiedene Szenarien ab, darunter ausreichend, exakt ausreichende und unzureichende Ressourcen sowie leere Kostenstrukturen. Diese Testklasse ist entscheidend für die Gewährleistung der Konsistenz bei Ressourcenoperationen und stellt sicher, dass keine unbeabsichtigten Abzüge oder logischen Fehler bei knappen Ressourcen auftreten.
* **AutoSizeLabelTests.cs**  
  Diese Testklasse validiert das Verhalten der AutoSizeLabel-Komponente, welche die Schriftgrösse dynamisch an die Grösse ihres übergeordneten Elements anpasst. Die Tests überprüfen u. a., ob die Komponente korrekt initialisiert wird, wie sie auf verschiedene Containergrössen reagiert, ob Ränder und Abstände berücksichtigt werden und ob Schwellenwerte zur Vermeidung unnötiger Aktualisierungen funktionieren. Spezialfälle wie leere Texte, minimale Fontgrössen sowie Breiten- oder Höhenbeschränkungen werden ebenfalls abgedeckt. Der Einsatz von UnityTest ermöglicht dabei die asynchrone Prüfung von UI-Änderungen im Editor-Modus und sichert eine realistische Testumgebung.
* **FillableRessourceContainerTests.cs**  
  Diese Testklasse prüft die visuelle und funktionale Korrektheit des FillableRessourceContainer, eines UI-Elements zur Anzeige von Füllständen. Die Tests decken die Initialisierung, Grenzwerte für die Füllmenge (Clamping zwischen 0 und 1), sowie die dynamische Anpassung der Container-Füllung basierend auf der Orientierung (horizontal oder vertikal) ab. Zudem wird sichergestellt, dass CSS-Klassen korrekt hinzugefügt werden, ungültige Werte ignoriert werden und sich Layout-Änderungen korrekt auf die Darstellungslogik auswirken. Die Tests nutzen Assertions auf VisualElement-Styles, um UI-Berechnungen exakt zu verifizieren.
* **GameOverlayTest.cs**  
  Diese Testklasse führt UI-Tests für das GameOverlay in Unity durch, insbesondere für das Tab-System und die Ressourcendarstellung. Sie prüft, ob zentrale UI-Elemente wie Trapezrahmen, Ressourcensicht, BuildingSelectorView und deren Unterstrukturen korrekt im Overlay vorhanden sind. Ausserdem wird das Verhalten beim Wechseln von Tabs getestet, inklusive Sichtbarkeit der jeweiligen Inhalte. Die Tests nutzen Unitys PlayMode-Tests mit UnityTest und kontrollieren DOM-Strukturen (VisualElement, TemplateContainer, TabView) auf Basis von Klassen- und Namenstypen. Eine Retry-Logik mit Timeout sichert gegen Race-Conditions beim Szenenladen ab.
* **TrapezElementTest.cs**  
  Diese Testklasse überprüft die Funktionalität des benutzerdefinierten TrapezElement-UI-Elements in Unity. Sie verwendet InputTestFixture, um Mausinteraktionen zu simulieren, und testet das Verhalten des Trapezelements bei verschiedenen Konfigurationen und Klickpositionen. Testschwerpunkte:  
   - Korrekte Initialisierung mit und ohne CSS-Selector (ClassListContains).

- Verhalten von IsPointInTrapez() bei Klicks innerhalb und ausserhalb des Trapezbereichs (auch wenn sie innerhalb des VisualElement-Bereichs liegen).

- Initialisierung des Testdokuments erfolgt über ein Prefab (UIDocumentTesting) aus dem Resources/Prefabs/Tests-Ordner.

- Nutzung relativer Bildschirmkoordinaten zur präzisen Steuerung von Mauspositionen im UI-Kontext.

- Die Tests prüfen also sowohl Logik als auch visuelles Verhalten des UI-Elements im Zusammenspiel mit der Input-System-Simulation.

* **HexGridTest.cs**  
  Die HexGridTest.cs-Klasse testet die Logik zur Platzierung von Gebäuden innerhalb des hexagonalen Rastersystems in FortressForge. Ziel ist es zu prüfen, ob das Platzierungssystem korrekt zwischen belegten, ungültigen und freien Feldern unterscheidet. Dabei wird ein simuliertes Terrain über einen FakeTerrainHeightProvider bereitgestellt, um externe Abhängigkeiten wie tatsächliche Geländehöhen zu vermeiden.

Im Setup wird ein vollständiger Simulationskontext aufgebaut, einschliesslich BuildingManager, EconomySystem und HexGridManager. Der HexGridData-Instanz wird ein Starter-Grid mit Radius 7 zugewiesen. Innerhalb dieses Testkontexts wird überprüft, ob die Methode ValidateBuildingPlacement erwartungsgemäss auf belegte oder ungültige Felder mit false und auf gültige, freie Felder mit true reagiert.

Die verwendeten Gebäudeformen basieren auf vier benachbarten Hex-Feldern, die im Test manuell definiert werden. Ein zentrales Testziel ist es, fehlerhafte Platzierungen zuverlässig zu verhindern – etwa bei überlappenden Tiles oder Koordinaten ausserhalb des Gitters. Gleichzeitig soll bei korrekter Konstellation eine gültige Platzierung garantiert sein.

Diese Testklasse dient als wichtiger Baustein zur Validierung des Platzierungssystems im Bausystem von FortressForge und stellt sicher, dass die Spiellogik auf Rasterebene robust und deterministisch funktioniert.

* **GlobalGridMeshRendererTest.cs**  
  Diese Testklasse überprüft die Funktionalität des GlobalGridMeshRenderer, welcher für die Darstellung des globalen Hexfeld-Rasters zuständig ist. Im Setup wird ein GameObject mit den benötigten Komponenten (MeshFilter, MeshRenderer, GlobalGridMeshRenderer) erstellt und mit einem Mock für die Geländehöhen sowie einer Konfiguration initialisiert.

Der Test Mesh\_IsGenerated\_WithVerticesAndIndices stellt sicher, dass das generierte Mesh korrekt erstellt wird und sowohl Scheitelpunkte als auch Indizes enthält. Damit wird garantiert, dass die visuelle Darstellung des Hexfeldrasters nicht leer bleibt und korrekt aufgebaut wird.

Durch das Verwenden eines Mock-Terrain-Providers kann der Test isoliert von der echten Gelände-Logik ausgeführt werden, was die Stabilität und Zuverlässigkeit der Tests erhöht.

* **HexTileHelperTest.cs**  
  Diese Tests prüfen verschiedene Hilfsmethoden des HexTileHelper im Umgang mit Hexfeld-Koordinaten.

ExtractShapeInformation wird getestet mit leeren und gefüllten Listen von HexTileEntry, um sicherzustellen, dass korrekt Koordinaten und Flags extrahiert werden.

GetRotatedShape wird auf verschiedene Rotationswinkel überprüft:

Leere Listen bleiben leer.

0° Rotation gibt die Original-Koordinaten zurück.

180° Rotation dreht die Koordinaten korrekt (z.B. (1,0) → (-1,0)).

Vollrotationen (360°, 720°) liefern wieder die Original-Koordinaten zurück.

GetAveragePosition berechnet den Mittelwert von Weltpositionen aus Hex-Koordinaten, sowohl für einzelne als auch für mehrere Punkte.

Die Tests validieren die korrekte Funktionsweise wichtiger Transformations- und Berechnungsmethoden im Hexfeld-System.

* **CreateGameRoomTest.cs**  
  Diese Testklasse überprüft die Kernfunktionen der Spielraum-Erstellung und -Verwaltung in FortressForge. Ziel ist es, sicherzustellen, dass ein Spielraum korrekt geöffnet wird, Spielerlisten richtig dargestellt und aktualisiert werden sowie Clients erfolgreich verbunden und entfernt werden können. Zusätzlich werden wichtige UI-Elemente wie Start- und Exit-Buttons, sowie IP- und Passwortanzeigen geprüft. Tests zum Start eines Matches und dem Verlassen des Raums durch den Server sind ebenfalls enthalten. Durch die Kombination von Unit- und Integrationstests mit Unitys Testframework wird die Stabilität des Lobby-Systems sichergestellt, inklusive Netzwerkverbindungen über FishNet.
* **LobbyGameRoomTestBaseSetup.cs**  
  Die Klasse LobbyGameRoomTestBaseSetup.cs bildet die Basis für UI-Tests der Lobby- und GameRoom-Oberflächen in FortressForge. Sie stellt grundlegende Setup-Routinen und Hilfsmethoden bereit, die in den abgeleiteten Testklassen genutzt werden können.

Im Setup wird die „LobbyScene“ geladen und wichtige UI-Elemente wie das LobbyMenu und der ViewManager identifiziert und überprüft, um sicherzustellen, dass die Tests auf einer korrekten Ausgangssituation aufbauen. Die Verknüpfung zu einer leeren Szene als nächstes Ziel (NextScene) wird ebenfalls hergestellt, um Seiteneffekte zu vermeiden.

Die Hilfsmethoden prüfen einerseits, ob Textfelder korrekt vorhanden sind und Eingabewerte angenommen werden. Andererseits wird die Sichtbarkeit und Aktivierung von VisualElements überprüft, was essenziell für UI-Tests ist, da Elemente zwar vorhanden, aber unsichtbar oder deaktiviert sein können.

Diese Basis bietet somit eine solide und wiederverwendbare Grundlage, um UI-Komponenten konsistent zu validieren, was besonders bei komplexeren Benutzeroberflächen mit mehreren Tabs und dynamischen Elementen wichtig ist. Die Vermeidung von Null-Referenzen und die Prüfung auf Sichtbarkeit stellen sicher, dass nachfolgende Tests präzise und stabil arbeiten.

* **LobbyUIVisualTest.cs**  
  Die Testklasse LobbyUIVisualTest.cs prüft die visuelle und funktionale Konsistenz des Lobby-Menüs in FortressForge. Ziel ist es, sicherzustellen, dass alle UI-Elemente der Lobby-Szene korrekt initialisiert, sichtbar und bedienbar sind. Dazu gehören die Überprüfung von Texteingabefeldern (z. B. Spielername, IP-Join), Slider für Spieleranzahl, Tab-View-Wechsel und Buttons.

Die Tests nutzen das Unity Test Framework, um asynchron zu prüfen, ob die UI-Elemente vorhanden sind, ihre Werte korrekt übernehmen und ob die Sichtbarkeit der verschiedenen Tabs korrekt zwischen „Create“, „Join“ und „IP-Join“ umgeschaltet wird. So wird verifiziert, dass das UI auf Benutzereingaben und Statusänderungen richtig reagiert.

Besonderer Fokus liegt auf der Prüfung von:

- Sichtbarkeit und Aktivierung der UI-Komponenten

- Korrekte Werteinstellungen bei Eingabefeldern und Slidern, inklusive Randwerten

- Funktionalität der Tab-Ansicht mit korrektem Sichtbarkeitsstatus der einzelnen Tabs

Durch diese umfassenden Tests wird sichergestellt, dass die Lobby-Oberfläche stabil, intuitiv bedienbar und visuell konsistent bleibt eine essentielle Grundlage für ein positives Spielerlebnis beim Einstieg in Mehrspielerpartien.

* **PlayerClientTest.cs**  
  Diese Testklasse überprüft die Funktionalität des PlayerClient-Konstruktors in verschiedenen Varianten. Ziel ist es sicherzustellen, dass Standardwerte korrekt gesetzt werden und die Eigenschaften wie PlayerName, PlayerID und IsHost zuverlässig initialisiert werden.

Es werden Fälle getestet, in denen der PlayerClient leer, mit vollem Satz an Parametern oder mit nur Teilparametern erzeugt wird. Dabei werden sowohl Default-Werte als auch spezifische Werte geprüft, um Inkonsistenzen bei der Spieler-Initialisierung auszuschliessen.

Diese Tests garantieren, dass die PlayerClient-Objekte konsistent erstellt werden, was wichtig für das Netzwerk- und Spielmanagement ist.

* **RemoteConnectionHandlerTests.cs**

Diese Testklasse prüft die Netzwerkanbindung im FortressForge-Projekt, speziell das Verhalten beim Verlassen eines Servers. Im Setup wird eine leere Szene geladen und ein NetworkManager-Prefab instanziiert, um eine Testumgebung zu schaffen.

Der zentrale Test LeaveServerTest startet Server und Client, erstellt einen RemoteConnectionHandler und ruft die Methode LeaveServer auf. Anschliessend wird geprüft, ob Server und Client korrekt beendet wurden, die Szene gewechselt wurde und keine NetworkManager-Klone zurückbleiben.

Diese Tests sind wichtig, um sicherzustellen, dass beim Verlassen eines Spiels alle Netzwerkressourcen korrekt freigegeben werden und keine unerwünschten Objekte zurückbleiben, was für die Stabilität und Performance des Spiels essenziell ist.