Phased Testing

Testing Systems Undergoing Change

Baubak Gandomi

gandomi@adobe.com / development@gandomi.com

Abstract

In this paper we analyze the challenges involved in verifying and testing the success of system changes. These operations are often overlooked when testing, however as systems are moving more and more to continuous delivery, such tests become increasingly relevant. Some the more common system changes are:

- Database changes
- Application upgrades
- Server and Cloud changes

In our experience, validating that a system works after a system change is very complex. From the outset, two initial challenges stand out:

- How do we make sure that the problem is related to the change, and is not inherent to the product?
- How can a problem be clearly identified and reproduced?

In our view, traditional test approaches have limitations when testing for change. Instead, we present a new approach to writing tests that allows us to completely test how a use case is affected by a system change. We call this approach Phased Testing. Phased Testing allows our test scenarios to be leveraged for validating system changes. This is done by empowering our scenarios in two ways:

1. We allow test scenarios to be interruptive.
2. A test scenario is declared only once, but we are able to execute it with all the possible interruptions it may be subjected to.

Biography

The author, Baubak Gandomi, studied at the Mid-Sweden University, Universität Leipzig (Germany) and finally graduated from the university of Stockholm, Sweden with a degree in Computer Sciences.

He is currently a test automation architect at Adobe, a position he has held for the last 9 years. Apart from upgrade tests he is very interested in continuous delivery methodologies, cross-product/teams testing and finding measurement methods for assessing functional and transactional coverage.

When not solving Quality and Test related problems, Baubak practices the martial art of Ninjutsu, and promotes the ugliness of the world through his blog, Ugly City Guide.

1 Problem Statement

Given the popularity of continuous delivery processes, it is becoming increasingly important to ensure that application changes do not cause a loss of service. Some of the more common system changes are:

- Database changes
- Application upgrades
- Server and Cloud changes

The traditional migration scenarios usually are a combination of two cases:

1. System changes on a system with existing data generated by executing old scenarios,
2. The execution of scenarios on a changed system.

Our experience is that validating that a system works after a system change is quite hard. We see the following challenges:

- How do we make sure that the problem is related to the change, and is not inherent to the product?
- How can a problem be clearly identified and reproduced?

1.1 Complicated Data

Traditional testing is insufficient because it often does not consider the data prior to the system change. Systems performing such tests usually manage a database that is regularly updated. Managing such databases is quite complex and is hard to scale. We identify the following problems:

- Mapping data to a specific test scenario is complex
- Mapping data to a specific product/application version is complex
- Updating the data is cumbersome.

1.2 The Time Problem

Most test scenarios imagine a static system. They will perform a set of actions and verify that the end result is as expected. There are however, two problems with the traditional approach:

1. User Transactions are not usually atomic. The way we use services is rarely a sequence of events with a predefined time interval between each action.
2. We are relying more and more on remote services of various kinds. Services can be updated while you are in the middle of a transaction. This is especially true for systems using continuous delivery.

This has the consequence that a system change can happen during a transaction.

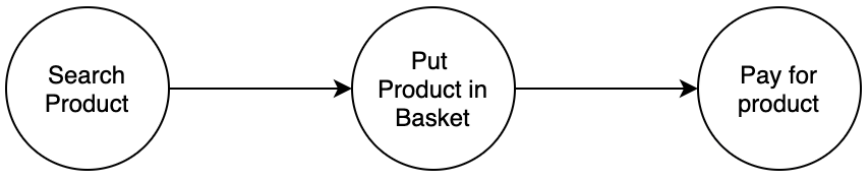

Figure 1 : A simple e-commerce scenario

An example is a simple retail site. As users, we will be performing searches, adding items to our basket, and finalizing the transaction. It is rare to perform all the steps in one go.

We usually add items to our basket, and even wait a few days before finally deciding on buying the goods we have selected. The full transaction, in reality, lasts a few days. System changes can arise at any point.

Figure 2 : What can happen between actions

Testing the example above would require us to first allow the scenario to be interrupted. Testing it in relation to a system change will require us to rerun the scenario four times in order to cover all of the possible cases within the transaction. This can be seen in Figure 3 below.
Traditional test frameworks were originally created for writing unit tests, and, because of this, they do not intuitively incorporate events or interruptions during the execution of a scenario.

In our first attempt, we were able to introduce interruptions in our test scripts by using conditional statements. We discovered that writing the scenario above would require us to rewrite the original scenario at least twice. This level of code copying grows with the number of steps. It introduces a complexity and a redundancy proportional to the number of steps.

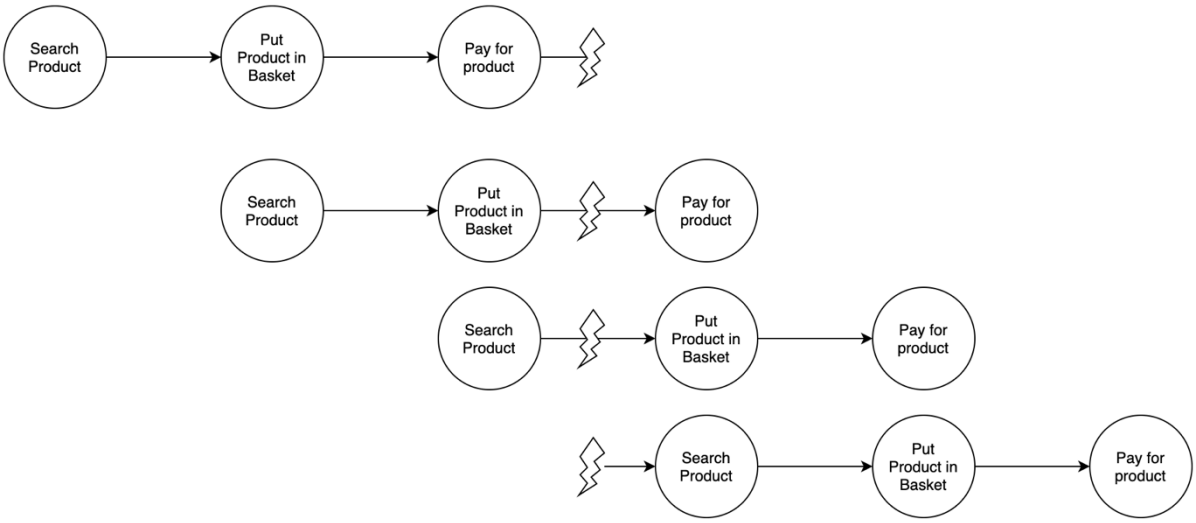

Figure 3 : The simple scenario regarding system changes. (The lightning sign indicates an interruption.)

For example, the scenario in Figure 3 will require us to rewrite the original scenario in Figure 1 at least twice. Any subsequent changes to the original scenario will have to be carried over to the copied versions. We have added an example of this in Appendix A.

2 Phased Testing

We developed Phased Testing to address the problems highlighted in the previous section.

Phased Testing allows our test scenarios to be leveraged, when needed, for validating system changes. By their very nature they also address the problem of test data. In this section we will be presenting the main concepts and building blocks behind this concept.

We have tested this approach by implementing it in the TestNG test framework. We will describe the implementation later in the section on Implementation.

2.1 Phases of a Test

When a test validates the correctness of a change, also called a “Phased Event”, we make a distinction between the steps executed before and after a system change. Each group of steps is executed in its own “phase”:

- **Producer**: The steps executed before a system change.
- **Consumer**: The steps executed after a system change.

The choice of the names is regarding the data being generated by the steps. The steps in the “producer phase” will be generating data. The steps in the “consumer phase” depend on the data generated in the producer phase.

Another way to look at it is that the steps in the producer phase will put the user of the scenario in a state, in which we expect them to find themselves after a system change.

A test scenario may also be executed outside of the notion of a “Phase”. In this case, the scenario is executed like any other test scenario without interruption. This has the advantage that you do not have to write dedicated scenarios. Your scenarios will be used on a regular basis for testing the normal functioning of your application. It is only when you decide to execute the in phases that it will be interrupted.

2.2 Approach

The guiding principle in our approach is that test scenarios should create the data they need. This may not be new, but it is an approach we find necessary for the solutions we present here.

What we propose is to assume a different approach to writing tests. We present two main paradigms:

1. A test scenario should be interruptible.
2. A test scenario is declared only once, but we should be able to execute it with all the possible interruptions it may be subject to.

To make our point, let us imagine a simple scenario with three steps:

Figure 4: Our scenario presented in steps

The first paradigm is that we expect that the scenario to be paused at any time along the timeline of its execution. In the example above, it should be possible to interrupt the scenario at any point around steps 1, 2 and 3. The tester should be able to pause a scenario at any step and carry-on where you left off whenever you want. This constitutes what we call a **Phased Test**.

The second paradigm states that we define a scenario only once, but it can either be executed as a normal scenario without interruptions, or it can be executed with all the possible interruptions. This execution mode is called "**Shuffling**".

2.3 Execution Modes

A phased test needs to define how it is executed when it is in a phase. This is called execution mode. We currently identify two execution modes:

- Shuffle Mode: The test is executed with all the possible interruptions it may encounter.
- Single Run Mode: The test is always interrupted at the same step.

2.3.1 Shuffled Mode

Shuffled execution will execute the scenario with all the possible interruptions. A scenario with 3 steps will be executed 4 times. Each execution instance, uniquely identified by the interruption point, is executed within a context we call a "Shuffle Group". Henceforth we will use the notion Shuffle Group to designate the execution instance of a phased test scenario.

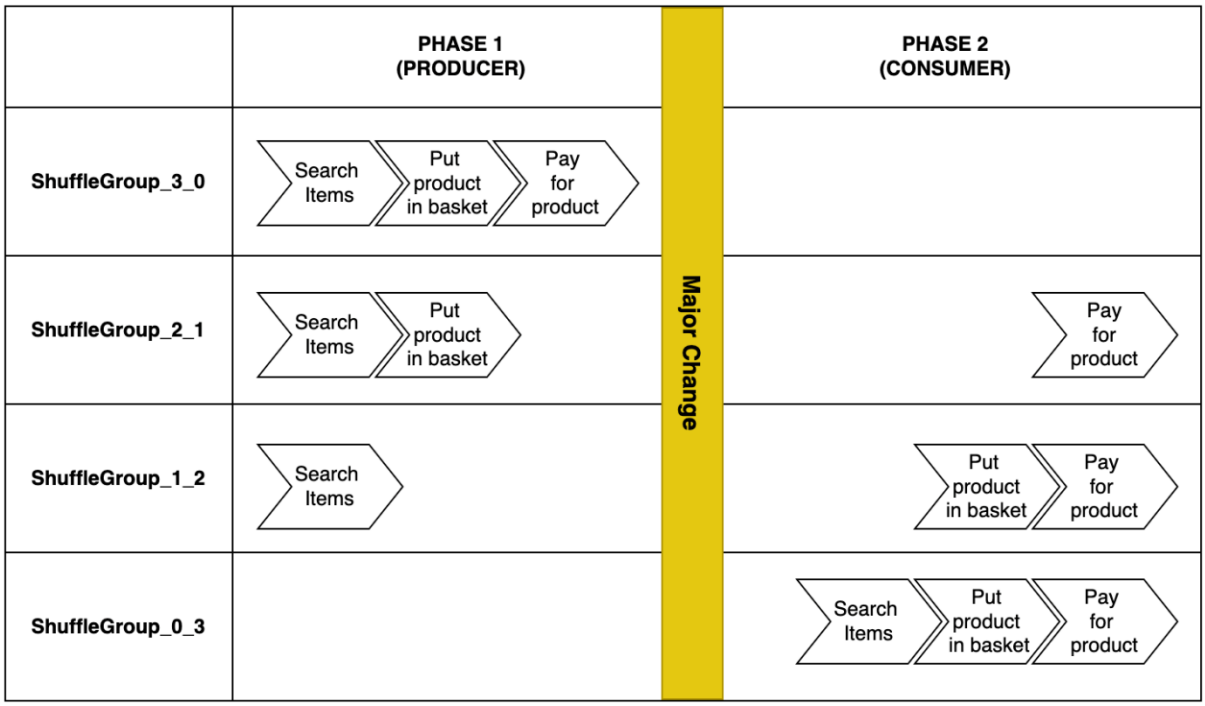

Figure 5: Shuffled Execution Mode

In Figure 5, although we have four executions, they remain the same scenario as in Figure 4: Our scenario . After the interruption, the test will continue where it left off.

We execute the scenario without interruption before and after the system for the following reasons:

- We can see if the scenario was working before the change.
- We generate data that will be used to test the change itself.
- We can see if the scenario can be executed completely after the system change.

2.3.1.1 Shuffle Groups

As mentioned earlier, each execution of a scenario with its interruption is uniquely identified by an execution context we call a “Shuffle Group”. This allows us to differentiate between the executions and to highlight the step in which the interruption occurred within the scenario execution.

We uniquely identify the Shuffle Group by the number of steps that are executed in each phase. For example, a scenario with 3 steps that is executed in Shuffle Mode, will yield the Shuffle Groups:

- 3_0
- 2_1
- 1_2
- 3_0

In the Shuffle Group 2_1, we know that the interruption occurs after the second step, whereas in Shuffle Group 1_2 the interruption happens between steps 1 & 2.

The number of Shuffle Groups in a Phased Scenario when executed in a Shuffled Mode is:

$$Number\ of\ Shuffled\ Groups = Number\ of\ Steps + 1$$

2.3.2 Single-Run Mode

Some use cases and scenarios are static in nature. By this we mean that the scenario will always be interrupted at the same step. The typical use case is when the interruption is due to a time-consuming process, like the heavy processing of data or the dependency on an external service. This requires an interruption at a specific step that will not change.

When a scenario is marked as “Single-Run”, it will only be executed once, and always interrupted at the same step.

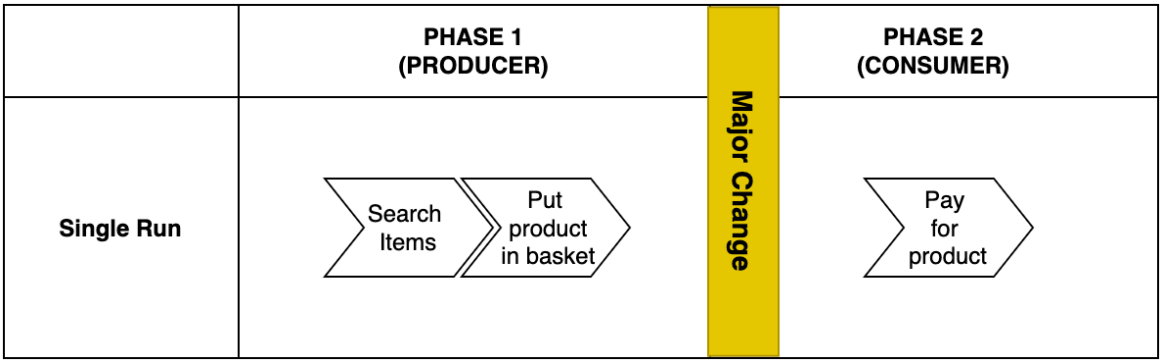

Figure 6: A single-run execution context

In Figure 6, the test scenario will always be interrupted after we put the product in the basket, and after the system change it will continue with the last step.

One of the main differences between the Shuffled, and the Single-Run modes is that in many use cases for the Single Run, the scenario cannot be executed in a non-phased mode.

2.4 Maintaining the Context

We need to set some mechanisms to ensure that Shuffle Groups can carry on wherever they are interrupted. These mechanisms concern the management and communication of data in Phased Tests. To implement these mechanisms, we need to enable two types of data communication in the system:

- Cross-Step Communication
- Cross-Phase Communication

One of the challenges when executing tests in Shuffle Mode is that the scenario is executed multiple times. We have introduced a notion of scope that requires that all data generated and retrieved are only visible within their Shuffle Group.

2.4.1 Cross-Step Communication

We need the steps to be able to carry on where the previous step left off especially if we have had to interrupt the tests to update the system. This means that they need to have access to, and to recognize the data generated in previous steps. Since our implementation of the Phase Testing anticipates an interruption in the test execution we cannot rely on global variables to pass information between steps. Consequently, each step stores the values needed for the following steps in a cache.

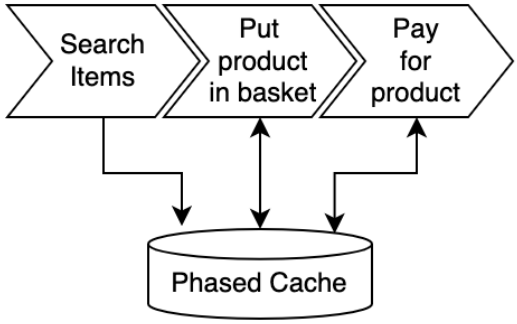

Figure 7: Communication between steps

We use two methods in a scenario to explicitly manage data in the Shuffle Group context:

- **Produce** allows a step to store data for future steps
- **Consume** allows a step to access data stored in previous steps

Because early steps will produce data, which is later consumed by the following steps, it is important to have a predictable order of execution between the steps. This is achieved differently based on the language and the used test drivers. We will describe our method in the section on Implementation and Defining a Phased Test.

The Phased Testing framework must also store information regarding the state of the Shuffle Group. This includes information such as:

- The current state of the Shuffle Group.
- Errors in previous steps.
- Duration information.
- The phase in which the error occurred.

This implicitly stored information allows us to represent the Shuffle Group as a whole test execution, where the results reflect the whole execution, and the duration is that of he full execution of the Shuffle Group.

2.4.2 Cross-Phase Communication

We do not make a big distinction between steps in the same or different phases. Like in the case of intra-step communication, the steps following a failure will be skipped for that context.

There is one main difference, however. At the end of the “producer phase” the cache is exported to a file, which is later consumed when the tests are executed in the “consumer phase”. The consumer phase imports the cache file and makes the data available to the phased tests in that phase.

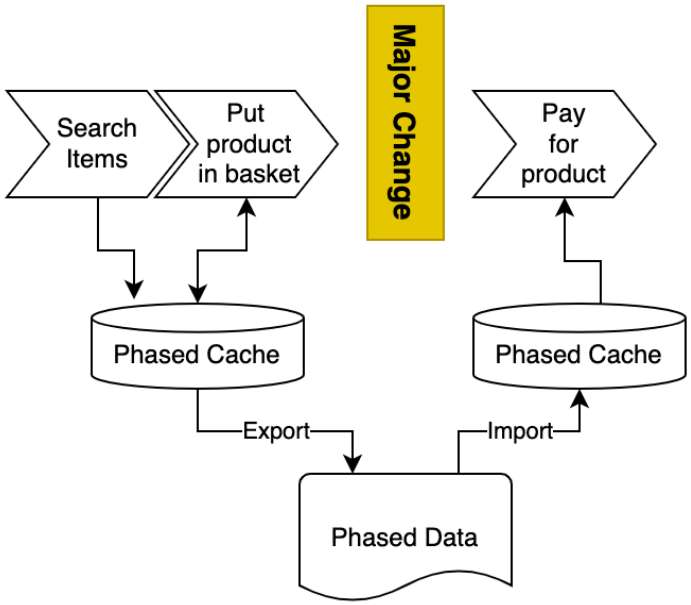

Figure 8: Communication between steps in different phases

3 Implementation

We implemented the Phased Test approach using the TestNG framework. The Data Provider notion used in TestNG, is particularly helpful, as it allows us to dynamically create a context in which all the steps of a Phased Test can be executed.

The Phased Test framework performs the “shuffling” when needed and interrupts the Shuffle Groups at the expected locations.

In our current implementation of the framework, the framework does not perform the system change itself. In our model and implementation, this is orchestrated by an automation/build system. We will cover more on this this aspect in chapter 3.4 on Orchestration. The system change, however, can be driven by the tool that is called from within this framework, but we do not include this aspect in this article.

3.1 Defining a Phased Test

To implement Phased Testing we had to review the notion of a test in the test frameworks. In most frameworks the test is a method. However, in our implementation of Phased Tests, the test is a class. Each method in the class is a test step.

We declare a test as a Phased Test by setting the annotation **@PhasedTest** on the test class. By setting the annotation, we let the system know if the test should be executed in Shuffle Mode or not:

```
@Test
@PhasedTest(canShuffle = true)
public class ShoppingBasket {

    public void step1_searchForProduct(String val) {
        //Perform actions
    }

    public void step2_addToShoppingBasket(String val) {
        //Perform Actions

        //Perform Assertions
    }

    public void step3_payForProduct(String val) {
        //Perform Actions

        //Perform Assertions
    }
}
```

Code Example 1: A simple Shuffled Test Scenario

In Code Example 1, when executed in phases, the scenario “ShoppingBasket” will be executed 6 times and each time will be interrupted at a different step.

As described in the section Cross-Step Communication, the order needs to be predictable as earlier steps produce data with is consumed in the following ones. Since in TestNG the tests are executed in an alphabetical order we use a nomenclature of prefixing the steps with their step numbers in our scenarios.

If we are in a Single-Run Mode, we set an annotation **@PhaseEvent** at the location where we expect the interruption to occur. The Phase Test framework will thus execute all the steps up to that point in the producer phase, interrupt the scenario, and, once restarted, continue with the steps following that annotation.

```
@Test
@PhasedTest(canShuffle = true)
public class ShoppingBasketSingleRun {

    public void step1_searchForProduct(String val) {
        //Perform actions
    }

    public void step2_addToShoppingBasket(String val) {
        //Perform Actions

        //Perform Assertions
    }

    @PhaseEvent

    public void step3_payForProduct(String val) {
        //Perform Actions

        //Perform Assertions
    }
}
```

Code Example 2: A simple Single Run Test Scenario

In Code Example 2, the scenario “ShoppingBasketSingleRun” is executed only once. In the Producer phase the first two steps, step1_searchForProduct and step2_addToShoppingBasket, are executed. In the Consumer phase only the last step, step3_payForProduct will be run, i.e. the interruption will occur before the method annotated with @PhaseEvent.

3.2 Storing Data Between Steps

As mentioned in the section Cross Step Communication, we store data both explicitly in the scenario, but also implicitly through the framework.

In the scenario, variables are stored and accessed using methods called **produce** and **consume**.

```java
@Test
@PhasedTest(canShuffle = true)
public class ShoppingBasket2 {

    public void step1_searchForProduct(String val) {
        //Search for product

        //Store value with key
        PhasedTestManager.produce("FoundProduct", "book A");
    }

    public void step2_addToShoppingBasket(String val) {
        // Fetch searched product
        String searchedProduct = PhasedTestManager.consume("FoundProduct");

        //add searchedProduct to basket

        //Store basket ID
        PhasedTestManager.produce("BasketID", "1");
    }

    public void step3_checkout(String val) {
        //Fetch basket ID using key
        String basketId = PhasedTestManager.consume("BasketID");


        //Fetch searched product
        String searchedProduct = PhasedTestManager.consume("FoundProduct");

        //Assertion
        //Check that the product is in the basket

        //Checkout
    }
}
```

Code Example 3: Passing data between scenario steps

In Code Example 3 above, we first search for the product “book A”, and we store the value by calling “produce” and using the key “FoundProduct”. This is later consumed in step2_addToShoppingBag & step3_checkout. We consume the value “book A”, by referring to the key with which it was stored, i.e., “FoundProduct”.

As mentioned in the section Maintaining the Context, the data we store using produce and consume are only visible to the Shuffle Group in which they are executed.

There is a risk for conflicts with the data being created when running tests in Shuffle Mode. If the data being created is hard coded, then the other Shuffle Groups of that scenario will be creating the same data. We think that using random data is usually a good solution to avoid these issues.

In the Code Example 3: Passing data between scenario steps, when executed in Shuffle Mode, as described in the section on Shuffle Groups, yields the following Shuffle Groups:

- 3_0 The scenario is run with the change at the end.
- 1_2 The scenario is run with the system change in the middle between steps 1 & 2.
- 2_1 The scenario is run with the system change in the middle between steps 2 & 3.
- 0_3 The scenario is run after the system change.

When “step1_searchProduct” of the scenario is executed in the Shuffle Group 0_3, the value “FoundProduct” is stored as property with the key:

Scenario name + storage key + Shuffle Group

The value “book A” is thus stored with the key:

demo.ShoppingBasket2(phased-shuffledGroup_3_0)->FoundProduct=book A.

This way of naming the values we store allows the data to be visible only in its Shuffle Group. This is done by s using the Shuffle Group name in the data we generate. Below is an excerpt for the data as it is stored:

```
demo.ShoppingBasket2(phased-shuffledGroup_3_0)->FoundProduct=book A
demo.ShoppingBasket2(phased-shuffledGroup_3_0)->Basket=1
demo.ShoppingBasket2(phased-shuffledGroup_2_1)->FoundProduct=book A
demo.ShoppingBasket2(phased-shuffledGroup_2_1)->Basket=1
demo.ShoppingBasket2(phased-shuffledGroup_1_2)->FoundProduct=book A
demo.ShoppingBasket2(phased-shuffledGroup_1_2)->Basket=1
```

Another piece of information that is passed between the steps is the state of the scenario. This lets us know if the scenario is failing or not in its Shuffle Group.

3.3 Managing Context Between Phases

As mentioned in the section on Cross Phase Communication, at the end of the “producer phase” the cache is exported to a file, which is later consumed when the tests are executed in the “consumer phase”. The consumer phase imports the cache file and makes the data available to the phased tests in that phase.

We realize that managing a file between phases can be complex. To address this issue, we introduced a functionality called the “Phased Data Broker”. By implementing the Data Broker, users of the Phased Tests can override the default way Phase Data is stored. When a Data Broker is implemented on a user system, it is used by the framework to store the Phased Data as per the wishes of the user.

For example, in the current implementation, we store the Phased Data files on an SFTP server. This avoids problems when parallelizing tests.

3.4 Orchestration

As mentioned earlier, we chose not to manage the system change itself within the framework. This means that we use an external system to prepare a test system to manage the phases and to perform the system change. In our case this system is a Jenkins Pipeline.

A typical execution of a system change test campaign includes the following steps:

1. Provision: Setting the System Under Test to the state before the change.
2. Run the scenarios in the Producer Phase.
3. Perform the system change.
4. Run the scenarios in the Consumer Phase.

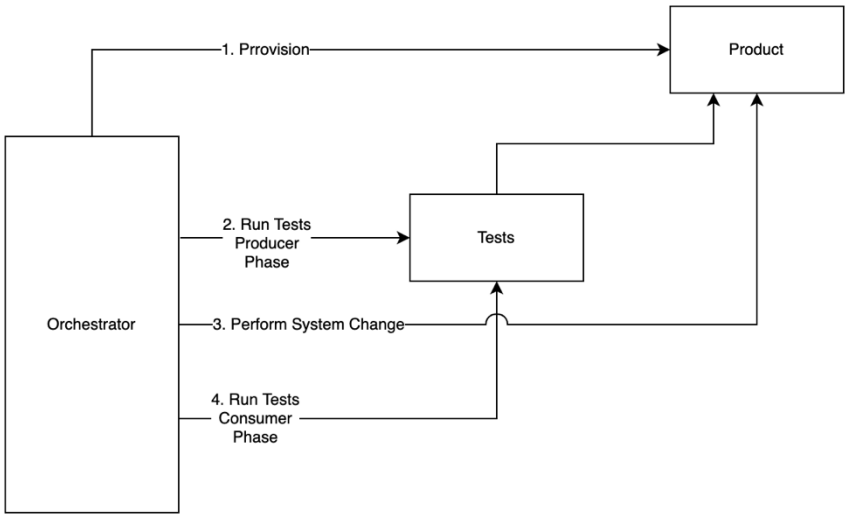

Figure 9: The execution of a typical Phased Test Campaign

The orchestrator, after having provisioned a test system, will run the tests in the Producer Phase by passing this as an execution property. After the Producer phase tests have finished, the orchestrator will perform the system change. Finally, when the change has happened, we will rerun the same tests, except this time they are in the Consumer Phase.

3.5 Results and Reports

In order to best reflect the results of Phased Tests, we report the results by Shuffle Group and Scenario.

The test report takes into consideration the full result of the scenario and its Shuffle Group. This includes the result and duration of the scenario and Shuffle Group spanning both phases. The consequence of this operation is that in most cases, we do not need to look at the results of the Producer phase, as it is already included in the report.

When a scenario Shuffle Group fails or is skipped, we provide the following information:

- The step in which the error occurred
- The phase where the error occurred
- The error messages

Since we include information about failures in the steps regardless of the phase, we usually focus mainly on the test report of the CONSUMER phase. The Scenario Shuffle Group result is based on the following cases:

PRODUCER Phase Result	**CONSUMER Phase Result**	**Report**
Passed	Passed	Passed
Failed	Skipped	Skipped with a description indicating in which step of the PRODUCER phase the Execution failed
Passed	Failed	Failed with a description of the step in which the scenario failed

4 Conclusions

The promise of Phased Tests is that of leveraging our test scripts to ensure that our application remains robust in the face of events. This will allow us to ensure that systems can carry on working as expected even after a major system change. As Phased Testing is a new concept, there is a lot of room for improvement.

We think that there is still work to be done in how we define and attach events to a scenario and its steps. For now, the events we have defined are interruptive and are executed before or after a scenario step. However, we would like to do research in the following domains:

Defining multiple events, will allow us to define a series of interruption types, so that we can test the system against each of these events.

As we mentioned before, our implementation of Phased Tests expects the scenarios to be interrupted, for our interruptive events to fold. We do, however, recognize that we have a class of "**Non-Interruptive Events**", that do not necessarily require the system to be interrupted to happen. Examples of this would the introduction of heavy loads in a system, or in the case of a driverless car, the introduction of an obstacle.

While speaking of parallel events, one will necessarily start considering **Parallel event** executions, which is a notion that will allow us to assess the effects of an event on any step of a scenario. In our current implementation all evens happen before or after a scenario step. We can imagine that an event happens during a scenario step.

Another area for further research in the field of Phased Tests would be to deduce the order of the steps by examining the "produced" and "consumed" data in each step. This would make the implementation more intuitive in the future. Although this would be of great interest, what is even more exciting is that the automatic detection of the step orders would then enable us to introduce a new test concept, namely the execution of a scenario with all the possible step combinations that would lead to the same end result.

Our experience is that we do not need to transform all scenarios into Phased Tests as it will make your System Change tests much longer. Good candidates for Phased Tests are the smoke tests or architecturally significant scenarios. Another good practice we have discovered in our research is to not add too many steps in the scenario as it increases the complexity, and scenarios will have a lot of Shuffle Groups as a result.

We see a lot of potential for Phased Tests, as it opens doors to testing the unexpected. We see an evolution where we can test a product against a large series of unexpected events. Today Phase Testing will help us predict problems related to major system changes. Tomorrow, this may go even further and allow us to predict stability regarding unexpected events.

Acknowledgements

The author of this paper would like to express gratitude to the following people (in alphabetical order):

- Mihai Bilan, for his great insights, ideas and his attentive proof-reading.
- Jean-Baptiste Garcia, for many off his useful suggestions and ideas.
- Felix Meschberger, for validation of the document, and being a great inspiration.
- Roy Ogus, for his aid in the creation off this document.
- Marco Stankovich, for providing valuable feedback on the phased tests implementation.
- Charles McBride, for his kind and helpful remarks.
- Qingfen Zhu, for providing valuable feedback on the first generation of the producer consumer concepts.

References

Cédric Beust, Hani Suleiman. 2007. *Next Generation Java Testing: TestNG and Advanced Concepts. Addison-Wesley Professional*

Marty Lewinter, Jeanine Meyer. 2015. *Elementary Number Theory with Programming. Wiley*

Baubak Gandomi. 2021. ”Automated Upgrade Testing: A Process to Test and Validate Software Upgrades”. Adobe Tech Blog, entry posted March 2021. https://medium.com/adobetech/automated-upgrade-testing-a-process-to-test-and-validate-software-upgrades-349e647f34f4 (accessed March 11 2021).

The Phased Testing TestNG Implementation. https://github.com/adobe/phased-testing.

Appendix A

Below is an example of the Shopping Basket scenario as a Phased Test, juxtaposed with the same scenario if we did not use Phased Tests.

The Shopping Basket Phased scenario. Here we only declare the scenario once. It will be used for both normal testing, and in Shuffle Mode:

```
@Test
@PhasedTest(canShuffle = true)
public class ShoppingBasket {

    public void step1_searchForProduct(String val) {
      //searchForProduct()
    }

    public void step2_addToShoppingBasket(String val) {
        //addToShoppingBasket()
    }

    public void step3_payForProduct(String val) {
      //payForProduct()
    }
}
```

The Shopping basket scenario in the case where we wish to implement all possible interruptions. (We have taken the liberty of still using Phases to highlight that we are in Producer/Consumer mode). If we chose to add a new step, we would have to add more methods, and update all methods.

```
public class ShoppingBasketWithIfs {

    //The normal test (0_3 and 3_0)
    @Test
    public void standardTest() {
        //searchForProduct()
        //addToShoppingBasket()
        //payForProduct()
    }

    @Test
    public void ShoppingBasket1_2() {
        if (Phases.PRODUCER.isSelected()) {
            //searchForProduct()
        } else {
            //addToShoppingBasket()
            //payForProduct()
        }
    }

    @Test
    public void ShoppingBasket2_1() {
        if (Phases.PRODUCER.isSelected()) {
            //searchForProduct()
            //addToShoppingBasket()
        } else {
            //payForProduct()
        }
    }
}
```