# The Auction Sniper – An ATDD/BDD Approach

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

In "Growing Object-Oriented Software, Guided by Tests", Steve Freeman and Nat Pryce excellently present driving the development of an application with tests. The example application on which they demonstrate their techniques has far more complexity than your typical TDD example. That makes it a good example of showing some ATDD/BDD techniques that are not usually demonstrated in simple applications.

In ATDD/BDD, scenarios that become tests describe the behavior of an application from the external view. These scenarios are implementation independent, so they do not necessarily drive the design of objects, as Steve and Nat show in their book. However this separation between what should be the behavior and how to implement that behavior works like the Separation of Concerns principle to reduce complexity. Steve and Nate avoid "Tests [that] are too tightly coupled to unrelated parts of the system or unrelated behavior of the object(s) they're testing". ATDD/BDD tests are just that.

### ATDD/BDD

In ATDD/BDD, the Triad consists of three perspectives – Customer, who provides the requirements; Developer, who implements the requirements, and Tester, who critically analyzes the requirements and the implementation. They collaborate to create a common understanding of the required external behavior of an application.

As Kent Beck suggests in his book, "Before you begin, write a list of all the tests you know you will have to write.". With ATDD/BDD, one can start with understanding the application context, the workflow, and scenarios that specify the external behavior. These scenarios are typically expressed as in a *Given/When/Then* format. A scenario becomes a test when it is executed, and the actual results are compared against the *Then* part.

## Unit Tests and External Tests

Unit tests typically check the behavior of a component (class, method, module). ATDD/BDD scenarios define the overall behavior of an application. You must demonstrate all these behaviors as part of the definition of done.

Scenario tests that faile do not necessarily point to a specific component, as unit tests do. But if you have just run a scenario test that passes, then you make a change, and the test fails when you run it again, you can be fairly sure of what caused the issue.

You may find there is duplication between an ATDD/BDD scenario test and a unit test. You can decide which way the behavior should be documented and remove the duplicate test.

In ATDD/BDD, you may find multiple *Then* steps in a scenario. If any of the expectations that these *Then* represents, the scenario fails. This avoids having multiple scenarios with the same *Given/When* steps.

With unit tests, one common advice is to avoid having multiple expectations (assertions) so that multiple failures of those assertions will not be hidden.

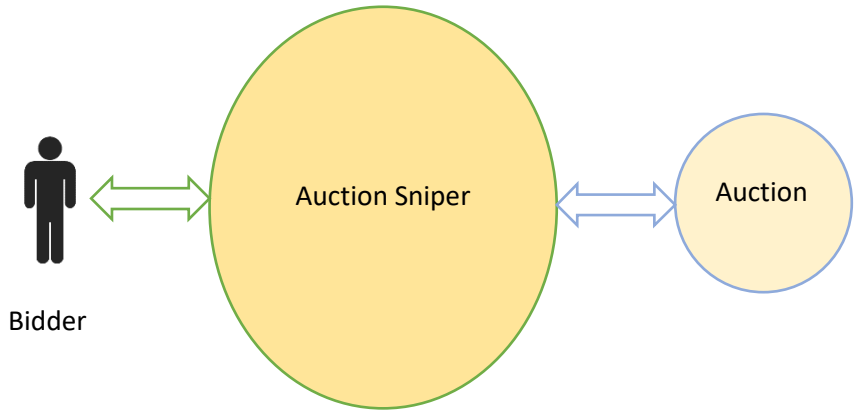
## The Example

As Steve and Nat describe it:

> *Auction Sniper*, an application that watches online auctions and automatically bids slightly higher whenever the price changes, until it reaches a stop-price or the auction closes.

## Context Diagram

The Triad can start by drawing a context diagram to show what's inside and outside of the application. One for Auction Sniper looks like this:




The context diagram shows what is outside the application. The outside entities are candidates for test doubles. This will be discussed in more detail later on.

## Interactions

The Bidder-Auction Sniper interaction looks like this:

- Input:
  - Item – identifies the item for which to make bids
  - Maximum bid – bid should not be higher than this
- Output:
  - Item
  - Last price – received from auction
  - Last bid – that Sniper made
  - Maximum bid – that Sniper can make
  - Status – a domain term shown next

The Auction Sniper – Auction interaction follows this message protocol

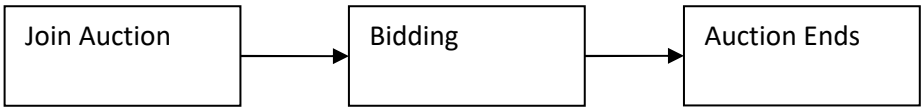
- Messages to Auction
  - Join
  - Bid <Price>
- Messages from Auction
  - Price <Current Price> <Increment> <Bidder>
  - Close

The <Current Price> is the current winning bid that was submitted by the <Bidder>. Another bidder must make a bid that is <Increment> higher than <Current Price>

Note the messages from an auction occur after a Join, after a Bid, or at any time due to another bidder making a bid.

## Flow

The Triad should have a common understanding of the workflow of an application. The flow for the Sniper looks like the following. The final step is "Auction Ends", which implies that the Auction has sent a Close message. However, we might want to add the user's ability to terminate bidding. That could be shown as a separate step coming from Bidding or just be incorporated into Auction Ends




## Domain Terms

As we discuss a feature or context, we may come across domain terms – words that represent concepts in a domain. For example, status appears to be domain term. So we can make up a scenario that describes the values of status.

```
Scenario: Domain Term Status
* The bidding status for an auction
| Value    | Notes                                                  |
| Joining  | Connecting to auction                                  |
| Bidding  | Making bids                                            |
| Winning  | Current bid is the winner                              |
| Losing   | Current price and maximum bid do not allow another bid |
| Won      | Auction ended and current bid is winner                |
| Lost     | Auction ended and did not win                          |
```

These are the statuses that ae displayed on the output. We'll see later how these statuses are determined.

## Business Rules

Business rules are at the core of many applications. They determine the flow through many applications. The Triad needs to form a common understanding of those rules and can do so by creating business rule scenarios. Business rules often consist of a calculation with inputs and output. In this example, there is one main business rule.

```
Scenario: Business Rule Make a Bid
* Make a bid increment greater than current price, unless over maximum bid
| Current Price  | Increment  | Bidder  | Maximum Bid  | To Bid  | Price  |
| 100            | 10         | Self    | 200          | no      |        |
| 100            | 10         | Other   | 200          | yes     | 110    |
| 190            | 10         | Other   | 200          | yes     | 200    |
| 191            | 10         | Other   | 200          | no      |        |
```

Creating separate business rules allows quick testing and gives confidence that the application will work as desired. The business rule can become much more complicated, such as making bids twice greater than the increment.

## Input to Output

All output of an application comes from somewhere. It could be from input, from other processes (e.g. the auction), from databases, from the results of business rules, fixed values, or some combination.

The following scenarios demonstrate where the output comes from.

The Triad starts with an overall context for each of the scenarios. Specifically, an auction exists for which the user is going to bid.

```
Background: Auction exists
Given auction exists for:
| Auction House  | Item  |
| Southabee      | 2134  |
```

The first step in the flow is to Join Auction. This produces two outputs – a Join message and the value of Maximum Bid. One could separate this scenario into two if needed. But since both outputs are required for the application to behave as desired, it can be more understandable to see them together.

```
Scenario: Join an auction with maximum bid
Given
When joining auction
| Item | Maximum Bid |
| 2134 | 100 |
Then message sent
| Type |
| Join |
And output shows
| Item  | Maximum Bid  | Status   |
| 2134  | 100          | Joining  |
```

When a Price message is received from the auction, the Current Price is updated.

```
Scenario: Receive price from auction
Given auction joined
| Item |
| 2134 |
When message received
| Type    | Curent Price  | Increment  | Bidder  |
| Price   | 110           | 10         | Other   |
Then output shows
| Item   | Last Price  |
| 2134   | 110         |
```

When a bid is made, then the Last Bid is updated.

```
Scenario: Make a bid
Given auction joined
| Item |
| 2134 |
When message sent
| Type   | Price  |
| Bid    | 200    |
Then output shows
| Item   | Last Bid  |
| 2134   | 200       |
```

Now we have covered where the output values originate, except for the Status. That comes next.

## A State called Status

The status represents the state of the bidder and the auction. There is a transition between states based on events. Two ways of representing state transitions are a state diagram and a state table. I prefer the state table. It can be easier to read, and it's amenable to storing in source code control and changes can be easily displayed.

The following table shows the states and the event that can occur. Each of the state-event entries could show just the next state. But in some case, an operation may need to be executed to determine the next state, as shown by the if/elses.

| State/. Event | Join sent | Close received | Price received Bidder = self | Price received Bidder = other | Maximum Bid changed |
|---|---|---|---|---|---|
| Initial | Joining | | | | |
| Joining | | Lost | Winning | Make bid<br>If bid<br>  Bidding<br>Else<br>  Losing | |
| Bidding | | Lost | Winning | Make bid<br>If no bid<br>  Losing<br>Else<br>  Bidding | |
| Winning | | Won | Winning | Make bid<br>If bid<br>  Bidding<br>Else<br>  Losing | |
| Won (terminal) | | | | | |
| Losing | | Lost | | | Make bid<br>If bid<br>  Bidding<br>Else<br>  Losing |
| Lost (terminal) | | | | | |

There is not an absolute way to represent a set of states. One could have a Bidding state with Winning and Losing being sub-states within that. There might be a "Bid sent" event which could represent the "If bid" part of the actions in some of the state/event entries.

The blank entries represent "impossible" transitions – the event should not occur for the state. A record of any times that happened could be analyzed for "what have we missed?". Some people put a "N/O" instead of blank, to show that it should "Not Occur".

If you closely examine the table, there few state/events that might or might not occur. For example, could you start by Winning? Perhaps the communication was interrupted, so you had to join again, and you had made the highest bid previously. Or could you get another Price message with your name while you are winning? Probably not, so that could be an "N/O".

You don't have to complete all the entries. However, every transition is a scenario that needs to be tested. So, if you're creating a list of all tests, the more you fill in, the more tests you'll have.

When you are testing state transitions from the external world, you could have a test point in the code where you set the current state, such as:

```
Scenario: Transition Joined to Lost State
Given current state is
| Status |
| Joined |
When messages received
| Type    | Current Price | Increment  | Bidder  |
| Price   | 110           | 10         | Other   |
| Close   |               |            |         |
Then output shows
| Status   |
| Lost     |
```

Alternatively, you could create each state from a series of events that produce it. That's what we'll show in the following scenarios:

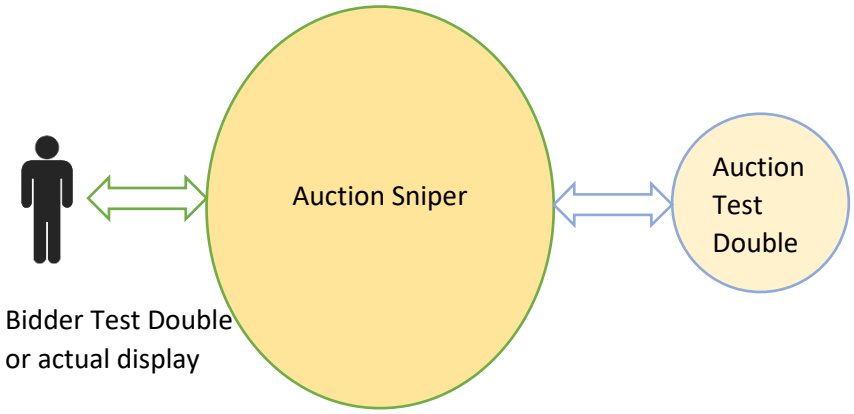
Here are some of the state transitions. First, we need to add the user identity to the background, so we'll know whether a Price message shows us or someone else is the bidder.

```
Background:
Given auction exists for:
| Auction House  | Item  |
| Southabee      | 2134  |
And user is
| User ID  |
| Self     |

Scenario: Transition Joined to Losing
# Note that Maximum Bid is low
When joining auction
| Item | Maximum Bid |
| 2134 | 1           |
When message sent
| Type | Notes           |
| Join | Status = Joined |
And message received
| Type    | Current Price | Increment  | Bidder  |
| Price   | 110           | 10         | Other   |
Then output shows
| Status   |
| Losing   |

Scenario: Transition Losing to Bidding
When joining auction
| Item | Maximum Bid |
| 2134 | 1           |
And message sent
| Type | Notes           |
| Join | Status = Joined |
And message received
| Type    | Curent Price  | Increment  | Bidder  | Notes           |
| Price   | 110           | 10         | Other   | Status = Losing |
```

```
And bid increased
| Item | Maximum Bid |
| 2134 | 200         |
And message sent
| Type  | Price  |
| Bid   | 120    |
Then output shows
| Status  |
| Bidding |
```

## Test Double

An auction test double would be useful in exercising these scenarios. Its behavior would be to wait for a message such as Join and then respond with one or more messages, as shown in the previous scenarios.




The previous scenarios seem a bit wordy (e.g., a sequence of “message sent” and “message received”). We could change the scenarios by using a slightly different interface that describes both the sent message (received by the auction) and the received message (sent by the auction). The test double would wait for a received message and then send the next message or messages. The delay, if not specified, would be immediately after the received or previously sent message.

The status column shows what the status should be once the message is sent or received. It is not used by the test double, but it could be used by the test to check that the status is correct. Here’s an example of a message sequence. Note that the column headers are on two lines to fit the table onto the page.

```
Scenario: Message sequence with Status
* Messages sent and received with status
| Delay     | Sent    |         | Received  |               |            |         | Status    |
|           | Type    | Price   | Type      | Current Price | Increment  | Bidder  |           |
|           | Join    |         |           |               |            |         | Joining   |
|           |         |         | Price     | 110           | 10         | Other   | Joined    |
|           | Bid     | 120     |           |               |            |         | Bidding   |
```

```
|           |        |        | Price      | 120           | 10          | Self    | Winning   |
|           |        |        | Price      | 210           | 10          | Other   | Losing    |
```

The above sequence could check that the proper status was being computed. Using this same interface and adding a delay of some time, say 5 seconds, allows one to visibly see the status changing on the display

```
Scenario: Message sequence with slow status changes on the display
* Messages sent and received with status
| Delay     | Sent    |        | Received   |               |             |         | Status    |
|           | Type    | Price  | Type       | Current Price | Increment   | Bidder  |           |
|     5     | Join    |        |            |               |             |         | Joining   |
|     5     |         |        | Price      | 110           | 10          | Other   | Joined    |
|     5     | Bid     | 120    |            |               |             |         | Bidding   |
|     5     |         |        | Price      | 120           | 10          | Self    | Winning   |
|     5     |         |        | Price      | 210           | 10          | Other   | Losing    |
```

A sequence with 0 delay between messages could check the display's response to quick updates.

```
Scenario: Message sequence with rapid Status changes on the display
* Messages sent and received with status
| Delay     | Sent    |        | Received   |               |             |         | Status    |
|           | Type    | Price  | Type       | Current Price | Increment   | Bidder  |           |
|     0     | Join    |        |            |               |             |         | Joining   |
|     0     |         |        | Price      | 110           | 10          | Other   | Joined    |
|     0     |         |        | Price      | 120           | 10          | Other   | Joined    |
|     0     |         |        | Price      | 130           | 10          | Other   | Joined    |
|     0     |         |        | Price      | 140           | 10          | Other   | Joined    |
```

A set of message sequences could be intermixed with user inputs, such as

```
Scenario: Update maximum bid during auction
When joining auction
| Item | Maximum Bid |
| 2134 | 1           |
Then message sequence is
| Delay     | Sent    |        | Received   |               |             |         | Status    |
|           | Type    | Price  | Type       | Current Price | Increment   | Bidder  |           |
|           | Join    |        |            |               |             |         | Joining   |
|           |         |        | Price      | 110           | 10          | Other   | Joined    |
|           | Bid     | 120    |            |               |             |         | Bidding   |
|           |         |        | Price      | 120           | 10          | Self    | Winning   |
|           |         |        | Price      | 210           | 10          | Other   | Losing    |
And bid increased
| Item | Maximum Bid |
| 2134 | 200         |
Then message sequence is
| Delay     | Sent    |        | Received   |               |             |         | Status    |
|           | Type    | Price  | Type       | Current Price | Increment   | Bidder  |           |
|           | Bid     | 220    |            |               |             |         |           |
|           |         |        | Price      | 220           | 10          | Self    | Winning   |
|           |         |        | Price      | 350           | 10          | Other   | Bidding   |
|           | Bid     | 360    |            |               |             |         |           |
|           |         |        | Price      | 400           | 10          | Other   | Losing    |
|           |         |        | Close      |               |             |         | Lost      |
```

## Some Details

Those who have looked at other examples of ATDD/BDD scenarios may notice a difference in style. The *Given/When/Then* steps in these examples give all of the data in the tables that follow. The phrasing of each of the steps is common (e.g., "Message sent", rather than "Price message sent"). This allows the step definitions (the code that connects the steps to the production code) to be the same. If additional

data is needed, additional columns can be added to the tables. If data is not needed for a particular scenario, the column can be left off and the data filled in from a default value.

The Auction test double represents the Auction production interface along with any necessary test setup methods. There are at least two implementations for the test double:

- Same process
- Separate process (e.g., for servers, micro-servers, etc.)

The same scenario can be used to setup either one, just the step definition changes to reflect the current implementation.

Anyone can create a more complicated scenario – with lots of Prices, Bids, Maximum Bid changes. Multiple concurrent auctions could use multiple test doubles.

## Summary

As Jerry Weinberg said, "If you can't come up with three solutions to a problem, then you probably don't understand the problem". Likewise, there are many approaches to developing a solution. Having multiple ways to do something gives you the opportunity to try the different approaches to see which one works best for you in a particular context. Try out the ATDD/BDD approach to the next feature you are implementing and see if it helps replace misunderstanding with shared understanding.