A Coalesced View of Software Development

Ken Pugh (ken@kenpugh.com) (https://kenpugh.com)

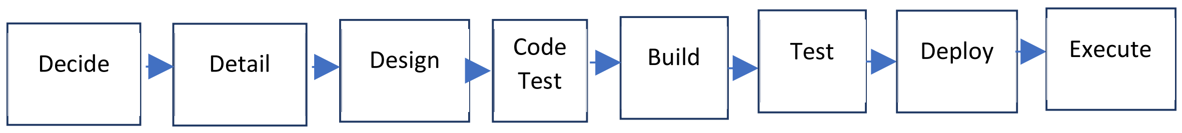
You've read about agile development, BDD/ATDD, agile architecture, and many of the other facets of agile. Many articles tend to discuss individual aspects, rather than putting them in a perspective of a whole development picture. Over the past year and a half, I've been coalescing ideas on software development. This view has emerged from my interactions with students, agile enthusiasts, and agile coaches, trainers, and authors. I've listed at the end many of the people from whom I've gathered different perspectives and feedback. This article merges in ideas that have appeared in my books on BDD/ATDD, software development, programming, and interface-oriented design.

The example used here was selected as most of the readers will have bought tickets on-line. It has simple and complicated parts. There are some details that will summarized. Many applications have similar characteristics to the example.

Overall Development Flow

Some common steps in development below are shown in a linear sequence. This does not imply that they occur only in that sequence. You can have feedback from later steps that cause a change in an earlier step. The cycling around the steps can be rapid (seconds) or slow (minutes / hours). Everyone adopts a flow that is appropriate for their context.

Here's a sample developmental workflow (or development value stream) that shows a stream from decision to deploy. At the end, you'll see how the steps that are described fit into this stream.



- Decide – determine that a requirement (feature or story) is going to be implemented
- Detail – discover the details of a requirement
- Design – explore possible implementations
- Code / Test – write the necessary code to pass the functional tests
- Build – create the deployable component
- Test – run cross-functional / non-functional tests, and explore with tests
- Deploy – deploy component to production
- Execute – run in production

The Example

The Tickzon company is considering presenting on-line events for a charge. The highest-level description is:

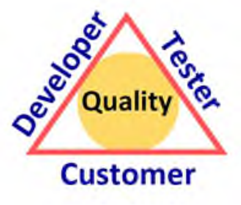
```
As an event manager, I want to create events, sell tickets, and
receive the proceeds.
```

This high-level item could be called a capability or a feature, depending on your point of view. For the purposes of this article, I'll call it a feature.

Participants

There are many roles that participate in developmental value stream. In most places, these roles have at least these three perspectives:

- The customer who provides the details of the requirements
- The developer who implements the requirements
- The tester who critically analyzes the requirements and the implementation



These perspectives may match an individual's role or they may be perspectives of two or three different people who are working on the feature. There could be many other stakeholders for a feature. There may also be a role who determines in what sequence requirements are implemented (such as a product manager).

The Hypothesis

The hypothesis for a feature describes the business value that is expected when it is in production. In this example, it might be:

```
We think that presenting paid on-line events will increase learning
due to the stake in the game from the attendees as measured by on-
line tests given during and after the event.
```

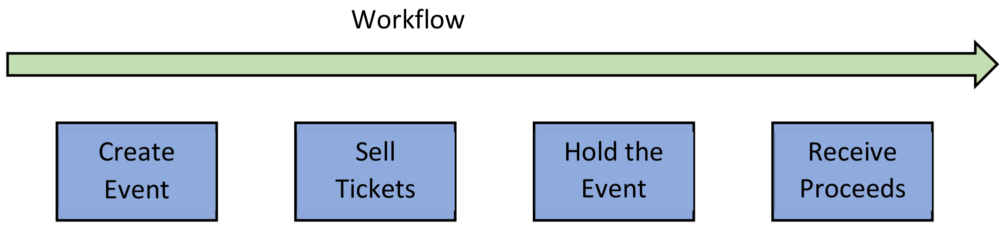
The metric for this feature will be checked once the feature is in production. If the metric is met, then the feature will be kept. If it is not met, then some options are:

- Modify the feature to see if an alternative will work (e.g. try something for lower expenses)
- Abandon the feature

Often software alone cannot achieve the expected business value. In this example, advertising may be needed so that attendees buy tickets for the events. A hypothesis for advertising would be created and that would be implemented by the appropriate team (e.g. marketing).

The Overall Operational Flow

There is an operational flow (or operational value stream) which shows how this feature initially works by showing the activities involved in the event. The flow is the context for the stories that are going to be developed (ala a story map). There may be alterations in the flow as the feature is used. The sample flow is shown as linear, but flows can have loops or alternative paths.



Development Decisions

The team at some point needs to decide how to implement this feature. Some possibilities are:

- Use third party for entire flow.
- Use third party for portion of the flow (e.g., handling the sales) and create own implementation for the rest.
- Create own implementation for everything.

Preferably the implementation will permit relatively easy substitution of one decision for another. Even if there is a third party for everything, some of the next steps could be done so that the third-party application can be tested to see if it meets the requirements.

High Level Scenarios

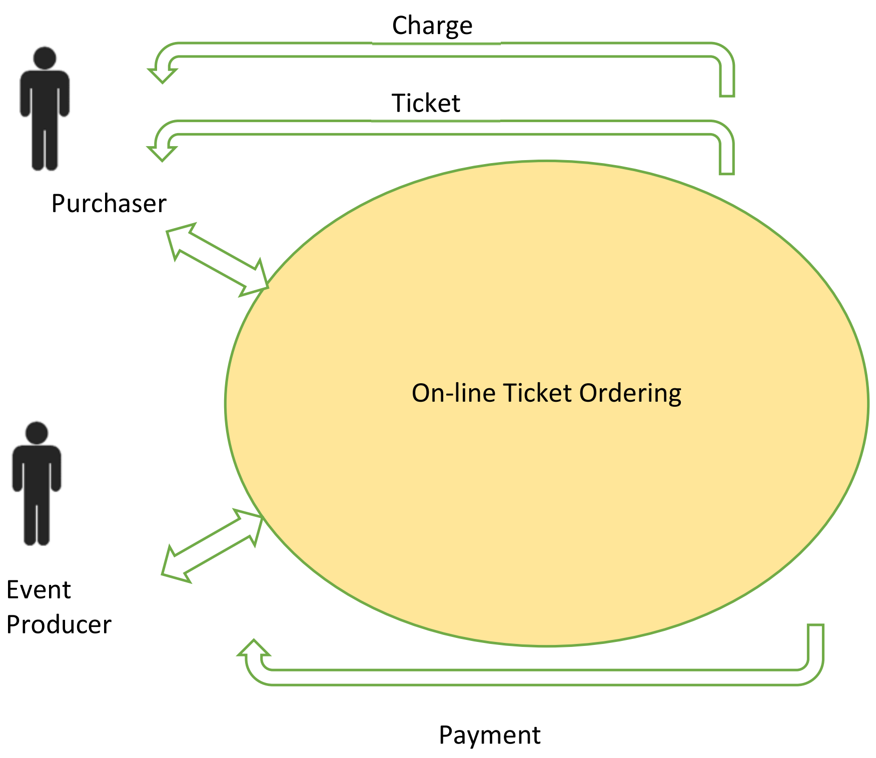
You can describe a feature or a story as a scenario. Scenarios are examples of the behavior that the feature or story represents. I'll discuss more about scenarios in a later step. These are scenarios for some of the steps in the operational workflow.

```
Scenario: Create Event
Given event does not exist
When producer creates event
Then event is available for ticket sales

Scenario: Sell Tickets
Given event is available for ticket sales
When a purchaser purchases a ticket
Then purchaser receives ticket in the mail
And purchaser is charged for the ticket

Scenario: Receive Payment
Given event has been held
When producer requests payment
```

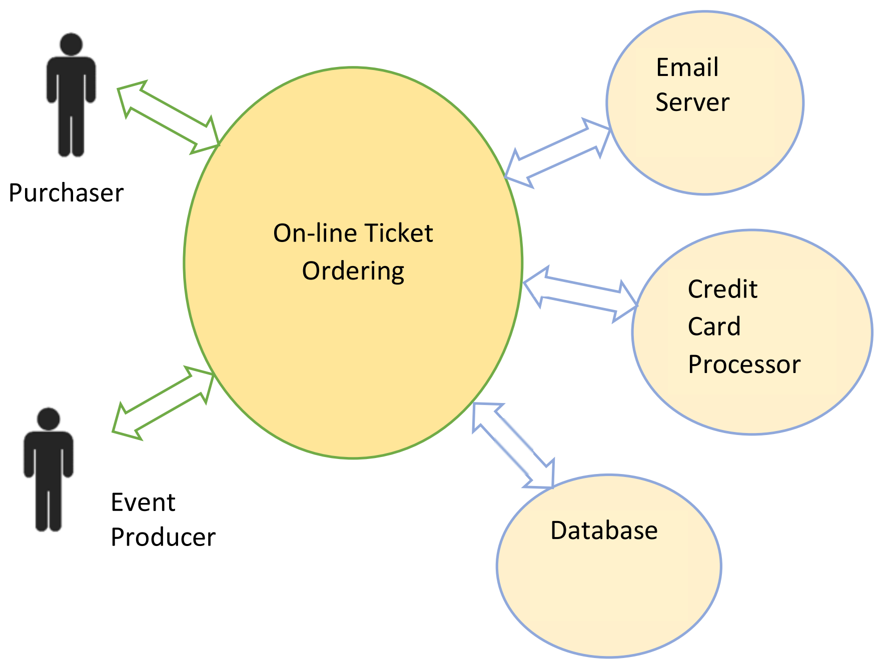
```
    Then payment amount is transferred to producer
```

The Actors and the Context

The actors (e.g. the producer and the purchaser) and their relationship to the application can be shown with a context diagram. At the highest level, the diagram might look like:



Here is a more detailed context diagram that shows the proposed application and the external components it communicates with to produce the effects (payment, charge, ticket) that are in the diagram above.



Exploration and Discovery

The above scenarios can be used as a starting point for creating more scenarios. These often come from the exceptions that might occur when something goes wrong. These exceptional scenarios can be discovered anywhere in the process, but the sooner they are discovered, the information can be used to make better decisions. For example:

```
Scenario: Create Event for Date in the Past
Scenario: Sell a Ticket after Event is over
Scenario: Purchaser’s Purchase Information is Invalid
Scenario: Purchaser Does Not Receive a Ticket
```

Scenarios into Tests

Let’s add some example data to the above scenarios. The additional data can suggest missing requirements. For instance, in creating these scenarios, the fact that the number of tickets should be reduced by each sale was left out. One convention is to put the names such as Price and Number Tickets in the left column when they are being supplied by an actor during the scenario and names on the top row if the items already exist (from another scenario) or are being created (and potentially passed to another scenario).

```
Scenario: Create event
Given event does not exist
When producer creates event
| Name             | Wonderful Time   |
| Date             | 6/1/2021         |
| Time             | 1:00 PM EDT      |
| Number Tickets   | 100              |
| Price             | $5.00             |
Then event is available for ticket sales
| Name              | Date       | Time           | Number Tickets  |
| Wonderful Time    | 6/1/2021   | 1:00 PM EDT    | 100             |

Scenario: Sell Ticket
Given event is available for ticket sales
| Name              | Date       | Time           | Number Tickets  |
| Wonderful Time    | 6/1/2021   | 1:00 PM EDT    | 100             |
When a purchaser buys a ticket
| Event                 | Wonderful Time             |
| Number Tickets        | 1                          |
| Email                 | sam@thisisonlyatest.com    |
| Credit Card Number    | 4005550000000019           |
Then purchaser receives ticket in an email containing
| Name              | Date       | Time           | Number Tickets  |
| Wonderful Time    | 6/1/2021   | 1:00 PM EDT    | 1               |
And purchaser is charged for the ticket
| Credit Card Number    | Item              | Charge   |
| 4005550000000019      | Wonderful Time    | $5.00    |
And number of tickets decreases
| Name              | Number Tickets    |
| Wonderful Time    | 99                |
```

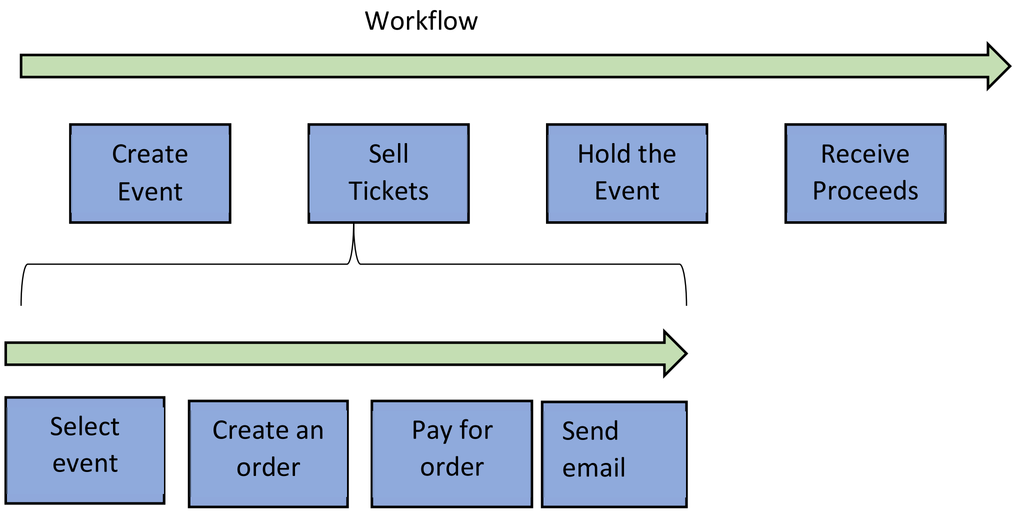
Scenarios and tests are closely inter-related. A test is a scenario that is executed and all of the expectations shown in the Then part are checked. If the expectations are met, the test passes. For the first scenario, the check would be:

```
Then CHECK event is available for ticket sales
| Name              | Date       | Time           | Number Tickets  |
| Wonderful Time    | 6/1/2021   | 1:00 PM EDT    | 100             |
```

These scenarios form the external tests for the requirement. They represent the desired behavior and are independent of the implementation. They could be executed manually or automatically (preferred).

Decomposition

High-level features are decomposed into stories that are independently developed. The acceptance criteria for stories typically describes some behavior. The behavior can be written in Given/When/Then form and thus form a test by checking the Then. The scenarios describe more detailed behavior of steps within the workflow. They typically describe how a user uses the system (so the scenarios have a parallel with the common use case).



```
Scenario: Create an order
Given event is available for ticket sales
| Name            | Date      | Time          | Number Tickets  |
| Wonderful Time  | 6/1/2021  | 1:00 PM EDT   | 100             |
When a purchaser selects a ticket to purchase
| Event               | Wonderful Time            |
| Number Tickets      | 1                         |
| Email               | sam@thisisonlyatest.com   |
Then an order is created
| Event           | Number Tickets   | Price   | Total   |
| Wonderful Time  | 1                | $5.00   | $5.00   |

Scenario: Pay for order with valid card
Given an order exists
| Event           | Number Tickets   | Price   | Total   |
| Wonderful Time  | 1                | $5.00   | $5.00   |
When the purchaser pays
| Name on Card        | Sam Jones                 |
| Credit Card Number  | 4005550000000019          |
| Expiration          | 12/2025                   |
| Security Code       | 123                       |
Then credit card transaction is sent to processor
| Credit Card Number  | Item              | Charge   | Expiration     |
Security Code   |
| 4005550000000019    | Wonderful Time    | $5.00    | 12/2025        |  123
|
```

Each of these scenarios has more detail than the overall scenario. Scenarios can be decomposed into smaller scenarios. Each scenario specifies a behavior and thus a test for that behavior.

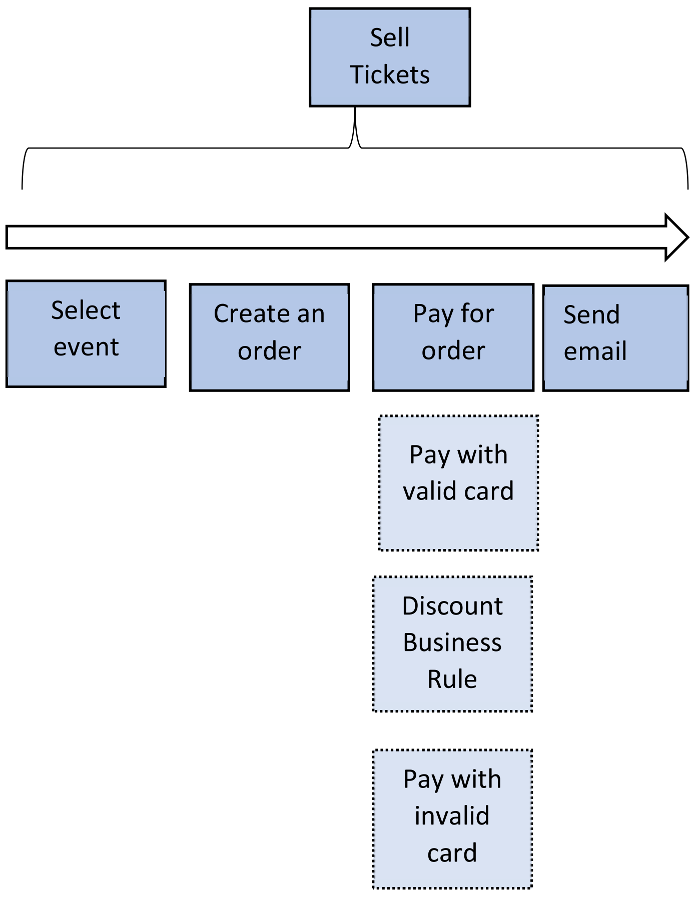
Business Rules and Domain Terms

In developing scenarios, one often comes across domain terms that form part of ubiquitous language shared between the triad. Many applications have business rules that are applied. Understanding and checking these business rules is an important part of development. Running the tests for rules and domain terms is typically fast, as there is no state that needs to be setup.

```
Scenario: Domain Term Number Tickets
* Number Tickets represents how many tickets are available or being sold
| Value  | Valid  | Notes               |
| 0      | No     | Must be at least 1  |
| 1      | Yes    |                     |
| 100    | Yes    | Maximum             |
| 101    | No     | Over maximum        |

Scenario: Business Rule Discount
* Discount is given for buying more than one ticket
| Number Tickets  | Discount Percentage  | Notes                       |
| 1               | 0%                   |                             |
| 2               | 5%                   | Discount for 2 to 5         |
| 5               | 5%                   |                             |
| 6               | 10%                  | Discount for 6 or more      |
| 100             | 10%                  |                             |
```

Mapping the Flow

The scenarios and business rules relate to steps in the workflow. This is a slightly different view of story mapping (Patton). The scenarios and business rules are shown under the steps in which they take place.

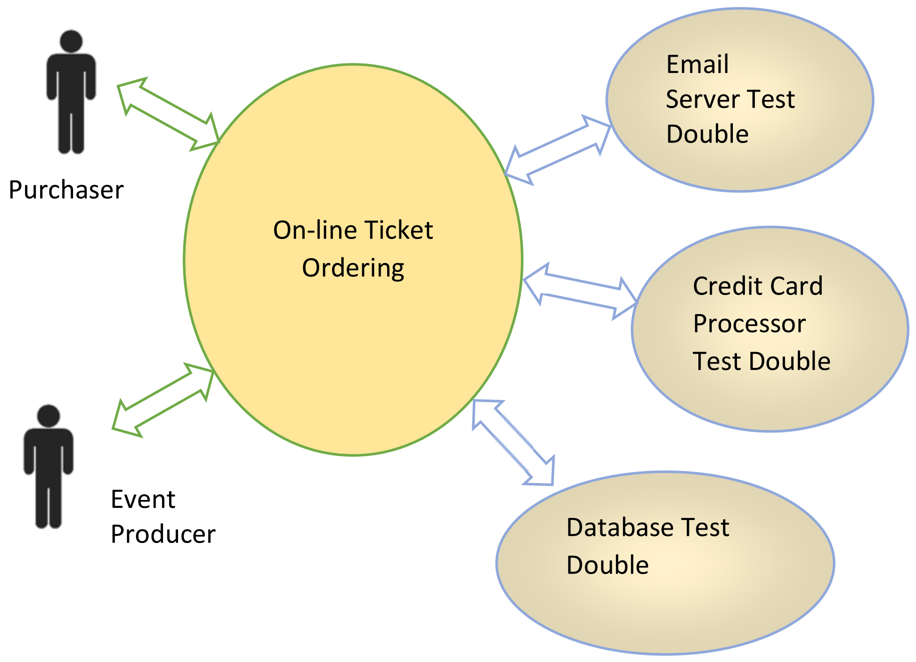


Minimum Marketable Feature / Minimum Business Increment

You do not have to implement every scenario for the feature to be released. At least one scenario needs to be in place for each step in the workflow for that feature. The Minimum Marketable Feature represents the fewest number of scenarios that are needed to be implemented to provide the business value that the feature represents. Then the workflow can be changed by incrementally implementing one or more additional scenarios or altering the requirements for an existing scenario.

The Detailed Context and Test Doubles

The expanded context diagram shows the external components which send the email, charge the card, and store the event data. The context diagram shows interactions with external components or systems. In order to run repeatable, fast tests on an application, you may need test doubles for those external interactions, as shown in the following diagram. (Test doubles are often called mocks or other names). The test doubles have the same interface as the real components (e.g., Email Server, Credit Card Processor). The doubles can be programmed to return the data that the real component would return. The double may be separate from the application (as they would be in production) or they may be integrated into a test version of the application (using dependency injection or other techniques).



Sequencing Development

One way to decide in which sequence scenarios should be developed is to start with a set of scenarios that allow for a complete workflow. This has various names, including Tracer Bullet (Pragmatic Programmers). It acts as an early check of the relationships and communication between the steps. It also can be used as a check that the development pipeline is setup to easily deploy the application.

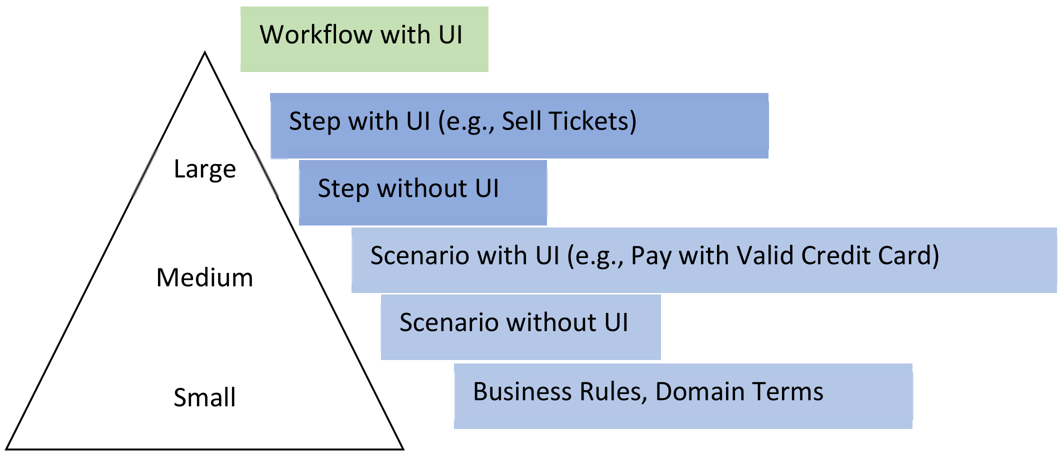
Checks and Tests and the Testing Matrix

The scenario tests which detail the requirements check that the application meets the requirements. They are executed over and over as regression tests to check that new implementations don't break them. Unless a requirement changes, these checks do not change. So they are sometimes called "checks" rather than "tests". Passing these checks is necessary, but not sufficient to have a quality application. An application needs to pass exploratory tests that investigate unexpected behavior, usability tests that make sure users can navigate the flow easily, and cross-functional (aka non-functional) tests for attributes as performance and security.

Functional	Cross Functional
Customer Tests / Checks (Scenarios, Business Rules, Domain Terms)	Usability, Exploratory
Technical Tests/ Checks (Components, Classes)	Performance, Security, etc.

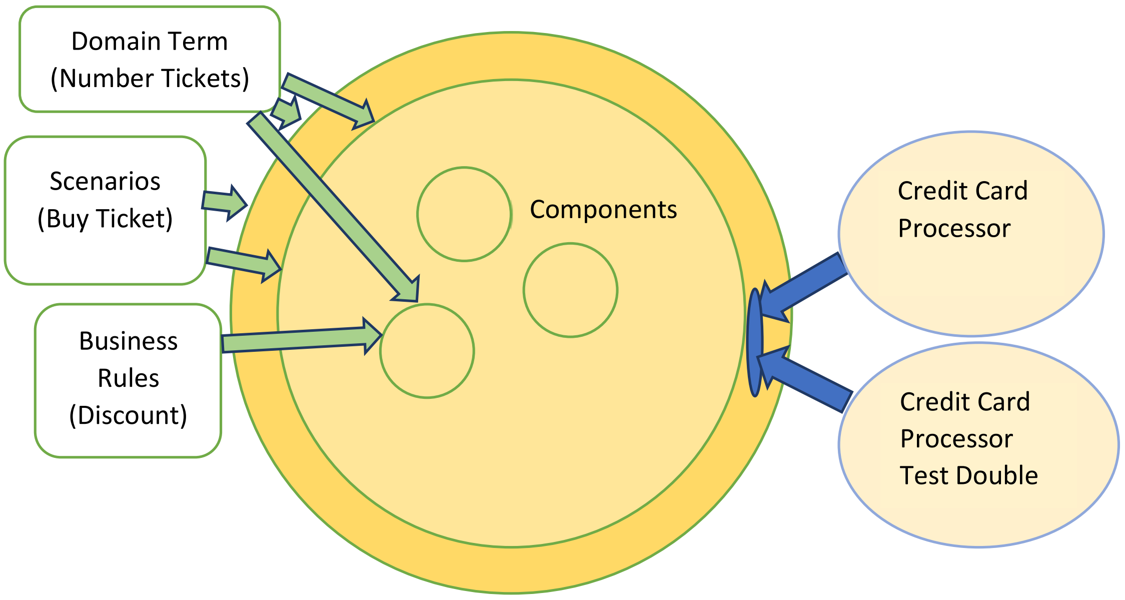
The Testing Pyramid

The testing pyramid suggests the number of tests at each level of the pyramid. The original pyramid used End-to-End, Integration, and Unit as the test types. One version from Google used Large, Medium, and Small. Here is how the tests might be distributed in this pyramid among those categories based on what they are testing.


Workflow with UI
Large
Step with UI (e.g., Sell Tickets)
Step without UI
Medium
Scenario with UI (e.g., Pay with Valid Credit Card)
Scenario without UI
Small
Business Rules, Domain Terms

Detailed Context and Architecture

The actors on the left side of the previous context diagram typically interact through a user interface. The external components on the right side may interact via a request/response protocol, such as REST or web services. A more detailed diagram shown below indicates how the scenarios can interact with the application. This looks much like a clean architecture or a hexagonal architecture diagram (Martin, Cockburn). Both the real external component and the test double for that component use the same interface to the application.

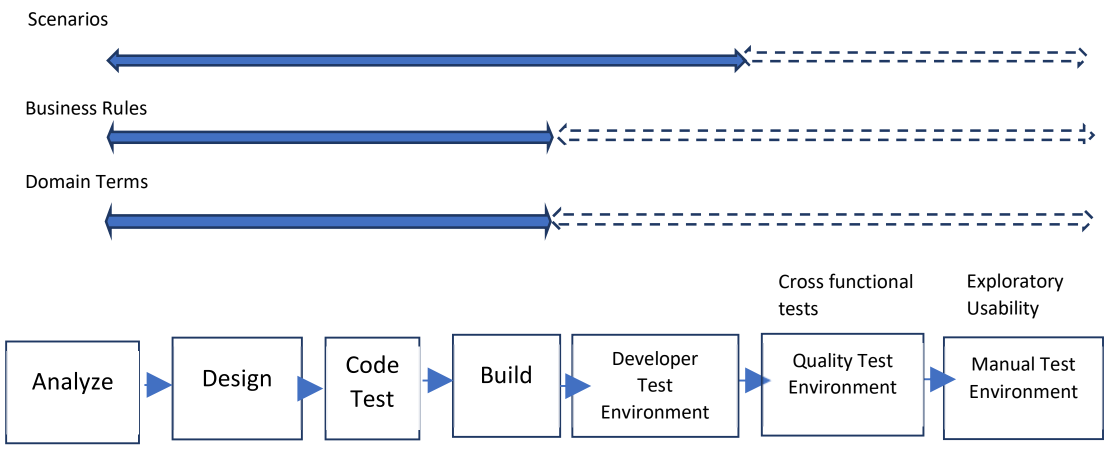

Domain Term (Number Tickets)
Scenarios (Buy Ticket)
Business Rules (Discount)
Components
Credit Card Processor
Credit Card Processor Test Double

Architecture and Design

The scenarios deal with external observable behavior that is required. The developers can select whatever application architecture, user interface framework, and internal design meets their desired qualities, such as maintainability, simplicity, consistency, testability. The other cross -functional (aka non-functional) requirements must also be meet by the chosen architecture and design. The architecture could be monolithic, microservices, or a combination of both.

The Scenario Flow

The diagram below shows a portion of the development flow where the scenarios, business rules, and domain terms can be created and run as automated tests. Business rule and domain term tests are typically implemented by small components which can be tested as part of the build. Some form of all the tests may be executed in all environments.



Complicated and Complex

The workflow/scenario approach shows one way to decompose a complicated flow into simple ones. The flows represent the journey for a single user or a single set of events from the triad's point of view. The flow should be the same regardless of whether there is a single user or multiple users going through it at the same time. It's a development decision on how to ensure the implementation provides that behavior. There are many forms of complex systems. One aspect is the having a shared resource that has competition for it. For example, multiple buyers may be purchasing tickets at the same time. The last step in the scenario was:

```
And number of tickets decreases
| Name            | Number Tickets  |
| Wonderful Time  | 99              |
```

The implementation needs to ensure that the decrease is handled properly when there are multiple purchasers concurrently going through the flow of purchasing tickets. Some shared resource issues may be mostly a development issue. However, some issues may suggest a change in the workflow. In this instance, further Triad discussion may suggest three alternatives:

- Having reservations, where tickets are reserved for a brief period of time until they are paid for and unreserved if they have not been paid within that time.
- Allowing overbooking
- Informing the purchaser that the tickets are no longer available since they did not complete the order quick enough

The outline of a scenario that describes the last alternative might be:

```
Given there are 100 tickets
When 101 purchasers are simultaneously trying to order them
Then 100 purchasers get their tickets
And 1 purchaser is disappointed
```

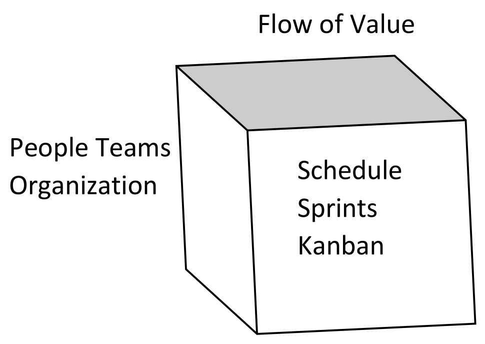
The INVEST Criteria

Scenarios align well with the INVEST criteria (Wake):

- I– Independent – each scenario can stand alone, although they may have to be implemented in a sequence
- N – Negotiable – the decomposition of a scenario may result in additional scenarios that might be delayed
- V – Valuable – the scenarios help provides the value in the feature that they represent
- E – Estimable – scenarios can estimated
- S – Small – scenarios are small
- T – Testable – the scenario is the test

Perspectives on Agile

One agile perspective revolves around people (teams, relationships, etc.). A second perspective considers schedules (sprints, backlogs, etc.). The workflow with the associated scenarios as described in this article are a third perspective on how value is delivered to the customer.



Applicability

This developmental flow represents one view that works, that may not be applicable to your environment. The flow is more applicable to systems where most stories represent a change or addition to a workflow requested by the customer, rather than experimental systems. The stories may represent cross-functional (aka non-functional) requirements that apply to parts or the whole of a workflow.

Change in Scenarios

If a scenario needs to change, then you alter the scenario. The test should fail since the implementation has not be altered for the requirement change. If the test passes, then you'll need to examine carefully the underlying test code and production code to see what is the reason.

If you do not have a scenario / test for a requirement that is changing, then you should first write one for the current requirement.

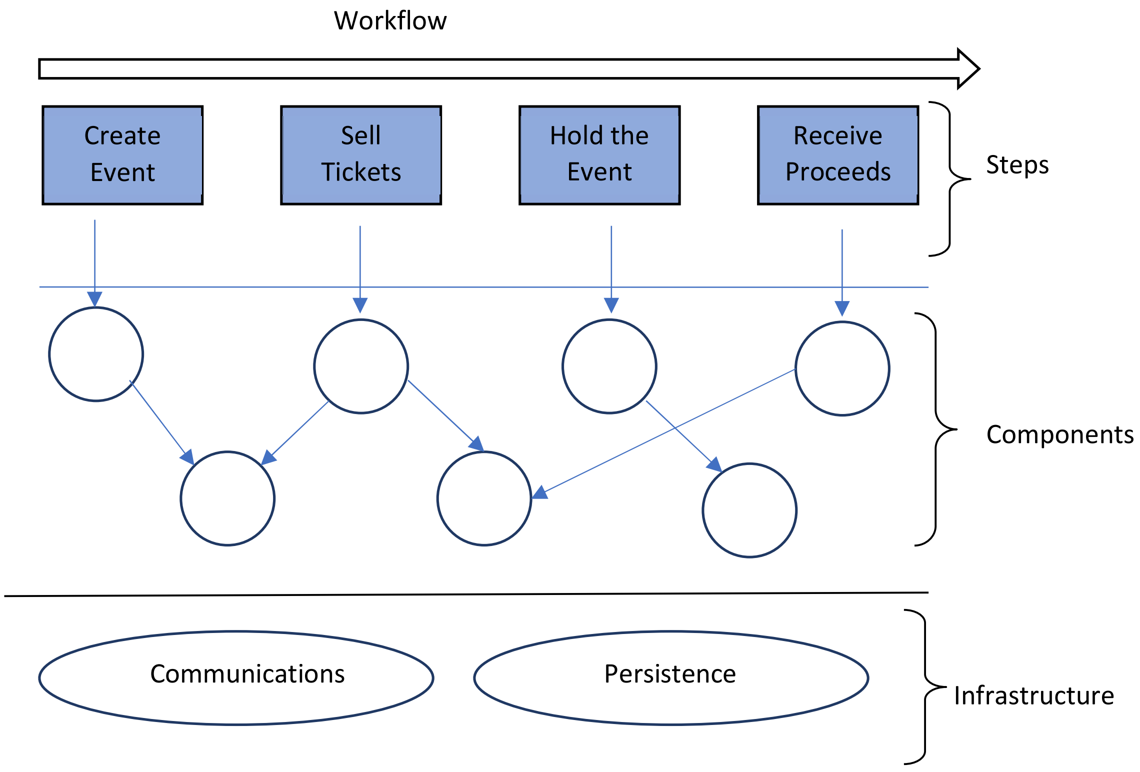
Run the test for the scenario and trace the implementation components that are executed to implement the scenario. You can do this for all scenarios to determine which ones are affected when you make a change in an implementation component. As a quick preliminary check during development, you should run the tests for those scenarios.

Who's Responsible?

Here are some possibilities. How you organize responsibilities is dependent on the system and your organizational size:

- One team is responsible for all the components necessary to implement the event ticketing workflow.
- Multiple teams are responsible for components for individual steps of the workflow (e.g. Sell Tickets, Receive Proceeds), and share responsibility for components used by multiple steps.
- Multiple teams are responsible for components for individual steps and other teams are responsible for components used by those steps, if they are shared with other workflows.

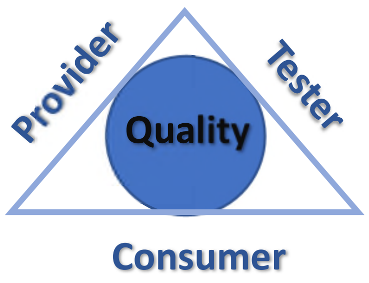
If there are multiple teams, they are usually in a workflow group (team of teams, Agile Release Train, etc.). Typically, most workflows have a common infrastructure (database, operating system, etc.) and a common developmental infrastructure (source code control, build, etc.).   Who provides these is a development/operations decision.

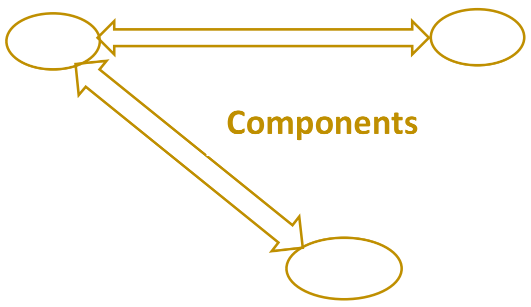


Component Triads

If there are separate executable components (e.g., services or microservices), or shared libraries, then the triad concept can be applied to these components using slightly different terminology:

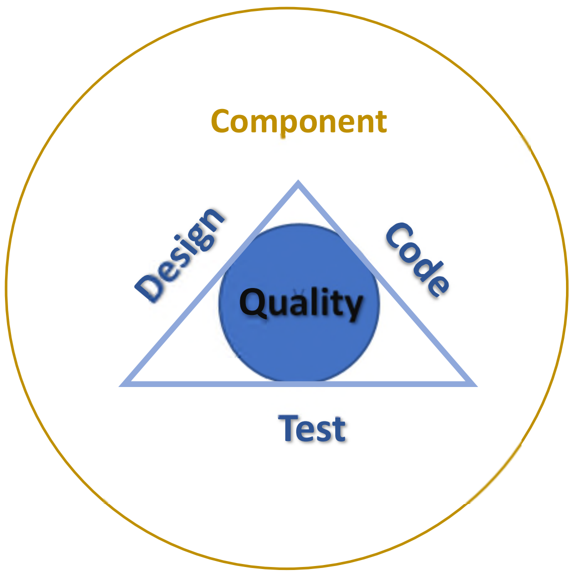
- Consumer – user(s) of a service/library
- Provider – developer of a service/library
- Tester – critically analyzes a service/library and its implementation





The component triad specifies the behavior of a component.   They can use scenarios written in the implementation language, in Gherkin, or in another form – whichever is most effective.

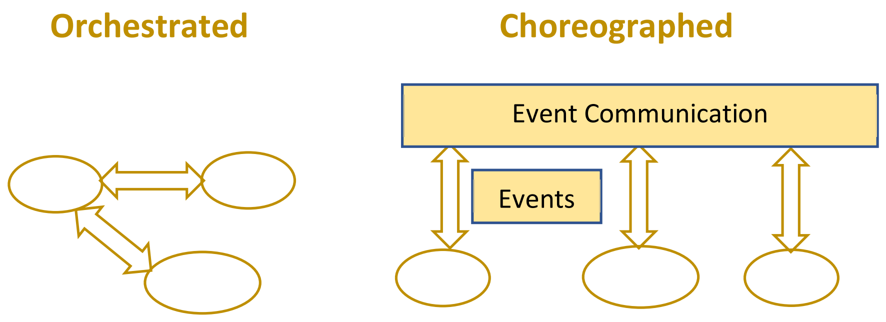
Within a component, there is a triad of perspectives, often represented by two developers who are pair programming



As Jerry Weinberg once said, “If you can’t come up with three solutions to a problem, you don’t understand the problem.”   When considering how to implement a component, you can come up with three possible implementations.   Then compare them to see which one might be the most appropriate. When coding that one, you might discover that a different one or some combination might work better.

Component Interaction

Components can interact with each other in at least two ways. Communication can be orchestrated - one component directly communicates with another either via a method call or a message. The response or result of that message may be received synchronously or asynchronously. Alternatively, interactions can be choreographed, which is more decentralized – for example, one component can generate an event that other components listen for and perform appropriate actions.   Orchestrated behavior is declarative; choreographed behavior is reactive.



Event / Response Behavior

Another way to specify the behavior of an application is with an event/response listing. An event is something that occurs externally to an application, such as user pushing a button, a message received from another system, a time period expiring, and so forth.   The response is how the application reacts to the event.   For example, the payment process might have an event/response list as:

Event	Response
User checks out	System requests payment information
Payment information submitted	System sends information to third party
Third party approves payment	System sends email to customer
Third party denies payment	System requests alternate payment

Event / response behavior can also be specified for choreographed components.

The response may be dependent on some state in the application. For example, the number of times that a “Third party denies payment” might be kept and if that number exceeded a limit, the response would be different, such as “System says to contact representative”.

Summary

This article is an initial effort to tie together many of the aspects in agile development- from the value stream to scenarios to development. The topics will be explored in separate articles. I welcome your feedback and input.

Acknowledgements

I gather inspiration from many sources. These are some of the people from whom I’ve received ideas and thoughts that went into this article.

Jeff Patton, Ward Cunningham, Eric Evans, Scott W. Ambler, Alistair Cockburn, Robert Martin, Gojko Adzic, Seb Rose, Aslak Hellesøy, Gáspár Nagy , Lisa Crispin, Janet Gregory, Matt Wynne, Joshua Kerievsky, Al Shalloway, Gerard Meszaros, James Bach, Michael Bolton, Daniel Terhorst-North, Larry Constantine, Lucy Lockwood, Michael Feathers, Martin Fowler, Ellen Gottesdiener, Andy Hunt, Dave Thomas, Cem Kaner, Craig Larman, Bas Vodde, Grigori Melnik, Bret Pettichord, Mary Poppendieck, Tom Poppendieck, Donald Reinertsen , Rebecca Wirfs-Brock , Jerry Weinberg, James Whittaker, Karl Wiegers, George Dinwiddie, Rob Myers, Bob Galen, Luke Hohmann, Johanna Rothman, GeePaw Hill, Elisabeth Hendrickson, James Grenning, Kevlin Henney, Doc Norton, Jeff Langr, James Shore, James Coplien, David Scott Bernstein, Ron Jeffries, Chet Hendrickson, Dean Leffingwell, James Grenning , Allen Holub, Thomas Owens, Kent Beck, William Wake, Mike Cohn