A Test Concept for the Development of Microservice-based Applications

Michael Schneider, Stephanie Zieschinski,
Hristo Klechorov, Lukas Brosch,
Patrick Schorsten, Sebastian Abeck
Research Group Cooperation & Management
Karlsruhe Institute of Technology (KIT)
Zirkel 2, 76131 Karlsruhe, Germany
email: (michael.schneider | sebastian.abeck)@kit.edu
(stephanie.zieschinski | hristo.klechorov | lukas.brosch)@student.kit.edu

Christof Urbaczek
xdi360 GmbH
Leopoldstraße 252b, 80807 München
email: (christof.urbaczek@xdi360.com)

***Abstract*—A microservice-based application is composed of several distributed microservices. When developing the microservices of the application, it is important to test that the requirements are met and that the application works as intended. Especially end-to-end tests require all involved microservices to be available for testing. A common way is to execute the tests via a continuous integration / continuous delivery pipeline. In this paper, we present a test concept for developing microservice-based applications which covers the different test types according to the test pyramid, from end-to-end, integration tests, and consumer-driven contract to unit tests. The test concept considers the entire test pyramid as part of the microservice engineering process. Furthermore, we show how the test concept can be executed during the development process using a continuous integration / continuous delivery pipeline by the example of a PredictiveCarMaintenance application.**

***Keywords*—microservices; development process; behavior-driven development; test pyramid; test concept; code quality; CI/CD.**

I. Introduction

A microservice-based application is composed of several independently developed and deployed small services. The microservices are loosely coupled into business-related cohesive functionalities that do one thing well [1]. Microservices communicate with each other via technology-independent interfaces to solve the more extensive business tasks. The architectural style Representational State Transfer (REST) by Roy Fielding [2] provides a lightweight way to define the microservices' web Application Programming Interfaces (APIs). As a result, each microservice can be developed separately by different development teams using different programming languages, and can be tested and deployed independently from each other. At the same time, testing the whole application becomes far more complex, since the microservices are distributed. Testing an application itself has to consider the whole test pyramid [3] and the different tests types. This includes unit, integration, Consumer-Driven Contract (CDC), and End-to-End (E2E) tests. However, especially E2E tests are important, since the interaction of microservices fulfill the business functionality of a microservice-based application which has to be tested [4]. In addition, all involved microservices need to be available for testing. To simplify the test process, a pipeline for Continuous Integration / Continuous Deployment (CI/CD) has to be set up to assist the development process and the use of the test concept. This enables the regression testing of the application on the level of the business requirements in form of user acceptance tests.

The development of microservice-based applications requires a systematic development approach so that developers know what to test. For the test concept, a systematic microservice engineering approach is followed. Therefore, the test concept is integrated into the microservice-based development process [5]. Testing is considered during the whole engineering process, including the requirements analysis, design and the implementation phase. Figure 1 displays an overview of the development process and the resulting test artifacts. In the requirements analysis, the required functionality is specified by several artifacts. For testing purposes, the acceptance criteria is specified as Gherkin features according to Behavior-Driven Development (BDD) practices, which are used for the development of end-to-end tests. Gherkin features embrace the natural language which simplifies the communication with the stakeholders requirements.

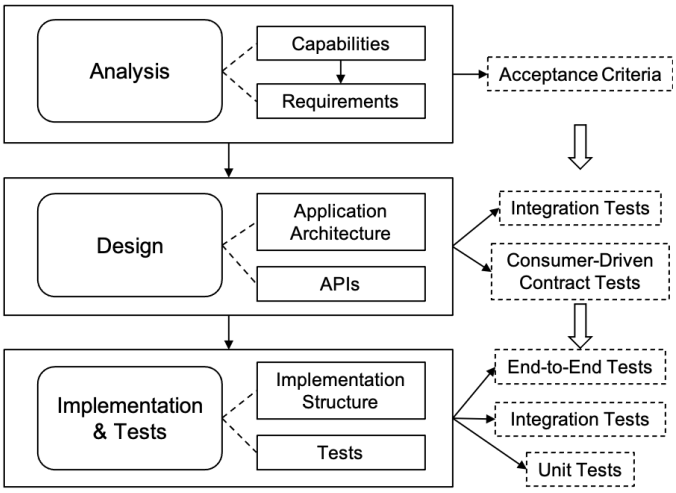

Figure 1. Development and Test Artifacts.

The design phase utilizes the artifacts from the analysis phase and forms the microservice architecture of the application by applying Domain-Driven Design (DDD) [6]. Important artifacts for the integration tests are the application architecture

 ISBN: 978-1-61208-894-5

and the API specifications of the microservices. The design artifacts, especially the API specification, are important for the CDC tests. The implementation phase utilizes the artifacts created in the analysis and design phase for the test implementation. The applicability of the test concept and the different tests is shown in detail by the example of a concrete microservice application, PredictiveCarMaintainance (PCM).

The main contributions of the article are: (i) a systematic test concept considering the test types of the test pyramid, E2E, integration and unit tests, extended with CDC tests and considering all test types during development; (ii) the integration of the test concept into a CI/CD pipeline.

The article is structured as follows: Section 2 presents the state-of-the-art in the area of testing microservices. Section 3 introduces the system under test (i.e., PCM) and the required artifacts. In Section 4, the test concept is introduced and explained by the example application PCM. The problem of test automation through the use of a CI/CD pipeline is tackled in Section 5. Results of the test concept are presented in Section 6. Section 7 summarizes the main results of our test concept and the main research issues we currently work on.

II. Related Work

Software tests are well introduced by several sources and placed into software engineering processes. O'Regan [7] provides an introduction to the field of software testing which contains a broad spectrum of related aspects, and further topics including software processes, and requirements engineering.

A reusable testing architecture is introducing by Rahman et al. [8] and proposes a dedicated application for automated acceptance testing. The concept provides separation of concerns among developers, testers and business analysts and is part of the test concept that is presented in this paper.

Savchenko et al. [9] provide a general testing process which extends the microservice development by several test steps, e.g., (internal functional) component testing, integration testing, and continuous system testing.

The conclusion that a microservice-based architecture requires more high-level testing especially on the end-to-end-side is discussed by Faragó et al. [4]. The reasoning behind this is that the interaction of microservices is the key to a working application.

John F. Smart [10] provides a more technical coverage of BDD practices and showcases a number of tools for different languages and frameworks, which aid developers in creating robust and sustainable tests. BDD can be seen as further development of Test-Driven Development (TDD) [11].

A case study was conducted to examine how a microservice-based application can be tested effectively by Lehvä et al. [12]. They do so by extending the traditional test pyramid with Consumer-Driven Contract (CDC) tests between integration and component tests. The study suggests that CDC tests could even replace integration tests, as they provide similar feedback, but only have a fraction of the development effort and execution time.

Wang et al. [13] present an API testing process which automatically gathers the API specifications from cloud websites and transforms the interpreted syntax and semantics of service data and operations into internal semi-formal representations from which the test cases are derived. This may be considered in further versions of the test concept.

Microservice-based applications require additional considerations during development because the applications are distributed and the services may be developed independently by different teams. Related work has influenced the result of the systematic microservice engineering approach that considers the entire test pyramid.

III. Application under Test

The application under test is the microservice-based application PredictiveCarMaintenance (PCM) which provides insight about a vehicle health. The application is developed using a systematic microservice engineering approach conceptualized specifically for the test concept.

During the requirements analysis, the cohesive functionalities are grouped into capabilities. The requirements of such a capability are described by User/System Interactions (USI) which are further represented as graphical USI flows. For acceptance testing, the end-to-end tests are systematically derived using Behavior-Driven Development (BDD) and the specified user interactions. Each step within the scenarios has a corresponding step definition, implemented during the development of the end-to-end tests. Furthermore, the scenarios describe the USIs under test. Smart [10] illustrates how unit tests can be derived from step definitions. Utilizing the approach, the application logic contained within a scenario can be developed in an iterative way.

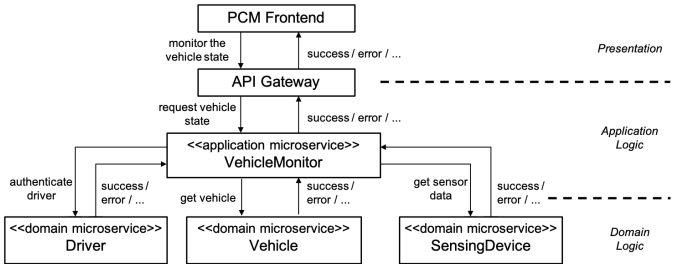

Figure 2. PCM Architecture Overview.

Figure 2 shows an overview of the derived architecture for the PCM application. The architecture was modeled during the design phase by applying Domain-Driven Design (DDD) concepts by Eric Evans [6]. The PCM application consists of the frontend, the API gateway, the application microservice VehicleMonitor, and the domain microservices Vehicle and Driver. Additionally, the application also communicates with the domain microservice SensingDevice. We differentiate between microservices which are only relevant for one application (the application microservices) and the application-agnostic microservices (domain microservices) which provide funcionality that can be reused by other applications. The

 ISBN: 978-1-61208-894-5

frontend of PCM allows the user to interact with the system and presents the information provided by the VehicleMonitor by requesting all data via the API gateway. The application microservice VehicleMonitor needs to authenticate the user by sending the corresponding requests to the microservice Driver. If the authentication is successful, VehicleMonitor gathers the required information by the domain microservice Vehicle and executes the application logic needed to support the USIs. To retrieve the sensor data, the VehicleMonitor communicates with the microservice SensingDevice. The microservices need to be orchestrated to fulfill the desired functionality. Therefore, the orchestration of the services for an application microservice is specified by task processes which describe a chain of service calls. The task processes itself are based on the concepts of the Business Process Executing Language (BPEL) [14] and the Service-oriented architecture Modeling Language (SoaML) [15].

The test concept is exemplary applied on the USI "Monitor the Vehicle State". The frontend calls the API gateway, which in turn calls the application microservice VehicleMonitor to receive the state of a vehicle. The application microservice first calls the microservice Driver for authentication and then calls the microservice vehicle for the information about a vehicle and its components. The detailed sensor data for each component is requested from SensingDevice. This information is used to derive a vehicle component's health state and the result is returned to the frontend.

IV. Test Concept

The development of tests follows a logical order, bottom-up. An overview of different types of tests used and the related artifacts is shown in Figure 3. As the development

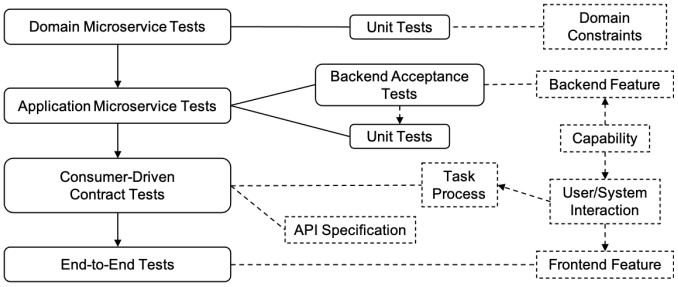

Figure 3. Overview of the Test Concept and Artifacts.

of microservice-based applications starts with the domain microservices, the tests for these microservices are created early. We differentiate between domain and application microservice unit testing, since domain microservices mostly contain simple functionality, i.e., CRUD operations. Therefore, corresponding tests are directly derived from the domain constraints. On the other hand, while implementing application microservices, two types of tests are developed: the backend acceptance and unit tests, which are derived from the former. After the implementation of, at least, one domain microservice, the development of one or more application microservices starts, including their tests, while deriving the functionality from the capabilities. When there are at least two microservices that communicate, Consumer-Driven Contract (CDC) tests can be applied, where the two most important artifacts are the task process and the API specification. The task process shows which microservices communicate and which data they access, whereas the API specification reveal how requests and responses are specified. The end-to-end tests form the highest layer of tests where great parts of the application under test are needed and are derived from the User/System Interactions (USI).

The BDD principle concentrates on the behavior and not on the concrete implementation of the software. The acceptance tests are written in the language Gherkin as features, that enable a common understanding of the software by using natural language. The Gherkin features formally specify the requirements of an application. The creation of those features involves a discussion between developers, testers and domain experts. The use of a ubiquitous language in those features helps additionally. For each capability, an application microservice is developed. The features are derived from the capabilities and contain scenarios comprised of steps. We derive the scenarios of a feature from the USIs which are further modeled as so-called USI flows. Figure 4 displays an example of the USI flow for monitoring a vehicle state.

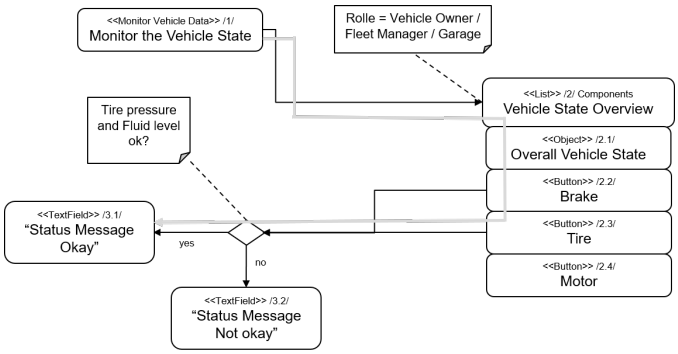

Figure 4. USI Flow for Monitor the Vehicle State.

Each path through a USI flow (one path is shown by the grey line) leads to a scenario. One of the resulting scenarios is shown in Figure 5.

```
1. Scenario: Monitor Component State (Success)
2. Given I am logged in as a vehicle owner,
       fleet manager or garage
3.    And the vehicle state overview is
          displayed
4. When I open the vehicle state overview for
       the motor
5. Then I see the detailed summary of the motor
6.    And a status message is displayed
```

Figure 5. Scenario for Monitor Component State.

 ISBN: 978-1-61208-894-5

A. Development of the Unit Tests for Domain Microservices

Due to the differences between domain and application logic, unit test development for domain microservices differs from the development of unit tests for an application microservice. Instead of application functionality, domain logic focuses on the application-agnostic domain logic which should be reusable by many applications from the same domain.

The domain knowledge of a bounded context can be expressed in an entity relation view which contains the domain objects and their relationships similar to a class diagram [16]. Figure 6 displays the entity relation view for the bounded context Vehicle. The entity relation view displays the entities Vehicle, VehicleComponent, Observation and Manufacturer. The vehicle is the most central domain entity. A vehicle consists of several vehicle components, such as brakes, tires and motor. These components are monitored by sensors. Sensors create observations that specify the time of the measurement and the observed measurement. Using the example of the method getObservationFromTimePeriod(), the constraints are defined and implemented in the following. This method makes it possible to display the observations of a sensor in a certain period of time.

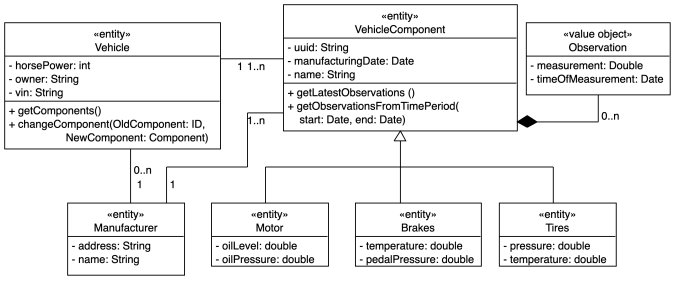

Figure 6. Entity Relation View of the Bounded Context Vehicle.

An observation is a measurement of a component value such as motor temperature at a specific time. The desired information may include only the latest observation or multiple observations for a specified time interval. The vehicle bounded context has no active influence on the observations itself. These observations are providedby a microservice SensingDevice.

The domain logic itself contains constraints which define the boundaries of the domain objects. The constraints of a domain are stated when the domain is modeled. This domain knowledge can be added formally to further specify the UML diagrams [17], e.g., the entity relation view of the bounded context vehicle, by the use of the Object Constraint Language (OCL). This leads to the advantage, that the model can be implemented and tested later. Constraints are derived from the domain knowledge (e.g., physical world constraints), gathered in the analysis phase and need to be enforced by implementation. Furthermore, the constraints need to be tested and therefore, are a valuable input for the unit tests of a domain microservice.

Figure 7 shows an excerpt of the constraints for the bounded context Vehicle. Using the method getObservationFromTimePeriod() and pre- and postconditions, the allowed transitions can be formally expressed with OCL. Lines 1-2 define the context and the considered method. In line 3, a precondition is specified. Here it is important that the end date of the observation is not before the start date. In lines 4-6, a postcondition is specified which ensures that the observations are not outside of the specified period.

```
1. context VehicleComponent::
       getObservationsFromTimePeriod(start:
       Date, end: Date):
2. pre: start.before(end)
3. post: forAll(o:Observation | (start.after(
       start)
4.     or start.equals(start))
5.     and (end.before(end) or end.equals(end)))
```

Figure 7. Excerpt of the Domain Constraints.

These constraints are implemented in the vehicle microservice's domain logic. Based on the underlying domain constraint the implementation of the method getObservationsForTimePeriod() is done, which is part of the VehicleComponent entity.

The pre- and post-conditions must be valid before and after the method is invoked, respectively. Therefore, each pre- and postcondition is checked through if statements. An example of a postcondition implementation is shown in Figure 8. If a condition is violated when a method executes, the method will throw an exception, which is an object that indicates that an error occurred.

Similarly, to guarantee that the requirements of the postconditions are met when the method has been called, the method uses an if statement in line 4. Only if all these conditions are met, the method returns a list with observations with regards to the specified time frame.

```
1. List<Observation> result = new ArrayList<>()
       ;
2. for (Observation o : this.observations) {
3.     ZonedDateTime t = o.getTimeOfMeasurement
           ();
4.     if((t.isAfter(start) || t.equals(start))
           &&
5.        (t.isBefore(end) || t.equals(end))) {
6.         result.add(o);
7.     }
8. }
```

Figure 8. Postcondition Implementation.

Test cases are derived from each constraint. There are two kinds of test cases and unit tests respectively: (i) the first kind asserts the method under test behaves as expected by feeding it with correct input and matching the output with expected output and (ii) the second kind asserts the input data is validated correctly by feeding the method under test with incorrect input and awaiting an exception to be thrown.

 ISBN: 978-1-61208-894-5

Test cases of type (i) use test data which conforms to all domain constraints. For the constraints presented in Figure 7, type (i) unit tests are going to test whether the correct set of observations is delivered as output by the method getObservationsFromTimePeriod(). Hence, two argument providers are presented in Figure 9 and 10. One serves as example for an arguments provider for unit tests of type (i) and the other - for unit tests of type (ii). It is good practice to develop one arguments provider class per unit test type.

ArgumentsProviderTypeI initializes the arguments for the test case (see lines 1-3) where the observations for the last month are requested. Those are a start date of one month ago, followed by an end date of today and the expected output which is provided by a separate class where expected result data is initialized or loaded from external files such as a test database or CSV table.

```
1. OutputProvider op = new OutputProvider();
2. ZonedDateTime start = ZonedDateTime.now().
     minusMonths(1);
3. ZonedDateTime end = ZonedDateTime.now();

4. return Stream.of(
5.   // Test Case 1: Last month
6.   Arguments.of(start, end, op.getOutput(1))
7. );
```

Figure 9. Class ArgumentsProviderTypeI.

The test data for type (ii) unit tests aims to violate the domain constraints. Each test case violates one specific constraint, thus accelerating the fault discovery process.

ArgumentsProviderTypeII initializes a test case that violates the time period constraint by having a start date after the end date (see lines 1-2). A minimal test suite must have at least one violating test case per domain constraint. Advanced test suites have multiple violating test cases.

```
1. ZonedDateTime start = ZonedDateTime.now();
2. ZonedDateTime end = ZonedDateTime.now().
     minusMonths(1);

3. return Stream.of(
4.    // Test Case 2: Time period violation
5.    Arguments.of(start, end)
6. );
```

Figure 10. Class ArgumentsProviderTypeII.

The resulting unit tests in Figures 11 and 12 receive the test data as a stream from the respective arguments provider class. The first unit test exemplifies unit tests of type (i). It receives input for the method under test and the expected output from the arguments provider. The expected output must abide by the constraints defined in the postcondition.

The second unit test is an example of type (ii) unit tests. Its arguments provider delivers the input and the unit test asserts

```
1. List<Observation> result =
2.    validTestComponent.
        getObservationsFromTimePeriod(start,
        end);
3. assertEquals(result, expectedResult);
```

Figure 11. Unit Test Type I.

```
1. assertThrows(IllegalArgumentException.class,
      () ->
2.    validTestComponent.
        getObservationsFromPeriod(start, end)
        );
```

Figure 12. Unit Test Type II.

that the proper exception is thrown. A method that throws multiple exceptions requires multiple type (ii) unit tests.

B. Backend Acceptance and Unit Tests of an Application Microservice

An application microservice is developed to support USIs for a specific capability. In order to ensure that the test suite exercises every bit of functionality developed for the capability, the BDD outside-in approach is adopted for the development of acceptance and unit tests for the application logic. First, the acceptance criteria is specified. Then, it is automated as backend acceptance tests. Unit tests are derived from the backend acceptance tests, whereby the behavior of the code is specified further. Finally, the application logic needs to satisfy the acceptance criteria and the tests are implemented.

The step definitions differ from those created for end-to-end tests in the way that UI step definitions manipulate frontend components (e.g., through page objects), whereas backend step definitions manipulate application code directly. Furthermore, these backend acceptance tests support the outside-in development approach, since unit tests can be derived from them. The implementation of a feature starts with the acceptance criteria and advances through the lower levels as illustrated by Figure 3. Backend Gherkin features are derived from the frontend step definitions. Figure 13 presents the backend scenario equivalent of the scenario in Figure 5.

There are multiple benefits from introducing backend acceptance tests. They provide assurance of the backend system functionality independent of the frontend. If an end-to-end test fails but its corresponding backend acceptance test is passed successfully, then the problem is located in the frontend. Backend acceptance tests execute faster than frontend acceptance tests, because UI slows down tests significantly [10]. Hence, testing the system without the UI layer allows for more tests in a shorter amount of time being both developed and executed.

The Gherkin features [11] are the central artifact for the testing of application microservices. For each of the (Given, When, Then) steps, the backend step definitions are specified by coding the function calls on the backend side. To fulfill the backend step definitions, the application logic is implemented

 ISBN: 978-1-61208-894-5

```
1. Scenario: Monitor Component State (Success)
2. Given the component with uuid "123..."
     exists
3. When the state of a component with uuid
     "123..." is requested
4. Then latest sensor information about the
     component is fetched
```

Figure 13. Scenario for Monitor Component State (Backend).

by writing the required unit tests in a first step and the application code to pass the unit tests in a second step. Smart [10] illustrates how unit tests can be derived from step definitions. Adopting this approach, in Figure 14, a step definition for the When step in Figure 13 is implemented. During the implementation of the step definitions, initial considerations for the application code are made. In the example, an operations class is modelled, which provides a method that fetches component information. The information itself is modelled as a list containing the various values provided by the domain microservice.

```
1. @When("When the state of a component with
     uuid <string> is requested")
2. public void request_component_info(String id
     ) throws Throwable {
3.   List componentInfo = operations.
       getComponentInfo(id);
4. }
```

Figure 14. Backend Acceptance Test Step Definition.

BDD treats unit tests as low-level executable specification, meaning the main focus is the behavior of the system, not the functionality of the separate methods. By following the method from Figure 3, this paradigm is enforced further. The unit tests are derived from the backend acceptance tests. Figure 15 illustrates the unit test derived from the When step in Figure 14. Infrastructural software units (e.g., database repositories, mappers, etc.) require unit tests as well.

```
1. public class StateOperationsTests {
2.    private StateOperations operations;
3.        ...
4.    @ParameterizedTest
5.    @ArgumentsSource(ArgumentsProvider.class)
6.    public void getComponentInfo_
7.    ShouldGatherComponentInfo(
8.       componentId, List expectedInfo) {
9.       assertThat(expectedInfo,
             samePropertyValuesAs(
10.          operations.getComponentInfo(
                componentId)));
11.   }
12. }
```

Figure 15. Unit Test Example.

C. Consumer-Driven Contract Tests

One of the main problems when dealing with a microservice-based application is the integration of microservices [18].

The main goal of integration tests is to find out whether changes break the application or not. For this, the affected services would need to be deployed which leads to slow tests. Testing the integration of microservices in an isolated way by using Consumer-Driven Contracts (CDC) can decrease the number of integrated tests and therefore decrease the duration of running all tests [18]. Those contracts document the communication between two services, where the caller of a service is called consumer and the callee is the provider. In this paper, the contract testing tool Pact is used for the CDC tests [19]. Pact offers implementations in many different programming languages, including Go and Java, meaning that Pact can directly be used for all of PCM's microservices. CDC tests in Pact consist of two steps: in the first step the contract is created by the consumer, by creating a Pact mock of the provider under test and specifying the expected response. In the second step, the previously defined request is sent to the provider and the real provider's response is compared to the expected response in the contract [20].

The needed contracts, where the microservice under test is the consumer, are derived from the task process of each of its microservice operation. An example for the microservice operation Monitor the Vehicle State is shown in Figure 16. Here, the microservice VehicleMonitor is the consumer and all the other microservices are providers.

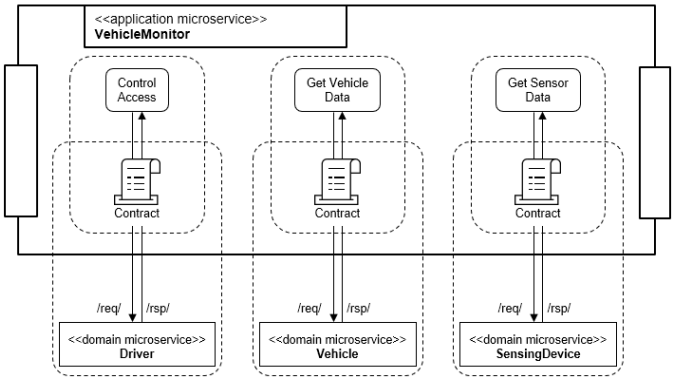

Figure 16. Contracts for the Microservice VehicleMonitor.

D. End-To-End Tests

The approach from [8] is adapted for the integrated tests, i.e., integration and end-to-end tests. As a result, a separate repository is used for those tests. Having those in each microservice repository would lead to high maintenance test suites, because step definitions cannot be reused across repositories. The test repository cannot access the internal code of the microservices, which means the whole application needs to be treated as a black box. Before the end-to-end tests can run, every required microservice needs to be deployed. The two

 ISBN: 978-1-61208-894-5

main questions to answer when end-to-end testing are what and how is tested. The end-to-end tests should not be used to reach a high coverage for all paths in an application, instead they should describe examples for the software's behavior [10]. To find such examples, USI flows are used. From a USI flow every path through the application for the considered user interaction can be derived. For the user interaction in Figure 4 twelve different paths can be found: two different states for both, brakes and tires and there are three different roles involved. Taking into account more components or adding additional decisions would drastically increase the number of application paths. This means, not every path should be mapped to a test, as many slow end-to-end tests would also slow down development [21]. As the user interaction is exactly the same for every role, it is sufficient to run this test only for one of the roles. This decreases the number of tests to four. By choosing one component for the test, only two test cases are left: (1) the component is okay and (2) the component is not okay. The data for the other component(s) as well as the access for the other roles can be tested in the integration tests. The resulting Gherkin feature is depicted in Figure 17. To increase readability both tests are combined into one scenario outline.

```
1. Feature: Monitor the Vehicle State
2.   As a vehicle owner, fleet manager and
       garage
3.   I want to see the overall state of a
       vehicle
4.   So that I can continuously monitor its
       state

5.   Scenario Outline: Display correct tire
       state
6.     Given I have opened PCM
7.       And I am logged in as a vehicle owner
8.       And the tire pressure is <tire pressure
           state>
9.     When I open the overview for the tires
10.    Then I see the tire state is <tire state>

11.    Examples:
12.      | tire pressure state | tire state |
13.      | not okay | not okay |
14.      | okay | okay |
```

Figure 17. Feature derived from the USI Flow.

In the following, the question of how the application should be tested is answered by using guidelines for creating end-to-end tests to ensure a good quality of the tests. Quality of tests has to be considered on two aspects: the test specification, i.e., Gherkin features, and the test implementation, i.e., the step definitions. End-to-end tests often have a high maintenance effort when they are written in an imperative way, because the features contain UI-specific or other irrelevant information. When the UI changes, both the step and its step definition need to change. To improve this, declarative features should be written, so a UI change would lead to a change only in the step definition, in case of a declarative feature [21]. In the example above, one could instead specify which exact inputs the user makes in order to log in. In this case, every time this user needs to login the step would need to update if the credentials would change.

Tests should provide feedback for the developers whether a change broke the application or not. When they need to wait very long for the test execution to finish, it affects the productivity. Moreover, not every edge case needs to be verified by an end-to-end test, those should be tested with unit tests [21].

Another problem that could occur in the test above could be inaccuracy. This problem is often indicated by imprecise language (e.g., "a user") or the use of the word "or".

Figure 18 shows a scenario that violates these guidelines. This scenario is inaccurate as it does not specify which user is logged in to the application (line 2). To get this information one would need to look into the step definition, which defies the purpose of BDD. The same applies to lines 4 and 5, where the concrete component is not specified.

```
1. Scenario: Monitor the Vehicle State (Success
     )
2. Given I am logged in as a vehicle owner,
     fleet manager or garage
3.   And The vehicle state overview is
       displayed
4. When I open the vehicle state overview for a
     "component"
5. Then I see the detailed summary of the "
     component"
```

Figure 18. Example of a Flawed Scenario.

An improved version of this scenario is displayed in Figure 19. It is now clear which user is logged in for the test case and which component is viewed. If the scenario needs to be tested for the other roles as well, this can be easily accomplished by using a scenario outline.

```
1. Scenario: Monitor the Vehicle State (Success
     )
2. Given I am logged in as a vehicle owner
3.   And the vehicle state overview is
       displayed
4. When I open the vehicle state overview for
     the motor
5. Then I see the detailed summary of the motor
```

Figure 19. Example of an Improved Version of the Scenario.

Software should be easy to change, therefore especially the end-to-end tests should be robust against changes, as they take a lot of time to implement [10]. To realize this, the Gherkin features should not contain implementation details that are prone to changing and leave out irrelevant information. When an application's implementation changes, only the corresponding step definition needs to be updated, the step can remain the same.

 ISBN: 978-1-61208-894-5

For the automation of the Gherkin features, step definitions need to be written. For those there are also guidelines defined to support automated testing. To increase maintainability, useful selectors should be chosen. A poor selector is one that is likely to change and is difficult to understand, an example is XPath. It is recommended to use IDs or similar attributes. By using the page object model maintainability can be even further increased. The page object model implements all interaction with the applications into classes called page objects. The page objects hide UI details from the test code [22].

One common problem for automated end-to-end tests are tests that sometimes pass or fail without an apparent reason. The reason could be race conditions. In web applications many things happen asynchronously, so often the order in which calls return cannot be known beforehand. A quick fix could be using fixed-length waiting times. This is not a suitable solution, as this increases the test execution time and on the other side it does only decrease the possibility of a race condition. Conditional waiting times should be applied instead [21].

A similar problem could occur when tests change persistent state, but do not reset it. In this case the success of a test would depend on the execution order. These side effects can lead to false negative test results, i.e., the functionality works, but the test fails. To prevent this, such a state should always be reset.

V. Pipeline Integration

The pipeline considers all types of tests that are used. Each microservice's repository includes all of its isolated tests (i.e., unit and Consumer-Driven Contract (CDC) tests), the integrated tests (i.e., integration and end-to-end tests) are stored in a dedicated test repository, following the approach of the reusable automated acceptance test architecture from [8], where the end-to-end tests are extracted into their own repository. An example of executed pipeline jobs on a change in the microservice VehicleMonitor is shown in Figure 20. On a commit in a repository all of its pipeline stages are executed, starting with the unit tests. The next tests that are run are the CDC tests. These tests are split into separate pipeline jobs ConsumerContract and ProviderContract. In the job ConsumerContract new contracts are created or existing ones are updated by sending them to the Pact broker. If contracts are changed in this stage, the affected providers are tested by running their pipeline through Pact's webhooks, where only the provider tests are executed. Therefore, a pipeline trigger token is created in the corresponding providers' repositories to start their pipeline from Pact. This also enables differentiating how the pipeline was started. After the VehicleMonitor's consumer tests are finished, its provider contracts are retrieved from the Pact broker and the microservice is tested. If the jobs for unit and contract testing were successful, the new version of the changed microservice is deployed. This will trigger the test repository pipeline in the pipeline job Downstream with its two jobs that run integration and end-to-end tests.

One of the most important aspects of a continuous integration pipeline is to have a short build time [23]. This leads to

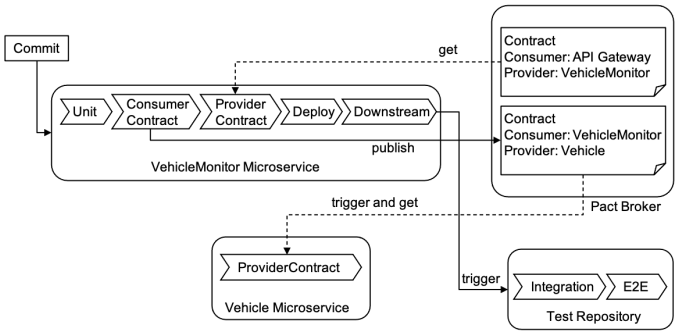

Figure 20. Pipeline for a Change in the VehicleMonitor Microservice.

quick feedback about breaking changes to the developers. One way to decrease the pipeline's build time is to test as much as possible on the lower levels of abstraction [24], without external dependencies that slow down testing. Typically, integration tests are used to test the communication between microservices, that often need to be deployed beforehand, and are therefore slow [9]. The communication between the microservices is tested with the much faster Consumer-Driven Contracts (CDC) that can greatly reduce the needed integration tests by replacing those [19]. The CDC tests are more stable than the integration tests as less moving parts are involved [12]. Error localization is simplified as in a CDC test only two services are considered at a time. By stopping pipeline execution on every failure and running the tests ordered by execution time, feedback times are additionally reduced: if there is an error in the unit tests this functionality will not work in the tests of higher level. Another important feature of a continuous integration pipeline is to run tests in a clone of the productive environment [23]. There are two deployment environments, called test and prod. Both deployment environments mirror the contents of one branch each. The test environment contains the state from the corresponding branch develop, and the contents of the branch master get deployed to the prod environment. As only those two branches get deployed, the integration and end-to-end tests are only applicable to those.

```
1. .no-trigger-token:
2.   rules:
3.     - if: '$CI_PIPELINE_SOURCE != "trigger"'

4. unit:
5.   extends: .no-trigger-token
6.   script: ...
```

Figure 21. Pipeline Configuration.

As GitLab is used, which only permits one pipeline per repository, a solution needed to be found that allows this structure with one single pipeline. Additionally, a goal was decreasing duplicates in this pipeline configuration as much as possible. This was achieved by the use of hidden pipeline jobs that contain the needed rules for all pipeline jobs and extend those as described in [25]. An excerpt for the pipeline

configuration is shown in Figure 21.

The job .no-trigger-token is hidden and contains a rule to execute a job only if the pipeline was not started by a trigger. There exists another hidden job, which contains the rule to make sure a job is only executed when the current branch corresponds to one of the two deployment environments, i.e., master and develop, and when the pipeline was not started by a trigger token. This hidden job is extended by all jobs that require a deployed microservice, i.e., ConsumerContract, ProviderContract, Deploy and Downstream.

VI. Results of the Test Concept

In the context of unit testing, the domain constraints provide a structured approach. These constraints describe in a formal way, which values for attributes or parameters for method calls are permissible, before starting the implementation. Furthermore, the constraints are used as a reference during the implementation and writing of unit tests. This has the advantage that during the implementation the already defied edge cases (within the constraints) can be used. Because the constraints are created separately from the implementation, the unit tests are correspondingly less influenced by the implementation. The constraints provide a golden thread, which is helpful when writing the unit tests.

Constraints can be efficiently utilized to create test cases for edge cases. Edge cases are on one side of the constraint and thus cover the area of the constraint. As a result, test cases which do not increase the test coverage, are minimized.

During the implementation of the domain microservice Vehicle, we applied the test concept. The structured approach and the domain constraints were helpful for the developers. This is because the systematic procedure by the test concept subdivides the unit testing of the domain microservice in several tasks. Therefore, we were able to assign these tasks among us appropriately and thus work on some tasks in parallel. When writing the unit tests, it was still an open topic which test data should be used for the tests. Furthermore, these can be systematically derived from the other existing artefacts of the domain microservice.

The test concept distinguishes between domain and application microservices. Therefore, the test concept with its artifacts supports the scope of the respective microservices. For example, the focus of an application microservice is more on testing the behavior (e.g., through Gherkin features). This was an advantage during the development of the application PCM because a more targeted procedure to writing unit and integration tests is possible.

Moreover, by shifting the tested functionality to tests of lower layers whenever possible, the development time of the tests as well as their execution time is minimized. Especially CDC tests are important in a microservice-based application to verify that different parts of the tested application can communicate.

The guidelines for end-to-end tests (E2E) can help improving the maintainability of those tests, by enabling faster development and reducing test cases as much as possible. The application of the guidelines leads to more stable tests.

The introduction of the uniform pipeline structure presented in this paper will simplify the application of Continuous Integration / Continuous Deployment, especially when it is used as a template, where only the microservice-specific scripts need to be customized.

VII. Conclusion and Future Work

We have introduced a test concept for the development of microservice-based applications and showed its applicability by the example of an excerpt of the microservice-based application PredictiveCarMaintenance (PCM). One of the main goals was to systematically test the application by using the artifacts and different test types (end-to-end, integration, consumer-driven contract, unit tests) and assist the developers in this process. By starting with the domain and its constraints, we test the domain microservices with unit test by deriving the test from the constraints. Testing the application microservice and its operations is done by a systematic derivation of backend acceptance tests which are transferred to unit tests. Next, we test the integration of the microservices with consumer-driven contract tests to test the microservices in an isolated way. Finally, end-to-end tests are developed using the guidelines provided to test the whole application at once. BDD simplifies the communication with the stackholders. Overall, the test concept provides what should be tested with which tests.

In addition, we integrated the different tests into a CI/CD pipeline and described how the different pipelines need to be triggered. The pipeline approach can be reused for further projects with minor configuration adjustments. As a result, we are convinced that applying the test concept leads to well tested microservice-based applications with a small effort. At the same time, the development of the application is simplified.

A further point to look at is that the test issues a deterministic request to the system under test and expects a predefined output or response from the system. A request usually requires concrete values and parameters. Therefore, a suitable selection of the test data is necessary, for example, to cover edge cases.

In addition, a systematic representation of the test data needs to be researched. The goal here is to arrange the test data into an orderly format for the representation of the test data. One goal of further research is to assist the developer with even more support for writing tests by providing additional guidelines for applying the test concept. This includes optimizations and enhancements of the presented test concept. In the future, the test concept needs to be applied to more applications which may lead to further insights to adapt the approach.

Moreover, the versioning of the CDC tests needs to be revised in the future, as the tests are currently only executable in master and development branches. The developers can benefit from guidelines that will be created in the future and can simplify the development of CDC tests.

 ISBN: 978-1-61208-894-5

REFERENCES

[1] N. Alshuqayran, N. Ali, and R. Evans, "A Systematic Mapping Study in Microservice Architecture," in *9th International Conference on Service-Oriented Computing and Applications (SOCA)*. IEEE, 2016, pp. 44–51.

[2] R. T. Fielding, "Rest: Architectural styles and the design of network-based software architectures," Ph.D. dissertation, University of California, 2000.

[3] A. S. Bueno, A. Gumbrecht, and J. Porter, "Testing Java Microservices: Using Arquillian, Hoverfly, AssertJ, JUnit, Selenium, and Mockito," 2018.

[4] D. Faragó and D. Sokenou, "Keynote: Microservices Testen Erfahrungsbericht und Umfrage," *Test, Analyse und Verifikation von Software (TAV) der Gesellschaft für Informatik (GI), Stuttgart*, 2019.

[5] B. Hippchen, P. Giessler, R. Steinegger, M. Schneider, and S. Abeck, "Designing Microservice-Based Applications by Using a Domain-Driven Design Approach," in *International Journal on Advances in Software*. IARIA, 2017, pp. 432–445.

[6] E. Evans, *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Addison-Wesley Professional, 2004.

[7] G. Regan, *Concise Guide to Software Testing*. Springer, 2019.

[8] M. Rahman and J. Gao, "A Reusable Automated Acceptance Testing Architecture for Microservices in Behavior-Driven Development," in *IEEE Symposium on Service-Oriented System Engineering*, 2015, pp. 321–325.

[9] D. Savchenko, G. Radchenko, T. Hynninen, and O. Taipale, "Microservice Test Process: Design and Implementation," in *International Journal on Information Technologies and Security*, 2018, pp. 13–24.

[10] J. F. Smart, *BDD in Action*. New York, NY, USA: Manning Publications, 2015.

[11] D. North, "Introducing BDD— Dan North & Associates," 2006.

[12] J. Lehvä, N. Mäkitalo, and T. Mikkonen, "Consumer-driven contract tests for microservices: A case study," in *International Conference on Product-Focused Software Process Improvement*, 2019, pp. 497–512.

[13] J. Wang, X. Bai, H. Ma, L. Li, and Z. Ji, "Cloud API Testing," in *10th IEEE International Conference on Software Testing, Verification and Validation Workshop (ICSTW)*, 2017.

[14] M. B. Juric, "A Hands-on Introduction to BPEL," *Oracle (white paper)*, p. 21, 2006, [retrieved 30/08/2021]. [Online]. Available: https://www.oracle.com/technical-resources/articles/matjaz-bpel.htm

[15] B. Elvesæter, A.-J. Berre, and A. Sadovykh, "Specifying Services using the Service Oriented Architecture Modeling Language (SoaML) - A Baseline for Specification of Cloud-based Services," in *CLOSER*, 2011, pp. 276–285.

[16] M. Schneider, B. Hippchen, P. Giessler, C. Irrgang, and S. Abeck, "Microservice Development Based on Tool-Supported Domain Modeling," in *Conference on Advances and Trends in Software Engineering (SOFTENG)*, 2019.

[17] Object Management Group, "Object Constraint Language," [retrieved 30/08/2021]. [Online]. Available: https://www.omg.org/spec/OCL/2.4

[18] S. Newman, *Building Microservices: Designing Fine-grained Systems*. O'Reilly Media, Inc., 2015.

[19] Pact, "Introduction," [retrieved 01/09/2021]. [Online]. Available: https://docs.pact.io/

[20] ——, "How Pact works," [retrieved 30/08/2021]. [Online]. Available: https://docs.pact.io/getting_started/how_pact_works

[21] M. Wynne and A. Hellesøy, *The Cucumber Book: Behaviour-Driven Development for Testers and Developers*. Pragmatic Bookshelf, 2012.

[22] M. Fowler, "PageObjects," https://martinfowler.com/bliki/PageObject.html, 2013.

[23] ——, "Continuous Integration," https://martinfowler.com/articles/continuousIntegration.html, 2006.

[24] H. Vocke, "The Practical Test Pyramid," https://martinfowler.com/articles/practical-test-pyramid.html, 2018.

[25] GitLab, "Keyword reference for the .gitlab-ci.yml file: Extends," [retrieved 30/08/2021]. [Online]. Available: https://docs.gitlab.com/ee/ci/yaml/#extends

 ISBN: 978-1-61208-894-5