

# Acceptance Testing for Continuous Delivery

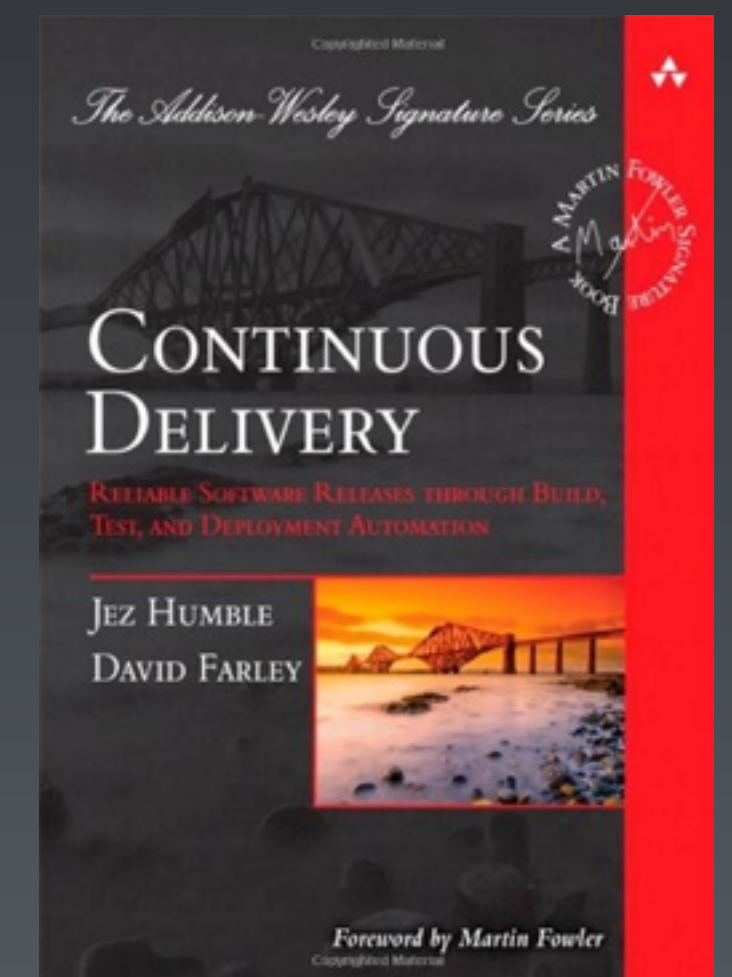
**Dave Farley**

<http://www.davefarley.net>

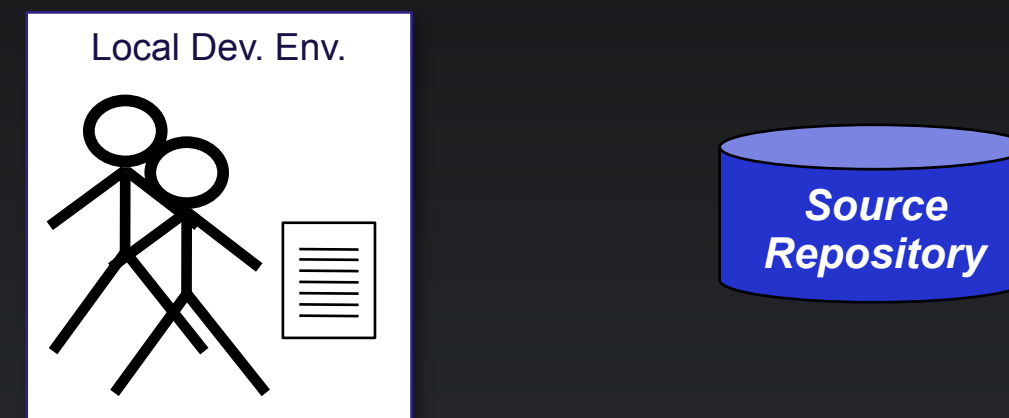
@davefarley77



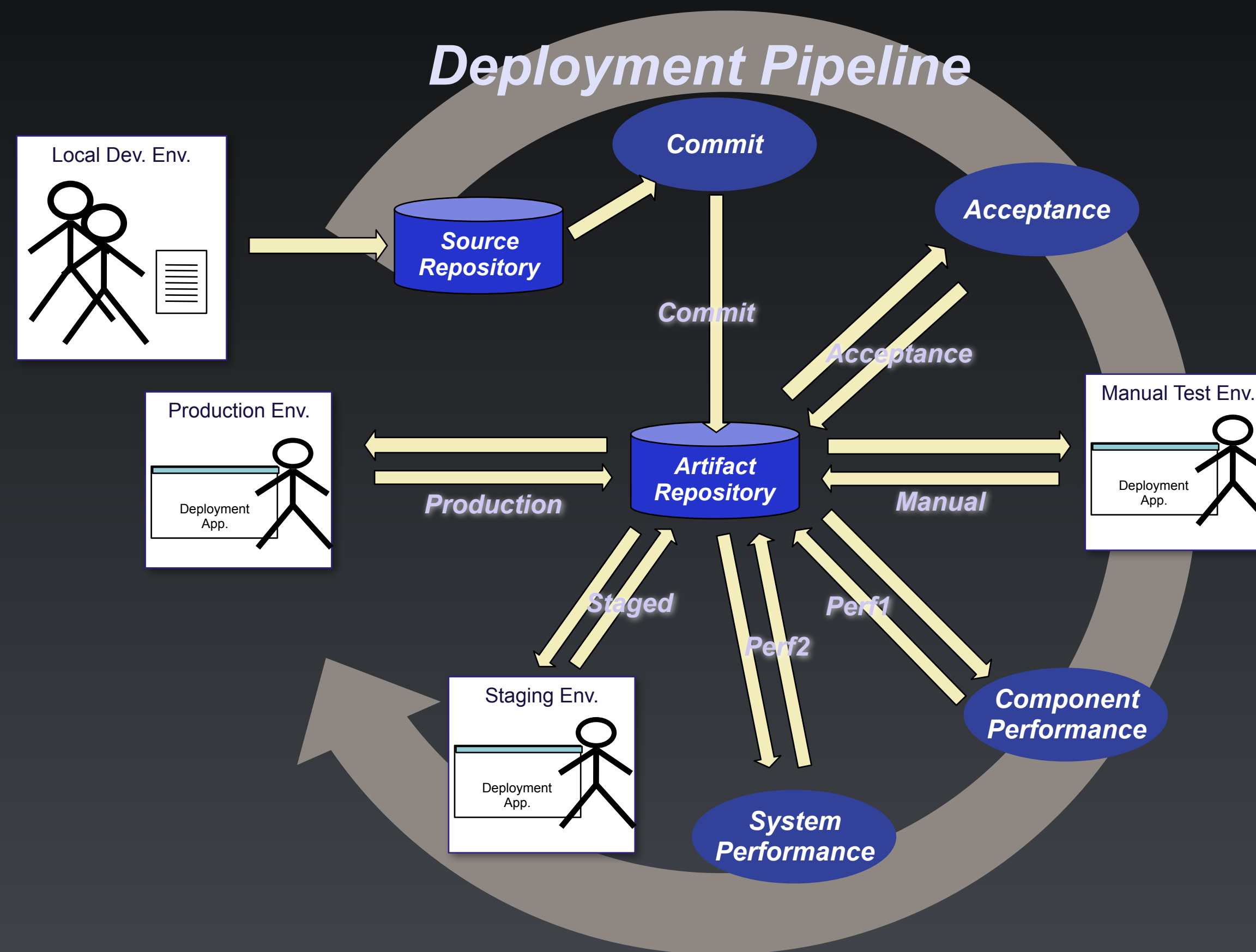
<http://www.continuous-delivery.co.uk>



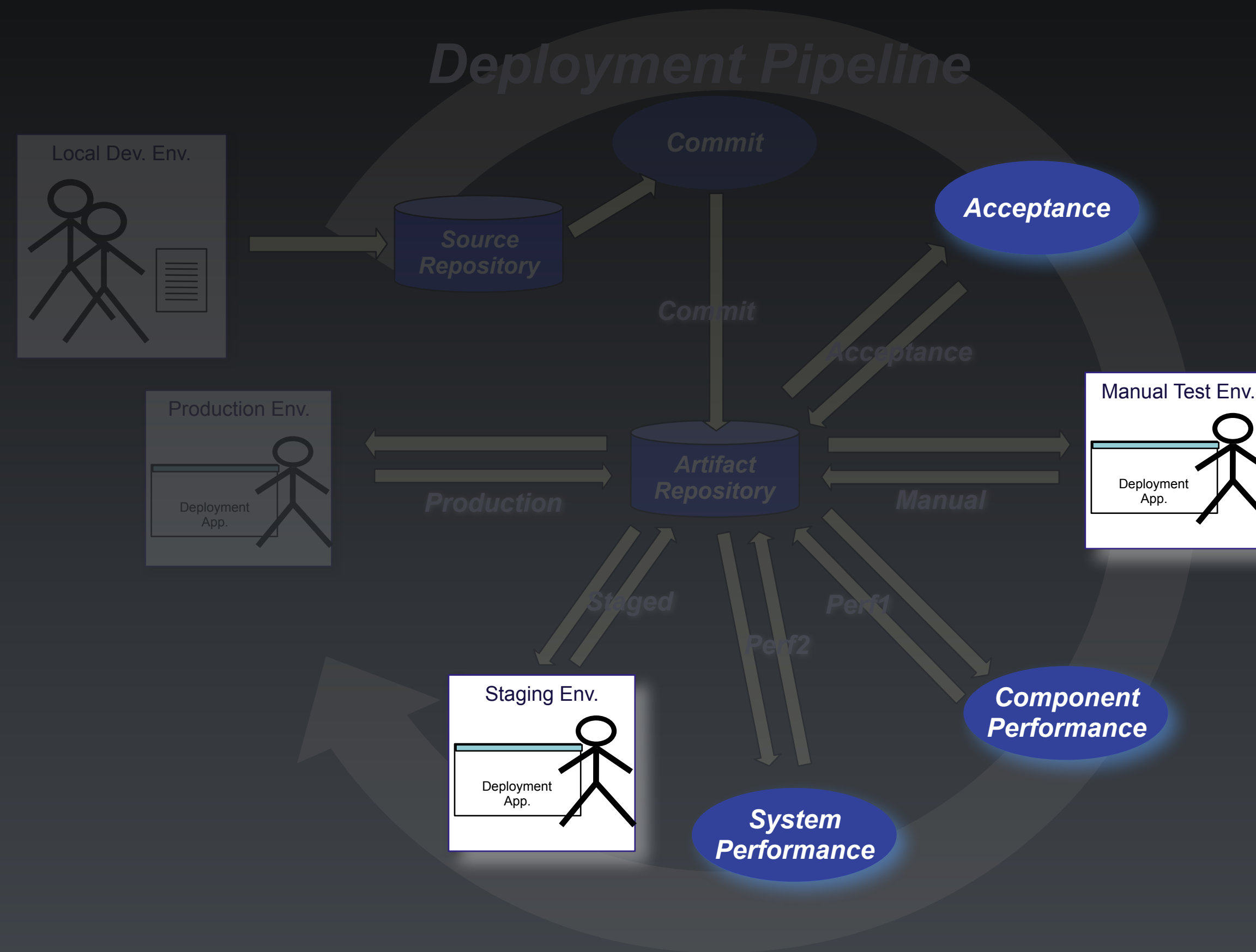
# The Role of Acceptance Testing



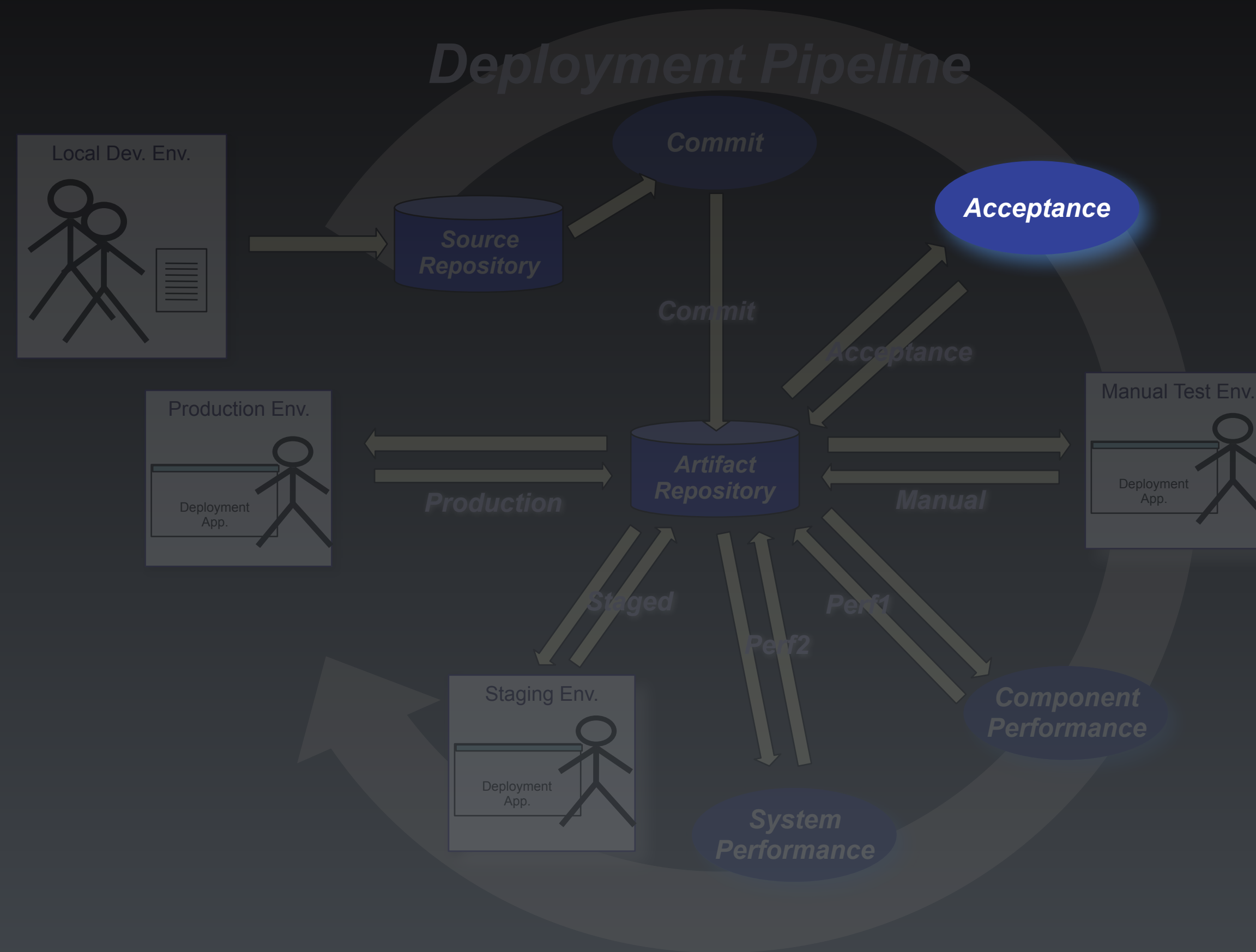
# The Role of Acceptance Testing



# The Role of Acceptance Testing



# The Role of Acceptance Testing



# What is Acceptance Testing?

Asserts that the code does what the users want.





# What is Acceptance Testing?

An automated “definition of done”





# What is Acceptance Testing?

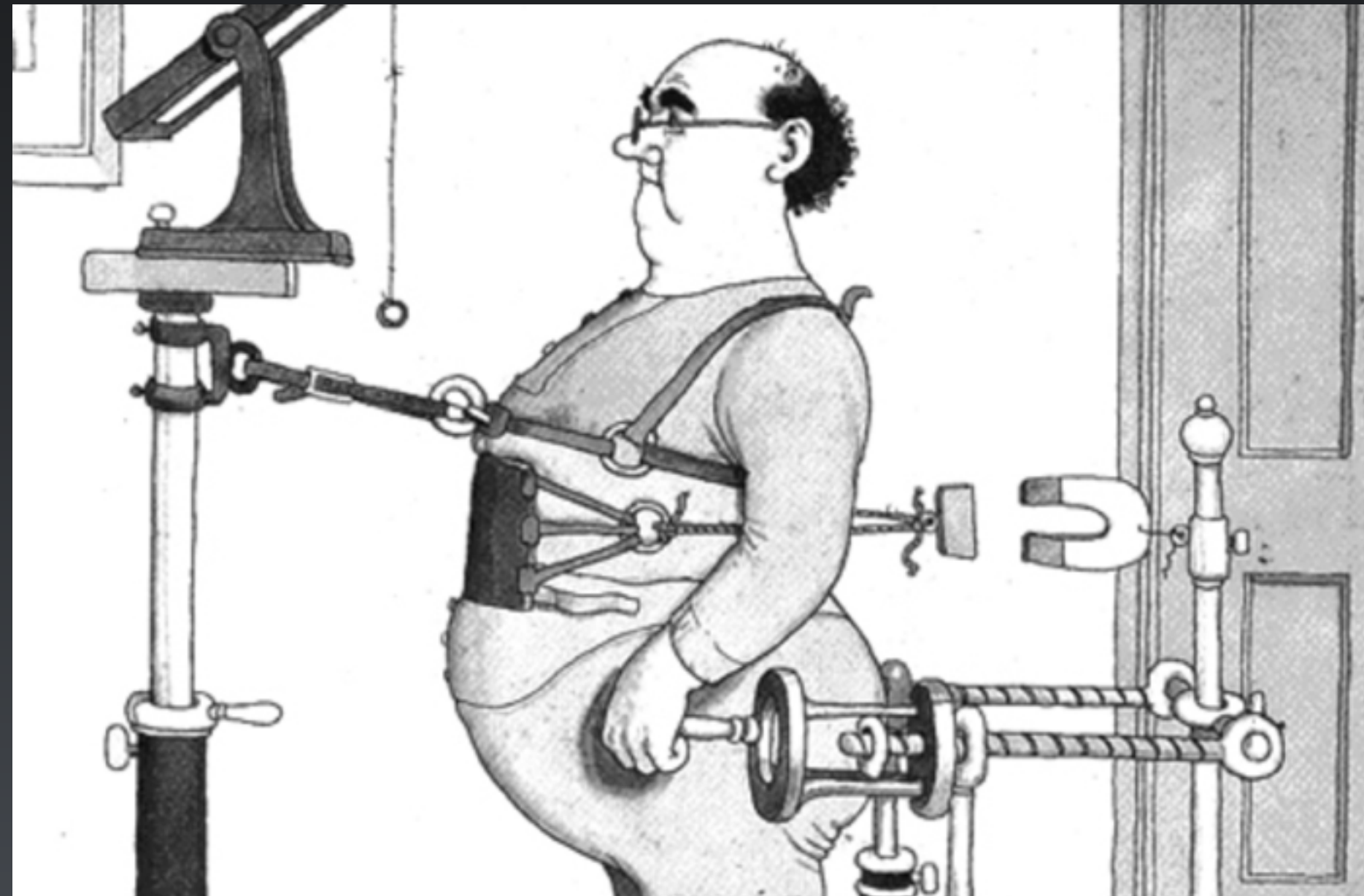
Asserts that the code works in a “production-like” test environment.





# What is Acceptance Testing?

A test of the deployment and configuration of a whole system.



# What is Acceptance Testing?

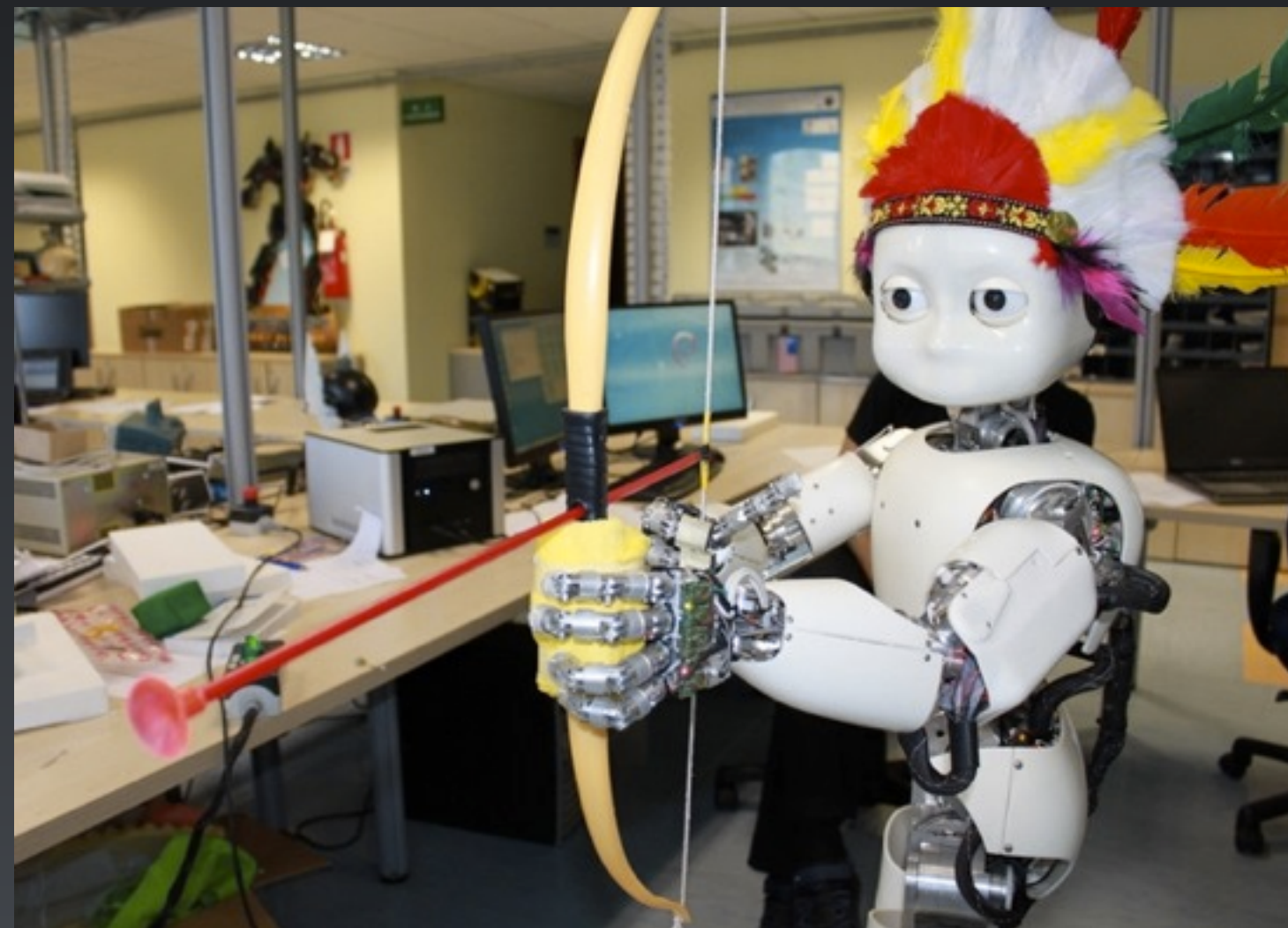
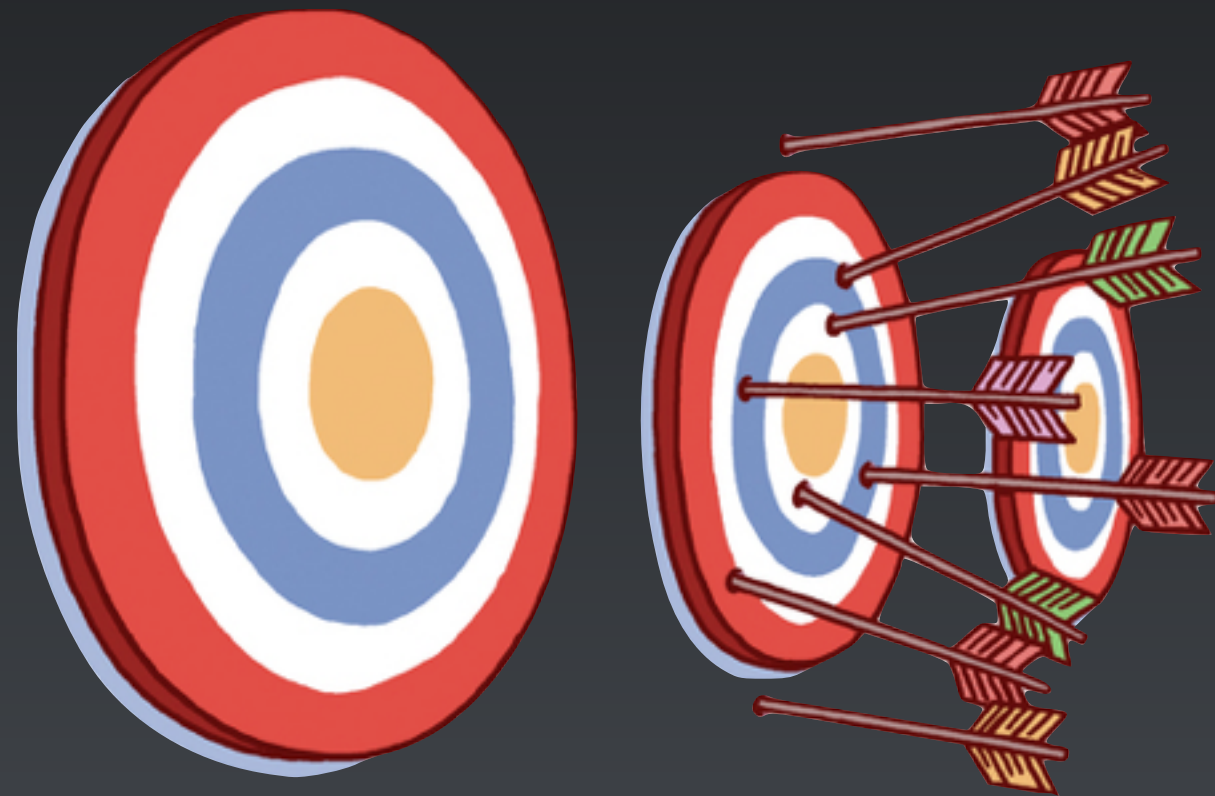
Provides timely feedback on stories - closes a feedback loop.





# What is Acceptance Testing?

Acceptance Testing, ATDD, BDD, Specification by Example, Executable Specifications.





# What is Acceptance Testing?

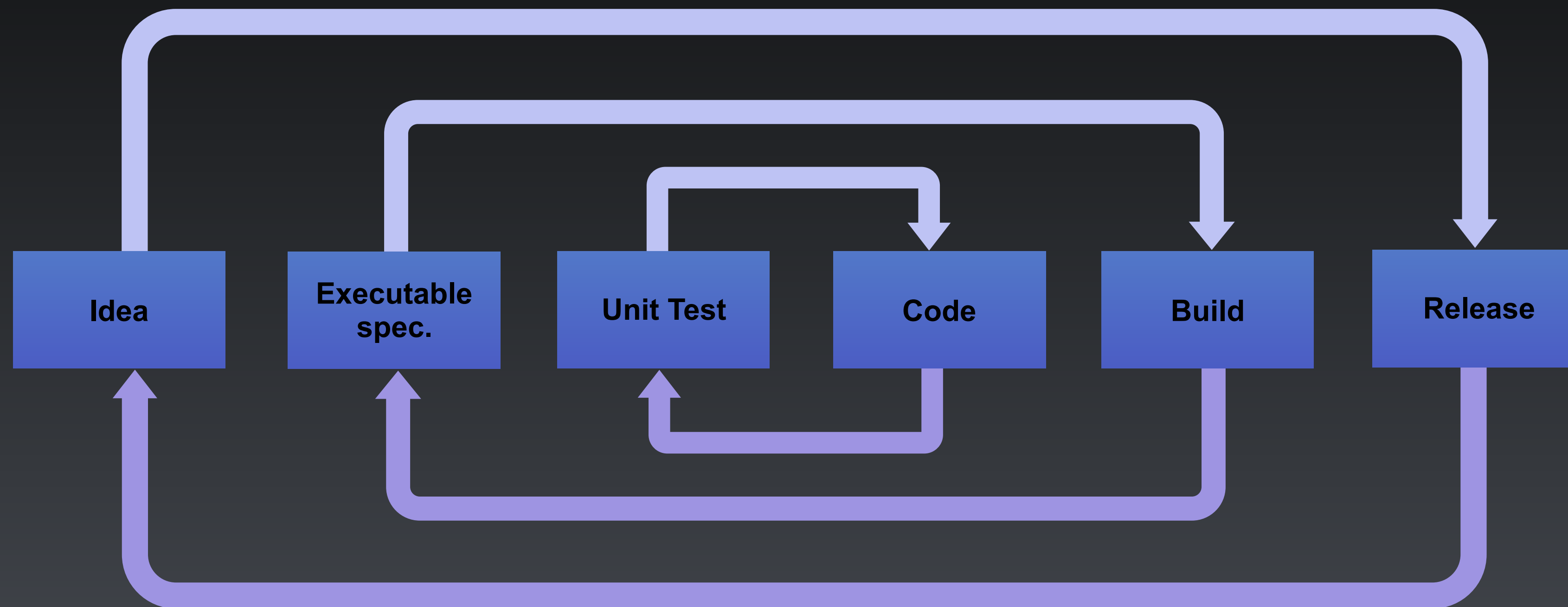
**A Good Acceptance Test is:**

**An Executable Specification of the  
Behaviour of the System**

# What is Acceptance Testing?

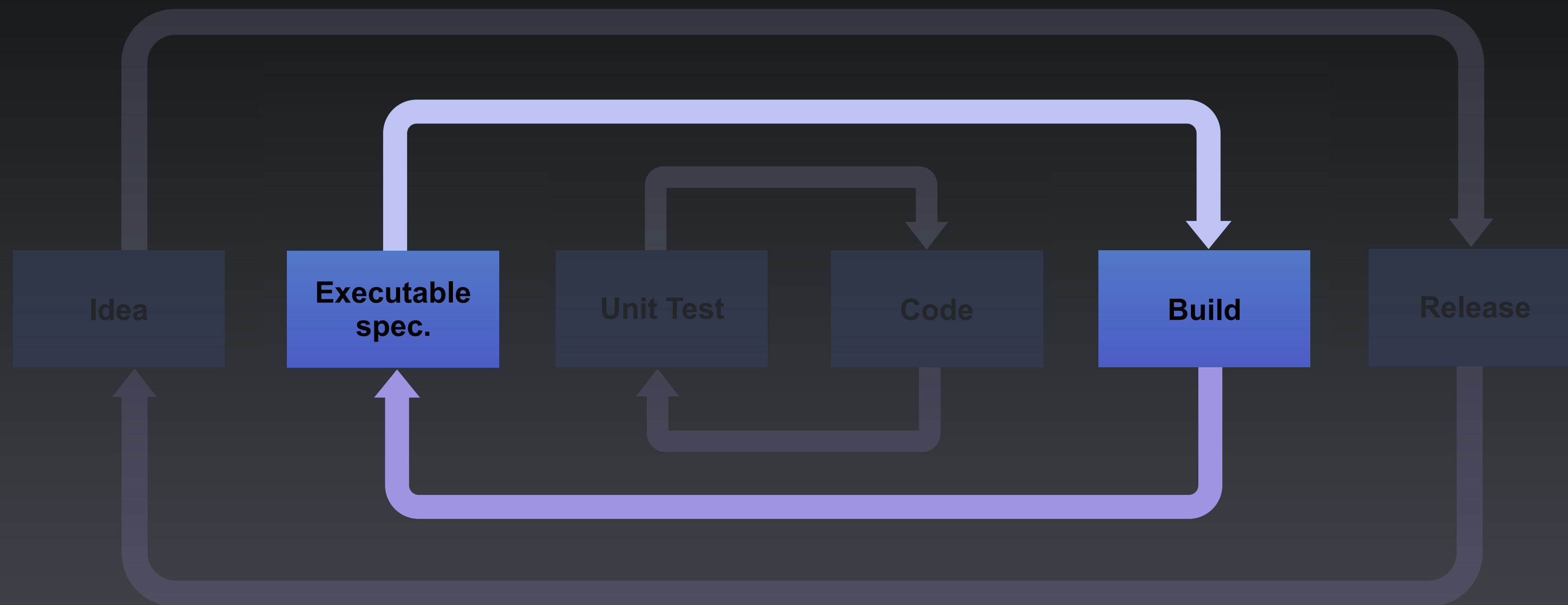


# What is Acceptance Testing?





# What is Acceptance Testing?



# So What's So Hard?

- Tests break when the SUT changes (Particularly UI)
- Tests are complex to develop
- This is a problem of design, the tests are too tightly-coupled to the SUT!
- The history is littered with poor implementations:
  - UI Record-and-playback Systems
  - Record-and-playback of production data
  - Dumps of production data to test systems
  - Nasty automated testing products.

# So What's So Hard?

- Tests break when the SUT changes (Particularly UI)
- Tests are complex to develop
- This is a problem of design, the tests are too tightly-coupled to the SUT!
- The history is littered with poor implementations:
  - UI Record-and-playback Systems
  - Record-and-playback of production data
  - Dumps of production data to test systems
  - Nasty automated testing products.



# Who Owns the Tests?

- Anyone can write a test
- Developers are the people that will break tests
- Therefore Developers own the responsibility to keep them working
- Separate Testing/QA team owning automated tests

# Who Owns the Tests?

- Anyone can write a test
- Developers are the people that will break tests
- Therefore Developers own the responsibility to keep them working
- ~~Separate Testing/QA team owning automated tests~~

# Who Owns the Tests?

**Developers Own  
Acceptance Tests!**



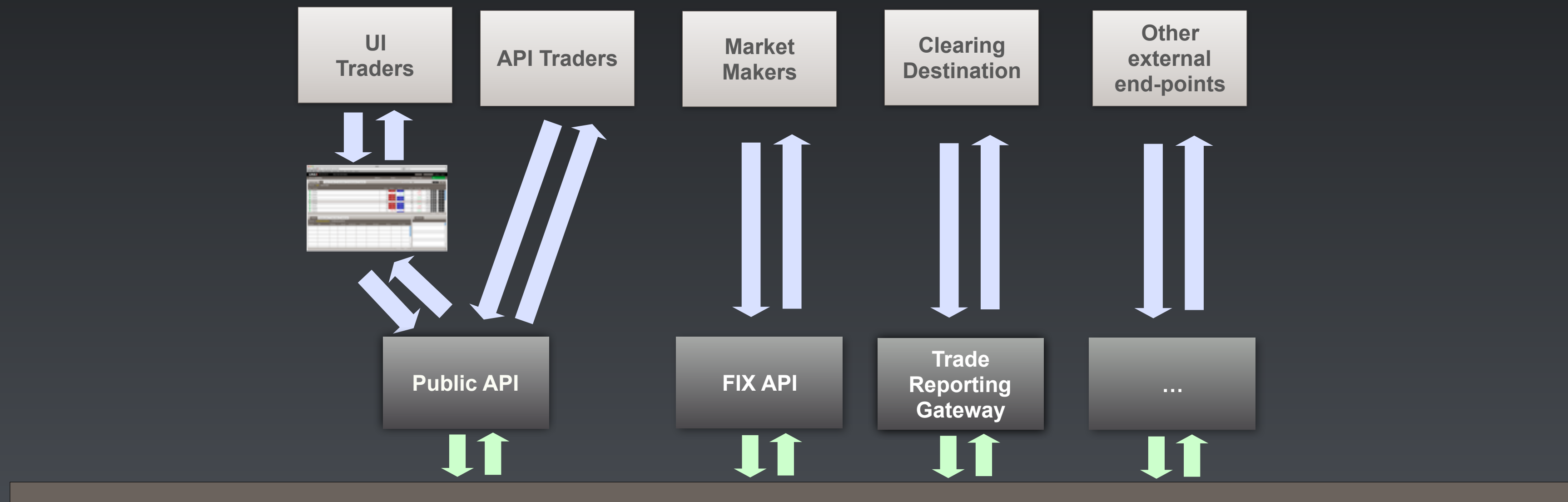
# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

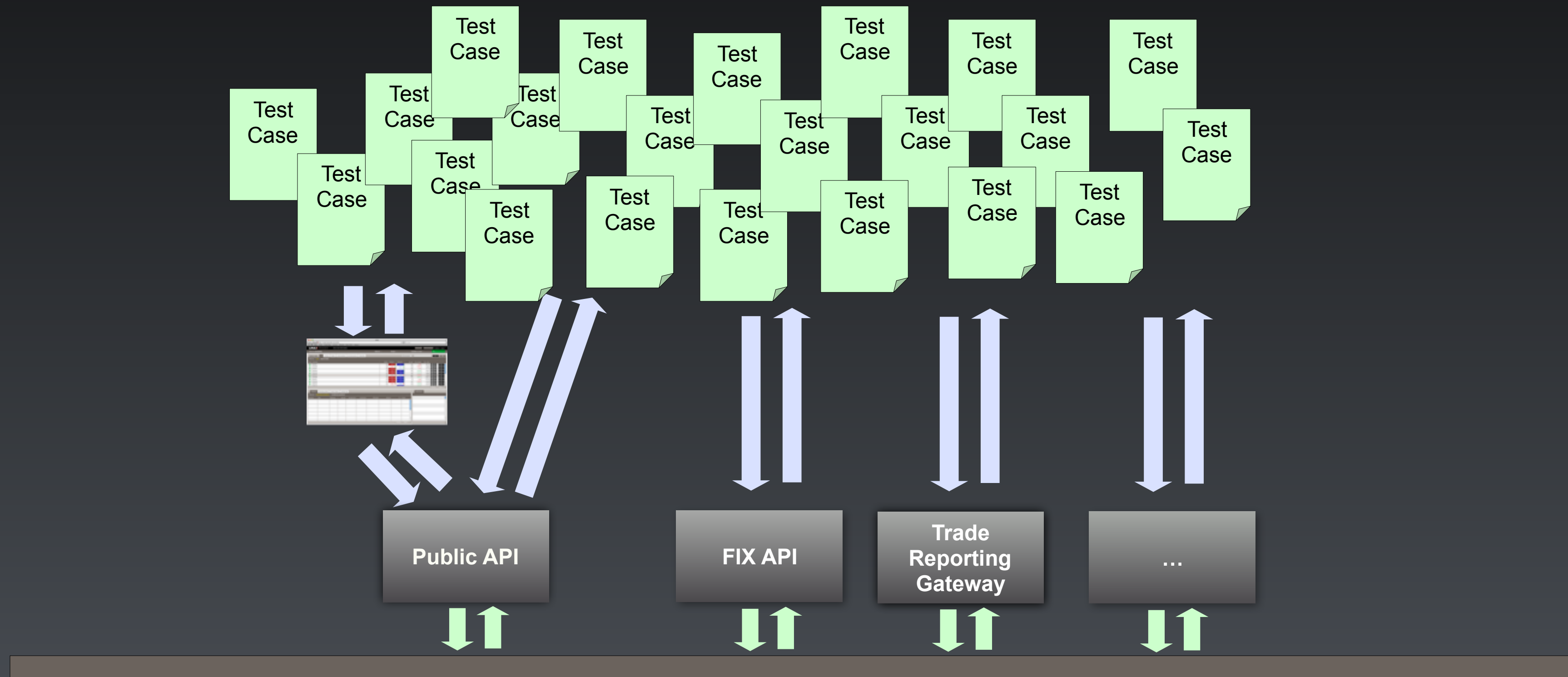
# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

# “What” not “How”

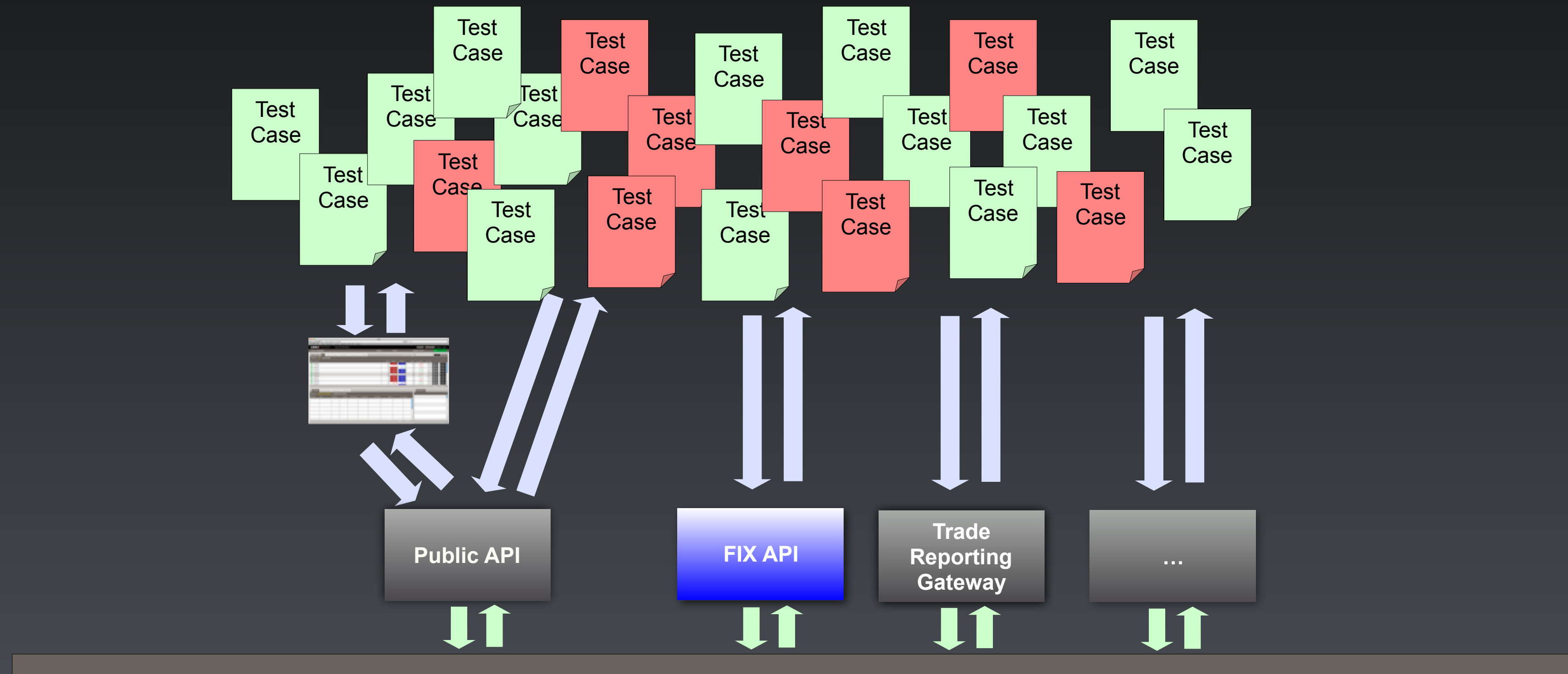


# “What” not “How”

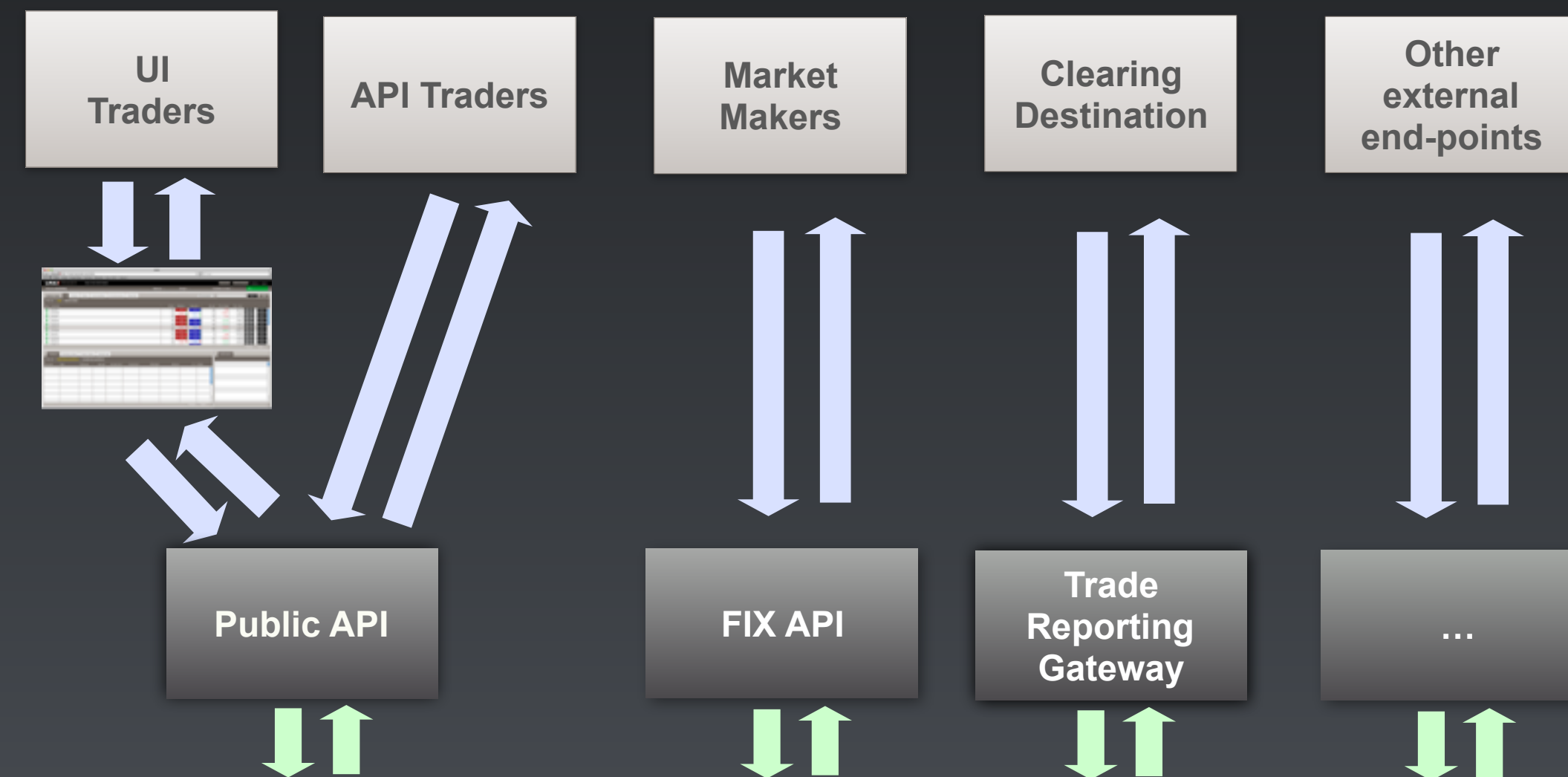
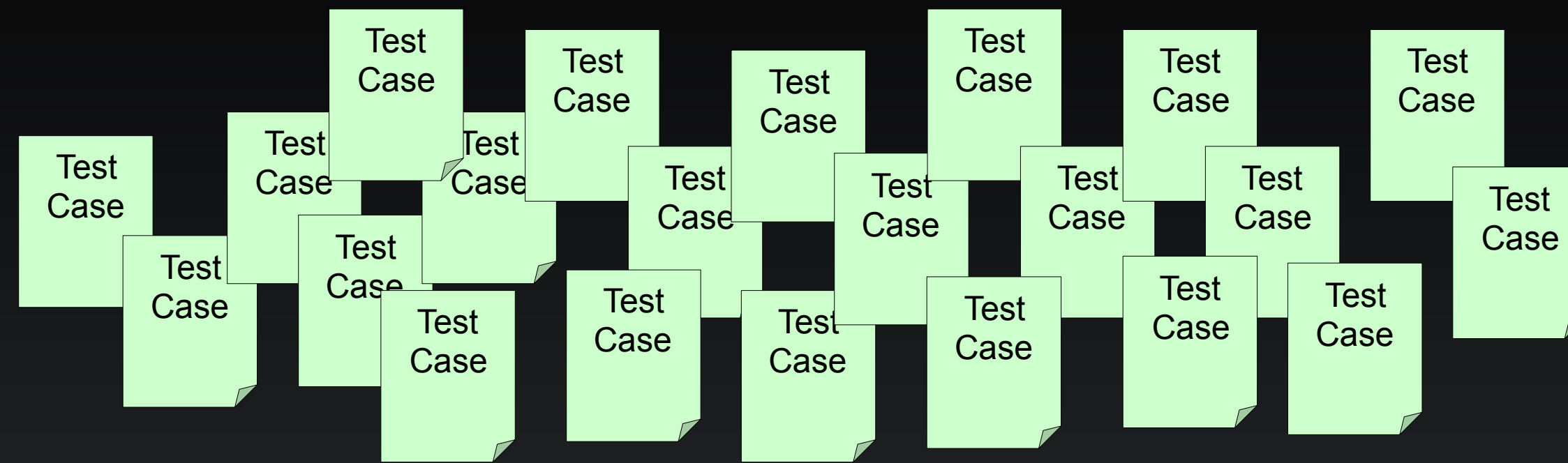




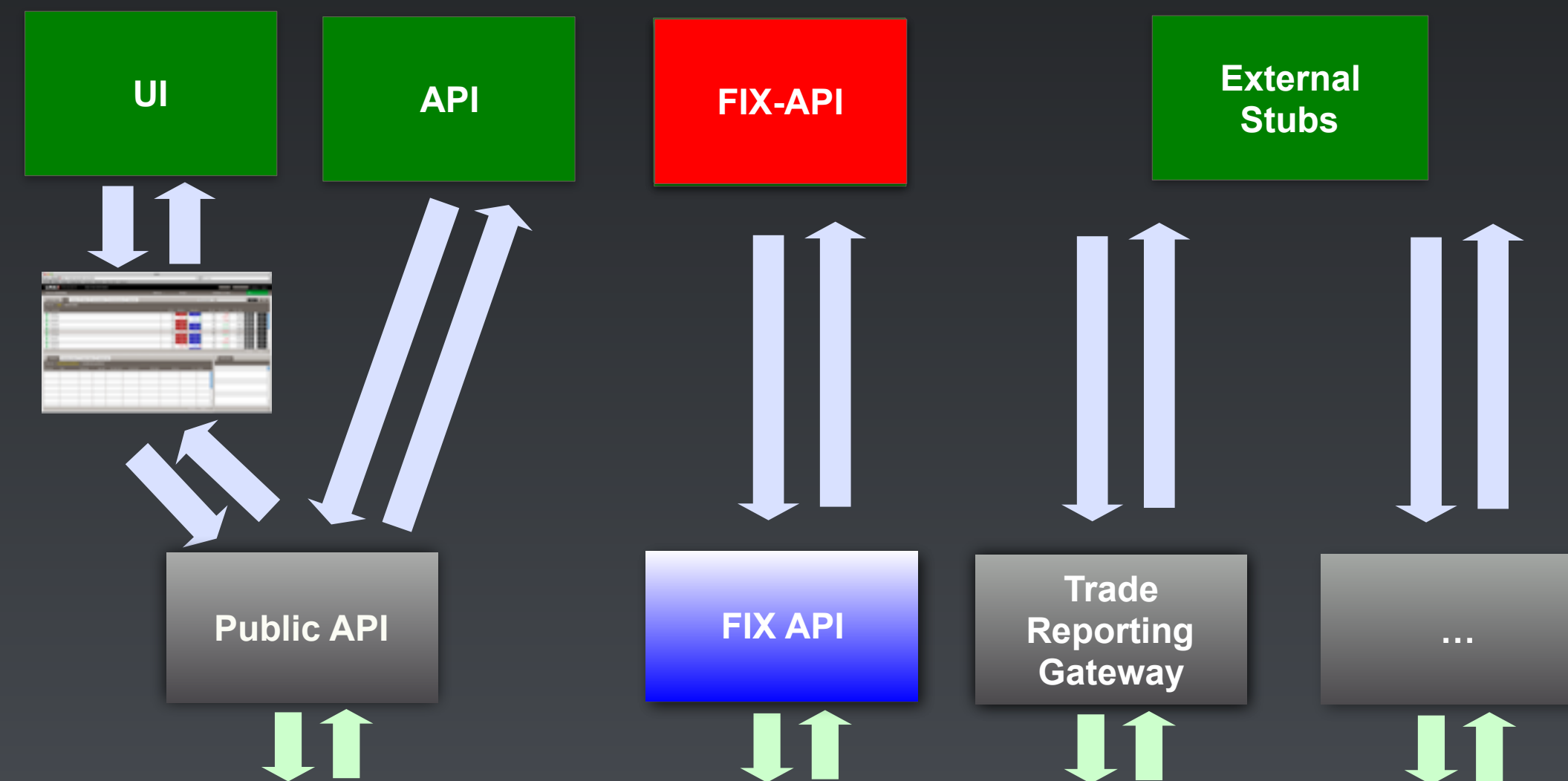
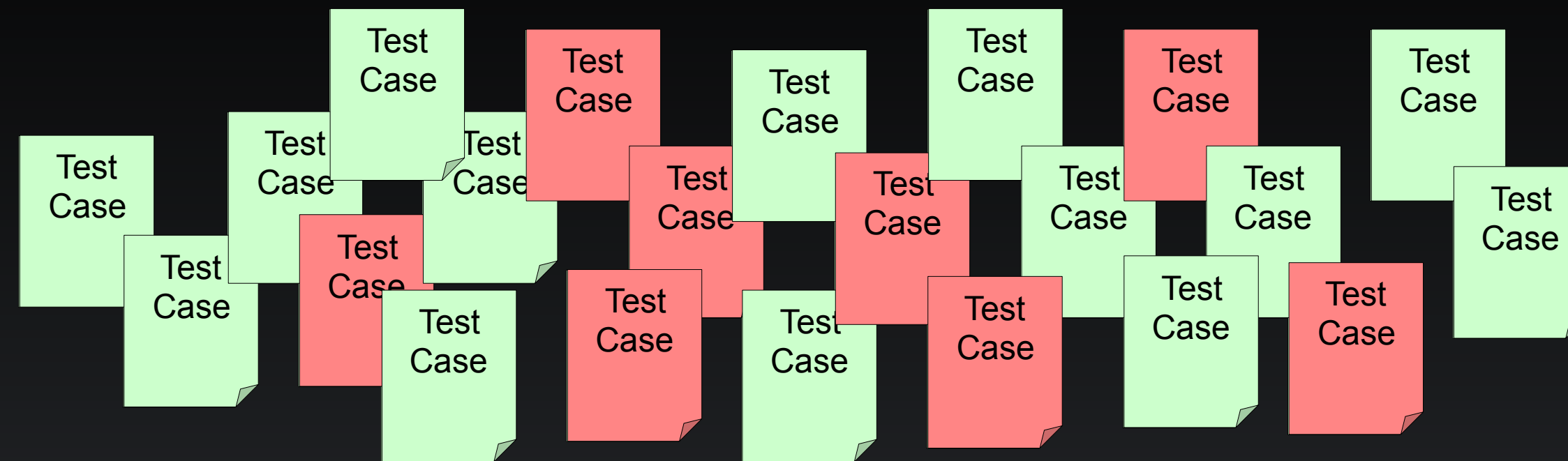
# “What” not “How”



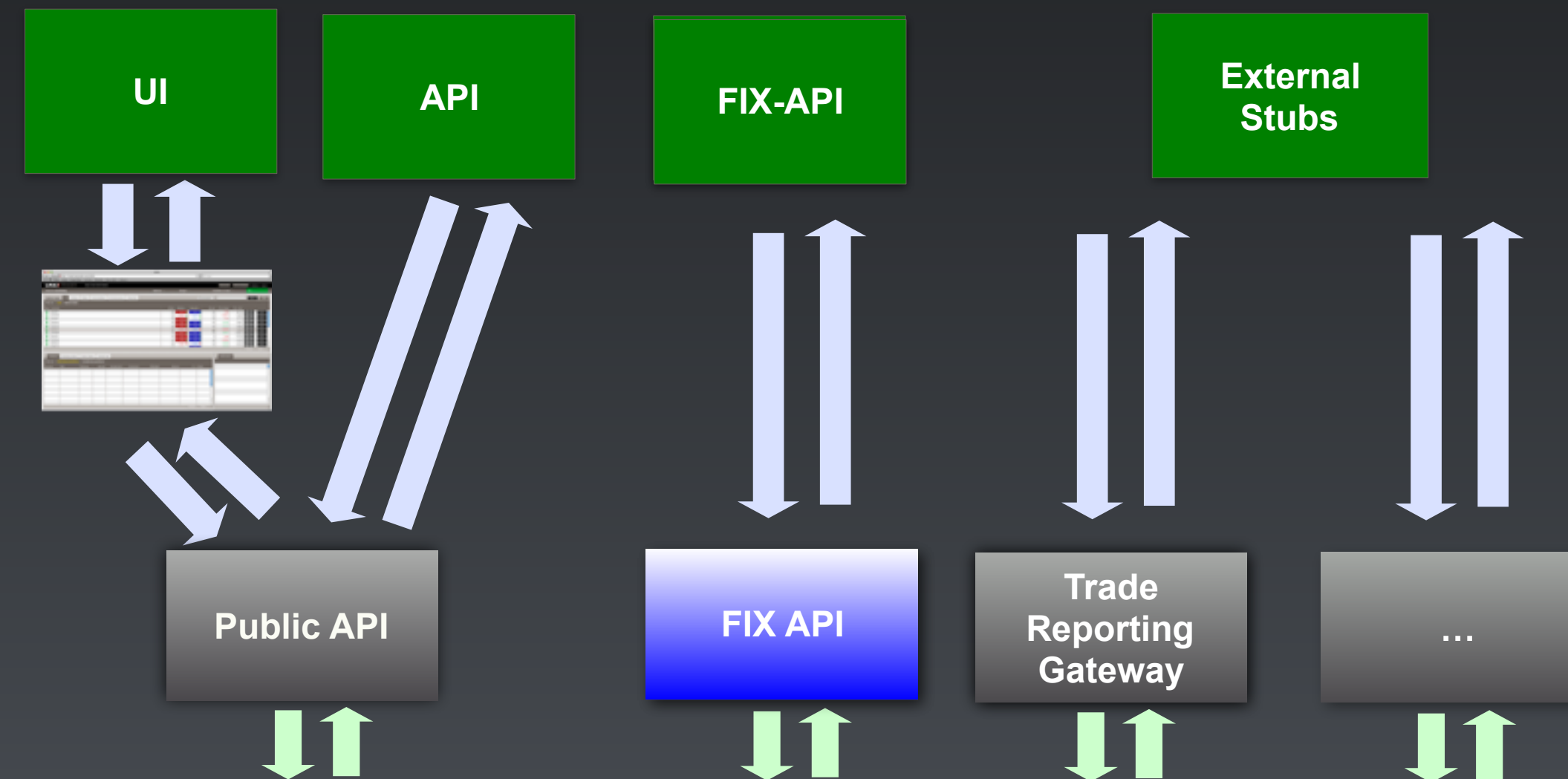
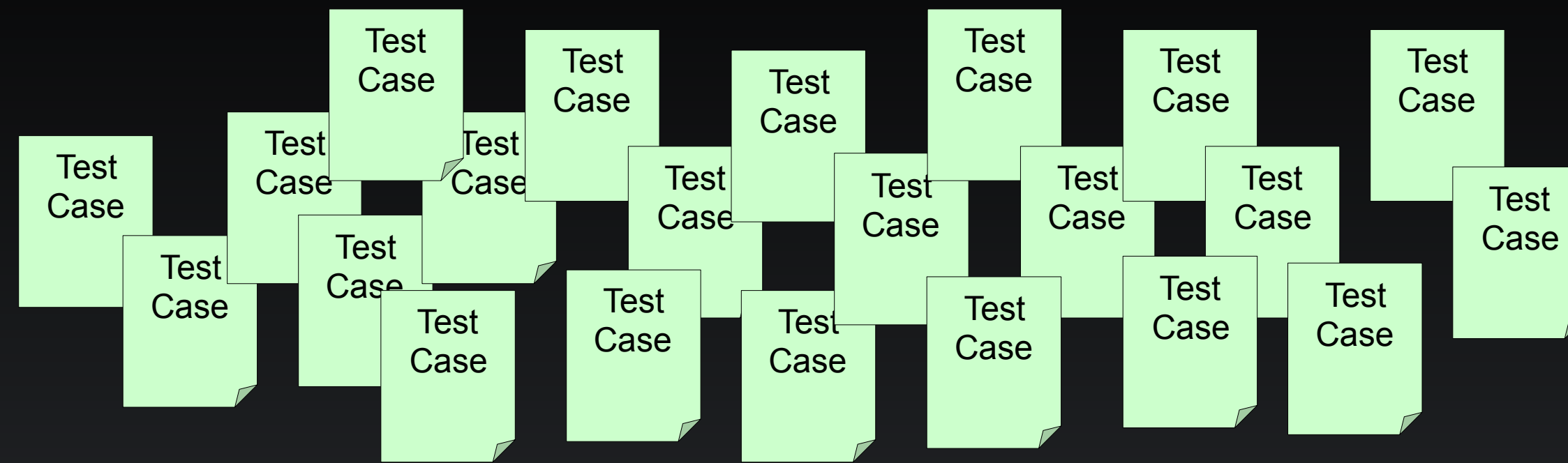
# “What” not “How”



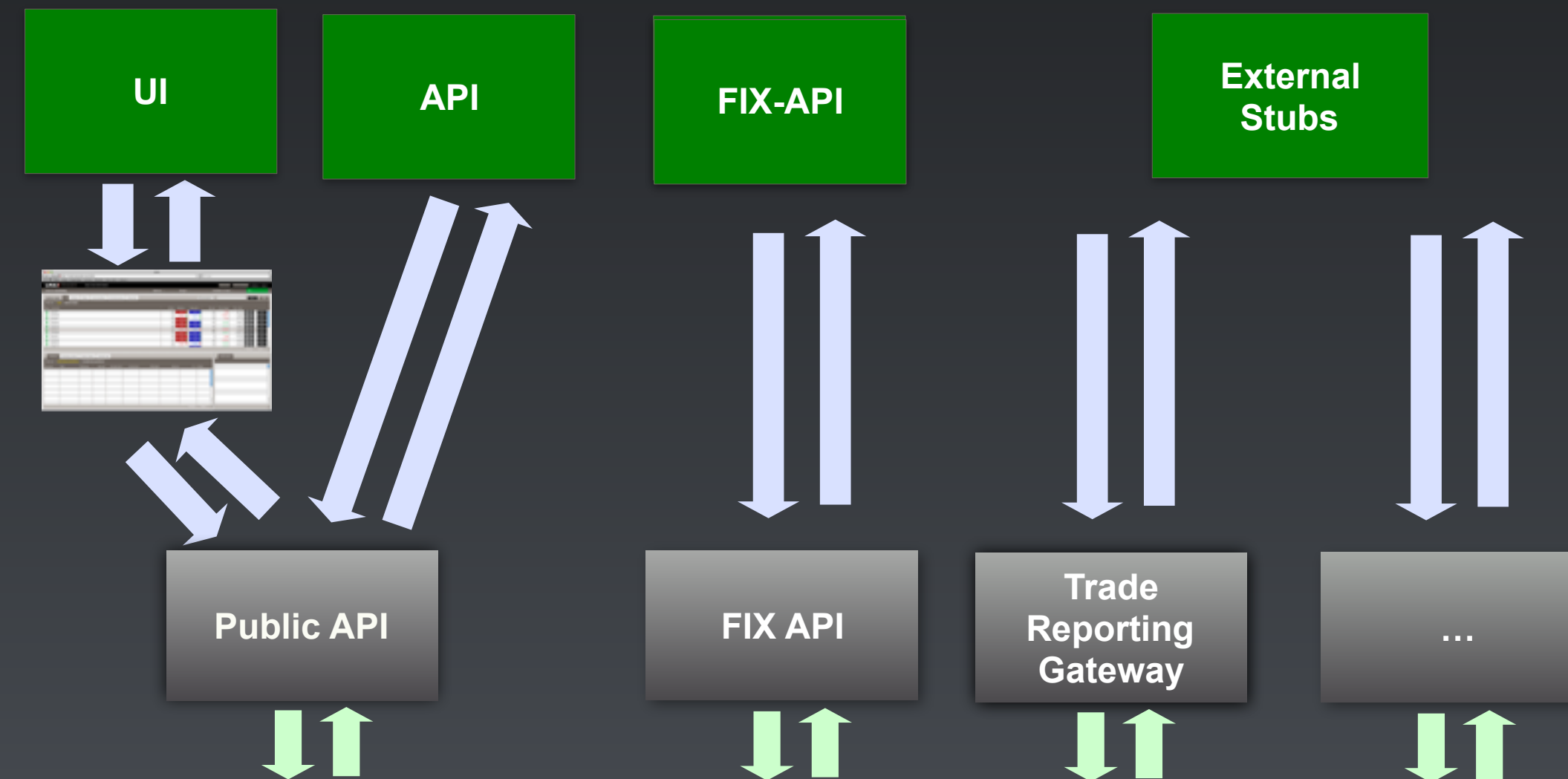
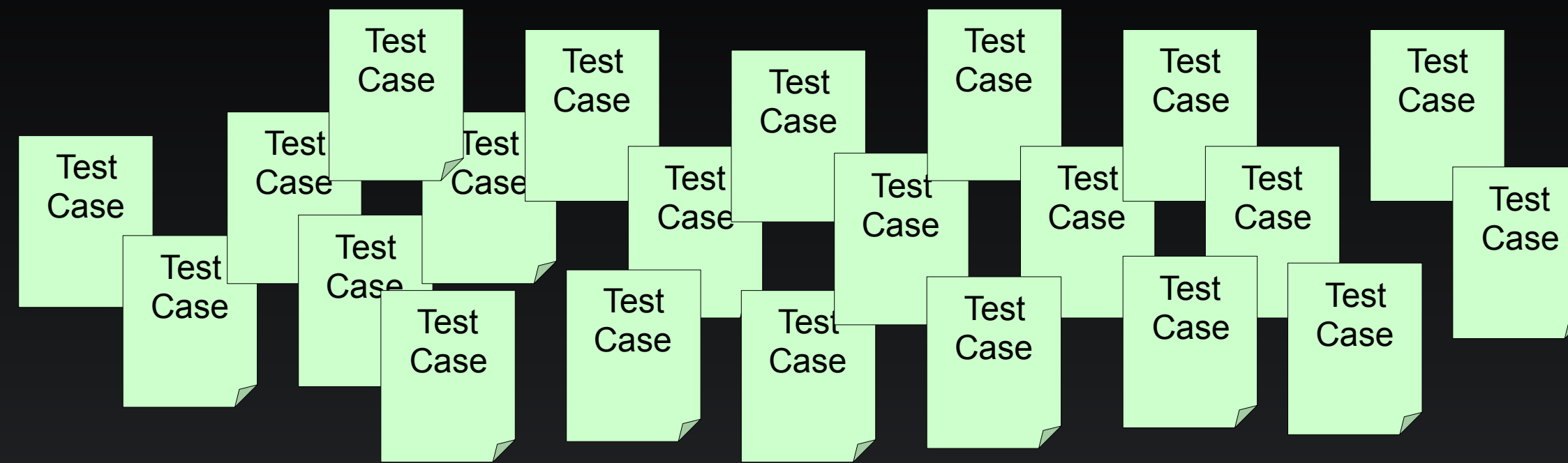
# “What” not “How”



# “What” not “How”

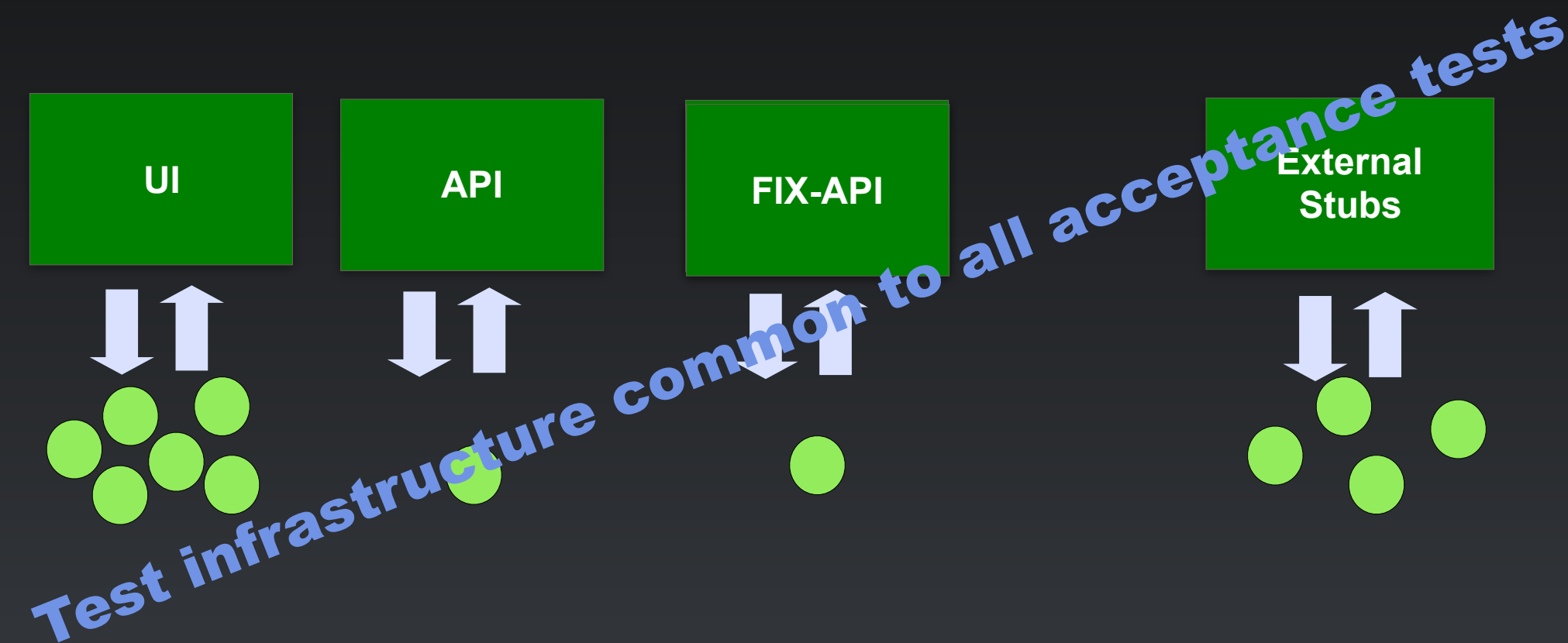


# “What” not “How”





# “What” not “How”



# “What” not “How” - Separate Deployment from Testing

- Every Test should control its start conditions, and so should start and init the app.
- Acceptance Test deployment should be a rehearsal for Production Release
- This separation of concerns provides an opportunity for optimisation
  - Parallel tests in a shared environment
  - Lower test start-up overhead

# “What” not “How” - Separate Deployment from Testing

- Every Test should control its start conditions, and so should start and init the app.
- Acceptance Test deployment should be a rehearsal for Production Release
- This separation of concerns provides an opportunity for optimisation
  - Parallel tests in a shared environment
  - Lower test start-up overhead

# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

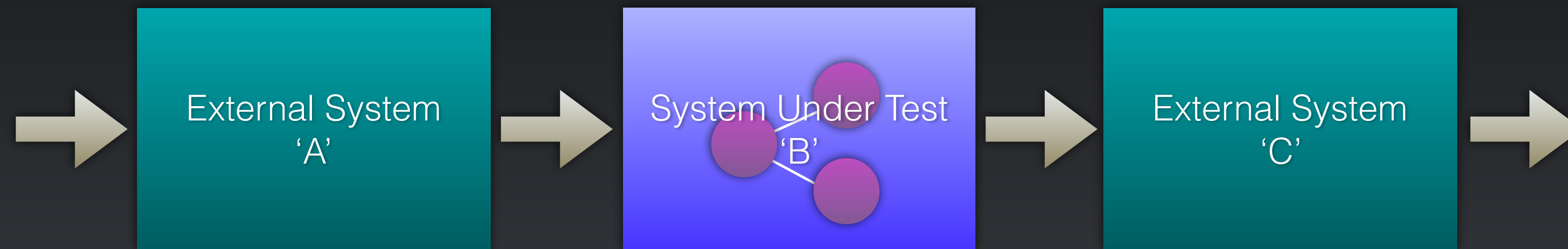


# Test Isolation

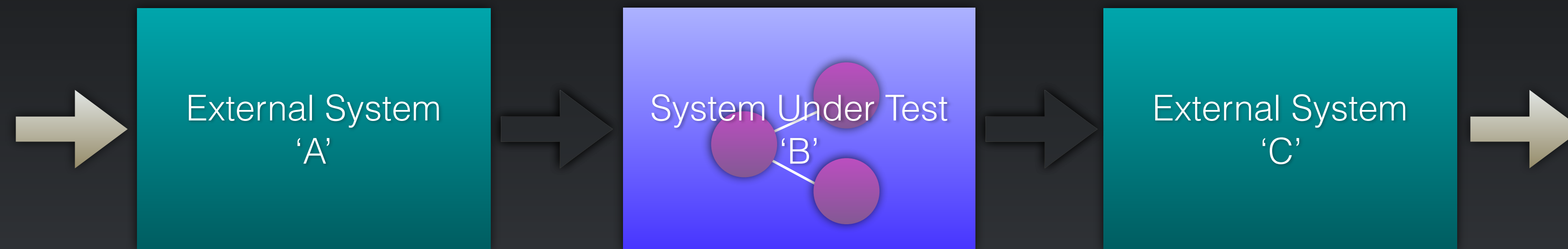
- Any form of testing is about evaluating something in controlled circumstances
- Isolation works on multiple levels
  - Isolating the System under test
  - Isolating test cases from each other
  - Isolating test cases from themselves (temporal isolation)
- Isolation is a vital part of your Test Strategy

# Test Isolation - Isolating the System Under Test

# Test Isolation - Isolating the System Under Test

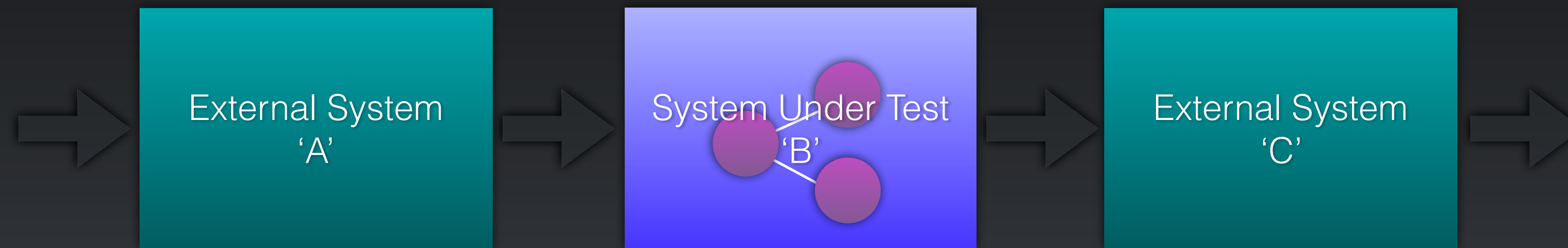


# Test Isolation - Isolating the System Under Test

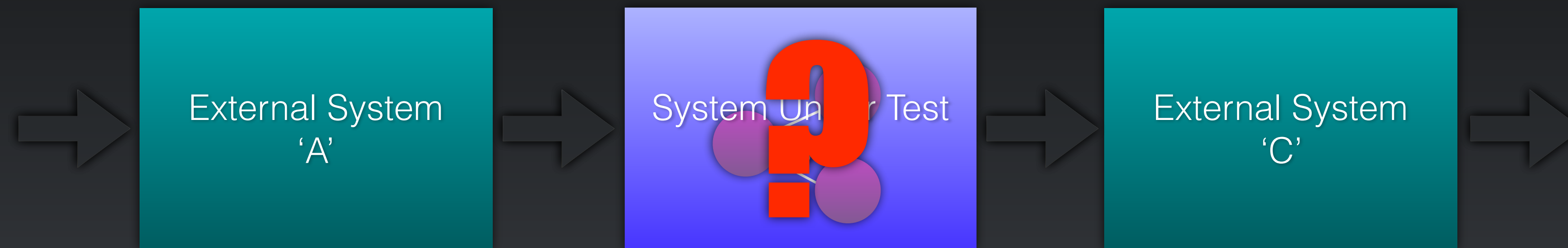




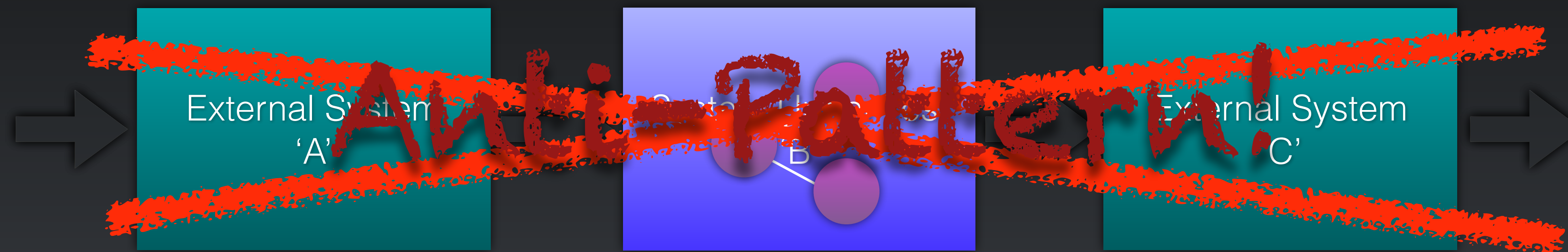
# Test Isolation - Isolating the System Under Test



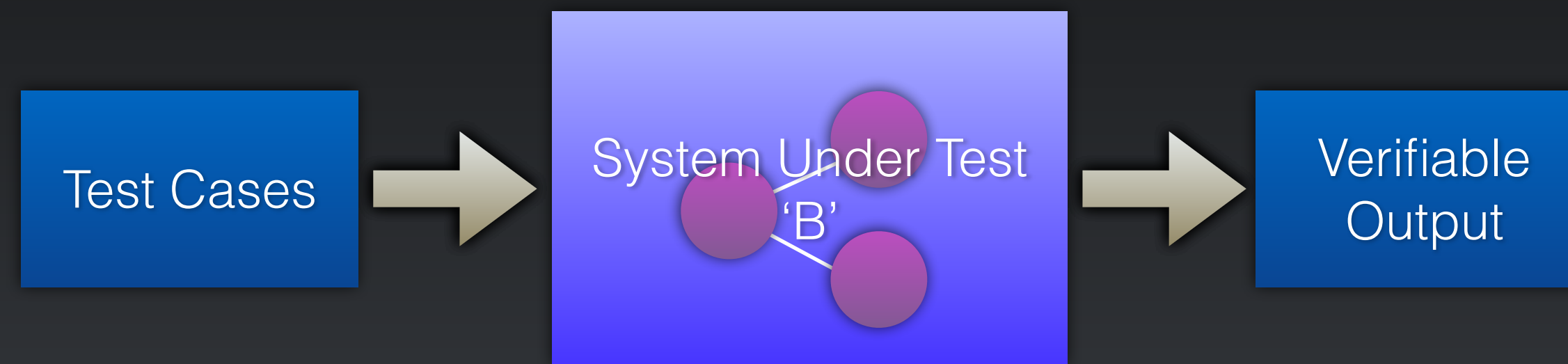
# Test Isolation - Isolating the System Under Test



# Test Isolation - Isolating the System Under Test



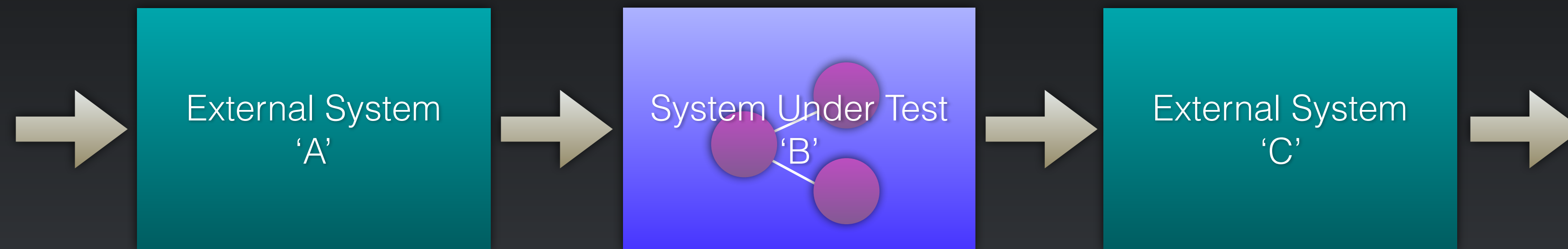
# Test Isolation - Isolating the System Under Test



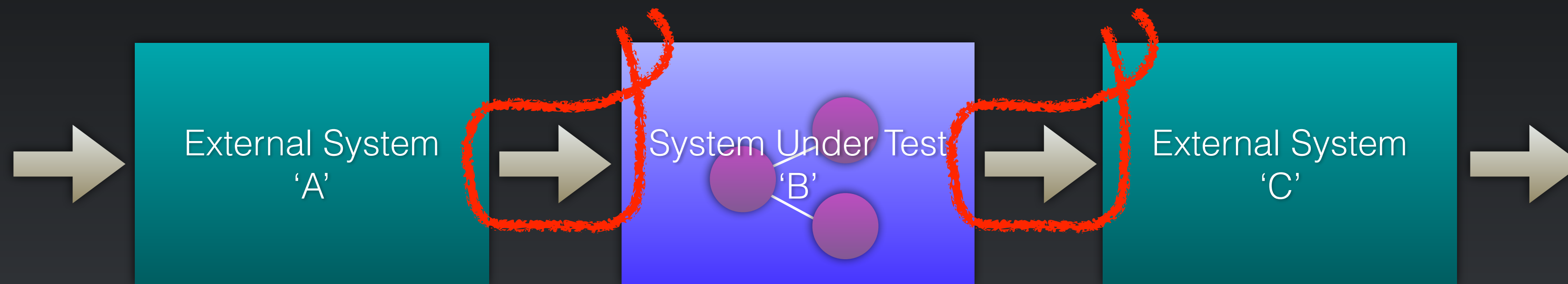


# Test Isolation - Validating The Interfaces

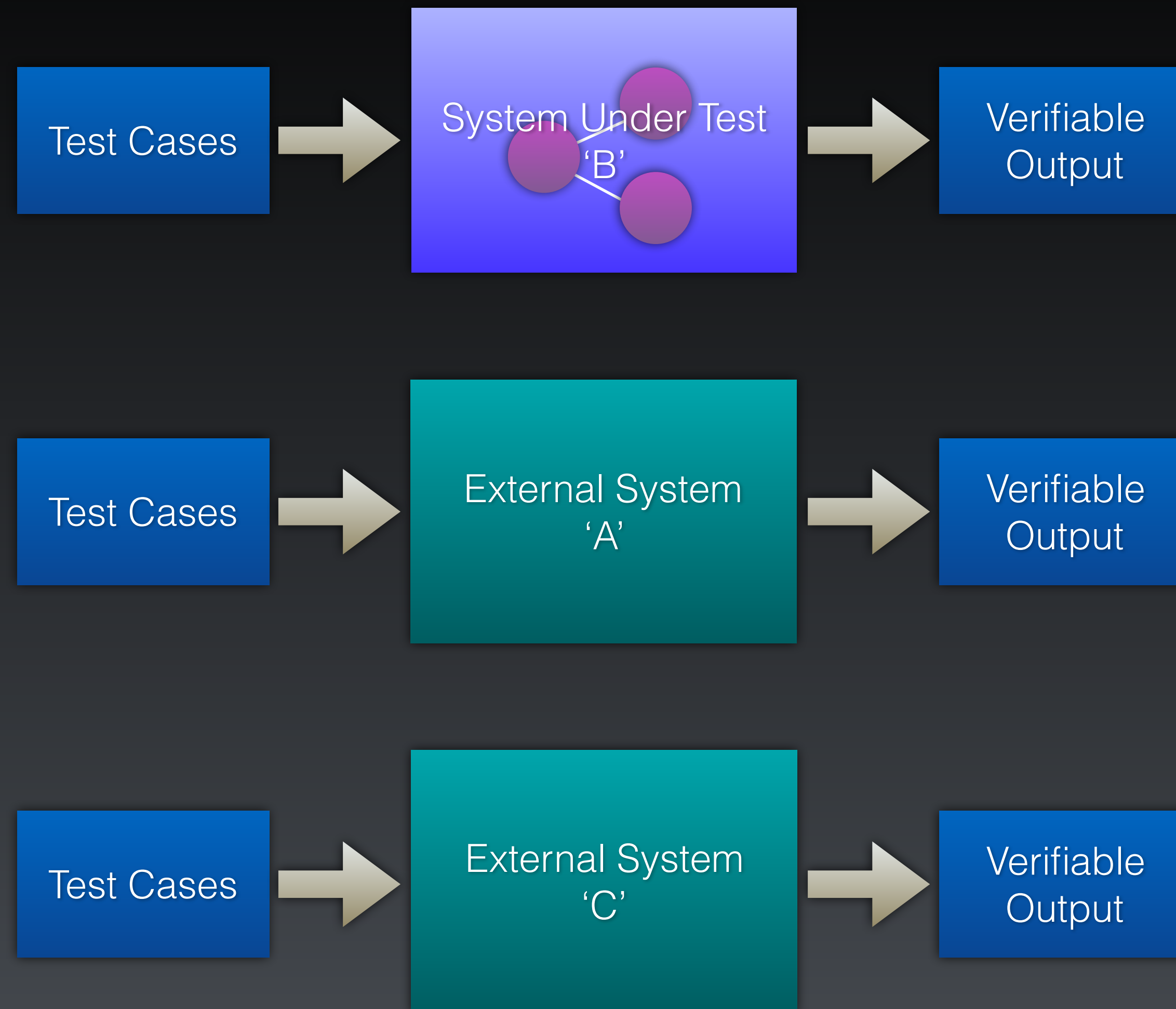
# Test Isolation - Validating The Interfaces



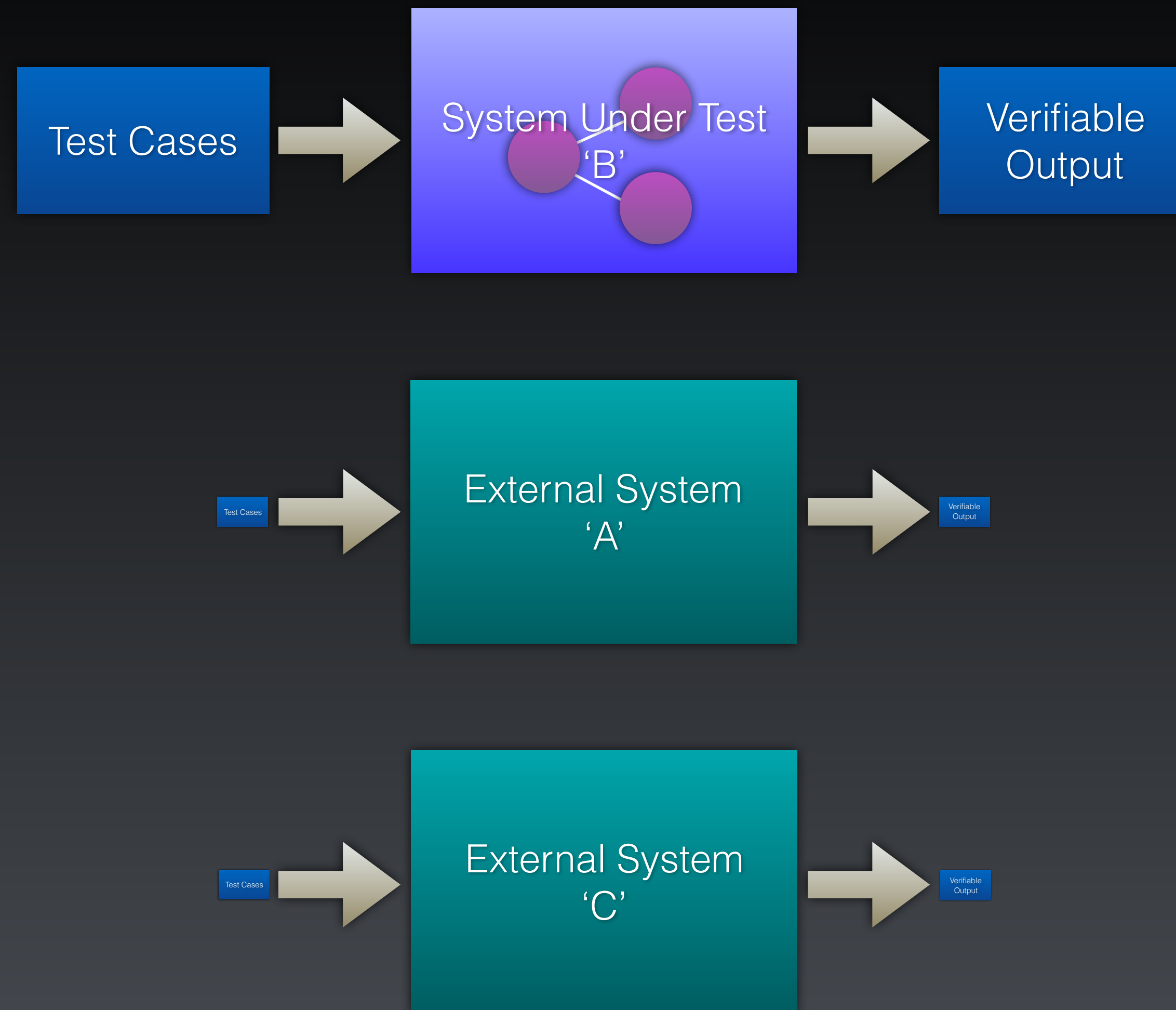
# Test Isolation - Validating The Interfaces



# Test Isolation - Validating The Interfaces



# Test Isolation - Validating The Interfaces





# Test Isolation - Isolating Test Cases

- Assuming multi-user systems...
- Tests should be efficient - We want to run LOTS!
- What we really want is to deploy once, and run LOTS of tests
- So we must avoid ANY dependencies between tests...
- Use natural functional isolation e.g.
  - If testing Amazon, create a new account and a new book/product for every test-case
  - If testing eBay create a new account and a new auction for every test-case
  - If testing GitHub, create a new account and a new repository for every test-case
  - ...

# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```

# Test Isolation - Temporal Isolation

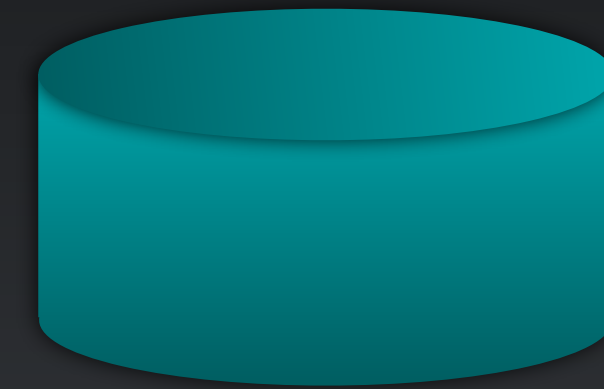
- We want repeatable results
- If I run my test-case twice it should work both times

```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```

# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```





# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

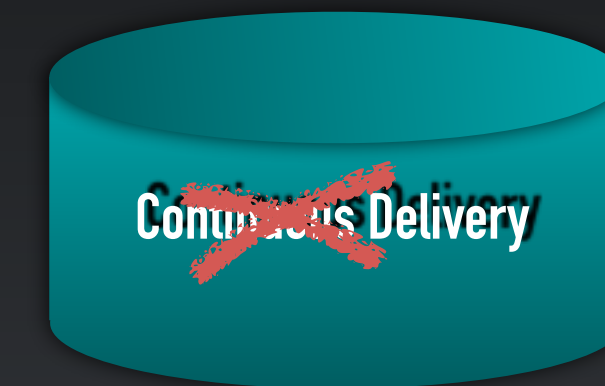
```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```



# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```



# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```



# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```



# Test Isolation - Temporal Isolation

- We want repeatable results
- If I run my test-case twice it should work both times

```
def test_should_place_an_order(self):  
    self.store.createBook("Continuous Delivery");  
  
    order = self.store.placeOrder(book="Continuous Delivery")  
  
    self.store.assertOrderPlaced(order)
```



- Alias your functional isolation entities
  - In your test case create account 'Dave' in reality, in the test infrastructure, ask the application to create account 'Dave2938472398472' and alias it to 'Dave' in your test infrastructure.



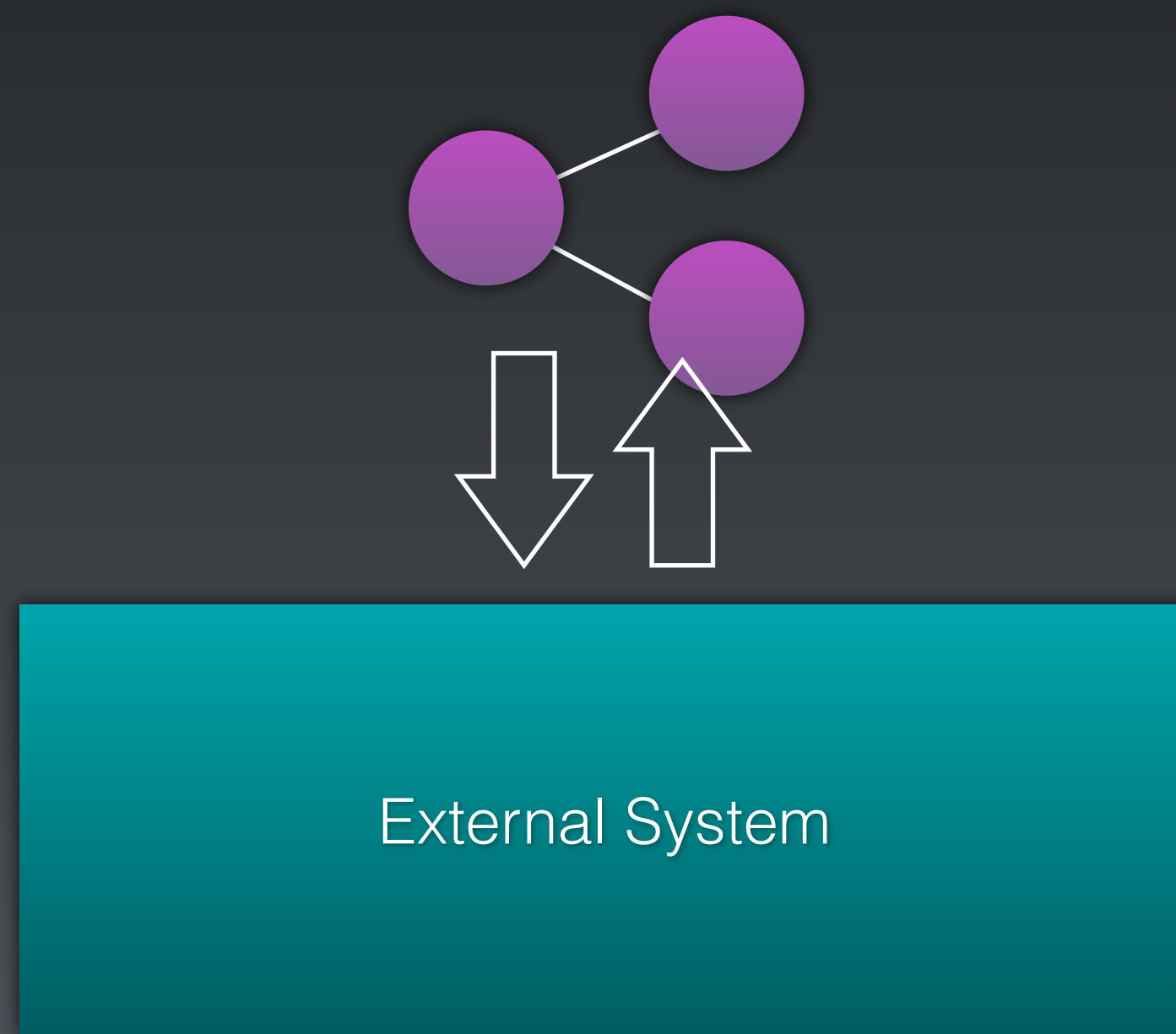
# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

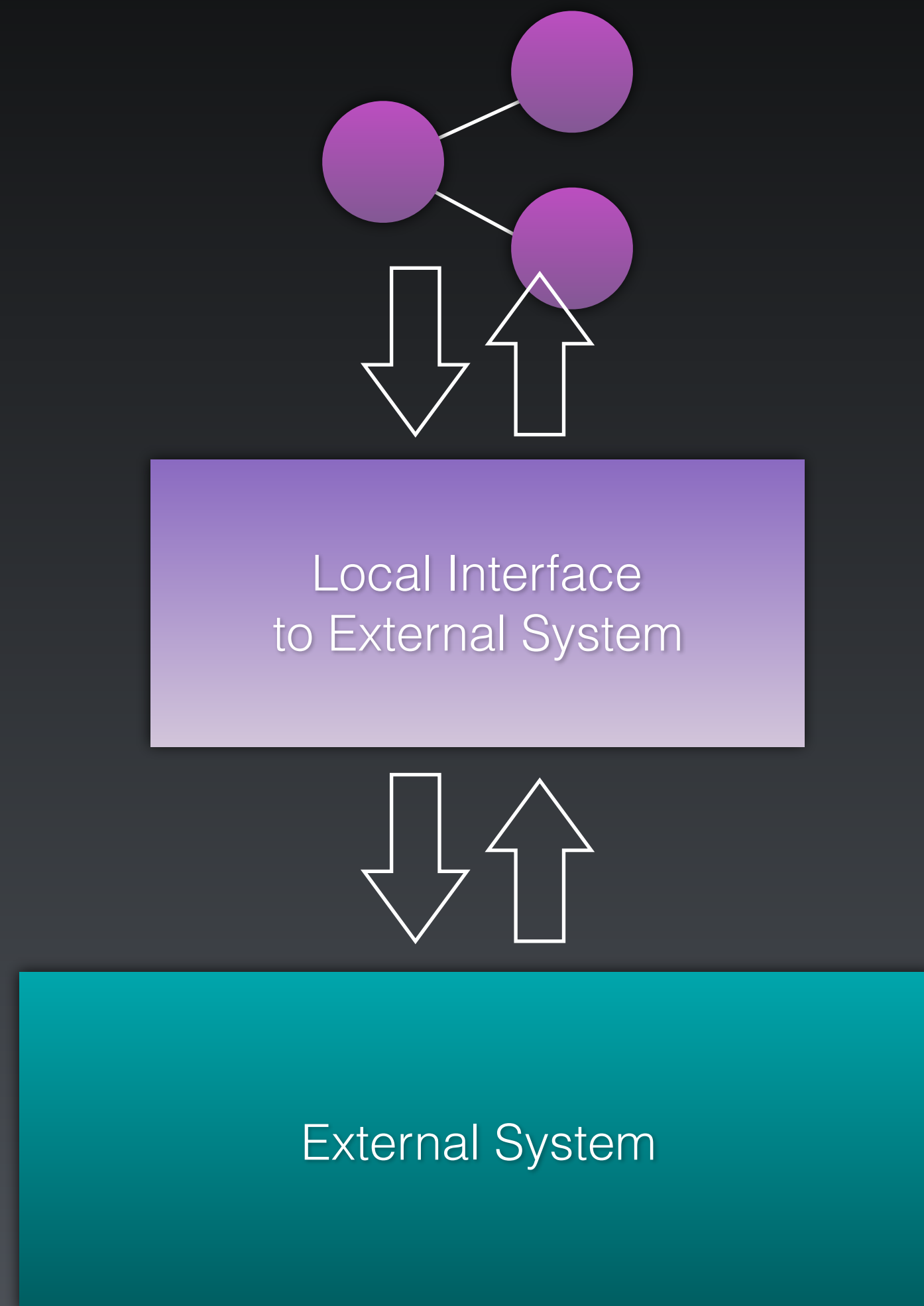
# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

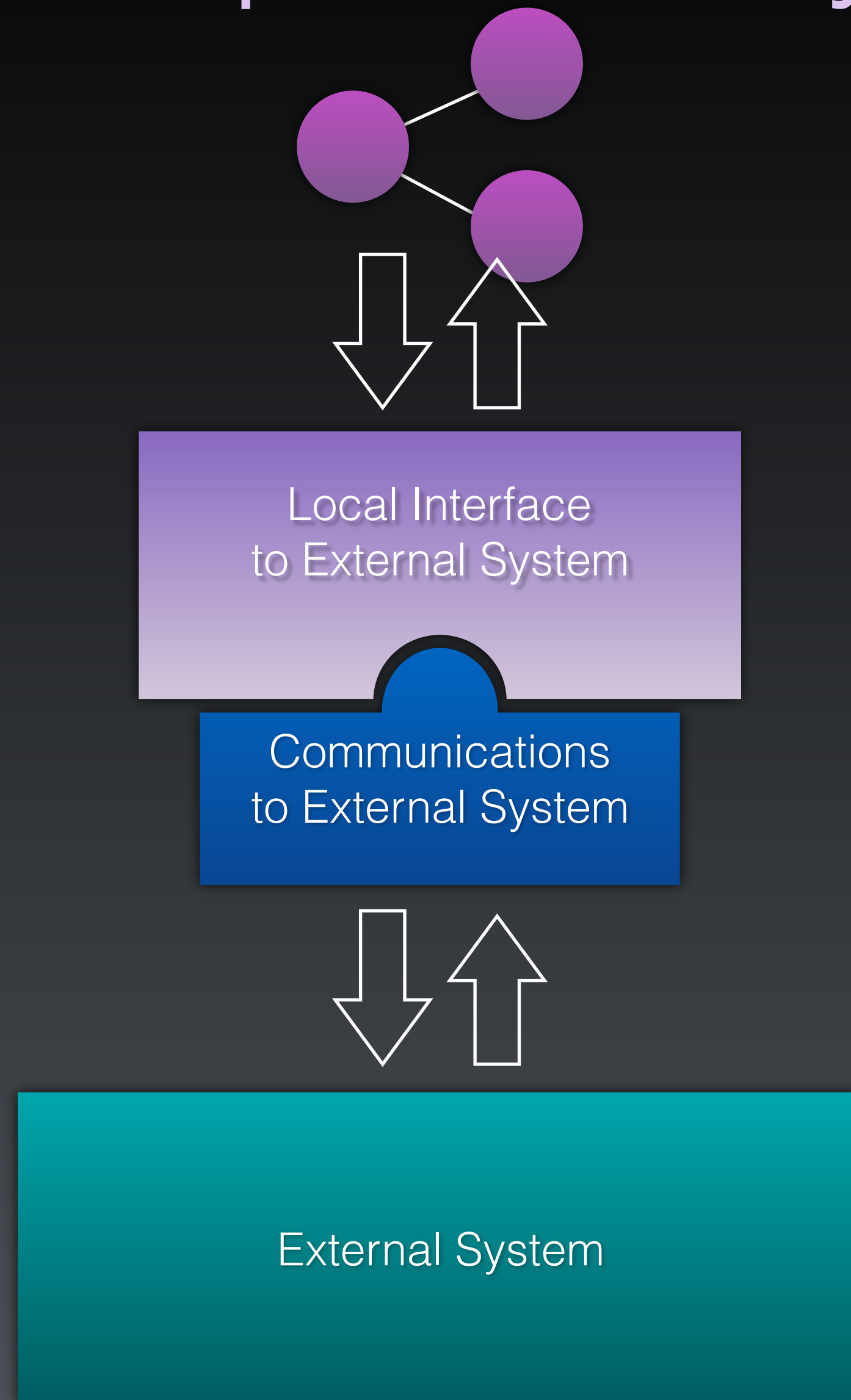
# Repeatability - Test Doubles



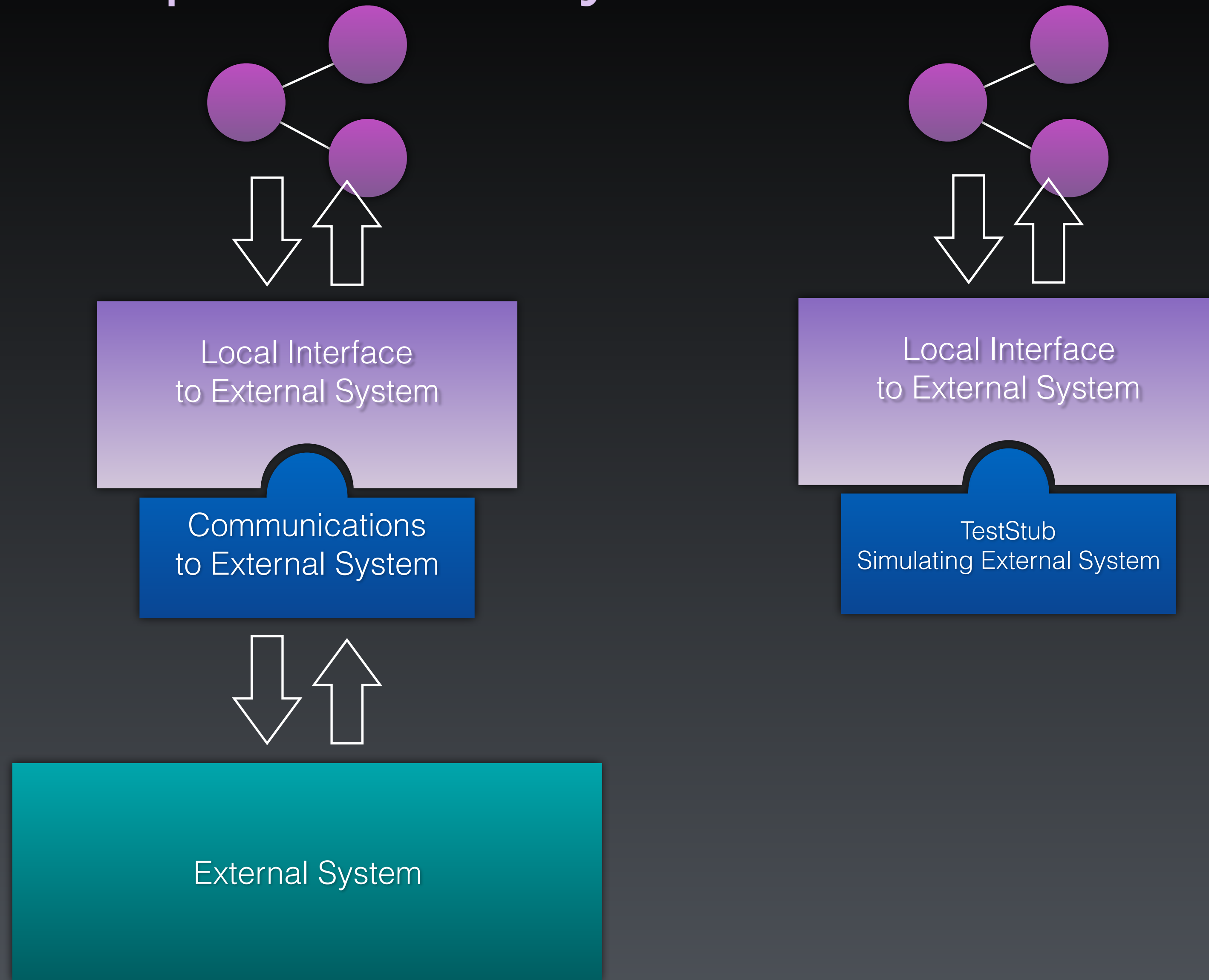
# Repeatability - Test Doubles



# Repeatability - Test Doubles

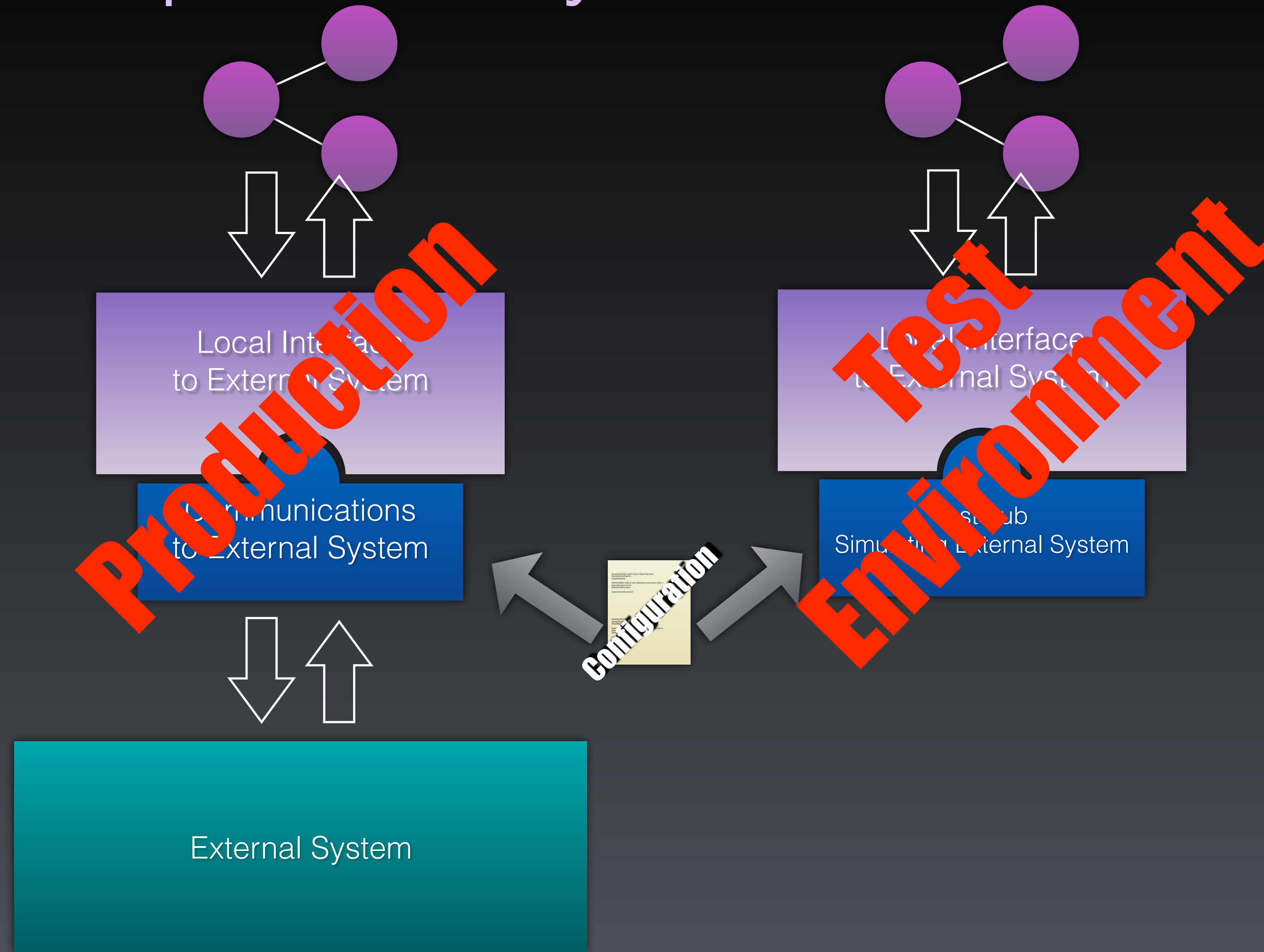


# Repeatability - Test Doubles

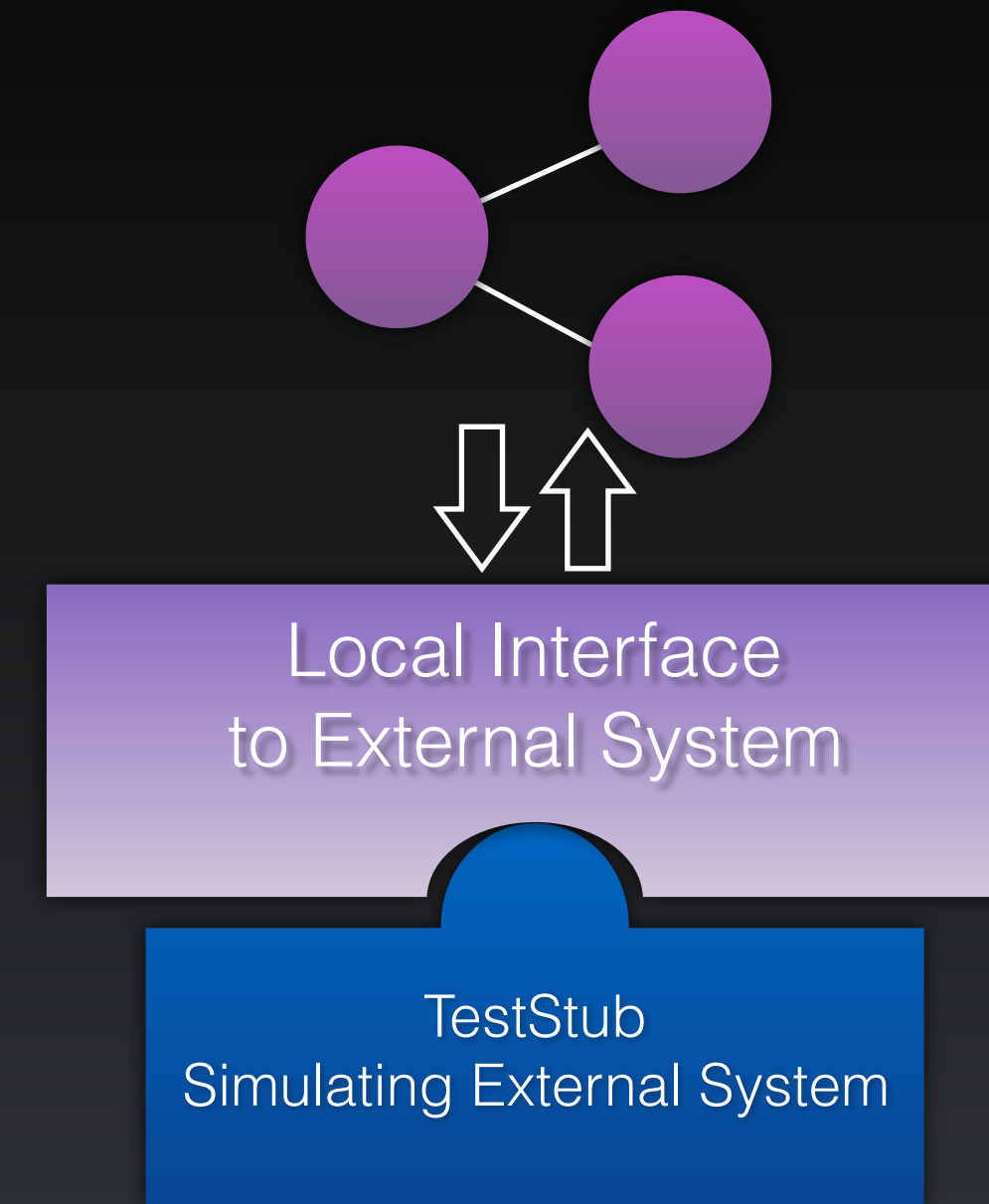




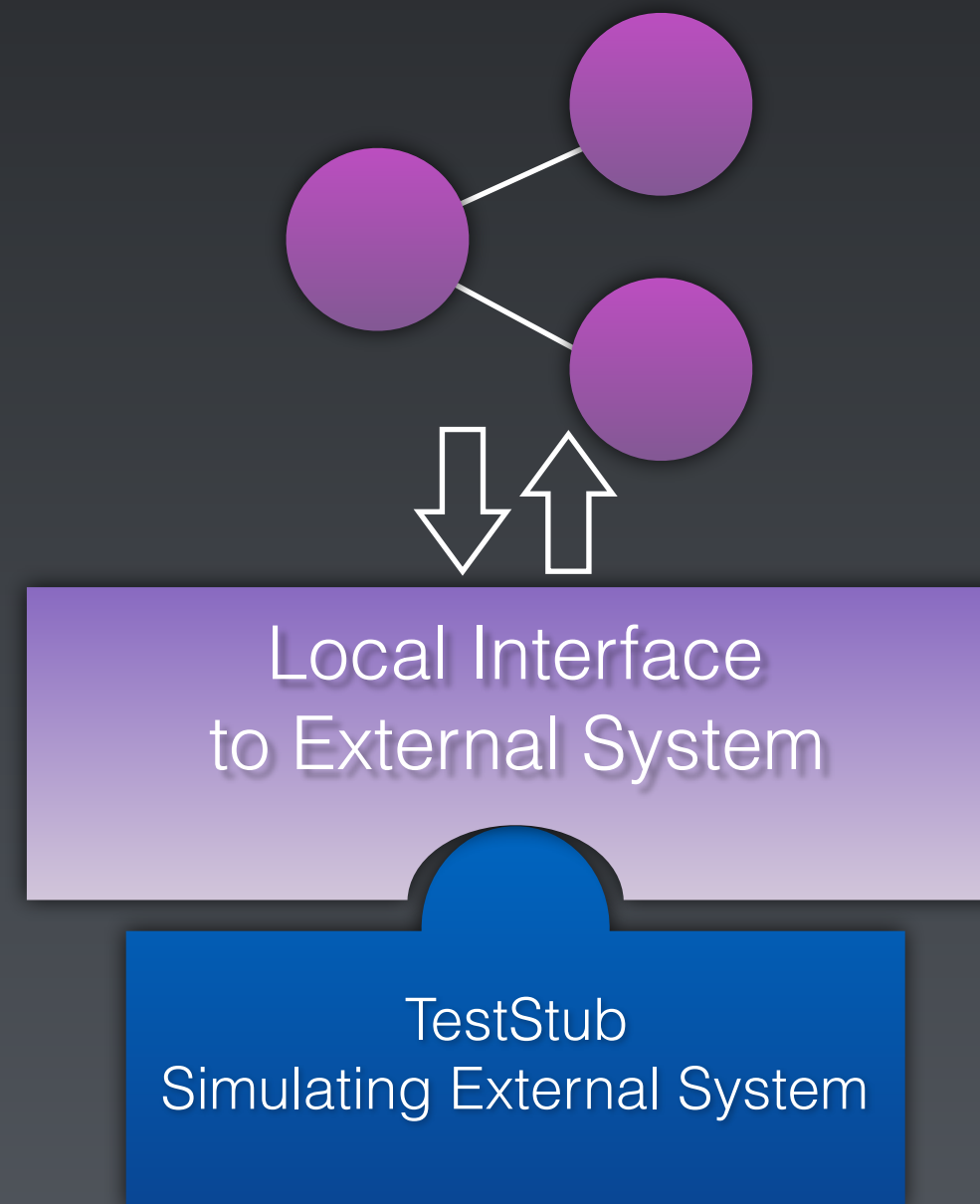
# Repeatability - Test Doubles



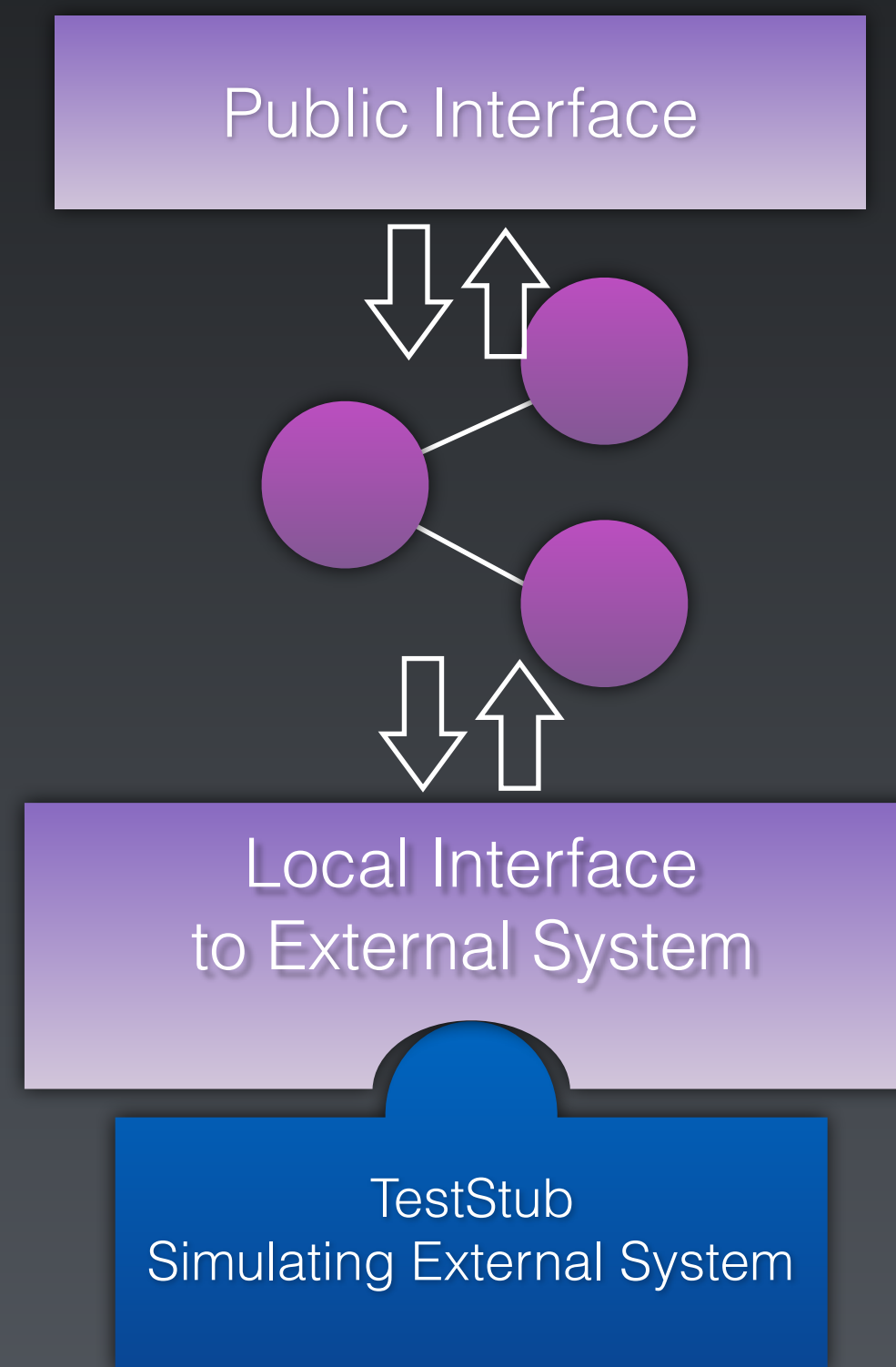
# Test Doubles As Part of Test Infrastructure



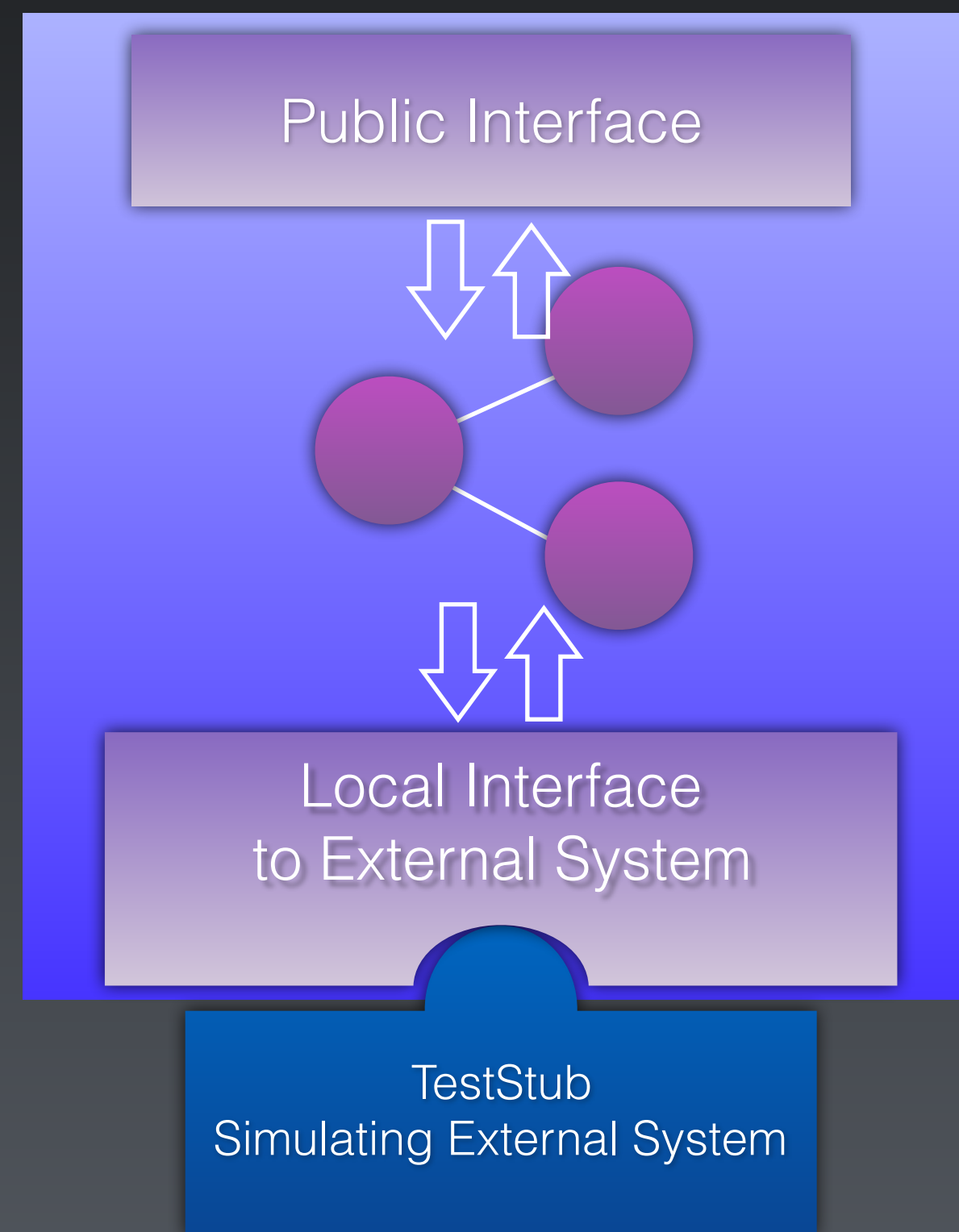
# Test Doubles As Part of Test Infrastructure



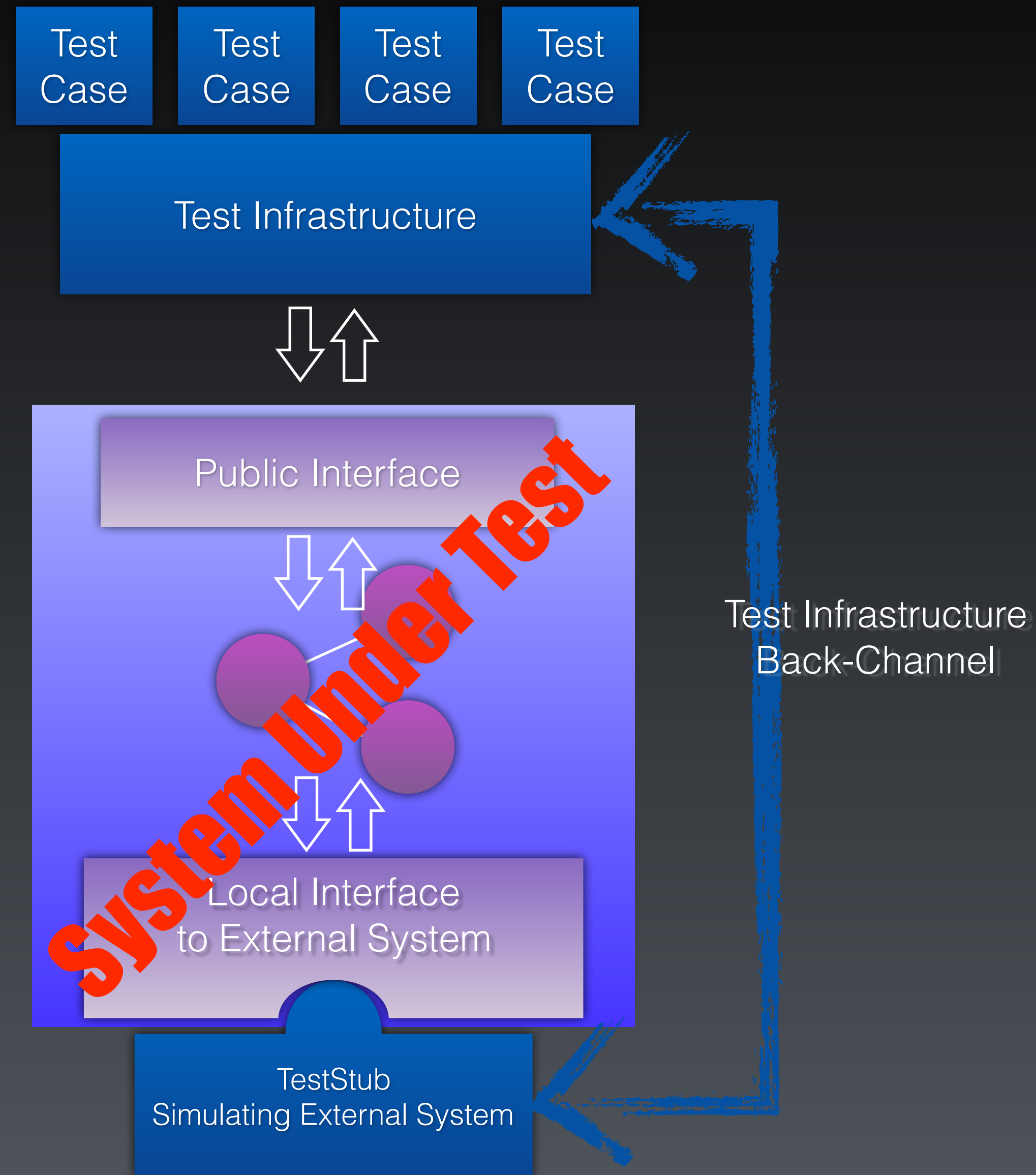
# Test Doubles As Part of Test Infrastructure



# Test Doubles As Part of Test Infrastructure



# Test Doubles As Part of Test Infrastructure





# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

# Language of the Problem Domain - DSL

- A Simple 'DSL' Solves many of our problems
  - Ease of TestCase creation
  - Readability
  - Ease of Maintenance
  - Separation of "What" from "How"
  - Test Isolation
  - The Chance to abstract complex set-up and scenarios
  - ...

# Language of the Problem Domain - DSL

```
@Test
public void shouldSupportPlacingValidBuyAndSellLimitOrders()
{
    trading.selectDealTicket("instrument");
    trading.dealTicket.placeOrder("type: limit", "bid: 4@10");
    trading.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to buy 4.00 contracts at 10.0");
    trading.dealTicket.dismissFeedbackMessage();

    trading.dealTicket.placeOrder("type: limit", "ask: 4@9");
    trading.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to sell 4.00 contracts at 9.0");
}
```

# Language of the Problem Domain - DSL

```
@Test
public void shouldSupportPlacingValidBuyAndSellLimitOrders()
{
    trading.selectDealTicket("instrument");
    trading.dealTicket.placeOrder("type: limit", "bid: 4@10");
    trading.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to buy 4.00 contracts at 10.0");
    trading.dealTicket.dismissFeedbackMessage();

    trading.dealTicket.placeOrder("type: limit", "ask: 4@9");
    trading.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to sell 4.00 contracts at 9.0");
}

@Test
public void shouldSuccessfullyPlaceAnImmediateOrCancelBuyMarketOrder()
{
    fixAPIMarketMaker.placeMassOrder("instrument", "ask: 11@52", "ask: 10@51", "ask: 10@50", "bid: 10@49");

    fixAPI.placeOrder("instrument", "side: buy", "quantity: 4", "goodUntil: Immediate", "allowUnmatched: true");
    fixAPI.waitForExecutionReport("executionType: Fill", "orderStatus: Filled",
        "side: buy", "quantity: 4", "matched: 4", "remaining: 0",
        "executionPrice: 50", "executionQuantity: 4");
}
```

# Language of the Problem Domain - DSL

```
@Test
public void shouldSupportPlacingValidBuyAndSellLimitOrders()
{
    trading.selectDealTicket("instrument");
    trading.dealTicket.placeOrder("type: limit", "bid: 4@10");
    trading.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to buy 4.00 contracts at 10.0");
    trading.dealTicket.dismissFeedbackMessage();

    trading.dealTicket.placeOrder("type: limit", "ask: 4@9");
    trading.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to sell 4.00 contracts at 9.0");
}

@Test
public void shouldSuccessfullyPlaceAnImmediateOrCancelBuyMarketOrder()
{
    fixAPIMarketMaker.placeMassOrder("instrument", "ask: 11@52", "ask: 10@51", "ask: 10@50", "bid: 10@49");

    fixAPI.placeOrder("instrument", "side: buy", "quantity: 4", "goodUntil: Immediate", "allowUnmatched: true");
    fixAPI.waitForExecutionReport("executionType: Fill", "orderStatus: Filled",
        "side: buy", "quantity: 4", "matched: 4", "remaining: 0",
        "executionPrice: 50", "executionQuantity: 4");
}

@Before
public void beforeEveryTest()
{
    adminAPI.createInstrument("name: instrument");
    registrationAPI.createUser("user");
    registrationAPI.createUser("marketMaker", "accountType: MARKET_MAKER");
    tradingUI.loginAsLive("user");
}
```



# Language of the Problem Domain - DSL

```
public void placeOrder(final String... args)
{
    final DslParams params =
        new DslParams(args,
            new OptionalParam("type").setDefault("Limit").setAllowedValues("limit", "market", "StopMarket"),
            new OptionalParam("side").setDefault("Buy").setAllowedValues("buy", "sell"),
            new OptionalParam("price"),
            new OptionalParam("triggerPrice"),
            new OptionalParam("quantity"),
            new OptionalParam("stopProfitOffset"),
            new OptionalParam("stopLossOffset"),
            new OptionalParam("confirmFeedback").setDefault("true"));

    getDealTicketPageDriver().placeOrder(params.value("type"),
        params.value("side"),
        params.value("price"),
        params.value("triggerPrice"),
        params.value("quantity"),
        params.value("stopProfitOffset"),
        params.value("stopLossOffset"));

    if (params.valueAsBoolean("confirmFeedback"))
    {
        getDealTicketPageDriver().clickOrderFeedbackConfirmationButton();
    }

    LOGGER.debug("placeOrder(" + Arrays.deepToString(args) + ")");
}
```

# Language of the Problem Domain - DSL

```
@Test
public void shouldSupportPlacingValidBuyAndSellLimitOrders()
{
    tradingUI.showDealTicket("instrument");
    tradingUI.dealTicket.placeOrder("type: limit", "bid: 4@10");
    tradingUI.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to buy 4.00 contracts at 10.0");
    tradingUI.dealTicket.dismissFeedbackMessage();

    tradingUI.dealTicket.placeOrder("type: limit", "ask: 4@9");
    tradingUI.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to sell 4.00 contracts at 9.0");
}

@Test
public void shouldSuccessfullyPlaceAnImmediateOrCancelBuyMarketOrder()
{
    fixAPIMarketMaker.placeMassOrder("instrument", "ask: 11@52", "ask: 10@51", "ask: 10@50", "bid: 10@49");

    fixAPI.placeOrder("instrument", "side: buy", "quantity: 4", "goodUntil: Immediate", "allowUnmatched: true");
    fixAPI.waitForExecutionReport("executionType: Fill", "orderStatus: Filled",
        "side: buy", "quantity: 4", "matched: 4", "remaining: 0",
        "executionPrice: 50", "executionQuantity: 4");
}
```

# Language of the Problem Domain - DSL

```
@Test
public void shouldSupportPlacingValidBuyAndSellLimitOrders()
{
    tradingUI.showDealTicket("instrument");
    tradingUI.dealTicket.placeOrder("type: limit", "bid: 4@10");
    tradingUI.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to buy 4.00 contracts at 10.0");
    tradingUI.dealTicket.dismissFeedbackMessage();

    tradingUI.dealTicket.placeOrder("type: limit", "ask: 4@9");
    tradingUI.dealTicket.checkFeedbackMessage("You have successfully sent a limit order to sell 4.00 contracts at 9.0");
}
```

```
@Test
public void shouldSuccessfullyPlaceAnImmediateOrCancelBuyMarketOrder()
{
    fixAPIMarketMaker.placeMassOrder("instrument", "ask: 11@52", "ask: 10@51", "ask: 10@50", "bid: 10@49");

    fixAPI.placeOrder("instrument", "side: buy", "quantity: 4", "goodUntil: Immediate", "allowUnmatched: true");
    fixAPI.waitForExecutionReport("executionType: Fill", "orderStatus: Filled",
        "side: buy", "quantity: 4", "matched: 4", "remaining: 0",
        "executionPrice: 50", "executionQuantity: 4");
}
```

# Language of the Problem Domain - DSL

```
@Channel(fixApi, dealTicket, publicApi)
@Test
public void shouldSuccessfullyPlaceAnImmediateOrCancelBuyMarketOrder()
{
    trading.placeOrder("instrument", "side: buy", "price: 123.45", "quantity: 4", "goodUntil: Immediate");

    trading.waitForExecutionReport("executionType: Fill", "orderStatus: Filled",
                                   "side: buy", "quantity: 4", "matched: 4", "remaining: 0",
                                   "executionPrice: 123.45", "executionQuantity: 4");
}
```

# Language of the Problem Domain - DSL

```
@Channel(fixApi, dealTicket, publicApi)
@Test
public void shouldSuccessfullyPlaceAnImmediateOrCancelBuyMarketOrder()
{
    trading.placeOrder("instrument", "side: buy", "price: 123.45", "quantity: 4", "goodUntil: Immediate");
    trading.waitForExecutionReport("executionType: Fill", "orderStatus: Filled",
                                   "side: buy", "quantity: 4", "matched: 4", "remaining: 0",
                                   "executionPrice: 123.45", "executionQuantity: 4");
}
```

# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient



# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

# Testing with Time

- Test Cases should be deterministic
- Time is a problem for determinism - There are two options:
  - Ignore time
  - Control time

# Testing With Time - Ignore Time

## ***Mechanism***

Filter out time-based values in your test infrastructure so that they are ignored

## ***Pros:***

- Simple!

## ***Cons:***

- Can miss errors
- Prevents any hope of testing complex time-based scenarios

# Testing With Time - Controlling Time

## ***Mechanism***

Treat Time as an external dependency, like any external system - and Fake it!

## ***Pros:***

- Very Flexible!
- Can simulate any time-based scenario, with time under the control of the test case.

## ***Cons:***

- Slightly more complex infrastructure

# Testing With Time - Controlling Time

```
@Test
public void shouldBeOverdueAfterOneMonth()
{
    book = library.borrowBook("Continuous Delivery");
    assertFalse(book.isOverdue());

    time.travel("+1 week");
    assertFalse(book.isOverdue());

    time.travel("+4 weeks");
    assertTrue(book.isOverdue());
}
```

# Testing With Time - Controlling Time

```
@Test
public void shouldBeOverdueAfterOneMonth()
{
    book = library.borrowBook("Continuous Delivery");
    assertFalse(book.isOverdue());

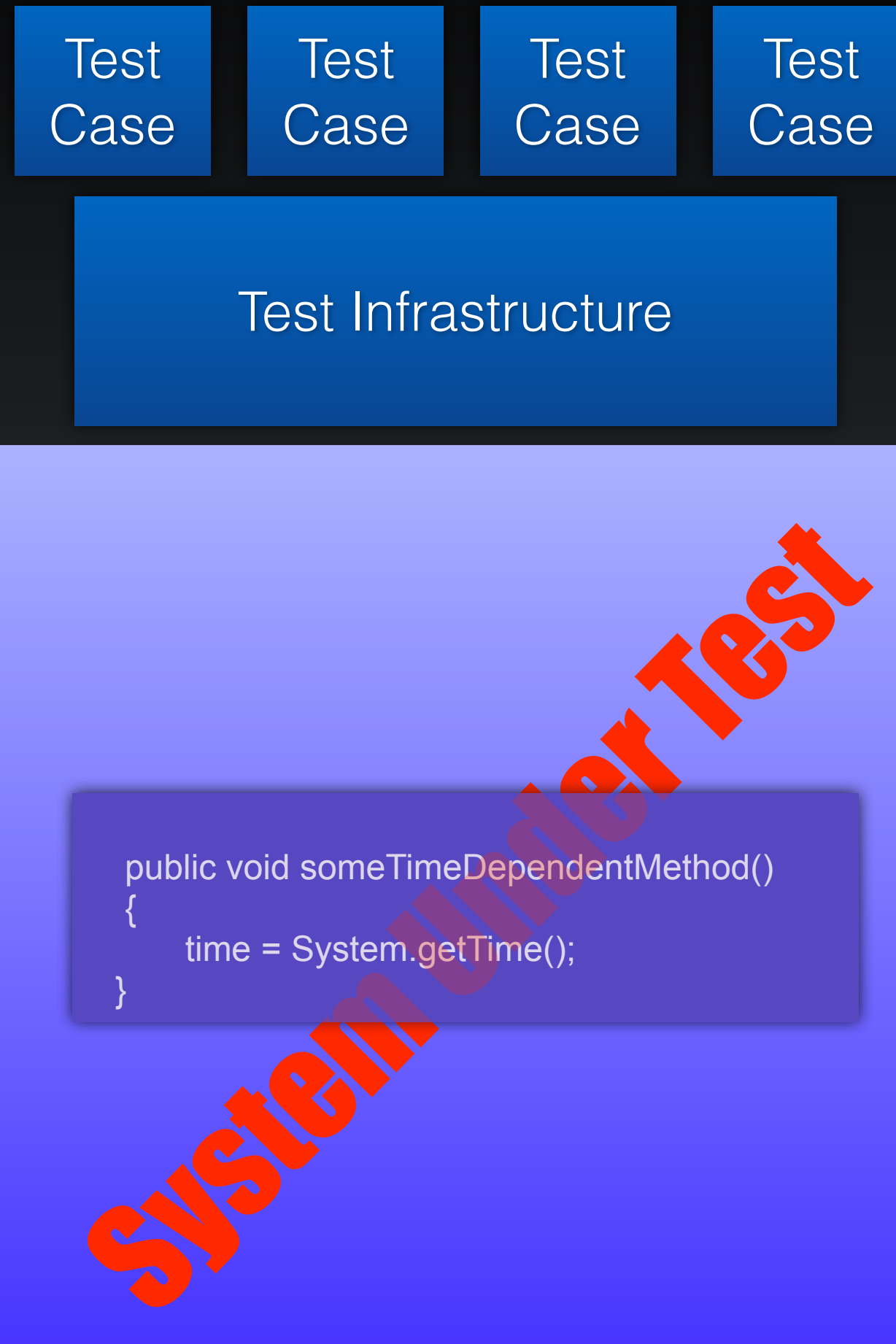
    time.travel("+1 week");
    assertFalse(book.isOverdue());

    time.travel("+4 weeks");
    assertTrue(book.isOverdue());
}
```

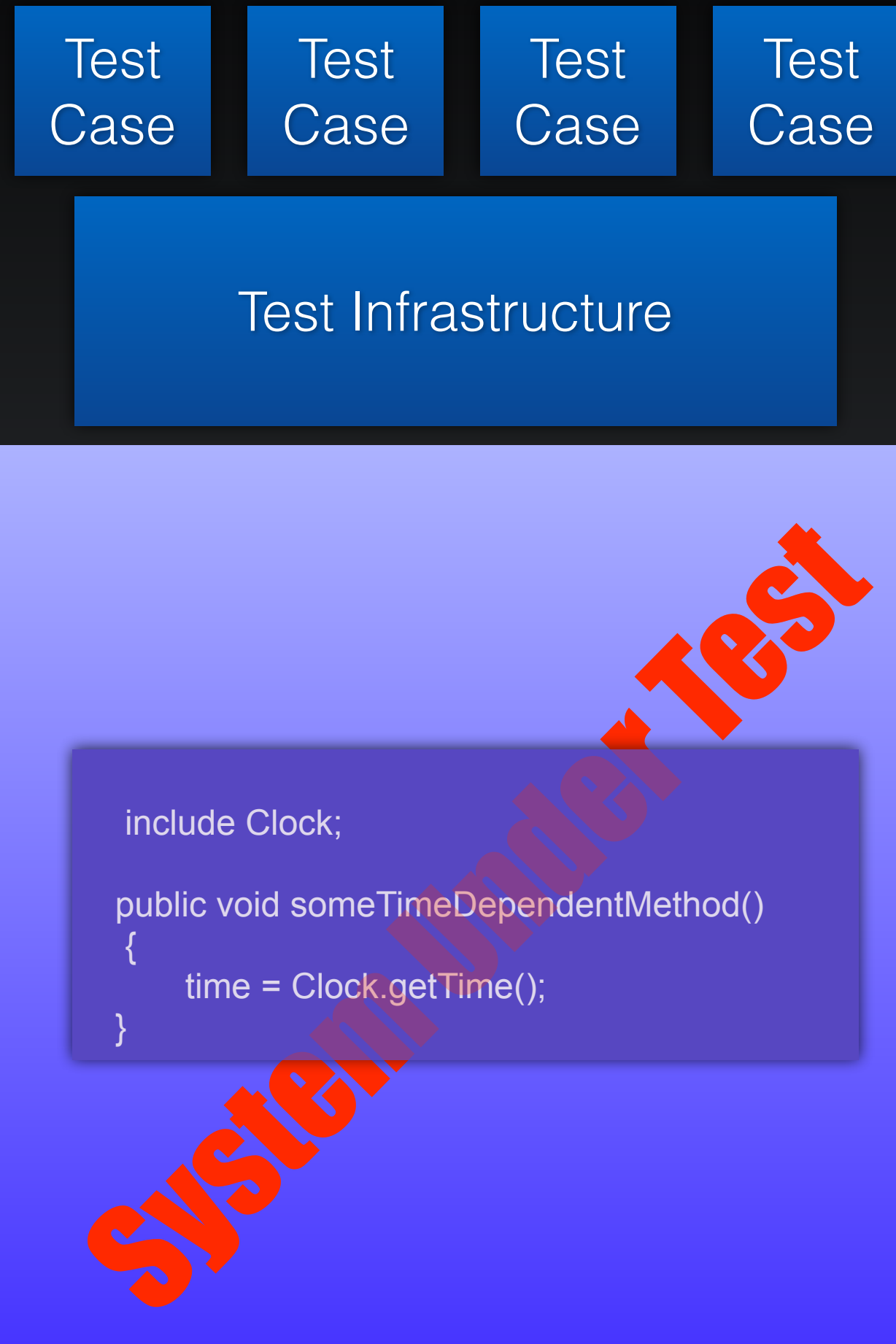


# Testing With Time - Controlling Time

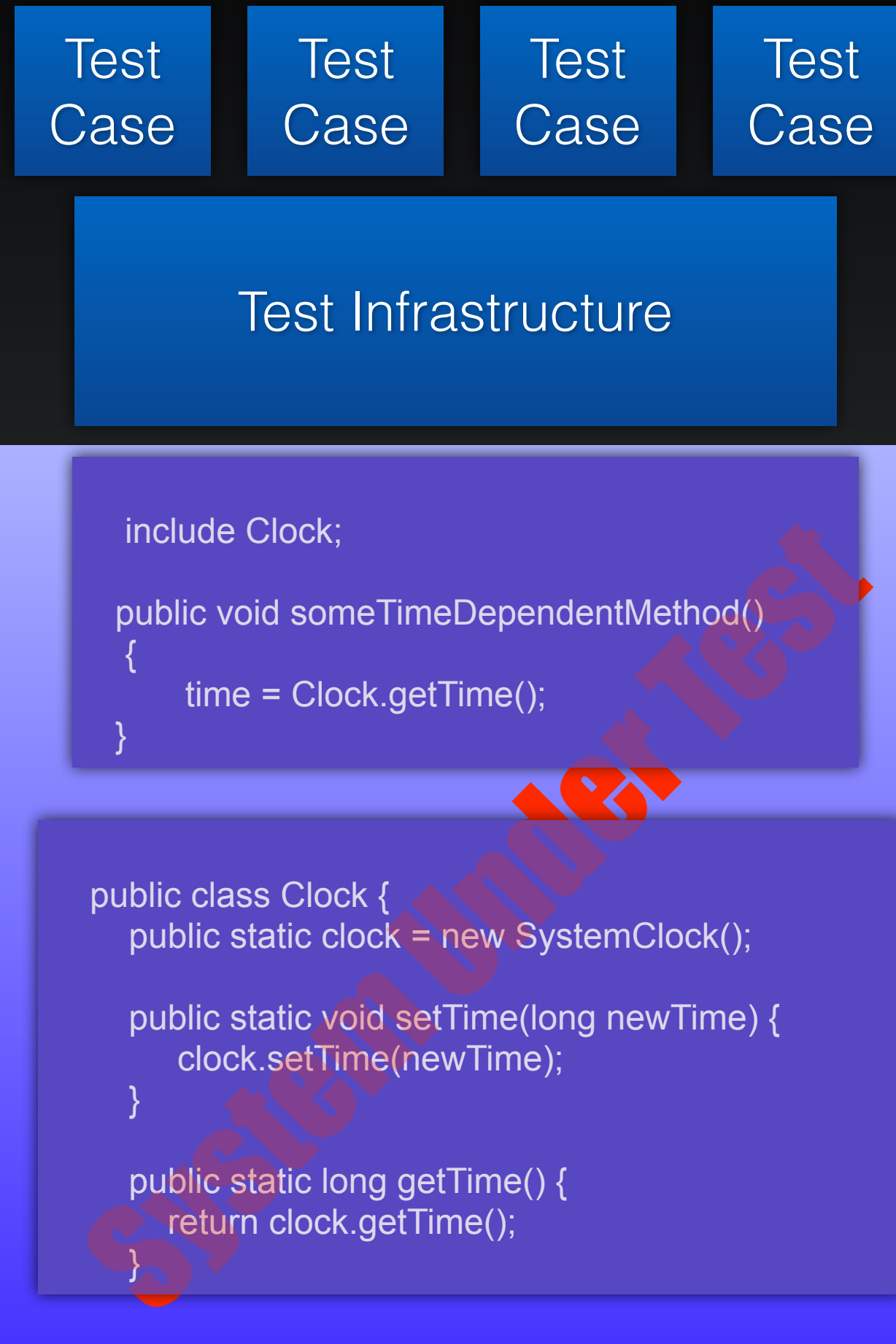
# Testing With Time - Controlling Time



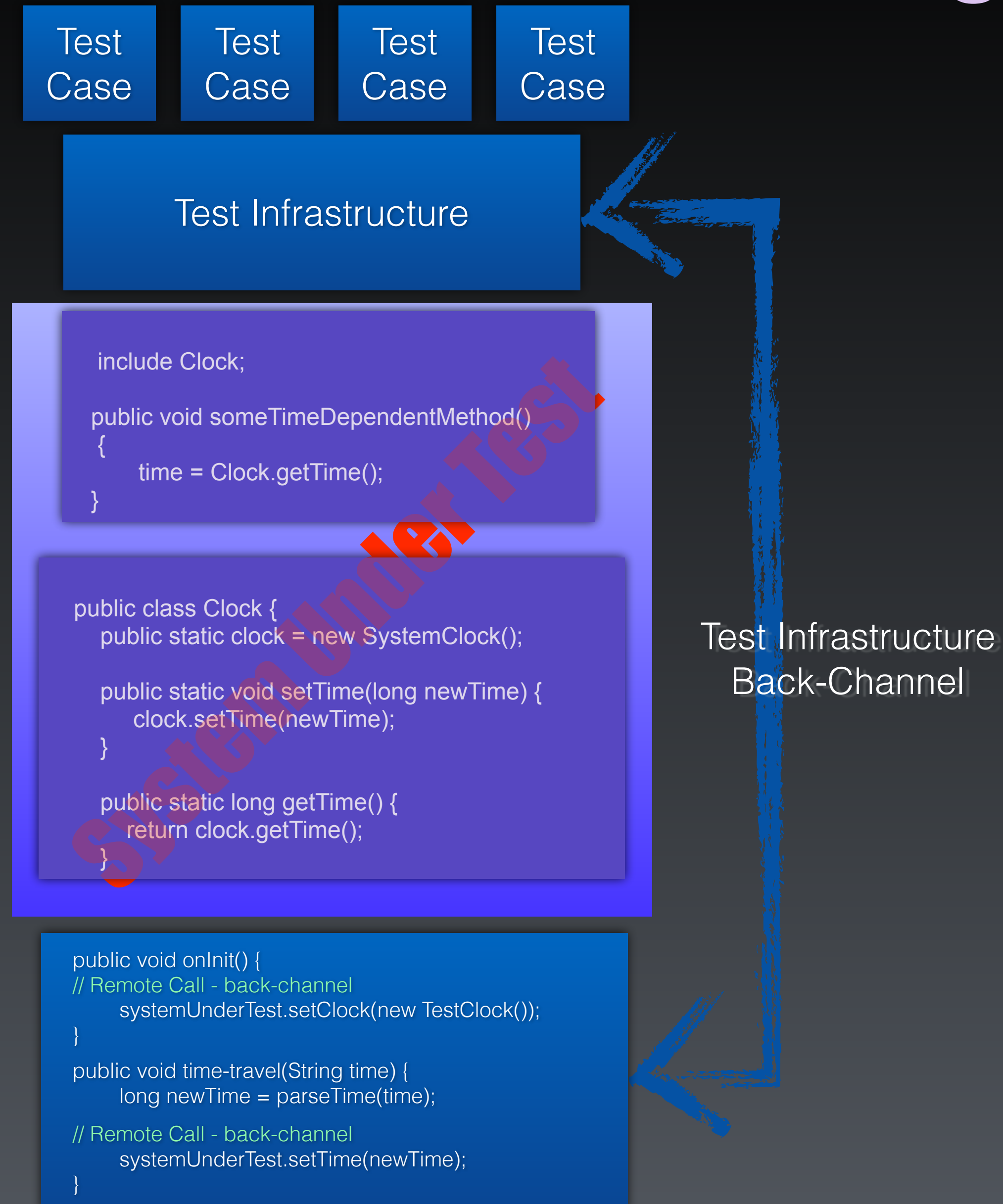
# Testing With Time - Controlling Time



# Testing With Time - Controlling Time



# Testing With Time - Controlling Time



# Test Environment Types

- Some Tests need special treatment.
- Tag Tests with properties and allocate them dynamically:

# Test Environment Types

- Some Tests need special treatment.
- Tag Tests with properties and allocate them dynamically:

```
@TimeTravel
@Test
public void shouldDoSomethingThatNeedsFakeTime ()
...
```

```
@Destructive
@Test
public void shouldDoSomethingThatKillsPartOfTheSystem()
...
```

```
@FPGA(version=1.3)
@Test
public void shouldDoSomethingThatRequiresSpecificHardware()
...
```



# Test Environment Types

- Some Tests need special treatment.
- Tag Tests with properties and allocate them dynamically:

```
@TimeTravel
```

```
@Test
```

```
public void shouldDoSomethingThatNeedsFakeTime ()
```

```
...
```

```
@Destructive
```

```
@Test
```

```
public void shouldDoSomethingThatKillsPartOfTheSystem()
```

```
...
```

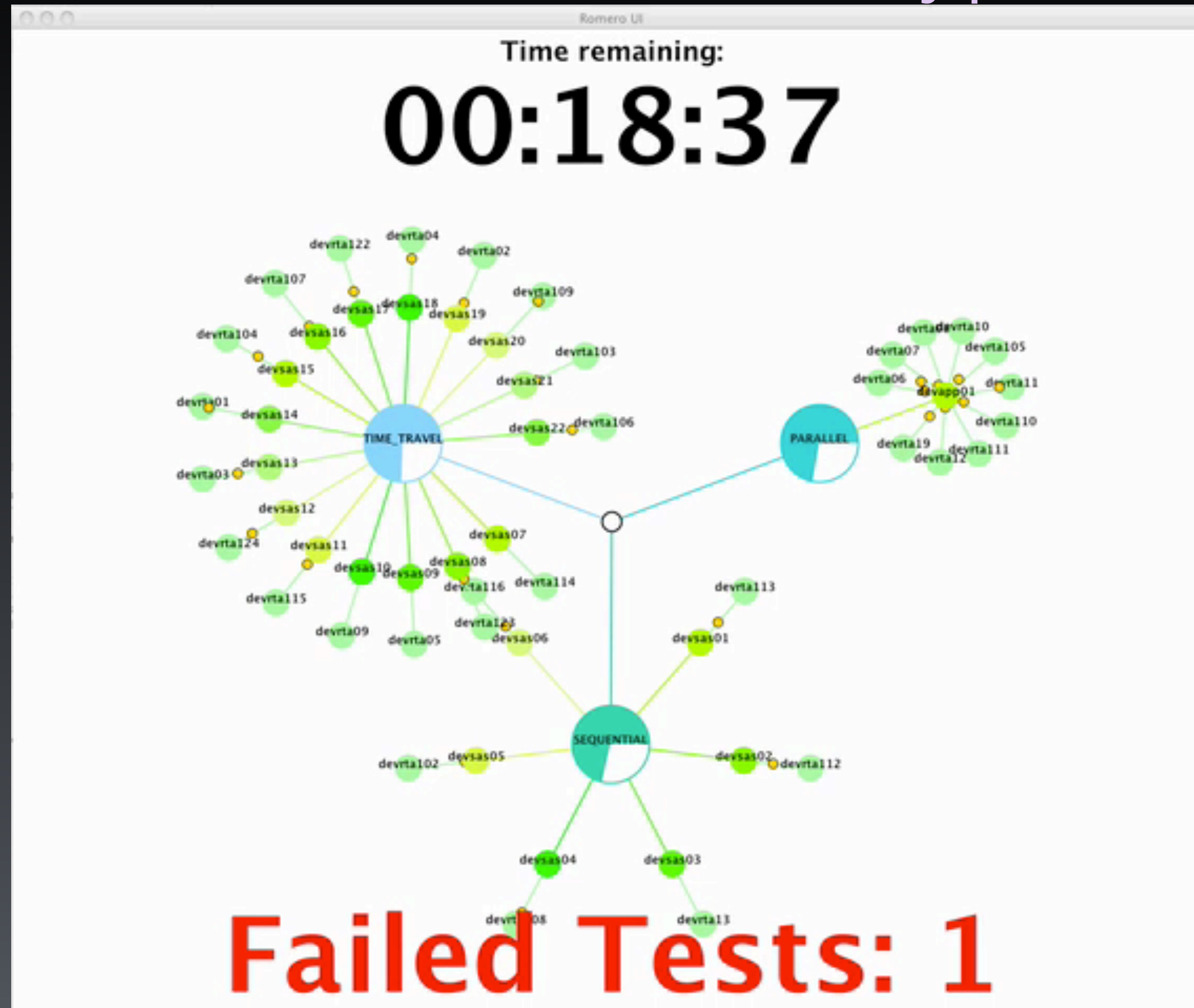
```
@FPGA(version=1.3)
```

```
@Test
```

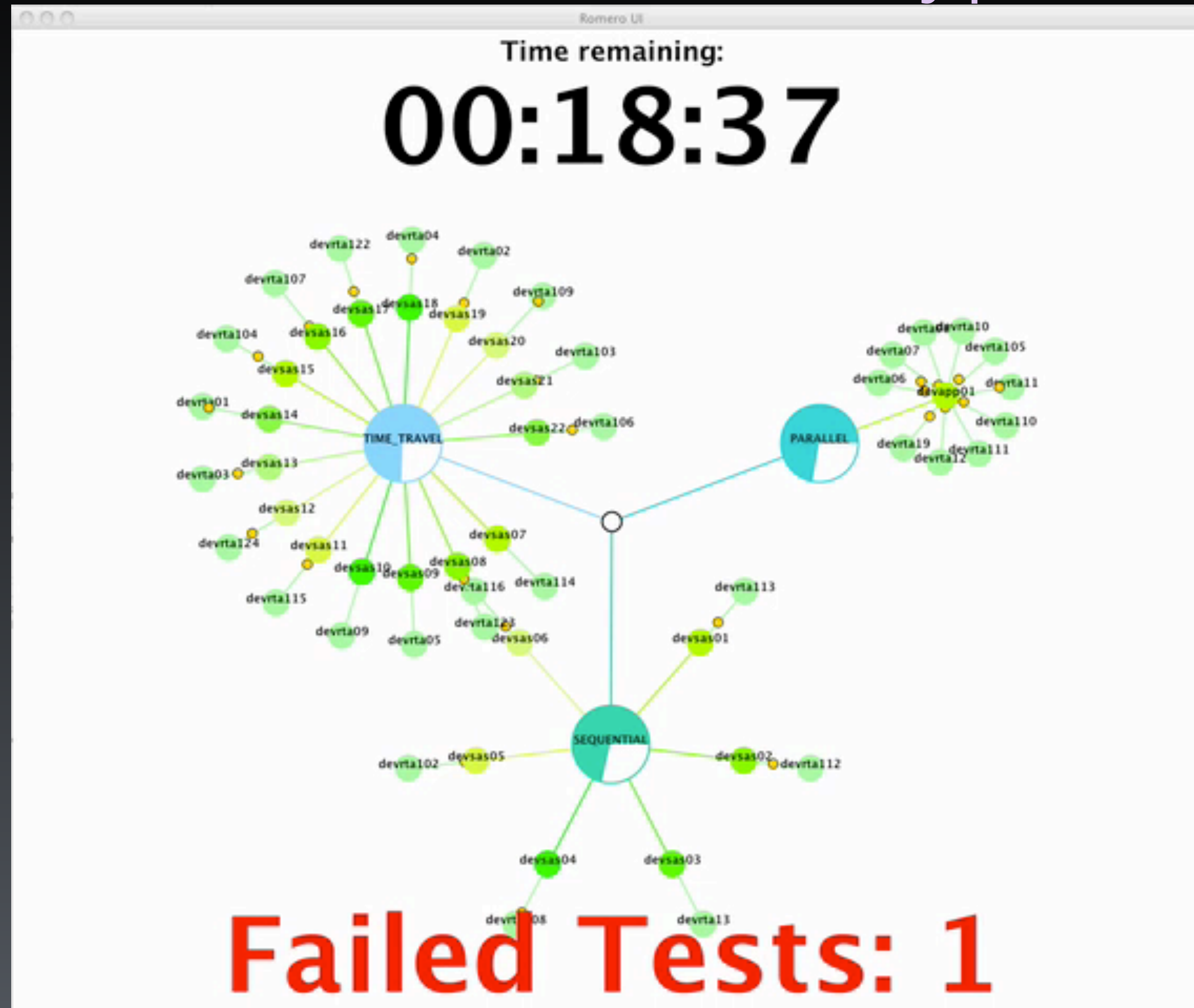
```
public void shouldDoSomethingThatRequiresSpecificHardware()
```

```
...
```

# Test Environment Types



# Test Environment Types



# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient



# Properties of Good Acceptance Tests

- “What” not “How”
- Isolated from other tests
- Repeatable
- Uses the language of the problem domain
- Tests ANY change
- Efficient

# Production-like Test Environments





# Production-like Test Environments



# Production-like Test Environments



# Production-like Test Environments





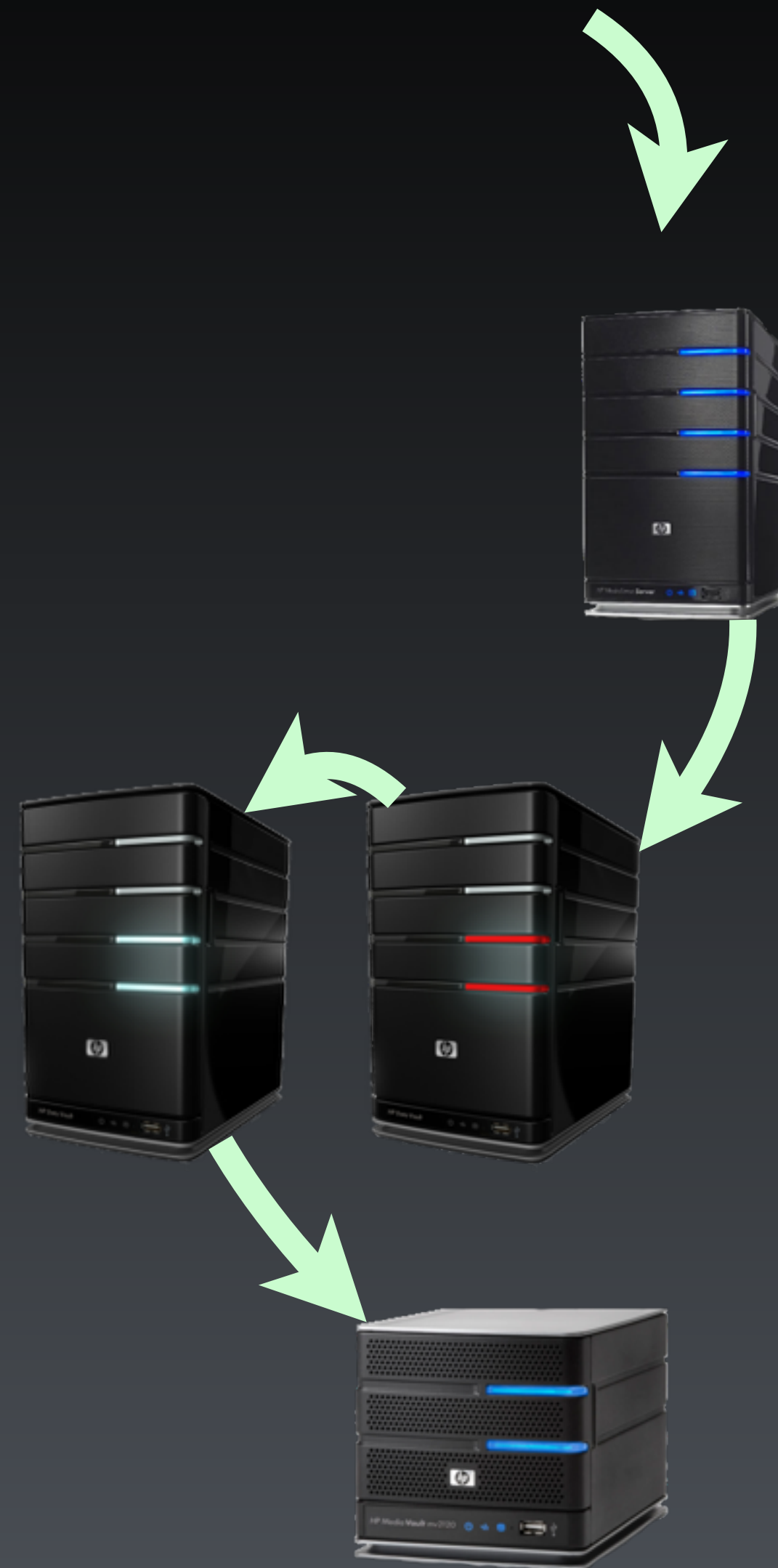
# Production-like Test Environments



# Production-like Test Environments



# Production-like Test Environments



# Make Test Cases Internally Synchronous



# Make Test Cases Internally Synchronous

- Look for a “Concluding Event” listen for that in your DSL to report an async call as complete

# Make Test Cases Internally Synchronous

- Look for a “Concluding Event” listen for that in your DSL to report an async call as complete

## *Example DSL level Implementation...*

```
public String placeOrder(String params...)
{
    orderSent = sendAsyncPlaceOrderMessage (parseOrderParams (params)) ;
    return waitForOrderConfirmedOrFailOnTimeout (orderSent) ;
}
```

# Make Test Cases Internally Synchronous

- Look for a “Concluding Event” listen for that in your DSL to report an async call as complete

## *Example DSL level Implementation...*

```
public String placeOrder(String params...)
{
    orderSent = sendAsyncPlaceOrderMessage(parseOrderParams(params));
    return waitForOrderConfirmedOrFailOnTimeout(orderSent);
}
```

# Make Test Cases Internally Synchronous

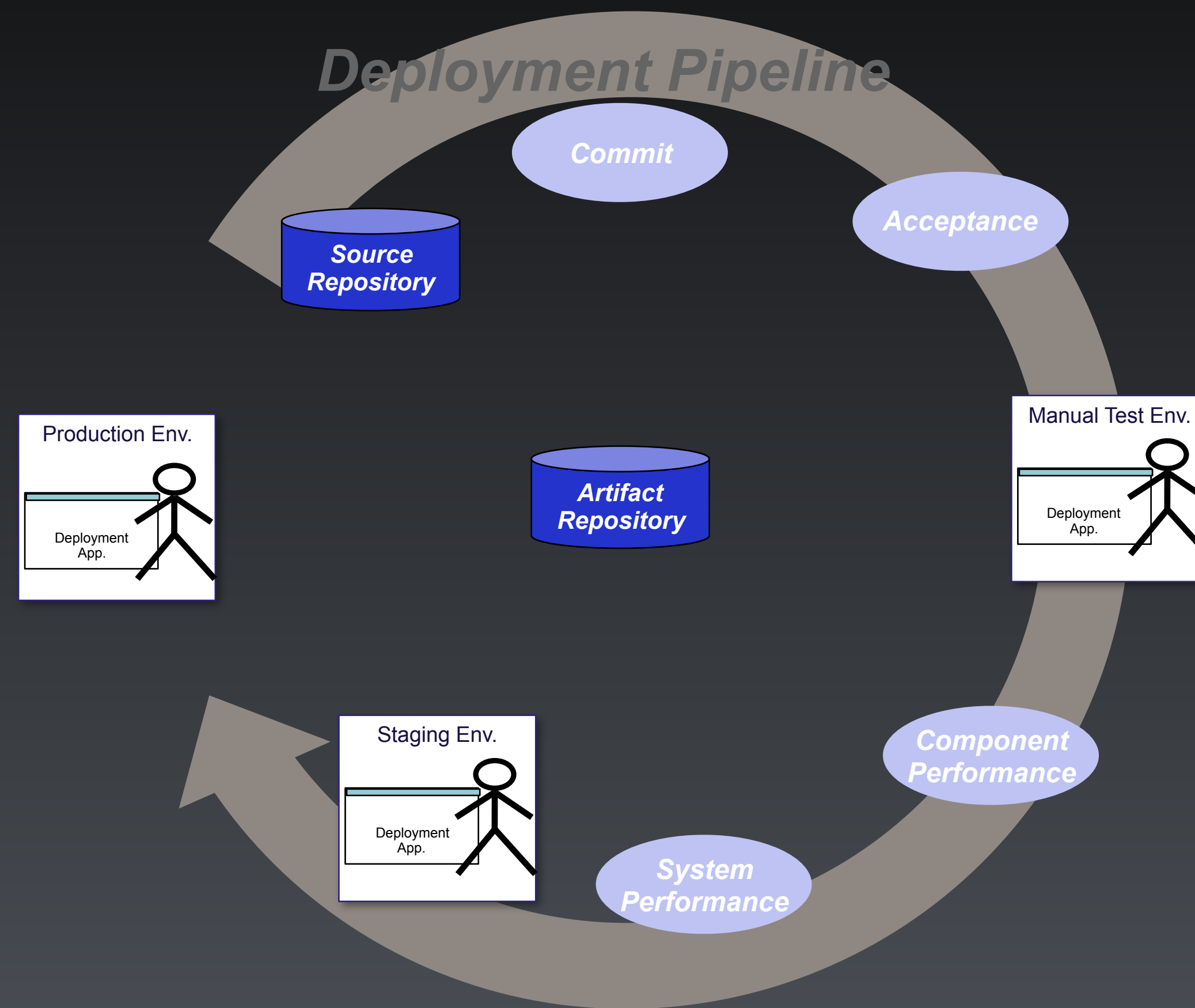
- Look for a “Concluding Event” listen for that in your DSL to report an async call as complete
- If you really have to, implement a “poll-and-timeout” mechanism in your test-infrastructure
- Never, Never, Never, put a “wait(xx)” and expect your tests to be (a) Reliable or (b) Efficient!

# Make Test Cases Internally Synchronous

- Look for a “Concluding Event” listen for that in your DSL to report an async call as complete
- If you really have to, implement a “poll-and-timeout” mechanism in your test-infrastructure
- Never, Never, Never run a “wait(x)” and expect your tests to be (a) Reliable or (b) Efficient!

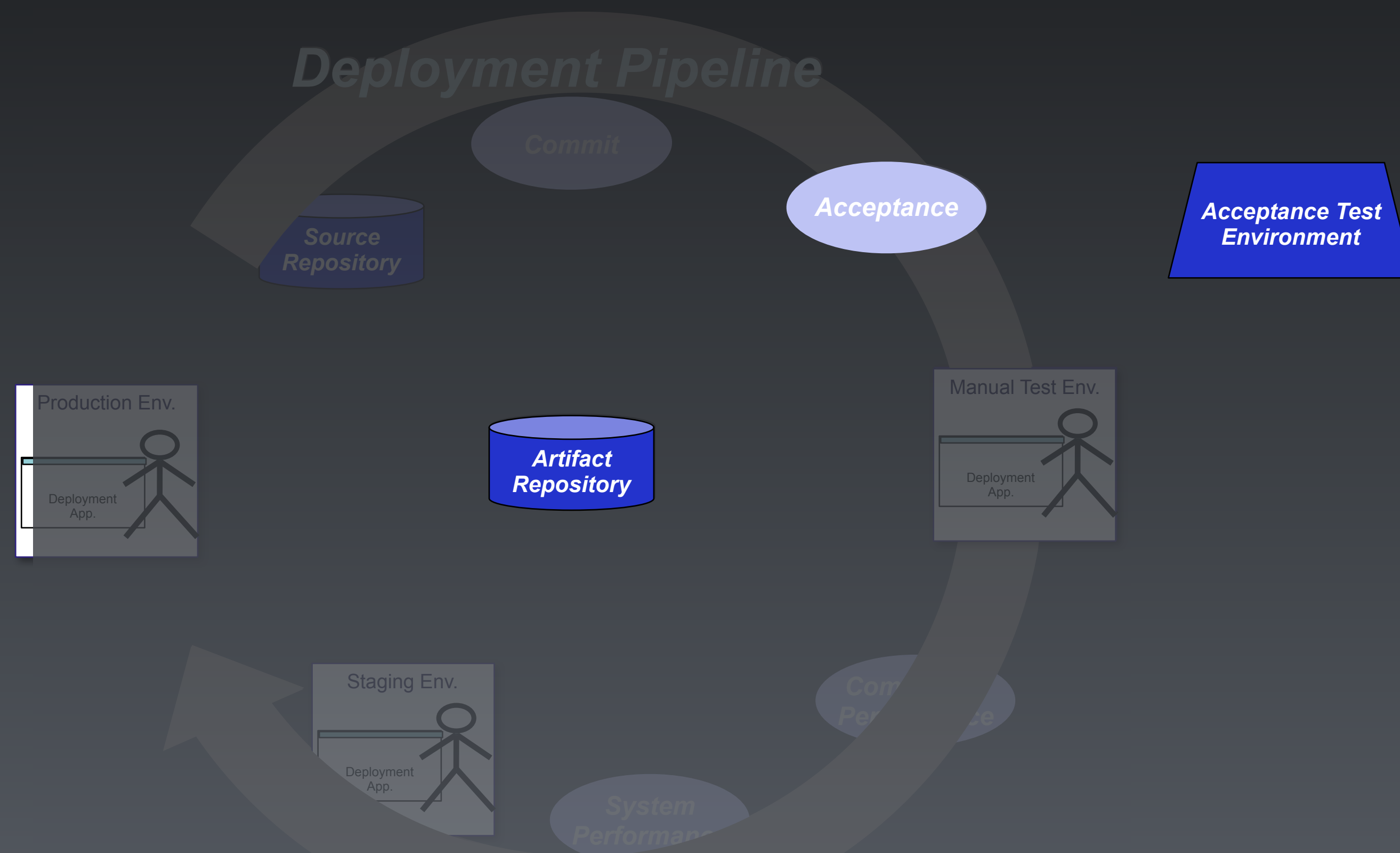
**Anti-Pattern!**

# Scaling-Up

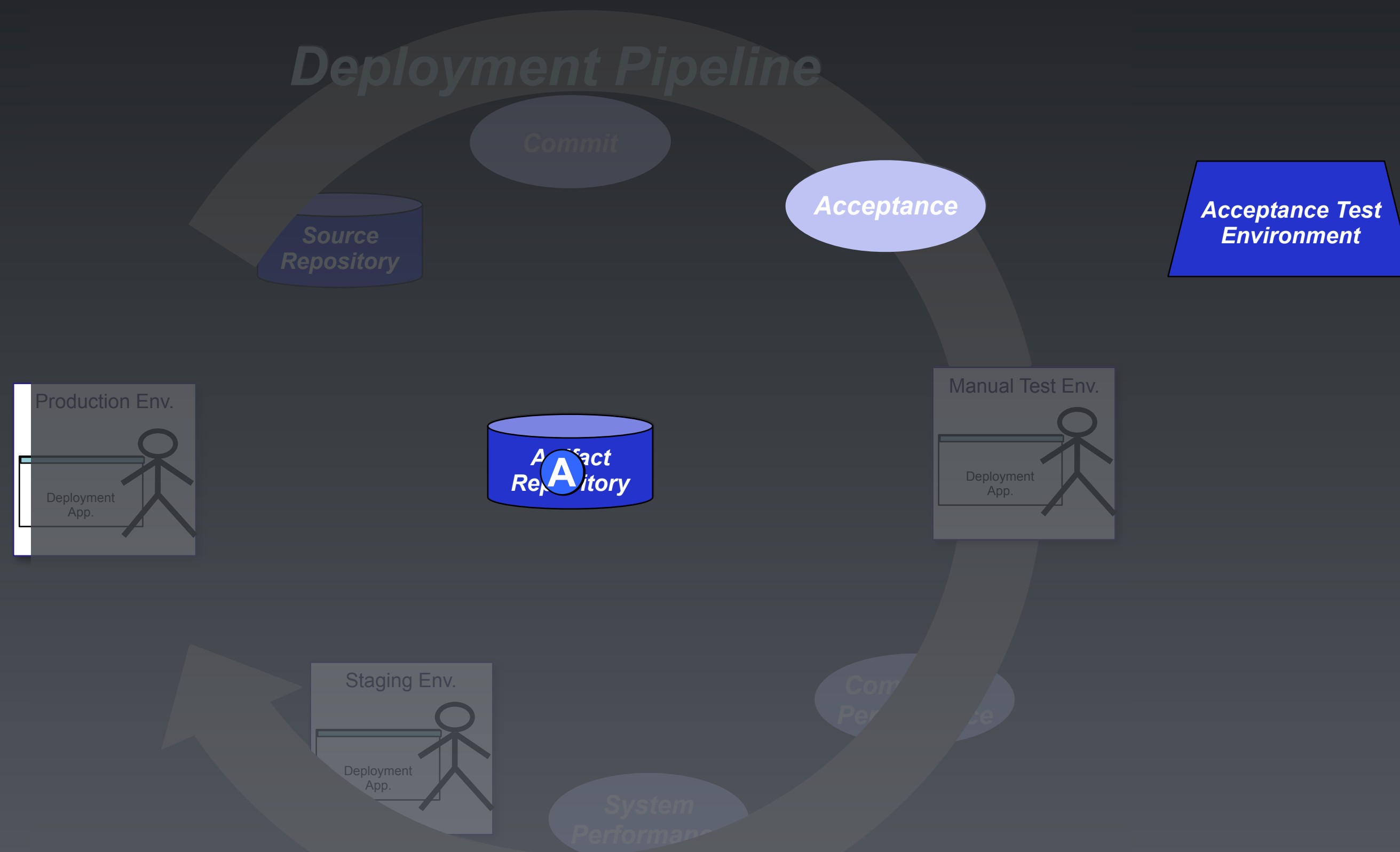




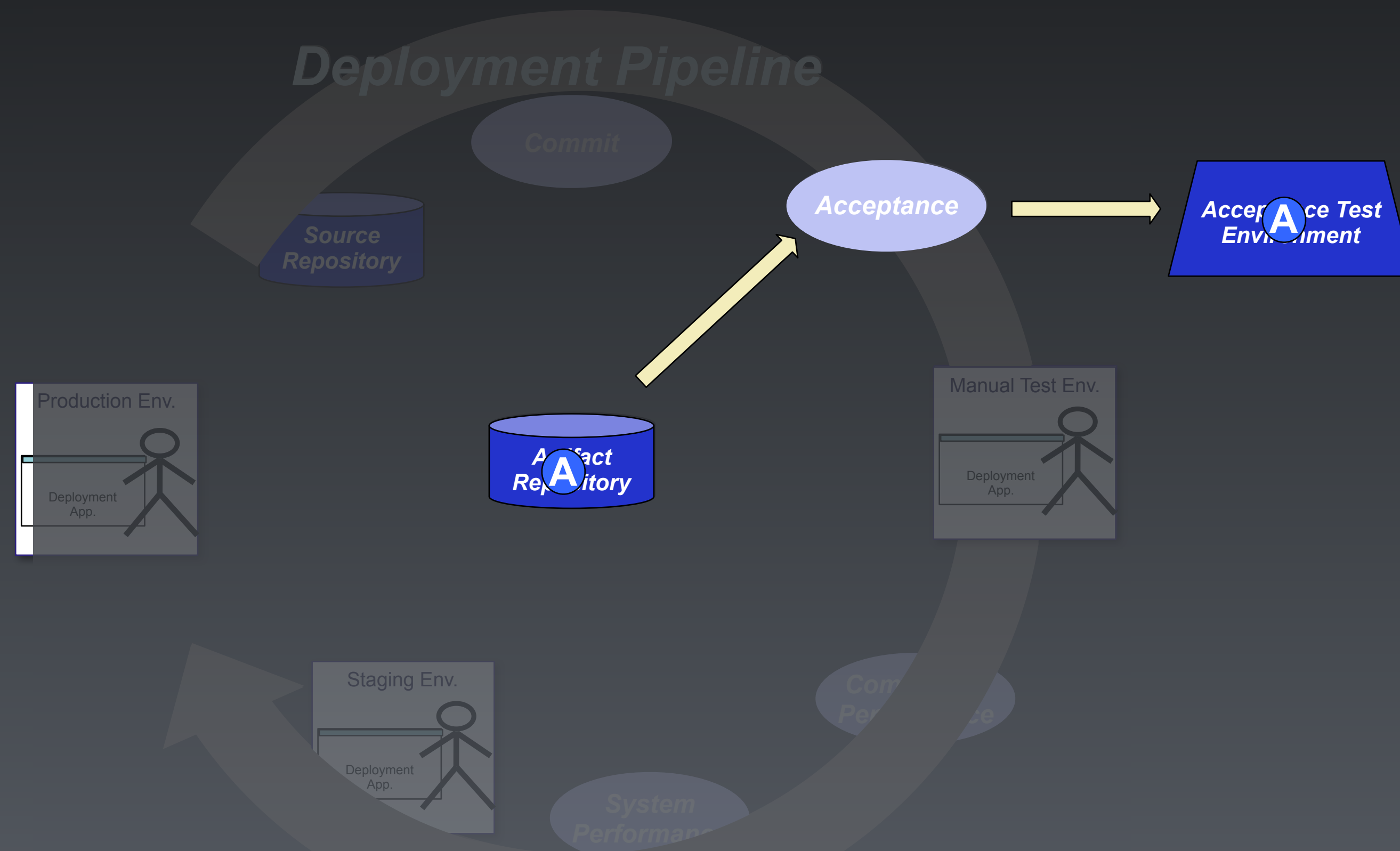
# Scaling-Up



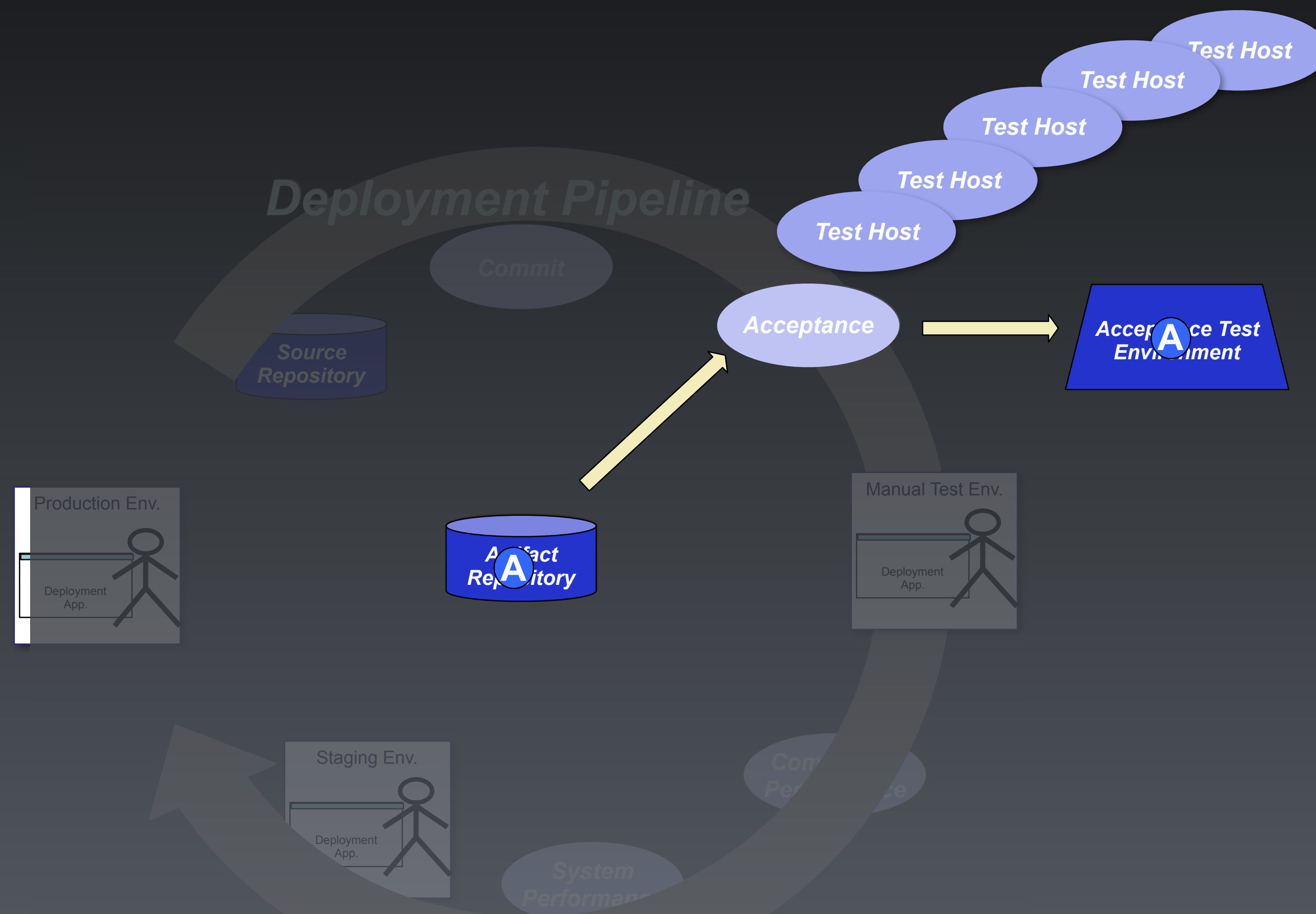
# Scaling-Up



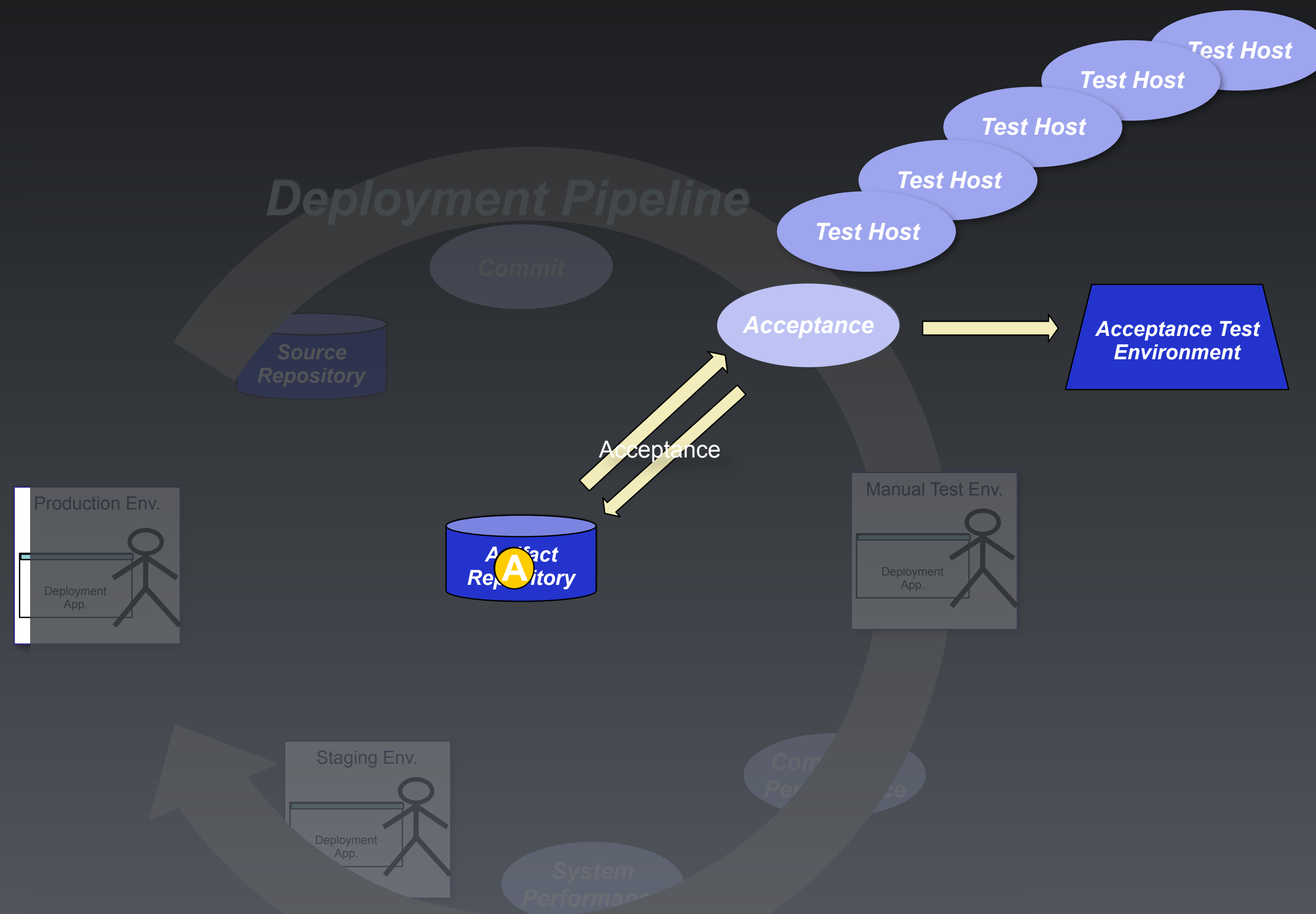
# Scaling-Up



# Scaling-Up



# Scaling-Up



# Anti-Patterns in Acceptance Testing



# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems

# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems
- **Don't** Record-and-playback production data. This has a role, but it is NOT Acceptance Testing

# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems
- **Don't** Record-and-playback production data. This has a role, but it is NOT Acceptance Testing
- **Don't** dump production data to your test systems, instead define the absolute minimum data that you need

# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems
- **Don't** Record-and-playback production data. This has a role, but it is NOT Acceptance Testing
- **Don't** dump production data to your test systems, instead define the absolute minimum data that you need
- **Don't** assume Nasty Automated Testing Products<sup>(tm)</sup> will do what you need. Be very sceptical about them. Start with YOUR strategy and evaluate tools against that.

# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems
- **Don't** Record-and-playback production data. This has a role, but it is NOT Acceptance Testing
- **Don't** dump production data to your test systems, instead define the absolute minimum data that you need
- **Don't** assume Nasty Automated Testing Products<sup>(tm)</sup> will do what you need. Be very sceptical about them. Start with YOUR strategy and evaluate tools against that.
- **Don't** have a separate Testing/QA team! Quality is down to everyone - Developers own Acceptance Tests!!!

# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems
- **Don't** Record-and-playback production data. This has a role, but it is NOT Acceptance Testing
- **Don't** dump production data to your test systems, instead define the absolute minimum data that you need
- **Don't** assume Nasty Automated Testing Products<sup>(tm)</sup> will do what you need. Be very sceptical about them. Start with YOUR strategy and evaluate tools against that.
- **Don't** have a separate Testing/QA team! Quality is down to everyone - Developers own Acceptance Tests!!!
- **Don't** let every Test start and init the app. Optimise for Cycle-Time, be efficient in your use of test environments.



# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems
- **Don't** Record-and-playback production data. This has a role, but it is NOT Acceptance Testing
- **Don't** dump production data to your test systems, instead define the absolute minimum data that you need
- **Don't** assume Nasty Automated Testing Products<sup>(tm)</sup> will do what you need. Be very sceptical about them. Start with YOUR strategy and evaluate tools against that.
- **Don't** have a separate Testing/QA team! Quality is down to everyone - Developers own Acceptance Tests!!!
- **Don't** let every Test start and init the app. Optimise for Cycle-Time, be efficient in your use of test environments.
- **Don't** include Systems outside of your control in your Acceptance Test Scope

# Anti-Patterns in Acceptance Testing

- **Don't** use UI Record-and-playback Systems
- **Don't** Record-and-playback production data. This has a role, but it is NOT Acceptance Testing
- **Don't** dump production data to your test systems, instead define the absolute minimum data that you need
- **Don't** assume Nasty Automated Testing Products<sup>(tm)</sup> will do what you need. Be very sceptical about them. Start with YOUR strategy and evaluate tools against that.
- **Don't** have a separate Testing/QA team! Quality is down to everyone - Developers own Acceptance Tests!!!
- **Don't** let every Test start and init the app. Optimise for Cycle-Time, be efficient in your use of test environments.
- **Don't** include Systems outside of your control in your Acceptance Test Scope
- **Don't** Put 'wait()' instructions in your tests hoping it will solve intermittency

# Tricks for Success

# Tricks for Success

- **Do** Ensure That Developers Own the Tests

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”



# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another
- **Do** Keep Your Tests Repeatable

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another
- **Do** Keep Your Tests Repeatable
- **Do** Use the Language of the Problem Domain - Do try the DSL approach, whatever your tech.

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another
- **Do** Keep Your Tests Repeatable
- **Do** Use the Language of the Problem Domain - Do try the DSL approach, whatever your tech.
- **Do** Stub External Systems

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another
- **Do** Keep Your Tests Repeatable
- **Do** Use the Language of the Problem Domain - Do try the DSL approach, whatever your tech.
- **Do** Stub External Systems
- **Do** Test in “Production-Like” Environments

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another
- **Do** Keep Your Tests Repeatable
- **Do** Use the Language of the Problem Domain - Do try the DSL approach, whatever your tech.
- **Do** Stub External Systems
- **Do** Test in “Production-Like” Environments
- **Do** Make Instructions Appear Synchronous at the Level of the Test Case

# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another
- **Do** Keep Your Tests Repeatable
- **Do** Use the Language of the Problem Domain - Do try the DSL approach, whatever your tech.
- **Do** Stub External Systems
- **Do** Test in “Production-Like” Environments
- **Do** Make Instructions Appear Synchronous at the Level of the Test Case
- **Do** Test for ANY change



# Tricks for Success

- **Do** Ensure That Developers Own the Tests
- **Do** Focus Your Tests on “What” not “How”
- **Do** Think of Your Tests as “Executable Specifications”
- **Do** Make Acceptance Testing Part of your “Definition of Done”
- **Do** Keep Tests Isolated from one-another
- **Do** Keep Your Tests Repeatable
- **Do** Use the Language of the Problem Domain - Do try the DSL approach, whatever your tech.
- **Do** Stub External Systems
- **Do** Test in “Production-Like” Environments
- **Do** Make Instructions Appear Synchronous at the Level of the Test Case
- **Do** Test for ANY change
- **Do** Keep your Tests Efficient

# Q&A



<http://www.continuous-delivery.co.uk>

**Dave Farley**

<http://www.davefarley.net>

@davefarley77

