Released Under Creative Commons by Naresh Jain

Avatars of TDD

Abstract:

It is very clear to most of the people that testing results in better design. But it might not be so obvious that your approach to testing or the way you think about tests can result in quite different tests and hence its impact on the design.

In this article, I've tried to highlight the commonly used avatars of TDD. By avatar I mean form, style and/or approach. Based on their approach, this article, categorizes the avatars of TDD into two broad categories, Outside-In and Inside-Out. Each category has two distinct avatars under them. I have used the class diagram of the resultant domain model and the test cases from the pairing sessions to explain each of the avatars.

Background:

Test Driven Development or Test Driven Design (TDD) has been around for a while and different developers practice it in different forms. This has resulted in a lot of debates about the right way to do TDD. Over the last two months I have spent some time pairing with TDD practitioners trying to understand their approach and style of TDD. I was not surprised to find quite different approaches. This article is an attempt to capture this information.

Target Audience:

This article is targeted at TDD practitioners. Most of these practitioners must be using these avatars without actually realizing the subtle difference between the avatars. I'm hoping that by highlighting these avatars, they will be able to understand the resultant design implications a little better.

This article is not a TDD tutorial or cook book. It does not explain how to become better at TDD. It just highlights some of my observations pairing with different practitioners.

Things that affect TDD avatar:

1. The way you think about your tests:
    a. Functional or high-level or coarse-grain
    b. Technical or low-level or fine-grained
2. Choice of the testing tool/framework used:
    a. Acceptance Testing tool like Fit/Fitnesse or Selenium
    b. Unit Testing framework like xUnit or TestNG
    c. Mocking framework like MockObjects or EasyMock
3. Choice of the programming language:
    a. Static language like Java or C#
    b. Dynamic language like Ruby or Python
4. Choice of Paradigm:
    a. Object Oriented
    b. Functional

Broad Categorization of TDD Avatars:

Broadly we can classify the avatars of TDD into 2 categories:

1. Outside-in
2. Inside-out

In the outside-in category developers starts with a high-level functional test at a story or use case level and drills into the low-level unit tests as development continues. The objective of development is to make the high-level functional tests work. In other words developers start with Story tests[1] or Business-facing team supporting tests and move towards Unit tests or Technology-facing programmer supporting tests. It turns out that these tests are heavily state-based tests.

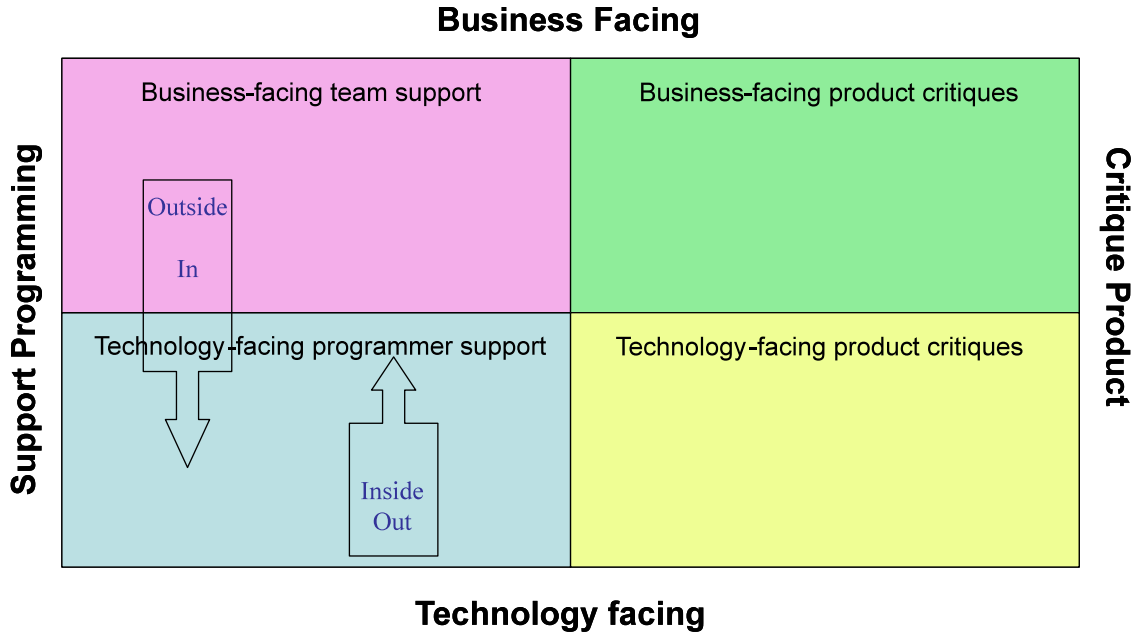


Diagram 1: Brian Marick's Test Categorization[2]

In the inside-out category the developers take a story or use case, do some really high-level analysis & design. Possibly, also some level of object modeling in their heads or on a white board. This activity should take less than 15-30 mins. Once they internalize the story or use case, they start writing technology-facing programmer supporting tests to drive the design. Very quickly they start building the story bottom-up. The objective of development is to make all these low-level unit tests working all the time. These tests are heavily interaction-based tests. I have not really seen these tests evolve into high-level acceptance tests and hence the inside-out arrow in diagram 1.

It is important to note that these avatars are not mutually exclusive. TDD practitioners tend to jump back and forth. But on a given project, the system design is heavily influenced by one of the avatars and not so much by the combination.

Testing avatars that fit under the Outside-in category:

1. Acceptance Tests Driven Development (ATDD) using an acceptance testing tool/framework. This avatar purely relies on acceptance tests written using the acceptance testing tool/framework to drive the development. It is also known as Example Driven Development (EDD). Quite often you find developers stubbing out external components in this avatar.
2. Acceptance Tests Driven Development (ATDD) using a Unit testing framework like xUnit. This avatar purely relies on acceptance tests written using a unit testing framework to drive the development. Sometimes developers also use a mocking framework to mock out external dependencies.

Testing avatars that fit under the Inside-out category:

1. Test Driven Development purely using a unit testing framework like xUnit. In this avatar, developers write unit tests to drive development. The tests are mainly state-based, asserting values at the end of the test method.
2. TDD using Mock Objects. In this avatar, developers write mostly interaction based unit tests using Mocking framework. There are still some state based unit tests but mostly at boundaries. For example, in this avatar, developers put a mock mail service to ensure an email gets sent. In the previous avatar developers might test state to verify that the sent mail folder now contains a copy of the mail.

This is just a beginning. I believe there are lots of avatars that I won't be able to cover here. Never the less, its worth mentioning Behavior Driven Development (BDD) [3] in this context.

The purpose of BDD is to bring behavior-based TDD (.i.e. interactions rather than just state) and acceptance-test driven design under a common linguistic umbrella. The idea is to bring the goodness of Outside-in and Inside-out tests into one avatar. The tests are really about interactions between objects and expected behavior of the system from outside-in. Each "test method" is called a "behavior" and takes the form of a sentence saying what the object should do.

Developers start with a Story, which contains Acceptance Criteria in the form of Scenarios. Each scenario has a number of steps, of the form: Given [some context] When [some event occurs] Then [some outcome should occur]. BDD jumps straight into the interactions-and-roles stage, encouraging developers to think of the behavior of the component that they are developing, and of the roles and responsibilities of the other objects it interacts with. Code developed using this avatar is characterized by narrow, well-named interfaces describing the roles the objects play for one another.

Following sections of the article are going to walk through each avatar with some code examples:

Problem definition:

Following is the description of a Veterinary Information System. I used this problem for all the pairing sessions. Thanks to Don Roberts for sharing this problem with us.

Design a Veterinary Information System for a Clinic. Veterinary medical system is very similar to human medical system with one exception, all the patients are animals. Each patient is owned by a person, who brings the patient to the clinic and pays the bills.

The person in charge for IT department of the clinic gave us the following Use Cases to start development.
Dave Atkins brings his pet named Fluffy into the clinic for a routine check up and shots. The veterinarian charges him for the routine office visit and the Rabies vaccination. Dave pays cash before he leaves and is provided with a receipt for the services.

In the following sections, I'll take this problem definition and drive my development using various TDD avatars.

Acceptance Tests Driven Development (ATDD) using an acceptance testing tool/framework:

For this example we'll use Java 5 as the programming language and Fitnesse as the acceptance testing tool.

`Fitnesse Document:`
Assertions: 17 right, 0 wrong, 0 ignored, 0 exceptions

com.vis.billing.fixtures.PaidCashBill

account details		
account number	patient name	owner name
1001	Fluffy	Dave Atkins

procedure details	
name	cost
Routine Office Visit	250
Rabies Vaccination	50

bill				
account number?	owner name?	patient name?	total?	paid?
1001	Dave Atkins	Fluffy	300	false

procedure details on the bill	
name	cost

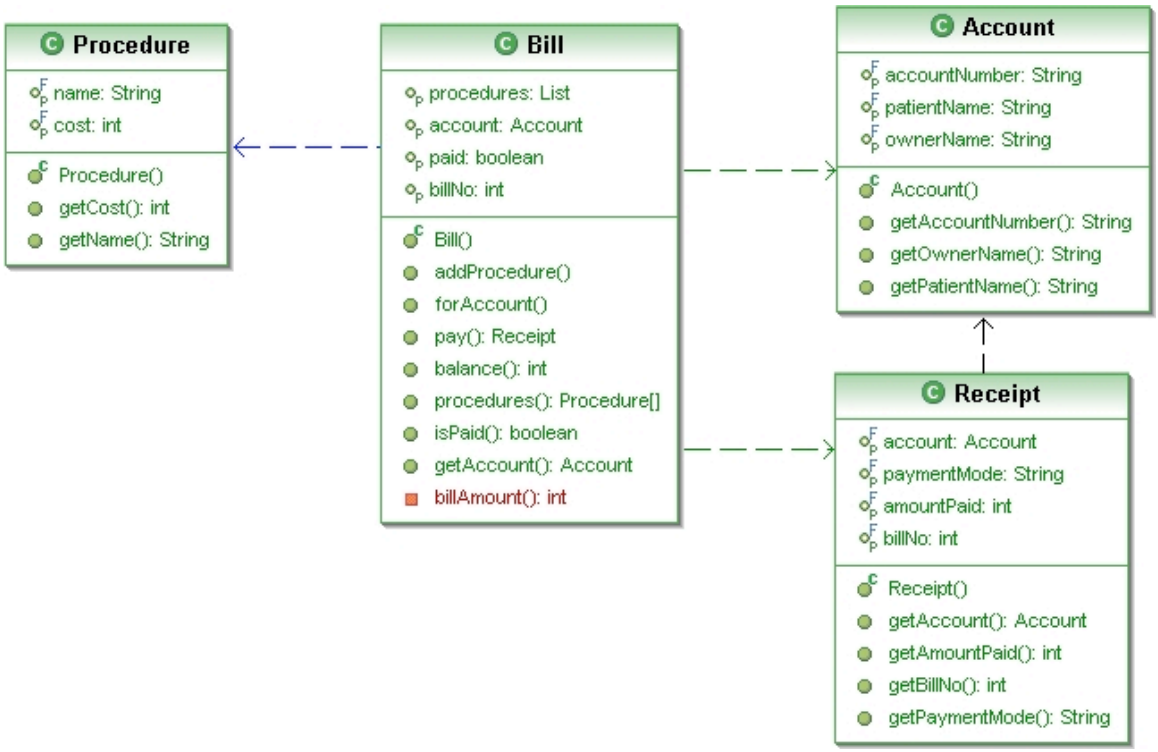
Routine Office Visit	250
Rabies Vaccination	50

pay	Cash				
patient name?	owner name?	account number?	bill no?	payment method?	amount paid?
Fluffy	Dave Atkins	1001	1	Cash	300

check	paid	true

check	total	0

Class Diagram:



Some Observations:

1. The tests are heavily state based
2. Lots of getters and setters in the code. This breaks encapsulation.
3. Encourages the creation of concrete classes over interfaces.
4. Quite effective at surfacing the domain objects.
5. Can involve the business roles very easily.
6. Helpful in building a common vocabulary or Domain Specific language which the whole team can use.

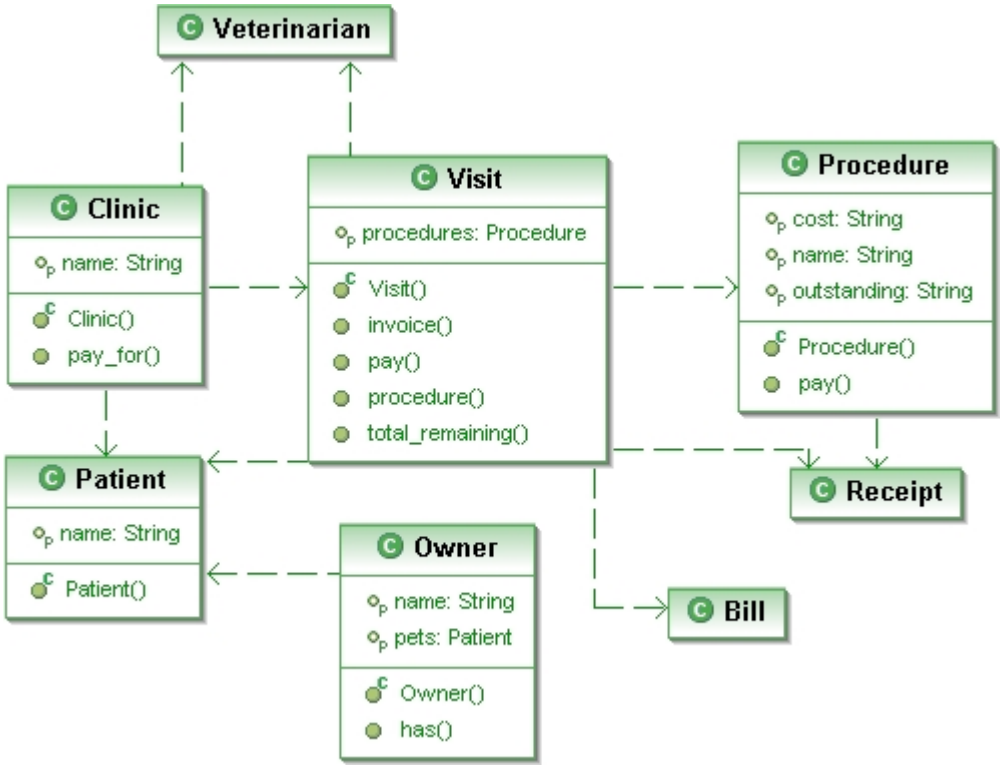
7. Very difficult to test/express negative path scenarios. It's not easy to simulate exceptions.
8. Does a great job at separating the data from the test logic. Very easy to pump in different data sets or different examples.
9. Makes it quite difficult to refactor the code as the fit document can go out of sync.
10. Poor support of integrating the tool within the IDE. Can be irritating to switch between the browser and IDE.

***Acceptance Tests Driven Development (ATDD) using unit testing tool/framework*:** For this example we'll use Ruby as the programming language and Test::Unit as the unit testing framework.

Acceptance test:

```
class FluffyTest < Test::Unit::TestCase
  def test_examination_and_shots
    vet = Veterinarian.new
    clinic = Clinic.new "Main Line health"
    dave = Owner.new "Dave"
    fluffy = Patient.new "Fluffy"
    dave.has fluffy
    visit = clinic.visit(fluffy, vet) do |v|
      v.procedure "Rabies vaccination", 50
    end

    invoice = visit.invoice
    assert_equal invoice.to_s,
<<-INVOICE
Routine visit: $200
Rabies vaccination: $50
Total: $250
INVOICE

    receipt = clinic.pay_for visit, 100

    assert_equal receipt.to_s,
<<-RECEIPT
Paid $100 for Routine visit
Paid $0 for Rabies vaccination
Outstanding: $150
RECEIPT
  end
end
```

Class Diagram:



Some observations:

1. The tests are heavily state based
2. From a developer's point of view this approach can be very quick and intuitive. Quick feedback, as you can execute the tests directly from the IDE.
3. Not as many getters and setters in the code, as in the previous case.
4. Does not force you to create Interfaces. You can create concrete classes or interfaces depending on your choice.
5. Lots of classes tend to be just dummy data holders. They don't seem to have much behavior on them. May be as we add more functionality these classes might be able to carry their weight, but right now they smell.
6. Data and test logic are not very clearly separated. Quite difficult to test same scenario with different data sets
7. Writing the tests in the same language as the production code, has the following
    a. Advantages:
        i. As the domain model evolves, refactoring the tests can be very easy
        ii. Tests tend to be far more powerful and much closer to the domain model
    b. Disadvantages:
        i. Developers tend to own and drive these tests. The tests become more technology facing than business facing
        ii. There is a big psychological barrier for the business roles to cross before they start using these tests.
8. As developers start evolving their domain model, new classes start evolving. If there is a complicated logic in some unit (class or method), developers start test

driving complicated logic using a technology-facing unit test. So they tend to switch the avatar.

Test Driven Development purely using a unit testing framework:

For this example we'll use Java 5 as the programming language and JUnit 4 as the unit testing framework.

Unit Tests:

`ClinicTest`

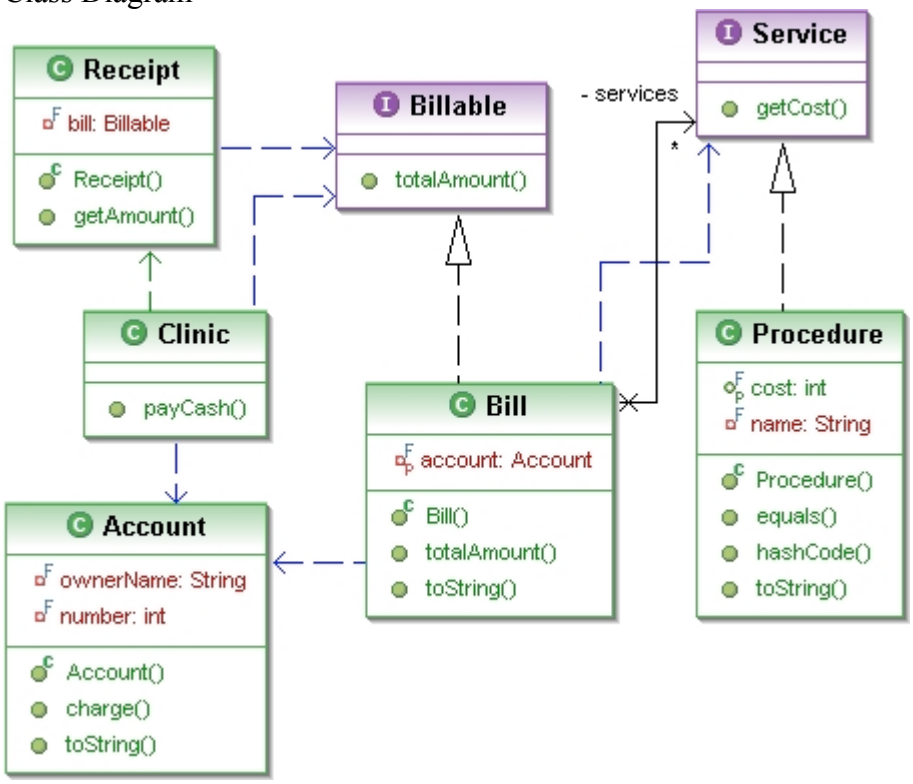
```
public class ClinicTest {

   private Account dave;
   private Clinic clinic;

   @Before
   public void setUp() {
      clinic = new Clinic();
   }

   @Test
   public void amountOnTheReceiptShouldMatchBillableAmount() throws Exception {
      Billable billable = new Billable() {
         public int totalAmount() {
            return 0;
         }
      };
      dave = new Account(101, "Dave");

      Receipt rcpt = clinic.payCash(dave, billable);
      assertEquals("Amount on receipt does match procedure cost", billable
            .totalAmount(), rcpt.getAmount());
   }

   @Test
   public void customerPayesBillableAmountForCashTransaction() throws Exception {
      final int billableAmount = 56;
      class AmountCharged {
         int charged;
      };
      final AmountCharged charged = new AmountCharged();

      Billable billable = new Billable() {
         public int totalAmount() {
            return billableAmount;
```

```
            }
        };

        dave = new Account(101, "Dave") {
            public void charge(int amount) {
                charged.charged = amount;
            }
        };

        clinic.payCash(dave, billable);
        assertEquals("Account is not charged billable amount", billableAmount,
                charged.charged);
    }
}
```

`BillableTest`

```
public class BillableTest {

    private static final Account daveSAccount = new Account(101, "Dave");
    private static final List<Service> services = new ArrayList<Service>();
    private Billable bill;

    @Test
    public void totalBillableAmountShouldBeZeroIfNoServicesAreProvided()
            throws Exception {
        bill = new Bill(daveSAccount, services);
        assertEquals("Total amount is not Zero", 0, bill.totalAmount());
    }

    @Test
    public void totalBillableAmountShouldBeTotalOfEachServiceProvided()
            throws Exception {
        services.add(new Procedure("Rabies Vaccination", 250));
        services.add(new Procedure("Regular office Visit", 50));

        bill = new Bill(daveSAccount, services);

        assertEquals("Total Amount is not 300", 300, bill.totalAmount());
    }

    @After
    public void cleanUp() {
        services.clear();
    }
}
```

Class Diagram



Observations:

1. Mostly state-based tests.
2. Nicely decoupled design with the two interfaces
3. There seems to a fair distribution of behavior throughout the system
4. We did not end up with dummy data holders.
5. Some developers also use poor man's mocks to test driven as they are not using a mocking framework. In static languages, the way we do this is by using anonymous inner classes to inject behavior into the dependent method. In dynamic languages developers can change the behavior of the dependent method directly from the test.
6. Some of the tests could have been simpler with usage of a mocking framework
7. Can result in well encapsulated code. Very little or not getter and setters.
8. As the domain model evolves, we refactored the code by extracting classes into independent classes. It can get tricky to figure out if you need to move the test along with it or leave the test methods in the original test.
9. Developers can get really quick feedback since the tests can be executed directly from the IDE.

TDD using Mock Objects:

For this example we'll use Java 5 as the programming language, JUnit 4 as the unit testing framework and EasyMock 2.2 as the Mocking Framework.
Unit Tests

`ChargeAccountForServices`:

```
public class ChargeAccountForServices {
   private static final Billable bill = createMock(Billable.class);
   private static final Accountable account = createMock(Accountable.class);
   private static final Clinic clinic = new Clinic();

   @Before
   public void setUp() {
      reset(account);
      reset(bill);
   }

   @Test
   public void shouldMakePaymentsAgainstAnAccount() throws Exception {
      account.paid(bill);
      replay(account);
      clinic.pay(300, bill, account);
      verify(account);
   }

   @Test
   public void shouldHaveZeroAmountDueOnReceiptIfCompletePaymentIsMade()
         throws Exception {
      expect(bill.amount()).andReturn(300);
      replay(bill);
      Receipt receipt = clinic.pay(300, bill, account);
      verify(bill);
      assertEquals(300, receipt.amount());
      assertEquals(0, receipt.amountDue());
   }

   @Test
   public void shouldDisplayAmountDueOnTheReceiptIfIncompletePaymentIsMade()
         throws Exception {
      expect(bill.amount()).andReturn(500);
      replay(bill);
      Receipt receipt = clinic.pay(300, bill, account);
      verify(bill);
      assertEquals(300, receipt.amount());
      assertEquals(200, receipt.amountDue());
   }
}
```

`CreateBillForClientAccount`

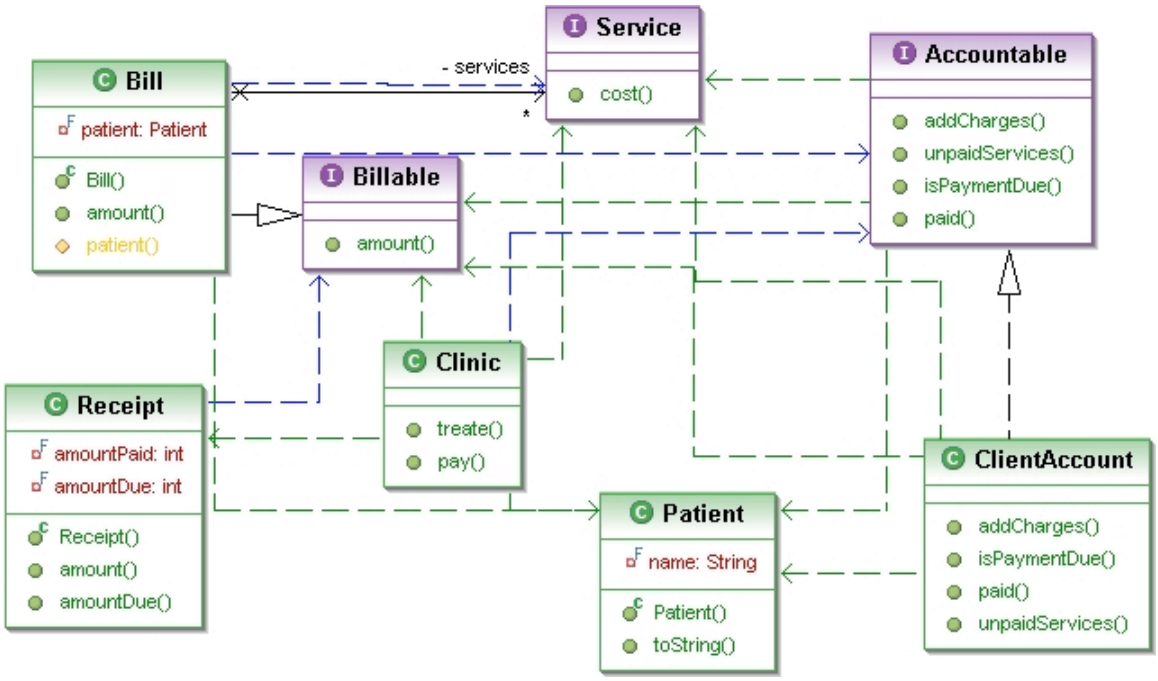
```
public class CreateBillForClientAccount {
   private static final List<Service> services = new ArrayList<Service>();
   private static final Accountable account = createMock(Accountable.class);
   private Bill bill;

   @Before
   public void setUp() {
      reset(account);
   }

   @Test
   public void shouldThrowExceptionIfAccountIsNotDueForPayment() throws Exception
{
      expect(account.isPaymentDue()).andReturn(false);
      replay(account);
      try {
         new Bill(account, null);
      } catch (NoPaymentDueException e) {
         // expected
      }
      verify(account);
   }

   @Test
   public void shouldCreateABillWithTheTotalCostOfAllTheServices()
         throws Exception {
      IMocksControl control = createControl();
      Service rabiesVaccination = control.createMock(Service.class);
      Service routineVisit = control.createMock(Service.class);
      services.add(rabiesVaccination);
      services.add(routineVisit);

      expect(account.isPaymentDue()).andReturn(true);
      expect(account.unpaidServices()).andReturn(services);
      bill();

      expect(rabiesVaccination.cost()).andReturn(250);
      expect(routineVisit.cost()).andReturn(50);

      control.replay();

      assertEquals(300, bill.amount());
      control.verify();
   }

   private void bill() throws NoPaymentDueException {
```

```
        replay(account);
        bill = new Bill(account, null);
        verify(account);
    }

    @After
    public void cleanUp() {
        services.clear();
    }
}
```

`HandleClientVisitDetails`

```
public class HandleClientVisitDetails {
    private static final Service rabiesVaccination = createNiceMock(Service.class);
    private static final Service routineVisit = createNiceMock(Service.class);
    private static final Clinic clinic = new Clinic();
    private static final Patient fluffy = new Patient("Fluffy");
    private static final Accountable account = createMock(Accountable.class);

    @Test
    public void shouldKeepTrackOfAllServicesTakenByThePatient() throws Exception {
        List<Service> services = new ArrayList<Service>();
        services.add(rabiesVaccination);
        services.add(routineVisit);

        reset(account);

        account.addCharges(fluffy, services);

        replay(account);
        clinic.treat(fluffy, services, account);
        verify(account);
    }

    @Test
    public void shouldNotKeepTrackOfVisitIfNoServicesAreTakenByThePatient()
            throws Exception {
        List<Service> services = new ArrayList<Service>();

        reset(account);

        replay(account);
        clinic.treate(fluffy, services, account);
        verify(account);
    }
}
```

Class Diagram:



Observations:

1. Mostly interaction-based tests.
2. Nicely decoupled design with the three interfaces
3. Good distribution of behavior throughout the system
4. Very helpful to test/express negative path scenarios. It's really simple to simulate exceptions.
5. Test names are not nouns of the form <Class>Test. They are explaining the behavior you have in mind. There is almost no reference to the word "Test".
6. Really focus on using tests as executable specification. Very helpful to generate documents from these tests.
7. The API is expressive and closer to natural speaking language. Has similar feel as the acceptance tests have. Helps in building a domain specific language
8. We did not end up with dummy data holders. In fact we have not yet implemented the Service interface.
9. Quick developer feedback as the tests can be executed from the IDE.

Conclusion

Hopefully by now it might be clear how your approach to testing or the way you think about tests can result in different types of tests and their impact on the design.

This article was the first attempt to understand these different avatars of TDD. There could possibly be subsequent articles explain each of these avatars in detail with a concrete example.

References

1. Brian Marick's Categorization of tests- http://www.testing.com/cgi-bin/blog/2003/08/21#agile-testing-project-1
2. Introduction to BDD: http://dannorth.net/introducing-bdd

Acknowledgement

I thank the all the TDD practitioners who paired with me on this problem. I also thank Dan Robert for letting me use this problem. I really appreciate all the people who reviewed this paper and gave me timely feedback to improve the quality of the article.

About the Author

Naresh Jain is an internationally recognized Technology & Process Expert. Over the last decade, he has helped many Fortune 500 companies like Google, Yahoo, Amazon, HP, Siemens Medical, GE Energy, Schlumberger, EMC, Alcatel Lucent, to name a few clients.

Naresh is leading two tech-startups, which build tablet-based adaptive educational apps for kids, conference management software, social-media search tool and a content curation and voting platform. His startups are trying to figure out the secret sauce for blending gamification and social learning using the latest gadgets.

As an independent consultant, Naresh worked with many fortune 500 software organizations and startups to deliver mission critical enterprise applications. Having played various roles of Founder, Agile Coach, Quality Evangelist, Technical Lead, Product Owner, Iteration Manager, Scrum Master, Developer, QA, Recruiter, Build Master, Mentor & Trainer, he is well equipped to help your entire organization to rapidly adapt Agile and Lean methods.

Naresh founded the Agile Software community of India, a registered non-profit society to evangelize Agile, Lean and other Light-weight Software Development methods in India. Naresh is responsible for conceptualizing, creating and organizing 50+ Software conferences worldwide.

More: http://nareshjain.com