

Konzept Test Orchestrator

Spezifikation

Version 1.0

Author: Vasil Denkov

Status: Proposed

Version: 1.0

This document is generated by Enterprise Architect. Do not edit!

1 Einführung

Die steigende Komplexität der Produktionsumgebungen und Anlagenkomponenten führt zusammen mit der wachsenden Anzahl an Anforderungen im Kontext der Wandelbarkeit zu einer neuen Ära der Qualitätssicherung im Bereich der industriellen Produktion. Laut Weyrich et. al. wird diese Komplexitätssteigerung durch

- die Heterogenität der Komponenten,
- die Wechselwirkungen innerhalb der Systeme,
- die Systemübergreifende Kommunikation,
- die Skalierung der Systeme

und andere Faktoren manifestiert. Während das RAMI 4.0 Modell zu einer Komplexitätsreduktion führt und die wichtigsten Aspekte der Industrie 4.0 Vision in der Form eine Schichtenarchitektur veranschaulicht, beschreibt es keine Methoden zur Validierung der korrekten Funktionsweise der bestehenden Anlagenressourcen. Durch seine Rolle als standardisierte und zentrale Auskunftsstelle eines Assets, bietet die Nutzung der Verwaltungsschale die Implementierung eines vendorübergreifenden Testprozesses. In Zusammenhang mit den Methoden des Softwaretestens, der virtuellen Inbetriebnahme und der Standardisierung von Submodellen ergeben sich viele Synergieeffekte.

Einerseits würde die Entwicklung einer standardisierten VWS für die virtuelle Inbetriebnahme das konkrete Simulationswerkzeug abstrahieren und eine lose Kopplung von Simulationsumgebungen an einem Testprozess ermöglichen.

Des Weiteren ist die Integration von ausführbaren Testskripten als Teil eines vom Testobjekt bereitgestellten Submodells auch eine Möglichkeit bei der Umsetzung einer Lösung zur automatisierten Testung von Anlagenkomponenten.

Auf der anderen Seite kann die VWS einer Anlagenkomponente, also eines Testobjekts, durch die Integration eines weiteren standardisierten Submodells Testkonfigurationen beinhalten bzw. bereitstellen, welche einen Testprozess definieren. Eine weitere Möglichkeit für die Persistenz der Testkonfigurationen bietet die Realisierung eines Ticket-Systems, welches Testkonfigurationen in einer Datenbank verwaltet. Durch die Implementierung einer Softwarekomponente, also eines Test Orchestrators, können die Testkonfigurationen ausgelesen und die entsprechende Testschritte durchgeführt werden.

Im Rahmen des Projekts OpenBaSys wurde eine Softwarearchitektur und -infrastruktur entwickelt, die als Basis für die Realisierung einer Industrie 4.0 konformen Testlösung dienen kann.

2 Ziel des Systems

Wie bereits erläutert wurde, dient die Verwaltungsschale durch die Implementierung eines Metamodells als Abstraktionsmittel bzw. "Single Point of Information" eines oder mehrerer Assets [Zie22]. Eine Verwaltungsschale lässt sich also als allgemeingültige Schnittstelle jeder Industrie 4.0 konformen Anlagenkomponente bezeichnen und stellt somit die digitale Basis für die Standardisierung eines vendorübergreifenden Testprozesses dar. Der Testprozess verfolgt folgende Ziele:

- Verifikation und Validierung von Komponenten und Anlagen nach veränderten Rahmenbedingungen

- Qualitätssicherung im operativen Betrieb
- Zertifizierung von Verwaltungsschalen, Submodellen und Funktionalitäten zum Zweck der Wiederverwendbarkeit
- Evaluierung von Anlagenkomponenten auf Basis spezifischer Anwendungsszenarien
- Protokollierung von Komponentenverhalten zum Zweck des Soll-/Ist Abgleichs

Ziel des ersten Schritts in der Realisierung einer Testautomatisierungslösung soll das Design und die Implementierung eines Test-Orchestrators (nachfolgend System genannt) auf Basis des BaSyx .NET SDK sein. Die Vorgehensweise charakterisiert sich durch drei Phasen.

- Spezifikation der Anforderungen für den Test-Orchestrator nach Analyse der bestehenden BaSyx Infrastruktur und unter Berücksichtigung aller Funktionalitäten und Komponenten
- Erstellung einer Architektur auf Basis der Anforderungsspezifikation
- Implementierung und Inbetriebnahme des Orchestrators durch den Einsatz geeigneter Testobjekte

Im Rahmen des praktischen Anteils der Abschlussarbeit muss also ein Demonstrator synthetisiert und implementiert werden. Dieser muss eine Testung von Anlagenkomponenten bzw. Prozessen ermöglichen. Der Test-Orchestrator muss Kommunikationsbeziehungen mit registrierten Verwaltungsschalen aufbauen und einen Test-Pipeline für jedes Testobjekt durchführen. Voraussetzungen dafür sind, dass das System aktive Verwaltungsschalen erkennen, adressieren und ggf. parametrieren muss, die strukturelle Architektur der Testobjekte prüfen kann und in der Lage ist, Verhaltentests auf Basis bestehender Technologien durchzuführen. Die Funktionalität des zu entwickelnden Systems ist entsprechend nachzuweisen. Hierfür werden Ressourcen in Form von Verwaltungsschalen und SPS-Programme verwendet.

3 Anwendungsszenarien

4 Allgemeine Anwendungsszenarien

4.1 Allgemeine Anwendungsszenarien

Die folgende Abbildung liefert einen groben Überblick über die Funktionalitäten des Systems. Aus dem Use-Case Diagramm geht hervor, dass die Testlösung zwei Testdurchführungsstrategien unterstützt und zum Zweck der Auswertung Testberichte generiert.

Das Modell wird in diesem Abschnitt oberflächlich erläutert, um das allgemeine Konzept zu veranschaulichen. Im folgenden Abschnitt erfolgt eine Erweiterung und detaillierte Beschreibung der primären Anwendungsszenarien.

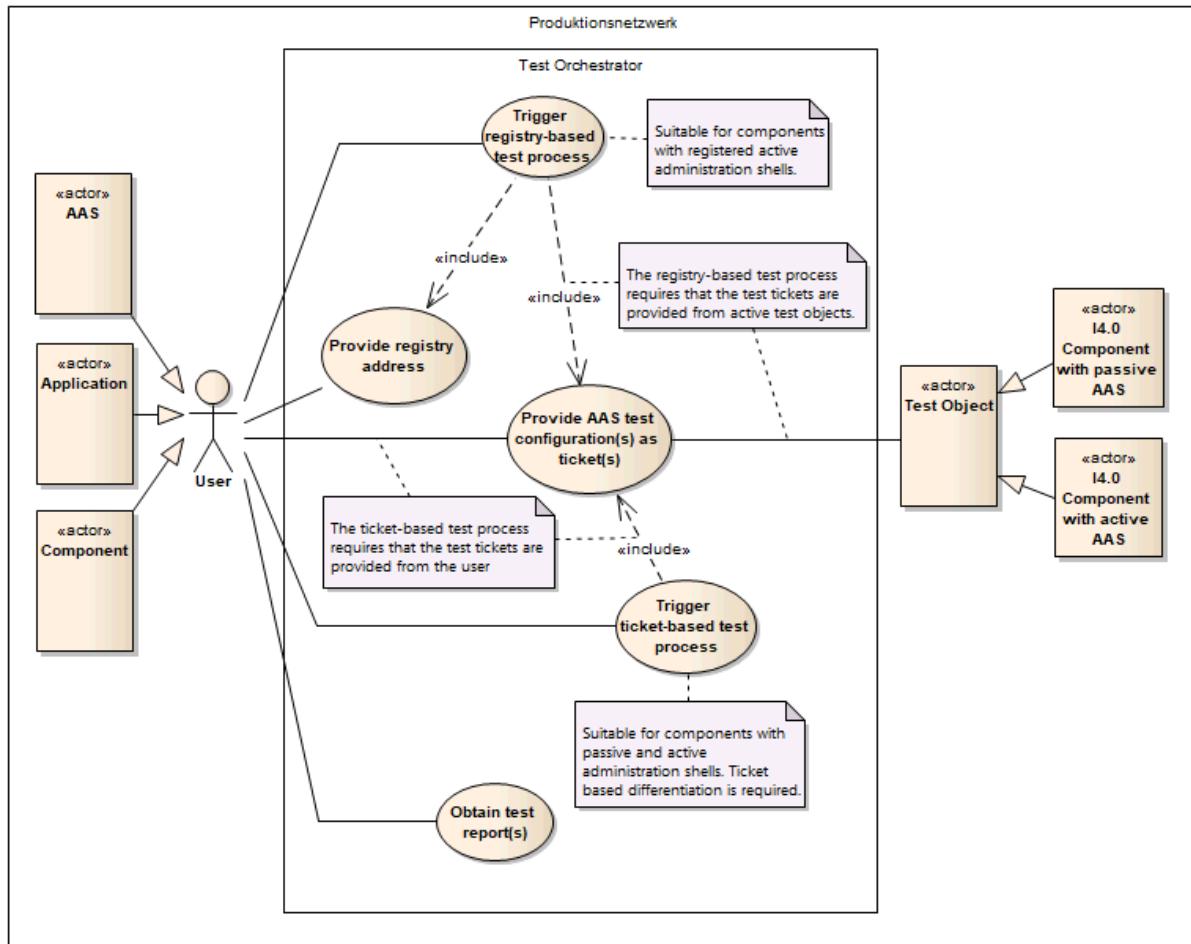


Abbildung 1: Allgemeine Anwendungsszenarien

User

Der Anwender des Systems kann sowohl einen Integrator als auch eine andere im gleichen Netzwerk integrierte Komponente sein. Im Rahmen des Diagramms werden Typ 2 bzw. Typ 3 Verwaltungsschalen, Software-Applikationen und Anlagenkomponenten wie z.B. Edge-Gateways als potenzielle Nutzer der Testlösung angesehen. Jede Entität, die mit einer Verwaltungsschale über die entsprechende Schnittstelle interagieren kann, gilt als potenzieller Anwender des Test Orchestrators. Als Schnittstelle zur Interaktion dient die [BaSyx AAS HTTP REST-API](#) und die [BaSyx Web UI](#).

Test Object

Als Testobjekt gilt jede Industrie 4.0 Komponente, die durch eine passive oder aktive VWS repräsentiert wird. Zum Zweck der Kommunikation mit dem Test-Orchestrator muss die aktive VWS die [BaSyx AAS HTTP REST-API](#) implementieren.

Trigger registry-based test process

Die erste Testdurchführungsstrategie basiert auf die Verfügbarkeit einer Registry Komponente und erfordert die Existenz bestimmter Testkonfigurationen (Tickets) als Teil eines vordefinierten Submodells innerhalb der Verwaltungsschale des Testobjekts.

Provide registry address

Da die BaSys-Registrierungskomponente durch eine aktive Verwaltungsschale repräsentiert wird, ist zum Zweck der Kommunikation zwischen dem Orchestrator und der Registry die Angabe der

entsprechenden Adresse erforderlich.

Trigger ticket-based test process

Im Rahmen des zweiten Szenario werden die Tickets vom Anwender des Systems bereitgestellt. Darauf basierend erfolgt die Verifikation aller durch Tickets beschriebenen Testobjekte.

Provide AAS test configuration(s) as ticket(s)

In Abhängigkeit von der Testdurchführungsstrategie müssen die Testkonfigurationen entweder vom Anwender oder vom Testobjekt bereitgestellt werden.

Obtain test report(s)

Die Testergebnisse werden in Form von Reports angefordert.

5 Primäre Anwendungsszenarien

5.1 Kernfunktionalitäten

Dieser Abschnitt beschreibt die Kernfunktionalitäten des Systems aus Sicht des Anwenders und soll einen Eindruck der gewünschten Funktionen des Systems liefern. Die folgende Abbildung stellt die Hauptfunktionalitäten des Systems dar. Die einzelnen Elemente der Abbildung sind nachfolgend detailliert erläutert.

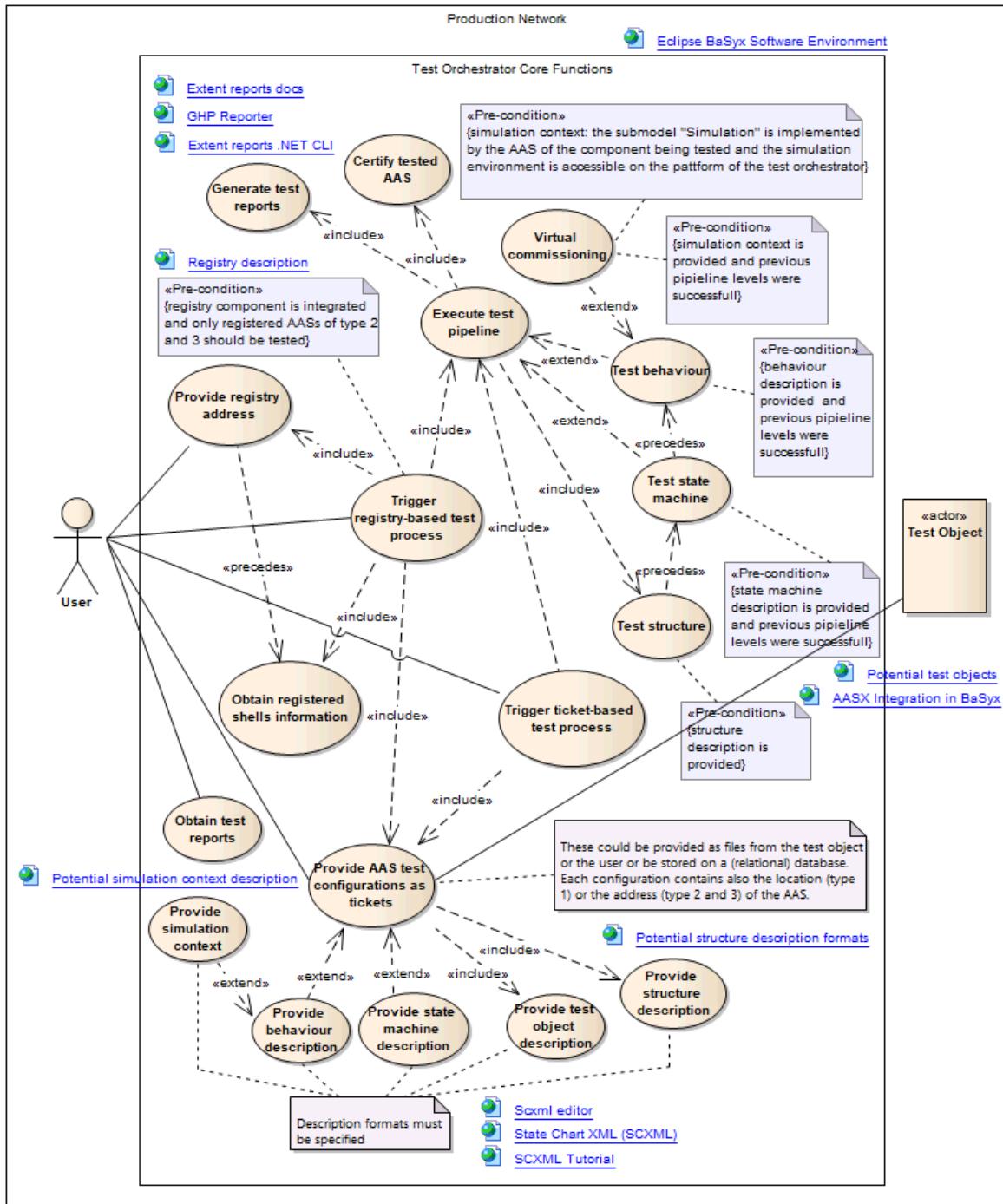


Abbildung 2: Kernfunktionalitäten

Test Object

Als Testobjekt gilt jede Industrie 4.0 Komponente, die durch eine passive oder aktive VWS repräsentiert wird. Zum Zweck der Kommunikation mit dem Test-Orchestrator muss die aktive VWS die BaSyx Asset Administration Shell HTTP [REST-API](#) implementieren.

User

Der Anwender des Systems kann sowohl einen Integrator als auch eine andere im gleichen Netzwerk

integrierte Komponente sein. Im Rahmen des Diagramms werden Typ 2 bzw. Typ 3 Verwaltungsschalen, Software-Applikationen und Anlagenkomponenten wie z.B. Edge-Gateways als potenzielle Nutzer der Testlösung angesehen. Jede Entität, die mit einer Verwaltungsschale über die entsprechende Schnittstelle interagieren kann, gilt als potenzieller Anwender des Test Orchestrators. Als Schnittstelle zur Interaktion dient eine REST-API und eine Web UI.

Trigger registry-based test process

Als Ausgangspunkt für die Testdurchführung und zum Zweck des Aufbaus der Kommunikationsbeziehung zwischen dem Orchestrator und potenziellen Testobjekten kann eine [AAS Registry](#) Komponente berücksichtigt werden. Voraussetzung für diese Art der Testdurchführung ist die Integration der Registrierungskomponente im Produktionsnetzwerk. Laut BaSys 4.0 beinhaltet die Registry Referenzen zu jeder registrierten und somit für andere Entitäten verfügbaren VWS. Wird eine Kommunikationsbeziehung zwischen dem Orchestrator und der Registry aufgebaut, so kann diese als Auskunftsstelle für aktive Testobjekte verwendet werden. Solcher Testprozess ist somit nur für Komponenten geeignet, die durch aktive Verwaltungsschalen repräsentiert sind. Passive Vertreter können im Rahmen eines solchen Testprozesses nicht verifiziert werden, da diese über keine Kommunikationsschnittstelle verfügen. Im Rahmen der Testung werden die registrierten Verwaltungsschalen und die durch diese repräsentierten Komponenten nacheinander sequenziell, auf Basis einer Testsequenz (nachfolgend Test-Pipeline genannt), verifiziert.

WICHTIG: Die Bereitstellung von Testkonfigurationen für jede registrierte VWS erfordert vom Anwender Kenntnis über die Anzahl der Registrierungen und spezifisches Wissen für jedes Testobjekt. Die Anzahl der registrierten Verwaltungsschalen ist für den Anwender nicht immer bekannt und das System muss von Anwendern ohne technisches Wissen nutzbar sein. Eine Lösung der Problematik bietet die Integration der Testkonfiguration innerhalb der Verwaltungsschale des Testobjekts an. Die Testkonfiguration (das Test-Ticket) muss vom Komponentenlieferanten bereitgestellt werden. Soll diese im Rahmen des Lebenszyklus der Komponente nicht manipuliert werden können, sind die entsprechende Schutz- bzw. Security-Mechanismen erforderlich.

Provide registry address

Als Voraussetzung für die Durchführung eines Testprozesses für alle in einer Registry angemeldeten Verwaltungsschalen, muss das System dem Anwender die Möglichkeit bieten die URL-Adresse der Registry Komponente bereitzustellen. Die Bereitstellung kann in Form einer Datei, als Argument beim Applikationsstart oder über die UI erfolgen.

Obtain registered shells information

Im Rahmen der registry-orientierten Testdurchführung muss der Orchestrator Information zu jeder registrierten VWS bereitstellen können. Zum Zweck der Verifikation kann die VWS des Testobjekts lokal im Speicher vorliegen.

Provide AAS test configurations as tickets

Das System muss dem Anwender die Möglichkeit bieten Testkonfigurationen für jedes Testobjekt bereitzustellen. Jede Testkonfiguration gilt als Grundlage für die Testdurchführung. Auf Basis der bereitgestellten Testkonfiguration muss der Orchestrator die vorzunehmende Testschritte ableiten und ausführen.

WICHTIG: Bei dem registry-orientierten Testprozess sind die Testkonfigurationen von den Testobjekten zu erwarten.

Provide test object description

Die Beschreibung des Testobjekts ermöglicht eine Unterscheidung zwischen passiven und aktiven Verwaltungsschalen. Auf Basis dieser information findet die Konstruktion des Testobjekts im Kontext des Softwaresystems.

Provide structure description

Unabhängig davon von welchem Typ die VWS des Testobjekts ist, muss der Anwender oder das Testobjekt dem Orchestrator eine Beschreibung bereitstellen, welche die Soll-Struktur der VWS in einer maschineninterpretierbaren Form spezifiziert. Anhand dieser Information können Strukturtests durchgeführt werden.

Unternehmen und Organisationen, derer Verwaltungsschalen interne und strukturelle Richtlinien einhalten müssen, können durch die Spezifikation einer allgemeinen Strukturbeschreibung verifiziert werden.

Provide state machine description

Repräsentiert das Testobjekt eine Zustandsmaschine, so ist die Verifikation ihrer Funktionsweise notwendig. Die Testung erfolgt durch die Initiierung von Transitionen und die Überprüfung, ob der Zustand nach der Transition mit den Anforderungen korrespondiert. Da von jeder VWS ein unterschiedliche Zustandsmaschine implementiert werden kann, ist auch hier eine Testkonfiguration erforderlich, welche das Verhalten des Automats beschreibt. Auf Basis der Inhalte dieser Beschreibung werden die Zustandstests durchgeführt. Aus diesem Grund muss die Testkonfiguration vertrauenswürdig sein. Diese Verifikation ist bei aktiven Verwaltungsschalen, die Führungskomponenten repräsentieren, durchzuführen.

Provide behaviour description

Handelt es sich um eine aktive VWS als Testobjekt, so kann zur Testkonfiguration eine Verhaltensspezifikation hinzugefügt werden, die als Grundlage für die Durchführung von Unit-Tests dient. Da z. Zt. die Implementierung von Fähigkeiten seitens der VWS nicht vollständig spezifiziert ist, werden im Rahmen des Dokuments als Verhalten die Operationen angesehen, die von einer VWS über die Submodelle angeboten sind. Zum Zweck der Verifikation dieser Operationen muss die Verhaltensspezifikation die Bezeichnung der Operation, die Testdaten und die erwarteten Zustände definieren.

Provide simulation context

Eine vollständige Testlösung soll zum Zweck der Verifikation eines von Typ 2 oder Typ 3 VWS repräsentierten Testobjekts als Teil einer Produktionsumgebung die automatisierte Durchführung einer virtuellen Inbetriebnahme ermöglichen. Hierfür benötigt den Orchestrator einen Simulationskontext. Dieser kann entweder vom Anwender als Datei oder vom Testobjekt als Submodell bereitgestellt werden. Als Vorlage kann die Struktur des Submodells [Simulation](#) in Betracht gezogen werden. Die Durchführung einer VIBN ist von der Simulationsumgebung und den angebotenen Schnittstellen stark abhängig. Hierfür wäre die Festlegung und Standardisierung einer (REST) API sinnvoll, welche als Abstraktionsschicht zwischen der konkreten Simulationsumgebung und seinen Nutzern dient. Diese Abstraktion würde die Unabhängigkeit jeder Testlösung von der einzusetzenden Simulationsumgebung garantieren und kann in Form einer Verwaltungsschale umgesetzt werden.

Trigger ticket-based test process

Der ticket-basierte Testprozess ist im Gegensatz zur registry-orientierten Testung für alle Arten von Verwaltungsschalen geeignet. Es ist eine Frage der Spezifikation, ob ein Ticket als Eintrag in einem relationalen Datenbank oder als Datei der Testlösung zur Verfügung gestellt wird. Für die erste Variante ist eine Schnittstelle erforderlich damit eine Verwendung von spezifischen Programmiersprachen seitens des Nutzers ausgeschlossen werden kann. Eine weitere Möglichkeit besteht in der direkten Integration der Datenbank im Kontext der Softwarekomponente *Test Orchestrator*. Der Zugriff zur Datenbank kann dann über die Operationen eines geeigneten Submodells stattfinden.

Als Eingangsinformationen für diesen Testprozess gelten Tickets, welche sowohl die

Testkonfiguration als auch den Zugriffspunkt des Testobjekts beschreiben. Für jede passive VWS wird im Ticket entweder den Pfad zur AASX-Datei oder die Datei selbst angegeben. Im Ticket jeder aktiven VWS muss die URL des Testobjekts zur Verfügung stehen.

Execute test pipeline

Im Rahmen des Testprozesses muss das System eine Test-Pipeline durchführen. Im Laufe der Pipeline sind die entsprechenden Teststufen auf Basis der vom Anwender bereitgestellten Testkonfigurationen vom Orchestrator zu identifizieren und durchzuführen.

Test structure

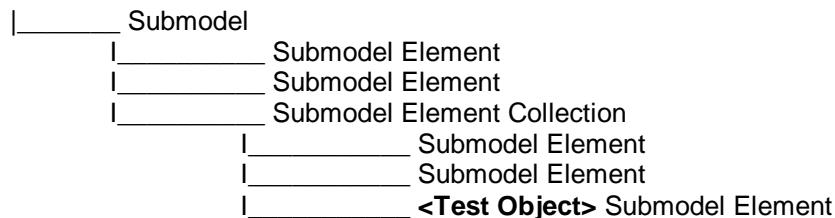
Auf Basis der vom Anwender bereitgestellten Strukturbeschreibung muss das System die Struktur der VWS der zu testenden Komponente verifizieren. Dieser Vorgang lässt sich darüber motivieren, dass die Struktur jeder VWS jederzeit manipulierbar ist. Als Strukturinformation dient sowohl die allgemeine Struktur der VWS in Hinsicht auf ihre Submodelle als auch die Anordnung der einzelnen Elemente jedes Teilmodells. Jede Informationseinheit, unabhängig davon, ob es sich um ein Submodell, Property oder eine Operation handelt, gilt im Kontext der VWS als Strukturinformation, derer Existenz überprüft werden muss.

Die potenzielle Verifikation eines VWS- oder Submodell-Elements, z.B. ein Submodell oder eine Operation, erfordert das Zugreifen bzw. die Existenz der übergeordneten Informationseinheit. Ist diese nicht vorhanden, so ist das Testen des Elements ausgeschlossen.

Beispiel:

Die folgende Abbildung veranschaulicht die vereinfachte Struktur einer Verwaltungsschale. Es ist eine einfache VWS mit einem Submodell und drei Eigenschaften zu sehen, wobei die dritte Eigenschaft einer Sammlung aus Submodell-Elementen entspricht und drei Properties beinhaltet.

Asset Administration Shell



Zum Zweck des Beispiels gilt das letzte Element als Verhalten im Kontext der Verwaltungsschale, z.B. eine Operation. Damit diese überhaupt verifiziert und zugegriffen werden kann sind

- die Existenz des Submodells
- die Existenz der Sammlung aus Submodell-Elementen

sicherzustellen. Erst nach der Verifikation, dass diese innerhalb der VWS-Struktur enthalten sind, kann die Existenz des eigentlichen Testobjekts überprüft werden.

Erst nach dieser Überprüfung wird die Operation als Gegenstand einer weiteren Stufe der Test-Pipeline angesehen.

Test state machine

Ist für das Testobjekt eine Testkonfiguration bereitgestellt worden, die eine Zustandsmaschine beschreibt, so muss ihre richtige Funktionsweise über die Verwaltungsschale entsprechend nachgewiesen werden. Voraussetzung dafür ist, dass

- die erfolgreiche bidirektionale Kommunikation zwischen dem Asset und der VWS sichergestellt
- die Steuerung des Automats über die Verwaltungsschale möglich und
- den aktuellen Zustand der Führungskomponente als Eigenschaft innerhalb der VWS enthalten ist
 - also vom Orchestrator ausgelesen werden kann.

Die Konfigurationsdatei kann die Identifikatoren der Eigenschaften spezifizieren damit es der

Testlösung bekannt ist durch welches Property den Zustand der Führungskomponente ausgelesen bzw. gesetzt werden kann.

Im Rahmen der Testung initiiert den Orchestrator Transitionen. Nach jeder Transition wird der aktuelle Zustand des Testobjekts mit dem erwarteten Zustand verglichen.

Test behaviour

Das über die VWS von dem Asset zur Verfügung gestellten Verhalten ist vom Orchestrator auf Basis der entsprechenden Beschreibung zu verifizieren. Da Unterschiedliche Möglichkeiten zur Darstellung von Verhalten im Kontext der Verwaltungsschale existieren (Fähigkeiten, Services, Operationen usw.) und es

- keine maschinelle Interpretation der Fähigkeiten auf Grund der unterschiedlichen Verwendung der Begrifflichkeit gegeben ist,
- keine beispielhafte Implementierung von Fähigkeiten zur Verfügung steht,
- keine eindeutige Vorgehensweise wie die Darstellung von Services und Fähigkeiten im Kontext einer VWS von einer Testlösung zu identifizieren ist gibt,

werden im Rahmen der Verhaltenstests nur die von den Submodellen bereitgestellten Operationen auf Grundlage der Testkonfiguration mittels Unit-Tests überprüft.

Virtual commissioning

Wird vom Anwender oder dem Testobjekt einen Simulationskontext bereitgestellt, so ist die Durchführung einer virtuellen Inbetriebnahme notwendig. Voraussetzung für diese ist die Verfügbarkeit der im Simulationskontext angegebenen Simulationsumgebung und die Ausführung der spezifizierten Simulationsdatei.

Certify tested AAS

Jede getestete VWS ist vom Orchestrator entsprechend zu gekennzeichnen.

Generate test reports

Im Verlauf der Testdurchführung muss der Orchestrator Test-Reports generieren.

Obtain test reports

Das System muss in der Lage sein, die im Laufe der Testdurchführung generierten Test-Reports dem Anwender nach Aufforderung bereitstellen zu können.

6 Anforderungen

Dieser Abschnitt beschreibt die funktionalen Anforderungen an das zu entwickelnde System. Unabhängig davon muss der Test Orchestrator jederzeit mit weiteren Funktionalitäten ausgestattet werden können.

7 Annahmen und Abhängigkeiten

Potenzielle Softwarebibliotheken für den Test Orchestrator:

Eclipse BaSyx SDK: <https://github.com/eclipse-basyx/basyx-dotnet-sdk>

Eclipse BaSyx Components: <https://github.com/eclipse-basyx/basyx-dotnet-components>

Entity Framework Core: <https://github.com/dotnet/efcore>

Auto Mapper: <https://github.com/AutoMapper/AutoMapper>

Json Serialization/Deserialization: <https://github.com/JamesNK/Newtonsoft.Json>

Test-Reports: <https://github.com/extent-framework/extentreports-csharp>

Test-Frameworks: <https://github.com/nunit/nunit>, <https://github.com/nunit/nunitlite>

Externe Softwarekomponenten und Werkzeuge:

BaSyx

<https://github.com/eclipse-basyx/basyx-dotnet-applications/tree/main/BaSyz.Registry.Server.Http.App>

Potenzielle passive Testobjekte: <https://admin-shell-io.com/samples/>

AAS

https://gitlab.rz.htw-berlin.de/OpenBasys/demonstator-rebuild/-/tree/main/src/Generator_Rebuild

AAS Schemas: <https://github.com/admin-shell-io/aas-specs/tree/master/schemas>

State Machine Descriptor Schema: <https://www.w3.org/TR/scxml/>

Simulationskontext:

<https://github.com/admin-shell-io/submodel-templates/tree/main/development/Simulation/1/0>

Asset Interface

Beschreibung:

<https://github.com/admin-shell-io/submodel-templates/tree/main/development/Asset%20Interface%20Description/1/0>

Software

Nameplate:

<https://github.com/admin-shell-io/submodel-templates/tree/main/development/Software%20Nameplate/1/0>

AAS Package Explorer: <https://github.com/admin-shell-io/aasx-package-explorer>

Dokumentation:

Modellierung: <https://sparxsystems.com/products/ea/10/>

7.1 Annahmen und Abhängigkeiten

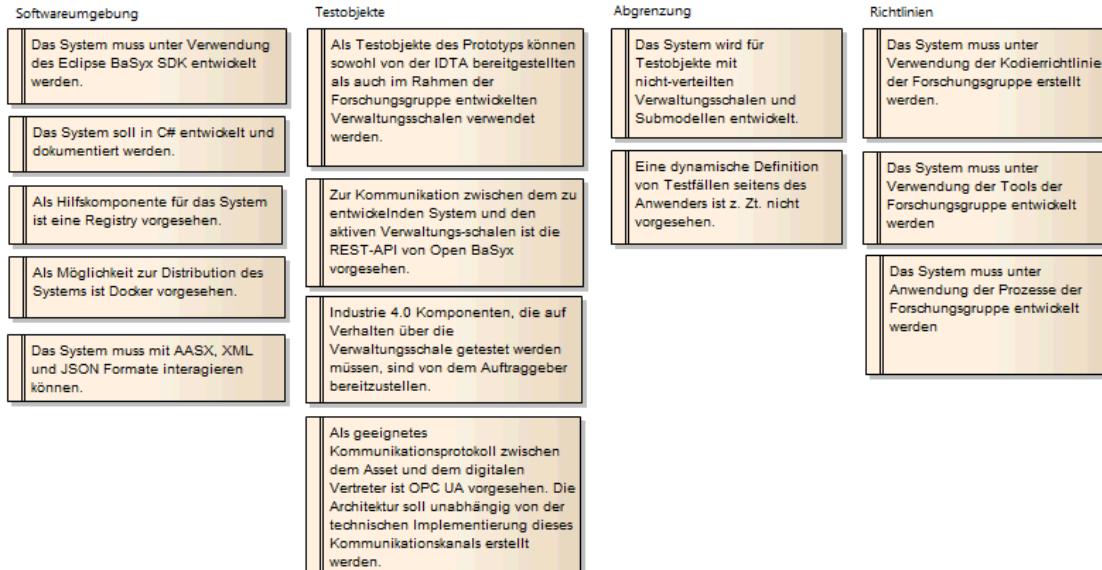


Abbildung 3: Annahmen und Abhängigkeiten

8 Kontext

Dieser Abschnitt gibt einen groben Einblick des Orchestrators im Kontext seiner Umgebung an und dient als Information für die Ableitung der in den folgenden Abschnitten definierten Anforderungen.

8.1 Registry-orientierter Testprozesskontext

Die folgende Abbildung stellt die Kommunikationsbeziehungen zwischen den Komponenten im Kontext eines registry-orientierten Testprozesses dar.

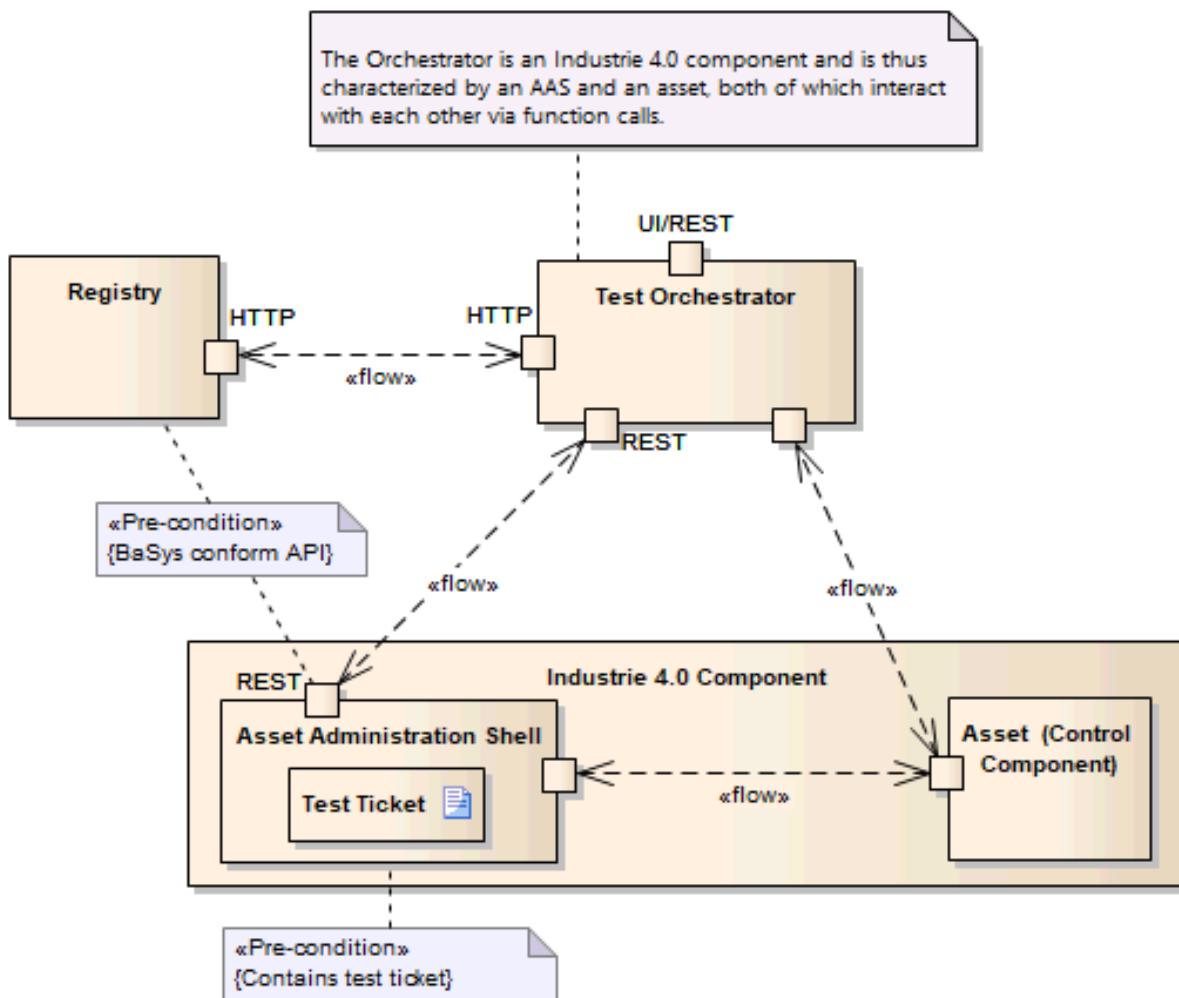


Abbildung 4: Registry-orientierter Testprozesskontext

Asset (Control Component)

Die Steuerungskomponente ist ein Bestandteil der I4.0 Komponente und kommuniziert mit der Verwaltungsschale über eine geeignete Schnittstelle.

Asset Administration Shell

Die Testung der Anlagekomponente erfolgt auf Basis der Verwaltungsschale. Hierfür gilt die entsprechende REST API als Basis. Die Testkonfigurationen müssen in einer geeigneten Form in der VWS enthalten sein.

Industrie 4.0 Component

Als Testobjekt gilt jede Steuerungskomponente, die von einer Typ 1, Typ 2 oder Typ 3 Verwaltungsschale repräsentiert wird. Es ist für den Orchestrator ohne Bedeutung wie die Steuerungskomponente mit der Verwaltungsschale kommuniziert bzw. wie die Daten innerhalb der VWS von der Steuerungskomponente aktualisiert oder ausgelesen werden. Die Schnittstelle der Industrie 4.0 Komponente entspricht der Schnittstelle der Verwaltungsschale.

Registry

Die Registry Komponente entspricht einem HTTP Server, der vom Orchestrator zum Zweck der Erfassung der Adressen aller registrierten Testobjekte verwendet wird, siehe [BaSyx](#).

Test Orchestrator

Im Rahmen einer registry-basierten Testung baut den Orchestrator einen Kommunikationskanal zur Registry auf. Auf Basis der erfassten Informationen wird eine Verbindung zum Testobjekt realisiert. Dazu gehört sowohl den Aufbau einer Kommunikationsbeziehung zur Verwaltungsschale als auch die direkte Verbindung der Testlösung zum Asset. Auf diese Weise lässt sich die richtige Synchronisation der beiden Bestandteile der Industrie 4.0 Komponente nachweisen. Während der Orchestrator mit der Registry und der VWS über bereitgestellte APIs kommuniziert, ist den Informationsaustausch mit dem Asset vom eingesetzten Kommunikationsprotokoll abhängig. Eine Beschreibung des Asset-Endpoints wäre für diese Kommunikationsbeziehung ebenfalls notwendig. Hierfür wäre die Standardisierung eines Submodells sinnvoll, die diese Informationen enthält. Als Basis könnte das von der IDTA geplante Asset Interface Description [Submodell](#) dienen. Eine weitere Möglichkeit wäre die Standardisierung von OPC UA als Kommunikationsprotokoll zwischen der AAS und dem Asset.

Test Ticket

Das Ticket beinhaltet alle Daten, die zum Zweck der Durchführung der Testpipeline notwendig sind.

Hierzu zählen

- Allgemeine Beschreibung des Testobjekts (obligatorisch)
- Strukturbeschreibung (obligatorisch)
- Beschreibung der Schnittstelle des Assets (optional)
- Beschreibung des Zustandsautomats (optional)
- Beschreibung des Verhaltens (optional)
- Simulationskontext (optional)

8.2 Ablauf Registry-orientierte Testdurchführung

Das folgende Sequenzdiagramm beschreibt den prinzipiellen Ablauf einer Registry-orientierten Testdurchführung und veranschaulicht die allgemeine Kommunikation zwischen den Komponenten. Die Abbildung stellt keine vollständige oder optimale Implementierung sondern einen Testablauf auf konzeptionelle Ebene dar.

Wie bereits erwähnt wurde, gilt als Voraussetzung für den Testprozess der Kommunikationsaufbau zwischen den Komponenten. Zum Zweck der dynamischen Spezifikation der Auskunftsstelle für Testobjekte, also der Registry, wird dem Anwender die Möglichkeit angeboten die Adresse der Registrierungskomponente dem System bereitzustellen und damit diesen Kommunikationsaufbau zu initiieren.

Während die erfolgreiche Verbindung zur Registry die erste Voraussetzung für diesen Testprozess ist, entspricht die Existenz des Test-Tickets innerhalb der VWS des Testobjekts einem weiteren kritischen Punkt.

Nach dem Kommunikationsaufbau werden die Adressen der registrierten Verwaltungsschalen vom Orchestrator abgefragt. Zum Zweck der Kommunikation mit dem Testobjekt wird für jede Registrierung einen HTTP Client instanziert. Über die HTTP Verbindung erfolgt die Bereitstellung des entsprechenden Test-Tickets. Jedes Ticket wird im Kontext der Softwareapplikation als Instanz der Klasse *Test Ticket* integriert. Dies gilt als Voraussetzung für den weiteren Ablauf des Testprozesses.

Aus dem Ticket und der AAS Client-Instanz wird vom System ein Testobjekt konstruiert. Dieses dient als Argument der Funktion zur Durchführung der Test-Pipeline. Im Rahmen der Testsequenz werden

alle Tests entsprechend protokolliert. Nach der Testdurchführung erfolgt die Instanziierung des Test-Protokolls innerhalb der VWS des Testobjekts.

Sind im Rahmen der Pipeline keine Ungenauigkeiten identifiziert worden, so erfolgt die entsprechende Kennzeichnung der Verwaltungsschale des Testobjekts.

Nach der Verifikation aller registrierten Verwaltungsschalen wird ein allgemeines Protokoll für alle Testobjekte erstellt und dem Anwender bereitgestellt.

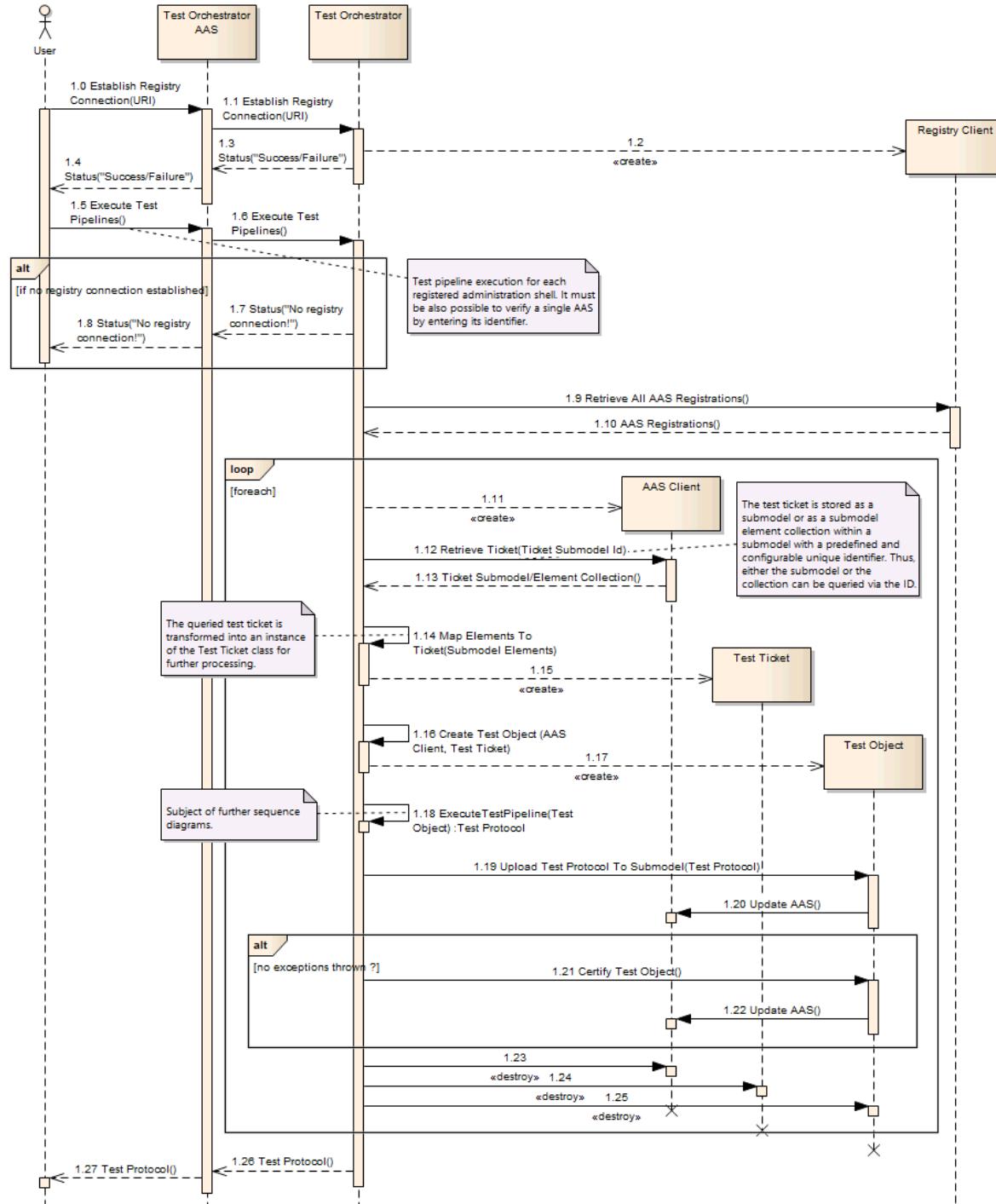


Abbildung 5: Ablauf Registry-orientierte Testdurchführung

8.3 Ticket-orientierter Testprozesskontext

Wie bereits erwähnt wurde, besteht die Idee der Ticket-orientierten Testdurchführung in der direkten Spezifikation und Zuordnung von Testkonfigurationen an bereits integrierten Industrie 4.0 Komponenten. Zum Zweck der Persistenz der Informationen kann eine Datenbank zum Einsatz kommen.

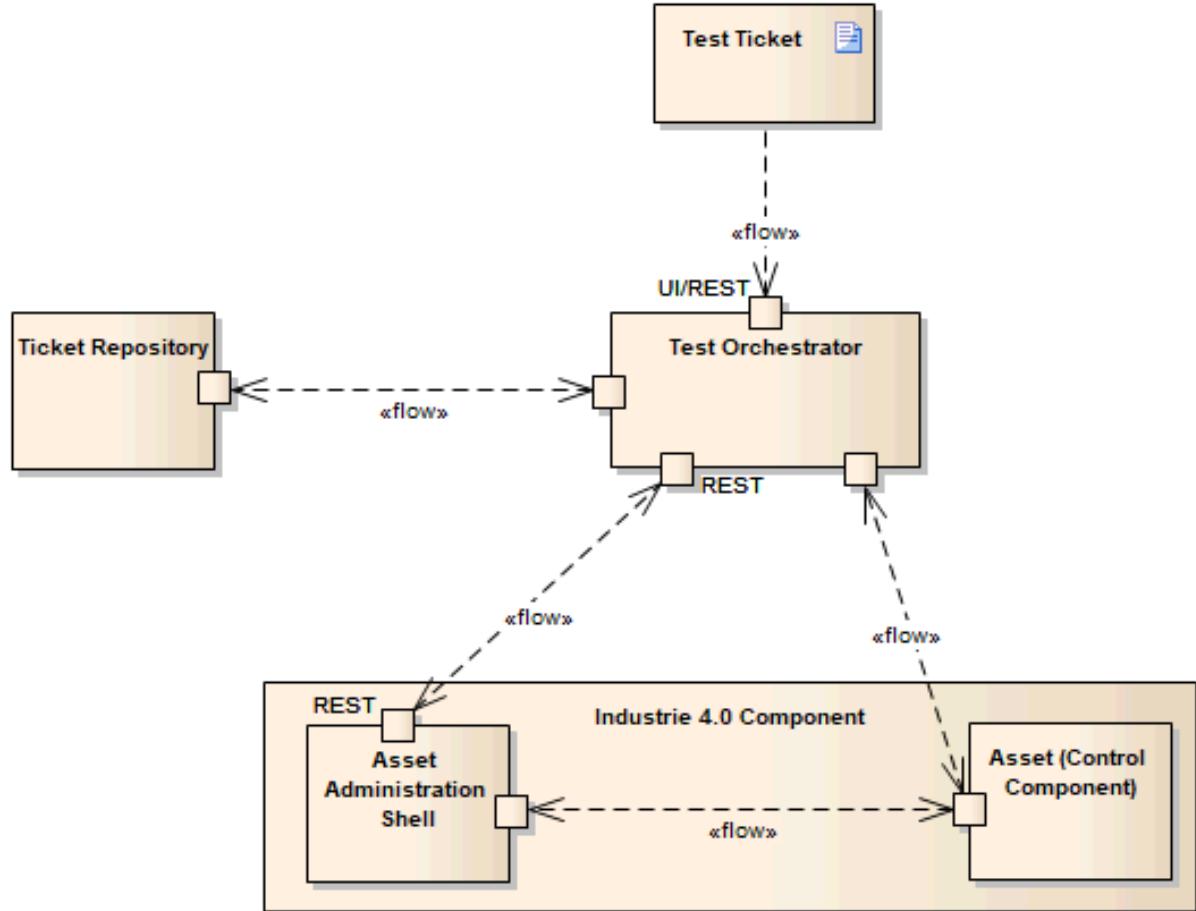


Abbildung 6: Ticket-orientierter Testprozesskontext

Test Orchestrator

Die Testkonfigurationen werden zusammen mit den Tickets dem Orchestrator über die entsprechende UI vom Anwender übergeben und innerhalb des Ticket Verzeichnisses persistiert. Im Rahmen der Testdurchführung extrahiert der Orchestrator die jedem Testobjekt zugeordneten Testkonfiguration und führt die Test-Pipeline durch.

Ticket Repository

Das Ticketverzeichnis dient als Auskunftsstelle für alle Testkonfigurationen. Der Anwender hat keinen Schreibzugriff zu dieser Komponente.

Test Ticket

Das Ticket beinhaltet alle Daten, die zum Zweck der Durchführung der Testpipeline notwendig sind.

Hierzu zählen

- Allgemeine Beschreibung des Testobjekts (obligatorisch)
- Strukturbeschreibung (obligatorisch)

- Beschreibung der Schnittstelle des Assets (optional)
- Beschreibung des Zustandsautomats (optional)
- Beschreibung des Verhaltens (optional)
- Simulationskontext (optional)

8.4 Ablauf Ticket-orientierte Testdurchführung

Das folgende Sequenzdiagramm beschreibt den prinzipiellen Ablauf einer Ticket-orientierten Testdurchführung und veranschaulicht die allgemeine Kommunikation zwischen den Komponenten. Die Abbildung stellt keine vollständige oder optimale Implementierung sondern einen Testablauf auf konzeptionelle Ebene dar.

Dem Anwender wird die Möglichkeit angeboten Test-Tickets in der Ticket Repository hochzuladen, zu aktualisieren und löschen. Im Rahmen des Testprozesses werden die Tickets aus dem Verzeichnis ausgelesen. Für jedes Ticket erfolgt die Instanziierung der Klasse *Test Ticket*.

Während alle Tickets, die für die Verifikation von Industrie 4.0 Komponenten mit aktiven Verwaltungsschalen eingesetzt werden, zum Zweck des Aufbaus der Kommunikationsbeziehung die URI der VWS beinhalten, wird mit den Tickets aller passiven Repräsentanten die VWS-Datei vom Anwender bereitgestellt.

Für jedes ausgelesene Ticket erfolgt eine Instanziierung der Klasse *Test Ticket*. Es wird also ein Mapping zwischen dem Ticket innerhalb des Verzeichnisses und das Ticket im Kontext der Softwareapplikation durchgeführt.

Die Testkonfiguration wird für die Konstruktion einer Instanz der Klasse *Test-Object* verwendet. Wird dem Konstruktor ein Ticket bereitgestellt, das eine URI beinhaltet, so wird eine Kommunikationsbeziehung zum AAS Client aufgebaut. Obwohl im Fall einer passiven VWS alle für die Testdurchführung notwendigen Informationen enthalten sind, erfordert die Schnittstelle des Moduls zum Ausführen der Test-Pipeline die Übergabe eines durch das Ticket erstellten Testobjekts.

Im nächsten Schritt wird die Pipeline durchgeführt. Es erfolgt eine Zuordnung des bei der Testung erstellten Test-Protokolls zum ausgelesenen Test-Ticket und eine darauffolgende Aktualisierung des Tickets im Verzeichnis.

Sind keine Abweichungen zwischen den Testergebnissen und den Ticket-Deskriptoren im Rahmen der Pipeline identifiziert worden, so erfolgt eine Kennzeichnung des Testobjekts als erfolgreich getestet.

Nach der Durchführung der Sequenz für jedes Testobjekt wird ein gemeinsames und allgemeines Testprotokoll für alle Testobjekte erstellt und dem Anwender bereitgestellt.

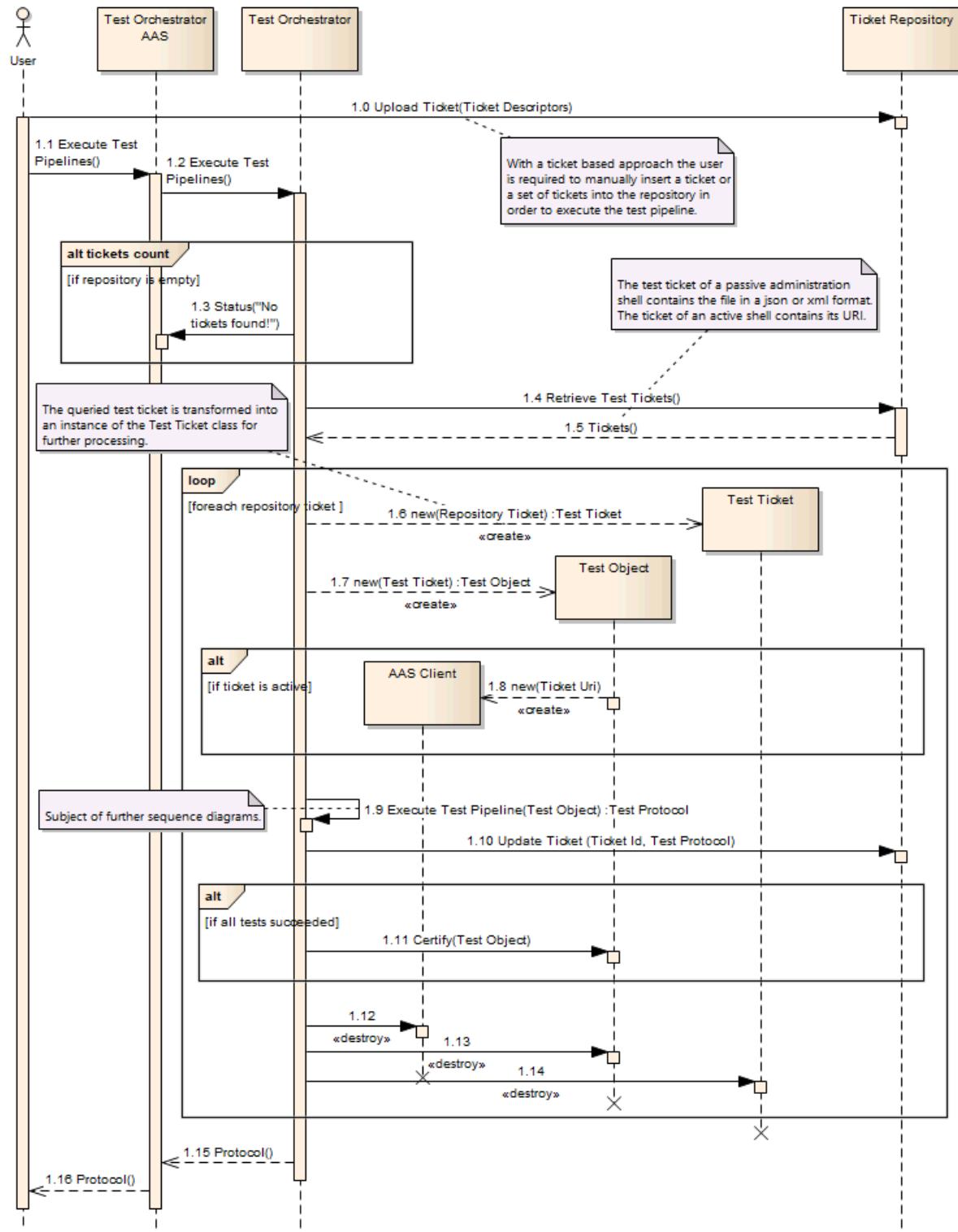


Abbildung 7: Ablauf Ticket-orientierte Testdurchführung

9 Funtionale Anforderungen

9.1 Applikation (Höherwertige Funktionalität)

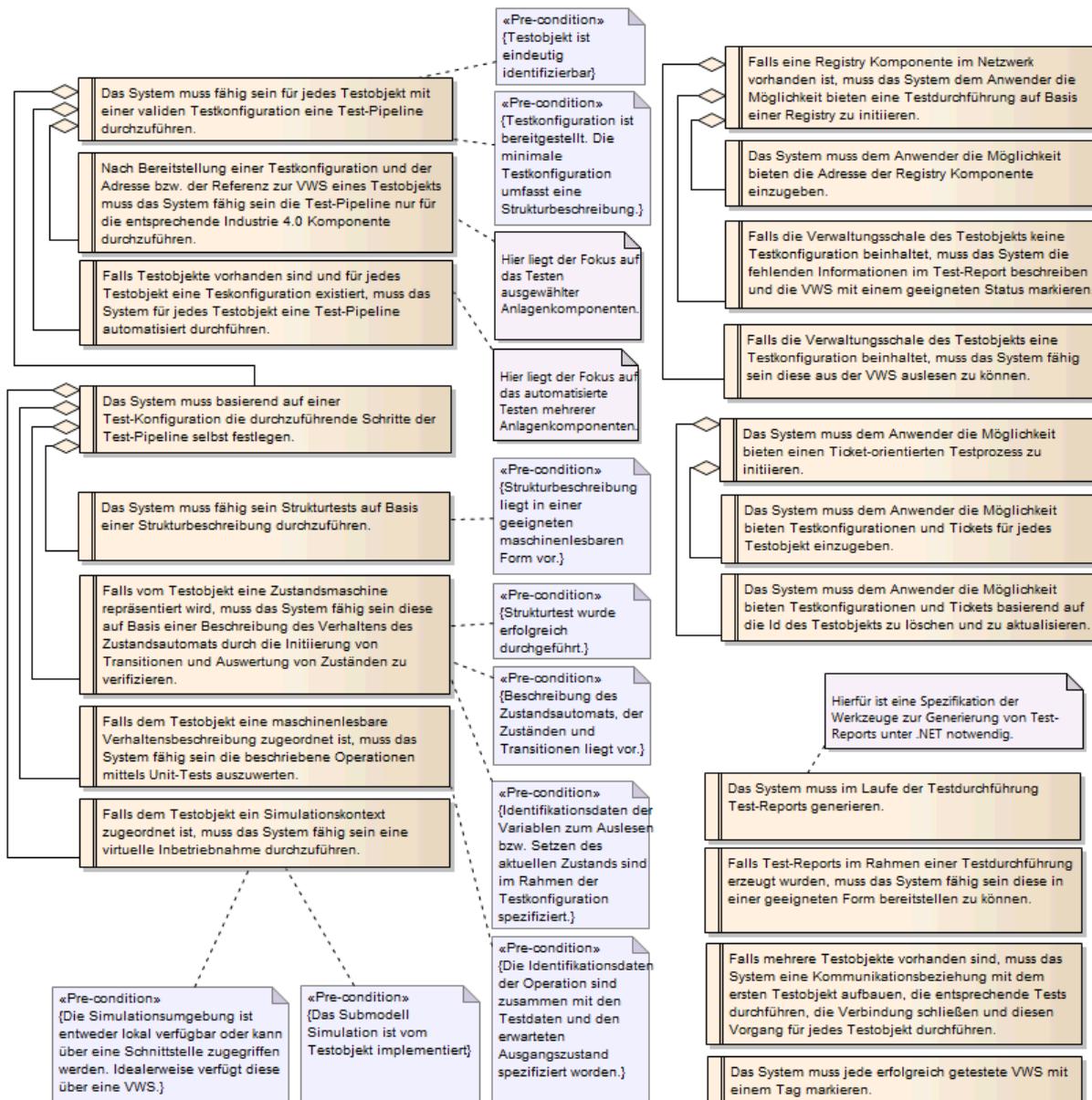


Abbildung 8: Applikation (Höherwertige Funktionalität)

Das System muss basierend auf einer Test-Konfiguration die durchzuführende Schritte der Test-Pipeline selbst festlegen.

Abgeleitet von:

Das System muss dem Anwender die Möglichkeit bieten die Adresse der Registry Komponente einzugeben.

Abgeleitet von:

Das System muss dem Anwender die Möglichkeit bieten einen Ticket-orientierten

Testprozess zu initiieren.

Abgeleitet von:

Das System muss dem Anwender die Möglichkeit bieten Testkonfigurationen und Tickets basierend auf die Id des Testobjekts zu löschen und zu aktualisieren.

Abgeleitet von:

Das System muss dem Anwender die Möglichkeit bieten Testkonfigurationen und Tickets für jedes Testobjekt einzugeben.

Abgeleitet von:

Das System muss fähig sein für jedes Testobjekt mit einer validen Testkonfiguration eine Test-Pipeline durchzuführen.

Abgeleitet von:

Das System muss fähig sein Strukturtests auf Basis einer Strukturbeschreibung durchzuführen.

Abgeleitet von:

Hierfür ist die Spezifikation eines geeigneten Formats zur Beschreibung des Soll-Stands der VWS-Struktur erforderlich. Dieser Testvorgang ist nicht optional und für jedes Testobjekt relevant.

Das System muss im Laufe der Testdurchführung Test-Reports generieren.

Abgeleitet von:

Das System muss jede erfolgreich getestete VWS mit einem Tag markieren.

Abgeleitet von:

Falls dem Testobjekt ein Simulationskontext zugeordnet ist, muss das System fähig sein eine virtuelle Inbetriebnahme durchzuführen.

Abgeleitet von:

Falls dem Testobjekt eine maschinenlesbare Verhaltensbeschreibung zugeordnet ist, muss das System fähig sein die beschriebene Operationen mittels Unit-Tests auszuwerten.

Abgeleitet von:

Im Kontext der Verwaltungsschale werden z. Zt. die Operationen als die gängigste Form zur Implementierung von Verhalten angesehen.

Diese sind mit den Funktionen bzw. Methoden in den prozeduralen und objektorientierten Sprachen vergleichbar und charakterisieren sich durch Eingangs- und Ausgangsvariablen. Ihre Verifikation kann also auf Basis von Unit-Tests stattfinden.

Falls die Verwaltungsschale des Testobjekts eine Testkonfiguration beinhaltet, muss das System fähig sein diese aus der VWS auslesen zu können.

Hierfür kann die Spezifikation eines Submodells *Test Specifications* zum Zweck des Prototyps sinnvoll sein.

Falls die Verwaltungsschale des Testobjekts keine Testkonfiguration beinhaltet, muss das System die fehlenden Informationen im Test-Report beschreiben und die VWS mit einem geeigneten Status markieren.

Verwaltungsschalen ohne Testkonfiguration sind vom Test-Orchestrator nicht verifizierbar.

Falls eine Registry Komponente im Netzwerk vorhanden ist, muss das System dem Anwender die Möglichkeit bieten eine Testdurchführung auf Basis einer Registry zu initiieren.

Abgeleitet von:

Falls mehrere Testobjekte vorhanden sind, muss das System eine Kommunikationsbeziehung mit dem ersten Testobjekt aufbauen, die entsprechende Tests durchführen, die Verbindung schließen und diesen Vorgang für jedes Testobjekt durchführen.

Abgeleitet von:

Falls Test-Reports im Rahmen einer Testdurchführung erzeugt wurden, muss das System fähig sein diese in einer geeigneten Form bereitstellen zu können.

Werkzeuge zur Erstellung von Test-reports müssen als Teil der Softwarearchitektur des Systems berücksichtigt werden. Idealerweise kann der Orchestrator Test-Reports als HTML-Files dem Anwender mit Visualisierung der Testfällen bereitstellen.

Falls Testobjekte vorhanden sind und für jedes Testobjekt eine Teskonfiguration existiert, muss das System für jedes Testobjekt eine Test-Pipeline automatisiert durchführen.

Abgeleitet von:

Die Initiierung der Pipeline kann als Operation der VWS des Test-Orchestrators implementiert werden.

Falls vom Testobjekt eine Zustandsmaschine repräsentiert wird, muss das System fähig sein diese auf Basis einer Beschreibung des Verhaltens des Zustandsautomats durch die Initiierung von Transitionen und Auswertung von Zuständen zu verifizieren.

Abgeleitet von:

Die erfolgreiche Synchronisation zwischen der VWS und dem Asset muss sichergestellt sein. Eine Möglichkeit dafür besteht in dem direkten Setzen des Assets in Zustand IDLE und das Auslesen des aktuellen Zustands über die Verwaltungsschale. Voraussetzung dafür ist, dass der Asset in einem anderen Zustand war.

Die Beschreibung des Zustandsautomats muss die Identifikationsdaten der Properties spezifizieren, welche zum Auslesen bzw. zum Setzen des aktuellen Zustands führen.

Der aktuelle Zustand muss über die VWS gesetzt und ausgelesen werden können.

Nach Bereitstellung einer Testkonfiguration und der Adresse bzw. der Referenz zur VWS eines Testobjekts muss das System fähig sein die Test-Pipeline nur für die entsprechende Industrie 4.0 Komponente durchzuführen.

Abgeleitet von:

Die Anforderung kann auf Basis einer VWS-Operation implementiert werden, die als Eingangsinformationen eine Testkonfiguration und zusammen mit der Adresse des Testobjekts beinhaltet. Die Operation kann einen Testbericht generieren und dieser dem Anwender als Ausgangsparameter bereitstellen.

10 Primäre APIs

Der folgende Abschnitt ermöglicht einen groben Überblick über die wichtigsten Softwaremodule des

Test-Orchestrators. Es werden ihre Zwecke, primäre Funktionalitäten und Charakteristiken diskutiert. Als Kernkomponenten gelten den Orchestrator zusammen mit einem Ticketverzeichnis, eine Testausführungskomponente und ein Modul zur Überführung von Test-Tickets in Testobjekten mit entsprechenden Testkonfigurationen.

11 Ticket Repository API

Die Ticket-Repository gilt als allgemeine Auskunftsstelle für Tickets und bietet Schnittstellen, die eine Ticket-Verwaltung ermöglichen. Zum Zweck der Ticket-orientierten Testdurchführung wird einen Zugriff zum Verzeichnis seitens des Orchestrators vorausgesetzt. Die konkrete Implementierung des Ticket-Verzeichnisses kann in Abhängigkeit vom Anwendungsfall und den Kundenpräferenzen variieren. Als geeignete Möglichkeiten gelten Datenbanken, Dateiverzeichnisse, externe Ticket-Systeme sowie Web-Anwendungen mit geeigneten Schnittstellen.

11.1 Ticket Repository

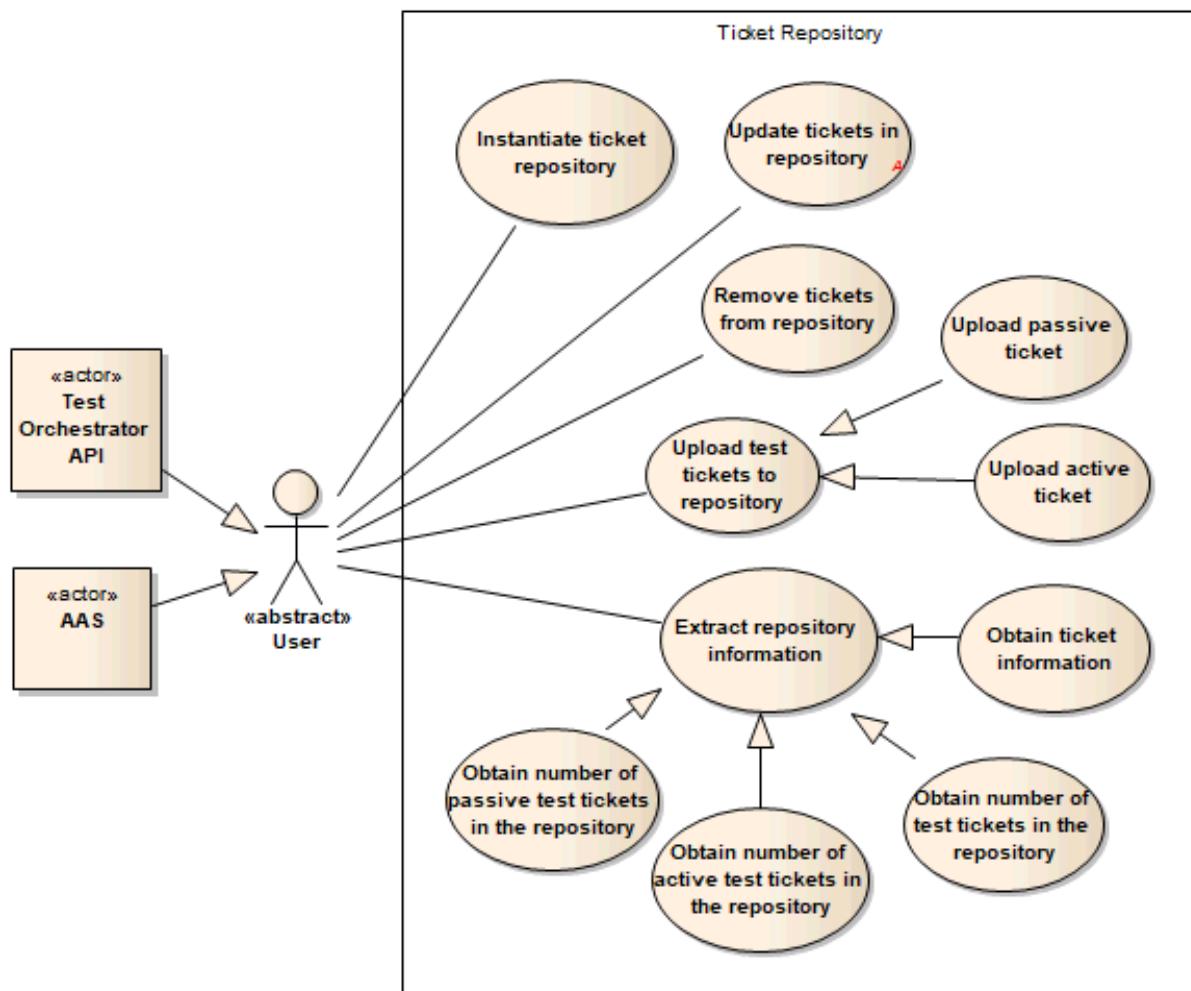


Abbildung 9: Ticket Repository

User

Der Anwender des Ticket-Verzeichnisses ist vom Design des Gesamtsystems abhängig. Das Verzeichnis kann entweder durch die API des Orchestrators nach einer Dependency Injection oder

direkt von der Verwaltungsschale der Testlösung zugegriffen werden.

Test Orchestrator API

Eine konkrete Instanz des Ticketverzeichnisses kann dem Test Orchestrator nach dem Dependency Injection Prinzip, also als Konstruktor-Argument, zur Verfügung gestellt werden. Dadurch erfolgt den Zugriff zur Repository auf Basis der Funktionalitäten des Test Orchestrators. Diese Variante dient der Komplexitätsreduktion auf die höchste Ebene des Gesamtsystems, da die Verwaltungsschale der Testlösung nur eine einzige Klasseninstanz, also diese des Test-Orchestrators, verwalten muss.

AAS

Als Anwender der Funktionalitäten des Ticket-Verzeichnisses lässt sich auch die Verwaltungsschale der Testlösung betrachten. In diesem Szenario verwaltet der digitale Vertreter zwei Instanzen. Die erste Instanz ist diese des Test Orchestrators. Diese charakterisiert sich durch einen Lesezugriff zum Ticketverzeichnis. Die zweite Instanz entspricht dem Ticketverzeichnis selbst. Über das entsprechende Submodell bekommt der Nutzer einen Schreibzugriff.

Upload active ticket

Die Instanz bietet dem Anwender die Möglichkeit Test-Tickets für aktive Verwaltungsschalen zum Ticketverzeichnis hinzuzufügen.

Upload passive ticket

Die Instanz bietet dem Anwender die Möglichkeit Test-Tickets für passive Verwaltungsschalen zum Ticketverzeichnis hinzuzufügen. Voraussetzung dafür ist, dass kein Ticket mit der gleichen Identifikationsskennzeichnung im Verzeichnis vorliegt. Im Rahmen des Uploads ist die AAS-Datei, also das Testobjekt, vom Anwender entsprechend bereitzustellen. Da das Hochladen von Dateien als Teil der Verwaltungsschale von der BaSyx-API nicht unterstützt wird, sind nicht die Dateien selbst, sondern ihre Inhalte beim Ticket-Upload zu übergeben.

Extract repository information

Das System bietet dem Anwender die Möglichkeit mit dem Ticketverzeichnis assoziierbare Informationen abzufragen.

Obtain number of active test tickets in the repository

Die Instanz kann die Anzahl der Tickets abfragen, welche einer aktiven Verwaltungsschale zugewiesen sind.

Obtain number of passive test tickets in the repository

Die Instanz kann die Anzahl der Tickets abfragen, welche einer passiven Verwaltungsschale zugewiesen sind.

Obtain number of test tickets in the repository

Die Instanz kann die Anzahl aller im Verzeichnis registrierten Tickets abfragen.

Obtain ticket information

Die Instanz bietet dem Anwender die Möglichkeit Ticket-Informationen abzufragen.

Remove tickets from repository

Die Instanz kann Ticket aus dem Ticketverzeichnis löschen.

Update tickets in repository

Die Instanz kann Tickets im Ticketverzeichnis aktualisieren. Als Eingangsinformation hier gilt die Identifikationskennzeichnung des Tickets.

Upload test tickets to repository

Die Instanz bietet dem Anwender die Möglichkeit Test-Tickets zum Ticketverzeichnis hinzuzufügen.

12 Test Object Provider API

Der TestObjectProvider kommuniziert mit dem Ticketverzeichnis. Seine Aufgabe besteht in der Interpretation der Ticket-Informationen. Auf Basis der Interpretation wird vom Provider ein aktives oder passives Testobjekt bereitgestellt.

12.1 Test-Object Provider API

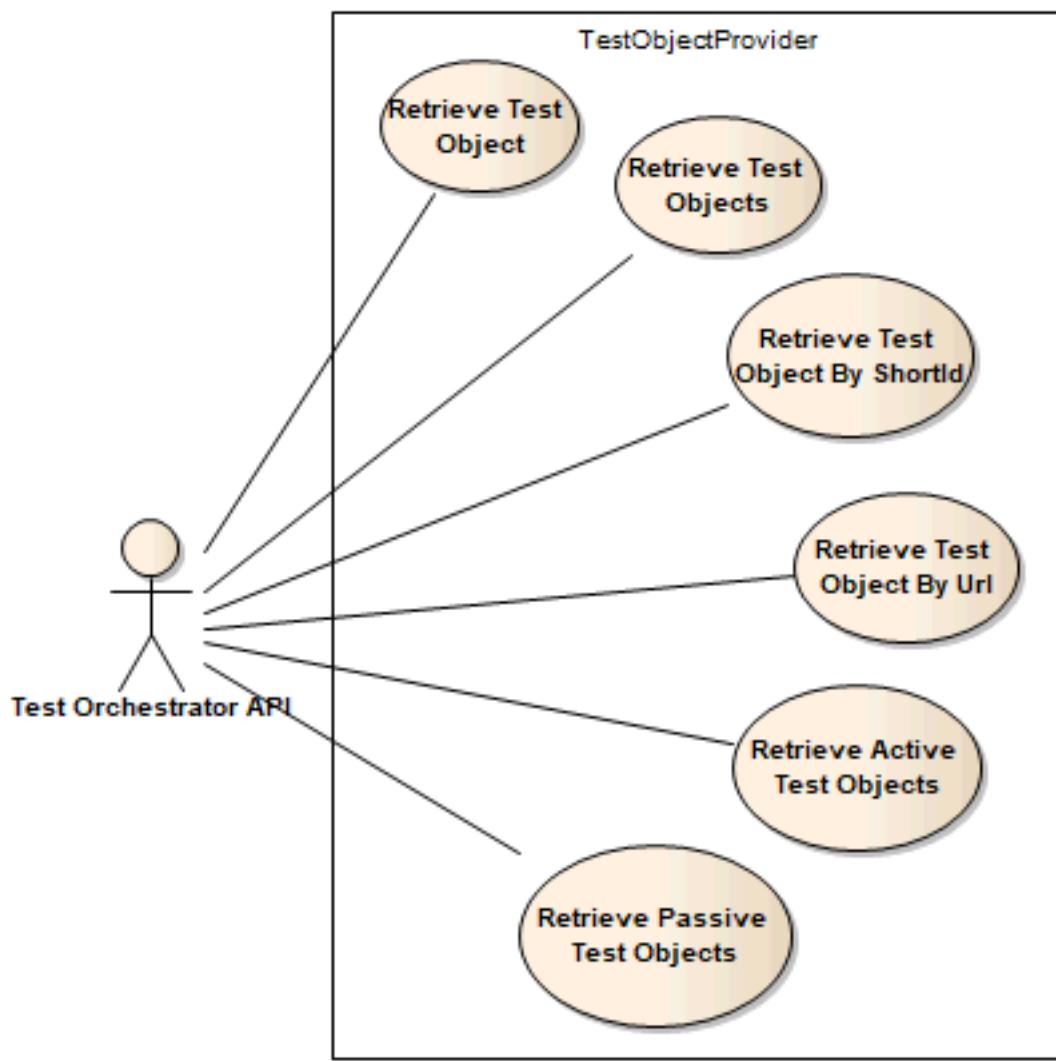


Abbildung 10: Test-Object Provider API

Test Orchestrator API

Zum Zweck der Testdurchführung verwendet die Test Orchestrator API die Funktionalitäten einer Klasse, welche zur Herstellung von Testobjekten basierend auf Ticket-Informationen verantwortlich ist.

Retrieve Active Test Objects

Die Funktion liefert eine Liste aus aller aktiven Testobjekten.

Retrieve Passive Test Objects

Die Funktion liefert eine Liste aus passiven Testobjekten.

Retrieve Test Object

Die Methode dient der Konstruktion eines geeigneten Testobjekts.

Retrieve Test Object By ShortId

Die Funktion liefert ein Testobjekt mit einem konkreten Ticket- und VWS-Identifikator.

Retrieve Test Object By Url

Die Funktion liefert ein aktives Testobjekt auf Basis der eingegebenen URL-Adresse.

Retrieve Test Objects

Die Funktion liefert eine Liste aus Testobjekten.

13 Test Runner API

13.1 Test Runner API

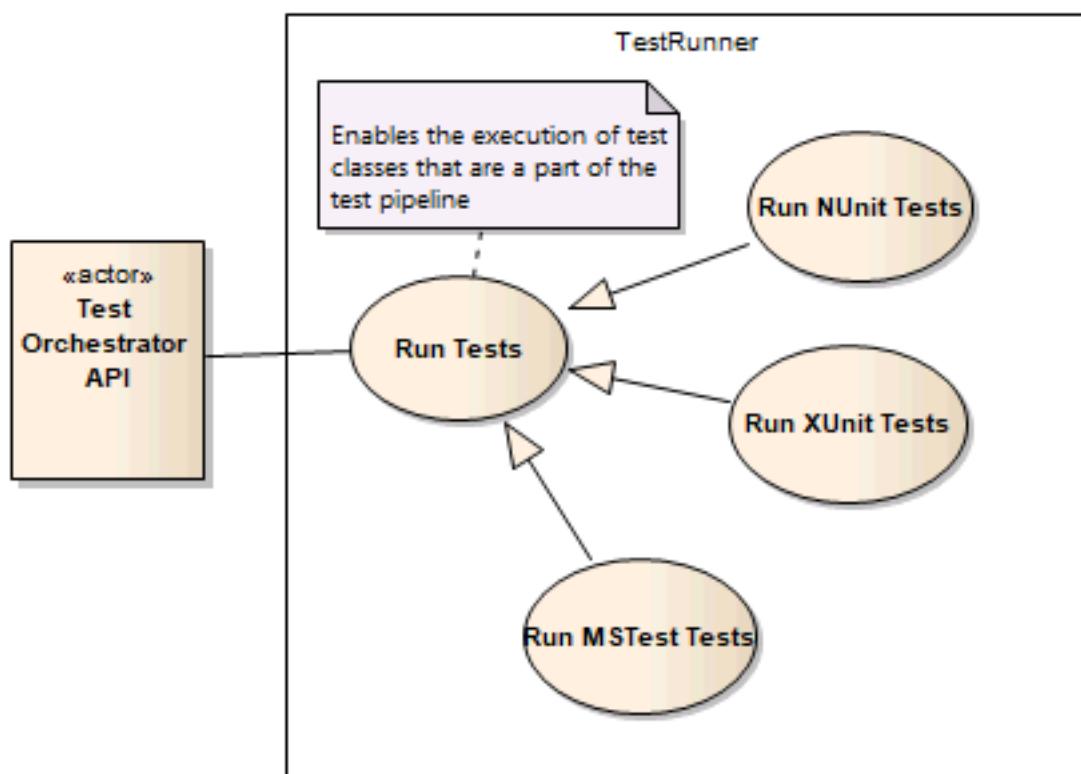


Abbildung 11: Test Runner API

Test Orchestrator API

Die Orchestrator API muss zum Zweck der Initiierung einer Test-Pipeline eine Komponente verwenden, die die Ausführung der entsprechenden Test-Klassen startet.

Run Tests

Die Funktion ermöglicht die Ausführung ausgewählter und vorimplementierter Testklassen. Diese wird angesichts der Ausführung der Test-Pipeline von der Orchestrator API benutzt. Zum Zweck der Kompatibilität des Gesamtsystems mit den gängigen .NET Test-Frameworks kann die Methode für jede Testumgebung entsprechend implementiert und im Kontext der Orchestrator API durch eine Schnittstelle abstrahiert werden. Dies würde die Durchführung der Test-Pipeline unabhängig von der im Rahmen der Erstellung der Testklassen eingesetzten Technologie garantieren.

Run NUnit Tests

Die Funktion ermöglicht die Ausführung von Testklassen, die auf das [NUnit](#) Test-Framework basieren.

Run XUnit Tests

Die Funktion ermöglicht die Ausführung von Testklassen, die auf das [XUnit](#) Test-Framework basieren.

Run MSTest Tests

Die Funktion ermöglicht die Ausführung von Testklassen, die auf das MSTest Test-Framework basieren.

14 Test Orchestrator API

14.1 Test Orchestrator API

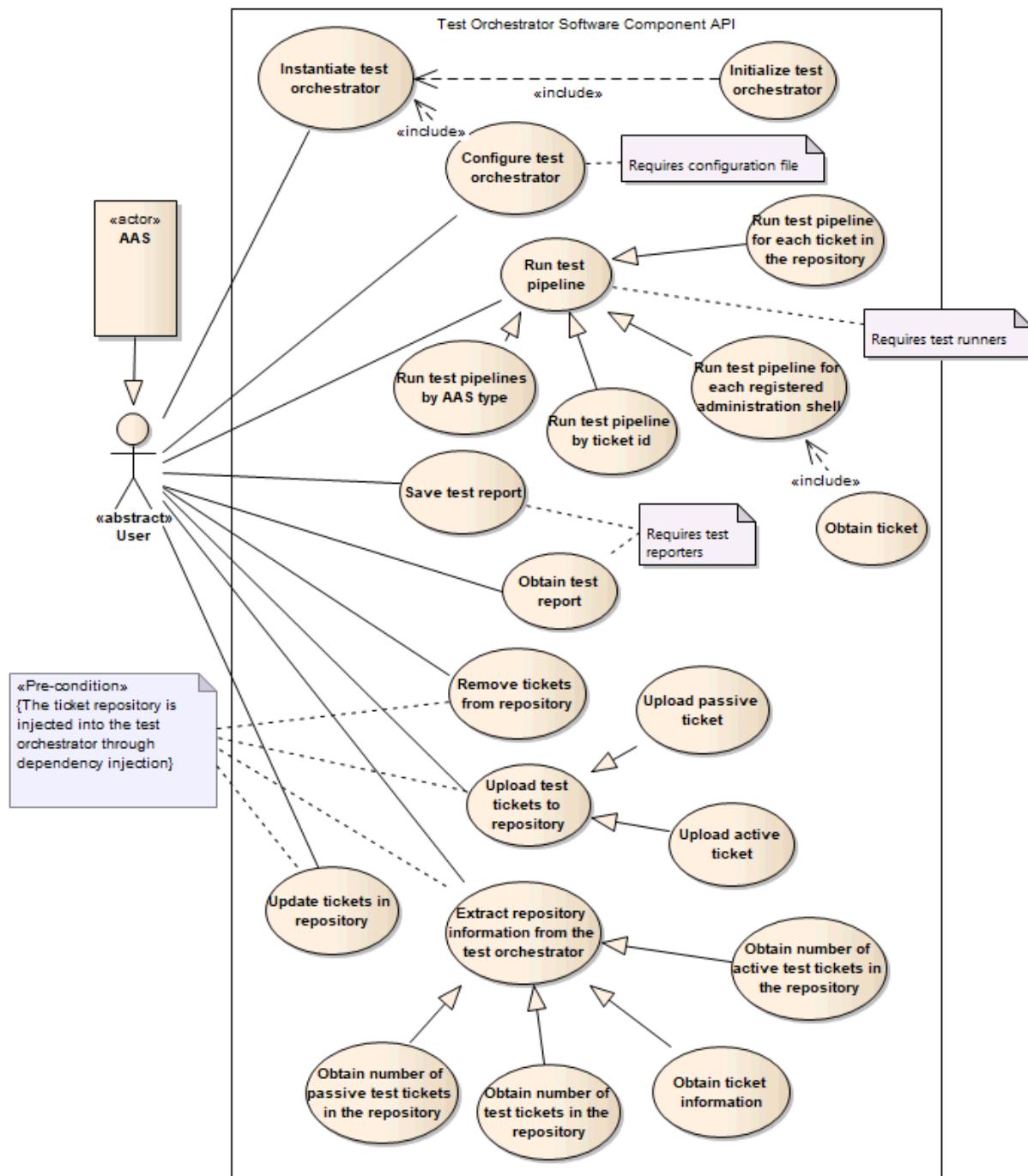


Abbildung 12: Test Orchestrator API

AAS

Die Testlösung setzt sich aus einer Softwarekomponente als Asset und einer Verwaltungsschale zusammen. Die Verwaltungsschale gilt als Anwender der Funktionalitäten der Test Orchestrator API. Im Kontext des digitalen Vertreters werden die Dienste des Orchestrators als Submodelle und Operationen bereitgestellt.

Upload active ticket

Die Instanz bietet dem Anwender die Möglichkeit Test-Tickets für aktive Verwaltungsschalen zum

Ticketverzeichnis hinzuzufügen.

Upload passive ticket

Die Instanz bietet dem Anwender die Möglichkeit Test-Tickets für passive Verwaltungsschalen zum Ticketverzeichnis hinzuzufügen. Voraussetzung dafür ist, dass kein Ticket mit der gleichen Identifikationsskennzeichnung im Verzeichnis vorliegt. Im Rahmen des Uploads ist die AAS-Datei, also das Testobjekt, vom Anwender entsprechend bereitzustellen. Da das Hochladen von Dateien als Teil der Verwaltungsschale von der BaSyx-API nicht unterstützt wird, sind nicht die Dateien selbst, sondern ihre Inhalte beim Ticket-Upload zu übergeben.

User

Der Anwender der Test Orchestrator API kann eine übergeordnete Softwarekomponente sein, welche die zur Verfügung stehende Funktionalitäten über eine geeignete Schnittstelle bereitstellt. Im Kontext der Testlösung werden die dargestellten Dienste mithilfe einer Verwaltungsschale über das Web zur Verfügung gestellt. Die API könnte auch zum Zweck der Erweiterung des Orchestrators um weitere Funktionalitäten und der Entwicklung weiterer Lösungen von Personen mit Fachkenntissen verwendet werden.

Instantiate test orchestrator

Die API muss dem Anwender die Möglichkeit bieten die Klasse *TestOrchestrator* zu instanziieren. Von der Klasse ist eine Interface zu implementieren, welche die Funktionalitäten der Komponente veranschaulicht und als einheitliche Schnittstelle für Erweiterungen gilt.

Initialize test orchestrator

Die Initialisierungsdaten der Softwarekomponente *TestOrchestrator* sind beim Anlegen der Instanz vom Anwender bereitzustellen. Alle Abhängigkeiten des Orchestrators, wie z.B. die konkrete Implementierung der Ticket Repository, sind als Argumente des Konstruktors auf Basis des Dependency Injection Prinzips zu übergeben. Dadurch kann die Austauschbarkeit der Komponenten zum Zweck eines potenziellen Kontextwechsels garantiert werden.

Configure test orchestrator

Zum Zweck der Konfiguration des Orchestrators ist eine Schnittstelle zu implementieren, welche bestimmte Daten aus einer Konfigurationsdatei ausliest. Die Konfigurationsdatei kann entweder als JSON oder XML gespeichert werden, muss von der Softwarekomponente deserialisiert werden können und ist beim Anlegen der Instanz dem Konstruktor zu übergeben.

Extract repository information from the test orchestrator

Das System bietet dem Anwender die Möglichkeit mit dem Ticketverzeichnis assoziierbare Informationen abzufragen.

Obtain number of active test tickets in the repository

Die Instanz kann die Anzahl der Tickets abfragen, welche einer aktiven Verwaltungsschale zugewiesen sind.

Obtain number of passive test tickets in the repository

Die Instanz kann die Anzahl der Tickets abfragen, welche einer passiven Verwaltungsschale zugewiesen sind.

Obtain number of test tickets in the repository

Die Instanz kann die Anzahl aller im Verzeichnis registrierten Tickets abfragen.

Obtain ticket information

Die Instanz bietet dem Anwender die Möglichkeit Ticket-Informationen abzufragen.

Remove tickets from repository

Die Instanz kann Ticket aus dem Ticketverzeichnis löschen.

Run test pipeline

Die Instanz muss dem Anwender die Möglichkeit bieten eine oder mehrere Test-Pipelines durchzuführen. Da die Tests durch den Einsatz unterschiedlicher .NET Test-Frameworks wie z.B. NUnit oder XUnit implementiert werden können, ist die entsprechende Realisierung durch Funktionsargumente vom Client zu spezifizieren. Somit kann die Abhängigkeit des Orchestrators vom konkreten Test-Framework eliminiert werden.

Run test pipeline by ticket id

Die Instanz kann eine Test-Pipeline auf Basis der Identifikation des Tickets durchführen. Die Ticket Id korrespondiert mit der IdShort der Verwaltungsschale und kann von der Strukturbeschreibung ausgelesen werden.

Run test pipeline for each registered administration shell

Das System kann eine Test-Pipeline für jede registrierte I4.0 Komponente durchführen. Diese Funktionalität setzt die Erreichbarkeit der Registry-Komponente und die Existenz eines Test-Tickets als Teil der VWS des Testobjekts voraus. Die Instanz muss das Ticket entsprechend auslesen können.

Obtain ticket

Im Rahmen der registry-orientierten Testdurchführung muss das System das Ticket (also die Testkonfiguration) aus der Verwaltungsschale des Testobjekts auslesen können. Hierfür ist die Definition eines geeigneten Submodells erforderlich. Kann das Ticket nicht interpretiert werden, so gilt dieser als ungültig und das Testobjekt kann nicht als verifiziert gekennzeichnet werden.

Run test pipeline for each ticket in the repository

Die Instanz kann eine Test-Pipeline für jedes Testobjekt durchführen, welches einem Ticket zugeordnet wird. Diese Funktionalität setzt einen Zugriff zum Ticketverzeichnis voraus.

Run test pipelines by AAS type

Die Instanz kann die Tickets im Ticketverzeichnis nach Typ der Verwaltungsschale filtern und für jede Klasse eine Test-Pipeline durchführen.

Save test report

Die Instanz kann Testberichte erstellen und speichern. Hierfür ist die Existenz einer Klasse erforderlich, welche für die Erstellung von Testberichten verantwortlich ist. Ein mögliches Format für die Protokollierung ist HTML.

Obtain test report

Der Anwender des Systems kann den Testbericht jedes getesteten Testobjekts abfragen.

Update tickets in repository

Die Instanz kann Tickets im Ticketverzeichnis aktualisieren. Als Eingangsinformationen hier gilt die Identifikationskennzeichnung des Tickets.

Upload test tickets to repository

Die Instanz bietet dem Anwender die Möglichkeit Test-Tickets zum Ticketverzeichnis hinzuzufügen.

15 Komponenten

In diesem Abschnitt werden die grundlegenden Module und Klassen beschrieben, die eine Möglichkeit zur softwaretechnischen Implementierung eines Test-Orchestrators darstellen. Die Ableitung der Module erfolgt nach dem Bottom-Up Prinzip.

15.1 Komponenten

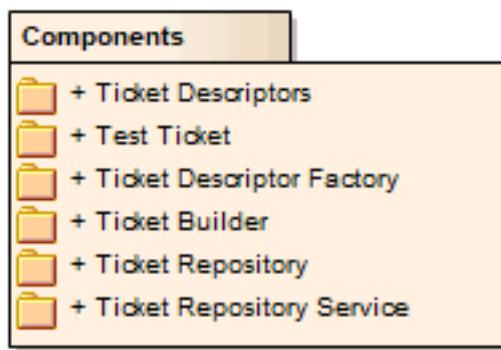


Abbildung 13: Komponenten

Components

Das Paket umfasst alle notwendigen Komponenten zur Realisierung eines MVP. Aus den Anforderungen wurden folgende Module abgeleitet:

- Test Ticket Descriptors
- Test Tickets
- Test Ticket Repository
- Test Objects
- Test Object Providers
- Test Runners
- Test Classes (vorimplementierte Pipeline Tests)
- Test Orchestrator Settings
- Test Orchestrator
- Test Orchestrator Asset Administration Shell
- Common

In den folgenden Abschnitten wird jedes Modul detailliert beschrieben.

16 Components

Das Paket umfasst alle notwendigen Komponenten zur Realisierung eines MVP. Aus den Anforderungen wurden folgende Module abgeleitet:

- Test Ticket Descriptors
- Test Tickets
- Test Ticket Repository
- Test Objects
- Test Object Providers
- Test Runners
- Test Classes (vorimplementierte Pipeline Tests)
- Test Orchestrator Settings

- Test Orchestrator
- Test Orchestrator Asset Administration Shell
- Common

In den folgenden Abschnitten wird jedes Modul detailliert beschrieben.

17 Ticket Descriptors

Wie bereits erwähnt wurde, stellt ein Ticket im Kontext der Testlösung die Eingangsinformationen zur Durchführung der gesamten Test-Pipeline dar. Laut Definition setzt sich diese aus

- Strukturtests,
- Kommunikationstests,
- zustandsbezogene Tests,
- Verhaltenstests sowie
- die Durchführung einer virtuellen Inbetriebnahme

zusammen.

Jeder Stufe des Testprozesses wird einen Deskriptor zugewiesen, der eine Erwartungshaltung spezifiziert und als Teil des Tickets verfügbar ist. Wie bereits spezifiziert wurde, muss das System dem Anwender die Möglichkeit bieten Test-Tickets dem Orchestrator zu übergeben. Der Nutzer muss also die entsprechende Deskriptoren in einer geeigneten Form der Testlösung zur Verfügung stellen können. Hierfür wurden im Rahmen der Konzepterstellung zwei Möglichkeiten identifiziert.

Variante 1:

Eine REST-Schnittstelle zum Überschreiben oder Ersetzen bereits existierender Dateien einer VWS ist als Teil des allgemeinen [Submodell-Controllers](#) im Rahmen des BaSyx .NET SDK implementiert worden. Die BaSyx [AAS REST-API](#) bietet jedoch keine Funktionalitäten zur dynamischen Speicherung von Dateien als Elemente einer Verwaltungsschale. Unter dynamische Speicherung wird das Hochladen einer bei der Erstellung der VWS unspezifizierten Anzahl von Dateien in Laufzeit verstanden.

Aufgrund der begrenzten Funktionalitäten der BaSyx [AAS REST-API](#), stellt die erste Möglichkeit die Implementierung einer weiteren REST-Funktionalität dar, welche ein direktes Datei-Upload aus beliebigen Endpunkten ermöglicht.

Als Vorteil dieses Ansatzes gilt die nutzerfreundliche Interaktion des menschlichen Nutzers mit der Testlösung. Bei einem Ticket-Upload können die Deskriptoren entweder durch die Spezifikation des lokalen Pfads oder mittels einer Upload-Taste ausgewählt werden.

Ein bedeutender Nachteil dieser Variante ist, dass eine solche Erweiterung einer Abweichung von der standardisierten Kommunikationsschnittstelle entspricht. Aus diesem Grund gilt die zweite Alternative als eine bessere Variante.

Variante 2:

Die Inhalt der entsprechenden XML oder JSON Deskriptor-Datei wird einer entsprechenden Orchestrator-Operation als Zeichenkette übergeben.

Der Nachteil dieses Ansatzes besteht in der aufwändigen Interaktion des menschlichen Nutzers mit der Testlösung. Der Aufwand wird durch das Kopieren und Einfügen von langen XML Bäumen und JSON Objekten dargestellt. Weiterhin muss die Testlösung die entsprechende Deskriptor-orientierte Validierungsmechanismen auf Basis der entsprechenden Schemas implementieren.

Als Vorteil gilt die Beibehaltung der standardisierten REST API.

17.1 Ticket Descriptors

Das nachfolgende Klassendiagramm veranschaulicht die softwaretechnische Umsetzung der Ticket-Deskriptoren. Diese gelten als Basis für die Definition des Metamodells eines Tickets. Zum Zweck der Erweiterbarkeit wurde jeder Deskriptor durch das entsprechende Interface abstrahiert. Dadurch kann jede Klasse, welche einer nachfolgend dargestellten Schnittstelle gerecht wird, als gültige Ersatzmöglichkeit betrachtet und ohne Softwareänderungen im System integriert werden.

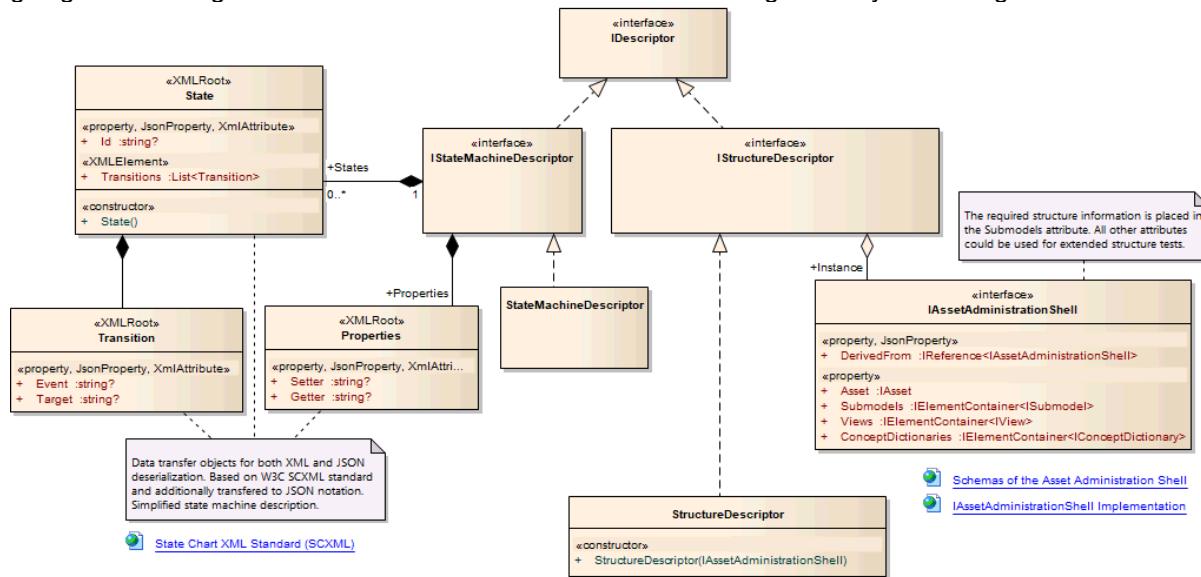


Abbildung 14: Ticket Descriptors

StructureDescriptor

Diese Klasse implementiert das **IStructureDescriptor**-Interface und kann somit als einen gültigen Strukturdeskriptor betrachtet werden.

StateMachineDescriptor

Diese Klasse dient als Schablone zur Erstellung von Deskriptoren diverser Zustandsmaschinen.

IDescriptor

Dieses Interface ist zum Zweck der Erweiterbarkeit aller Ticket-Deskriptoren entstanden und dient als gemeinsame Programmierstelle zur Spezifikation ihrer gemeinsamen Charakteristiken. Im Rahmen der prototypischen Realisierung findet die Schnittstelle keine Anwendung. Sie gilt als Super-Klasse aller Deskriptor-Interfaces und wurde zum Zweck der Implementierung eines Deskriptor-Verwaltungssystems als Gegenstand zukünftiger Arbeiten implementiert.

IStateMachineDescriptor

Zum Zweck der universellen Beschreibung einer Zustandsmaschine im Kontext des Test-Orchestrators wurde das [SCXML Standard](#) identifiziert. Die wichtigsten Eigenschaften des Metamodells einer SCXML-Zustandsmaschine, nähmlich die Zustände und Transitionsereignisse, können manuell als Klassen oder aus dem XML-Schema mittels eines Generators implementiert bzw. abgeleitet werden. Die Schnittstelle beinhaltet die Liste aller Zustände zusammen mit den Identifikationsdaten der VWS-Eigenschaften, welche den Zugriff zur Führungskomponente ermöglichen und zum Auslesen bzw. Setzen des aktuellen Zustands führen.

IAssetAdministrationShell

Das Eclipse-Basyx Interface [IAssetAdministrationShell](#) gehört zum [BaSyx .NET SDK](#) und repräsentiert die Eigenschaften, durch welche jede Verwaltungsschale bzw. Industrie 4.0

Komponente charakterisiert wird. Von besonderer Bedeutung für die Verifikation der VWS-Struktur einer Komponente ist das zusammengesetzte Instanzattribut *Submodels*, welches Informationen über alle Submodelle und ihre Elemente beinhaltet.

IStructureDescriptor

Die Strukturbeschreibung beinhaltet alle Informationen zur Struktur einer Verwaltungsschale und kann somit als eine Sammlung von statischen Daten angesehen werden. Der Deskriptor muss die Identifikationsdaten aller Submodelle, Submodell-Elemente und Elementensammlungen in einer hierarchischen Form, also zusammen mit den entsprechenden Beziehungen zueinander, als Grundlage für die Strukturtests bereitstellen können.

Als geeignetes Format zur Spezifikation der erwarteten Struktur werden im Kontext des Prototyps XML und JSON angesehen. Für die eigentliche Semantik der Strukturbeschreibung eignen sich die von der GitHub-Organisation [admin-shell-io by IDTA](#) bereitgestellten Schemas, siehe [aas-specs](#). Da diese i.d.R. zur Erstellung von Typ 1 Verwaltungsschalen verwendet werden, sind die Schemas für Strukturspezifikationen sehr gut geeignet und von mehreren Organisationen bereits validiert worden.

Für die Strukturtests kann entweder das ganze Schema oder nur den Anteil übernommen werden, welcher die Strukturinformationen beschreibt. Es wird zwischen zwei Möglichkeiten unterschieden.

Verwendung des gesamten Schema

Durch die Festlegung des gesamten Metamodells als Grundlage für die Durchführung von Strukturtests entsteht ein Erweiterungspotenzial für die gesamte Testlösung. Das Format bietet der Testlösung die Möglichkeit nicht nur strukturorientierte Tests durchzuführen sondern alle statischen Daten einer VWS zu verifizieren. Des Weiteren können somit Strukturbeschreibungen direkt mithilfe des Package-Explorers generiert und dem Orchestrator nachfolgend zur Verfügung gestellt werden.

Ableitung der Strukturinformationen aus dem Schema

Eine weitere Möglichkeit entspricht der Identifikation und Ableitung der notwendigen Strukturinformationen aus der gesamten Vorlage. Im Rahmen dieses Ansatzes wird vom Anwender die Strukturspezifikation manuell und in textueller Form, als JSON oder XML-Datei und auf Basis der erwähnten Schemas, spezifiziert und dem Orchestrator über die entsprechende Schnittstelle bereitgestellt. Hier operiert der Orchestrator also nicht mit der gesamten Beschreibung einer passiven VWS, sondern mit dem Teil davon, welcher die Submodelle und ihre Strukturen beschreibt.

Es besteht die Frage wie die Integration der dateibasierten Strukturbeschreibung, also die XML bzw. JSON Datei, in der .NET Softwareumgebung stattfindet. Hierfür existieren unterschiedliche Möglichkeiten.

Variante 1:

Minimalbeispiel für eine durch XML beschriebene Typ 1 VWS:
<https://github.com/admin-shell-io/aas-specs/blob/master/schemas/xml/examples/minimum.xml>

Minimalbeispiel für eine durch JSON beschriebene Typ 1 VWS:
<https://github.com/admin-shell-io/aas-specs/blob/master/schemas/json/examples/miniJsonExample.json>

Aus den beiden Minimalbeispielen und den bereits erwähnten Schemas geht hervor, dass die Strukturinformationen als Array aus JSON bzw. XML-Objekten gekapselt sind. Nach einer Ableitung der entsprechenden C#-Klassen aus den Beschreibungen lassen sich die Strukturdaten mittels geeigneter Bibliotheken auslesen. Die Ableitung der Klassen kann entweder manuell seitens des Programmierers oder durch Generatoren wie das [XML Schema Definition Tool](#) realisiert werden.

Variante 2

Zum Zweck des Umgangs mit externen Verwaltungsschalen definiert Eclipse Basyx die

entsprechende Funktionalitäten, siehe [Basyx.Models.Export](#). Das Paket implementiert Klassen, welche sowohl für die Transformation von Verwaltungsschalen in XML und JSON Dateien (Export) als auch für eine Deserialisierung (Import) verantwortlich sind.

Zum Zweck der Erweiterbarkeit des strukturorientierten Testvorgangs können die Informationen nach der Deserialisierung durch die BaSyx Schnittstelle `IAssetAdministrationShell` repräsentiert werden. Auf diese Weise operiert der Orchestrator mit zwei Instanzen dieses Interface. Während die erste Instanz das eigentliche Testobjekt repräsentiert, beschreibt die zweite Instanz seine Soll-Struktur.

18 Test Ticket

18.1 Test Ticket

Die folgende Abbildung stellt die Struktur eines Tickets dar.

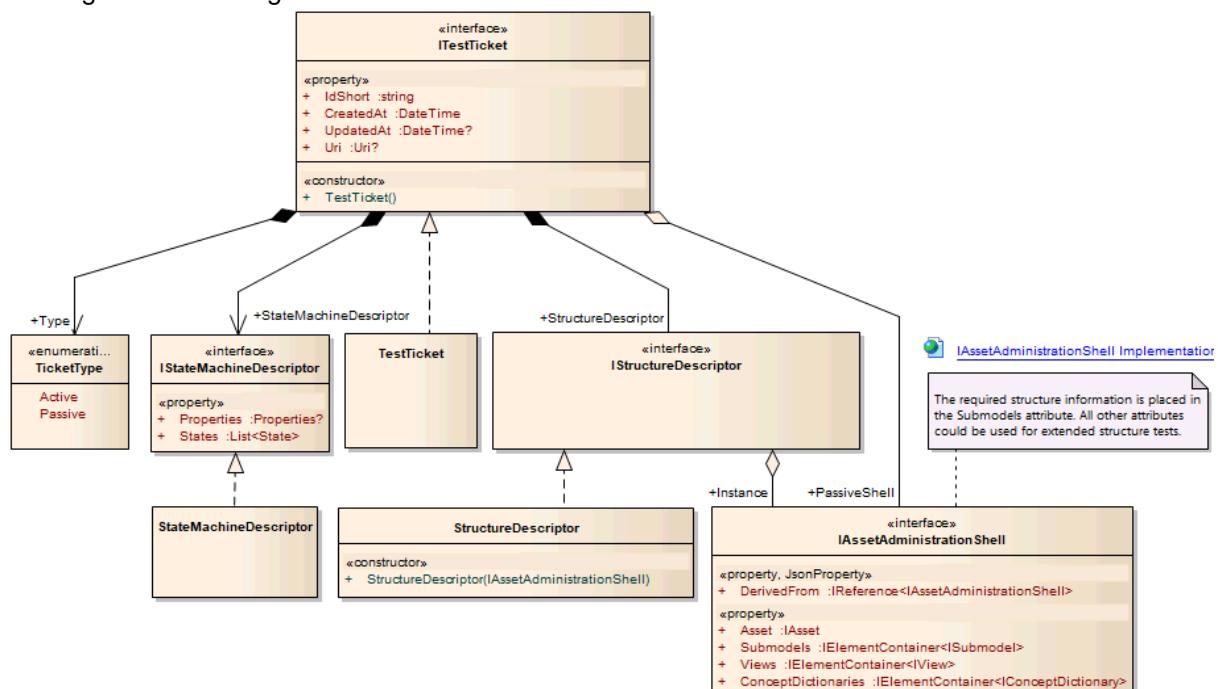


Abbildung 15: Test Ticket

19 Ticket Descriptor Factory

19.1 Ticket Descriptor Factory

Das folgende Klassendiagramm illustriert das Modul, welches für die Instanziierung der Ticket-Deskriptoren verantwortlich ist und somit eine Basis für die Erstellung eines Ticket bietet.

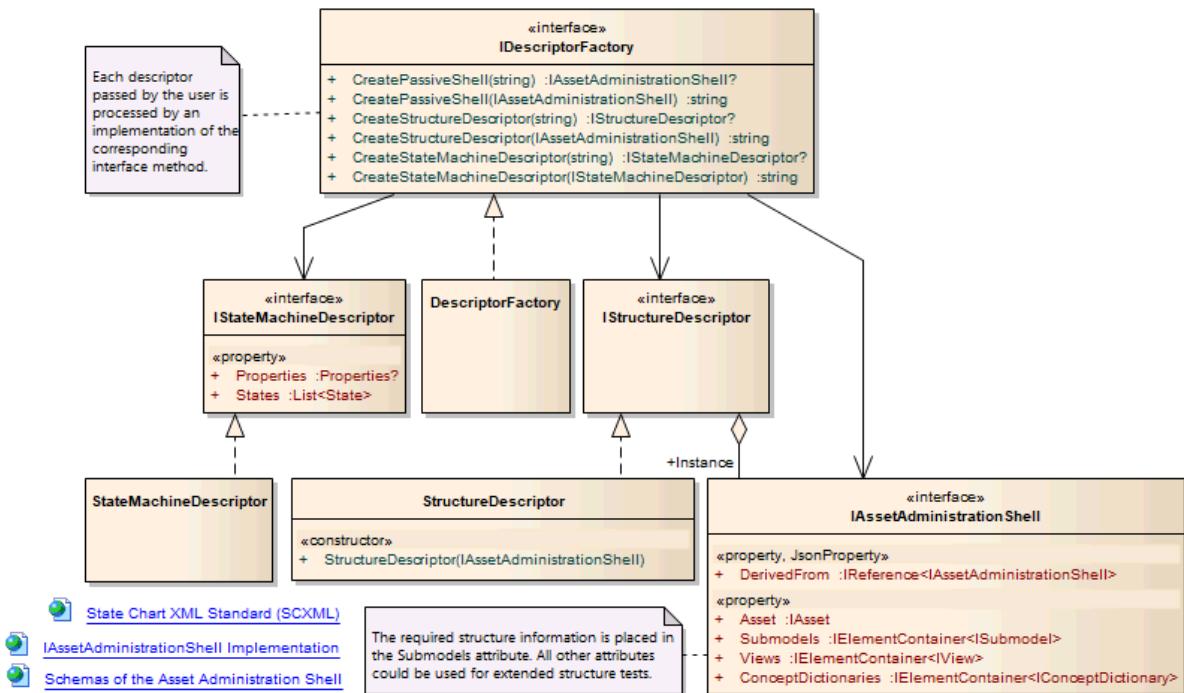


Abbildung 16: Ticket Descriptor Factory

DescriptorFactory

Konkrete Implementierung der Schnittstelle **IDescriptorFactory**.

IDescriptorFactory

Diese Schnittstelle definiert Methoden zur Serialisierung und Deserialisierung der entsprechenden Deskriptoren. Alle Methoden zur Serialisierung bekommen als Eingangsinformation eine Instanz der zu serialisierenden Klasse und geben eine XML oder JSON Zeichenkette zurück. Die Funktionalitäten zu Deserialisierung ermöglichen die Transformation von Zeichenketten in C# Klassen. Da die Interaktion des Anwenders mit dem System die Übergabe von JSON und XML-Inhalten voraussetzt, werden im Rahmen der Methoden zur Deserialisierung die bereitgestellten Inhalte validiert. Nach der Validierung erfolgt eine Instantiierung des Deskriptors. Dies gilt als Voraussetzung für die Erstellung eines Test-Tickets.

20 Ticket Builder

20.1 TicketBuilder

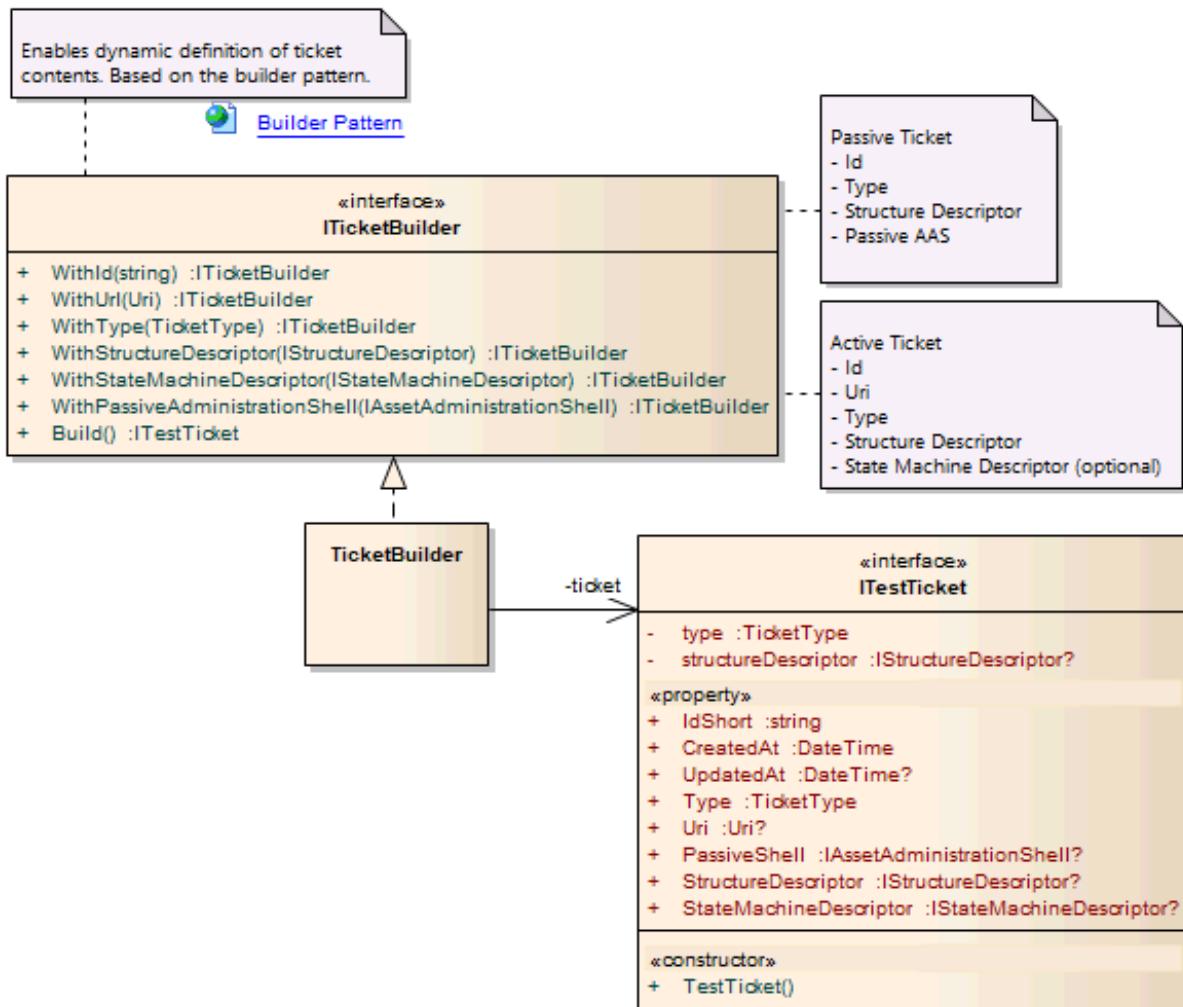


Abbildung 17: TicketBuilder

TicketBuilder

Diese Klasse repräsentiert eine konkrete Implementierung der Ticket Builder Schnittstelle.

ITicketBuilder

Es wird zwischen passiven, aktiven und proaktiven Verwaltungsschalen differenziert. Im Fokus des Konzepts liegt die Verifikation von Testobjekten mit aktiven und passiven Verwaltungsschalen. In Abhängigkeit vom Typ der VWS unterscheidet sich die Inhalt eines Tickets.

Ein passives Ticket charakterisiert sich durch die folgende Informationen:

- Ticket Id
- Ticket Type
- Structure Descriptor
- Passive AAS (XML/JSON)

Ein aktives Ticket beinhaltet:

- Ticket Id
- Ticket Type
- Structure Descriptor

- State Machine Descriptor (optional)

Das Ticket Builder Interface ermöglicht die dynamische Konstruktion von Tickets mit unterschiedlichen Inhalten durch die Existenz einer einzigen Klasse. Ein Ticket wird im Kontext der Softwareapplikation nur durch den Einsatz des Ticket Builder erstellt.

21 Ticket Repository

21.1 TicketRepository

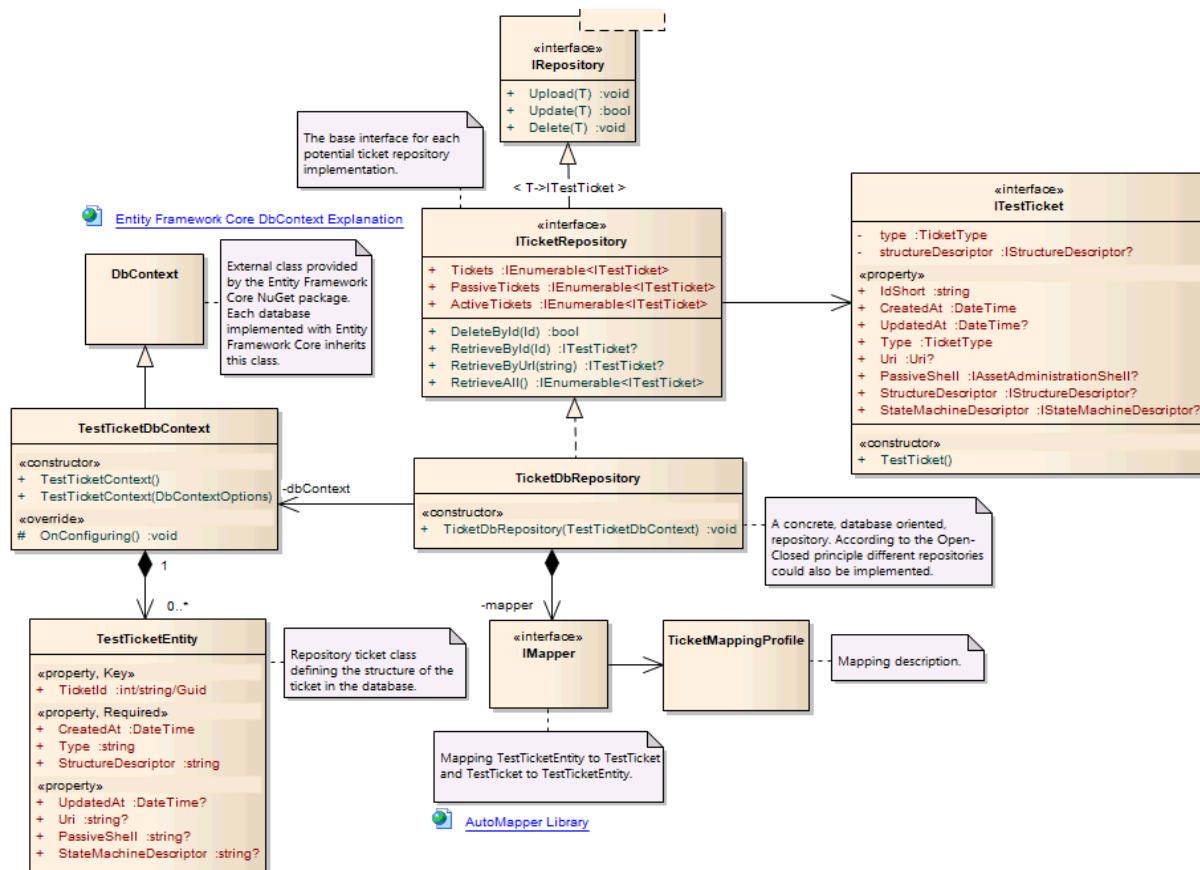


Abbildung 18: TicketRepository

TicketDbRepository

Diese Klasse entspricht einer konkreten Implementierung des `ITicketRepository` Interface. Die konkrete Implementierung benutzt als Persistenzort für die Tickets eine SQL Datenbank, welche durch die `TestTicketDbContext`-Klasse repräsentiert wird.

IRepository

Das `IRepository` Interface definiert die grundlegenden Operationen eines Ticket Verzeichnisses.

ITicketRepository

Durch diese Schnittstelle werden die grundlegenden Dienste des Ticket-Verzeichnisses um weitere

Funktionalitäten ergänzt. Jede konkrete Implementierung der Repository muss die spezifizierte Operationen erfüllen, damit Sie in Zusammenhang mit dem Test-Orchestrator verwendet werden kann.

Die Dienste wurden bereits im Rahmen der Anwendungsfalldiagramme erwähnt.

22 Ticket Repository Service

22.1 Ticket Repository Service

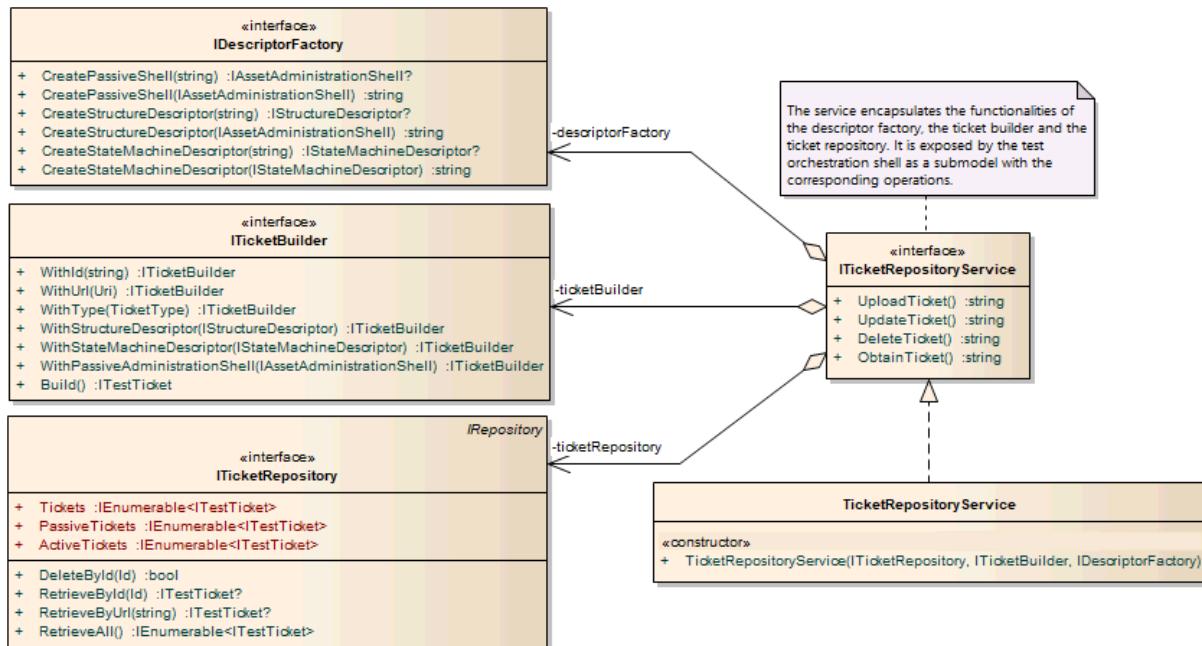


Abbildung 19: Ticket Repository Service

TicketRepositoryService

Aus dem Klassendiagramm geht hervor, dass auf Basis der Dependency Injection-Methode die Instanzen der Schnittstellen IDescriptorFactory, ITicketBuilder und ITicketRepository als Argumente des Konstruktors übergeben werden. Somit kapselt der Dienst ihre Funktionalitäten und stellt diese über die vom Interface definierte Instanzmethoden zur Verfügung.

ITicketBuilder

ITicketRepositoryService

Dieses Interface definiert die Funktionalitäten des Ticket Repository Dienstes. Diese werden von den Operationen eines im Rahmen der nächsten Abschnitten definierten Submodells aufgerufen.

