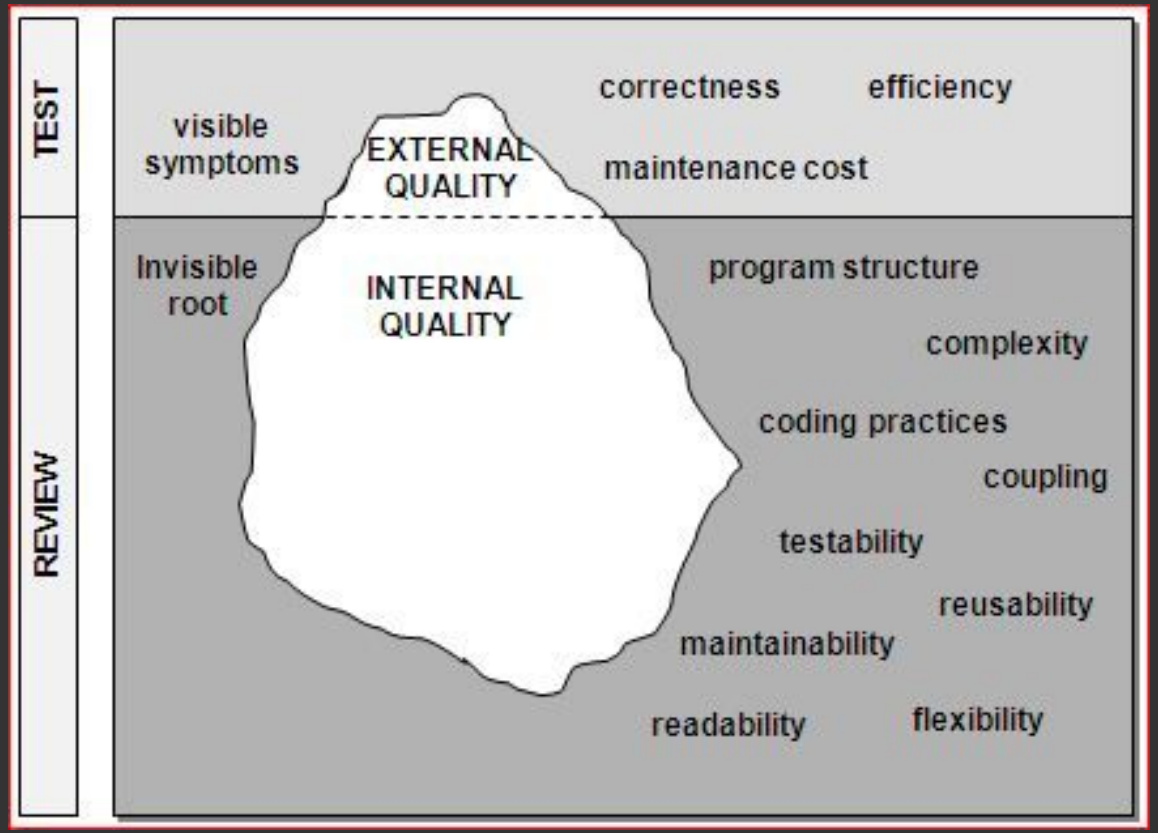
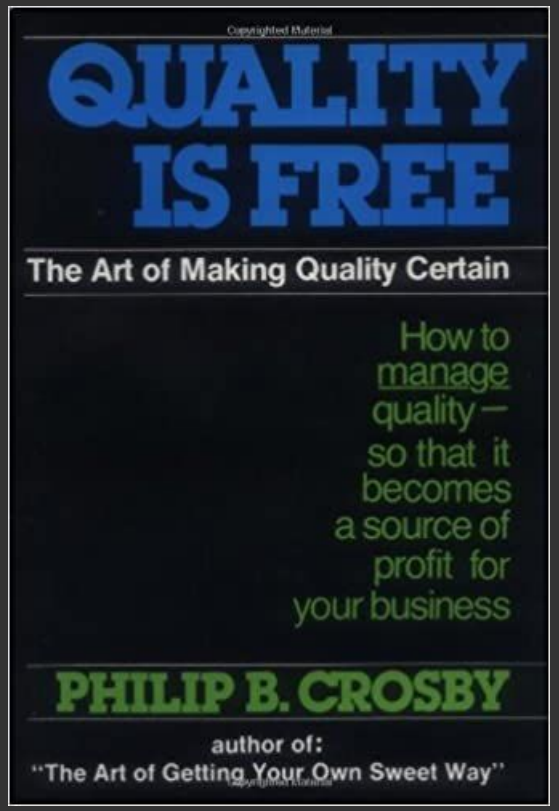
Beyond acceptance tests

Moving to the future

The Software Quality iceberg

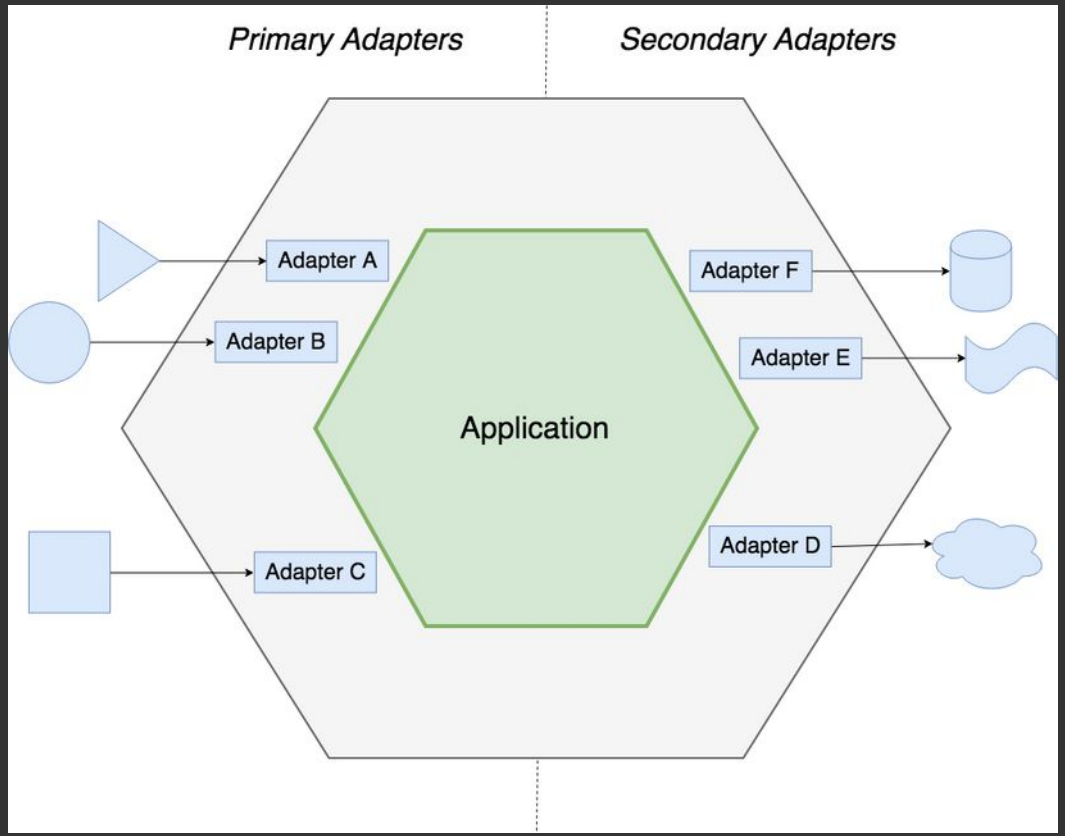


from
Cost of Quality...

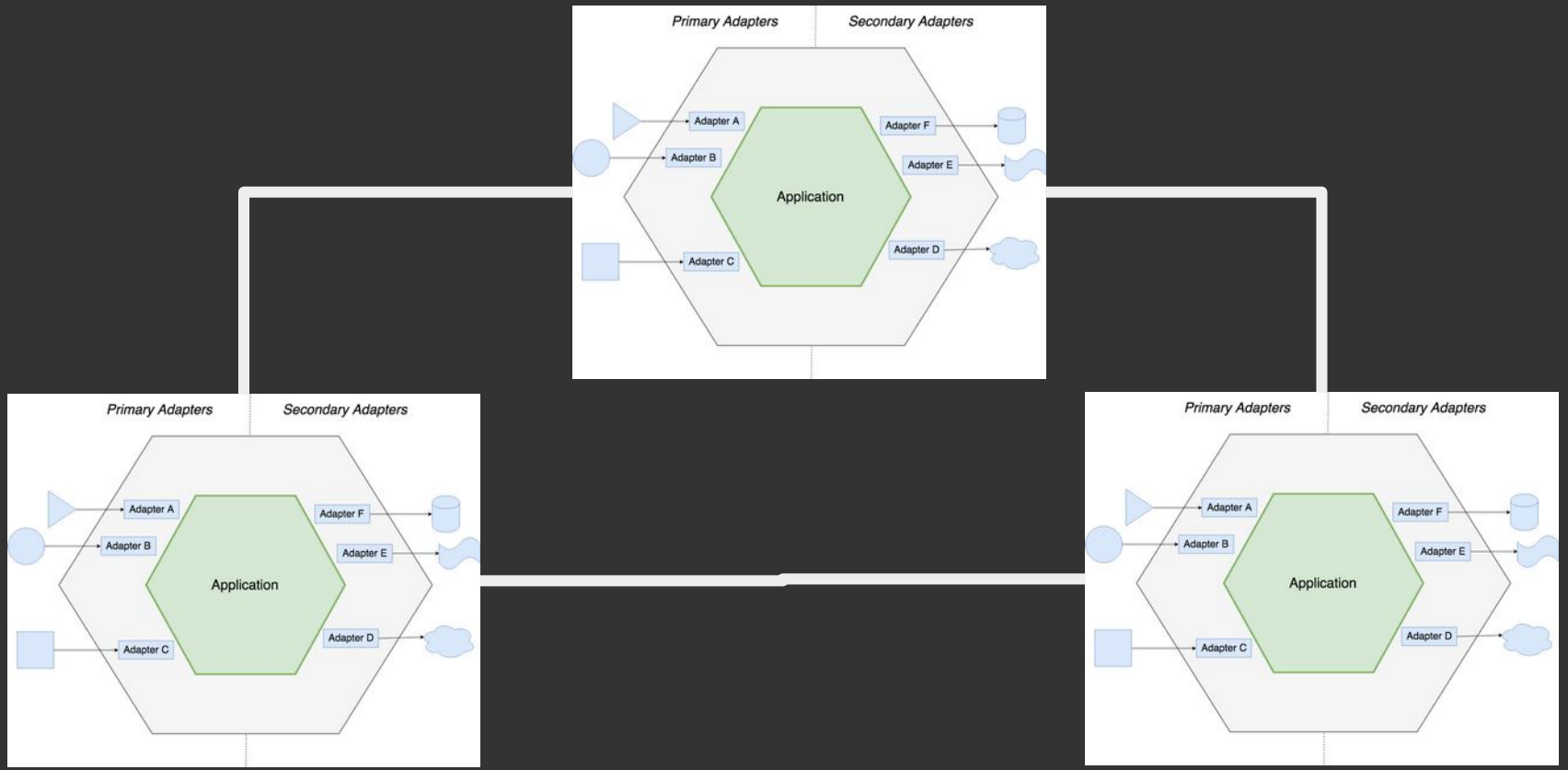


...to the Price of Nonconformance (not meeting quality standards)

Hexagonal architecture

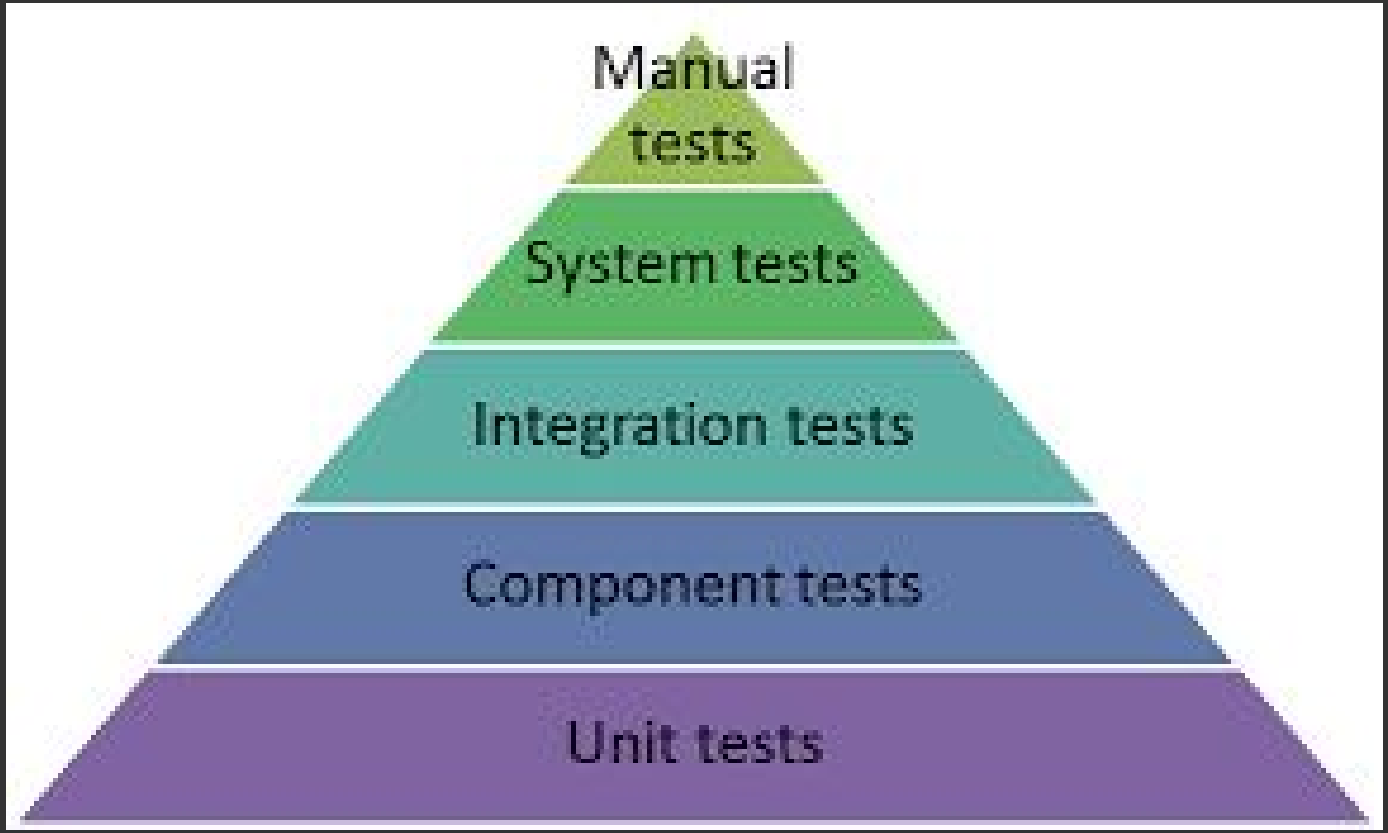


Hexagonal architecture on Microservices



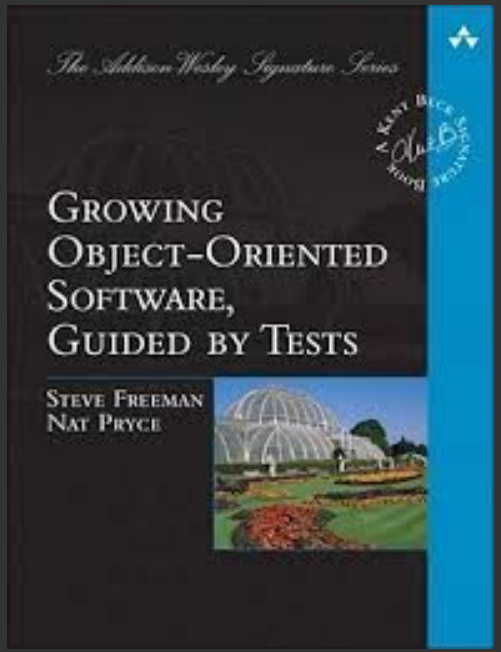
Our journey

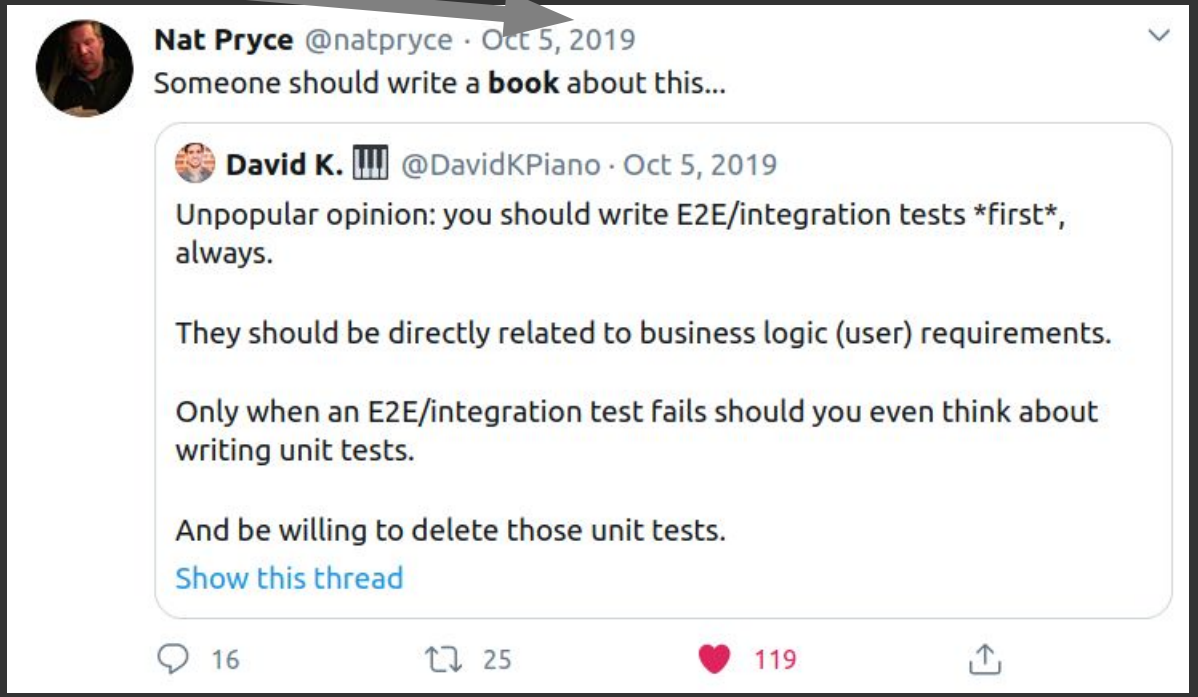
Test Pyramid



Write Acceptance Tests First

2009





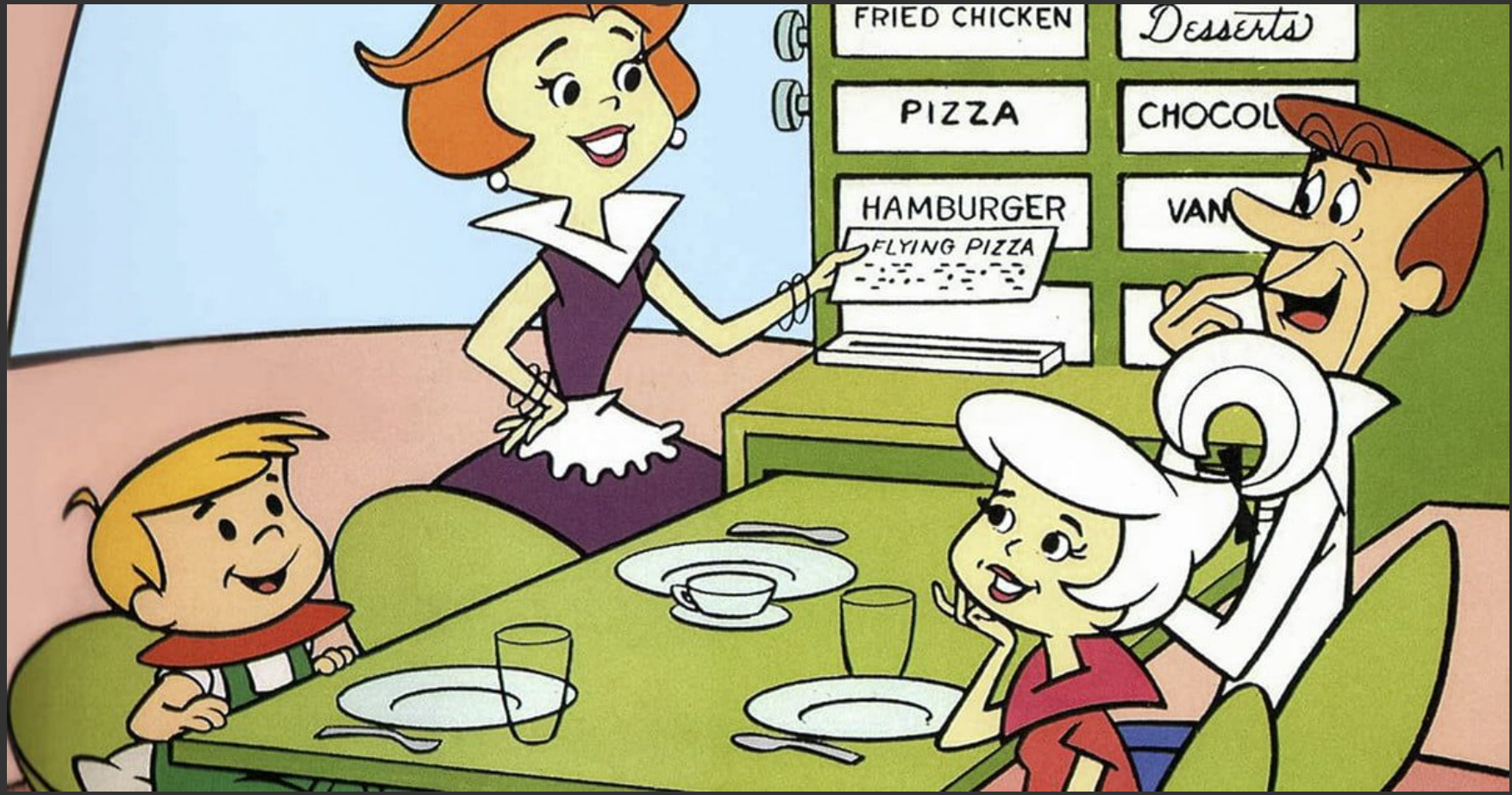
Cucumber

```
Feature: In order to let customers organise their information across
themes on various pages.
         As an administrator of a micro-site
         I want to be able to add subpages

Scenario: Adding a subpage
  Given I am logged in
  Given a micro-site with a home page
  When I press "Add subpage"
  And I fill in "Title" with "Gallery"
  And I press "Ok"
  Then I should see a document called "Gallery"
```

Screenplay Pattern

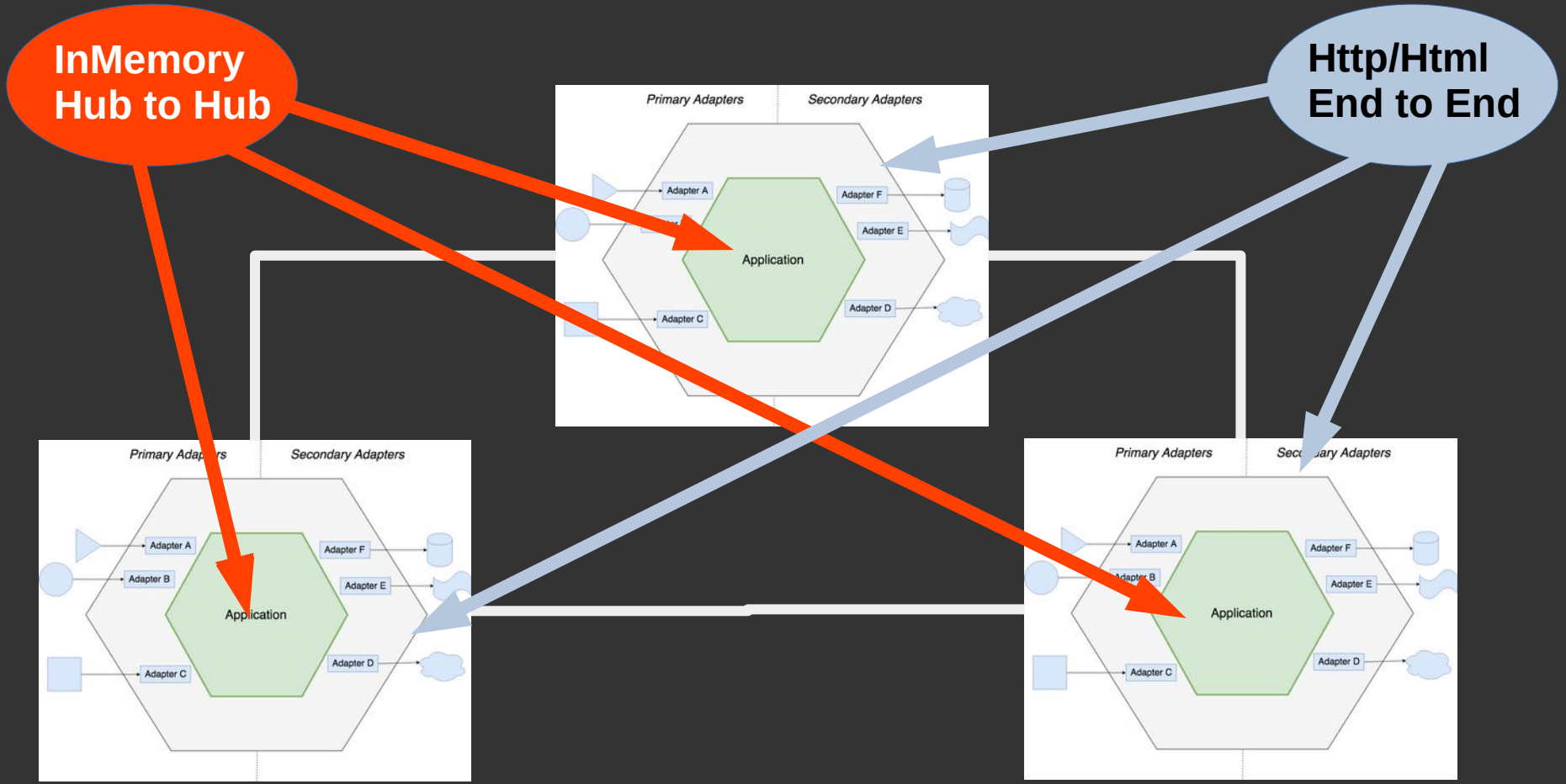
```
@RunWith(SerenityRunner.class)
public class SearchByKeywordStory {

    Actor anna = Actor.named("Anna");

    @Before
    public void annaCanBrowseTheWeb() {
        anna.can(BrowseTheWeb.with(herBrowser));
    }

    @Test
    public void search_results_should_show_the_search_term_in_the_title() {

        givenThat(anna).wasAbleTo(openTheApplication);

        when(anna).attemptsTo(Search.forTheTerm("BDD In Action"));

        then(anna).should(eventually(seeThat(TheWebPage.title(),
                                     containsString("BDD In Action"))));
    }
}
```

http://serenity-bdd.info/docs/articles/screenplay-tutorial.html


FRIED CHICKEN
Desserts
PIZZA
CHOCOL
HAMBURGER
VAN
FLYING PIZZA

Our choice

1 Domain Under Test – 2 Protocols



Domain-Driven Tests

Our needs

1. Use the domain, not the UI, to describe the scenario
2. Run multiple times to verify different abstraction levels
3. Express expectations at higher level

Domain-Driven Tests
Benefits

1. Test the feature works end-to-end
2. Document the feature
3. No business logic in the infrastructure layer
4. No infrastructure details in the business logic

Domain-Driven Tests

The process

1. Write the Http DDT
2. Implement the adapter
3. Define the model
4. Write the InMemory DDT
5. Close the gaps

uberto / **pesticide**

Unwatch 1 | Star 4

Code | Issues 0 | Pull requests 0 | Actions | Projects 0 | Wiki | Security 0 | Insights | Settings

A Library To Write Domain-Driven Tests, written in Kotlin on top of Junit5

Manage topics

71 commits | **1** branch | **0** packages | **0** releases | **1** contributor | Ap

Branch: **master** | **New pull request** | **Create new file** | **Upload files** | **Find file** | **Clone**

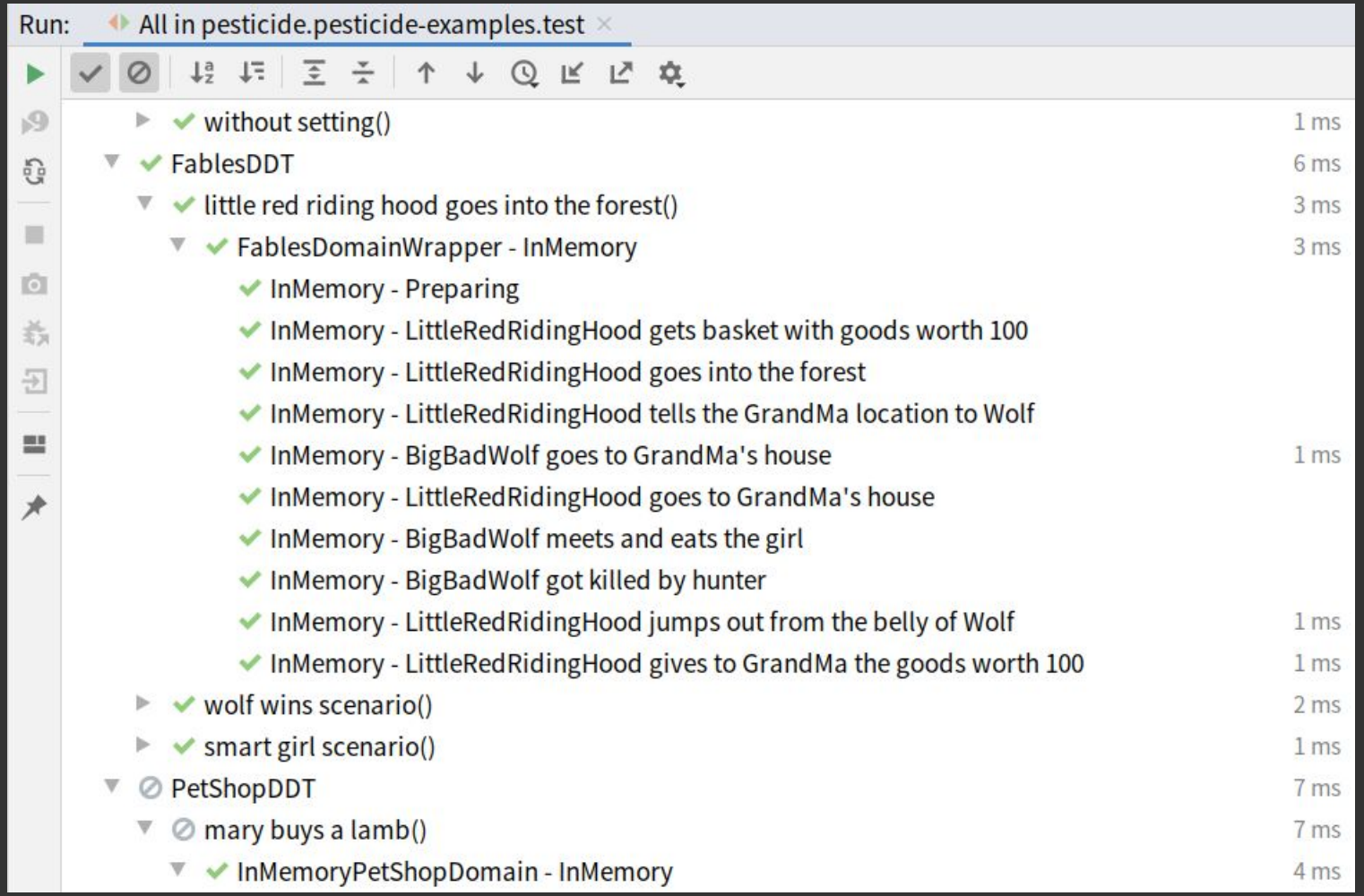
uberto added deps in readme — Latest commit 366508

gradle/wrapper	Initial commit
pesticide-core	improved the step auto naming
pesticide-examples-java	improved the step auto naming
pesticide-examples	improved the step auto naming
scripts	better explanations
.gitignore	separated examples from pesticide-core

What does it look like

```
class FablesDDT : DomainDrivenTest<FablesDomainWrapper>(setOf(FablesDomainWrapper())) {

    val littleRedRidingHood by NamedActor(::Human)
    val bigBadWolf by NamedActor(::Wolf)

    @DDT
    fun `little red riding hood goes into the forest`() : Stream<DynamicContainer> = ddtScenario {

        setting { this: FablesDomainWrapper
            aGrandMaLivingAloneIntoTheForest()
        } atRise play(
            littleRedRidingHood.`gets basket with goods worth $`( value: 100),
            littleRedRidingHood.`goes into the forest`(),
            littleRedRidingHood.`tells the GrandMa location to Wolf`(),
            bigBadWolf.`goes to GrandMa's house`(),
            littleRedRidingHood.`goes to GrandMa's house`(),
            bigBadWolf.`meets and eats the girl`(),
            bigBadWolf.`got killed by hunter`(),
            littleRedRidingHood.`jumps out from the belly of Wolf`(),
            littleRedRidingHood.`gives to GrandMa the goods worth $`( expectedValue: 100)
        )
    }
```

What does it look like



Kent Beck @KentBeck · Apr 23

Here's an exercise to discover the quality of your design as revealed by tests:

1. Print a test
2. Explain the scenario being tested aloud
3. While you explain, highlight the part of the test you mention
4. Look at the density of highlights

7 120 318

Kent Beck @KentBeck · Apr 23

I've seen tests that were 5% information and 95% test mechanism & extraneous setup. The design of the code being tested in such a case is likely to have many opportunities for improvement.

Try this and post your results as replies.

9 19 99

Uberto Barbini
@ramtop

Replying to @KentBeck

Easy with Pesticide :)
you are forced to put all the setup inside the scenario or the actors

Family Picture

Feature
Scenario
DomainUnderTest
Protocol
Actor
Step
DSL for tests

Elements

`DomainUnderTest:` where we define the api to the domain for tests

`Protocol:` how we are accessing it. InMemory, Http, Html...

Elements

```
class HttpRestPetshopDomain(val host: String, val port: Int) : PetShopDomainWrapper {

    val client = JettyClient()

    private fun uri(path: String) :String  = "http://$host:$port/$path"

    fun addPetRequest(pet: Pet) :Request  = Request(POST, uri( path: "pets")).body(pet.toJson())
```

```
class InMemoryPetShopDomain() : PetShopDomainWrapper {

    private val hub = PetShopHub()

    override val protocol = InMemoryHubs

    override fun prepare(): DomainSetUp = Ready
```

```
interface PetShopDomainWrapper : DomainUnderTest<DdtProtocol> {

    fun populateShop(vararg pets: Pet): PetShopDomainWrapper
    fun BuyPet.tryIt(): PetShopDomainWrapper

    fun PetPrice.askIt(): PetShopDomainWrapper
    fun PetList.askIt(): PetShopDomainWrapper
}

fun allPetShopAbstractions() :Set<PetShopDomainWrapper>  = setOf(
    InMemoryPetShopDomain(),
    HttpRestPetshopDomain( host: "localhost",  port: 8082)
)
```

Elements

Actor: it represent the idea of the real person interacting with the system. We don’t want to write test only covering technicality.

Step: a task that can be completed by the Actor. Here we put low level assertions and test mechanisms.

Elements

```
data class PetBuyer(override val name: String) : DdtActor<PetShopDomainWrapper>() {

    fun `check that the price of $ is $`(petName: String, expectedPrice: Int) : DdtStep<P
        step(petName, expectedPrice) { this: PetShopDomainWrapper
            PetPrice(petName) { price ->
                expectThat(price).isEqualTo(expectedPrice)
            }.askIt()
        }

    fun `buy a $`(petName: String) : DdtStep<PetShopDomainWrapper> =
        step(petName) { this: PetShopDomainWrapper
            BuyPet(petName).tryIt()
        }
```

Elements

Feature: a feature of the system we want to test. It correspond to a DDT test class, with multiple tests inside.

Scenario: a single test method (technically a test factory). It sets up some conditions and let the actors play.

Elements

```
class PetShopDDT : DomainDrivenTest<PetShopDomainWrapper>(
    allPetShopAbstractions()
) {

    val mary by NamedActor(::PetBuyer)

    @DDT
    fun `mary buys a lamb`() : Stream<DynamicContainer>  = ddtScenario {
        val lamb = Pet( name: "lamb",  price: 64)
        val hamster = Pet( name: "hamster",  price: 128)
        setting { this: PetShopDomainWrapper
            populateShop(lamb, hamster)
        } atRise play(
            mary.`check that the price of $ is $`( petName: "lamb",  expectedPrice: 64),
            mary.`check that the price of $ is $`( petName: "hamster",  expectedPrice: 128),
            mary.`buy a $`( petName: "lamb"),
            mary.`check that there are no more $ for sale`( petName: "lamb")
        ) ^ddtScenario
    }
```

Design FAQ

Why actors at all?
Why same actor for all protocols?
Why WIP with a due date?
Why a test for step?
Why not GWT format?
Why a setting block?
Why list of steps?

Live coding (AMA)