UNIVERSIDADE DA CORUÑA

FACULTAD DE INFORMÁTICA

DEPARTAMENTO DE TECNOLOGÍAS DE LA INFORMACIÓN Y LAS COMUNICACIONES

PROYECTO DE FIN DE CARRERA DE INGENIERÍA TÉCNICA EN INFORMÁTICA DE SISTEMAS

Realización de un Estudio Práctico del impacto producido por la aplicación del BDD a un diseño DDD siguiendo los principios SOLID y usando la plataforma de Microsoft .NET

Tomo II: Estudio Práctico

Autor: David García Chaves

Tutor: Alfonso Castro Martínez

A Coruña, septiembre de 2010

Índice

Resumen de Contenidos

Pese a que consideramos que lo más recomendable es zambullirse directamente en el Estudio sin que nos detallen donde se aplica un determinado principio o se explica una práctica concreta, hemos decidido incluir un Resumen de Contenidos por Capítulo a modo de guía.

Capítulo 1: El Aggregate Root Cargo (Parte I).

En este capítulo el nivel de detalle desde el punto de vista BDD es alto, como demuestra esta frase que nos encontramos en la introducción del mismo:

"(...) se detallará y justificará cada una de las lineas de código que surjan a raíz de las **BDD_Specs** en el **DDD_Aggregate_Root** `Cargo`."

Los aspectos más reseñables desde el punto de vista de las BDD_Specs son lo siguientes:

- Una vez finalizada la micro-iteración de la BDD_Spec definida en el apartado 1.3 (la primera BDD_Spec), se procede a identificar el RED-GREEN-REFACTOR en la misma.
- En la BDD_Spec definida en el apartado 1.5 se utiliza por primera vez Rhino.Mocks.
- En la BDD_Spec definida en el apartado 1.6:
  - Se introduce el SRP y su relación con los objetos colaboradores y el concepto de "delegar".
  - Nos encontramos con un error de diseño grave, detectado, precisamente, por la práctica de BDD (Un problema de tipos entre Clase e Interface).
  - Se produce por primera vez feedback procedente de la micro-iteración, que nos ayuda a refactorizar la BDD_Spec formal utilizando un lenguaje más preciso que probablemente repercuta en el DDD_Ubiquitous_Language.

- En la BDD_Spec definida en el apartado 1.8 ya no se detalla el proceso de la micro-iteración paso a paso.
- En la BDD_Spec definida en el apartado 1.10:
  - Por primera vez se contextualiza el Domain Model, intentando dar una visión global de qué es lo que intentamos resolver. Esto se debe a que dicha BDD_Spec necesita de esa contextualización, mientras que las BDD_Specs anteriores eran suficientes para entender el problema que teníamos entre manos.
  - Por su complejidad, se retoma la descripción de todos las etapas incluidas en la micro-iteración (exceptuando las triviales).
  - Por una definición incompleta de los requisitos, nos vemos en la obligación de cambiar la BDD_Spec y dividirla en dos BDD_Specs distintas.
  - Se introduce el tema de las Dependencias Ocultas en oposición al principio DIP del S.O.L.I.D., y más en concreto, a la IoC/DI.
  - Se introduce la necesidad de una DDD_Factory que se encargue de la creación y verificación de Invariantes del DDD_Aggregate_Root Cargo. Ésto era algo que se podía intuir desde el inicio del desarrollo del DDD_Aggregate_Root Cargo, pero que sin embargo ha sido postergado hasta que no fuese algo imprescindible. Se trataría claramente de una forma de eliminar desperdicio (waste) y cumplir con los principios Lean.
  - Se abre la puerta de forma explícita a una de las Lineas Futuras de la Memoria: Introducir un Framework IoC/DI como Ninject!.
  - Se culmina la implementación con una discusión en la que se pone de manifiesto como la Práctica de Diseño Agile BDD y la Práctica de Diseño DDD, admiten distintas interpretaciones en la implementación de un modelo, así como del diseño del mismo (objetivo primordial de este Estudio). A modo casi de anexo se detalla el proceso necesario para poder postergar dicha decisión ayudándonos del Sistema de Control de Versiones Distribuido Git.

Capítulo 2: El Aggregate Root Cargo.

En este capítulo se busca trasladar el enfoque de la mecánica de la práctica del BDD al componente de diseño del mismo, como demuestra esta frase de la introducción:

“...debemos tener en cuenta que algunos procesos triviales que han sido tratados con profusión y todo lujo de detalles anteriormente (...), no van a recibir ahora la misma clase de atención. Intentaremos, sobre todo, destacar las **reacciones en cadena** que se produzcan.”

Que es aclarada todavía más en este otro parrafo:

“Podemos considerar la analogía de la conducción de un vehículo. Al conducir un coche no pensamos en cambiar las marchas del mismo, simplemente lo hacemos, aunque al principio sí que era necesario fijarnos y recordar que debíamos cambiar de marcha. Con el tiempo nos acostumbramos y podemos centrarnos en lo importante, como no atropellar a los peatones, respetar las señales y circular por nuestro carril sin invadir el carril contrario, por poner un ejemplo.”

Los aspectos más reseñables desde el punto de vista de las BDD_Specs son lo siguientes:

- Previamente al desarrollo de la primera BDD_Spec, se muestra como implementar las interfaces IValueObject<T> y ISpecification<T>, que encapsulan, respectivamente, los patrones DDD_Value_Object y DDD_Specification.
- En la BDD_Spec definida en el apartado 2.3:
  - Se introduce por primera vez la nomenclatura Context / Specification para referirnos a una BDD_Spec, y se establece la equivalencia con NombreDeLaClase / BloqueIT respectivamente.
  - Se crea el SUT a mano, para mitigar los problemas derivados de una limitación de Machine.Specifications.DevelopWithPassion relativa a la inyección de parámetros del mismo tipo en el constructor.
- En la BDD_Spec definida en el apartado 2.5, se introduce un nuevo punto de fricción entre el diseño DDD y las conclusiones que parece mostrarnos la práctica del BDD.
- En la BDD_Spec definida en el apartado 2.6:
  - Se implementa el Happy Day Scenario asociado a la DDD_Specification, mientras en los siguientes apartados (2.7 a 2.10) se implementan los escenarios complementarios.

    - Se demuestra como corregir un error de diseño, que arrastrábamos desde BDD_Specs anteriores y que es una nueva muestra de la incidencia del BDD en los diseños DDD.
- En la BDD_Spec definida en el apartado 2.7, se consigue que los errores lógicos que arrastrábamos desde la BDD_Spec anterior se corrijan, al vernos obligados a escribir el código que de verdad evalúa las condiciones para satisfacer la DDD_Specification.
- Al refactorizar las BDD_Specs de los apartados 2.6 a 2.10:
    - Se muestran dos versiones diferentes de las misma. En ellas se establece una confrontación entre la exactitud en cuanto a la descripción de las interacciones frente a la sencillez semántica a la hora de describir el modelo. Se aclara que en el Estudio se favorece la primera opción frente a la segunda.
    - Se establece la norma no escrita de favorecer que las Specifications (Bloques IT) contengan una sola linea de código.
- En las 5 BDD_Specs definidas en el apartado 2.12:
    - Se implementa el comportamiento heredado por ser un DDD_Value_Object, quedando perfectamente definido el comportamiento deseado, así como todas las interacciones con los objetos colaboradores.
    - Se varía el enfoque y se muestra como implementaríamos las BDD_Specs si no tuviésemos que estar justificando cada paso que damos.
- En la BDD_Spec definida en el apartado 2.13, se ponen de manifiesto las raíces Interaction-Based Testing TDD del BDD y como se puede diseñar un sistema en base a la delegación de responsabilidades a los objetos colaboradores.
- En las BDD_Specs definidas en el apartado 2.14:
    - Se implementa una DDD_Factory atendiendo a la necesidad de satisfacer las invariantes de creación del objeto, así como a la violación flagrante del SRP que supone lanzar excepciones desde el constructor.
    - Se introducen las opciones existentes para testar excepciones desde Machine.Specifications.
- En las BDD_Specs definidas en el apartado 2.16:
    - Se ahonda nuevamente en la problemática de tener que enfrentar un diseño establecido DDD a la maquinaria pesada del BDD, y como ante ese

escenario, nos vemos en la obligación de refinar el Domain Model todavía más.

    - Se ponen de manifiesto los problemas que se derivan de que los tipos definidos por el Framework .NET no dispongan de una interface asociada. En este caso se da la circunstancia de que si el Framework .NET hubiese sido diseñado desde cero a día de hoy y siguiendo una aproximación de diseño BDD, probablemente dispondría de dichas interfaces. Esto se traduce en un cambio de paradigma, pasando del consabido Interaction-Based Testing TDD al State-Based Testing TDD, mostrando el grado de flexibilidad inherente al BDD.

- En las BDD_Specs definidas en el apartado 2.17, se pone de manifiesto la diferencia entre una aserción que consta de varias comprobaciones pero que describe un solo concepto, y una aserción que al contar con varias comprobaciones oculta el hecho de que debería descomponerse en tantas aserciones como comprobaciones contiene, al describir varios conceptos diferentes.

Capítulo 3: Value Object Itinerary. Aplicando BDD a código Preexistente (Legacy Code).

En este capítulo, se deja de lado la naturaleza Test-First del BDD, y se adapta a un escenario en el que es imposible aplicar dicha política. En los escenarios compatibles con el Test-First, primero escribimos los tests y a continuación escribimos el código que hace que los tests pasen. Sin embargo, si el código ya está escrito de antemano, es más difícil medir la utilidad del BDD.

Se inicia el capítulo con:

- Un repaso a la metodología de la micro-iteración.
- La constatación de algunas norma básicas del DDD.
- La contextualización del escenario Legacy Code o Brownfield Development.
- Se detalla una técnica más “Orientada a Objetos” para sustituir los enums.

Los aspectos más interesantes desde el punto de vista de las BDD_Specs son los siguientes:

- En la BDD_Spec definida en el apartado 3.4, se introduce de manera práctica, la versión adaptada a escenarios Legacy Code del BDD.
- En las BDD_Specs definidas en el apartado 3.5:
  - Se introduce la versión adaptada aplicada a un escenario compuesto (Happy Day Scenario y Complementarios).
  - Se depura un escenario imposible, consiguiendo por tanto, un código más simple y expresivo.
- En las BDD_Specs definidas en el apartado 3.7, se depuran varios escenarios imposibles, y tras varias refactorizaciones muy agresivas se consigue reducir el método probado de cuatro a una sola linea.
- En las BDD_Specs definidas en el apartado 3.8:
  - Se prueban las excepciones lanzadas por el constructor pero sin usar las extensiones DevelopWithPassion.
  - Se descarta el uso de una DDD_Factory por la naturaleza del propio Legacy Code que nos impide asegurar que la DDD_Factory vaya a ser usada siempre, pese a ser una solución mucho más elegante y que además cumpliría el SRP.
- En las BDD_Specs definidas en el apartado 3.10, se vuelve a cambiar el paradigma del Interaction-Based Testing TDD por el State-Based Testing TDD y nuevamente comprobamos el comportamiento impecable del BDD ante este tipo de situaciones.
- En la BDD_Spec definida en el apartado 3.11, se muestra una forma indirecta de testar casos en los que resulta imposible mockar lo que necesitamos. Esta técnica es muy común en escenarios donde el BDD es utilizado para desarrollar Controllers de la terna MVC.
- En las BDD_Specs definidas en el apartado 3.13, nos enfrentamos ante el escenario más extremo de todo el Estudio. Un escenario con violación del SRP y OCP y posiblemente también del LSP. Ante esto, las BDD_Specs adquieren dimensiones aberrantes, indicativas de las distintas violaciones de los principios S.O.L.I.D.. Son las BDD_Specs más complejas y su valor radica, más allá de conseguir su objetivo, en ser un magnífico ejercicio por el que medir nuestro nivel de comprensión de los conceptos tratados en el Estudio.

Al final del capítulo podemos consultar el antes/después del Legacy Code, para comprobar de primera mano, cual ha sido el impacto de la aplicación del BDD en el mismo.

A partir de ahora, los cuatro capítulos restantes, se centrarán en resolver problemas concretos, en contraposición a los tres primeros capítulos, donde principalmente, se mostraba la Práctica de Diseño Agile BDD con profusión de ejemplos. Estos últimos cuatro capítulos van preparando el terreno para poder ir culminando el Estudio, por ello son más cortos y con un enfoque más claro.

Capítulo 4: Value Object RouteStatus

En este capítulo nos enfrentamos a la tarea de sustituir un objeto de tipo enum, por una clase, en concreto un DDD_Value_Object, siguiendo las Prácticas asociadas al BDD.

Esta discusión viene heredada del Capítulo 3, donde ya sugerimos una solución para conseguir este objetivo. De hecho, dejábamos la puerta abierta a algún tipo de mejora que nos permitiese diseñar este tipo de objetos usando BDD, como si de cualquier otro objeto se tratase, pese a tener que respectar las particularidades de dicho tipo de objetos.

Iniciamos el capítulo con:

- Una mirada en profundidad sobre los mecanismos de las extensiones DevelopWithPassion de Machine.Specifications que hacen que el SUT funcione.
- Las equivalencias en código de los conceptos asociados a Context / Specification y Assert, así como aclaramos la posible confusión entre Context y lo representado por el bloque Establish de Machine.Specifications.

Los aspectos más interesantes desde el punto de vista de las BDD_Specs son:

- En la BDD_Spec definida en el apartado 4.3, se toma, inicialmente, un camino erróneo a la hora de solucionar el problema de los enums y es la aplicación de

la propia metodología de la micro-iteración, la que descubre nuestro mal juicio y nos permite enmendarlo.

- En la BDD_Spec definida en el apartado 4.4, se plantea un escenario similar al descrito por la BDD_Spec del apartado 4.3, pero sin los errores iniciales de la misma.

Al final del capítulo, y como consecuencia de la implementación de otro objeto similar al tratado en este capítulo, detectamos un repetición de código, que nos permite refactorizar hacia una generalización del concepto repetido. De esta forma mejoramos ostensiblemente nuestro código. Al haber esperado a que esa repetición se produjese y no habernos anticipado a la misma, mostramos, nuevamente, como tratamos de evitar la generación de desperdicios (waste) siguiendo la filosofía Lean de una forma totalmente práctica.

Capítulo 5: Refactorizando RouteSpecificationFactory

En este capítulo refinamos el concepto de DDD_Factory, gracias al uso de interfaces y a la aplicación tanto del DDD_Ubiquitous_Language como al concepto, también asociado al DDD, de “revelar intención”.

Todo esto con un enfoque Lean, ya que, parte de estas refactorizaciones, podían haber sido realizadas antes. Sin embargo, si hubiésemos obrado así, hubiésemos generado desperdicio (waste). La mejor manera de actuar es, por tanto, esperar a que el propio código pida los cambios.

Se empaquetan todos los cambios en torno a una DDD_Factory a nivel DDD_Aggregate, de forma que pueda crear cualquier instancia abstracta que se encuentre en los límites de su influencia.

Esta serie de cambios nos facilita el escenario perfecto para poder introducir una aplicación práctica del principio DDD_Intention-Revealing_Interfaces y del ISP del S.O.L.I.D., que culmina con la refactorización final.

Capítulo 6: El Aggregate Root Cargo (Parte II)

La única razón para este capítulo es que habíamos dejado pendiente la implementación de Equals(), esperando una situación un poco más propicia.

Más allá de este hecho, lo único destacable es que se pone de manifiesto que cumplir con el DIP, y más en concreto con la Inyección de Dependencias, nos facilita la tarea de escribir BDD_Specs que prueben indirectamente lo que necesitamos.

Capítulo 7: Aggregate Root Location – Llevando el BDD y el DDD hasta sus últimas consecuencias

Con este capítulo culminamos el estudio y ya del propio título se desprende esa sensación de finalización.

Para poder contextualizar la discusión, comenzamos el capítulo con una descripción bastante detallada de los elementos DDD que van entrar en juego. Esto es una novedad, ya que hasta ahora, rara era la vez en la que describíamos el Domain Model con tal nivel de detalle, pues con la contextualización propuesta en las BDD_Specs, era más que suficiente para cumplir nuestros propósitos.

El desarrollo de las BDD_Specs y su implementación, apenas suponen un desafío, ya que, se repiten conceptos tratados ampliamente con anterioridad.

La colisión se va a producir en el momento en el que, para realizar un cálculo de un hash code, debemos delegar en uno de los objetos colaboradores inyectados a través del constructor en una DDD_Factory, pues dicho objeto no está definido por nosotros, sino que viene directamente definido por el Framework .NET.

Este problema con los tipos definidos por el Framework .NET ya había surgido en el contexto de las BDD_Specs definidas en el apartado 2.16. Por lo tanto, ya deberíamos tener la preparación suficiente para entender lo que se discute aquí.

Se hace una recopilación / enumeración de los síntomas detectados a medida que vamos implementando una BDD_Spec, e incluso se sigue adelante como si esos síntomas no se presentasen, hasta que el propio Machine.Specifications nos devuelve a la realidad con un error no esperado.

Ante la contingencia, se establecen dos soluciones diferentes:

- La primera solución viola el espíritu del BDD, pero cumple con lo pactado a través de las BDD_Specs (que se supone salen de una análisis de requisitos, y por tanto son un reflejo del Domain Model creado según las Prácticas de Diseño DDD).
- La segunda solución es un claro ejemplo del espíritu del BDD, pero dinamita (o refina) el diseño original DDD al añadir un nuevo DDD_Value_Object.

Con una valoración de las implicaciones de una u otra solución se finaliza el Estudio, a través del siguiente párrafo:

"(...) no existe un diseño original correcto. El diseño es algo que va evolucionando a cada instante. Por lo tanto, mientras los cambios que hagamos vengan motivados por las razones correctas seremos siempre partidarios de poner en juego de forma activa aquello de **Embrace Change**."

Apéndice I: Introducción al BDD con Machine.Specifications

Es la introducción perfecta al Estudio, si no se tienen conocimientos previos acerca de ninguna de las materias tratadas.

Se detalla desde el proceso de instalación, hasta la creación de las primeras BDD_Specs, tanto con Machine.Specifications como con sus extensiones DevelopWithPassion.

Apéndice II: Interfaces DDD

En este Apéndice se detallan los conceptos e interfaces detrás de:

- DDD_Entity (IEntity).
- DDD_Value_Object (IValueObject).
- DDD_Domain_Event (IDomainEvent).

Apéndice III: Implementando el Patrón DDD_Specification (Versión BDD)

Dependiendo de nuestros conocimientos previos, este apéndice puede ser visto de dos maneras completamente diferentes:

- Si tenemos ciertos conocimientos sobre la Práctica de Diseño DDD, podemos tomarlo a modo de introducción al BDD, ya que, al retorcer el patrón DDD_Specification, podemos hacernos una idea muy clara de la potencia del BDD como herramienta de diseño.
- Si no tenemos conocimientos previos sobre DDD, podemos tomarlo como un Epílogo al Estudio, de forma que comprobemos hasta que punto puede ser útil la combinación DDD+BDD. En este caso, puede servir también de recordatorio de las prácticas y conceptos asociados al BDD.

En este Apéndice se detalla el proceso que nos lleva a implementar el patrón DDD_Specification siguiendo la Práctica de Diseño Agile BDD:

- Se comenta la importancia de la DI/IoC (Dependency Injection / Inverse of Control) y su impacto en la testabilidad de un sistema.
- Se consigue eliminar completamente la necesidad de la clase base abstracta Specification<T>, que dificultaba la aplicación de la Práctica de Diseño Agile BDD.
- Se simplifica la interface ISpecification<T>, ganando en expresividad e intención, y sin perder por ello funcionalidad.
- Se aplica BDD y se aísla la implementación de las clases:
  - AndSpecification<T>.
  - OrSpecification<T>.
  - NotSpecification<T>.
- Se aplica tanto el SRP como el OCP al desarrollo.
- Se utilizan Extension Methods para implementar los métodos:
  - and<T>().
  - or<T>().
  - not<T>().

Apéndice IV: Primeros pasos con GIT

En este apéndice detallamos los procesos más comunes asociados a la utilización del Sistema de Control de Versiones Distribuido Git, incluyendo los pasos necesarios para la configuración de nuestro primer repositorio, así como la comunicación con un repositorio remoto alojado en github.

Finalmente se explican los pasos del Check-In Dance.

Capítulo 1: El Aggregate Root Cargo (Parte I)

1.1 Introducción

Uno de los puntos fuertes más evidentes del **BDD** (ya lo era también del TDD Mockista o Interaction-Based Testing TDD), es que nos permite desarrollar y testar código que a su vez depende de otras clases que ni siquiera hemos escrito.

Ésto nos va a permitir, por ejemplo, comenzar el desarrollo del **DDD_Aggregate** `Cargo`, por el objeto central (`Cargo`), que es el que debería tener mayor número de dependencias dentro del **DDD_Aggregate** al que pertenece (léase también como "el que necesita que los demás objetos estén escritos y funcionando").

...Y todo ésto con un sabor especial a **BDD**.

Vamos, por tanto, a aprovechar la oportunidad que nos brinda la implementación del **DDD_Aggregate_Root** `Cargo` para mostrar toda la potencia y flexibilidad del **BDD**. Intentaremos utilizar este capítulo como mero reflejo de la implementación real de la clase `Cargo`, para intentar alejarnos de los planteamientos más teóricos y sumergirnos directamente en la práctica.

Para ello se detallará y justificará cada una de las lineas de código que surjan a raíz de las **BDD_Specs** en el **DDD_Aggregate_Root** `Cargo`.

1.2 Preparando la BDD_Spec inicial

Empezamos por tanto con la clase `CargoSpecs.cs` que es donde encontraremos las **BDD_Specs** que nos permiten guiar el desarrollo de la futura clase `Cargo.cs` (**DDD_Aggregate_Root** del **DDD_Aggregate** `Cargo`).

En cuanto al bloque de referencias a `namespaces`, inicialmente debería bastarnos con:

```
using System;
using dddsample.domain.shared;
using Machine.Specifications;
using Machine.Specifications.DevelopWithPassion.Rhino;
using Rhino.Mocks;
```

Es importante señalar que ninguna de esas referencias han sido añadidas manualmente, puesto que el imprescindible plug-in **JetBrains ReSharper**, se encarga de ello de forma completamente transparente.

> **NOTA**: Omitimos el `namespace`, ya que tan solo dificultaría más la tarea de escribir el código en este documento por razones de reducción del espacio al aumentar la sangría.

Una vez aclarado este prerrequisito, necesario para que todo funcione correctamente, continuamos con los pasos previos a la implementación de la **BDD_Spec** en si:

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo> {}
```

A través de `Observes<Contract, ClassUnderTest>` dejamos claro que:

- Nuestro SUT (System Under Test) va a ser de tipo `ICargo`.
- Estas **BDD_Specs** dirigen el diseño e implementación de la clase `Cargo` que implementa la interface `ICargo`.
- Con la **clase abstracta** `concern_for_cargo`, proporcionamos a nuestras **BDD_Specs** de un contenedor donde poder alojar el código común a todas ellas.

Para conseguir que esa clase tan sencilla compile y deje de tener ese color rojo tan preocupante tanto en `ICargo` como en `Cargo`, necesitamos crear primeramente la interface `ICargo`:

```
public interface ICargo {}
```

Con ésto ya conseguimos que la referencia a `ICargo` en `concern_for_cargo` adquiera un color más saludable:

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo> {}
```

A continuación debemos generar la clase `Cargo` y obligarla a que implemente la interface `ICargo`:

```
public class Cargo : ICargo {}
```

De esa forma `concern_for_cargo` ya compilaría (nada de rojo):

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo> {}
```

en directo contraste con el inicial:

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo> {}
```

Pero no se acaba aquí la cosa, ya que, sabemos que `Cargo` es una **DDD_Entity**, por lo que debe implementar (lanzando una excepción que diga que no está implementado) a su vez la interface `IEntity<T>`, por lo que acabaríamos con algo así:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }
}
```

Ésto es todo lo que necesitamos para poder empezar con nuestra primera **BDD_Spec** para `Cargo`.

1.3 BDD_Spec - When asked about its tracking identificator

```
When asked about its tracking identificator
   It should provide the cargo id.
```

Una **BDD_Spec** tan simple como ésta, nos va a permitir generar las interfaces de gran parte de los colaboradores del **DDD_Aggregate_Root** `Cargo`.

1.3.1 Primera reacción en cadena

Empezamos por detallar la **BDD_Spec** y observaremos la **reacción en cadena** que se produce:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);
}
```

para eliminar el color rojo, basta con definir `result` y `tracking_id`:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId tracking_id;
    static ITrackingId result;
}
```

Ésto, a su vez, nos genera más código de color rojo, que se resuelve definiendo la interface `ITrackingId`:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId tracking_id;
    static ITrackingId result;
}

public interface ITrackingId {}
```

Pero no nos quedemos en los pequeños detalles y analicemos más concienzudamente que es lo que acabamos de presenciar.

Al vernos en la necesidad de conseguir que compile la **BDD_Spec** que estamos escribiendo (eliminar el código de color rojo), se ha producido, tal y como habíamos previsto, una **reacción en cadena**. Esta **reacción en cadena** es en realidad un proceso a través del cual vamos **diseñando** nuestro sistema y es uno de los ejes cardinales del **BDD**. Así es como se diseña. Creamos primero el código cliente para a continuación, en pasos **muy pequeños**, ir guiando nuestro diseño hacia donde queremos que vaya. Probablemente lo más asombroso de este proceso es que sin haber finalizado de escribir la **BDD_Spec**, ni haber intentado hacer que el código del **DDD_Aggregate_Root** `Cargo` cumpla con la **BDD_Spec** (pase el test), ya hemos avanzado notablemente en el diseño de nuestra aplicación.

Antes de proseguir, queremos resaltar que el código referente a todas estas clases e interfaces que se derivan de la **BDD_Spec**, se encuentra en el propio fichero de la **BDD_Spec**:

```
using System;
using dddsample.domain.model.cargo.aggregate;
using dddsample.domain.shared;
using Machine.Specifications;
using Machine.Specifications.DevelopWithPassion.Rhino;
using Rhino.Mocks;

namespace dddsample.specs.domain.model.cargo.aggregate
{
    public abstract class concern_for_cargo : Observes<ICargo, Cargo> {}

    public class when_asked_about_its_tracking_identificator :
                 concern_for_cargo
    {
        It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

        static ITrackingId tracking_id;
        static ITrackingId result;
    }

    public interface ICargo {}

    public class Cargo : ICargo, IEntity<Cargo>
    {
```

```
        public bool has_the_same_identity_as(Cargo the_other_entity)
        {
            throw new NotImplementedException();
        }
    }

    public interface ITrackingId {}
}
```

Ésta es una práctica habitual (que no obligatoria) en el **BDD** y en el TDD, ya que de esta forma, mientras vamos guiando, y por tanto, generando el código, mantenemos un control visual más directo de los elementos que entran en juego.

Cuando separemos las clase e interfaces y las enviemos a donde deben residir, lo especificaremos. Por el momento, y hasta que no se diga lo contrario, todo el código que creemos se encuentra en el fichero `CargoSpecs.cs`.

1.3.2 Ejercitando el SUT

Retomamos en este punto la **BDD_Spec**.

Para desencadenar el comportamiento que estamos creando, necesitamos invocar al SUT a través de un bloque `BECAUSE`:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    Because of = () => result = sut.tracking_id();

    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId tracking_id;
    static ITrackingId result;
}
```

De esta forma, definimos como pretendemos invocar al método del SUT que desencadena la funcionalidad, `tracking_id()`. Está en color rojo, porque todavía no lo hemos creado (otra cuestión importante, escribimos código en la **BDD_Spec** que invoca métodos que ni siquiera existen).

Para solucionarlo, necesitamos generar el método `tracking_id()` en la interface

`ICargo`:

```
public interface ICargo
{
    ITrackingId tracking_id();
}
```

De esta forma se ha ido el color rojo de nuestra **BDD_Spec**:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    Because of = () => result = sut.tracking_id();

    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId tracking_id;
    static ITrackingId result;
}
```

Pero ahora nos encontramos con que nuestro **DDD_Aggregate_Root** `Cargo` no implementa todos los métodos de `ICargo`, lo que nos obliga a implementar `tracking_id()` (lanzando una excepción que diga que no está implementado):

```
public class Cargo : ICargo, IEntity<Cargo>
{
    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }

    public ITrackingId tracking_id()
    {
        throw new NotImplementedException();
    }
}
```

Y aquí terminaría la **reacción en cadena** provocada por el bloque `BECAUSE`.

1.3.3 Estableciendo el contexto

Lo siguiente que necesitamos en nuestra **BDD_Spec**, es establecer el contexto necesario para que podamos siquiera ejecutarla. Para ello necesitamos un bloque `ESTABLISH`, que volverá a desencadenar una nueva **reacción en cadena**:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();
    };

    Because of = () => result = sut.tracking_id();

    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId tracking_id;
    static ITrackingId result;
}
```

Gracias a `the_dependency<T>()`, estamos estableciendo que `tracking_id` y `route_specification` son parámetros necesarios para construir el SUT (es decir el **DDD_Aggregate_Root** `Cargo`).

Para corregir el color rojo del bloque de código anterior, empezamos por generar la interface `IRouteSpecification`:

```
public interface IRouteSpecification {}
```

y creamos el campo (field) `route_specification` del tipo `IRouteSpecification`:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();
    };

    Because of = () => result = sut.tracking_id();

    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId tracking_id;
    static ITrackingId result;
    static IRouteSpecification route_specification;
}
```

Con ésto finalizamos la codificación de la **BDD_Spec**:

```
When asked about its tracking identificator
   It should provide the cargo id.
```

Es importante volver a resaltar como a través del proceso (y como consecuencia del mismo) por el que vamos escribiendo el código perteneciente a la **BDD_Spec**, vamos también diseñando y dotando de comportamiento a `ICargo`, `Cargo` e `ITrackingId`. Es decir, el proceso de diseño derivado de escribir la **BDD_Spec** influye directamente en el diseño de `ICargo`, `Cargo` e `ITrackingId`.

1.3.3.1 Consecuencias ocultas

La **reacción en cadena** anterior, todavía no se ha detenido, ya que, si definimos dos parámetros (`tracking_id`, `route_specification`) necesarios para crear el SUT, necesitaremos un constructor que los admita, y por el momento nuestro **DDD_Aggregate_Root** `Cargo`, no dispone de ninguno.

Por lo tanto necesitamos:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification){}

    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }

    public ITrackingId tracking_id()
    {
        throw new NotImplementedException();
    }
}
```

Una última consecuencia de la **reacción en cadena** sería la necesidad de considerar la naturaleza de `IRouteSpecification`, ya que tal y como la **DDD_Intention-Revealing_Interface** `IRouteSpecification` indica a través de la parte "Specification", parece claro que tratamos con una interface que debería a su vez implementar la interface `ISpecification<T>`, y de hecho éste es el caso.

Sin embargo, todavía no queda clara la naturaleza del Genérico `<T>`. Por ello, creemos que hasta que no sea completamente imprescindible, no haremos que `IRouteSpecification` implemente `ISpecification<T>`. Pero tomamos nota.

1.3.4 Ejecutando la BDD_Spec

Llegados a este punto, ejecutamos la **BDD_Spec**. Ésto es posible gracias a que poco a poco hemos ido preocupándonos de que ya "compilase" (léase "eliminar el código de color rojo").

En nuestro caso hemos decidido ejecutar la **BDD_Spec** usando **TestDriven.NET**, obteniendo el siguiente resultado:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id (FAIL)

Test 'should provide the cargo id' failed:
       System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.
       domain\model\cargo.aggregate\CargoSpecs.cs(49,0): at
dddsample.specs.domain.model.cargo.aggregate.Cargo.tracking_id()
       domain\model\cargo.aggregate\CargoSpecs.cs(20,0): at
dddsample.specs.domain.model.cargo.aggregate.when_asked_about_its_tracking
_identificator.<.ctor>b__1()
          --- End of inner exception stack trace ---
       at System.RuntimeMethodHandle._InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, SignatureStruct& sig,
MethodAttributes methodAttributes, RuntimeType typeOwner)
       at System.RuntimeMethodHandle.InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, Signature sig, MethodAttributes
methodAttributes, RuntimeType typeOwner)
       at System.Reflection.RuntimeMethodInfo.Invoke(Object obj,
BindingFlags invokeAttr, Binder binder, Object[] parameters, CultureInfo
culture, Boolean skipVisibilityChecks)
       at System.Delegate.DynamicInvokeImpl(Object[] args)
       at System.Delegate.DynamicInvoke(Object[] args)
       at
Machine.Specifications.Utility.RandomExtensionMethods.<InvokeAll>b__1(Dele
gate item)
       at Machine.Specifications.Utility.RandomExtensionMethods.Each[T]
```

```
(IEnumerable`1 enumerable, Action`1 action)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 actions)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 becauseActions)
      at Machine.Specifications.Model.Context.EstablishContext()

0 passed, 1 failed, 0 skipped, took 1,14 seconds (Machine.Specifications
0.3.0).
```

que confirma que ha compilado, pero que ha fallado (tal y como esperábamos, pues no hemos implementado nada que nos permita disponer de la funcionalidad sugerida por la **BDD_Spec**).

1.3.4.1 Analizando el fallo

Analicemos brevemente el log resultante de la ejecución de la **BDD_Spec**.

Sabemos que falla por tres cosas.

Primero:

```
when asked about its tracking identificator
» should provide the cargo id (FAIL)
```

Difícil no notar ese (**FAIL**).

Segundo:

```
Test 'should provide the cargo id' failed:
```

Sin comentarios.

Tercero:

```
0 passed, 1 failed, 0 skipped, took 1,14 seconds (Machine.Specifications
0.3.0).
```

Como en el caso anterior sobran los comentarios.

Todo el grueso del log, carece de relevancia para nosotros. Sin embargo ésto no siempre va a ser así, ya que habrá veces en las que detectaremos errores serios gracias a él. Por ejemplo, cuando tengamos la convicción de que todo debería ir bien y no vaya.

1.3.4.2 Corrigiendo el fallo

Vamos por tanto a escribir el código necesario para que ese **FAIL** desparezca.

Nos centramos ahora en la clase `Cargo`, en su constructor y en su método `tracking_id()`.

El código necesario para hacer que el test pase, no es complejo en absoluto. Sin embargo, al escribir solo el código necesario para que pase, si que es probable que se produzcan sorpresas, y no tanto por el código escrito, como por el no escrito.

La **BDD_Spec** nos está pidiendo que le devolvamos el identificador del **DDD_Aggregate_Root** `Cargo`. Con esto debería ser suficiente:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
    }

    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }

    public ITrackingId tracking_id()
    {
        return this.underlying_tracking_id;
    }
}
```

Comentar que, de forma innata, al crear el código, tendemos a seguir el flujo de ejecución.

Es decir, primero se produce la creación del objeto, en el constructor, por lo que escribimos:

```
public Cargo(ITrackingId tracking_id, IRouteSpecification route_specification)
{
    this.underlying_tracking_id = tracking_id;
}
```

y nos encontramos con nuestro viejo amigo el código de color rojo, ya que `underlying_tracking_id` no existe todavía.

Siguiendo el camino que nos marca el rojo (y en este caso además, el camino del flujo de ejecución), solucionamos este problema creando el campo `underlying_tracking_id`, que nos permite almacenar internamente el identificador:

```
ITrackingId underlying_tracking_id;
```

Una vez hecho esto, devolvemos el dato pedido:

```
public ITrackingId tracking_id()
{
    return this.underlying_tracking_id;
}
```

¿Ya está?.

No, hay algo que nos falta.

El identificador no es algo que queremos que se vaya modificando por ahí, sino que es algo que una vez asignado por el constructor, "es para toda la vida". Así que:

```
readonly ITrackingId underlying_tracking_id;
```

Y ahora sí que hemos terminado.

Antes de pasar a los comentario corremos la **BDD_Spec** para ver si con esto es suficiente:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id


1 passed, 0 failed, 0 skipped, took 0,88 seconds (Machine.Specifications
0.3.0).
```

Ha habido suerte. El test pasa.

Hemos completado nuestra primera **BDD_Spec** y el código que hace que todo funcione. Ya tenemos lista la primera característica de nuestro **DDD_Aggregate_Root** `Cargo`.

1.3.5 Comentarios

Vamos ahora con algunos comentarios sobre la **BDD_Spec** recién creada.

Debería llamarnos la atención que el constructor del **DDD_Aggregate_Root** `Cargo` reciba dos parámetros (`tracking_id` y `route_specification`) y solo use uno (`tracking_id`). Y lo más increíble de todo es que compila y funciona. Esto dice mucho de la potencia y flexibilidad del **BDD**, ya que permite que nos centremos en una sola cosa a la vez.

El método heredado de `IEntity<T>` `has_the_same_identity_as()` ni siquiera está implementado (devuelve una excepción). Nuevamente, esto es algo que no debe extrañarnos si practicamos **BDD**, ya que al utilizar y depender de interfaces en vez de clases concretas siempre que podemos, obtenemos este tipo de flexibilidad. Cuando decidamos que queremos crear una **BDD_Spec** sobre `has_the_same_identity_as()` ya lo implementaremos. Nosotros tenemos el control sobre qué se implementa y cuando se implementa.

Aún así, podríamos plantearnos la siguiente cuestión: ¿Como podemos afirmar que

hemos avanzado si esto parece un galimatías de interfaces absurdas que ni siquiera tienen nada dentro como es el caso de `ITrackingId` o `IRouteSpecification`?.

Esta reacción es común cuando nos encontramos dando los primeros pasos con el **BDD** (es exactamente lo mismo si practicamos TDD Mockista). No vamos a mentir, al principio cuesta llevar el ritmo y no perderse, pero con un poco de experiencia y algo de voluntad, esa sensación de sentirse abrumado por tener que:

- Crear la **BDD_Spec**.
- Diseñar.
- Manejar las interacciones entre objetos.
- Y finalmente implementar.

Desaparece. Y lo más curioso es que esa sensación acaba por invertirse. Así, cuando en el futuro nos encontremos con código fuente que no sigue el paradigma del BDD, o al menos disponga de una suite de tests bien diseñada, es cuando pasamos a sentirnos abrumados.

Antes de finalizar el ciclo de la primera **BDD_Spec**:

```
When asked about its tracking identificator
   It should provide the cargo id.
```

Vamos a realizar un par de refactorizaciones, más cosméticas que otra cosa, para que nos ayuden a poner todo lo que vamos escribiendo en su sitio.

1.3.6 Refactorizaciones

Es evidente que el lugar de `Cargo`, `ICargo` e `ITrackingId` no puede ser el fichero `CargoSpecs.cs` (ya habíamos comentado algo sobre el tema), y más concretamente el `namespace`:

```
namespace dddsample.specs.domain.model.cargo.aggregate
```

Ha llegado la hora de situar nuestras recién diseñadas clases e interfaces en el

`namespace` correcto:

```
namespace dddsample.domain.model.cargo.aggregate
```

Hacer este tipo de cosas teniendo **JetBrains ReSharper**, es trivial, ya que él mismo detecta estos cambios y busca la solución. En este caso, añadir un nuevo:

```
using dddsample.domain.model.cargo.aggregate;
```

a `CargoSpecs.cs`.

Si no tenemos instalado **JetBrains ReSharper**, puede ser que **Visual Studio 2010** se encargue de ello automáticamente. En último caso, siempre podemos escribir la linea anterior a mano.

Una vez arreglado esto, deberíamos volver a correr la **BDD_Spec**, por si acaso hemos roto algo.

Por suerte para nosotros

```
1 passed, 0 failed, 0 skipped, took 0,65 seconds (Machine.Specifications
0.3.0).
```

todo ha ido bien.

La siguiente refactorización afectaría a:

- `ICargo`
- `Cargo`
- `ITrackingId`
- `IRouteSpecification`

Y consistiría en crear un fichero independiente para cada uno de ellos, de forma que podamos colocarlos, de forma física, en el `namespace` correcto.

De nuevo, gracias a **JetBrains ReSharper**, es una tarea trivial.

Corremos los test de nuevo y comprobamos que seguimos con la luz verde.

1.3.7 Finalizando el proceso con git

Una vez llegado a este punto, deberíamos enviar los cambios a nuestro **repositorio** en **github**.

Para ello tendremos que ejecutar de forma secuencial los siguientes comandos:

```
git status
```

Nos indicaría cuales son los cambios.

```
git add -A
```

Añadiría los nuevos ficheros y carpetas al **git tracking**.

```
git commit -m "Finished first CargoSpec. Created ICargo, Cargo,
ITrackingId, and IRouteSpecification. Fixed error in IEntity<T>, now using
the correct one and not the experimental one."
```

Confirmaría los cambios realizados y además asociaría a dichos cambios el mensaje que se encuentra entre las comillas.

```
git checkout master
```

Nos movería a la **rama (branch) master**.

```
git merge dev
```

Pasaría los cambios realizados en la **rama dev** a la **rama master**.

```
git push origin
```

Enviaría los cambios a nuestra cuenta de **github**.

Y con esto, ya estaríamos listos.

1.3.8 El mantra RED-GREEN-REFACTOR

Ahora parece un buen momento para identificar el mantra del TDD "RED-GREEN-REFACTOR" en el proceso completo que acabamos de presenciar:

RED: Cada vez que escribimos una linea de código de la **BDD_Spec**, lo primero que hemos hecho ha sido corregir el que quedaba en color rojo, el código que no compilaba. Pero no es éste el RED del que habla el mantra, sino que más bien se refiere a ejecutar el test, la **BDD_Spec**, y hacer que falle.

Nosotros lo hicimos, si bien, apenas lo comentamos, para evitar despistarnos en el proceso de implementación de la **BDD_Spec** que estábamos intentando explicar.

Si vamos al log de cuando falló el test, en "todo el grueso del log" que "carece de relevancia para nosotros" (citamos), nos encontramos con:

```
Test 'should provide the cargo id' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.
```

Y más en concreto con:

```
System.NotImplementedException: The method or operation is not
implemented.
```

Ahí tenemos nuestro RED.

GREEN: Ésta es fácil, cuando pasamos el test con aquel:

```
1 passed, 0 failed, 0 skipped, took 0,88 seconds (Machine.Specifications
0.3.0).
```

REFACTOR: En nuestro caso ha sido mínimo. Hemos colocado las clases e interfaces creadas en su namespace correspondiente, una vez habíamos comprobado que el test pasaba. A continuación hemos creado un fichero para cada clase o interface que

hemos movido a donde se supone deben estar.

1.4 BDD_Spec - When asked about its route specification

Tal y como hicimos en la **BDD_Spec** anterior, empezamos por definir la **BDD_Spec** en texto:

```
When asked about its route specification
   It should provide the route specification.
```

Y luego en código:

```
public class when_asked_about_its_route_specification : concern_for_cargo
{
    It should_provide_the_route_specification = () =>
       result.ShouldEqual(route_specification);
}
```

Nuestro primer objetivo es arreglar el código de color rojo. Por lo tanto, basta con que definamos dos campos:

```
public class when_asked_about_its_route_specification : concern_for_cargo
{
    It should_provide_the_route_specification = () =>
       result.ShouldEqual(route_specification);

    static IRouteSpecification result;
    static IRouteSpecification route_specification;
}
```

Está claro cual es el objetivo, así que vamos con el bloque `BECAUSE`:

```
public class when_asked_about_its_route_specification : concern_for_cargo
{
    Because of = () => result = sut.route_specification();

    It should_provide_the_route_specification = () =>
       result.ShouldEqual(route_specification);

    static IRouteSpecification result;
    static IRouteSpecification route_specification;
}
```

Para poder arreglar el código de color rojo, necesitamos, primeramente, añadir un

nuevo método a la interface `ICargo`:

```
public interface ICargo
{
    ITrackingId tracking_id();
    IRouteSpecification route_specification();
}
```

Y a continuación implementar su esqueleto (devolviendo una excepción) en el **DDD_Aggregate_Root** `Cargo`:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
    }

    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }

    public ITrackingId tracking_id()
    {
        return this.underlying_tracking_id;
    }

    public IRouteSpecification route_specification()
    {
        throw new NotImplementedException();
    }
}
```

Hay una gran diferencia con respecto a la **BDD_Spec** anterior, y es que ahora, tanto `ICargo` como `Cargo` se encuentran es sus propios ficheros en su propio `namespace`. Por esa razón, la velocidad a la hora de saltar de un fichero a otro es crucial. De nuevo, **JetBrains ReSharper** hace esto posible.

1.4.1 Volando sobre el teclado con ReSharper

Vamos a detallar cual es el flujo de trabajo de esta última adición de código:

Partimos de:

```
public class when_asked_about_its_route_specification : concern_for_cargo
{
    Because of = () => result = sut.route_specification();

    (....)
}
```

Para situarnos al comienzo del código de color rojo, jugamos con:

```
ALT + F12: Nos lleva a "Next Error" (es el que usamos en este caso).
SHIFT ALT + F12: Nos lleva a "Previous Error".
```

hasta situarnos donde queremos (como solo hay un error en nuestro fichero, al probar con cualquiera de las dos opciones conseguimos lo que buscamos).

A continuación:

```
ALT + ENTER: Nos permite abrir las sugerencias ("View Action List").
```

Una vez abierta la "**View Action List**", nos encontramos con un menú desplegable con la siguiente opción ya preseleccionada:

```
"Create method ICargo.route_specification"
```

Pulsamos:

```
ENTER: Para confirmar la selección.
```

El **ReSharper** nos lleva entonces a la interface `ICargo` y ha escrito por nosotros:

```
IRouteSpecification route_specification();
```

permitiéndonos modificar el tipo devuelto por la interface.

Como es el correcto, pulsamos:

```
ENTER: Para confirmar el tipo devuelto.
```

A continuación si volvemos a abrir la "**View Action List**":

```
ALT + ENTER: Para abrir la "View Action List".
```

nos encontramos con una única sugerencia ya preseleccionada:

```
"Implement member in derived classes"
```

y eso es justo lo que queremos hacer, así que:

```
ENTER: Para confirmar la selección.
```

Esto nos llevaría a la clase `Cargo`, donde **ReSharper** escribe por nosotros:

```
public IRouteSpecification route_specification()
{
    throw new NotImplementedException();
}
```

Ahora necesitamos volver a `CargoSpecs`, y para ello tenemos dos opciones:

A:

```
CTRL + E: Abre el menú "Recent Files".
Pulsamos DOWN ARROW y ENTER para saltar a CargoSpecs.
```

B:

```
SHIFT + CTRL + N: Abre el menú "Enter file name".
Tecleamos las iniciales mayúsculas de CargoSpecs, es decir CS, y a
continuación ENTER para confirmar.
```

Ya estamos de vuelta y todo el código ha sido escrito por nosotros sin ni siquiera tener que tocar el ratón.

Vamos a recapitular, ya que este tipo de flujos son los que nos permitirán "volar" con

el teclado y programar a una velocidad adecuada.

```
ALT + F12
ALT + ENTER
ENTER
ENTER
ALT + ENTER
ENTER
```

Y para volver a CargoSpecs:

```
CTRL + E
DOWN ARROW
ENTER
```

Ya está. Que la magia del **ReSharper** vaya contigo.

Ahora completamente en serio. Sería imposible practicar **BDD** a una velocidad aceptable sin una herramienta como **ReSharper**. Es más, en la actualidad (en realidad desde hace un par de años), hay un movimiento bastante importante de gente que está empezando a usar un plug-in para **Visual Studio** llamado **ViEmu**...

...y sí, es un emulador de **Vim**.

La razón, no es otra, que poder navegar mejor por el código y el resultado es asombroso.

Sea como fuere, es de capital importancia, aprender este tipo de flujos que nos permiten ahorrar mucho tiempo evitando tareas tediosas y que debemos recalcar, se encuentran en las antípodas de las herramientas de **Code Generation**.

1.4.2 Estableciendo el contexto

Antes de ir con el contexto, no debemos perder de vista que el proceso que acabamos de realizar con el **ReSharper**, es una consecuencia de un acto de diseño derivado de la práctica del **BDD**.

Diseñamos cuando escribimos aún en color rojo:

```
sut.route_specification();
```

Y cuando implementamos ese método en su interface y en sus clases derivadas.

Una vez terminada la llamada al SUT y atendida la **reacción en cadena** provocada por ella, necesitamos establecer el contexto de nuestra **BDD_Spec** vía bloque `ESTABLISH`.

No debería sorprendernos el hecho de que éste sea exactamente el mismo que en la **BDD_Spec** anterior.

Simplemente definimos los dos argumentos que le pasamos al constructor del SUT:

```
public class when_asked_about_its_route_specification : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();
    };

    Because of = () => result = sut.route_specification();

    It should_provide_the_route_specification = () =>
       result.ShouldEqual(route_specification);

    static IRouteSpecification result;
    static IRouteSpecification route_specification;
}
```

Igual que la vez anterior, solucionamos el código de color rojo con un campo que lo defina.

Por lo tanto nuestra **BDD_Spec** final es:

```
public class when_asked_about_its_route_specification : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();
    };
```

```
    Because of = () => result = sut.route_specification();

    It should_provide_the_route_specification = () =>
       result.ShouldEqual(route_specification);

    static IRouteSpecification result;
    static IRouteSpecification route_specification;
    static ITrackingId tracking_id;
}
```

1.4.3 Implementando la funcionalidad necesaria

Ejecutamos la **BDD_Spec** a través de **TestDriven.NET**, para confirmar que efectivamente el test falla y nos da un error de tipo "No Implementado":

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification (FAIL)


Test 'should provide the route specification' failed:
       System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.
       domain\model\cargo.aggregate\Cargo.cs(27,0): at
dddsample.domain.model.cargo.aggregate.Cargo.route_specification()
       domain\model\cargo.aggregate\CargoSpecs.cs(36,0): at
dddsample.specs.domain.model.cargo.aggregate.when_asked_about_its_route_sp
ecification.<.ctor>b__1()
          --- End of inner exception stack trace ---
       at System.RuntimeMethodHandle._InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, SignatureStruct& sig,
MethodAttributes methodAttributes, RuntimeType typeOwner)
       at System.RuntimeMethodHandle.InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, Signature sig, MethodAttributes
methodAttributes, RuntimeType typeOwner)
       at System.Reflection.RuntimeMethodInfo.Invoke(Object obj,
BindingFlags invokeAttr, Binder binder, Object[] parameters, CultureInfo
culture, Boolean skipVisibilityChecks)
       at System.Delegate.DynamicInvokeImpl(Object[] args)
       at System.Delegate.DynamicInvoke(Object[] args)
       at
Machine.Specifications.Utility.RandomExtensionMethods.<InvokeAll>b__1(Dele
gate item)
```

```
      at Machine.Specifications.Utility.RandomExtensionMethods.Each[T]
(IEnumerable`1 enumerable, Action`1 action)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 actions)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 becauseActions)
      at Machine.Specifications.Model.Context.EstablishContext()

1 passed, 1 failed, 0 skipped, took 2,03 seconds (Machine.Specifications
0.3.0).
```

Vemos, por tanto, que obtenemos nuestro fallo.

Ahora necesitamos implementar el código del **DDD_Aggregate_Root** `Cargo` que nos permite pasar el test (y ojo, que no rompa el que ya funciona).

De manera equivalente a lo que hicimos para la anterior **BDD_Spec**:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
    }

    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }

    public ITrackingId tracking_id()
    {
        return this.underlying_tracking_id;
    }

    public IRouteSpecification route_specification()
    {
        return this.underlying_route_specification;
    }
}
```

Hemos completado el código del constructor.

Pero no cantemos victoria antes de tiempo. Necesitamos lanzar nuestra **BDD_Spec** para confirmar que el test pasa:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification


2 passed, 0 failed, 0 skipped, took 0,59 seconds (Machine.Specifications
0.3.0).
```

No ha habido sorpresas de última hora. Ya tenemos dos **BDD_Specs** implementadas.

Ya estamos listos para enviar nuestro código a **github**. Vamos a obviar esta parte ya que el proceso es siempre el mismo. En caso de duda podemos revisar la sección **Finalizando el proceso con git** de la **BDD_Spec** anterior.

1.5 BDD_Spec - When asked about its origin location

Esta **BDD_Spec** es más interesante, ya que vamos a tener dos bloques `IT`, además de otros sustos.

```
When asked about its origin location
   It should provide the origin location.
   It should leverage the route specification origin location.
```

1.5.1 Los bloques IT

El código correspondiente a los bloques `IT` de nuestra **BDD_Spec** es éste:

```
public class when_asked_about_its_origin_location : concern_for_cargo
{
    It should_provide_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    It should_leverage_the_route_specification_origin_location = () =>
       route_specification.received(x => x.origin());
}
```

Al solucionar el código rojo se produce la siguiente **reacción en cadena**:

```
public class when_asked_about_its_origin_location : concern_for_cargo
{
    It should_provide_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    It should_leverage_the_route_specification_origin_location = () =>
       route_specification.received(x => x.origin());

    static ILocation the_origin_location;
    static ILocation result;
    static IRouteSpecification route_specification;
}
```

Para poder solucionar la parte relativa a `x.origin()`, hemos tenido que hacer:

```
public interface IRouteSpecification
{
    ILocation origin();
```

```
}
```

Por lo tanto, el código grita la necesidad de:

```
public interface ILocation {}
```

que situamos inmediatamente en su fichero correspondiente y en su `namespace` adecuado:

```
namespace dddsample.domain.model.cargo.aggregate
{
    public interface ILocation {}
}
```

Con esto arreglamos todo el código rojo y finaliza la **reacción en cadena**.

1.5.2 El bloque BECAUSE

Para que las aserciones de los bloques `IT` se cumplan, necesitamos un bloque `BECAUSE` en el que ejercitemos el SUT:

```
public class when_asked_about_its_origin_location : concern_for_cargo
{
    Because of = () => result = sut.origin_location();

    It should_provide_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    It should_leverage_the_route_specification_origin_location = () =>
       route_specification.received(x => x.origin());

    static ILocation the_origin_location;
    static ILocation result;
    static IRouteSpecification route_specification;
}
```

Esto nos obliga a crear el método `origin_location()` tanto en la interface `ICargo` como en la clase `Cargo` (Ver en la **BDD_Spec** anterior el apartado del **ReSharper** para automatizar el proceso).

Éste es el resultado en `ICargo`:

```
public interface ICargo
{
    ITrackingId tracking_id();
    IRouteSpecification route_specification();
    ILocation origin_location();
}
```

y en `Cargo`:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
    }

    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }

    public ITrackingId tracking_id()
    {
        return this.underlying_tracking_id;
    }

    public IRouteSpecification route_specification()
    {
        return this.underlying_route_specification;
    }

    public ILocation origin_location()
    {
        throw new NotImplementedException();
    }
}
```

1.5.3 El bloque ESTABLISH

A la hora de establecer el contexto con el bloque `ESTABLISH` encontramos algunos añadidos (por fin vamos a usar Rhino.Mocks), como podemos apreciar en la **BDD_Spec**

definitiva:

```
public class when_asked_about_its_origin_location : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();

        the_origin_location = an<ILocation>();
        route_specification
            .Stub(x => x.origin())
            .Return(the_origin_location);
    };

    Because of = () => result = sut.origin_location();

    It should_provide_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    It should_leverage_the_route_specification_origin_location = () =>
       route_specification.received(x => x.origin());

    static ILocation the_origin_location;
    static ILocation result;
    static IRouteSpecification route_specification;
    static ITrackingId tracking_id;
}
```

Al igual que en las dos **BDD_Specs** anteriores, definimos los argumentos que serán inyectados al constructor del SUT. Pero además usamos `route_specification` como un Mock al establecer que:

- Esperamos una llamada al método `origin()` y...
- ...cuando ésta se produzca debemos devolver el recién creado FAKE `the_origin_location`:

En código:

```
route_specification
    .Stub(x => x.origin())
    .Return(the_origin_location);
```

Donde:

```
the_origin_location = an<ILocation>();
```

1.5.3.1 Desmenuzando el bloque IT

Ahora es el momento de volver a hablar de:

```
It should_leverage_the_route_specification_origin_location = () =>
   route_specification.received(x => x.origin());
```

Aquí estamos estableciendo como condición de que "should leverage the route specification origin location" (algo así como "debería delegar en `route_specification.origin()`"), que se produzca esa llamada:

```
route_specification.received(x => x.origin());
```

1.5.4 Pasando los tests

Corremos los dos tests.

Deberían fallar ya que no están implementados:

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   when asked about its tracking identificator
   » should provide the cargo id

   when asked about its route specification
   » should provide the route specification

   when asked about its origin location
   » should provide the origin location (FAIL)
   » should leverage the route specification origin location (FAIL)


   Test 'should provide the origin location' failed:
          System.Reflection.TargetInvocationException: Exception has been
   thrown by the target of an invocation. --->
   System.NotImplementedException: The method or operation is not
   implemented.
          domain\model\cargo.aggregate\Cargo.cs(34,0): at
   dddsample.domain.model.cargo.aggregate.Cargo.origin_location()
          domain\model\cargo.aggregate\CargoSpecs.cs(57,0): at
```

```
dddsample.specs.domain.model.cargo.aggregate.when_asked_about_its_origin_l
ocation.<.ctor>b__2()
         --- End of inner exception stack trace ---
      at System.RuntimeMethodHandle._InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, SignatureStruct& sig,
MethodAttributes methodAttributes, RuntimeType typeOwner)
      at System.RuntimeMethodHandle.InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, Signature sig, MethodAttributes
methodAttributes, RuntimeType typeOwner)
      at System.Reflection.RuntimeMethodInfo.Invoke(Object obj,
BindingFlags invokeAttr, Binder binder, Object[] parameters, CultureInfo
culture, Boolean skipVisibilityChecks)
      at System.Delegate.DynamicInvokeImpl(Object[] args)
      at System.Delegate.DynamicInvoke(Object[] args)
      at
Machine.Specifications.Utility.RandomExtensionMethods.<InvokeAll>b__1(Dele
gate item)
      at Machine.Specifications.Utility.RandomExtensionMethods.Each[T]
(IEnumerable`1 enumerable, Action`1 action)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 actions)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 becauseActions)
      at Machine.Specifications.Model.Context.EstablishContext()

Test 'should leverage the route specification origin location' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.
      domain\model\cargo.aggregate\Cargo.cs(34,0): at
dddsample.domain.model.cargo.aggregate.Cargo.origin_location()
      domain\model\cargo.aggregate\CargoSpecs.cs(57,0): at
dddsample.specs.domain.model.cargo.aggregate.when_asked_about_its_origin_l
ocation.<.ctor>b__2()
         --- End of inner exception stack trace ---
      at System.RuntimeMethodHandle._InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, SignatureStruct& sig,
MethodAttributes methodAttributes, RuntimeType typeOwner)
      at System.RuntimeMethodHandle.InvokeMethodFast(IRuntimeMethodInfo
method, Object target, Object[] arguments, Signature sig, MethodAttributes
methodAttributes, RuntimeType typeOwner)
      at System.Reflection.RuntimeMethodInfo.Invoke(Object obj,
BindingFlags invokeAttr, Binder binder, Object[] parameters, CultureInfo
culture, Boolean skipVisibilityChecks)
      at System.Delegate.DynamicInvokeImpl(Object[] args)
      at System.Delegate.DynamicInvoke(Object[] args)
      at
Machine.Specifications.Utility.RandomExtensionMethods.<InvokeAll>b__1(Dele
gate item)
```

```
      at Machine.Specifications.Utility.RandomExtensionMethods.Each[T]
(IEnumerable`1 enumerable, Action`1 action)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 actions)
      at
Machine.Specifications.Utility.RandomExtensionMethods.InvokeAll(IEnumerabl
e`1 becauseActions)
      at Machine.Specifications.Model.Context.EstablishContext()

2 passed, 2 failed, 0 skipped, took 2,22 seconds (Machine.Specifications
0.3.0).
```

1.5.4.1 El segundo bloque IT

Vamos primero a pasar el segundo bloque `IT`:

```
It should_leverage_the_route_specification_origin_location = () =>
   route_specification.received(x => x.origin());
```

Para que pase necesitamos que ocurra lo siguiente (hemos eliminado la parte del código que no se refiere a esta **BDD_Spec** y solo añade ruido):

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
        route_specification.origin();
    }

    (....)

    public ILocation origin_location()
    {
        throw new NotImplementedException();
    }
}
```

La parte que hace que el test pase es la del constructor:

```
route_specification.origin();
```

Con eso solo, debería pasar, pero no lo hace, ya que nos sigue dando el mismo error:

```
Test 'should leverage the route specification origin location' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.
```

Y lo curioso es que es cierto. Aunque el requisito se cumple, el método invocado en el bloque `BECAUSE`:

```
Because of = () => result = sut.origin_location();
```

sigue lanzando una excepción:

```
public ILocation origin_location()
{
    throw new NotImplementedException();
}
```

¿Y como podemos comprobar que esto que decimos es verdad?

Muy fácil, necesitamos que `origin_location()` devuelva un valor válido y no una excepción. Pero no tiene porque ser EL valor que buscamos:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;
    readonly ILocation any_origin_location;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
        route_specification.origin();
    }

    (....)

    public ILocation origin_location()
```

```
    {
        return this.any_origin_location;
    }
}
```

Es decir, creamos un `ILocation` llamado `any_origin_location`, al que no le asignamos valor alguno, simplemente nos da los medios para "engañar" al método `origin_location()` y proveerle a la salida un tipo valido, evitando tener que lanzar una excepción:

```
public ILocation origin_location()
{
    return this.any_origin_location;
}
```

Es importante fijarse que en el constructor **NO ASIGNAMOS** el valor devuelto por `route_specification.origin()`:

```
public Cargo(ITrackingId tracking_id, IRouteSpecification route_specification)
{
    (....)
    route_specification.origin();
}
```

Corremos el test ahora y obtenemos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location (FAIL)
» should leverage the route specification origin location


Test 'should provide the origin location' failed:
       Machine.Specifications.SpecificationException: Should equal
ILocationProxy81c2704787ff43dc8a8ab67018e80010 but is [null]
       at Machine.Specifications.ShouldExtensionMethods.ShouldEqual[T](T
actual, T expected)
       domain\model\cargo.aggregate\CargoSpecs.cs(60,0): at
dddsample.specs.domain.model.cargo.aggregate.when_asked_about_its_origin_l
```

```
ocation.<.ctor>b__3()
       at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
       at Machine.Specifications.Model.Specification.Verify()

3 passed, 1 failed, 0 skipped, took 0,85 seconds (Machine.Specifications
0.3.0).
```

Así que podemos dar por zanjada y probada la cuestión que planteábamos anteriormente.

1.5.4.2 El primer bloque IT

Vamos ahora con el test que falla, que corresponde al bloque `IT`:

```
It should_provide_the_origin_location = () =>
   result.ShouldEqual(the_origin_location);
```

y que a estas alturas ya deberíamos estar hartos de ver, ya que es equivalente a los de las dos **BDD_Specs** anteriores.

Lo realmente interesante es ver la razón por la que falla:

```
Test 'should provide the origin location' failed:
      Machine.Specifications.SpecificationException: Should equal
ILocationProxy81c2704787ff43dc8a8ab67018e80010 but is [null]
```

Ya no es la consabida:

```
The method or operation is not implemented.
```

Ahora nos dice que esperaba un valor concreto (no hay que asustarse por el valor en si, ya que, es el resultado de como funciona internamente Rhino.Mocks):

```
ILocationProxy81c2704787ff43dc8a8ab67018e80010
```

Pero recibe otro distinto:

```
[null]
```

Y eso es justo lo que cabría esperar, ya que recordemos que estamos "engañando" al método `origin_location()` enviándole el tipo correcto, pero en realidad le estamos asignando `null` y eso, precisamente, es lo que nos dice el error del test:

> "*Hay algo que va mal, ya que, esperaba un valor concreto, tan concreto como esta monstruosidad, ILocationProxy81c2704787ff43dc8a8ab67018e80010, pero en realidad me estás enviando un NULL.*"

El problema ya sabemos donde está, de hecho es tan sencillo hacer pasar el test como:

```
public Cargo(ITrackingId tracking_id, IRouteSpecification route_specification)
{
    (....)
    any_origin_location = route_specification.origin();
}
```

Ya lo hemos hecho en las **BDD_Specs** anteriores.

Lo importante aquí es suponer lo siguiente. Imaginemos que hemos escrito este código:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;
    readonly ILocation underlying_origin_location;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
        route_specification.origin();
    }

    (....)

    public ILocation origin_location()
    {
        return this.underlying_origin_location;
    }
}
```

Si hubiésemos escrito ese código, sí que tiene toda la pinta de que estamos cometiendo un error que no deseábamos que ocurriese, ya que, aunque el código sea el mismo que cuando engañábamos al método `route_specification()`, ahora el nombre que le damos al campo encargado de guardar el punto de origen (`underlying_origin_location`) revela la intención de que QUERÍAMOS haberle asignado ese valor y no devolver un NULL.

Lo bueno de todo esto es que "estamos cubiertos". Es decir, si cometemos algún error, por muy tonto que sea, vamos a "estar cubiertos" y en algún punto fallará nuestra suite de **BDD_Specs**.

Lo que probablemente estábamos intentando escribir era (atención a la diferencia en el constructor):

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;
    readonly ILocation underlying_origin_location;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
        this.underlying_origin_location = route_specification.origin();
    }

    (....)

    public ILocation origin_location()
    {
        return this.underlying_origin_location;
    }
}
```

que sí pasa el test:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id
```

```
  when asked about its route specification
  » should provide the route specification

  when asked about its origin location
  » should provide the origin location
  » should leverage the route specification origin location


  4 passed, 0 failed, 0 skipped, took 0,92 seconds (Machine.Specifications
  0.3.0).
```

1.5.5 Finalizando la BDD_Spec

Nuestra clase `Cargo` (nuestro **DDD_Aggregate_Root** `Cargo`) quedaría así, por lo tanto:

```
public class Cargo : ICargo, IEntity<Cargo>
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;
    readonly ILocation underlying_origin_location;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
        this.underlying_origin_location = route_specification.origin();
    }

    public bool has_the_same_identity_as(Cargo the_other_entity)
    {
        throw new NotImplementedException();
    }

    public ITrackingId tracking_id()
    {
        return this.underlying_tracking_id;
    }

    public IRouteSpecification route_specification()
    {
        return this.underlying_route_specification;
    }

    public ILocation origin_location()
    {
        return this.underlying_origin_location;
    }
```

```
}
```

Estamos listos para empaquetarlo todo y mandarlo a **github**.

1.6 BDD_Spec - When comparing the cargo identity

Parece un buen momento para eliminar ese:

```
throw new NotImplementedException();
```

del método:

```
public bool has_the_same_identity_as(Cargo the_other_entity)
```

Así que vamos con esa **BDD_Spec**:

> *When comparing the cargo identity*
>
> *It should leverage the tracking identity collaborator.*

1.6.1 El bloque IT

La **BDD_Spec** en código, con su **bloque** IT, sería ésta:

```
public class when_comparing_the_cargo_identity : concern_for_cargo
{
    It should_leverage_the_tracking_identity_collaborator = () =>
       tracking_id.received(x => x
                  .has_the_same_value_as(the_to_compare_cargo.tracking_id()));
}
```

Procedamos a arreglar el código de color rojo.

El caso de `tracking_id` y `the_to_compare_cargo` es trivial, ya lo hemos visto varias veces:

```
public class when_comparing_the_cargo_identity : concern_for_cargo
{
    It should_leverage_the_tracking_identity_collaborator = () =>
       tracking_id.received(x => x
                  .has_the_same_value_as(the_to_compare_cargo.tracking_id()));

    static ITrackingId tracking_id;
    static ICargo the_to_compare_cargo;
}
```

Pero el caso de `has_the_same_value_as()` presenta una **reacción en cadena** muy interesante.

1.6.1.1 Delegando y aplicando el SRP

Para poder entender la **reacción en cadena**, debemos, primero, entender donde se produce esa llamada.

La intención de lo que pretendemos hacer es clara. Vamos a aplicar el **SRP**, de forma que, la lógica (léase también como "el algoritmo") que va a hacer que nuestro **DDD_Aggregate_Root** `Cargo` pueda evaluar su identidad con respecto a otro **DDD_Aggregate_Root** `Cargo` no va a estar, como cabría suponer, en su método `has_the_same_identity_as()`, sino que, tal y como especificamos en el bloque `IT`, va a delegar ese calculo, esa implementación del algoritmo, a un objeto del tipo `ITrackingId` a través de su método `has_the_same_value_as()`. En concreto a la instancia `tracking_id`. Así que el `has_the_same_value_as()` de color rojo en el código anterior corresponde a `tracking_id.has_the_same_value_as()`, que no debe ser confundido con el `has_the_same_identity_as()` del **DDD_Aggregate_Root** `Cargo`.

La lógica que hay detrás de esto hay que atribuírsela tanto al **SRP** como al proceso de diseño continuo que nos facilita el **BDD**. Y es que no nos cansaremos de decirlo: **diseño**, **diseño**, **diseño**. Estamos diseñando. En cualquier fase del desarrollo, diseñamos.

¿Cual es el problema entonces?

El problema es que `ITrackingId` no dispone todavía de ese método, y lo más importante, ¿tiene sentido crear ese método en `ITrackingId`?.

La respuesta inmediata a esta pregunta es "NO", ya que, todavía no hemos decidido que va a ser `ITrackingId` "de mayor".

1.6.1.2 ITrackingId "cuando crezca"

Ahora es el momento de tomar esa decisión. De asignar una responsabilidad a `ITrackingId`.

> **NOTA**: Quizás esto nos suene a una mini-discusión en la primera **BDD_Spec** de `CargoSpecs` en la que creábamos la interface `IRouteSpecification` y "quedaba anotado" (esa fue la expresión que usamos) que parecía que `IRouteSpecification` iba a necesitar implementar `ISpecification<T>`, pero aplazábamos la decisión hasta que fuese estrictamente necesario.
>
> Pues efectivamente, sería el mismo problema. Pero en el caso de `ITrackingId`, ahora, es estrictamente necesario.

`ITrackingId` va a ser un **DDD_Value_Object**. Por lo tanto va a tener que implementar `IValueObject<T>`. Y esa es la clave para responder a la pregunta anterior de si tiene o no sentido crear el método `has_the_same_value_as()` en `ITrackingId`.

No tiene sentido "crearlo" en `ITrackingId`, ya que al implementar `IValueObject<T>`, ese método ya le viene heredado.

Por lo tanto para corregir el código de color rojo basta con:

```
public interface ITrackingId : IValueObject<ITrackingId> {}
```

1.6.2 Ejercitando el SUT

Podemos continuar con la **BDD_Spec** ejercitando el SUT. Puesto que ya somos veteranos en esto, incluimos también el campo `result` para centrarnos en lo verdaderamente importante:

```
public class when_comparing_the_cargo_identity : concern_for_cargo
{
    Because of = () =>
       result = sut.has_the_same_identity_as(the_to_compare_cargo);
```

```
    It should_leverage_the_tracking_identity_collaborator = () =>
       tracking_id.received(x => x
                   .has_the_same_value_as(the_to_compare_cargo.tracking_id()));

    static ITrackingId tracking_id;
    static ICargo the_to_compare_cargo;
    static bool result;
}
```

Y cual es nuestra sorpresa, el SUT no puede acceder a `has_the_same_identity_as()`, tal y como marca el código rojo:

```
sut.has_the_same_identity_as(the_to_compare_cargo);
```

No puede ser, nuestro SUT (es decir `Cargo`), implementa `IEntity<T>`:

```
public interface IEntity<T>
{
    bool has_the_same_identity_as(T the_other_entity);
}
```

por lo tanto tiene acceso a `has_the_same_identity_as()`, ...¿o no?

1.6.2.1 Interfaces vs Clases

Hay un pequeño error en ese razonamiento y no tiene que ver directamente con el código rojo.

El tipo del SUT viene determinado a través de `Observes<Contract, ClassUnderTest>` .Y en concreto por la parte de `Contract` y no por la parte de `ClassUnderTest`. Por lo tanto, teniendo en cuenta que:

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo> {}
```

concluimos que el SUT no es de tipo `Cargo`:

```
public class Cargo : ICargo, IEntity<Cargo> {(....)}
```

sino de tipo `ICargo`:

```
public interface ICargo {(....)}
```

Y la confusión (y el código rojo) viene de que es `Cargo` quien implementa `IEntity<Cargo>` por lo que es `Cargo` el que accede a `has_the_same_identity_as()` mientras que `ICargo` no tiene esa posibilidad. Ergo, el código rojo:

```
sut.has_the_same_identity_as(the_to_compare_cargo);
```

Esto es la constatación de un error de diseño. Y gracias a la práctica del **BDD** nos estamos enfrentando a él.

Vamos con la **reacción en cadena**.

Si nos fijamos en lo que acabamos de hacer con `ITrackingId`, podemos intuir la solución:

```
public interface ITrackingId : IValueObject<ITrackingId> {}
```

Efectivamente, vamos a hacer que `ICargo` implemente `IEntity<T>` (antes era `Cargo` quien lo implementaba), donde `T` ahora va a ser `ICargo`. Es decir:

```
public interface ICargo : IEntity<ICargo> {(....)}
```

frente a:

```
public class Cargo : ICargo, IEntity<Cargo> {(....)}
```

Por lo tanto, `Cargo` se convierte en:

```
public class Cargo : ICargo {(....)}
```

De esta forma, nuestro **DDD_Aggregate_Root** `Cargo`, ahora, implementa únicamente `ICargo`, mientras que `ICargo` es el encargado de implementar `IEntity<ICargo>`. Por lo tanto, `Cargo` sigue implementando `IEntity`, aunque en este caso, lo hace a través de `ICargo` y no directamente.

Pero la **reacción en cadena** no acaba ahí, ya que nuestro:

```
public bool has_the_same_identity_as(Cargo the_other_entity)
{
    throw new NotImplementedException();
}
```

ya no cumple con la signatura de `IEntity<ICargo>`. Debería recibir un `ICargo` como argumento en vez de un `Cargo`. Por lo tanto:

```
public bool has_the_same_identity_as(ICargo the_other_entity)
{
    throw new NotImplementedException();
}
```

Y ahora viene la magia de las interfaces y del BDD al unísono y es que es lícito creer que:

> "*Estos cambios han debido corromper alguna de las funcionalidades ya programadas.*"

Bueno, esa es una de las razones por las que estamos tomándonos la molestia de construir nuestra suite de **BDD_Specs**. Para poder responder a eso de una forma automatizada y relativamente segura.

Vamos a correr los tests y corregiremos lo que esté roto. Pero cual va a ser nuestra sorpresa:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
» should leverage the route specification origin location

when comparing the cargo identity
» should leverage the tracking identity collaborator (FAIL)
```

```
Test 'should leverage the tracking identity collaborator' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

4 passed, 1 failed, 0 skipped, took 0,99 seconds (Machine.Specifications
0.3.0).
```

Las tres **BDD_Specs** anteriores (con sus cuatro tests asociados) no fallan. El único que falla es el que estamos implementando, y es algo esperado, ya que para empezar ni siquiera lo hemos terminado. De hecho, es un milagro que simplemente compile (la buena práctica de ir siempre corrigiendo el código de color rojo parece que empieza a dar sus frutos).

Así que, para no dispersarnos, los cambio tan radicales de diseño que acabamos de hacer **no afectan** a la funcionalidad que tenemos programada.

No queremos pecar de optimistas, pero parece que estamos haciendo bien las cosas.

Este cambio en el diseño, de por sí, justificaría un **commit** en **git**.

1.6.3 Estableciendo el contexto

Continuamos con nuestra **BDD_Spec**.

Pasamos a establecer el contexto con el bloque `ESTABLISH`, y no nos encontramos nada nuevo, así que procedemos con el código de la **BDD_Spec** completo:

```
public class when_comparing_the_cargo_identity : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();

        the_to_compare_cargo = an<ICargo>();
```

```
        tracking_id
           .Stub(x => x.has_the_same_value_as(
                        the_to_compare_cargo.tracking_id()))
           .Return(true);
    };

    Because of = () =>
       result = sut.has_the_same_identity_as(the_to_compare_cargo);

    It should_leverage_the_tracking_identity_collaborator = () =>
       tracking_id.received(x => x
                   .has_the_same_value_as(the_to_compare_cargo.tracking_id()));

    static ITrackingId tracking_id;
    static ICargo the_to_compare_cargo;
    static bool result;
    static IRouteSpecification route_specification;
}
```

Lo único reseñable sería:

```
tracking_id
   .Stub(x => x.has_the_same_value_as(the_to_compare_cargo.tracking_id()))
   .Return(true);
```

Estamos diciendo que esperamos una llamada a `tracking_id.has_the_same_value_as()` con el argumento `the_to_compare_cargo.tracking_id()` y le pedimos que devuelva `true`.

1.6.4 Ejecutando la suite de BDD_Specs

Corremos el test y obtenemos nuestro error:

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   when asked about its tracking identificator
   » should provide the cargo id

   when asked about its route specification
   » should provide the route specification

   when asked about its origin location
   » should provide the origin location
   » should leverage the route specification origin location

   when comparing the cargo identity
```

```
  » should leverage the tracking identity collaborator (FAIL)


  Test 'should leverage the tracking identity collaborator' failed:
        System.Reflection.TargetInvocationException: Exception has been
  thrown by the target of an invocation. --->
  System.NotImplementedException: The method or operation is not
  implemented.

        (....)

  4 passed, 1 failed, 0 skipped, took 0,80 seconds (Machine.Specifications
  0.3.0).
```

Ya estamos listos para implementar la funcionalidad.

Si hemos seguido con atención el desarrollo de las **BDD_Specs** anteriores deberíamos entender que con esto:

```
public class Cargo : ICargo
{
    readonly ITrackingId underlying_tracking_id;
    readonly IRouteSpecification underlying_route_specification;
    readonly ILocation underlying_origin_location;

    public Cargo(ITrackingId tracking_id,
                 IRouteSpecification route_specification)
    {
        this.underlying_tracking_id = tracking_id;
        this.underlying_route_specification = route_specification;
        this.underlying_origin_location = route_specification.origin();
    }

    public bool has_the_same_identity_as(ICargo the_other_entity)
    {
        underlying_tracking_id
            .has_the_same_value_as(the_other_entity.tracking_id());
        return false;
    }

    (....)
}
```

debería bastar para pasar el test.

Más en concreto con:

```
public bool has_the_same_identity_as(ICargo the_other_entity)
{
    underlying_tracking_id
       .has_the_same_value_as(the_other_entity.tracking_id());
    return false;
}
```

Así que antes de hacer más comentarios, vamos a correr los tests:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
» should leverage the route specification origin location

when comparing the cargo identity
» should leverage the tracking identity collaborator


5 passed, 0 failed, 0 skipped, took 0,79 seconds (Machine.Specifications
0.3.0).
```

Efectivamente, todo va a pedir de boca.

... ¿o no?.

1.6.4.1 Consideraciones

Debemos hacer varias consideraciones:

- El test funciona, y la funcionalidad asociada al test hace lo que tiene que hacer (delega el cálculo de la igualdad al objeto que queremos (**SRP**)).
- Si bien es cierto el punto anterior, no es menos cierto que, no tenemos en cuenta para nada el resultado del cálculo.
- Claramente nos falta una aserción y probablemente tengamos que redefinir el contexto.

1.6.5 Redefiniendo el contexto

Vamos a ponernos con este último punto.

Parece que esto tiene más sentido:

```
When comparing two equal cargoes
   It should leverage the tracking identity collaborator.
```

que no el antiguo:

```
When comparing the cargo identity
   It should leverage the tracking identity collaborator.
```

ya que si atendemos al contexto que hemos definido en el bloque `ESTABLISH`:

```
tracking_id
   .Stub(x => x.has_the_same_value_as(the_to_compare_cargo.tracking_id()))
   .Return(true);
```

con ese:

```
.Return(true);
```

estamos simulando el comportamiento de dos `ITrackingIds` que tienen el mismo valor.

Ahora bien, parece que todavía tendría más sentido si añadimos una nueva aserción:

```
When comparing two equal cargoes
   It should confirm that they have the same identity
   It should leverage the tracking identity collaborator.
```

que en código sería:

```
It should_confirm_that_they_have_the_same_identity = () =>
   result.ShouldBeTrue();
```

Pero debemos recordar que en nuestra implementación hemos obligado a `has_the_same_identity_as()` a devolver siempre `false`:

```
public bool has_the_same_identity_as(ICargo the_other_entity)
{
    underlying_tracking_id
       .has_the_same_value_as(the_other_entity.tracking_id());
    return false;
}
```

Por lo que tanto, al ejecutar la suite de **BDD_Specs**, obtenemos un **FAIL** como un piano:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
» should leverage the route specification origin location

when comparing two equal cargoes
» should confirm that they have the same identity (FAIL)
» should leverage the tracking identity collaborator


Test 'should confirm that they have the same identity' failed:
      Machine.Specifications.SpecificationException: Should be [true] but
is [false]
      at
Machine.Specifications.ShouldExtensionMethods.ShouldBeTrue(Boolean
condition)
      domain\model\cargo.aggregate\CargoSpecs.cs(94,0): at
dddsample.specs.domain.model.cargo.aggregate.when_comparing_two_equal_carg
oes.<.ctor>b__3()
      at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
      at Machine.Specifications.Model.Specification.Verify()

5 passed, 1 failed, 0 skipped, took 0,90 seconds (Machine.Specifications
0.3.0).
```

Si nos fijamos en la razón concreta por la que falla:

```
Test 'should confirm that they have the same identity' failed:
      Machine.Specifications.SpecificationException: Should be [true] but
is [false]
```

confirmamos nuestro razonamiento.

Vamos a arreglarlo, y para ello vamos a intentar, no solo "*escribir únicamente el mínimo código necesario para que el test pase*" como reza el conocido principio del TDD:

```
public bool has_the_same_identity_as(ICargo the_other_entity)
{
    underlying_tracking_id
       .has_the_same_value_as(the_other_entity.tracking_id());
    return true;
}
```

Sino que vamos a intentar "*escribir únicamente el mínimo código necesario Y QUE ADEMÁS, TENGA SENTIDO, para que el test pase*", frase atribuida a **Jean-Paul Boodhoo**.

En nuestro caso sería:

```
public bool has_the_same_identity_as(ICargo the_other_entity)
{
    return underlying_tracking_id
              .has_the_same_value_as(the_other_entity.tracking_id());
}
```

y que además debería satisfacer nuestra última aserción.

Comprobamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
```

```
» should leverage the route specification origin location

when comparing two equal cargoes
» should confirm that they have the same identity
» should leverage the tracking identity collaborator


6 passed, 0 failed, 0 skipped, took 0,74 seconds (Machine.Specifications
0.3.0).
```

Efectivamente, todo funciona como esperábamos

1.6.5.1 Comprobando la condición complementaria

Una cuestión que podríamos plantearnos es que, ya que tenemos:

```
When comparing two equal cargoes
   It should confirm that they have the same identity.
```

¿no deberíamos comprobar también esto otro?:

```
When comparing two different cargoes
   It should confirm that they have different identity.
```

Desde luego es una cuestión lícita, ya que, siguiendo nuestro propio razonamiento, el caso:

```
When comparing two equal cargoes
```

sería nuestro Happy Day Scenario, mientras que:

```
When comparing two different cargoes
```

sería otro escenario diferente.

En este caso es difícil dar una respuesta contundente.

1.6.5.2 TDD Tradicional vs TDD Mockista

Supongamos que en vez de:

```
When comparing two equal cargoes
   It should confirm that they have the same identity.
   It should leverage the tracking identity collaborator.
```

tuviésemos esto otro:

```
When comparing two equal cargoes
   It should confirm that they have the same identity.
```

Entonces, la respuesta si que es contundente. Es necesario implementar:

```
When comparing two different cargoes
   It should confirm that they have different identity.
```

Pero la clave en nuestro caso está en:

```
When comparing two equal cargoes
   It should leverage the tracking identity collaborator.
```

Esta aserción es la que marca la diferencia, ya que con ella lo que estamos probando es la interacción entre un modulo y sus dependencias.

Esta discusión es casi tan vieja como el TDD, ya que es la misma discusión que hay entre TDD Tradicional (State-Based Testing TDD) frente a TDD Mockista (Interaction-Based Testing TDD), donde para testar la operación suma de una calculadora:

- TDD Tradicional: Compruebo que cuando le paso 15 y 20 me devuelve 35 y quizás también compruebo que cuando le paso 6 y 10 me devuelve 16.
- TDD Mockista: Compruebo que cuando le paso dos sumandos enteros, se produce una única llamada a la función suma de la calculadora y ésta me devuelve un único valor que es también un entero.

Aunque el ejemplo escogido quizá no sea el más adecuado, ilustra bastante bien la base del problema.

¿Como trasladamos eso a nuestro programa?

Claramente, al tener:

```
When comparing two equal cargoes
   It should leverage the tracking identity collaborator.
```

no podemos negar que nos estamos situando del lado del TDD Mockista o lo que es lo mismo del Interaction-Based Testing TDD.

Usando la analogía de la calculadora, estaríamos comprobando que "cuando le paso dos enteros se produce una única llamada a la función suma de la calculadora...". Y al tener a mayores:

```
When comparing two equal cargoes
   It should confirm that they have the same identity.
```

estaríamos comprobando también que "...y ésta me devuelve un único valor que es también un entero". Por lo tanto, en nuestro caso:

```
When comparing two different cargoes
   It should confirm that they have different identity.
```

es redundante.

Aún así no nos cansaremos de repetir que implementar ese escenario tampoco estaría mal, simplemente indicaría que le damos mucha importancia al hecho de cubrir cualquier posible contingencia.

1.6.5.3 Contradicción

Y una vez dicho, explicado y razonado este punto, vamos a contradecirnos y, teniendo en cuenta la vocación pedagógica de este documento, hemos decidido ser redundantes e implementar la **BDD_Spec**:

```
public class when_comparing_two_different_cargoes : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();

        the_to_compare_cargo = an<ICargo>();
        tracking_id
           .Stub(x => x.has_the_same_value_as(
                       the_to_compare_cargo.tracking_id()))
           .Return(false);
    };

    Because of = () =>
       result = sut.has_the_same_identity_as(the_to_compare_cargo);

    It should_confirm_that_they_have_different_identity = () =>
       result.ShouldBeFalse();

    static ITrackingId tracking_id;
    static ICargo the_to_compare_cargo;
    static bool result;
    static IRouteSpecification route_specification;
}
```

No hace falta hacer ningún comentario a mayores, más allá de mostrar la prueba de que sin haber tocado el código de nuestro **DDD_Aggregate_Root** `Cargo`, la **BDD_Spec** funciona:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
» should leverage the route specification origin location
```

```
when comparing two equal cargoes
» should confirm that they have the same identity
» should leverage the tracking identity collaborator

when comparing two different cargoes
» should confirm that they have different identity


7 passed, 0 failed, 0 skipped, took 0,93 seconds (Machine.Specifications
0.3.0).
```

1.6.6 BDD vs BDUF

Un último apunte antes de proceder a empaquetar y a **github**.

Debería ser evidente que el diseño de la aplicación es un proceso constante (lo hemos repetido hasta la saciedad) y en pequeños pasos, de tal forma que, en cualquier fase o etapa puede surgir un nuevo apunte que nos lleve a la necesidad de modificar el diseño de nuestro sistema. Esto es algo que contrasta enormemente con los sistemas que siguen el paradigma del **BDUF** (**Big Design Up Front**), del cual nos situamos en las antípodas.

1.7 Refactorizando CargoSpecs

Desde el principio hemos tenido la **clase abstracta** `concern_for_cargo` y no le hemos sacado mucho partido, más allá de usarla como simple intermediario para declarar el `Observes<Contract, ClassUnderTest>`:

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo>
```

Eso va a cambiar, ya que vamos a ver como nos puede ayudar a reducir el **ruido** de nuestras **BDD_Specs**.

Para ello, escribimos:

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo>
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();
    };

    protected static ITrackingId tracking_id;
    protected static IRouteSpecification route_specification;
}
```

Es decir, hemos cogido todo el código relativo a la **inyección de dependencias** del constructor del SUT y lo hemos aislado en una **clase base** que heredan todas las **BDD_Specs** de `Cargo`.

Es así como reduciremos el **ruido** de todas nuestras **BDD_Specs**.

Tomemos como ejemplo del **ANTES / DESPUÉS** la **BDD_Spec**:

ANTES:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
```

```
        route_specification = the_dependency<IRouteSpecification>();
    };

    Because of = () => result = sut.tracking_id();

    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId tracking_id;
    static ITrackingId result;
    static IRouteSpecification route_specification;
}
```

DESPUES:

```
public class when_asked_about_its_tracking_identificator : concern_for_cargo
{
    Because of = () => result = sut.tracking_id();

    It should_provide_the_cargo_id = () => result.ShouldEqual(tracking_id);

    static ITrackingId result;
}
```

O bien otro ejemplo un poco más complejo como esta otra **BDD_Spec**:

ANTES:

```
public class when_comparing_two_equal_cargoes : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();

        the_to_compare_cargo = an<ICargo>();
        tracking_id
           .Stub(x => x.has_the_same_value_as(
                        the_to_compare_cargo.tracking_id()))
           .Return(true);
    };

    Because of = () =>
       result = sut.has_the_same_identity_as(the_to_compare_cargo);

    It should_confirm_that_they_have_the_same_identity = () =>
       result.ShouldBeTrue();

    It should_leverage_the_tracking_identity_collaborator = () =>
```

```
        tracking_id.received(x => x
                    .has_the_same_value_as(the_to_compare_cargo.tracking_id()));

    static ITrackingId tracking_id;
    static ICargo the_to_compare_cargo;
    static bool result;
    static IRouteSpecification route_specification;
}
```

DESPUES:

```
public class when_comparing_two_equal_cargoes : concern_for_cargo
{
    Establish context = () =>
    {
        the_to_compare_cargo = an<ICargo>();
        tracking_id
           .Stub(x => x.has_the_same_value_as(
                        the_to_compare_cargo.tracking_id()))
           .Return(true);
    };

    Because of = () =>
       result = sut.has_the_same_identity_as(the_to_compare_cargo);

    It should_confirm_that_they_have_the_same_identity = () =>
       result.ShouldBeTrue();

    It should_leverage_the_tracking_identity_collaborator = () =>
       tracking_id.received(x => x
                   .has_the_same_value_as(the_to_compare_cargo.tracking_id()));

    static ICargo the_to_compare_cargo;
    static bool result;
}
```

Parece que mejoramos bastante en el aspecto de la legibilidad, sin perder apenas conocimiento del contexto.

Aunque la mayor parte de las veces extraer código de una suite de **BDD_Specs** y pasarla a una **clase base abstracta** debe ser considerado un **anti-pattern** (ya que es muy fácil descontextualizar una **BDD_Spec** y hacer que de la impresión de que las cosas ocurren de forma casi mágica), si se hace juiciosamente y con un poco de cuidado, como en este caso, el resultado es el opuesto.

Sea como fuere, ya tenemos los conocimientos para poder hacerlo cuando lo consideremos necesario.

Por último, no debemos olvidarnos de correr la suite de **BDD_Specs** para comprobar que no hemos roto nada:

```
7 passed, 0 failed, 0 skipped, took 1,38 seconds (Machine.Specifications
0.3.0).
```

1.8 BDD_Spec - When comparing to a null cargo

Nuestro método `has_the_same_identity_as(ICargo the_other_cargo)` tiene una vulnerabilidad muy grande. Al no comprobar que `the_other_cargo` no sea `null` y tener un comportamiento asociado en el que espera que un objeto colaborador (`ITrackingId`) se encargue de implementar la funcionalidad, tal y como se muestra en la **BDD_Spec** siguiente:

```
When comparing two equal cargoes
   It should leverage the tracking identity collaborator.
```

podría producirse una excepción al efectuarse la llamada al colaborador, ya que:

```
the_other_entity.tracking_id();
```

no podría ejecutarse.

Por lo tanto, necesitamos la siguiente **BDD_Spec** que nos cubra en el caso de producirse dicha contingencia:

```
When comparing to a null cargo
   It should confirm that they have different identity.
   It should not leverage the tracking identity collaborator.
```

> **NOTA**: Esta **BDD_Spec** formaría parte de los escenarios complementarios al Happy Day Scenario.

Es decir, no solo aseguramos que van a ser diferentes al compararse:

```
It should confirm that they have different identity.
```

sino que, estamos diseñando como queremos que se comporte el sistema con respecto a las interacciones con sus dependencias, al obligarlo, en este caso, a no delegar el calculo a `ITrackingId`:

It should not leverage the tracking identity collaborator.

1.8.1 La BDD_Spec en código

Puesto que ya somos unos expertos **BDDistas**, podemos ir directamente a la **BDD_Spec**, sin tener que detallar como la hemos ido escribiendo y corrigiendo, al mismo tiempo, el código rojo.

> **NOTA**: En caso de dudas con respecto a este tema, es interesante volver a consultar el desarrollo detallado de las **BDD_Specs** anteriores.

Además no se produce ninguna **reacción en cadena**, por lo que apenas podremos comentar nada al respecto:

```
public class when_comparing_to_a_null_cargo : concern_for_cargo
{
    Establish context = () =>
    {
        a_null_to_compare_cargo = null;
        tracking_id
           .Stub(x => x.has_the_same_value_as(an<ICargo>().tracking_id()));
    };

    Because of = () =>
       result = sut.has_the_same_identity_as(a_null_to_compare_cargo);

    It should_confirm_that_they_have_different_identity = () =>
       result.ShouldBeFalse();

    It should_not_leverage_the_tracking_identity_collaborator = () =>
       tracking_id.never_received(x => x
                  .has_the_same_value_as(an<ICargo>().tracking_id()));

    static bool result;
    static ICargo a_null_to_compare_cargo;
}
```

Y recordemos que:

```
public abstract class concern_for_cargo : Observes<ICargo, Cargo>
```

```
{
    Establish context = () =>
    {
        tracking_id = the_dependency<ITrackingId>();
        route_specification = the_dependency<IRouteSpecification>();
    };

    protected static ITrackingId tracking_id;
    protected static IRouteSpecification route_specification;
}
```

Simplemente comentar que:

```
an<ICargo>().tracking_id()
```

en ambos casos es `null`.

Podemos comprobarlo de dos formas:

- Asignando ese valor a un campo y creando una aserción que lo compruebe.
- Usando el **debugger**.

1.8.1.1 Machine.Specifications y los null

Esto último plantea una cuestión interesante, ¿cuales de los siguientes campos es `null`?:

```
Establish context = () =>
{
    a_null_to_compare_cargo = null;
    an_icargo = an<ICargo>();
    an_icargo_tracking_id = an_icargo.tracking_id();
    an_itracking_id = an<ITrackingId>();
};
```

- `a_null_to_compare_cargo`: Aquí no hay discusión posible, es `null`.
- `an_icargo`: No es `null`, es algo parecido a `{ICargoProxyf74dc6fd2e1646d...}`.
- `an_icargo_tracking_id`: Éste es el mismo caso de nuestra **BDD_Spec**. Es `null`.
- `an_itracking_id`: No es `null`, es algo parecido a `{ITrackingIdProxy4b466c1...}`.

Se puede comprobar, tal y como explicamos unas lineas más arriba.

Según esto podemos concluir que también podíamos haber escrito nuestra **BDD_Spec**, tal que así:

```
public class when_comparing_to_a_null_cargo : concern_for_cargo
{
    Establish context = () =>
    {
        a_null_to_compare_cargo = null;
        tracking_id
           .Stub(x => x.has_the_same_value_as(null));
    };

    Because of = () =>
       result = sut.has_the_same_identity_as(a_null_to_compare_cargo);

    It should_confirm_that_they_have_diferent_identity = () =>
       result.ShouldBeFalse();

    It should_not_leverage_the_tracking_identity_collaborator = () =>
       tracking_id.never_received(x => x.has_the_same_value_as(null));

    static bool result;
    static ICargo a_null_to_compare_cargo;
}
```

Esta versión nos parece, incluso, más clara, así que, nos quedamos con ella.

1.8.2 Satisfaciendo las aserciones

Una vez aclarado ésto, intentemos no perder de vista el objetivo, que es, evitar que un `ICargo null` nos rompa el sistema.

Para satisfacer ambas aserciones, basta con:

```
public class Cargo : ICargo
{
    (....)

    public bool has_the_same_identity_as(ICargo the_other_entity)
    {
        return the_other_entity != null &&
               underlying_tracking_id
```

```
                .has_the_same_value_as(the_other_entity.tracking_id());
    }

    (....)
}
```

para obtener:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
» should leverage the route specification origin location

when comparing two equal cargoes
» should confirm that they have the same identity
» should leverage the tracking identity collaborator

when comparing two different cargoes
» should confirm that they have different identity

when comparing to a null cargo
» should confirm that they have different identity
» should not leverage the tracking identity collaborator


9 passed, 0 failed, 0 skipped, took 0,90 seconds (Machine.Specifications
0.3.0).
```

1.8.3 Más diseño

Si leemos el log anterior con atención, nos damos cuenta de que hay algo que chirría.

Cuando leemos:

```
when comparing two equal cargoes
» should confirm that they have the same identity
» should leverage the tracking identity collaborator
```

o:

```
when comparing to a null cargo
» should confirm that they have different identity
» should not leverage the tracking identity collaborator
```

nos queda claro cual es el comportamiento esperado con respecto a la interacción con `ITrackingId`:

- En el primer caso: Delega.
- En el segundo caso: No delega.

Pero, ¿que ocurre en este caso?

```
when comparing two different cargoes
» should confirm that they have different identity
```

No sabemos si delega o no.

Antes, cuando no teníamos la **BDD_Spec** que cubría el comportamiento de la comparación con un `ICargo null`, no molestaba tanto a la vista. Teníamos:

- El Happy Day Scenario ,cuando los dos `ICargoes` son iguales.
- Uno escenario complementario, cuando son distintos, que presuponíamos comprobaría de la misma forma (delegando).

Pero ahora, ya no queda tan claro, pues hay un caso en el que no delega (comparar con un `ICargo null`).

Sobre la necesidad o no de haber incluido una **BDD_Spec** como la de comparar dos `ICargoes` distintos, ya hemos discutido. Nos sirvió de escusa para hacer un poco de proto-historia del **BDD** con su primo-hermano el TDD, y no tenemos intención de volver a explicar esos conceptos.

Lo que hemos decidido es añadir una aserción más, que nos sirva para disipar cualquier duda:

```
When comparing two different cargoes
   It should confirm that they have different identity.
   It should leverage the tracking identity collaborator.
```

No vamos a mostrar aquí el código (siempre podemos consultar el código fuente directamente), ya que no tienen mayor relevancia.

Sin embargo, esto sí que es interesante:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
» should leverage the route specification origin location

when comparing two equal cargoes
» should confirm that they have the same identity
» should leverage the tracking identity collaborator

when comparing two different cargoes
» should confirm that they have different identity
» should leverage the tracking identity collaborator

when comparing to a null cargo
» should confirm that they have different identity
» should not leverage the tracking identity collaborator


10 passed, 0 failed, 0 skipped, took 0,98 seconds (Machine.Specifications
0.3.0).
```

Ahora podemos comprobar como la inconsistencia a la hora de leer las **BDD_Specs** que propiciaron este cambio, ya no existe.

Además hemos conseguido hacer pasar nuestra suite de **BDD_Specs** sin necesidad de haber añadido una sola linea de código a nuestro **DDD_Aggregate_Root** `Cargo`.

1.9 BDD_Spec - Sobrescribiendo GetHashCode() y ToString()

Se considera una buena práctica, sobrescribir los métodos de `Object`, a saber:

- `GetHashCode()`
- `ToString()`
- `Equals()`

Si bien sabemos gracias al **LSP** de las posibles implicaciones negativas de esta tarea.

Las **BDD_Specs** asociadas a `GetHashCode()` y `ToString()` podrían ser éstas:

```
When asked about its hash code
   It should leverage the tracking identity hash code.

When asked about its string representation
   It should leverage the tracking identity string representation.
```

Por lo tanto, volvemos a delegar la tarea de los cálculos a un colaborador (`ITrackingId`). **Más diseño**.

Su versión en código no introduce nada que no hayamos visto con anterioridad:

HashCode:

```
public class when_asked_about_its_hash_code : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id
           .Stub(x => x.GetHashCode())
           .Return(new int());
    };

    Because of = () => sut.GetHashCode();

    It should_leverage_the_tracking_identity_hash_code = () =>
       tracking_id.received(x => x.GetHashCode());
}
```

`ToString:`

```
public class when_asked_about_its_string_representation : concern_for_cargo
{
    Establish context = () =>
    {
        tracking_id
           .Stub(x => x.id())
           .Return(string.Empty);
    };

    Because of = () => sut.ToString();

    It should_leverage_the_tracking_identity_string_representation = () =>
       tracking_id.received(x => x.id());
}
```

Para que esto funcione, necesitamos hacer los siguientes cambios tanto en la interface `ITrackingId`:

```
public interface ITrackingId : IValueObject<ITrackingId>
{
    int GetHashCode();
    string id();
}
```

como en la interface `ICargo`:

```
public interface ICargo : IEntity<ICargo>
{
    ITrackingId tracking_id();
    IRouteSpecification route_specification();
    ILocation origin_location();

    int GetHashCode();
    string ToString();
}
```

Los cambios necesarios que tenemos que hacerle a nuestro **DDD_Aggregate_Root** `Cargo` para que implemente esta nueva funcionalidad, van en consonancia con el estilo de lo que hemos venido haciendo para todas las **BDD_Specs** de `Cargo`:

```
public class Cargo : ICargo
{
```

```
    (....)

    public override int GetHashCode()
    {
        return underlying_tracking_id.GetHashCode();
    }

    public override string ToString()
    {
        return underlying_tracking_id.id();
    }

    (....)
}
```

De esta forma obtenemos como resultado el esperado:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about its tracking identificator
» should provide the cargo id

when asked about its route specification
» should provide the route specification

when asked about its origin location
» should provide the origin location
» should leverage the route specification origin location

when comparing two equal cargoes
» should confirm that they have the same identity
» should leverage the tracking identity collaborator

when comparing two diferent cargoes
» should confirm that they have diferent identity
» should leverage the tracking identity collaborator

when comparing to a null cargo
» should confirm that they have diferent identity
» should not leverage the tracking identity collaborator

when asked about its hash code
» should leverage the tracking identity hash code

when asked about its string representation
» should leverage the tracking identity string representation


12 passed, 0 failed, 0 skipped, took 1,29 seconds (Machine.Specifications
0.3.0).
```

Hasta aquí nada nuevo ni excesivamente complicado.

Seguimos usando las interfaces como un contrato que se debe cumplir. Además, reforzamos el concepto de qué es lo que necesita una clase para considerarse del tipo `ICargo`, al haber añadido los dos métodos `ToString()` y `GetHashCode()` a la propia interface `ICargo` de manera explícita.

Otra cuestión a la que ya deberíamos empezar a acostumbrarnos es a que gracias al poder combinado del **BDD** y el correcto uso de interfaces, podemos tener aserciones para código como:

```
underlying_tracking_id.GetHashCode();
underlying_tracking_id.id();
```

Y todo esto, sin ni siquiera haber escrito una sola implementación concreta de `ITrackingId`, por poner un ejemplo.

1.10 BDD_Spec – When assigning a route described by an itinerary

1.10.1 Breve descripción del problema a resolver

Hasta ahora no hemos considerado necesario describir lo que va haciendo la aplicación, pues con las **BDD_Specs** era más que suficiente. Sin embargo, para continuar el desarrollo se hace necesario un poco de contextualización.

Cuando creamos un envío (`ICargo`) se supone que alguien lo ha tenido que contratar (**booking**). De esa forma creamos el envío con:

- Un identificador (`ITrackingId`) que nos permite distinguirlo de los demás envíos.
- Un origen (`ILocation`).
- Unos requisitos asociados a la ruta (`IRouteSpecification`).

Durante un breve periodo de tiempo tras la contratación (**booking**) del envío (`ICargo`), éste no tiene un itinerario asociado. Es en este momento cuando el personal de la empresa, pide una lista de rutas validas que cumplan los requisitos asociados a la ruta (`IRouteSpecification`) y asigna el envío (`ICargo`) a una de esas rutas válidas. Es precisamente esa ruta la que viene descrita por un itinerario (`IItinerary`).

Así que en `ICargo` tiene que haber un método del estilo de: `assign_the_route_described_by(IItinerary this_itinerary)`.

1.10.2 La BDD_Spec

Podemos partir, inicialmente, de la siguiente **BDD_Spec**, ya que define la situación de una forma bastante ajustada:

```
When assigning a route described by an itinerary
   It should validate it is a not null itinerary.
   It should be stored somewhere.
   It should reflect into the actual transportation of the cargo.
```

Esta vez parece que se aproxima una buena **reacción en cadena**, así que vamos paso a paso (omitiendo los completamente triviales).

1.10.3 Los bloques IT

El código de esta **BDD_Spec** podría ser:

```
public class when_assigning_a_route_described_by_an_itinerary :
concern_for_cargo
{
    It should_validate_it_is_a_not_null_itinerary = () =>
       the_itinerary.ShouldNotBeNull();

    It should_be_stored_somewhere = () =>
       sut.itinerary().ShouldEqual(the_itinerary);

    It should_reflect_into_the_actual_transportation_of_the_cargo = () =>
       sut.delivery()
          .received(x => x
          .update_on_routing(the_route_specification, the_itinerary));

    static IItinerary the_itinerary;
    static IDelivery the_delivery;
}
```

Como siempre, la **reacción en cadena** nos lleva a moldear el sistema de forma acorde. En este caso, necesitamos crear dos nuevas interfaces:

```
public interface IDelivery {}
public interface IItinerary {}
```

En `IDelivery` vamos a usar el siguiente método:

```
public interface IDelivery
{
    void update_on_routing(IRouteSpecification the_route_specification,
                           IItinerary the_itinerary);
}
```

Mientras, en `ICargo`, aparecen dos nuevos métodos que nos devuelve un `IItinerary` y un `IDelivery` respectivamente:

```
public interface ICargo : IEntity<ICargo>
```

```
{
    (....)

    IItinerary itinerary();
    IDelivery delivery();

    (....)
}
```

Y como consecuencia, también se reflejan en `Cargo`:

```
public class Cargo : ICargo
{
    (....)

    public IItinerary itinerary()
    {
        throw new NotImplementedException();
    }

    public IDelivery delivery()
    {
        throw new NotImplementedException();
    }

    (....)
}
```

En cuanto enviamos las dos nuevas interfaces a su sitio ya nos hemos deshecho de todo el código rojo.

1.10.4 Ejercitando el SUT

Vamos a ejercitar el SUT:

```
public class when_assigning_a_route_described_by_an_itinerary :
concern_for_cargo
{
    Because of = () => sut.assing_the_route_described_by(the_itinerary);

    It should_validate_it_is_a_not_null_itinerary = () =>
       the_itinerary.ShouldNotBeNull();

    It should_be_stored_somewhere = () =>
       sut.itinerary().ShouldEqual(the_itinerary);
```

```
    It should_reflect_into_the_actual_transportation_of_the_cargo = () =>
       sut.delivery()
          .received(x => x
          .update_on_routing(the_route_specification, the_itinerary));

    static IItinerary the_itinerary;
    static IDelivery the_delivery;
}
```

Para deshacernos del código rojo, basta con añadir ese método tanto a `ICargo`:

```
public interface ICargo : IEntity<ICargo>
{
    (....)

    void assing_the_route_described_by(IItinerary the_itinerary);

    (....)
}
```

como a `Cargo`:

```
public class Cargo : ICargo
{
    (....)

    public void assing_the_route_described_by(IItinerary the_itinerary)
    {
        throw new NotImplementedException();
    }

    (....)
}
```

Una vez hecho esto, podemos pasar a establecer el contexto.

1.10.5 El bloque ESTABLISH

Establecer el contexto es muy sencillo, tal y como se muestra en la **BDD_Spec** ya finalizada:

```
public class when_assigning_a_route_described_by_an_itinerary :
concern_for_cargo
{
    Establish context = () =>
```

```
    {
        the_itinerary = an<IItinerary>();
        the_delivery = an<IDelivery>();

        the_delivery
           .Stub(x => x.update_on_routing(
                        the_route_specification, the_itinerary));
    };

    Because of = () => sut.assing_the_route_described_by(the_itinerary);

    It should_validate_it_is_a_not_null_itinerary = () =>
       the_itinerary.ShouldNotBeNull();

    It should_be_stored_somewhere = () =>
       sut.itinerary().ShouldEqual(the_itinerary);

    It should_reflect_into_the_actual_transportation_of_the_cargo = () =>
       sut.delivery()
          .received(x => x
          .update_on_routing(the_route_specification, the_itinerary));

    static IItinerary the_itinerary;
    static IDelivery the_delivery;
}
```

Solo nos resta obtener el error esperado al ejecutar la **BDD_Spec**:

```
System.Reflection.TargetInvocationException:
Exception has been thrown by the target of an invocation.
 ---> System.NotImplementedException: The method or operation is not
implemented.
```

1.10.6 Red-Green-Refactor y Arrange-Act-Assert

Una vez entendido el proceso que hay que seguir, en general, a la hora de practicar **BDD**:

- RED
- GREEN
- REFACTOR

Y que en particular, el estilo que debemos perseguir con las **BDD_Specs**, es el conocido como AAA:

- Arrange
- Act
- Assert

podemos establecer una relación directa con los distintos bloques puestos a nuestra disposición vía `machine.specifications`:

- Assert --> bloques `IT`.
- Act --> bloque `BECAUSE`.
- Arrange --> bloque `ESTABLISH`.

El orden, en este caso, es de vital importancia. Debemos ir primero con los Assert, a continuación con el Act, y finalmente con el Arrange (tal y como hemos venido haciendo hasta ahora).

Como decíamos, una vez que tenemos claros estos conceptos, rara será la vez en la que nos encontremos escribiendo código en grandes cantidades y/o con un grado de complejidad alto. Más bien, todo lo contrario. Escribiremos poco código y éste será fácil de escribir (y por suerte también de leer).

1.10.7 Pasando el primer bloque IT

Vamos a empezar por el primer bloque `IT`:

```
It should_validate_it_is_a_not_null_itinerary = () =>
   the_itinerary.ShouldNotBeNull();
```

La intención es clara, sin embargo la realización es incorrecta. Si comprobamos que:

```
the_itinerary.ShouldNotBeNull();
```

estaremos comprobando que no le pasamos un itinerario nulo, pero no estaremos comprobando que en caso de que le pasemos un itinerario nulo, nuestro SUT sabrá tratarlo acordemente.

Cometer este tipo de errores es común y no tienen la menor importancia siempre y cuando los detectemos a tiempo. Cuando desarrollamos siguiendo el paradigma del **BDD**, el código está en un estado de constante cambio. No es algo malo, más bien al contrario, es un síntoma de que estamos ajustándonos a los requisitos buscando la mejor solución a cada momento.

Vamos a corregir el bloque `IT` para que refleje lo que de verdad pretendemos:

```
It should_validate_it_is_a_not_null_itinerary = () =>
   sut.itinerary().ShouldNotBeNull();
```

Esto sí que se parece más a lo que necesitamos.

Suponemos que el itinerario ha pasado por todo el procesado a cargo del **DDD_Aggregate_Root** `Cargo`, y cuando lo pedimos de vuelta no puede ser nulo.

Pero esto plantea una cuestión a mayores para comprobar que no "traga" con un `null`. En ningún sitio tenemos definida la condición por la que comprobamos ese escenario (enviarle un `null`).

Incluso hay otro problema más grave, si le enviamos un `null`, debería rechazarlo. Pero si lo rechaza, no lo asigna internamente, con lo que:

```
sut.itinerary()
```

podría ser `null`, ya que no tiene porqué haber sido asignado nunca.

Cuando llegamos a situaciones tan absurdas como ésta, uno debe plantarse si tiene los requisitos correctos. En este caso concreto, podría ser el momento ideal para consultar con un compañero (aunque se supone que al practicar **BDD**, practicamos pair programming, así que deberíamos tener a un compañero a nuestro lado) nuestro problema.

Después de pensarlo un rato, llegamos a la conclusión de que el problema es un problema de base. No basta con validar que no sea `null`, debemos lanzar una

excepción en caso de ser `null`. De esta forma, reforzamos todavía más la idea de que no puede ser `null` y además, curiosamente, encontramos una forma muy sencilla de comprobar lo que queremos. Ahora basta con que comprobemos que al pasarle un `null` se lanza una excepción. Pero esto nos obliga a dividir nuestra **BDD_Spec** en dos, ya que ahora nos hace falta algo como:

```
When assigning a route described by a null itinerary
   It should throw an argument null exception.
```

que dejaremos para un poco más adelante, ya que nuestra idea es implementar antes el Happy Day Scenario. Y si hay alguien que considera que enviar un argumento nulo a un método para provocar una excepción es un Happy Day Scenario, debería pensarse mucho un cambio de profesión.

1.10.8 Pasando el segundo bloque IT

Vamos con el segundo bloque `IT` que parece mucho más del estilo del Happy Day Scenario:

```
It should_be_stored_somewhere = () =>
   sut.itinerary().ShouldEqual(the_itinerary);
```

Es decir, necesitamos un campo privado en el que almacenarlo y de rebote (también se puede ver como una **mini reacción en cadena**) estamos sugiriendo la necesidad del método `itinerary()`, del que no teníamos constancia hasta ahora y que necesitará de su propia **BDD_Spec**.

Vamos a por el código en `Cargo`:

```
IItinerary underlying_itinerary;

public void assing_the_route_described_by(IItinerary the_itinerary)
{
    this.underlying_itinerary = the_itinerary;
}
```

Y en cuanto al método `itinerary()`:

```
public IItinerary itinerary()
{
    return underlying_itinerary;
}
```

Esto debería bastar:

```
------ Test started: Assembly: dddsample.specs.dll ------

(....)

when assigning a route described by an itinerary
» should be stored somewhere
» should reflect into the actual transportation of the cargo (FAIL)


Test 'should reflect into the actual transportation of the cargo' failed:
      System.NotImplementedException: The method or operation is not
implemented.
      domain\model\cargo.aggregate\Cargo.cs(42,0): at
dddsample.domain.model.cargo.aggregate.Cargo.delivery()
      domain\model\cargo.aggregate\CargoSpecs.cs(172,0): at
dddsample.specs.domain.model.cargo.aggregate.when_assigning_a_route_descri
bed_by_an_itinerary.<.ctor>b__4()
      at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
      at Machine.Specifications.Model.Specification.Verify()

13 passed, 1 failed, 0 skipped, took 1,22 seconds (Machine.Specifications
0.3.0).
```

Por fin algo de GREEN.

1.10.9 Pasando el tercer bloque IT

Vamos con el tercer bloque IT:

```
It should_reflect_into_the_actual_transportation_of_the_cargo = () =>
   sut.delivery()
      .received(x => x
      .update_on_routing(the_route_specification, the_itinerary));
```

No es la primera vez que nos encontramos con este tipo de aserción que se encarga de comprobar que se produce una interacción del SUT con alguna de sus

dependencias o colaboradores.

¿Cual es el problema aquí?

El objeto `IDelivery` no es inyectado en el constructor de `ICargo`, y ese simple hecho, hace que `IDelivery` se convierta en un miembro del "**lado oscuro de la fuerza**", una dependencia oculta.

En el código original en java solucionan esto con un método `static`, pero en nuestro caso, **Interface-Based Programming** y métodos `static` son agua y aceite. No mezclan.

> **NOTA**: La implementación original en java de la aplicación DDDSample a cargo de los propios Eric Evans y Jimmy Nilsson se puede encontrar en esta dirección web: http://dddsample.sourceforge.net/

Barajamos dos opciones:

La primera sería seguir el camino marcado por el código original en java y declarar el método `static`, con lo que ya podemos olvidarnos de mockar `IDelivery` y hacer aserciones sobre si se ha invocado determinado método en él.

La segunda sería considerar el método `static` como una **DDD_Factory** que devuelve objetos `IDelivery`. Para ello necesitaríamos:

- Convertirlo en no `static`.
- Desincrustarlo de la implementación de `IDelivery` (presumiblemente con un nombre tan original como `Delivery : IDelivery`).
- Darle su propio objeto (algo así como `DeliveryService` o `DeliveryFactory`).
- Inyectar a continuación un objeto `IDelivery` en el constructor de `Cargo`.

El impacto de esta segunda opción, sería tal que habría que revisar todas las **BDD_Specs** de `CargoSpecs`. Sin embargo, al haber hecho uso de la **clase base abstracta** `concern_for_cargo` para refactorizar el código pertinente a la creación del SUT, realizaríamos los cambios exclusivamente en ese emplazamiento y pudiera ser que no

afectase en nada más a las **BDD_Specs** de `CargoSpecs`.

El impacto que pueda tener en el resto de las clases que conforman el **DDD_Aggregate** `Cargo` debería ser nulo ya que, en principio, solo invocamos ese método desde el **DDD_Aggregate_Root** `Cargo`.

Esta segunda opción es otro ejemplo más de como, si practicamos **BDD**, debemos estar listos para diseñar en cualquier momento.

Viendo que ya desde el principio nos hemos alejado muy mucho de la versión original java, si escogiésemos la opción 1, no podría ser simplemente por mantenernos relativamente cerca del código original, sino que debería haber una razón de peso, como la falta de alternativas viables, para tomar esa decisión.

> **NOTA**: El que nos alejemos más y más del código original de java, no es algo que hagamos por gusto ni a propósito, sino que al intentar recrear un sistema de estas características desde un planteamiento **BDD**, es lógico que acabemos haciendo **Interface-Based Programming**, con lo que nos situaríamos en una carretera diferente a la de la implementación original. Es el propio proceso el que nos guía poco a poco en una dirección concreta.

Desde luego, la opción 2 tiene más encanto. Nos permite mantener intacta nuestra fe en el **BDD** y además, nos parece la decisión correcta si intentamos aplicar el **SRP**.

Otro beneficio añadido, en caso de escoger la opción 2, es que sería la decisión correcta si quisiésemos introducir, más adelante, un framework **DI / IoC** como **Ninject!**.

Por el contrario, no podemos negar que la opción 2 supone una carga extra en cuanto a **Complejidad Innecesaria** (**Needless Complexity**) se refiere, ya que el hecho de contar con una **DDD_Factory** para crear `ICargoes` complica las cosas.

Sea como fuere, hemos decidido que no nos parece el mejor momento para escoger entre estas dos opciones, así que lo que nos gustaría, sería poder salvar la

documentación generada y volver al punto del **commit** anterior, pero a su vez queremos poder retomar todo lo discutido y el código asociado a ello. ¿Como hacemos esto?

1.10.10 Git al rescate

Muy fácil, **git** convierte este tipo de problemas en algo trivial.

Vamos, inicialmente, a repasar donde estamos, para, a continuación, ver a donde queremos ir.

1.10.10.1 Donde estamos

Estamos en la **rama dev** y según nos informa:

```
git status
```

hemos modificado los siguientes ficheros:

- `DDD_Sample Proceso.odt`: La base de este documento.
- `CargoSpecs.cs`.
- `dddsample.csproj`.
- `Cargo.cs`.
- `ICargo.cs`.

y hemos creado estos otros:

- `IDelivery.cs`.
- `IItinerary.cs`.

1.10.10.2 A donde queremos llegar

Vamos paso a paso:

Primero

Pretendemos separar `DDD_Sample_Proceso.odt`, de los demás ficheros que han sido creados y/o modificados.

Para ello hacemos lo siguiente:

```
git add "documentation/DDD_Sample_Proceso.odt"
```

De esta forma este fichero pasa a ser el único susceptible de un **commit**. Si nos encontramos en una situación donde queremos separar, digamos, tres ficheros del resto, tenemos que repetir el paso:

```
git add "FICHERO"
```

tantas veces como ficheros queramos que se incluyan en el **commit**.

Ya hemos separado el fichero que queremos someter a un **commit**, del resto.

Segundo

Hacemos el **commit** ya que solo va a afectar al fichero que queremos.

Para ello ejecutamos:

```
git commit -m "MENSAJE"
```

o en caso de que tengamos configurado **git** con un editor por defecto:

```
git commit
```

y a continuación escribimos el **MENSAJE** en el editor de texto.

Una vez hecho esto ya tenemos salvado en **git** el cambio en el fichero que queríamos, de forma que si ejecutásemos un:

```
git status
```

ya solo encontraríamos los ficheros que no queremos perder pero que tampoco queremos todavía hacer **commit**, sino, guardarlos para cuando más adelante nos sintamos más confiados de poder solucionar el problema satisfactoriamente.

Tercero

Salvamos los cambios a una nueva **rama** fácilmente identificable.

Para ello desde la **rama dev** (que no hemos abandonado desde el primer paso), creamos una nueva **rama** con un nombre que nos resulte lo suficientemente fácil de identificar cuando queramos retomar el problema.

Así que basta con ejecutar:

```
git checkout -b assign-the-route-described-by-featurette
```

De esta forma hemos creado la **rama assign-the-route-described-by-featurette** que nos parece lo suficientemente descriptiva como para identificarla con posterioridad.

En esta **rama**, tenemos todos los cambio que hemos realizado. Es decir, nuestro trabajo no se ha ido al garete.

Cuarto

Volvemos a nuestra **rama dev**.

Para ello ejecutamos desde la **rama assign-the-route-described-by-featurette** en la que nos encontramos:

```
git checkout dev
```

Lo que nos devuelve a la **rama dev** e importa todos los cambios de la **rama** de origen **assign-the-route-described-by-featurette** en la que nos encontrábamos.

Quinto

Limpiamos la **rama dev**.

Para ello basta con ejecutar la siguiente secuencia de comandos **git**:

```
git reset --hard
git clean -d -f
```

De forma que situamos la **rama dev** en el estado asociado al **commit** anterior (en nuestro caso el que ejecutamos en el segundo paso), y además le pedimos que borre todo lo que encuentre que no pertenezca a ese **commit**.

Esta limpieza se produce ÚNICAMENTE en la **rama** actual, es decir **dev**, pero deja intactas el resto de las **ramas** como **master** o **assign-the-route-described-by-featurette**.

Sexto

Pasamos a la **rama master** y hacemos **merge** con la **rama dev**.

Para ello ejecutamos los dos comandos siguientes:

```
git checkout master
git merge dev
```

Poco más podemos comentar ya que esto forma parte del habitual **check-in dance**. Simplemente hemos "traído" el **commit** del segundo paso a la **rama master**.

Séptimo

Enviamos los cambios al **repositorio remoto** (**github**).

Para ello ejecutamos:

```
git push origin master
```

Éste es otro paso habitual en el **check-in dance**.

Octavo

Volvemos a la **rama dev** para seguir trabajando en una nueva característica.

Para ello ejecutamos el "conocido":

```
git checkout dev
```

que también forma parte del **check-in dance**.

Con esto terminamos.

Destacar la importancia de seguir los pasos en orden ya que si invertimos alguno de ellos el resultado puede alejarse de lo esperado.

Capítulo 2: Value Object RouteSpecification

2.1 Introducción

Vamos a cambiar de tercio. Hemos decidido que queremos implementar la funcionalidad de `IRouteSpecification`. Pero antes, vamos a ofrecer una escueta descripción que nos ayude a situarnos.

La responsabilidad de `IRouteSpecification` es:

"*Describir el Origen (origin) y el Destino (destination) de un Envío (`ICargo`), así como la fecha de entrega limite (arrival deadline) del mismo.*"

Es un **DDD_Value_Object** que situamos en el **DDD_Aggregate** `Cargo`, por lo que necesitaremos implementar `IValueObject<T>`.

Es también, a su vez, una **DDD_Specification** tal y como su nombre indica, por lo que necesitaremos implementar la interface `ISpecification<T>`. De todas formas, este aspecto no debería cogernos por sorpresa, ya que lo hemos dejado caer en una par de ocasiones.

Antes de ir con las **BDD_Specs**, debemos tener en cuenta que algunos procesos triviales que han sido tratados con profusión y todo lujo de detalles anteriormente (sobre todo en el desarrollo asociado a `CargoSpecs`), no van a recibir ahora la misma clase de atención. Intentaremos, sobre todo, destacar las **reacciones en cadena** que se produzcan, ya que, al fin y al cabo, estamos diseñando una aplicación, así como algunos procesos inherentes a la mecánica del **BDD**, que, a base de repetirse, se convierten en una segunda piel que nos ayuda a centrarnos en lo importante, sin por ello descuidar aspectos básicos.

Podemos considerar la analogía de la conducción de un vehículo. Al conducir un coche no pensamos en cambiar las marchas del mismo, simplemente lo hacemos, aunque al principio sí que era necesario fijarnos y recordar que debíamos cambiar de marcha. Con el tiempo nos acostumbramos y podemos centrarnos en lo importante, como no atropellar a los peatones, respetar las señales y circular por nuestro carril sin invadir el carril contrario, por poner un ejemplo.

2.2 concern_for_route_specification

Comenzaremos por algo sencillo, la **clase base abstracta** que nos sirve de puerta de entrada a las **BDD_Specs** y que situaremos en un fichero con el nada original nombre de `RouteSpecificationSpecs.cs`.

Para empezar, necesitamos especificar el `Observes<Contract, ClassUnderTest>`:

```
namespace dddsample.specs.domain.model.cargo.aggregate
{
    public abstract class concern_for_route_specification :
                          Observes<IRouteSpecification, RouteSpecification>{}
}
```

Si hacemos como Dorothy y seguimos el camino de baldosas rojas, iniciaremos la **reacción en cadena**:

```
public class RouteSpecification : IRouteSpecification
{
    public ILocation origin()
    {
        throw new NotImplementedException();
    }
}
```

> **NOTA**: Recordar que el método `origin()` surgió mientras diseñábamos la dupla `ICargo` / `Cargo`.

Además, sabemos por la descripción inicial, que `IRouteSpecification` debe implementar tanto `IValueObject<T>` por ser un **DDD_Value_Object** como `ISpecification<T>` por ser una **DDD_Specification**.

Vamos primeramente con el contrato `IValueObject<T>`, que ya deberíamos intuir que será implementado por `IRouteSpecification` y no directamente por `RouteSpecification` (si bien, puesto que `RouteSpecification` implementa `IRouteSpecification`, implementará de rebote `IValueObject<T>`):

```
public interface IRouteSpecification : IValueObject<IRouteSpecification>
```

```
{
    ILocation origin();
}
```

Esto nos obliga a implementar los métodos heredados de `IValueObject<IRouteSpecification>` en `RouteSpecification`:

```
public class RouteSpecification : IRouteSpecification
{
    public ILocation origin()
    {
        throw new NotImplementedException();
    }

    public bool has_the_same_value_as(
                    IRouteSpecification the_other_value_object)
    {
        throw new NotImplementedException();
    }
}
```

Una vez solucionada la parte relativa a `IValueObject<IRouteSpecification>`, procederemos con el contrato `ISpecification<T>`. Pero para ello necesitamos decidir quién es esa `<T>`. Es decir, cuando ejecutemos el método `is_satisfied_by()` de la `ISpecification<T>`, ¿quién queremos que satisfaga esa condición?.

La respuesta es el itinerario. Vamos a verificar que un posible itinerario satisfaga los requisitos asociados a la ruta. Es decir:

```
public interface IRouteSpecification : ISpecification<IItinerary>,
                                       IValueObject<IRouteSpecification>
{
    ILocation origin();
}
```

Para eliminar el color rojo debemos crear la nueva interface:

```
public interface IItinerary {}
```

vieja conocida nuestra de `CargoSpecs` (de hecho, está confinada en el limbo de las interfaces gracias a nuestra **rama assign-the-route-described-by-featurette**).

Pero no acaba ahí la **reacción en cadena**, ya que debemos implementar los métodos propios de `ISpecification<IItinerary>`:

```
public class RouteSpecification : IRouteSpecification
{
    public ILocation origin()
    {
        throw new NotImplementedException();
    }

    public bool has_the_same_value_as(
                    IRouteSpecification the_other_value_object)
    {
        throw new NotImplementedException();
    }

    public bool is_satisfied_by(IItinerary this_itinerary)
    {
        throw new NotImplementedException();
    }
}
```

Y aquí si que finaliza la **reacción en cadena**.

Lo más curioso es que ni siquiera hemos escrito una sola **BDD_Spec**, y ya disponemos de un armazón realmente convincente.

Enviamos tanto `RouteSpecification` como `IItinerary` al `namespace` correcto:

```
namespace dddsample.domain.model.cargo.aggregate
```

y podemos empezar con nuestra primera **BDD_Spec**.

2.3 RouteSpecificationSpecs - When asked for its origin location

Vamos con la primera **BDD_Spec** de `RouteSpecification`:

```
When asked for its origin location
   It should give back the origin location.
```

2.3.1 La BDD_Spec en código

Empezamos definiendo el `Context / Specification` (para nosotros conocido hasta ahora como `NombreDeLaClase / BloqueIT`):

```
public class when_asked_for_its_origin_location :
             concern_for_route_specification
{
    It should_give_back_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    static ILocation result;
    static ILocation the_origin_location;
}
```

Ejercitamos, a continuación, el SUT con un bloque `BECAUSE`:

```
public class when_asked_for_its_origin_location :
             concern_for_route_specification
{
    Because of = () => result = sut.origin();

    It should_give_back_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    static ILocation result;
    static ILocation the_origin_location;
}
```

Y finalmente establecemos las condiciones en las que queremos que esto ocurra con un bloque `ESTABLISH` que ayuda a describir el contexto inicial:

```
public class when_asked_for_its_origin_location :
             concern_for_route_specification
{
```

```
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = new DateTime();

        create_sut_using(() =>
           new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline));
    };

    Because of = () => result = sut.origin();

    It should_give_back_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    static ILocation result;
    static ILocation the_origin_location;
    static ILocation the_destination_location;
    static DateTime the_arrival_deadline;
}
```

Es importante destacar la necesidad de crear el SUT manualmente, en oposición a:

```
Establish context = () =>
{
    the_origin_location = the_dependency<ILocation>();
    the_destination_location = the_dependency<ILocation>();
    provide_a_basic_sut_constructor_argument(new DateTime());
};
```

por culpa de que el constructor del SUT recibe dos parámetro del mismo tipo, y ya vimos durante el desarrollo de `ISpecification<T>` que ante este escenario y con el objetivo de evitar problemas extraños de consistencia en los tests, mejor recurrir a la invocación manual empaquetada gracias a `create_sut_using()`.

2.3.2 Buscando el RED

Ejecutamos nuestra suite de **BDD_Specs** para obtener nuestro fallo y éste no llega porque se produce un error. Necesitamos definir un constructor que admita los tres argumentos:

```
public class RouteSpecification : IRouteSpecification
```

```
{
    public RouteSpecification(ILocation the_origin_location,
                              ILocation the_destination_location,
                              DateTime the_arrival_deadline)
    {
        throw new NotImplementedException();
    }

    (....)
}
```

Volvemos a intentar obtener nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for its origin location
» should give back the origin location (FAIL)

Test 'should give back the origin location' failed:
       System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

0 passed, 1 failed, 0 skipped, took 5,99 seconds (Machine.Specifications
0.3.0).
```

Que ahora por fin obtenemos.

2.3.3 Buscando el GREEN

A estas alturas deberíamos saber como solucionar el fallo.

Necesitamos escribir la mínima cantidad de código posible, pero siempre teniendo en cuenta que dicho código debe tener sentido:

```
public class RouteSpecification : IRouteSpecification
{
    readonly ILocation underlying_origin_location;

    public RouteSpecification(ILocation the_origin_location,
                              ILocation the_destination_location,
                              DateTime the_arrival_deadline)
```

```
    {
        this.underlying_origin_location = the_origin_location;
    }

    public ILocation origin()
    {
        return this.underlying_origin_location;
    }

    (....)
}
```

Comprobamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for its origin location
» should give back the origin location


1 passed, 0 failed, 0 skipped, took 0,62 seconds (Machine.Specifications
0.3.0).
```

Nada nuevo bajo el sol.

Vamos a aprovechar el impulso para solventar otra **BDD_Spec** análoga, antes de empaquetar y a **github**.

2.4 RouteSpecificationSpecs - When asked for its destination location

Ésta sería nuestra **BDD_Spec**:

```
When asked for its destination location
   It should give back the destination location.
```

2.4.1 El proceso completo

La **BDD_Spec** en código:

```
public class when_asked_for_its_destination_location :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = new DateTime();

        create_sut_using(() =>
           new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline));
    };

    Because of = () => result = sut.destination();

    It should_give_back_the_destination_location = () =>
       result.ShouldEqual(the_destination_location);

    static ILocation result;
    static ILocation the_origin_location;
    static ILocation the_destination_location;
    static DateTime the_arrival_deadline;
}
```

La **reacción en cadena**:

```
public interface IRouteSpecification : ISpecification<IItinerary>,
                                       IValueObject<IRouteSpecification>
{
    ILocation origin();
```

```
    ILocation destination();
}
```

y:

```
public class RouteSpecification : IRouteSpecification
{
    readonly ILocation underlying_origin_location;

    public RouteSpecification(ILocation the_origin_location,
                              ILocation the_destination_location,
                              DateTime the_arrival_deadline)
    {
        this.underlying_origin_location = the_origin_location;
    }

    public ILocation origin()
    {
        return this.underlying_origin_location;
    }

    public ILocation destination()
    {
        throw new NotImplementedException();
    }
}
```

El RED:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for its destination location
» should give back the destination location (FAIL)

Test 'should give back the destination location' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)


0 passed, 1 failed, 0 skipped, took 2,59 seconds (Machine.Specifications
0.3.0).
```

Arreglamos el fallo:

```
public class RouteSpecification : IRouteSpecification
{
    readonly ILocation underlying_origin_location;
    readonly ILocation underlying_destination_location;

    public RouteSpecification(ILocation the_origin_location,
                              ILocation the_destination_location,
                              DateTime the_arrival_deadline)
    {
        this.underlying_origin_location = the_origin_location;
        this.underlying_destination_location = the_destination_location;
    }

    public ILocation destination()
    {
        return this.underlying_destination_location;
    }

    (....)
}
```

El GREEN:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for its destination location
» should give back the destination location


1 passed, 0 failed, 0 skipped, took 0,58 seconds (Machine.Specifications
0.3.0).
```

Como no necesitamos REFACTOR, ahora sí que empaquetamos y a **github**.

2.5 RouteSpecificationSpecs - When asked for its arrival deadline

Con esta **BDD_Spec**, completaríamos el procesado vía el constructor de los tres argumentos inyectados. Si tenemos en cuenta que en las dos **BDD_Specs** anteriores fueron el origen y destino del envío los argumentos procesados, en esta **BDD_Spec**, nos encargaremos de la "fecha de entrega limite".

Para ello vamos primero con la **BDD_Spec**:

```
When asked for its arrival deadline
   It should give back the arrival deadline.
```

2.5.1 La BDD_Spec en código

En código sería algo así como:

```
public class when_asked_for_its_arrival_deadline :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = new DateTime();

        create_sut_using(() =>
           new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline));
    };

    Because of = () => result = sut.arrival_deadline();

    It should_give_back_the_arrival_deadline = () =>
       result.ShouldEqual(the_arrival_deadline);

    static DateTime result;
    static ILocation the_origin_location;
    static ILocation the_destination_location;
    static DateTime the_arrival_deadline;
}
```

Y ahora viene la pregunta, ¿si esta **BDD_Spec** es completamente análoga a las dos

anteriores, por que toda la ceremonia? ¿No hubiese sido más lógico haberla incluido en el epígrafe anterior o al menos pasar por ella rápidamente?

La respuesta podría ser afirmativa. Sin embargo hay un pequeño matiz que queríamos tratar. Un matiz desde la perspectiva del diseño.

2.5.1.1 .NET Datetime vs IDate

La dependencia (puesto que es inyectada en el constructor de `RouteSpecification`):

```
static DateTime the_arrival_deadline = new DateTime();
```

es una instancia de tipo `DateTime`, que es un tipo definido e implementado directamente por el **.NET Framework**. Y ahora viene el pero. Debemos estar seguros de que ese tipo (`DateTime`), define completamente a nuestra "fecha de entrega limite", ya que si no fuese así, la siguiente **BDD_Spec** sería mucho más deseable (reproducimos únicamente los bloques `IT` de la misma, para evitar distracciones):

```
public class when_asked_for_its_arrival_deadline :
             concern_for_route_specification
{
    It should_give_back_the_arrival_deadline = () =>
       result.ShouldEqual(the_arrival_deadline);

    It should_leverage_the_arrival_deadline_property = () =>
       the_arrival_deadline.received(x => x.my_method());
}
```

La razón habría que buscarla en que de esta forma delegaríamos la responsabilidad del acto de devolver nuestra "fecha de entrega limite" a un método (`my_method()`) perteneciente a su propia clase. O al menos, presupondríamos algún tipo de calculo o comprobación previa a devolver la "fecha de entrega limite", que en ningún caso debería realizarse en el objeto `RouteSpecification`.

Siguiendo este razonamiento, `the_arrival_deadline` pasaría a ser una instancia de un objeto tipo `IArrivalDealine` (que a su vez, pasaría a ser un **DDD_Value_Object**) y que básicamente empaquetaría internamente un tipo `DateTime`, exponiendo únicamente la funcionalidad necesaria del mismo y ocultando la complejidad no necesaria para

nuestro propósito, al menos, visto desde la perspectiva de `RouteSpecification`.

Es más, muy probablemente, en un sistema completo que gestionase la actividad de una empresa de paquetería, existiría un tipo `IDate` que empaquetaría a un objeto `DateTime`, siendo `IDate` el contrato ideal tanto para `IArrivalDeadline` como para un más que posiblemente necesario `IBookingDatetime`.

Éste es el pero. La razón por la que queríamos hacer hincapié en esta **BDD_Spec**, aparentemente básica.

Una vez hechos estos comentarios, y habiendo tomado inicialmente el camino más directo (usar `DateTime`), el resto del proceso para terminar la **BDD_Spec** es trivial.

2.5.2 Finalizando la BDD_Spec

Eliminamos el código de color rojo y obtenemos nuestro fallo:

```
public interface IRouteSpecification : ISpecification<IItinerary>,
                                       IValueObject<IRouteSpecification>
{
    ILocation origin();
    ILocation destination();
    DateTime arrival_deadline();
}

public class RouteSpecification : IRouteSpecification
{
    (....)

    public DateTime arrival_deadline()
    {
        throw new NotImplementedException();
    }

    (....)
}
```

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   when asked for its arrival deadline
   » should give back the arrival deadline (FAIL)

   Test 'should give back the arrival deadline' failed:
         System.Reflection.TargetInvocationException: Exception has been
```

```
   thrown by the target of an invocation. --->
   System.NotImplementedException: The method or operation is not
   implemented.

   (....)

   0 passed, 1 failed, 0 skipped, took 0,63 seconds (Machine.Specifications
   0.3.0).
```

Implementamos la funcionalidad y comprobamos que el test pasa:

```
public class RouteSpecification : IRouteSpecification
{
    readonly ILocation underlying_origin_location;
    readonly ILocation underlying_destination_location;
    readonly DateTime underlying_arrival_deadline;

    public RouteSpecification(ILocation the_origin_location,
                              ILocation the_destination_location,
                              DateTime the_arrival_deadline)
    {
        this.underlying_origin_location = the_origin_location;
        this.underlying_destination_location = the_destination_location;
        this.underlying_arrival_deadline = the_arrival_deadline;
    }

    public ILocation origin()
    {
        return this.underlying_origin_location;
    }

    public ILocation destination()
    {
        return this.underlying_destination_location;
    }

    public DateTime arrival_deadline()
    {
        return this.underlying_arrival_deadline;
    }
}
```

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   when asked for its arrival deadline
   » should give back the arrival deadline


   1 passed, 0 failed, 0 skipped, took 0,66 seconds (Machine.Specifications
   0.3.0).
```

Empaquetamos y a **github**.

2.6 RouteSpecificationSpecs - When asked if the specification is satisfied by an itinerary that satisfies the route specification

Vamos ahora con la **BDD_Spec** más interesante del **DDD_Value_Object** `IRouteSpecification`, la que nos va a permitir implementar el contrato fijado por `ISpecification<T>` y que asumimos al establecer que:

```
interface IRouteSpecification : ISpecification<IItinerary>
```

La **BDD_Spec**, sería ésta:

```
When asked if the specification is satisfied by an itinerary that
satisfies the route specification
   It should confirm that the itinerary satisfies the route
specification.
   It should leverage the itinerary initial departure location.
   It should leverage the origin location identity comparer.
   It should leverage the itinerary final arrival location.
   It should leverage the destination location identity comparer.
   It should leverage the itinerary final arrival date.
   It should check that the arrival deadline is met.
```

2.6.1 Consideraciones para la implementación

Ante una **BDD_Spec** de este tamaño, lo primero que debemos hacer es no asustarnos, ya que, toda la experiencia previa que hemos ido adquiriendo nos permitirá solucionar escenarios, al menos, tan complejos como éste. Y lo que es más importante; no hay una sola cosa que vayamos a necesitar para conseguir nuestro objetivo que no hayamos visto hasta ahora.

Nuestra segunda consideración tiene que ver con la discusión que expusimos en la **BDD_Spec** anterior. Y es que parece que vamos a necesitar un contrato propio para nuestra "fecha de entrega limite", ya que:

```
It should check that the arrival deadline is met.
```

nos deja muy poco margen.

Sin embargo, vamos a dejar la implementación de esa aserción para el final, de forma que empezaremos (esta vez sí, paso a paso) con el resto de las aserciones (o `Specifications` si usamos la nomenclatura propia del `Context / Specification`).

2.6.2 Implementando el bloque IT #1

Recordamos que el bloque `IT` que vamos a satisfacer viene definido por esta aserción:

> *It should confirm that the itinerary satisfies the route specification.*

Aparentemente, no debería suponer problema alguno, ya que hemos hecho cosas parecidas con anterioridad:

```
It should_confirm_that_the_itinerary_satisfies_the_route_specification = () =>
   result.ShouldBeTrue();
```

Si seguimos el camino de las baldosas rojas, nos obliga a definir `result` como:

```
static bool result;
```

Mientras que la ejecución del SUT a través de un bloque `BECAUSE` sería:

```
Because of = () =>
   result = sut.is_satisfied_by(
               the_itinerary_that_satisfies_the_route_specification);
```

Para eliminar el código de color rojo, necesitamos que:

```
static IItinerary the_itinerary_that_satisfies_the_route_specification;
```

Éste es un buen momento para recapitular y poder ver al completo el código que ya llevamos escrito:

```
public class when_asked_if_the_specification_is_satisfied_by
             _an_itinerary_that_satisfies_the_route_specification :
             concern_for_route_specification
```

```
{
    Because of = () =>
       result = sut.is_satisfied_by(
                    the_itinerary_that_satisfies_the_route_specification);

    It should_confirm_that_the_itinerary
       _satisfies_the_route_specification = () =>
       result.ShouldBeTrue();

    static bool result;
    static IItinerary the_itinerary_that_satisfies_the_route_specification;
}
```

Vamos ahora a establecer el contexto en el que se ejerce el SUT, a través de un bloque `ESTABLISH`:

```
Establish context = () =>
{
    the_origin_location = an<ILocation>();
    the_destination_location = an<ILocation>();
    the_arrival_deadline = new DateTime();

    create_sut_using(() =>
       new RouteSpecification(the_origin_location,
                              the_destination_location,
                              the_arrival_deadline));
};

static ILocation the_origin_location;
static ILocation the_destination_location;
static DateTime the_arrival_deadline;
```

No ha sido tan complicado. Por el momento no nos hemos encontrado con un solo elemento con el que no estuviésemos familiarizados.

2.6.2.1 Consideraciones

Es importante comentar que empezamos por la implementación de la **BDD_Spec** correspondiente al Happy Day Scenario:

```
When asked if the specification is satisfied by an itinerary that
satisfies the route specification
   It should confirm that the itinerary satisfies the route
specification.
```

```
(...)
```

Dentro de esta **BDD_Spec**, empezamos a su vez por la que se encarga del resultado de ejercitar el SUT, lo que nos permite definir el bloque `BECAUSE`, que sabemos será común a todos los demás bloques `IT` de la **BDD_Spec**.:

```
It should confirm that the itinerary satisfies the route specification.
```

Confirmamos que nuestra política de nombrado de métodos y variables es magnífica ya que, por poner un ejemplo, el bloque `BECAUSE` :

```
Because of = () =>
   result = sut.is_satisfied_by(
                the_itinerary_that_satisfies_the_route_specification);
```

se puede leer como la siguiente frase:

> "*Because of the system under test is satisfied by the itinerary that satisfies the route specification*"

que apenas deja lugar a la confusión, y cuando, como va a ser el caso de esta **BDD_Spec**, contamos con varias aserciones, es imprescindible entender en todo momento, de forma nada ambigua, qué es lo que estamos declarando con el código.

2.6.2.2 Refactorizando la creación del SUT

El código del bloque `ESTABLISH` se encarga única y exclusivamente de preparar el camino para construir el SUT y es el mismo en todas las **BDD_Specs** de nuestro **DDD_Value_Object** `IRouteSpecification`. Por lo tanto, al igual que hicimos en las **BDD_Specs** del **DDD_Aggregate_Root** `ICargo`, podemos refactorizarlo y enviarlo a `concern_for_route_specification`:

```
public abstract class concern_for_route_specification :
                      Observes<IRouteSpecification, RouteSpecification>
{
    Establish context = () =>
    {
```

```
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = new DateTime();

        create_sut_using(() =>
           new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline));
    };

    static ILocation the_origin_location;
    static ILocation the_destination_location;
    static DateTime the_arrival_deadline;
}
```

2.6.2.3 Consecuencias de la refactorización

Un cambio así hace que tengamos que refactorizar las **BDD_Specs** ya existentes, eliminando el código repetido.

Así, las tres **BDD_Specs** creadas en los epígrafes anteriores quedarían tal que así:

```
When asked for its origin location
   It should give back the origin location.
```

```
public class when_asked_for_its_origin_location :
             concern_for_route_specification
{
    Because of = () => result = sut.origin();

    It should_give_back_the_origin_location = () =>
       result.ShouldEqual(the_origin_location);

    static ILocation result;
}
```

```
When asked for its destination location
   It should give back the destination location.
```

```
public class when_asked_for_its_destination_location :
             concern_for_route_specification
{
    Because of = () => result = sut.destination();
```

```
    It should_give_back_the_destination_location = () =>
       result.ShouldEqual(the_destination_location);

    static ILocation result;
}
```

```
When asked for its arrival deadline
   It should give back the arrival deadline.
```

```
public class when_asked_for_its_arrival_deadline :
             concern_for_route_specification
{
    Because of = () => result = sut.arrival_deadline();

    It should_give_back_the_arrival_deadline = () =>
       result.ShouldEqual(the_arrival_deadline);

    static DateTime result;
}
```

Antes de proseguir con nuestra **BDD_Spec**, necesitamos confirmar que con esta refactorización no hemos roto nada.

Para ello ejecutamos nuestra suite de **BDD_Specs**:

```
------ Test started: Assembly: dddsample.specs.dll ------

(....)

when asked for its origin location
» should give back the origin location

when asked for its destination location
» should give back the destination location

when asked for its arrival deadline
» should give back the arrival deadline


15 passed, 0 failed, 0 skipped, took 1,18 seconds (Machine.Specifications
0.3.0).
```

Y podemos estar satisfechos ya que todo va bien.

> **NOTA**: Tenemos 15 test correctos porque ejecutamos todos los tests que hay en `dddsample.specs.dll`. Por lo tanto estamos ejecutando tanto los relativos al **DDD_Aggregate_Root** `ICargo` como los del **DDD_Value_Object** `IRouteSpecification`. Aun así en el log solo mostramos los detalles de los que nos interesan.

2.6.2.4 RED-GREEN

Tras esta pequeña deriva, vamos a continuar con nuestra **BDD_Spec**, de la que, recordemos:

- Tenemos definida la aserción a través del bloque `IT`.
- Tenemos definida la ejecución del SUT a través del bloque `BECAUSE`.
- Tenemos definido el contexto a través de la **clase base abstracta** `concern_for_route_specification` y su bloque `ESTABLISH`.

Por lo tanto, siguiendo con el proceso, deberíamos poder obtener nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
(FAIL)

Test 'should confirm that the itinerary satisfies the route specification'
failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

0 passed, 1 failed, 0 skipped, took 0,75 seconds (Machine.Specifications
0.3.0).
```

Y de hecho, lo obtenemos.

El siguiente paso debe ser, escribir el código que nos permita pasar el test.

A saber:

```
public class RouteSpecification : IRouteSpecification
{
    public bool is_satisfied_by(IItinerary the_itinerary)
    {
        return true;
    }
}
```

que efectivamente nos asegura pasar el test, como prueba el siguiente log:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification


1 passed, 0 failed, 0 skipped, took 0,66 seconds (Machine.Specifications
0.3.0).
```

Todo funciona tal y como esperábamos.

2.6.2.5 Entendiendo la solución alcanzada

Es importante entender que la implementación de `is_satisfied_by()` devuelve `true` siempre:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    return true;
}
```

Esto nos permite pasar el test, pero claramente no estamos haciendo lo que queremos (comprobar que el `IItinerary` satisface la `IRouteSpecification`).

La razón para volver a este tan primitivo:

> "*escribir únicamente el mínimo código necesario para que el test pase*"

propio del TDD, hay que situarla en la complejidad de la **BDD_Spec** que estamos tratando.

Esto es algo que ya comentamos con anterioridad, cuando las **BDD_Specs** sean relativamente sencillas podemos aplicar el:

> "*escribir únicamente el mínimo código necesario Y QUE ADEMÁS, TENGA SENTIDO, para que el test pase*"

Pero cuando nos encontremos con escenarios más complejos, la vuelta a los orígenes nos va a ayudar a no perdernos por los vericuetos de la implementación.

De esta forma podremos centrarnos en ir paso a paso satisfaciendo los distintas aserciones.

2.6.3 Implementando el bloque IT #2

Recordamos que el bloque `IT` que vamos a satisfacer viene definido por esta aserción:

```
It should leverage the itinerary initial departure location.
```

Este estilo de aserciones son el paradigma del **BDD** y de su primo hermano el TDD Mockista o Interaction-Based Testing TDD.

Vamos con su versión en código:

```
It should_leverage_the_itinerary_initial_departure_location = () =>
   the_itinerary_that_satisfies_the_route_specification
      .received(x => x.initial_departure_location());
```

Esto provoca una **mini reacción en cadena**, obligándonos a definir el método en la interface `IItinerary` para eliminar el código rojo:

```
public interface IItinerary
{
    ILocation initial_departure_location();
```

```
}
```

Si recapacitamos un poco, lo único que estamos haciendo es diseñar como queremos que se comporte nuestro programa, a través de la delegación de responsabilidades a quién corresponda. En nuestro caso, para saber cual es el punto de origen del itinerario, delegamos en un objeto `IItinerary`. En otras palabras, de una forma casi innata, estamos cumpliendo con el **SRP**.

Antes de obtener nuestro fallo, necesitamos definir `the_itinerary_satisfies_the_route_specification` como un mock y asignarle el comportamiento deseado.

Para ello nos valemos de un bloque `ESTABLISH`:

```
Establish context = () =>
{
    the_itinerary_that_satisfies_the_route_specification = an<IItinerary>();
    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.initial_departure_location())
       .Return(an<ILocation>());
};
```

Con lo que ya estamos listos para obtener nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location (FAIL)


Test 'should leverage the itinerary initial departure location' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException:
IItinerary.initial_departure_location(); Expected #1..2147483647, Actual
#0.
      at Rhino.Mocks.RhinoMocksExtensions.AssertWasCalled[T](T mock,
Action`1 action, Action`1 setupConstraints)
      at
Machine.Specifications.DevelopWithPassion.Rhino.VoidMethodCallOccurance`1.
.ctor(T mock, Action`1 action)
      at
Machine.Specifications.DevelopWithPassion.Rhino.RhinoMocksExtensions.recei
```

```
ved[T](T mock, Action`1 item)
      domain\model\cargo.aggregate\RouteSpecificationSpecs.cs(68,0): at
dddsample.specs.domain.model.cargo.aggregate.when_asked_if_the_specificati
on_is_satisfied_by_an_itinerary_that_satisfies_the_route_specification.<.c
tor>b__4()
      at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
      at Machine.Specifications.Model.Specification.Verify()

1 passed, 1 failed, 0 skipped, took 1,05 seconds (Machine.Specifications
0.3.0).
```

Podemos comprobar que nuestro test anterior sigue pasando:

```
» should confirm that the itinerary satisfies the route specification
```

Mientras que nuestro test actual no pasa:

```
» should leverage the itinerary initial departure location (FAIL)
```

Y además nos da el fallo adecuado:

```
Rhino.Mocks.Exceptions.ExpectationViolationException:
IItinerary.initial_departure_location(); Expected #1..2147483647, Actual
#0.
```

Es decir, Rhino.Mocks esperaba que se produjese 1 llamada:

```
Expected #1
```

a `initial_departure_location()` en `IItinerary`.

Pero en realidad no se ha producido ninguna:

```
Actual #0
```

El siguiente paso es escribir el código que nos permita pasar el test.

No tiene mayor complicación:

```
public class RouteSpecification : IRouteSpecification
```

```
{
    (....)

    public bool is_satisfied_by(IItinerary the_itinerary)
    {
       the_itinerary.initial_departure_location();
       return true;
    }

    (....)
}
```

Comprobamos nuevamente con nuestra suite de **BDD_Specs**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that satisfies the
route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location


2 passed, 0 failed, 0 skipped, took 0,80 seconds (Machine.Specifications
0.3.0).
```

Todo va bien.

Vamos con el siguiente bloque `IT`.

2.6.4 Implementando el bloque IT #3

Recordamos que el bloque `IT` que vamos a satisfacer viene definido por esta aserción:

> *It should leverage the origin location identity comparer.*

Ésta es otra aserción de **diseño**, puesto que volvemos a delegar.

Su versión en código es muy interesante, pues pone en funcionamiento lo escrito en la aserción anterior:

```
It should_leverage_the_origin_location_identity_comparer = () =>
```

```
   the_origin_location
      .received(x => x
      .has_the_same_value_as(
         the_itinerary_that_satisfies_the_route_specification
            .initial_departure_location()));
```

Debido a la limitación de espacio inherente a este texto, la sangría puede confundir a más de uno. Lo único que estamos afirmando en este código es que `the_origin_location` va a recibir una llamada a su método `has_the_same_value_as()` con el argumento `the_itinerary_that_satisfies_the_route_specification.initial_departure_location()`. Ni más ni menos.

Para que el código de color rojo deje de tener ese color, necesitamos definir la naturaleza de `ILocation`. Y ésta no es otra que la de **DDD_Value_Object**, con lo que solo necesitamos implementar la interface `IValueObject<T>`:

```
public interface ILocation : IValueObject<ILocation> {}
```

Nueva constatación de algo que hemos repetido hasta la saciedad, con el **BDD** no escribimos código, con el **BDD** diseñamos la aplicación a través del código.

Exactamente igual que con el bloque `IT` anterior, antes de obtener nuestro fallo, necesitamos definir `the_origin_location` como un mock y asignarle el comportamiento deseado.

En este caso `the_origin_location` ya ha sido definido previamente como un mock a través de la **clase base abstracta** `concern_for_route_specification`:

```
the_origin_location = an<ILocation>();
```

y debemos recordar, además, que es una dependencia de nuestro SUT:

```
create_sut_using(() =>
   new RouteSpecification(the_origin_location,
                          the_destination_location,
                          the_arrival_deadline));
```

Para asignarle el comportamiento deseado, nos valemos de un bloque `ESTABLISH`:

```
Establish context = () =>
{
    the_itinerary_that_satisfies_the_route_specification = an<IItinerary>();
    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.initial_departure_location())
       .Return(an<ILocation>());

    the_origin_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .initial_departure_location()))
       .Return(true);
};
```

que nos permite obtener nuestro fallo sin afectar al resto de las aserciones ya codificadas:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer (FAIL)


Test 'should leverage the origin location identity comparer' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException:
IValueObject`1.has_the_same_value_as(ILocationProxy075bbe2c85fe451390e963c
1ae35e4d4); Expected #1..2147483647, Actual #0.

(....)

2 passed, 1 failed, 0 skipped, took 0,86 seconds (Machine.Specifications
0.3.0).
```

Es decir, esperábamos una llamada a `has_the_same_value_as()`:

```
Expected #1
```

pero ésta no se produce:

```
Actual #0
```

Para remediarlo necesitamos implementar el comportamiento descrito en nuestro **DDD_Value_Object** `RouteSpecification`, tal que así:

```
public class RouteSpecification : IRouteSpecification
{
    readonly ILocation underlying_origin_location;
    readonly ILocation underlying_destination_location;
    readonly DateTime underlying_arrival_deadline;

    public RouteSpecification(ILocation the_origin_location,
                              ILocation the_destination_location,
                              DateTime the_arrival_deadline)
    {
        this.underlying_origin_location = the_origin_location;
        this.underlying_destination_location = the_destination_location;
        this.underlying_arrival_deadline = the_arrival_deadline;
    }

    (....)

    public bool is_satisfied_by(IItinerary the_itinerary)
    {
        underlying_origin_location
           .has_the_same_value_as(the_itinerary.initial_departure_location());

        return true;
    }
}
```

De esta forma provocamos que el test pase:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer


3 passed, 0 failed, 0 skipped, took 0,78 seconds (Machine.Specifications
0.3.0).
```

Esto ya va tomando forma.

2.6.5 Implementando el bloque IT #4

Recordamos que el bloque `IT` que vamos a satisfacer viene definido por esta aserción:

> *It should leverage the itinerary final arrival location.*

Se trata de una situación análoga a **Implementando el bloque IT #2**. Por lo tanto, todo lo que era aplicable en ese caso, lo es también en éste. De nuevo más **diseño**.

El código de la aserción es:

```
It should_leverage_the_itinerary_final_arrival_location = () =>
   the_itinerary_that_satisfies_the_route_specification
      .received(x => x.final_arrival_location());
```

Que provoca la **reacción en cadena**:

```
public interface IItinerary
{
    ILocation initial_departure_location();
    ILocation final_arrival_location();
}
```

solucionando, con ello, el código en rojo.

En el bloque `ESTABLISH` definimos el comportamiento esperado para el mock (las últimas tres lineas de código):

```
Establish context = () =>
{
    the_itinerary_that_satisfies_the_route_specification = an<IItinerary>();
    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.initial_departure_location())
       .Return(an<ILocation>());

    the_origin_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .initial_departure_location()))
       .Return(true);
```

```
    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.final_arrival_location())
       .Return(an<ILocation>());
};
```

Fijémonos en la similitud con el **Implementando el bloque IT #2**:

```
el bloque IT #2


    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.initial_departure_location())
       .Return(an<ILocation>());


el bloque IT #4


    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.final_arrival_location())
       .Return(an<ILocation>());
```

Vamos con nuestro fallo del tipo:

```
  Expected #1, Actual #0
```

que obtenemos sin problemas:

```
  ------ Test started: Assembly: dddsample.specs.dll ------

  when asked if the specification is satisfied by an itinerary that
  satisfies the route specification
  » should confirm that the itinerary satisfies the route specification
  » should leverage the itinerary initial departure location
  » should leverage the origin location identity comparer
  » should leverage the itinerary final arrival location (FAIL)


  Test 'should leverage the itinerary final arrival location' failed:
        Rhino.Mocks.Exceptions.ExpectationViolationException:
  IItinerary.final_arrival_location(); Expected #1..2147483647, Actual #0.

  (....)

  3 passed, 1 failed, 0 skipped, took 0,93 seconds (Machine.Specifications
  0.3.0).
```

Escribimos el código necesario para que el test pase:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    underlying_origin_location
       .has_the_same_value_as(the_itinerary.initial_departure_location());

    the_itinerary.final_arrival_location();

    return true;
}
```

Y comprobamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location


4 passed, 0 failed, 0 skipped, took 0,86 seconds (Machine.Specifications
0.3.0).
```

Listo. Una menos.

2.6.6 Implementando el bloque IT #5

Recordamos que el bloque `IT` que vamos a satisfacer viene definido por esta aserción:

> *It should leverage the destination location identity comparer.*

Análoga a **Implementando el bloque IT #3**.

Empezamos por el bloque `IT`:

```
It should_leverage_the_destination_location_identity_comparer = () =>
   the_destination_location
```

```
      .received(x => x
      .has_the_same_value_as(
         the_itinerary_that_satisfies_the_route_specification
            .final_arrival_location()));
```

Que en realidad simplemente establece que `the_destination_location` va a recibir una llamada a su método `has_the_same_value_as()` con el argumento `the_itinerary_that_satisfies_the_route_specification.final_arrival_location()`.

El comportamiento del mock, queda definido en el bloque `ESTABLISH` (las últimas cuatro lineas de código):

```
Establish context = () =>
{
    the_itinerary_that_satisfies_the_route_specification = an<IItinerary>();
    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.initial_departure_location())
       .Return(an<ILocation>());

    the_origin_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .initial_departure_location()))
       .Return(true);

    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.final_arrival_location())
       .Return(an<ILocation>());

    the_destination_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .final_arrival_location()))
       .Return(true);
};
```

Nuevamente, resulta interesante señalar la similitud con el **Implementando el bloque IT #3**:

```
el bloque IT #3
```

```
    the_origin_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .initial_departure_location()))
```

```
        .Return(true);
```

el bloque IT #5

```
    the_destination_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .final_arrival_location()))
       .Return(true);
```

Vamos a por nuestro fallo:

```
   Expected #1, Actual #0

   ------ Test started: Assembly: dddsample.specs.dll ------

   when asked if the specification is satisfied by an itinerary that
   satisfies the route specification
   » should confirm that the itinerary satisfies the route specification
   » should leverage the itinerary initial departure location
   » should leverage the origin location identity comparer
   » should leverage the itinerary final arrival location
   » should leverage the destination location identity comparer (FAIL)


   Test 'should leverage the destination location identity comparer' failed:
         Rhino.Mocks.Exceptions.ExpectationViolationException:
   IValueObject`1.has_the_same_value_as(ILocationProxy24c923ecb3a441929f6b062
   70468b877); Expected #1..2147483647, Actual #0.

   (....)


   4 passed, 1 failed, 0 skipped, took 1,32 seconds (Machine.Specifications
   0.3.0).
```

Y la solución al mismo:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    underlying_origin_location
       .has_the_same_value_as(the_itinerary.initial_departure_location());

    underlying_destination_location
       .has_the_same_value_as(the_itinerary.final_arrival_location());
```

```
    return true;
}
```

Ahora nuestro test pasa:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer


5 passed, 0 failed, 0 skipped, took 0,82 seconds (Machine.Specifications
0.3.0).
```

Esto va rápido. A por otra.

2.6.7 Implementando el bloque IT #6

Recordamos que el bloque `IT` que vamos a satisfacer viene definido por esta aserción:

> *It should leverage the itinerary final arrival date.*

Análoga a **Implementando el bloque IT #2** y a **Implementando el bloque IT #4**.

Sin más dilación, el bloque `IT`:

```
It should_leverage_the_itinerary_final_arrival_date = () =>
   the_itinerary_that_satisfies_the_route_specification
      .received(x => x.final_arrival_date());
```

Que nos obliga, para eliminar el código rojo, a:

```
public interface IItinerary
{
    ILocation initial_departure_location();
    ILocation final_arrival_location();
    DateTime final_arrival_date();
```

```
}
```

Definimos el comportamiento del mock (las tres última lineas):

```
Establish context = () =>
{
    the_itinerary_that_satisfies_the_route_specification = an<IItinerary>();
    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.initial_departure_location())
       .Return(an<ILocation>());

    the_origin_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .initial_departure_location()))
       .Return(true);

    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.final_arrival_location())
       .Return(an<ILocation>());

    the_destination_location
       .Stub(x => x.has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .final_arrival_location()))
       .Return(true);

    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.final_arrival_date())
       .Return(new DateTime());
};
```

Nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer
» should leverage the itinerary final arrival date (FAIL)


Test 'should leverage the itinerary final arrival date' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException:
```

```
IItinerary.final_arrival_date(); Expected #1..2147483647, Actual #0.

(....)

5 passed, 1 failed, 0 skipped, took 1,01 seconds (Machine.Specifications
0.3.0).
```

La implementación de la funcionalidad:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    underlying_origin_location
       .has_the_same_value_as(the_itinerary.initial_departure_location());

    underlying_destination_location
       .has_the_same_value_as(the_itinerary.final_arrival_location());

    the_itinerary.final_arrival_date();

    return true;
}
```

Y la prueba de que todo ha ido tal y como esperábamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer
» should leverage the itinerary final arrival date


6 passed, 0 failed, 0 skipped, took 0,80 seconds (Machine.Specifications
0.3.0).
```

Vamos con el último bloque `IT`.

2.6.8 Implementando el bloque IT #7

Recordamos que el bloque `IT` que vamos a satisfacer viene definido por esta aserción:

```
It should check that the arrival deadline is met.
```

Considerando como hemos definido

```
DateTime the_arrival_deadline
```

no podemos comprobar la aserción.

Esto debería hacer que nos planteásemos, al menos, las dos siguientes preguntas:

Primera: ¿Por que no podemos comprobarlo?.

Segunda: Si no podemos comprobarlo, ¿como es que teníamos ese requisito?, ese:

```
It should check that the arrival deadline is met
```

Podemos responder a la primera pregunta de una forma técnica.

Para poder poder comprobar una interacción con una dependencia, necesitamos mockar la dependencia y luego establecer tanto el comportamiento esperado del mock como la aserción donde se defina qué es lo que queremos que ocurra.

Puesto que `DateTime` es un tipo definido por el **.NET Framework**, del que no existe una `IDateTime`, no podemos mockarlo con Rhino.Mocks.

Existen formas alternativa para intentar hacer esto, pero son **hacks** y eso es precisamente lo que no queremos.

2.6.8.1 Detectando un error de diseño

Para responder a la segunda pregunta tenemos que ir a la raíz misma del **BDD**.

Desde ese punto de vista, podemos considerar que lo que pretendemos es establecer la necesidad de que, de alguna forma, comprobemos que:

Para que se cumpla la condición (léase `is_satisfied_by()` devuelva `true`), necesitamos establecer una condición que se aplica al `arrival_deadline`.

Hasta ahora, al haber diseñado el sistema tal y como queríamos (teniendo por ello que alejarnos del código java del ejemplo original), habíamos usado interfaces para ayudarnos a cumplir el **SRP**, lo que, indirectamente nos daba la oportunidad de mockar las dependencias.

Pero tal y como comentábamos a la hora de implementar la **BDD_Spec**:

```
When asked for its arrival deadline
```

modelamos el concepto asociado a "la fecha limite de entrega" a través de un objeto `DateTime`, en lugar de crear nuestra propia interface del estilo a `IArrivalDeadline` que nos permitiese encapsular la funcionalidad que nos interesa del tipo `DateTime`.

Esa decisión (qué es la decisión que tomaron en el ejemplo original en java), desde un punto exclusivamente orientado por el **BDD**, ya no nos pareció del todo correcta. Y ahora se confirma que no es una buena decisión de diseño, si queremos que el **BDD** guíe dicho diseño.

> **NOTA**: Consultar la discusión inicial de la **BDD_Spec When asked for its arrival deadline**, donde mostramos nuestras reticencias a la hora de usar `Datetime` directamente.

En realidad podemos ver esta necesidad de cambio como una **reacción en cadena**, ya que, aunque no la provoca un bloque de código de color rojo, si nos paramos a pensar en el significado de un bloque de color rojo, no es otro que una funcionalidad que todavía no está disponible, pero que necesitamos para que nuestro sistema funcione.

Por lo tanto, ahora, el bloque de color rojo, nos viene de más arriba. Nos viene de la **BDD_Spec** en si, ya que hay una `Specification` (siguiendo la nomenclatura `Context / Specification`) que se ha puesto de color rojo.

Al final, es nuestro **diseño** el que se pone de un color rojo alarmante.

Procedamos con el cambio.

2.6.8.2 Corrigiendo un error de diseño

Vamos a intentar detallar el proceso paso a paso para limitar los daños colaterales.

Primero

Crear la interface `IArrivalDeadline`:

```
public interface IArrivalDeadline {}
```

Segundo

Darle sentido dentro de un entorno **DDD**.

En este caso sería un **DDD_Value_Object**:

```
public interface IArrivalDeadline : IValueObject<IArrivalDeadline> {}
```

Con estos dos simples pasos, ya habríamos creado el tipo del objeto.

Tercero

Cambiar el constructor de `RouteSpecification`, para que admita un `IArrivalDeadline` en lugar de un `Datetime` como argumento.

En este tercer paso, vamos a echar mano de la magnifica capacidad que tiene **ReSharper** para refactorizar código:

```
Ctrl + F6 sobre el constructor.
```

Te permite cambiar la signatura del mismo:

```
public RouteSpecification(ILocation the_origin_location,
                          ILocation the_destination_location,
                          IArrivalDeadline the_arrival_deadline)
```

Cuarto

Dentro de `RouteSpecification`, cambiar el tipo del campo `underlying_arrival_deadline` de `Datetime` a `IArrivalDeadline`:

```
readonly IArrivalDeadline underlying_arrival_deadline;
```

Quinto

Dentro de `IRouteSpecification`, cambiar la signatura del método `arrival_deadline()`.

Otra vez haciendo uso de **ReSharper:**

```
Ctrl + F6
```

Para poder lograr:

```
IArrivalDeadline arrival_deadline();
```

y además, de rebote, obtenemos en `RouteSpecification`:

```
public IArrivalDeadline arrival_deadline()
{
    return this.underlying_arrival_deadline;
}
```

En este momento, `IRouteSpecification` y `RouteSpecification`, no tienen ya ninguna dependencia de `DateTime`, sino que dependen de `IArrivalDeadline`.

Sexto

En `RouteSpecificationSpecs` buscamos cuales son los puntos que ya no compilan.

ReSharper al rescate:

```
Alt + F12
```

Para poder detectar y saltar entre estos puntos.

Encontramos 2 puntos.

Para solucionar el problema, basta con cambiar el tipo `DateTime` por `IArrivalDeadline`.

Tanto en el primer punto:

```
public abstract class concern_for_route_specification :
                      Observes<IRouteSpecification, RouteSpecification>
{
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = an<IArrivalDeadline>();

        create_sut_using(() =>
           new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline));
    };

    protected static ILocation the_origin_location;
    protected static ILocation the_destination_location;
    protected static IArrivalDeadline the_arrival_deadline;
}
```

como en el segundo:

```
public class when_asked_for_its_arrival_deadline :
             concern_for_route_specification
{
    Because of = () => result = sut.arrival_deadline();

    It should_give_back_the_arrival_deadline = () =>
       result.ShouldEqual(the_arrival_deadline);

    static IArrivalDeadline result;
}
```

Séptimo

Ejecutamos la suite de **BDD_Specs**, para comprobar si hemos roto algo:

```
21 passed, 0 failed, 0 skipped, took 1,93 seconds (Machine.Specifications
0.3.0).
```

Octavo

Hacemos una última comprobación para ver que no nos dejamos ningún `DateTime` atrás.

Vamos a por las interfaces que representan a los colaboradores de `RouteSpecification`. Es decir:

- `ILocation`
- `IItinerary`

Descubrimos una referencia perdida en `IItinerary` que corregimos:

```
IArrivalDeadline final_arrival_date();
```

Noveno

Comprobamos los posibles problemas derivados del paso anterior.

Encontramos uno en `RouteSpecificationSpecs`, que corregimos.

```
the_itinerary_that_satisfies_the_route_specification
   .Stub(x => x.final_arrival_date())
   .Return(an<IArrivalDeadline>());
```

Décimo

Volvemos a ejecutar la suite de **BDD_Specs**:

```
21 passed, 0 failed, 0 skipped, took 1,11 seconds (Machine.Specifications
0.3.0).
```

Asombroso, utilizar Interfaces en combinación con **BDD**, merece la pena.

Todo funciona correctamente.

Podemos seguir **Implementando el bloque IT #7**.

2.6.9 Implementando el bloque IT #7 (Continuación)

Podríamos reescribir la `Specification` como:

> *It should leverage the arrival deadline time check.*

Que a su vez traduciríamos en código:

```
It should_leverage_the_arrival_deadline_time_check = () =>
   the_arrival_deadline
      .received(x => x
      .is_afterwards_than(
         the_itinerary_that_satisfies_the_route_specification
            .final_arrival_date()));
```

Estaríamos estableciendo que `the_arrival_deadline` va a recibir una llamada a su método `is_afterwards_than()` con el argumento `the_itinerary_that_satisfies_the_route_specification.final_arrival_date()`.

Eliminamos el rojo con:

```
public interface IArrivalDeadline : IValueObject<IArrivalDeadline>
{
    bool is_afterwards_than(IArrivalDeadline final_arrival_date);
}
```

Y definimos el mock como (las últimas cuatro lineas de código):

```
Establish context = () =>
{
    the_itinerary_that_satisfies_the_route_specification = an<IItinerary>();
    the_itinerary_that_satisfies_the_route_specification
       .Stub(x => x.initial_departure_location())
       .Return(an<ILocation>());

    the_origin_location
       .Stub(x => x.has_the_same_value_as(
```

```
            the_itinerary_that_satisfies_the_route_specification
               .initial_departure_location()))
         .Return(true);

      the_itinerary_that_satisfies_the_route_specification
         .Stub(x => x.final_arrival_location())
         .Return(an<ILocation>());

      the_destination_location
         .Stub(x => x.has_the_same_value_as(
            the_itinerary_that_satisfies_the_route_specification
               .final_arrival_location()))
         .Return(true);

      the_itinerary_that_satisfies_the_route_specification
         .Stub(x => x.final_arrival_date())
         .Return(an<IArrivalDeadline>());

      the_arrival_deadline
         .Stub(x => x.is_afterwards_than(
            the_itinerary_that_satisfies_the_route_specification
               .final_arrival_date()))
         .Return(true);
};
```

Confirmamos nuestro fallo, que debe ser del tipo:

```
Expected #1, Actual #0


------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer
» should leverage the itinerary final arrival date
» should leverage the arrival deadline time check (FAIL)


Test 'should leverage the arrival deadline time check' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException:
IArrivalDeadline.is_afterwards_than(IArrivalDeadlineProxya5d44cac2d54404ca
256d3e582c1b926); Expected #1..2147483647, Actual #0.

(....)

6 passed, 1 failed, 0 skipped, took 0,95 seconds (Machine.Specifications
```

```
  0.3.0).
```

Escribimos el código que hace que pase el test:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    underlying_origin_location
       .has_the_same_value_as(the_itinerary.initial_departure_location());

    underlying_destination_location
       .has_the_same_value_as(the_itinerary.final_arrival_location());

    underlying_arrival_deadline
       .is_afterwards_than(the_itinerary.final_arrival_date());

    return true;
}
```

Comprobamos que el test pasa:

```
  ------ Test started: Assembly: dddsample.specs.dll ------

  when asked if the specification is satisfied by an itinerary that
  satisfies the route specification
  » should confirm that the itinerary satisfies the route specification
  » should leverage the itinerary initial departure location
  » should leverage the origin location identity comparer
  » should leverage the itinerary final arrival location
  » should leverage the destination location identity comparer
  » should leverage the itinerary final arrival date
  » should leverage the arrival deadline time check


  7 passed, 0 failed, 0 skipped, took 0,84 seconds (Machine.Specifications
  0.3.0).
```

Ya lo tenemos, nuestras 7 aserciones pasan.

2.6.10 ¿Era esto lo que buscábamos?

Ya hemos conseguido implementar la **BDD_Spec** completa, pero sin embargo no hemos obtenido la funcionalidad buscada.

Consideremos, para ello, el código que la implementa:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    underlying_origin_location
       .has_the_same_value_as(the_itinerary.initial_departure_location());

    underlying_destination_location
       .has_the_same_value_as(the_itinerary.final_arrival_location());

    underlying_arrival_deadline
       .is_afterwards_than(the_itinerary.final_arrival_date());

    return true;
}
```

Las interacciones con los objetos colaboradores se producen (de hecho los tests pasan), pero no se tienen en cuenta a la hora de obtener el resultado final (devolvemos siempre `true`).

La razón por la que ocurre esto es muy sencilla y debemos recordar que ya nos encontramos con un escenario parecido implementando las **BDD_Specs** de `Cargo`. Necesitamos más **BDD_Specs** que nos obliguen a llegar al código que buscamos.

Probablemente, si hubiésemos seguido la máxima de:

> "*escribir únicamente el mínimo código necesario Y QUE ADEMÁS, TENGA SENTIDO, para que el test pase*"

en vez de la que usamos

> "*escribir únicamente el mínimo código necesario para que el test pase*"

podríamos haber llegado ya a la solución buscada, pero (y éste es un pero enorme), probablemente no hubiésemos conseguido resolver cada una de las aserciones (bloque `IT`), con la facilidad con la que lo hemos hecho.

Sea como fuere, vamos a obligar al código a que haga lo que queremos. Para ello implementaremos la siguiente **BDD_Spec**.

2.7 RouteSpecificationSpecs - When asked if the specification is satisfied by an itinerary with an invalid initial departure location

Estaríamos hablando, por tanto, de la cuestión opuesta a la descrita en la **BDD_Spec** anterior.

Sin entrar en el debate TDD Mockista vs TDD Tradicional del que ya dimos cuenta mientras implementábamos un escenario parecido en `CargoSpecs`, la **BDD_Spec** sería la siguiente:

```
When asked if the specification is satisfied by an itinerary with an
invalid initial departure location
   It should confirm that the itinerary does not satisfy the route
specification.
   It should leverage the itinerary initial departure location.
   It should leverage the origin location identity comparer.
   It should not leverage the itinerary final arrival location.
   It should not leverage the destination location identity comparer.
   It should not leverage the itinerary final arrival date.
   It should not leverage the arrival deadline time check.
```

Cuyo código correspondería a:

```
public class when_asked_if_the_specification_is_satisfied_by
             _an_itinerary_with_an_invalid_initial_departure_location :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_itinerary_with_an_invalid_initial_departure_location =
           an<IItinerary>();

        the_itinerary_with_an_invalid_initial_departure_location
           .Stub(x => x.initial_departure_location())
           .Return(an<ILocation>());

        the_origin_location
           .Stub(x => x.has_the_same_value_as(
              the_itinerary_with_an_invalid_initial_departure_location
                 .initial_departure_location()))
           .Return(false);
    };
```

```
    Because of = () =>
       result = sut.is_satisfied_by(
                    the_itinerary_with_an_invalid_initial_departure_location);

    It should_confirm_that_the_itinerary
       _does_not_satisfy_the_route_specification = () =>
       result.ShouldBeFalse();

    It should_leverage_the_itinerary_initial_departure_location = () =>
       the_itinerary_with_an_invalid_initial_departure_location
          .received(x => x.initial_departure_location());

    It should_leverage_the_origin_location_identity_comparer = () =>
       the_origin_location
          .received(x => x
          .has_the_same_value_as(
             the_itinerary_with_an_invalid_initial_departure_location
                .initial_departure_location()));

    It should_not_leverage_the_itinerary_final_arrival_location = () =>
       the_itinerary_with_an_invalid_initial_departure_location
          .never_received(x => x.final_arrival_location());

    It should_not_leverage_the_destination_location_identity_comparer = () =>
       the_destination_location
          .never_received(x => x
          .has_the_same_value_as(
             the_itinerary_with_an_invalid_initial_departure_location
                .final_arrival_location()));

    It should_not_leverage_the_itinerary_final_arrival_date = () =>
       the_itinerary_with_an_invalid_initial_departure_location
          .never_received(x => x.final_arrival_date());

    It should_not_leverage_the_arrival_deadline_time_check = () =>
       the_arrival_deadline
          .never_received(x => x
          .is_afterwards_than(
             the_itinerary_with_an_invalid_initial_departure_location
                .final_arrival_date()));

    static bool result;
    static IItinerary the_itinerary_with_an_invalid_initial_departure_location;
}
```

No deberíamos tener demasiados problemas para poder seguir el código que acabamos de escribir, si habíamos entendido la implementación de la **BDD_Spec** anterior.

2.7.1 Breve descripción del código escrito

2.7.1.1 El contexto

Lo único que estamos estableciendo es un entorno simulado donde tenemos un:

```
itinerary_with_an_invalid_initial_departure_location
```

Por lo que cuando se produzca la comprobación en `RouteSpecification`:

```
underlying_origin_location
   .has_the_same_value_as(the_itinerary.initial_departure_location())
```

queremos que el resultado sea:

```
.Return(false);
```

Y es precisamente ese contexto el que establecemos en el bloque `ESTABLISH`.

2.7.1.2 Las aserciones

Con respecto a las aserciones (bloques `IT`), podemos comentar lo siguiente:

Esperamos que no se satisfaga la `ISpecification<IItinerary>`:

```
It should_confirm_that_the_itinerary
   _does_not_satisfy_the_route_specification = () =>
   result.ShouldBeFalse();
```

Esperamos que dentro de `RouteSpecification`, se compruebe que efectivamente la condición que queremos que falle, falle de verdad:

```
It should_leverage_the_itinerary_initial_departure_location = () =>
   the_itinerary_with_an_invalid_initial_departure_location
      .received(x => x.initial_departure_location());

It should_leverage_the_origin_location_identity_comparer = () =>
   the_origin_location
      .received(x => x
```

```
      .has_the_same_value_as(
         the_itinerary_with_an_invalid_initial_departure_location
            .initial_departure_location()));
```

De ahí los:

```
.received
```

El resto de las condiciones, no deben verificarse, ya que ya ha fallado la primera. Por lo tanto da igual lo que ocurra con esas condiciones ya que la `ISpecification<IItinerary>` no se va a cumplir:

```
It should_not_leverage_the_itinerary_final_arrival_location = () =>
   the_itinerary_with_an_invalid_initial_departure_location
      .never_received(x => x.final_arrival_location());

It should_not_leverage_the_destination_location_identity_comparer = () =>
   the_destination_location
      .never_received(x => x
      .has_the_same_value_as(
         the_itinerary_with_an_invalid_initial_departure_location
            .final_arrival_location()));

It should_not_leverage_the_itinerary_final_arrival_date = () =>
   the_itinerary_with_an_invalid_initial_departure_location
      .never_received(x => x.final_arrival_date());

It should_not_leverage_the_arrival_deadline_time_check = () =>
   the_arrival_deadline
      .never_received(x => x
      .is_afterwards_than(
         the_itinerary_with_an_invalid_initial_departure_location
            .final_arrival_date()));
```

Por ello los:

```
.never_received
```

Y eso es todo, no hay más.

2.7.2 Describiendo el RED

Vamos a por los fallos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary with an
invalid initial departure location
» should confirm that the itinerary does not satisfy the route
specification (FAIL)
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should not leverage the itinerary final arrival location (FAIL)
» should not leverage the destination location identity comparer (FAIL)
» should not leverage the itinerary final arrival date (FAIL)
» should not leverage the arrival deadline time check (FAIL)

(....)

2 passed, 5 failed, 0 skipped, took 1,10 seconds (Machine.Specifications
0.3.0).
```

Que son exactamente los fallos que esperábamos.

Vamos a describirlos brevemente:

```
Test 'should confirm that the itinerary does not satisfy the route
specification' failed:
      Machine.Specifications.SpecificationException: Should be [false] but
is [true]
```

Fallo lógico, ya que esperamos un `false` y nuestro método devuelve siempre `true`.

```
Test 'should not leverage the itinerary final arrival location' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
IItinerary.final_arrival_location(); would not be called, but it was found
on the actual calls made on the mocked object.

(....)

Test 'should not leverage the destination location identity comparer'
failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
IValueObject`1.has_the_same_value_as(null); would not be called, but it
was found on the actual calls made on the mocked object.

(....)

Test 'should not leverage the itinerary final arrival date' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
```

```
IItinerary.final_arrival_date(); would not be called, but it was found on
the actual calls made on the mocked object.

(....)

Test 'should not leverage the arrival deadline time check' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
IArrivalDeadline.is_afterwards_than(null); would not be called, but it was
found on the actual calls made on the mocked object.
```

Es decir, esperábamos que no se llamasen a estas dependencias y sin embargo se llaman.

De nuevo, era lo que esperábamos.

2.7.3 El camino al GREEN

La forma más sencilla de resolver los fallos es hacer lo correcto. Es decir, escribir el código que de verdad evalúe (y tenga en cuenta) las condiciones para satisfacer la **DDD_Specification**.

Curiosamente, esto era lo esperábamos de esta **BDD_Spec**, que nos acorralase de tal forma que la única solución (o al menos la más sencilla) fuese escribir el código correcto:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    return the_itinerary != null &&
           underlying_origin_location.has_the_same_value_as(
              the_itinerary.initial_departure_location()) &&
           underlying_destination_location.has_the_same_value_as(
              the_itinerary.final_arrival_location()) &&
           underlying_arrival_deadline.is_afterwards_than(
              the_itinerary.final_arrival_date());
}
```

Ejecutamos los test y nos encontramos con que:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary with an
invalid initial departure location
```

```
» should confirm that the itinerary does not satisfy the route
specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should not leverage the itinerary final arrival location
» should not leverage the destination location identity comparer
» should not leverage the itinerary final arrival date
» should not leverage the arrival deadline time check


7 passed, 0 failed, 0 skipped, took 0,87 seconds (Machine.Specifications
0.3.0).
```

Ahora si que lo tenemos.

De hecho si corremos todos los tests (incluidos los de la **BDD_Spec** anterior), pasan todos.

Vamos a completar el comportamiento suponiendo que la condición que falla es alguna otra.

2.8 RouteSpecificationSpecs - When asked if the specification is satisfied by an itinerary with an invalid final arrival location

La **BDD_Spec** que vamos a implementar es ésta:

```
When asked if the specification is satisfied by an itinerary with an
invalid final arrival location
   It should confirm that the itinerary does not satisfy the route
specification.
   It should leverage the itinerary initial departure location.
   It should leverage the origin location identity comparer.
   It should leverage the itinerary final arrival location.
   It should leverage the destination location identity comparer.
   It should not leverage the itinerary final arrival date.
   It should not leverage the arrival deadline time check.
```

En código:

```
public class when_asked_if_the_specification_is_satisfied_by
             _an_itinerary_with_an_invalid_final_arrival_location :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_itinerary_with_an_invalid_final_arrival_location =
           an<IItinerary>();

        the_itinerary_with_an_invalid_final_arrival_location
           .Stub(x => x.initial_departure_location())
           .Return(an<ILocation>());
        the_itinerary_with_an_invalid_final_arrival_location
           .Stub(x => x.final_arrival_location())
           .Return(an<ILocation>());

        the_origin_location
           .Stub(x => x.has_the_same_value_as(
              the_itinerary_with_an_invalid_final_arrival_location
                 .initial_departure_location()))
           .Return(true);
        the_destination_location
           .Stub(x => x.has_the_same_value_as(
              the_itinerary_with_an_invalid_final_arrival_location
                 .final_arrival_location()))
           .Return(false);
    };

    Because of = () =>
```

```
    result = sut.is_satisfied_by(
                    the_itinerary_with_an_invalid_final_arrival_location);

    It should_confirm_that_the_itinerary
       _does_not_satisfy_the_route_specification = () =>
       result.ShouldBeFalse();

    It should_leverage_the_itinerary_initial_departure_location = () =>
       the_itinerary_with_an_invalid_final_arrival_location
          .received(x => x.initial_departure_location());

    It should_leverage_the_origin_location_identity_comparer = () =>
       the_origin_location
          .received(x => x
          .has_the_same_value_as(
             the_itinerary_with_an_invalid_final_arrival_location
                .initial_departure_location()));

    It should_leverage_the_itinerary_final_arrival_location = () =>
       the_itinerary_with_an_invalid_final_arrival_location
          .received(x => x.final_arrival_location());

    It should_leverage_the_destination_location_identity_comparer = () =>
       the_destination_location
          .received(x => x
          .has_the_same_value_as(
             the_itinerary_with_an_invalid_final_arrival_location
                .final_arrival_location()));

    It should_not_leverage_the_itinerary_final_arrival_date = () =>
       the_itinerary_with_an_invalid_final_arrival_location
          .never_received(x => x.final_arrival_date());

    It should_not_leverage_the_arrival_deadline_time_check = () =>
       the_arrival_deadline
          .never_received(x => x
          .is_afterwards_than(
             the_itinerary_with_an_invalid_final_arrival_location
                .final_arrival_date()));

    static bool result;
    static IItinerary the_itinerary_with_an_invalid_final_arrival_location;
}
```

Similar a la anterior **BDD_Spec**, solo que ahora estamos simulando que habiéndose cumplido la primera condición, la segunda falla (y por tanto la tercera no debe siquiera comprobarse).

Para ello establecemos el contexto adecuando en el bloque `ESTABLISH` y ajustamos

nuestras aserciones en los bloques `IT` (interesante fijarse en cuales reciben el tratamiento `.received()` y cuales `.never_received()`).

Ejecutamos los tests y:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary with an
invalid final arrival location
» should confirm that the itinerary does not satisfy the route
specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer
» should not leverage the itinerary final arrival date
» should not leverage the arrival deadline time check


7 passed, 0 failed, 0 skipped, took 0,69 seconds (Machine.Specifications
0.3.0).
```

Funcionan sin tocar nada.

Parece que ahora vamos por el camino correcto.

2.9 RouteSpecificationSpecs - When asked if the specification is satisfied by an itinerary with an invalid final arrival date

Primero la **BDD_Spec**:

```
When asked if the specification is satisfied by an itinerary with an
invalid final arrival date
   It should confirm that the itinerary does not satisfy the route
specification.
   It should leverage the itinerary initial departure location.
   It should leverage the origin location identity comparer.
   It should leverage the itinerary final arrival location.
   It should leverage the destination location identity comparer.
   It should leverage the itinerary final arrival date.
   It should leverage the arrival deadline time check.
```

Traducida a código:

```
public class when_asked_if_the_specification_is_satisfied_by
             _an_itinerary_with_an_invalid_final_arrival_date :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_itinerary_with_an_invalid_final_arrival_date = an<IItinerary>();

        the_itinerary_with_an_invalid_final_arrival_date
           .Stub(x => x.initial_departure_location())
           .Return(an<ILocation>());
        the_itinerary_with_an_invalid_final_arrival_date
           .Stub(x => x.final_arrival_location())
           .Return(an<ILocation>());
        the_itinerary_with_an_invalid_final_arrival_date
           .Stub(x => x.final_arrival_date())
           .Return(an<IArrivalDeadline>());

        the_origin_location
           .Stub(x => x.has_the_same_value_as(
              the_itinerary_with_an_invalid_final_arrival_date
                 .initial_departure_location()))
           .Return(true);
        the_destination_location
           .Stub(x => x.has_the_same_value_as(
              the_itinerary_with_an_invalid_final_arrival_date
                 .final_arrival_location()))
           .Return(true);
```

```
        the_arrival_deadline
           .Stub(x => x.is_afterwards_than(
              the_itinerary_with_an_invalid_final_arrival_date.final_arrival_da
te()))
           .Return(false);
    };

    Because of = () =>
       result = sut.is_satisfied_by(
                      the_itinerary_with_an_invalid_final_arrival_date);

    It should_confirm_that_the_itinerary
       _does_not_satisfy_the_route_specification = () =>
       result.ShouldBeFalse();

    It should_leverage_the_itinerary_initial_departure_location = () =>
       the_itinerary_with_an_invalid_final_arrival_date
          .received(x => x.initial_departure_location());

    It should_leverage_the_origin_location_identity_comparer = () =>
       the_origin_location
          .received(x => x
          .has_the_same_value_as(
             the_itinerary_with_an_invalid_final_arrival_date
                .initial_departure_location()));

    It should_leverage_the_itinerary_final_arrival_location = () =>
       the_itinerary_with_an_invalid_final_arrival_date
          .received(x => x.final_arrival_location());

    It should_leverage_the_destination_location_identity_comparer = () =>
       the_destination_location
          .received(x => x
          .has_the_same_value_as(
             the_itinerary_with_an_invalid_final_arrival_date
                .final_arrival_location()));

    It should_leverage_the_itinerary_final_arrival_date = () =>
       the_itinerary_with_an_invalid_final_arrival_date
          .received(x => x.final_arrival_date());

    It should_leverage_the_arrival_deadline_time_check = () =>
       the_arrival_deadline
          .received(x => x
          .is_afterwards_than(
             the_itinerary_with_an_invalid_final_arrival_date
                .final_arrival_date()));

    static bool result;
    static IItinerary the_itinerary_with_an_invalid_final_arrival_date;
}
```

Ahora estamos simulando que habiéndose cumplido la primera y la segunda condición, la tercera falla.

Para ello establecemos el contexto adecuado en el bloque `ESTABLISH` y ajustamos nuestras aserciones en los bloques `IT` (todos reciben el tratamiento `.received()`).

Ejecutamos los tests y:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by an itinerary with an
invalid final arrival date
» should confirm that the itinerary does not satisfy the route
specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer
» should leverage the itinerary final arrival date
» should leverage the arrival deadline time check


7 passed, 0 failed, 0 skipped, took 0,68 seconds (Machine.Specifications
0.3.0).
```

Nuevamente funcionan sin tener que tocar nada.

2.10 RouteSpecificationSpecs - When asked if the specification is satisfied by a null itinerary

En este caso, ésta sería la **BDD_Spec**:

```
When asked if the specification is satisfied by a null itinerary
   It should confirm that the itinerary does not satisfy the route
specification.
   It should not leverage the origin location identity comparer.
   It should not leverage the destination location identity comparer.
   It should not leverage the arrival deadline time check.
```

En código:

```
public class when_asked_if_the_specification_is_satisfied_by_a_null_itinerary :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_null_itinerary = null;
    };

    Because of = () => result = sut.is_satisfied_by(the_null_itinerary);

    It should_confirm_that_the_itinerary
       _does_not_satisfy_the_route_specification = () =>
       result.ShouldBeFalse();

    It should_not_leverage_the_origin_location_identity_comparer = () =>
       the_origin_location
          .never_received(x => x.has_the_same_value_as(an<ILocation>()));

    It should_not_leverage_the_destination_location_identity_comparer = () =>
       the_destination_location
          .never_received(x => x.has_the_same_value_as(an<ILocation>()));

    It should_not_leverage_the_arrival_deadline_time_check = () =>
       the_arrival_deadline
          .never_received(x => x.is_afterwards_than(an<IArrivalDeadline>()));

    static bool result;
    static IItinerary the_null_itinerary;
}
```

Lo más interesante es constatar como vamos a forzar que la condición "¿es `null`?" sea la primera en ser comprobada, ya que, sino es así, obtendremos una bonita

`NullReferenceException`.

Para ello usamos dos

```
.never_received(x => x.has_the_same_value_as(an<ILocation>()));
```

y un

```
.never_received(x => x.is_afterwards_than(an<IArrivalDeadline>()));
```

Con esto pretendemos comprobar que no recibimos la llamada a ese método, sea cual sea el argumento que le pasemos.

Comprobamos que los fallos son los que buscamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by a null itinerary
» should confirm that the itinerary does not satisfy the route
specification (FAIL)
» should not leverage the origin location identity comparer (FAIL)
» should not leverage the destination location identity comparer (FAIL)
» should not leverage the arrival deadline time check (FAIL)

Test 'should confirm that the itinerary does not satisfy the route
specification' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)

Test 'should not leverage the origin location identity comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)

Test 'should not leverage the destination location identity comparer'
failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)
```

```
Test 'should not leverage the arrival deadline time check' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)


0 passed, 4 failed, 0 skipped, took 1,98 seconds (Machine.Specifications
0.3.0).
```

Es decir obtenemos cuatro:

```
System.NullReferenceException: Object reference not set to an instance of
an object.
```

Necesitamos que haya una condición que compruebe si es `null`, y que además sea la primera:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    return the_itinerary != null &&
           underlying_origin_location.has_the_same_value_as(
              the_itinerary.initial_departure_location()) &&
           underlying_destination_location.has_the_same_value_as(
              the_itinerary.final_arrival_location()) &&
           underlying_arrival_deadline.is_afterwards_than(
              the_itinerary.final_arrival_date());
}
```

Que hace lo que esperamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if the specification is satisfied by a null itinerary
» should confirm that the itinerary does not satisfy the route
specification
» should not leverage the origin location identity comparer
» should not leverage the destination location identity comparer
» should not leverage the arrival deadline time check


4 passed, 0 failed, 0 skipped, took 0,79 seconds (Machine.Specifications
0.3.0).
```

2.11 Refactorizando When asked if the specification is satisfied by an itinerary that satisfies the route specification

En este epígrafe vamos a ver como podríamos refactorizar esta **BDD_Spec**, buscando una mayor coherencia semántica así como las ventajas y desventajas de una aproximación de este estilo.

2.11.1 La refactorización propuesta

Nuestra **BDD_Spec** actual es la siguiente:

```
v1

When asked if the specification is satisfied by an itinerary that
satisfies the route specification
   It should confirm that the itinerary satisfies the route
specification.
   It should leverage the itinerary initial departure location.
   It should leverage the origin location identity comparer.
   It should leverage the itinerary final arrival location.
   It should leverage the destination location identity comparer.
   It should leverage the itinerary final arrival date.
   It should leverage the arrival deadline time check.
```

Pero podríamos haber optado por esta otra, donde el apartado semántico está más desarrollado:

```
v2

When asked if the specification is satisfied by an itinerary that
satisfies the route specification
   It should confirm that the itinerary satisfies the route
specification.
   It should check that the itinerary initial departure location is
right.
   It should check that the itinerary final arrival location is right.
   It should check that the itinerary final arrival date is within
boundaries.
```

Lo único que hemos hecho ha sido convertir:

```
It should leverage the itinerary initial departure location.
It should leverage the origin location identity comparer.
```

en:

```
It should check that the itinerary initial departure location is right.
```

Éstas dos:

```
It should leverage the itinerary final arrival location.
It should leverage the destination location identity comparer.
```

en:

```
It should check that the itinerary final arrival location is right.
```

Y finalmente éstas otras dos:

```
It should leverage the itinerary final arrival date.
It should leverage the arrival deadline time check.
```

en:

```
It should check that the itinerary final arrival date is within
boundaries.
```

2.11.2 Descripción de las interacciones vs Semántica

No se puede decir que una esté bien y otra mal, o que una esté mejor que otra. Ambas responden a diferentes maneras de enfocar el desarrollo de la aplicación.

La **v1** da prioridad a la descripción exacta de las interacciones entre los distintos

objetos y eso es algo muy valioso, ya que con tan solo leer la **BDD_Spec**, podemos entender lo que hace al nivel de la interacción entre objetos.

La **v2** da prioridad a la semántica, de forma que, si por la razón que fuere, el cliente (que no tiene ni idea de informática), tuviese que ver esta **BDD_Spec**, al menos entendería algo más que con la **v1**.

Por norma, vamos a preferir la **v1**, pero es importante entender que esta decisión entra dentro de la preferencia personal.

2.11.3 La naturaleza de los bloques IT

Desde un punto de vista técnico, podemos comparar el bloque `IT` de por ejemplo:

```
v1

It should leverage the itinerary initial departure location.
It should leverage the origin location identity comparer.
```

frente a:

```
v2

It should check that the itinerary initial departure location is right.
```

En código:

```
v1


It should_leverage_the_itinerary_initial_departure_location = () =>
   the_itinerary_that_satisfies_the_route_specification
      .received(x => x.initial_departure_location());

It should_leverage_the_origin_location_identity_comparer = () =>
   the_origin_location
      .received(x => x
      .has_the_same_value_as(
         the_itinerary_that_satisfies_the_route_specification
```

```
            .initial_departure_location()));
```

frente a:

v2

```
It should_check_that_the_itinerary_initial_departure_location_is_right = () =>
{
    the_itinerary_that_satisfies_the_route_specification
       .received(x => x.initial_departure_location());

    the_origin_location
       .received(x => x
       .has_the_same_value_as(
          the_itinerary_that_satisfies_the_route_specification
             .initial_departure_location()));
};
```

Visto así, la **v1** tiene un gran punto a su favor, y es que cumple con la norma no escrita de:

- 1 bloque `IT` -> 1 linea de código.

Mientras que **v2** violaría esa norma (2 lineas de código).

Podemos rastrear el origen de esta norma en la motivación que se esconde tras los bloques `IT`, que no es otra que la de ser una herramienta para comprobar una sola cosa (de ahí lo de una linea de código). Si comprobamos dos o más cosas, probablemente estemos haciendo demasiado de golpe.

De nuevo, la linea que separa una opción de la otra es muy fina, ya que se podría argumentar que a veces, para comprobar un solo concepto lógico hacen falta más de una linea de código.

Las razones han sido expuestas. Ahora es el momento en el que cada uno debe escoger qué es lo que necesita.

> **NOTA**: El código que hemos tenido que escribir para solucionar cada una

de las aserciones (hacer que pasen cada uno de los tests) ha sido trivial. Lo verdaderamente complicado ha sido diseñar el sistema correctamente, de forma que cada objeto tenga clara su responsabilidad. De nuevo vemos como **BDD** es equivalente a **Diseño**.

2.12 RouteSpecificationSpecs - Implementando el comportamiento IValueObject<T>

En esta sección vamos a afrontar la implementación de un serie de **BDD_Specs** relacionadas entre si y que definen el comportamiento heredado de `IValueObject<T>` a través de su método `has_the_same_value_as()`:

```
public interface IValueObject<T>
{
    bool has_the_same_value_as(T the_other_value_object);
}
```

Las cinco **BDD_Specs** a las que nos referimos son:

```
When asked if two similar route specifications have the same value
   It should confirm that they have the same value.
   It should leverage the origin location value comparer.
   It should leverage the destination location value comparer.
   It should leverage the arrival deadline value comparer.

When asked if two route specifications with different origin location
have the same value
   It should confirm that they have different value.
   It should leverage the origin location value comparer.
   It should not leverage the destination location value comparer.
   It should not leverage the arrival deadline value comparer.

When asked if two route specifications with different destination
location have the same value
   It should confirm that they have different value.
   It should leverage the origin location value comparer.
   It should leverage the destination location value comparer.
   It should not leverage the arrival deadline value comparer.

When asked if two route specifications with different arrival deadline
have the same value
   It should confirm that they have different value.
   It should leverage the origin location value comparer.
   It should leverage the destination location value comparer.
   It should leverage the arrival deadline value comparer.

When asked if a null route specification has the same value as the
current specification
   It should confirm that they have different value.
   It should not leverage the origin location value comparer.
   It should not leverage the destination location value comparer.
```

```
It should not leverage the arrival deadline value comparer.
```

En ellas queda perfectamente definido el comportamiento deseado, así como todas las interacciones con los objetos colaboradores.

Puesto que no es la primera vez que vamos a implementar una funcionalidad parecida, esta vez vamos a cambiar el enfoque ligeramente para intentar simular como haríamos esto en un entorno de desarrollo real (en un trabajo, vamos).

Intentaremos, por tanto, mostrar como lo haríamos nosotros si no tuviésemos que explicarlo paso a paso.

Antes de afrontar esta sección, es recomendable entender perfectamente todo lo tratado hasta el momento, ya que muchas de las cuestiones en las que hacíamos hincapié con anterioridad, ahora serán pasadas por alto.

Empezamos.

2.12.1 Implementación del Happy Day Scenario

Lo primero que necesitamos es identificar el Happy Day Scenario para tener un punto de partida.

Ésta sería la **BDD_Spec** que lo define:

```
When asked if two similar route specifications have the same value
   It should confirm that they have the same value.
   It should leverage the origin location value comparer.
   It should leverage the destination location value comparer.
   It should leverage the arrival deadline value comparer.
```

A continuación la traduciríamos a código, siguiendo el orden ya conocido de:

1. Clase `when_...`
2. Bloque `IT` principal (Assert)

3. Bloque `BECAUSE` (Act)
4. El resto de los bloques `IT` (Assert)
5. Bloque `ESTABLISH` (Arrange)

En todo momento corregimos el código rojo nada más presentarse.

```
public class when_asked_if_two_similar_route_specifications
             _have_the_same_value :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_other_route_specification = an<IRouteSpecification>();

        the_other_route_specification
           .Stub(x => x.origin())
           .Return(the_origin_location);
        the_origin_location
           .Stub(x => x.has_the_same_value_as(the_origin_location))
           .Return(true);

        the_other_route_specification
           .Stub(x => x.destination())
           .Return(the_destination_location);
        the_destination_location
           .Stub(x => x.has_the_same_value_as(the_destination_location))
           .Return(true);

        the_other_route_specification
           .Stub(x => x.arrival_deadline())
           .Return(the_arrival_deadline);
        the_arrival_deadline
           .Stub(x => x.has_the_same_value_as(the_arrival_deadline))
           .Return(true);
    };

    Because of = () =>
       result = sut.has_the_same_value_as(the_other_route_specification);

    It should_confirm_that_they_have_the_same_value = () =>
       result.ShouldBeTrue();

    It should_leverage_the_origin_location_value_comparer = () =>
       the_origin_location
          .received(x => x.has_the_same_value_as(the_origin_location));

    It should_leverage_the_destination_location_value_comparer = () =>
       the_destination_location
          .received(x => x.has_the_same_value_as(the_destination_location));
```

```
    It should_leverage_the_arrival_deadline_value_comparer = () =>
       the_arrival_deadline
          .received(x => x.has_the_same_value_as(the_arrival_deadline));

    static bool result;
    static IRouteSpecification the_other_route_specification;
}
```

Este tipo de código debería resultar familiar a estas alturas:

- El primer bloque `IT` comprueba la salida del método (TDD Tradicional).
- Los tres bloques `IT` restantes comprueban las interacciones (TDD Mockista).
- En el bloque `ESTABLISH`:
  - Establecemos las comparaciones consigo mismo de:
    - `the_origin_location`
    - `the_arrival_location`
    - `the_arrival_deadline`
  - Simulamos una salida positiva (`true`) para cada una de las 3 comparaciones.

Ejecutamos la **BDD_Spec**, buscando el tipo de error que esperamos:

```
Not Implemented
```

en este caso.

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two similar route specifications have the same value
» should confirm that they have the same value (FAIL)
» should leverage the origin location value comparer (FAIL)
» should leverage the destination location value comparer (FAIL)
» should leverage the arrival deadline value comparer (FAIL)

Test 'should confirm that they have the same value' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)
```

```
Test 'should leverage the origin location value comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

Test 'should leverage the destination location value comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

Test 'should leverage the arrival deadline value comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)


0 passed, 4 failed, 0 skipped, took 1,03 seconds (Machine.Specifications
0.3.0).
```

Una vez obtenidos esos cuatro errores, vamos a implementar la funcionalidad.

Puesto que tenemos interacciones (delegación también sería una palabra correcta) con dependencias, escribimos primero éstas, para no perdernos:

```
public bool has_the_same_value_as(IRouteSpecification the_other)
{
   underlying_origin_location
      .has_the_same_value_as(the_other.origin());
   underlying_destination_location
      .has_the_same_value_as(the_other.destination());
   underlying_arrival_deadline
      .has_the_same_value_as(the_other.arrival_deadline());

   return false;
}
```

y volvemos a ejecutar la **BDD_Spec** para ver si se nos ha olvidado algo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two similar route specifications have the same value
» should confirm that they have the same value (FAIL)
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should leverage the arrival deadline value comparer

Test 'should confirm that they have the same value' failed:
      Machine.Specifications.SpecificationException: Should be [true] but
is [false]
      at
Machine.Specifications.ShouldExtensionMethods.ShouldBeTrue(Boolean
condition)
      domain\model\cargo.aggregate\RouteSpecificationSpecs.cs(336,0): at
dddsample.specs.domain.model.cargo.aggregate.when_asked_if_two_similar_rou
te_specifications_have_the_same_value.<.ctor>b__8()
      at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
      at Machine.Specifications.Model.Specification.Verify()

3 passed, 1 failed, 0 skipped, took 0,85 seconds (Machine.Specifications
0.3.0).
```

Perfecto, es exactamente lo que buscábamos:

- Las tres aserciones basadas en la interacción con dependencias pasan (las tres últimas).
- La aserción que comprueba el resultado no pasa ya que la hemos hecho fallar a propósito para comprobar que efectivamente podemos fallarla por las razones adecuadas. Para conseguirlo:
  - Esperábamos un `true` (**result.ShouldBeTrue()**).
  - Forzamos un `false` (**return false**).
  - El test nos marca nuestro error (**Should be [true] but is [false]**)

Ahora que ya hemos comprobado que no nos olvidamos de nada, transformamos el código escrito en lo que buscamos.

> **NOTA**: "Escribir únicamente el mínimo código necesario Y QUE ADEMÁS, TENGA SENTIDO, para que el test pase."

```
public bool has_the_same_value_as(IRouteSpecification the_other)
{
    return underlying_origin_location
               .has_the_same_value_as(the_other.origin()) &&
           underlying_destination_location
               .has_the_same_value_as(the_other.destination()) &&
           underlying_arrival_deadline
               .has_the_same_value_as(the_other.arrival_deadline());
}
```

Volvemos a ejecutar la **BDD_Spec**, esta vez con la esperanza de obtener un resultado positivo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two similar route specifications have the same value
» should confirm that they have the same value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should leverage the arrival deadline value comparer


4 passed, 0 failed, 0 skipped, took 0,86 seconds (Machine.Specifications
0.3.0).
```

Dicho y hecho.

Vamos a por la siguiente.

2.12.2 Implementación de los Scenarios complementarios al Happy Day Scenario

Vamos con las cuatro **BDD_Specs** que quedan, que se podrían caracterizar por qué:

- Provocamos el fallo de la primera condición.
- Provocamos el fallo de la segunda condición.
- Provocamos el fallo de la tercera condición.
- Ejercitamos el caso extraordinario.

Si todo ha ido bien, probablemente no tengamos que escribir código de la aplicación excepto en el cuarto caso.

2.12.2.1 Regla del Sacacorchos para la traducción a código de las interacciones

Cuando proporcionamos un mock, entonces el:

```
(...) should leverage (...)
```

de la **BDD_Spec**, se traducirá en un `.received()` en el código.

Cuando no proporcionamos un mock, entonces el:

```
should not leverage
```

de la **BDD_Spec**, se traducirá en un `.never_received()` en el código.

2.12.2.2 El que provoca el fallo de la primera condición

Buscamos hacer que falle:

```
this.underlying_origin_location.has_the_same_value_as(the_other.origin())
```

Por lo tanto:

```
When asked if two route specifications with different origin location
have the same value
   It should confirm that they have different value.
   It should leverage the origin location value comparer.
   It should not leverage the destination location value comparer.
   It should not leverage the arrival deadline value comparer.
```

Como podemos leer en la **BDD_Spec**, para hacer fallar la primera condición:

- Necesitamos proporcionar un mock del mismo que devuelva `false`.
- No necesitamos proporcionar un mock del segundo.

- No necesitamos proporcionar un mock del tercero.

Así que:

```
public class when_asked_if_two_route_specifications
             _with_different_origin_location_have_the_same_value :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_other_route_specification = an<IRouteSpecification>();

        the_other_route_specification
           .Stub(x => x.origin())
           .Return(the_origin_location);
        the_origin_location
           .Stub(x => x.has_the_same_value_as(the_origin_location))
           .Return(false);
    };

    Because of = () =>
       result = sut.has_the_same_value_as(the_other_route_specification);

    It should_confirm_that_they_have_different_value = () =>
       result.ShouldBeFalse();

    It should_leverage_the_origin_location_value_comparer = () =>
       the_origin_location
          .received(x => x.has_the_same_value_as(the_origin_location));

    It should_not_leverage_the_destination_location_value_comparer = () =>
       the_destination_location
          .never_received(x => x
          .has_the_same_value_as(the_destination_location));

    It should_not_leverage_the_arrival_deadline_value_comparer = () =>
       the_arrival_deadline
          .never_received(x => x.has_the_same_value_as(the_arrival_deadline));

    static bool result;
    static IRouteSpecification the_other_route_specification;
}
```

Forzamos el fallo de la primera condición gracias a:

```
.Return(false);
```

Ejecutamos la **BDD_Spec**. No esperamos fallos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two route specifications with different origin location have
the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should not leverage the destination location value comparer
» should not leverage the arrival deadline value comparer


4 passed, 0 failed, 0 skipped, took 0,67 seconds (Machine.Specifications
0.3.0).
```

2.12.2.3 El que provoca el fallo de la segunda condición

Buscamos provocar el fallo de:

```
this.underlying_destination_location
       .has_the_same_value_as(the_other.destination())
```

con esta **BDD_Spec**:

```
When asked if two route specifications with different destination
location have the same value
   It should confirm that they have different value.
   It should leverage the origin location value comparer.
   It should leverage the destination location value comparer.
   It should not leverage the arrival deadline value comparer.
```

De nuevo, leyendo en la **BDD_Spec**, para hacer fallar la segunda condición:

- Necesitamos proporcionar un mock del primero que devuelva `true`.
- Necesitamos proporcionar un mock del segundo que devuelva `false`.
- No necesitamos proporcionar un mock del tercero.

El código:

```
public class when_asked_if_two_route_specifications
             _with_different_destination_location_have_the_same_value :
             concern_for_route_specification
```

```
{
    Establish context = () =>
    {
        the_other_route_specification = an<IRouteSpecification>();

        the_other_route_specification
           .Stub(x => x.origin())
           .Return(the_origin_location);
        the_origin_location
           .Stub(x => x.has_the_same_value_as(the_origin_location))
           .Return(true);

        the_other_route_specification
           .Stub(x => x.destination())
           .Return(the_destination_location);
        the_destination_location
           .Stub(x => x.has_the_same_value_as(the_destination_location))
           .Return(false);
    };

    Because of = () =>
       result = sut.has_the_same_value_as(the_other_route_specification);

    It should_confirm_that_they_have_different_value = () =>
       result.ShouldBeFalse();

    It should_leverage_the_origin_location_value_comparer = () =>
       the_origin_location
          .received(x => x.has_the_same_value_as(the_origin_location));

    It should_leverage_the_destination_location_value_comparer = () =>
       the_destination_location
          .received(x => x.has_the_same_value_as(the_destination_location));

    It should_not_leverage_the_arrival_deadline_value_comparer = () =>
       the_arrival_deadline
          .never_received(x => x.has_the_same_value_as(the_arrival_deadline));

    static bool result;
    static IRouteSpecification the_other_route_specification;
}
```

De nuevo no esperamos fallos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two route specifications with different destination location
have the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
```

```
  » should not leverage the arrival deadline value comparer


  4 passed, 0 failed, 0 skipped, took 0,75 seconds (Machine.Specifications
  0.3.0).
```

2.12.2.4 El que provoca el fallo de la tercera condición

Esta vez pretendemos buscar el fallo de:

```
this.underlying_arrival_deadline
       .has_the_same_value_as(the_other.arrival_deadline())
```

Y ésta es la **BDD_Spec**:

```
When asked if two route specifications with different arrival deadline
have the same value
   It should confirm that they have different value.
   It should leverage the origin location value comparer.
   It should leverage the destination location value comparer.
   It should leverage the arrival deadline value comparer.
```

Interpretando la **BDD_Spec**, para hacer fallar la tercera condición:

- Necesitamos proporcionar un mock del primero que devuelva `true`.
- Necesitamos proporcionar un mock del segundo que devuelva `true`.
- Necesitamos proporcionar un mock del tercero que devuelva `false`.

Antes de mostrar el código, señalar que es similar al del Happy Day Scenario, con la salvedad de que esta vez:

- Necesitamos que el tercer mock devuelva `false`.
- En el primer `IT` necesitamos `ShouldBeFalse()`.

```
public class when_asked_if_two_route_specifications
            _with_different_arrival_deadline_have_the_same_value :
            concern_for_route_specification
{
    Establish context = () =>
    {
```

```
        the_other_route_specification = an<IRouteSpecification>();

        the_other_route_specification
           .Stub(x => x.origin())
           .Return(the_origin_location);
        the_origin_location
           .Stub(x => x.has_the_same_value_as(the_origin_location))
           .Return(true);

        the_other_route_specification
           .Stub(x => x.destination())
           .Return(the_destination_location);
        the_destination_location
           .Stub(x => x.has_the_same_value_as(the_destination_location))
           .Return(true);

        the_other_route_specification
           .Stub(x => x.arrival_deadline())
           .Return(the_arrival_deadline);
        the_arrival_deadline
           .Stub(x => x.has_the_same_value_as(the_arrival_deadline))
           .Return(false);
    };

    Because of = () =>
       result = sut.has_the_same_value_as(the_other_route_specification);

    It should_confirm_that_they_have_different_value = () =>
       result.ShouldBeFalse();

    It should_leverage_the_origin_location_value_comparer = () =>
       the_origin_location
          .received(x => x.has_the_same_value_as(the_origin_location));

    It should_leverage_the_destination_location_value_comparer = () =>
       the_destination_location
          .received(x => x.has_the_same_value_as(the_destination_location));

    It should_leverage_the_arrival_deadline_value_comparer = () =>
       the_arrival_deadline
          .received(x => x.has_the_same_value_as(the_arrival_deadline));

    static bool result;
    static IRouteSpecification the_other_route_specification;
}
```

De nuevo no esperamos errores:

```
  ------ Test started: Assembly: dddsample.specs.dll ------

  when asked if two route specifications with different arrival deadline
```

```
have the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should leverage the arrival deadline value comparer


4 passed, 0 failed, 0 skipped, took 0,91 seconds (Machine.Specifications
0.3.0).
```

2.12.2.5 El caso extraordinario

Ahora si que vamos a tener que añadir código.

Ésta es la **BDD_Spec**:

```
When asked if a null route specification has the same value as the
current specification
   It should confirm that they have different value.
   It should not leverage the origin location value comparer.
   It should not leverage the destination location value comparer.
   It should not leverage the arrival deadline value comparer.
```

Leyendo en la **BDD_Spec**:

- No necesitamos proporcionar un mock del primero.
- No necesitamos proporcionar un mock del segundo.
- No necesitamos proporcionar un mock del tercero.

La idea es obligar al código a que antes de nada compruebe el "no es `null`".

El código:

```
public class when_asked_if_a_null_route_specification
            _has_the_same_value_as_the_current_specification :
            concern_for_route_specification
{
    Establish context = () => the_other_route_specification = null;

    Because of = () =>
       result = sut.has_the_same_value_as(the_other_route_specification);
```

```
    It should_confirm_that_they_have_different_value = () =>
       result.ShouldBeFalse();

    It should_not_leverage_the_origin_location_value_comparer = () =>
       the_origin_location
          .never_received(x => x.has_the_same_value_as(the_origin_location));

    It should_not_leverage_the_destination_location_value_comparer = () =>
       the_destination_location
          .never_received(x => x
          .has_the_same_value_as(the_destination_location));

    It should_not_leverage_the_arrival_deadline_value_comparer = () =>
       the_arrival_deadline
          .never_received(x => x.has_the_same_value_as(the_arrival_deadline));

    static bool result;
    static IRouteSpecification the_other_route_specification;
}
```

En vez de usar un mock como hasta ahora, usamos simplemente un objeto normal a `null`:

```
the_other_route_specification = null;
```

Esta vez si esperamos fallos. De hecho deberían fallar todos por:

```
NullReferenceException
```

Y lo hacen:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if a null route specification has the same value as the current
specification
» should confirm that they have different value (FAIL)
» should not leverage the origin location value comparer (FAIL)
» should not leverage the destination location value comparer (FAIL)
» should not leverage the arrival deadline value comparer (FAIL)

Test 'should confirm that they have different value' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)
```

```
Test 'should not leverage the origin location value comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)

Test 'should not leverage the destination location value comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)

Test 'should not leverage the arrival deadline value comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. ---> System.NullReferenceException:
Object reference not set to an instance of an object.

(....)


0 passed, 4 failed, 0 skipped, took 0,66 seconds (Machine.Specifications
0.3.0).
```

Para solucionarlo basta con añadir la comprobación de `null` como primera condición:

```
public bool has_the_same_value_as(IRouteSpecification the_other)
{
    return the_other != null &&
           this.underlying_origin_location
              .has_the_same_value_as(the_other.origin()) &&
           this.underlying_destination_location
              .has_the_same_value_as(the_other.destination()) &&
           this.underlying_arrival_deadline
              .has_the_same_value_as(the_other.arrival_deadline());
}
```

y volvemos a comprobar:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if a null route specification has the same value as the current
specification
» should confirm that they have different value
» should not leverage the origin location value comparer
» should not leverage the destination location value comparer
» should not leverage the arrival deadline value comparer
```

```
4 passed, 0 failed, 0 skipped, took 0,79 seconds (Machine.Specifications
0.3.0
```

2.12.3 Consideraciones finales

Antes de poder dar por finalizada la implementación de esta funcionalidad, debemos correr la suite de **BDD_Specs** y comprobar que todas funcionan:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for its origin location
» should give back the origin location

when asked for its destination location
» should give back the destination location

when asked for its arrival deadline
» should give back the arrival deadline

when asked if the specification is satisfied by an itinerary that
satisfies the route specification
» should confirm that the itinerary satisfies the route specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer
» should leverage the itinerary final arrival date
» should leverage the arrival deadline time check

when asked if the specification is satisfied by an itinerary with an
invalid initial departure location
» should confirm that the itinerary does not satisfy the route
specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should not leverage the itinerary final arrival location
» should not leverage the destination location identity comparer
» should not leverage the itinerary final arrival date
» should not leverage the arrival deadline time check

when asked if the specification is satisfied by an itinerary with an
invalid final arrival location
» should confirm that the itinerary does not satisfy the route
specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
```

```
» should leverage the destination location identity comparer
» should not leverage the itinerary final arrival date
» should not leverage the arrival deadline time check

when asked if the specification is satisfied by an itinerary with an
invalid final arrival date
» should confirm that the itinerary does not satisfy the route
specification
» should leverage the itinerary initial departure location
» should leverage the origin location identity comparer
» should leverage the itinerary final arrival location
» should leverage the destination location identity comparer
» should leverage the itinerary final arrival date
» should leverage the arrival deadline time check

when asked if the specification is satisfied by a null itinerary
» should confirm that the itinerary does not satisfy the route
specification
» should not leverage the origin location identity comparer
» should not leverage the destination location identity comparer
» should not leverage the arrival deadline time check

when asked if two similar route specifications have the same value
» should confirm that they have the same value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should leverage the arrival deadline value comparer

when asked if two route specifications with different origin location have
the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should not leverage the destination location value comparer
» should not leverage the arrival deadline value comparer

when asked if two route specifications with different destination location
have the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should not leverage the arrival deadline value comparer

when asked if two route specifications with different arrival deadline
have the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should leverage the arrival deadline value comparer

when asked if a null route specification has the same value as the current
specification
» should confirm that they have different value
```

```
» should not leverage the origin location value comparer
» should not leverage the destination location value comparer
» should not leverage the arrival deadline value comparer


55 passed, 0 failed, 0 skipped, took 1,15 seconds (Machine.Specifications
0.3.0).
```

Ahora si que hemos finalizado.

2.13 RouteSpecificationSpecs - When asked about the route specification hash code

En esta **BDD_Spec** se va a poner de manifiesto cuales son nuestras verdaderas motivaciones, y la prioridad que damos al Interaction-Based Testing TDD.

Vamos a por ello.

La **BDD_Spec** sería ésta:

```
When asked about the route specification hash code
   It should leverage the origin location hash code.
   It should leverage the destination location hash code.
   It should leverage the arrival deadline hash code.
```

De aquí ya podríamos sacar varias conclusiones:

- Estamos calculando un valor numérico (el **hash code**) y no tenemos ni una sola aserción sobre el resultado en si.
- Sin embargo, tenemos 3 aserciones sobre la interacción con sus dependencias / colaboradores.

Vamos a ver el código, que, a estas alturas, no debería plantear ningún desafío:

```
public class when_asked_about_the_route_specification_hash_code :
             concern_for_route_specification
{
    Establish context = () =>
    {
        the_origin_location
           .Stub(x => x.GetHashCode())
           .Return(new int());

        the_destination_location
           .Stub(x => x.GetHashCode())
           .Return(new int());

        the_arrival_deadline
           .Stub(x => x.GetHashCode())
           .Return(new int());
    };
```

```
    Because of = () => sut.GetHashCode();

    It should_leverage_the_origin_location_hash_code = () =>
       the_origin_location
          .received(x => x.GetHashCode());

    It should_leverage_the_destination_location_hash_code = () =>
       the_destination_location
          .received(x => x.GetHashCode());

    It should_leverage_the_arrival_deadline_hash_code = () =>
       the_arrival_deadline
          .received(x => x.GetHashCode());
}
```

Nuestra indiferencia ante el resultado numérico es total, basta considerar que nuestros 3 mocks devuelven un número por defecto:

```
.Return(new int());
```

El bloque `BECAUSE`, ni siquiera almacena el resultado, tan solo se ejecuta:

```
sut.GetHashCode();
```

Antes de ir con la prueba del algodón, vamos a arreglar un poco el entorno para que no obtengamos errores raros.

Primeramente, declaramos el método `GetHashCode()` en las interfaces de los colaboradores, a saber, `ILocation`:

```
public interface ILocation : IValueObject<ILocation>
{
    int GetHashCode();
}
```

e `IArrivalDeadline`:

```
public interface IArrivalDeadline : IValueObject<IArrivalDeadline>
{
    bool is_afterwards_than(IArrivalDeadline final_arrival_date);
    int GetHashCode();
}
```

A continuación declaramos el método `GetHashCode()` en la interface de nuestro SUT, `IRouteSpecification`:

```
public interface IRouteSpecification : ISpecification<IItinerary>,
                                       IValueObject<IRouteSpecification>
{
    ILocation origin();
    ILocation destination();
    IArrivalDeadline arrival_deadline();
    int GetHashCode();
}
```

y le damos la implementación por defecto en `RouteSpecification`:

```
public override int GetHashCode()
{
    return base.GetHashCode();
}
```

Ahora ejecutamos la **BDD_Spec**, esperando obtener los tres errores adecuados:

```
  Expected #1, Actual #0

  ------ Test started: Assembly: dddsample.specs.dll ------

  when asked about the route specification hash code
  » should leverage the origin location hash code (FAIL)
  » should leverage the destination location hash code (FAIL)
  » should leverage the arrival deadline hash code (FAIL)

  Test 'should leverage the origin location hash code' failed:
        Rhino.Mocks.Exceptions.ExpectationViolationException:
  ILocation.GetHashCode(); Expected #1..2147483647, Actual #0.

  (....)

  Test 'should leverage the destination location hash code' failed:
        Rhino.Mocks.Exceptions.ExpectationViolationException:
  ILocation.GetHashCode(); Expected #1..2147483647, Actual #0.

  (....)

  Test 'should leverage the arrival deadline hash code' failed:
        Rhino.Mocks.Exceptions.ExpectationViolationException:
  IArrivalDeadline.GetHashCode(); Expected #1..2147483647, Actual #0.

  (....)
```

```
0 passed, 3 failed, 0 skipped, took 1,18 seconds (Machine.Specifications
0.3.0).
```

Nada que no esperásemos.

Vamos a arreglarlo.

Empezamos con las interacciones con las dependencias:

```
public override int GetHashCode()
{
    this.underlying_origin_location.GetHashCode();
    this.underlying_destination_location.GetHashCode();
    this.underlying_arrival_deadline.GetHashCode();

    return new int();
}
```

y comprobamos que no nos hemos olvidado de nada:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about the route specification hash code
» should leverage the origin location hash code
» should leverage the destination location hash code
» should leverage the arrival deadline hash code


3 passed, 0 failed, 0 skipped, took 1,16 seconds (Machine.Specifications
0.3.0).
```

Lo tenemos dominado.

Nos falta implementar un algoritmo de **hash code**, y éste es tan valido como cualquier otro:

```
public override int GetHashCode()
{
    int result =  this.underlying_origin_location.GetHashCode();
    result = result * 397 + this.underlying_destination_location.GetHashCode();
    result = result * 397 + this.underlying_arrival_deadline.GetHashCode();
    return result;
```

```
}
```

> **NOTA**: No tomar este algoritmo de **hash code** como "un buen" algoritmo para calcular **hash codes**. En nuestro caso cumple perfectamente su propósito, pero en un entorno diferente al pedagógico, deberían hacerse algunas consideraciones extra.

Las cosas deberían seguir funcionando, y así es, ya que volvemos a obtener:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about the route specification hash code
» should leverage the origin location hash code
» should leverage the destination location hash code
» should leverage the arrival deadline hash code


3 passed, 0 failed, 0 skipped, took 0,89 seconds (Machine.Specifications
0.3.0).
```

Antes de continuar, es muy importante que entendamos, que lo que acabamos de hacer ha sido diseñar el calculo del **hash code**. No nos importa su resultado, simplemente las partes que lo conforman (los objetos que nos ayudan a implementar la funcionalidad).

Una vez hecho esto, podríamos añadir la siguiente aserción, si con ello quedamos más tranquilos:

> *It should calculate the hash code according to a given algorithm.*

Que en código sería:

```
It should_calculate_the_hash_code_according_to_a_given_algorithm = () =>
   result.ShouldEqual(316812);
```

Esto nos obligaría a cambiar ligeramente el código de la **BDD_Spec**:

```
public class when_asked_about_the_route_specification_hash_code :
```

```
            concern_for_route_specification
{
    Establish context = () =>
    {
        the_origin_location
           .Stub(x => x.GetHashCode())
           .Return(2);

        the_destination_location
           .Stub(x => x.GetHashCode())
           .Return(4);

        the_arrival_deadline
           .Stub(x => x.GetHashCode())
           .Return(6);
    };

    Because of = () => sut.GetHashCode();

    It should_leverage_the_origin_location_hash_code = () =>
       the_origin_location
          .received(x => x.GetHashCode());

    It should_leverage_the_destination_location_hash_code = () =>
       the_destination_location
          .received(x => x.GetHashCode());

    It should_leverage_the_arrival_deadline_hash_code = () =>
       the_arrival_deadline
          .received(x => x.GetHashCode());

    It should_calculate_the_hash_code_according_to_a_given_algorithm = () =>
       result.ShouldEqual(316812);

    static int result;
}
```

De todas formas comprobamos que funciona sin tocar nada del código de `GetHashCode()`, lo que prueba que lo habíamos hecho bien:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about the route specification hash code
» should leverage the origin location hash code
» should leverage the destination location hash code
» should leverage the arrival deadline hash code
» should calculate the hash code according to a given algorithm


4 passed, 0 failed, 0 skipped, took 0,94 seconds (Machine.Specifications
```

```
0.3.0).
```

Ya estaríamos dispuestos a empaquetar y a **github**.

2.14 Asegurándonos de que el el proceso de construcción es correcto

Por el momento, la construcción de `RouteSpecification` se ha limitado a ofrecer a las **dependencias inyectadas** un campo privado en el cual ser almacenadas:

```
public RouteSpecification(ILocation the_origin_location,
                          ILocation the_destination_location,
                          IArrivalDeadline the_arrival_deadline)
{
    this.underlying_origin_location = the_origin_location;
    this.underlying_destination_location = the_destination_location;
    this.underlying_arrival_deadline = the_arrival_deadline;
}
```

Sin embargo, no hemos realizado ninguna comprobación sobre las propias dependencias para asegurarnos de que se cumplen las invariantes necesarias para que el **DDD_Value_Object** `RouteSpecification` se cree en un estado correcto.

Podríamos listar esas invariantes como:

1. Debe existir un punto de origen (`origin_location`).
2. Debe existir un punto de destino (`arrival_location`).
3. Debe existir una fecha limite de entrega (`arrival_deadline`).
4. Los puntos de origen y destino no pueden ser los mismos.

Si cualquiera de estas invariantes no se cumpliese, por necesidad, deberíamos evitar que se cree el objeto, ya que, algo ha ido mal.

La mejor forma de reforzar este concepto es lanzar excepciones si cualquiera de las invariantes no se cumple.

En el ejemplo java, estas excepciones, se lanzan desde el constructor. Teniendo en cuenta que una futura inclusión de un framework de **DI / IoC** (como por ejemplo **Ninject!**) podría modificar las cosas, lanzar excepciones desde el constructor nos parece que viola el **SRP**. El constructor hace demasiadas cosas. Por esa razón, y teniendo en cuenta que disponemos de las dependencias de esta clase en el momento

de la construcción, una **DDD_Factory** parece la mejor solución.

Podríamos no crear esa **DDD_Factory** y lanzar esas excepciones desde el constructor, ya que lo bueno de nuestra aproximación desde un paradigma **BDD**, es que nos permite cambiar de opinión en cualquier momento y, por poner un ejemplo, refactorizar hacia una **DDD_Factory**.

Sin embargo el **SRP** pesa mucho. Un constructor no debería lanzar excepciones, así que vamos a escoger la solución de la **DDD_Factory**.

Esperemos que esto no nos acarree demasiados quebraderos de cabeza con el código existente y nos tengamos que arrepentir de nuestra decisión.

Vamos por lo tanto con las **BDD_Spec** que implementan este comportamiento:

```
When attempting to inject a null origin location into the route
specification factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject a null destination location into the route
specification factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject a null arrival deadline into the route
specification factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject the same origin and destination location into
the route specification factory
   It should leverage the origin location value comparer.
   It should throw an argument exception.
   It should throw an invariant violated exception message.
```

2.14.1 RouteSpecificationFactorySpecs - When attempting to inject a null origin location into the route specification factory

Esta **BDD_Spec** es la que nos va a permitir generar el armazón básico de la **DDD_Factory**. Podríamos haber empezado por cualquiera de ellas. En este caso el

orden no es importante ya que difícilmente podemos identificar un Happy Day Scenario aquí, pues todas van a lanzar excepciones.

Vamos con la **BDD_Spec**:

```
When attempting to inject a null origin location into the route
specification factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.
```

2.14.1.1 Consideraciones previas

La idea es tanto traducir la **BDD_Spec** a código como, en el proceso, ir tomando diferentes decisiones que afectarán al diseño. Aunque a estas alturas deberíamos tener cierta práctica, en cuanto a escribir e implementar **BDD_Specs** se refiere, en este caso vamos a volver a un nivel de detalle grande, ya que, vamos a introducir nuevas características del framework **BDD** `machine.specifications.developwithpassion`.

Otra cuestión interesante, será comprobar lo sencillo que resulta implementar un **DDD_Factory** (conceptual) a través de una de las implementaciones del **GoF** (concreta), guiados por el **BDD**.

2.14.1.2 La clase base abstracta concern for route specification factory

Empezamos con el código:

```
public abstract class concern_for_route_specification_factory :
                      Observes<RouteSpecificationFactory> { }
```

y antes de abordar el problema indicado por el código rojo, debemos hacer un comentario sobre la diferencia entre los anteriores `Observes<Contract, ClassUnderTest>`:

```
Observes<IRouteSpecification, RouteSpecification>
```

y que ahora se ha convertido en `Observes<ClassUnderTest>`:

```
Observes<RouteSpecificationFactory>
```

No estamos usando una interface para la **DDD_Factory**. Y aunque mucho nos tememos que va a ser una decisión de la que nos arrepintamos en cuanto nos veamos en la situación de necesitar establecer, a través de un mock, que alguno de los métodos de `RouteSpecificationFactory` "sea llamado tantas veces con los siguientes parámetros" y no podamos, por el momento vamos a mantenerlo. Ya abordaremos ese problema si es que se produce.

Vamos a por el código rojo:

```
public class RouteSpecificationFactory {}
```

Fácil. Pero lo importante no ha sido escribir esa linea de código, sino la decisión de diseño que tomamos unos instantes antes y que acabamos de detallar.

2.14.1.3 Capturando excepciones en una BDD_Spec

El código de la **BDD_Spec** con sus aserciones sería:

```
public class when_attempting_to_inject_a_null_origin_location
            _into_the_route_specification_factory :
            concern_for_route_specification_factory
{
    It should_throw_a_null_argument_exception = () =>
       exception_thrown_by_the_sut.ShouldBeAn<ArgumentNullException>();

    It should_throw_an_invariant_violated_exception_message = () =>
       exception_thrown_by_the_sut
          .ShouldContainErrorMessage(
             "Invariant Violated: origin location is required.");
}
```

No hay novedades en el frente de la mecánica del proceso de escribir **BDD_Specs**. Empezamos como siempre, por las aserciones, por los bloques `IT`. Ahora bien, estamos usando por primera vez la propiedad `readonly`:

```
exception_thrown_by_the_sut
```

Tal y como su nombre indica, nos referimos a las excepciones lanzadas por el SUT, que en este caso es la **DDD_Factory** `RouteSpecificationFactory`.

Por lo tanto, analizando una a una las aserciones, nos encontramos con que:

```
It should_throw_a_null_argument_exception = () =>
   exception_thrown_by_the_sut.ShouldBeAn<ArgumentNullException>();
```

Comprueba que se lanza una excepción del tipo `ArgumentNullException`.

```
It should_throw_an_invariant_violated_exception_message = () =>
   exception_thrown_by_the_sut
      .ShouldContainErrorMessage(
         "Invariant Violated: origin location is required.");
```

Comprueba que el mensaje de error de la excepción lanzada sea el que nosotros especificamos.

Seguimos con el proceso, por lo que necesitamos ejercitar el SUT a través de un bloque `BECAUSE`:

```
Because of = () =>
   catch_exception(() =>
      sut.create_route_specification_using(the_origin_location,
                                           the_destination_location,
                                           the_arrival_deadline));
```

Hay bastante código rojo.

Vamos primero con la parte trivial y que estamos hartos de ver, los tres argumentos del método que se convierten en campos de nuestra **BDD_Spec**:

```
static ILocation the_origin_location
static ILocation the_destination_location;
static IArrivalDeadline the_arrival_deadline;
```

Además es importante destacar que son los mismos tres argumentos que hemos estado **inyectando** al constructor de `RouteSpecification`.

Pero no nos dispersemos. Veamos como queda nuestro código tras estas tres lineas:

```
Because of = () =>
   catch_exception(() =>
      sut.create_route_specification_using(the_origin_location,
                                           the_destination_location,
                                           the_arrival_deadline));
```

Mucho mejor, ahora solo está en rojo lo que debe estar en rojo. Es evidente que ese `create_route_specification_using()` es el método a través del cual ejercitaremos la creación de objetos. Más diseño, por lo tanto.

Tenemos que implementar su armazón:

```
public class RouteSpecificationFactory
{
    public IRouteSpecification create_route_specification_using(
                                  ILocation the_origin_location,
                                  ILocation the_destination_location,
                                  IArrivalDeadline the_arrival_deadline)
    {
        throw new NotImplementedException();
    }
}
```

Ahí lo tenemos.

El método `create_route_specification_using()` crea objetos del tipo `IRouteSpecification`. Aquí si nos aprovechamos de la interface.

Revisamos nuestro bloque `BECAUSE`:

```
Because of = () =>
   catch_exception(() =>
      sut.create_route_specification_using(the_origin_location,
                                           the_destination_location,
                                           the_arrival_deadline));
```

Ya no hay código rojo, lo que nos permite comentar ese `catch_exception(() => mi_metodo())`, que lo único que hace es capturar la excepción que lanza `mi_método()`; en este caso `create_route_specification_using()`.

Una vez aclarado esto, podemos finalizar nuestra **BDD_Spec** estableciendo el contexto en el que se va a desarrollar, a través de un bloque `ESTABLISH`. Puesto que no hay nada excesivamente interesante en el, presentamos la **BDD_Spec** completa:

```
public class when_attempting_to_inject_a_null_origin_location
             _into_the_route_specification_factory :
             concern_for_route_specification_factory
{
    Establish context = () =>
    {
        the_origin_location = null;
        the_destination_location = an<ILocation>();
        the_arrival_deadline = an<IArrivalDeadline>();
    };

    Because of = () =>
       catch_exception(() =>
          sut.create_route_specification_using(the_origin_location,
                                               the_destination_location,
                                               the_arrival_deadline));

    It should_throw_a_null_argument_exception = () =>
       exception_thrown_by_the_sut.ShouldBeAn<ArgumentNullException>();

    It should_throw_an_invariant_violated_exception_message = () =>
       exception_thrown_by_the_sut
          .ShouldContainErrorMessage(
             "Invariant Violated: origin location is required.");

    static ILocation the_origin_location;
    static ILocation the_destination_location;
    static IArrivalDeadline the_arrival_deadline;
}
```

Para provocar la situación buscada, hacemos que

```
the_origin_location = null;
```

en el bloque `ESTABLISH`.

2.14.1.4 RED en un entorno de captura de excepciones

Ejecutamos la **BDD_Spec** y esperamos los fallos:

```
------ Test started: Assembly: dddsample.specs.dll ------
```

```
when attempting to inject a null origin location into the route
specification factory
» should throw a null argument exception (FAIL)
» should throw an invariant violated exception message (FAIL)

Test 'should throw a null argument exception' failed:
      Machine.Specifications.SpecificationException: Should be of type
System.ArgumentNullException but is of type System.NotImplementedException

(....)

Test 'should throw an invariant violated exception message' failed:
      Machine.Specifications.SpecificationException: Should contain
"Invariant Violated: origin location is required." but is "The method or
operation is not implemented."

(....)


0 passed, 2 failed, 0 skipped, took 1,06 seconds (Machine.Specifications
0.3.0).
```

Vamos a analizar, por un instante, los fallos obtenidos.

Empezamos por:

```
Should be of type System.ArgumentNullException but is of type
System.NotImplementedException
```

Que es justo lo que queremos ya que, en estos momentos, nuestro método lanza:

```
throw new NotImplementedException()
```

para facilitarnos la tarea de compilación.

Pero debemos estar muy satisfechos ya que por el momento el test captura la excepción, y es eso lo que nos está diciendo: "he capturado una excepción, pero no es del tipo de la que me dijisteis que debo esperar."

El otro fallo ahonda en el mismo tema:

```
Should contain "Invariant Violated: origin location is required." but is
"The method or operation is not implemented."
```

Es decir, captura el mensaje de la excepción, solo que, el mensaje no es el esperado, puesto que nuestro método lanza la siguiente excepción con el único propósito de poder compilar el código:

```
throw new NotImplementedException()
```

Esa excepción contiene el mensaje por defecto:

```
The method or operation is not implemented.
```

Por lo tanto todo va bien.

2.14.1.5 GREEN en un entorno de captura de excepciones

Vamos ahora a implementar el comportamiento deseado:

```
public IRouteSpecification create_route_specification_using(
                              ILocation the_origin_location,
                              ILocation the_destination_location,
                              IArrivalDeadline the_arrival_deadline)
{
    if (the_origin_location == null)
       throw new ArgumentNullException(
                     "the_origin_location",
                     "Invariant Violated: origin location is required.");
}
```

Con esto deberíamos pasar el test, si no fuese por el "insignificante detalle" de que no compila, ya que no devolvemos un `IRouteSpecification` (de hecho no devolvemos nada) y aún por encima no tenemos una aserción para comprobarlo.

Aquí es donde aplicamos la versión más pragmática de

> *"Escribir únicamente el mínimo código necesario Y QUE ADEMÁS, TENGA SENTIDO, para que el test pase."*

ya que, precisamente la tarea de una **DDD_Factory** es ocultar al cliente detalles de la construcción de un objeto que necesita (no es la única tarea de una **DDD_Factory**, pero

por el momento nos vale).

Dicho de otra forma, vamos a tener un dependencia oculta.

Por lo tanto:

```
public IRouteSpecification create_route_specification_using(
                              ILocation the_origin_location,
                              ILocation the_destination_location,
                              IArrivalDeadline the_arrival_deadline)
{
    if (the_origin_location == null)
       throw new ArgumentNullException(
                    "the_origin_location",
                    "Invariant Violated: origin location is required.");

    return new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline);
}
```

Y esto sí que pasa el test:

```
------ Test started: Assembly: dddsample.specs.dll ------

when attempting to inject a null origin location into the route
specification factory
» should throw a null argument exception
» should throw an invariant violated exception message


2 passed, 0 failed, 0 skipped, took 0,67 seconds (Machine.Specifications
0.3.0).
```

2.14.2 RouteSpecificationFactorySpecs - When attempting to inject a null destination location into the route specification factory y When attempting to inject a null arrival deadline into the route specification factory

Ambas son muy similares a la que acabamos de hacer.

Recordemos sus **BDD_Specs**:

```
When attempting to inject a null destination location into the route
specification factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject a null arrival deadline into the route
specification factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.
```

No tienen mayor interés. Se pueden consultar en el código fuente en caso de duda.

El resultado es:

```
public IRouteSpecification create_route_specification_using(
                              ILocation the_origin_location,
                              ILocation the_destination_location,
                              IArrivalDeadline the_arrival_deadline)
{
    if (the_origin_location == null)
       throw new ArgumentNullException(
                    "the_origin_location",
                    "Invariant Violated: origin location is required.");

    if (the_destination_location == null)
       throw new ArgumentNullException(
                    "the_destination_location",
                    "Invariant Violated: destination location is required.");

    if (the_arrival_deadline == null)
       throw new ArgumentNullException(
                    "the_arrival_deadline",
                    "Invariant Violated: arrival deadline is required.");

    return new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline);
}
```

2.14.3 RouteSpecificationFactorySpecs - When attempting to inject the same origin and destination locations into the route specification factory

Esta **BDD_Spec** es más interesante:

```
When attempting to inject the same origin and destination location into
the route specification factory
   It should leverage the origin location value comparer.
   It should throw an argument exception.
   It should throw an invariant violated exception message.
```

La segunda y tercera aserción, no merecen la pena ser comentadas (más de lo mismo), pero la primera nos plantea un desafío curioso.

Veamos el código de la **BDD_Spec** suprimiendo la segunda y tercera aserción por cuestión de claridad (en el código fuente se puede consultar completa):

```
public class when_attempting_to_inject_the_same_origin
             _and_destination_locations_into_the_route_specification_factory :
             concern_for_route_specification_factory
{
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = an<IArrivalDeadline>();

        the_origin_location
           .Stub(x => x.has_the_same_value_as(the_destination_location))
           .Return(true);
    };

    Because of = () =>
       catch_exception(() =>
          sut.create_route_specification_using(the_origin_location,
                                               the_destination_location,
                                               the_arrival_deadline));

    It should_leverage_the_origin_location_value_comparer = () =>
       the_origin_location
          .received(x => x.has_the_same_value_as(the_destination_location));

    static ILocation the_origin_location;
    static ILocation the_destination_location;
    static IArrivalDeadline the_arrival_deadline;
}
```

2.14.3.1 Analizando la situación

En principio no parece nada fuera de lo común:

- Tenemos los tres argumentos de la factoría mockados.
- Esperamos que se compare el origen y el destino y los forzamos a que sean iguales.
- Capturamos la excepción en el `BECAUSE` porque hace falta para las aserciones que no mostramos.
- Establecemos que se ha debido llamar al comparador a través del ya conocido `.received()`.

El código que soluciona esto es bien sencillo:

```
public IRouteSpecification create_route_specification_using(
                              ILocation the_origin_location,
                              ILocation the_destination_location,
                              IArrivalDeadline the_arrival_deadline)
{
    if (the_origin_location == null)
       throw new ArgumentNullException(
                    "the_origin_location",
                    "Invariant Violated: origin location is required.");

    if (the_destination_location == null)
       throw new ArgumentNullException(
                    "the_destination_location",
                    "Invariant Violated: destination location is required.");

    if (the_arrival_deadline == null)
       throw new ArgumentNullException(
                    "the_arrival_deadline",
                    "Invariant Violated: arrival deadline is required.");

    if (the_origin_location.has_the_same_value_as(the_destination_location))
        throw new ArgumentException(
                    "Invariant Violated: origin and destination locations
                     can't be the same.");

    return new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline);
}
```

Ejecutamos el test y:

```
   ------ Test started: Assembly: dddsample.specs.dll ------
```

```
when attempting to inject the same origin and destination locations into
the route specification factory
» should leverage the origin location value comparer (FAIL)
» should throw an argument exception
» should throw an invariant violated exception message

Test 'should leverage the origin location value comparer' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException:
IValueObject`1.has_the_same_value_as(ILocationProxycd18dd3bd42e4f979f7fef7
ecd751b48); Expected #1..2147483647, Actual #0.

(....)


2 passed, 1 failed, 0 skipped, took 0,83 seconds (Machine.Specifications
0.3.0).
```

Obtenemos un fallo (**NO ESPERADO**) del tipo:

```
Expected #1, Actual #0
```

2.14.3.2 Qué ocurre cuando nos pasamos con las Specifications

Vamos a seguir el razonamiento:

El tipo de error nos indica que

```
the_origin_location.has_the_same_value_as(the_destination_location)
```

no se ha ejecutado.

Por lo tanto no ha entrado dentro del `if` para lanzar la excepción, así que, deberían haber fallado los 3 tests... pero solo falla uno.

¿Que es lo que ocurre?

Al ejercitar el SUT a través del bloque `BECAUSE`, capturamos la excepción. Es precisamente esta captura la que provoca el fallo no deseado. En realidad hace lo que queremos y esperamos pero, la mecánica que tenemos para comprobarlo, nos lo imposibilita. Podemos comprobar este último aspecto con el **debugger**.

Éste es un buen momento para plantearnos si no estábamos asertando de más.

Si consideramos que implícitamente hemos comprobado que esa llamada se produce, puesto que se lanza la excepción, podemos concluir que el problema estriba en que jamás debimos intentar hacer la comprobación explícita.

Por lo tanto nuestra **BDD_Spec** debería ser ésta:

```
When attempting to inject the same origin and destination location into
the route specification factory looking for the exception
   It should throw an argument exception.
   It should throw an invariant violated exception message.
```

Y ya la hemos cumplido.

Antes de empaquetar y a **github**, deberíamos hacer todavía unos cuantos cambios.

2.14.4 Usando la DDD_Factory

Empezamos planteándonos la necesidad de una **DDD_Factory** para solucionar la violación del **SRP** que se produciría en el constructor de `RouteSpecification` si empezase a lanzar excepciones y acabamos implementando la **DDD_Factory** guiados por el **BDD**.

¿Y para que hemos creado una **DDD_Factory** si no es para usarla?.

Podemos empezar por `RouteSpecificationSpecs`.

Recordemos qué es lo que tenemos:

```
public abstract class concern_for_route_specification :
                      Observes<IRouteSpecification, RouteSpecification>
{
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = an<IArrivalDeadline>();
```

```
        create_sut_using(() =>
           new RouteSpecification(the_origin_location,
                                  the_destination_location,
                                  the_arrival_deadline));
    }

    protected static ILocation the_origin_location;
    protected static ILocation the_destination_location;
    protected static IArrivalDeadline the_arrival_deadline;
}
```

Necesitamos cambiar el constructor por la **DDD_Factory.**

Algo tan sencillo como:

```
public abstract class concern_for_route_specification :
                      Observes<IRouteSpecification, RouteSpecification>
{
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = an<IArrivalDeadline>();
        the_route_specification_factory = new RouteSpecificationFactory();

        create_sut_using(() =>
           the_route_specification_factory
              .create_route_specification_using(the_origin_location,
                                                the_destination_location,
                                                the_arrival_deadline));
    }

    protected static ILocation the_origin_location;
    protected static ILocation the_destination_location;
    protected static IArrivalDeadline the_arrival_deadline;
    protected static RouteSpecificationFactory the_route_specification_factory;
}
```

Esto es lo único que hay que cambiar en `RouteSpecificationSpecs`, ya que si corremos los tests, obtenemos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for its origin location
» should give back the origin location

when asked for its destination location
```

```
  » should give back the destination location

  when asked for its arrival deadline
  » should give back the arrival deadline

  when asked if the specification is satisfied by an itinerary that
  satisfies the route specification
  » should confirm that the itinerary satisfies the route specification
  » should leverage the itinerary initial departure location
  » should leverage the origin location identity comparer
  » should leverage the itinerary final arrival location
  » should leverage the destination location identity comparer
  » should leverage the itinerary final arrival date
  » should leverage the arrival deadline time check

  when asked if the specification is satisfied by an itinerary with an
  invalid initial departure location
  » should confirm that the itinerary does not satisfy the route
  specification
  » should leverage the itinerary initial departure location
  » should leverage the origin location identity comparer
  » should not leverage the itinerary final arrival location
  » should not leverage the destination location identity comparer
  » should not leverage the itinerary final arrival date
  » should not leverage the arrival deadline time check

  when asked if the specification is satisfied by an itinerary with an
  invalid final arrival location
  » should confirm that the itinerary does not satisfy the route
  specification
  » should leverage the itinerary initial departure location
  » should leverage the origin location identity comparer
  » should leverage the itinerary final arrival location
  » should leverage the destination location identity comparer
  » should not leverage the itinerary final arrival date
  » should not leverage the arrival deadline time check

  when asked if the specification is satisfied by an itinerary with an
  invalid final arrival date
  » should confirm that the itinerary does not satisfy the route
  specification
  » should leverage the itinerary initial departure location
  » should leverage the origin location identity comparer
  » should leverage the itinerary final arrival location
  » should leverage the destination location identity comparer
  » should leverage the itinerary final arrival date
  » should leverage the arrival deadline time check

  when asked if the specification is satisfied by a null itinerary
  » should confirm that the itinerary does not satisfy the route
  specification
  » should not leverage the origin location identity comparer
```

```
» should not leverage the destination location identity comparer
» should not leverage the arrival deadline time check

when asked if two similar route specifications have the same value
» should confirm that they have the same value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should leverage the arrival deadline value comparer

when asked if two route specifications with different origin location have
the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should not leverage the destination location value comparer
» should not leverage the arrival deadline value comparer

when asked if two route specifications with different destination location
have the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should not leverage the arrival deadline value comparer

when asked if two route specifications with different arrival deadline
have the same value
» should confirm that they have different value
» should leverage the origin location value comparer
» should leverage the destination location value comparer
» should leverage the arrival deadline value comparer

when asked if a null route specification has the same value as the current
specification
» should confirm that they have different value
» should not leverage the origin location value comparer
» should not leverage the destination location value comparer
» should not leverage the arrival deadline value comparer

when asked about the route specification hash code
» should calculate the hash code according to a given algorithm
» should leverage the origin location hash code
» should leverage the destination location hash code
» should leverage the arrival deadline hash code


59 passed, 0 failed, 0 skipped, took 1,71 seconds (Machine.Specifications
0.3.0).
```

Maravilloso. No hemos roto nada.

Pero podemos ir un poco más lejos.

En este momento, el constructor de `RouteSpecification` tiene una visibilidad `public` que no deja de ser una invitación a que alguien lo use, mientra que nosotros pretendemos que para construir una `RouteSpecification` tengamos que acceder a través de la **DDD_Factory** creada.

Para solucionar esta incongruencia, basta con cambiar la visibilidad del constructor a `internal`, de forma que desde fuera del `assembly` no podamos ver ni usar el constructor directamente.

Por lo tanto:

```
internal RouteSpecification(ILocation the_origin_location,
                            ILocation the_destination_location,
                            IArrivalDeadline the_arrival_deadline)
```

Ejecutamos todos los tests que hemos escrito hasta ahora para asegurarnos que no hemos roto nada y obtenemos un hermoso:

```
107 passed, 0 failed, 0 skipped, took 1,57 seconds (Machine.Specifications
0.3.0).
```

Ahora sí que empaquetamos y a **github**.

2.15 ArrivalDeadlineSpecs

Antes de poder dar por finalizada la implementación del **DDD_Value_Object** `RouteSpecification`, debemos atar un par de cabos sueltos.

Recordemos que creamos la interface `IArrivalDeadline` para poder **inyectársela** al constructor y poder mockarla.

Ahora sería un buen momento para implementar su comportamiento.

Hasta el momento, ésta es la interface:

```
public interface IArrivalDeadline : IValueObject<IArrivalDeadline>
{
    bool is_afterwards_than(IArrivalDeadline final_arrival_date);
    int GetHashCode();
}
```

Por lo tanto vamos a tener que implementar, al menos, 3 métodos; los dos que aparecen arriba más el que heredamos de `IValueObject<T>`:

```
public interface IValueObject<T>
{
    bool has_the_same_value_as(T the_other_value_object);
}
```

2.15.1 De IArrivalDeadline a IDate

Lo primero de todo es que `IArrivalDeadline` no parece ahora tan buen nombre ya que nuestras **BDD_Specs** podrían tener el siguiente aspecto:

```
When asked if a past arrival deadline is posterior to the current
arrival deadline
   It should confirm that it is posterior.
```

Limita mucho el campo de acción. Da la impresión de que esta clase no valdría para otra cosa que las `arrival_deadlines`, cuando en realidad estamos empaquetando un `DateTime`.

Otra cuestión relevante es que estamos usando:

```
(...) is posterior to (...)
```

Esto nos guiará hacia nuevas refactorizaciones que afecten al nombre que le hemos dado a los métodos de `IArrivalDeadline`.

Vamos, por tanto, a empezar a tomar decisiones y, porqué no decirlo, a rectificar lo que no nos gusta del trabajo que ya hemos hecho.

Hemos decidido cambiar `IArrivalDeadline` por `IDate`, ya que refleja mucho mejor nuestra intención y suena más natural. De hecho la **BDD_Spec** anterior ahora cobra un nuevo sentido:

```
When asked if a past date is posterior to the current date
   It should confirm that it is posterior.
```

Con respecto a que refleje mucho mejor nuestra intención, podemos considerar que cuando introdujimos la interface `IArrivalDeadline`, fue precisamente para evitar tener que tratar directamente con el tipo definido por el **.NET Framework** `Datetime`, que nos limitaba en el diseño y a la hora de establecer expectativas en los mocks. En realidad, lo único que pretendíamos era obtener mayor flexibilidad a la hora de tratar con las fechas.

Para proceder con esta refactorización, el primer paso es cambiar el nombre `IArrivalDeadline` por `IDate` y eso es algo que **ReSharper** hace fabulosamente bien.

A continuación, ejecutamos nuestras **BDD_Specs** para comprobar que no hay nada de lo que tengamos que arrepentirnos.

El segundo paso consiste en cambiar el nombre del método `is_afterwards_than()` por uno que suene mejor como `is_posterior_to()`. De nuevo dejamos a **ReSharper** que se encargue del trabajo sucio.

Volvemos a ejecutar nuestras **BDD_Specs** para comprobar que este último cambio ha ido bien.

Vamos a comprobar, ahora, como han afectado, a la legibilidad de nuestro código, estos dos cambios.

Por ejemplo en la implementación de `is_satisfied_by()` de `CargoSpecs`:

```
public bool is_satisfied_by(IItinerary the_itinerary)
{
    return the_itinerary != null &&
           underlying_origin_location.has_the_same_value_as(
              the_itinerary.initial_departure_location()) &&
           underlying_destination_location.has_the_same_value_as(
              the_itinerary.final_arrival_location()) &&
           underlying_arrival_deadline.is_posterior_to(
              the_itinerary.final_arrival_date());
}
```

Fijémonos en las dos últimas lineas, del código anterior, aquí formateadas en una sola:

```
underlying_arrival_deadline.is_posterior_to(the_itinerary.final_arrival_date())
```

Queda bastante clara la intención.

Veamos como ha afectado el cambio a `IDate`.

Por ejemplo en la signatura del constructor de `RouteSpecification`:

```
internal RouteSpecification(ILocation the_origin_location,
                            ILocation the_destination_location,
                            IDate the_arrival_deadline)
```

No pierde ni un ápice de expresividad. Más bien al contrario, ahora queda perfectamente claro y establecido que `the_arrival_deadline` es un fecha. Pero dejemos de preocuparnos por los cambios que ya hemos hecho y ahora sí, vamos a por las **BDD_Specs** de `IDate`.

2.16 DateSpecs (antes conocidas como ArrivalDeadlineSpecs)

Éstas son todas las **BDD_Specs** que creemos deberían definir el comportamiento deseado.

Primero las encargadas del comportamiento asociado a `is_posterior_to()`:

```
When asked if a past date is posterior to the current date
   It should confirm that it is posterior.

When asked if a future date is posterior to the current date
   It should confirm that it is previous instead.

When asked if the current date is posterior to the current date
   It should confirm that it is not posterior.
```

A continuación, las encargadas del comportamiento asociado a `has_the_same_value_as()`:

```
when comparing two dates with the same attribute
   It should confirm they have the same value.

when comparing two dates with different attribute
   It should confirm they have not the same value.

when comparing any date with a null date
   It should confirm they have not the same value.
```

Después, la encargada del comportamiento `datetime_value()`:

```
When returning the datetime
   It should be the same used in construction.
```

Y finalmente la encargada del comportamiento de `GetHashCode()`:

```
When asked for the hash code
   It should return the hash code based on the construction datetime
hash code.
```

> **NOTA**: La implementación de `Equals()` no se trata aquí, pero si figura en el código fuente. Aun así unas paginas más adelante implementaremos el `Equals()` de RouteSpecification siguiendo los principios del **BDD**.

La **clase base abstracta** que usaremos para los tests es:

```
public abstract class concern_for_date : Observes<IDate, Date>{}
```

Y se convierte únicamente en una pasarela para usar el `Observes<Contract, ClassUnderTest>`.

2.16.1 DateSpecs asociadas a is_posterior_to()

2.16.1.1 Las limitaciones de diseño del .NET Framework

Hasta ahora habíamos intentado basar nuestras aserciones, en las interacciones con las dependencias / colaboradores.

Esto era posible gracias a que estábamos tratando con objetos creados y diseñados por nosotros, pero este tipo de comportamiento se da de bruces con la cruda realidad que representa el **.NET Framework**. Y nos referimos a la falta de interfaces para un montón de tipos, que si se hubiesen diseñado de otra forma, serían fácilmente **mockables** y por lo tanto podríamos realizar aserciones sobre las interacciones con ellos.

Por lo tanto, para sortear el obstáculo, cambiamos ligeramente de paradigma y para las **BDD_Specs** siguientes primaremos el resultado final y no las interacciones que se producen para que lleguemos al mismo.

2.16.1.2 El Happy Day Scenario

De esta forma, esta **BDD_Spec**:

```
When asked if a past date is posterior to the current date
   It should confirm that it is posterior.
```

Se traduce en código por:

```
public class when_asked_if_a_past_date_is_posterior_to_the_current_date :
             concern_for_date
{
    Establish context = () =>
    {
        a_past_date = new Date(DateTime.MinValue);

        create_sut_using(() => new Date(DateTime.Now));
    };

    Because of = () => result = sut.is_posterior_to(a_past_date);

    It should_confirm_that_it_is_posterior = () => result.ShouldBeTrue();

    static bool result;
    static IDate a_past_date;
}
```

Donde cabe destacar que:

- Al no poder mockar `DateTime`, nos vemos en la obligación de crear manualmente el SUT a través de `create_sut_using()`, que antes solo usábamos en caso de que al menos dos de los argumentos del constructor fuesen del mismo tipo, para asegurarnos que el resultado era consistente.
- Únicamente comparamos dos valores concretos de `Datetime`:
  - El valor mínimo vía `a_past_date`.
  - El valor actual a través de la **inyección** en el constructor.
- **Inyectamos** al constructor un `Datetime`. Y es que una cosa es que no podamos mockar tanto como quisiésemos y otra muy distinta renunciar a nuestro principios y dejar de favorecer, siempre que podamos, la **DI** y por tanto el **DIP**.

El código que nos permite pasar el test, es muy sencillo:

```
public class Date : IDate
{
    DateTime underlying_date;
```

```
    public Date(DateTime the_date)
    {
        this.underlying_date = the_date;
    }

    public bool is_posterior_to(IDate the_other_date)
    {
        return this.underlying_date > the_other_date.datetime_value();
    }

(....)

}
```

Hemos visto este tipo de estructura interna de clases hasta la saciedad. Así que no vamos a comentar nada más a este respecto.

2.16.1.3 Los escenarios complementarios

Con este mismo código también pasaríamos los escenarios complementarios al Happy Day Scenario, que venían definidas por las siguientes **BDD_Specs**:

```
When asked if a future date is posterior to the current date
   It should confirm that it is previous instead.

When asked if the current date is posterior to the current date
   It should confirm that it is not posterior.
```

Y que en versión código no ofrecen ninguna sorpresa con respecto a la **BDD_Spec** que acabamos de ver:

```
public class when_asked_if_a_future_date_is_posterior_to_the_current_date :
             concern_for_date
{
    Establish context = () =>
    {
        a_future_date = new Date(DateTime.MaxValue);

        create_sut_using(() => new Date(DateTime.Now));
    };

    Because of = () => result = sut.is_posterior_to(a_future_date);
```

```
    It should_confirm_that_it_is_previous_instead = () =>
       result.ShouldBeFalse();

    static bool result;
    static IDate a_future_date;
}

public class when_asked_if_the_current_date_is_posterior_to_the_current_date :
             concern_for_date
{
    Establish context = () =>
    {
        the_current_date = new Date(DateTime.MaxValue);

        create_sut_using(() => new Date(DateTime.MaxValue));
    };

    Because of = () => result = sut.is_posterior_to(the_current_date);

    It should_confirm_that_it_is_not_posterior = () => result.ShouldBeFalse();

    static bool result;
    static IDate the_current_date;
}
```

No debemos perder de vista que todo el proceso aprendido de RED-GREEN-REFACTOR, todavía es aplicable y si no hemos hecho hincapié en ello en esta sección, ha sido porque nos parecía más relevante centrarnos en el cambio de paradigma sin ninguna distracción.

2.16.2 DateSpecs asociadas a has_the_same_value_as()

Es similar a las anteriores.

Vamos primero con el Happy Day Scenario:

```
when comparing two dates with the same attribute
   It should confirm they have the same value.
```

que en código sería:

```
public class when_comparing_two_dates_with_the_same_attribute :
             concern_for_date
```

```
{
    Establish context = () =>
    {
        the_other_date = new Date(DateTime.MaxValue);
        create_sut_using(() => new Date(DateTime.MaxValue);
    };

    Because of = () => result = sut.has_the_same_value_as(the_other_date);

    It should_confirm_they_have_the_same_value = () => result.ShouldBeTrue();

    static bool result;
    static IDate the_other_date;
}
```

De nuevo, nada de mocks.

Para provocar que dos `IDates` sean iguales, usamos, en ambos casos:

```
DateTime.MaxValue
```

La **BDD_Spec** que representa el primer escenario complementario, es exactamente igual pero con la salvedad de que usaremos dos `IDates` distintas:

```
public class when_comparing_two_dates_with_the_different_attribute :
             concern_for_date
{
    Establish context = () =>
    {
        the_other_date = new Date(DateTime.Now);
        create_sut_using(() => new Date(DateTime.MaxValue);
    };

    Because of = () => result = sut.has_the_same_value_as(the_other_date);

    It should_confirm_they_have_not_the_same_value = () =>
       result.ShouldBeFalse();

    static bool result;
    static IDate the_other_date;
}
```

La que se encarga de comprobar el comportamiento en el segundo escenario complementario, simplemente fuerza la situación descrita por la **BDD_Spec** con un `null`:

```
public class when_comparing_any_date_with_a_null_date : concern_for_date
{
    Establish context = () =>
    {
        the_other_date = null;
        create_sut_using(() => new Date(DateTime.MaxValue);
    };

    Because of = () => result = sut.has_the_same_value_as(the_other_date);

    It should_confirm_they_have_not_the_same_value = () =>
       result.ShouldBeFalse();

    static bool result;
    static IDate the_other_date;
}
```

El método que permite pasar los tests es:

```
public bool has_the_same_value_as(IDate the_other_date)
{
    return the_other_date != null &&
           underlying_date == the_other_date.datetime_value();
}
```

2.16.3 DateSpecs asociadas a datetime_value()

Es la más sencilla de todas.

La **BDD_Spec:**

```
When returning the datetime
   It should be the same used in construction.
```

Y el código:

```
public class when_returning_the_datetime : concern_for_date
{
    Establish context = () =>
    {
        the_datetime = DateTime.Now;

        create_sut_using(() => new Date(the_datetime));
    };
```

```
    Because of = () => result = sut.datetime_value();

    It should_be_the_same_used_in_construction = () =>
       result.ShouldEqual(the_datetime);

    static DateTime result;
    static DateTime the_datetime;
}
```

Nada que no hayamos visto.

El método todavía es más sencillo, si cabe:

```
public DateTime datetime_value()
{
    return underlying_date;
}
```

Vamos con la última, que quizás tenga un poco más de relevancia.

2.16.4 DateSpecs asociadas a GetHashCode()

En ésta, ni siquiera vamos a comprobar que hemos realizado el cálculo correctamente o que hemos delegado el cálculo al objeto correcto. Lo único que haremos es comprobar que el valor coincide con el del objeto a quien le delegamos la responsabilidad de realizar el cálculo (el `Datetime` inyectado).

Al igual que antes, primero la **BDD_Spec**:

```
When asked for the hash code
   It should return the hash code based on the construction datetime
hash code.
```

Y después el código:

```
public class when_asked_for_the_hash_code : concern_for_date
{
    Establish context = () =>
    {
        the_datetime = DateTime.Now;
```

```
        create_sut_using(() => new Date(the_datetime));
    };

    Because of = () => result = sut.GetHashCode();

    It should_return_the_hash_code_based_on
       _the_construction_datetime_hash_code = () =>
       result.ShouldEqual(the_datetime.GetHashCode());

    static int result;
    static DateTime the_datetime;
}
```

Éste sería el código del método que confirma que realizamos la delegación:

```
public override int GetHashCode()
{
    return this.underlying_date.GetHashCode();
}
```

Y con esto finalizamos la implementación del **DDD_Value_Object** `Date`, y quedamos a expensas de finalizar la implementación del **DDD_Value_Object** `RouteSpecification`, en cuanto nos ocupemos del comportamiento asociado al método `Equals()`.

2.17 RouteSpecificationSpecs - Implementando Equals()

Antes de empezar con la implementación del método `Equals()` de `RouteSpecification` debemos dejar claro que nuestra idea es evitar su uso, ya que, para eso `RouteSpecification` es un **DDD_Value_Object** e implementa por tanto un método `has_the_same_value_as()` que desempeña la misma función. A mayores, `has_the_same_value_as()`, a la hora de realizar la comparación de igualdad solo admite dos **DDD_Value_Objects** `RouteSpecification`, en claro contraste con `Equals()` que admite la comparación entre un **DDD_Value_Object** `RouteSpecification` y un objeto de cualquier tipo, abriendo la puerta a un montón de problemas potenciales.

Por esa razón, no debería sorprendernos el hecho de que dentro del propio `Equals()` deleguemos la comparación a `has_the_same_value_as()` siempre que sea seguro.

Queremos, por lo tanto, mostrar como podemos implementar `Equals()` siguiendo nuestro enfoque **BDD**.

Vamos con nuestra primera **BDD_Spec**.

2.17.1 El Happy Day Scenario

La definición formal de la **BDD_Spec** es muy sencilla y representa el Happy Day Scenario:

```
When comparing two route specifications with the same attributes using
equals
   It should leverage the route specification value object comparer.
   It should confirm they are equal.
```

Vamos a intentar aclarar la primera aserción.

Cuando afirmamos esto:

```
It should leverage the route specification value object comparer.
```

intentamos poner de manifiesto con ese

```
(...) value object comparer
```

que `RouteSpecification`, al ser un **DDD_Value_Object**, ya tiene un método para comparar objetos de tipo `RouteSpecification`.

Por lo tanto, pretendemos que se delegue esa comprobación a ese método (`has_the_same_value_as()`).

Con un poco de código debería quedar claro.

2.17.1.1 Los bloques IT y la comprobación indirecta

Vamos primero con los dos bloques `IT`:

```
public class when_comparing_two_route_specifications
             _with_the_same_attributes_using_equals :
             concern_for_route_specification
{
    It should_leverage_the_handling_activity_value_object_comparer = () =>
    {
        the_origin_location
           .received(x => x.has_the_same_identity_as(the_origin_location));
        the_destination_location
           .received(x => x
           .has_the_same_identity_as(the_destination_location));
        the_arrival_deadline
           .received(x => x.has_the_same_value_as(the_arrival_deadline));
    };

    It should_confirm_they_are_equal = () => result.ShouldBeTrue();

    static bool result;
}
```

La segunda aserción es trivial así que vamos a obviarla. Sin embargo, la primera es muy interesante, ya que consta a su vez de tres comprobaciones. Cada una de esas comprobaciones se encarga de validar que en efecto hemos realizado la llamada a los métodos de igualdad definidos por nosotros para cada uno de las dependencias de nuestra clase.

Recordemos que nuestro SUT se construye a través de la **DDD_Factory** con los siguientes parámetros:

```
the_route_specification_factory
   .create_route_specification_using(the_origin_location,
                                     the_destination_location,
                                     the_arrival_deadline));
```

De forma que son esos tres parámetros los que validamos.

Sin embargo, habíamos quedado en que pretendíamos delegar la comprobación a `has_the_same_value_as()` de `IRouteSpecification`. Y eso no es lo que ocurre en la primera aserción.

De hecho si eso ocurriese, nuestro bloque `IT` sería algo parecido a:

```
It should_leverage_the_route_specification_value_object_comparer = () =>
   sut.received(x => x
      .has_the_same_value_as(the_other_route_specification));

static IRouteSpecification the_other_route_specification;
```

Eso es lo que nosotros pretendemos hacer y de hecho sería la traducción directa de la **BDD_Spec**.

Probablemente podríamos haber empezado la sección con ese bloque `IT`. ¿Por qué no lo hemos hecho?

La respuesta la tenemos en que por desgracia nuestro SUT no es un mock y no podemos asertar algo como:

```
sut.received(x => x.has_the_same_value_as(...));
```

Sin embargo eso no supone que no podamos testar exactamente ese comportamiento y para ello debemos recordamos la implementación de `has_the_same_value_as()`:

```
public bool has_the_same_value_as(IRouteSpecification the_other)
{
```

```
    return the_other != null &&
           underlying_origin_location
              .has_the_same_identity_as(the_other.origin()) &&
           underlying_destination_location
              .has_the_same_identity_as(the_other.destination()) &&
           underlying_arrival_deadline
              .has_the_same_value_as(the_other.arrival_deadline());
}
```

Es decir, si queremos saber si se produce la llamada a `has_the_same_value_as()` y no podemos comprobarlo de una forma directa (y sencilla) con:

```
sut.received(x => x.has_the_same_value_as(...));
```

lo que si podemos hacer es comprobarlo de una manera indirecta, comprobando que se producen las llamada a los tres métodos de igualdad de los tres objetos dentro de `has_the_same_value_as()`:

```
underlying_origin_location
   .has_the_same_identity_as(the_other.origin()) &&
underlying_destination_location
   .has_the_same_identity_as(the_other.destination()) &&
underlying_arrival_deadline
   .has_the_same_value_as(the_other.arrival_deadline());
```

Y si nos fijamos de nuevo en nuestro bloque `IT`, observamos que eso es exactamente lo que hacemos:

```
It should_leverage_the_handling_activity_value_object_comparer = () =>
{
    the_origin_location
       .received(x => x.has_the_same_identity_as(the_origin_location));
    the_destination_location
       .received(x => x.has_the_same_identity_as(the_destination_location));
    the_arrival_deadline
       .received(x => x.has_the_same_value_as(the_arrival_deadline));
};
```

Esa es la razón por la cual nuestro bloque `IT` consta de tres validaciones, en claro contraste a tener tres bloque `IT` individuales.

Esas tres comprobaciones se encargan de un solo concepto, la llamada a `sut.has_the_same_value_as()`, que no puede ser comprobada de forma directa.

Una última cuestión, la comprobación es del estilo:

```
the_origin_location.has_the_same_identity_as(the_origin_location));
```

en vez de:

```
the_origin_location.has_the_same_identity_as(another_origin_location));
```

ya que esta **BDD_Spec** simula el Happy Day Scenario y por tanto esperamos que todo vaya bien y las dos `RouteSpecifications` sean iguales.

Esperamos que con esto haya quedado aclarado.

2.17.1.2 El bloque BECAUSE

Vamos con el bloque `BECAUSE`:

```
Because of = () => result = sut.Equals(the_other_route_specification);

static IRouteSpecification the_other_route_specification;
```

Aparentemente no debería suponer ningún desafío, pero sin embargo hay un pequeño matiz que debemos tener en cuenta. Cuando efectuamos la llamada a `Equals()` efectuamos una llamada al `Equals()` de `Object` no de `IRouteSpecification`. Nosotros queremos hacer la llamada al `Equals()` de `IRouteSpecification`. Por lo tanto debemos añadir el método `Equals()` a esa interface:

```
public interface IRouteSpecification : ISpecification<IItinerary>,
                                       IValueObject<IRouteSpecification>
{
    ILocation origin();
    ILocation destination();
    IDate arrival_deadline();
    int GetHashCode();
    bool Equals(object the_to_compare_object);
}
```

Ahora si que estamos haciendo lo que queremos.

2.17.1.3 El bloque ESTABLISH

Finalmente tendríamos el bloque `ESTABLISH`, donde:

- Crearemos `the_other_route_specification` a través de la **DDD_Factory**.
- Simularemos que en efecto los tres parámetros son iguales con tres mocks.

En código:

```
Establish context = () =>
{
    the_other_route_specification =
       the_route_specification_factory
          .create_route_specification_using(the_origin_location,
                                            the_destination_location,
                                            the_arrival_deadline);

    the_origin_location
       .Stub(x => x.has_the_same_identity_as(the_origin_location))
       .Return(true);
    the_destination_location
       .Stub(x => x.has_the_same_identity_as(the_destination_location))
       .Return(true);
    the_arrival_deadline
       .Stub(x => x.has_the_same_value_as(the_arrival_deadline))
       .Return(true);
};
```

2.17.1.4 El esqueleto de Equals()

Antes de ejecutar la **BDD_Spec**, deberíamos crear el armazón de `Equals()` en `RouteSpecification` y obligarlo a devolver una excepción:

```
public override bool Equals(object the_to_compare_object)
{
    throw new NotImplementedException();
}
```

2.17.1.5 RED

Ahora si que podemos ejecutar nuestra **BDD_Spec** esperando nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two route specifications with the same attributes using
equals
» should leverage the route specification value object comparer (FAIL)
» should confirm they are equal (FAIL)

Test 'should leverage the route specification value object comparer'
failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

Test 'should confirm they are equal' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)


0 passed, 2 failed, 0 skipped, took 1,08 seconds (Machine.Specifications
0.3.0).
```

Exactamente lo que esperábamos:

```
The method or operation is not implemented.
```

2.17.1.6 GREEN

Ahora debemos implementar la funcionalidad.

En este caso es muy sencillo ya que lo único que queremos es delegar en `has_the_same_value_as()`, pero con la particularidad de que debemos hacer una conversión de tipos para que `has_the_same_value_as()` admita el argumento:

```
public override bool Equals(object the_to_compare_object)
{
    return this.has_the_same_value_as(
                   the_to_compare_object as IHandlingActivity);
}
```

Así de simple.

Si ejecutamos ahora nuestra **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two route specifications with the same attributes using
equals
» should leverage the route specification value object comparer
» should confirm they are equal


2 passed, 0 failed, 0 skipped, took 0,65 seconds (Machine.Specifications
0.3.0).
```

Ya lo tenemos.

Hemos logrado comprobar indirectamente que:

> *It should leverage the route specification value object comparer.*

2.17.2 El contrario del Happy Day Scenario

En esta **BDD_Spec** pretendemos comprobar que también se cumple el contrario del Happy Day Scenario. Por lo tanto estaremos en el supuesto de que los dos `RouteSpecifications` que vamos a comparar son distintos:

> *When comparing two route specifications with the different attributes using equals*
>
> *It should leverage the route specification value object comparer.*
>
> *It should confirm they are different.*

El código referente a los bloques `IT`, cambia de forma un tanto sutil, con respecto a la anterior **BDD_Spec**:

```
public class when_comparing_two_route_specifications
             _with_different_attributes_using_equals :
             concern_for_route_specification
{
```

```
    It should_leverage_the_route_specification_value_object_comparer = () =>
    {
        the_origin_location
           .received(x => x.has_the_same_identity_as(the_origin_location));
        the_destination_location
           .received(x => x
           .has_the_same_identity_as(the_destination_location));
        the_arrival_deadline
           .never_received(x => x.has_the_same_value_as(the_arrival_deadline));
    };

    It should_confirm_they_are_different = () => result.ShouldBeFalse();

    static bool result;
}
```

Vamos con el primer cambio:

```
ANTERIOR ("Happy Day Scenario")

It should_confirm_they_are_equal = () => result.ShouldBeTrue();

ACTUAL

It should_confirm_they_are_different = () => result.ShouldBeFalse();
```

Ahora comprobamos que sean diferentes y antes comprobábamos que fuesen iguales. Así de simple.

Vamos con el segundo cambio, que va a ser más interesante:

```
ANTERIOR ("Happy Day Scenario")

It should_leverage_the_handling_activity_value_object_comparer = () =>
{
    the_origin_location
       .received(x => x.has_the_same_identity_as(the_origin_location));
    the_destination_location
       .received(x => x.has_the_same_identity_as(the_destination_location));
    the_arrival_deadline
       .received(x => x.has_the_same_value_as(the_arrival_deadline));
};

ACTUAL

It should_leverage_the_handling_activity_value_object_comparer = () =>
{
    the_origin_location
```

```
       .received(x => x.has_the_same_identity_as(the_origin_location));
    the_destination_location
       .received(x => x.has_the_same_identity_as(the_destination_location));
    the_arrival_deadline
       .never_received(x => x.has_the_same_value_as(the_arrival_deadline));
};
```

Es más fácil de apreciar aquí:

```
ANTERIOR ("Happy Day Scenario")

the_arrival_deadline
   .received(x => x.has_the_same_value_as(the_arrival_deadline));

ACTUAL

the_arrival_deadline
   .never_received(x => x.has_the_same_value_as(the_arrival_deadline));
```

La única diferencia es que antes esperábamos que se llamase a `has_the_same_value_as()`, mientras que ahora esperamos que esa llamada no se produzca.

Para entender esto, debemos volver a recordar qué es lo que estamos haciendo y nos es otra cosa que:

```
sut.received(x => x.has_the_same_value_as(...));
```

Pero indirectamente, pues nuestro SUT no es un mock y por tanto no podemos comprobarlo directamente.

La intención tanto en la **BDD_Spec** anterior como en ésta es la misma. Entonces ¿por que cambiamos el código? ¿no debería ser el mismo código?.

Por desgracia la respuesta a ambas preguntas es que NO y la aclaración de por qué es así, la dejamos para cuando lleguemos al bloque `ESTABLISH`, ya que entonces será mucho más fácil de entender.

Seguimos con el bloque `BECAUSE`, que es el mismo que en la **BDD_Spec** anterior:

```
Because of = () => result = sut.Equals(the_other_route_specification);
```

Y llegamos al bloque `ESTABLISH`, que esperamos nos ayude con las respuestas. En este caso, el contexto que necesitamos establecer debe:

- Crear `the_other_route_specification` a través de la **DDD_Factory**.
- Simular que en efecto *al menos uno de los tres parámetros es diferente* con mocks.

Vamos con el código:

```
Establish context = () =>
{
    the_other_route_specification =
       the_route_specification_factory
          .create_route_specification_using(the_origin_location,
                                            the_destination_location,
                                            the_arrival_deadline);

    the_origin_location
       .Stub(x => x.has_the_same_identity_as(the_origin_location))
       .Return(true);
    the_destination_location
       .Stub(x => x.has_the_same_identity_as(the_destination_location))
       .Return(false);
};
```

Cuando simulamos el comportamiento de `the_destination_location` en esta **BDD_Spec**, devolvemos `false`:

```
ANTERIOR ("Happy Day Scenario")

the_destination_location
   .Stub(x => x.has_the_same_identity_as(the_destination_location))
   .Return(true);

ACTUAL

the_destination_location
   .Stub(x => x.has_the_same_identity_as(the_destination_location))
   .Return(false);
```

Esa es la clave ya que al haber implementado `has_the_same_value_as()` con el operador `&&`, en cuanto encuentra un `false` deja de comprobar el resto de las

condiciones.

Por lo tanto si encuentra un `false` al comprobar `the_destination_location` no comprobará `the_arrival_deadline` (de ahí también la razón de que no implementemos un mock para él):

```
the_arrival_deadline
   .never_received(x => x.has_the_same_value_as(the_arrival_deadline));
```

Ya hemos resuelto el misterio.

Ahora hay que comprobar que tenemos razón.

Si ejecutamos la **BDD_Spec** debería pasar:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two route specifications with different attributes using
equals
» should leverage the route specification value object comparer
» should confirm they are different


2 passed, 0 failed, 0 skipped, took 0,78 seconds (Machine.Specifications
0.3.0).
```

Pasa.

Podría ser un falso positivo pero en este caso no lo es. El tema de como hacer fallar una **BDD_Spec** para comprobarlo está ampliamente documentado a lo largo de este Estudio. Así que evitaremos distraernos en este momento con eso.

2.17.3 Null y Equals()

Seguimos complementando al Happy Day Scenario.

Esta vez con:

```
When comparing any route specifications with a null route specification
using equals
   It should leverage the route specification value object comparer.
   It should confirm they are different.
```

Los bloques `IT` nos deparan de nuevo sorpresas:

```
public class when_comparing_any_route_specifications
             _with_a_null_route_specification_using_equals :
             concern_for_route_specification
{
    It should_leverage_the_route_specification_value_object_comparer = () =>
    {
        the_origin_location
           .never_received(x => x
           .has_the_same_identity_as(the_origin_location));
        the_destination_location
           .never_received(x => x
           .has_the_same_identity_as(the_destination_location));
        the_arrival_deadline
           .never_received(x => x.has_the_same_value_as(the_arrival_deadline));
    };

    It should_confirm_they_are_different = () => result.ShouldBeFalse();

    static bool result;
}
```

Ahora esperamos un `.never_received()` en los tres parámetros ya que vamos a enviar un `null` para comparar y por tanto debería solucionarse en la primera condición:

```
public bool has_the_same_value_as(IRouteSpecification the_other)
{
    return
       the_other != null &&        (CONDICION 1)
       underlying_origin_location
          .has_the_same_identity_as(the_other.origin()) &&
       (el resto de condiciones)
}
```

Por lo tanto no vamos a recibir las otras tres llamadas.

Esto simplifica enormemente el bloque `ESTABLISH`, como podemos apreciar aquí (junto al bloque `BECAUSE` que vuelve a ser el mismo de las dos **BDD_Specs** anteriores):

```
Establish context = () =>
{
    the_other_route_specification = null;
};

Because of = () => result = sut.Equals(the_other_route_specification);

static IRouteSpecification the_other_route_specification;
```

Si ejecutamos la **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing any route specifications with a null route specification
using equals
» should leverage the route specification value object comparer
» should confirm they are different


2 passed, 0 failed, 0 skipped, took 1,06 seconds (Machine.Specifications
0.3.0).
```

Funciona.

2.17.4 Dos objetos de distinto tipo

Vamos ahora con una de las características más cuestionables de `Equals()`, permitir la comparación de dos objetos que ni siquiera son del mismo tipo:

```
When comparing any route specifications with a different type object
using equals
   It should leverage the route specification value object comparer.
   It should confirm they are different.
```

Los bloques `IT` coinciden con los de la **BDD_Spec** anterior, mientras que el bloque `BECAUSE` cambia gracias a un matiz que establecemos en el bloque `ESTABLISH`:

```
Establish context = () =>
{
    not_a_route_specification = an<ILeg>();
};

Because of = () => result = sut.Equals(not_a_route_specification);
```

```
static ILeg not_a_route_specification;
```

El `not_a_route_specification` podría ser de cualquier tipo mientras fuese distinto a `IRouteSpecification`. Nosotros escogimos `ILeg`, pero podría haber sido cualquier otro.

Lo interesante no es eso, sino el porqué de que esto vaya a funcionar.

Debemos recordar como hemos implementado `Equals()`:

```
public override bool Equals(object the_to_compare_object)
{
    return this.has_the_same_value_as(
                   the_to_compare_object as IRouteSpecification);
}
```

ya que ahí está la clave:

```
the_to_compare_object as IRouteSpecification
```

Si `the_to_compare_object` no es del tipo `IRouteSpecification`, su valor pasa a ser `null`.

Por lo tanto, estaríamos en el mismo caso que en la **BDD_Spec** anterior:

```
public bool has_the_same_value_as(IRouteSpecification the_other)
{
    return
       the_other != null &&        (CONDICION 1)
       underlying_origin_location
          .has_the_same_identity_as(the_other.origin()) &&
       (el resto de condiciones)
}
```

Resolveríamos a través de la CONDICION 1.

Comprobamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing any route specifications with a different type object using
equals
» should leverage the route specification value object comparer
```

```
» should confirm they are different


2 passed, 0 failed, 0 skipped, took 1,04 seconds (Machine.Specifications
0.3.0).
```

Ya está.

Con esto finalizamos la implementación de `Equals()` **BDD Style**.

Capítulo 3: El Value Object Itinerary - Aplicando BDD a codigo preexistente (Legacy Code)

3.1 Introducción

No todo el mundo puede permitirse el lujo de empezar una aplicación desde cero. Muchas veces nos vemos en la obligación de tener que trabajar con código ya existente (**legacy code**). En este caso es imposible aplicar el proceso `RED-GREEN-REFACTOR`, que es el que, muchas veces inconscientemente, hemos ido introduciendo en los capítulos anteriores.

Para casos como estos, es importante disponer de herramientas que nos permitan evaluar la calidad del código escrito, su corrección y minimizar el riesgo de errores no esperados si nos vemos en el caso de tener que refactorizarlo.

La buena noticia es que el **BDD** puede ayudarnos en nuestro cometido, incluso aunque tengamos que variar nuestro `RED-GREEN-REFACTOR` y por tanto, renegar, de nuestra política de Test-First.

Una advertencia. Éste capítulo no debe servir de escusa para saltarnos lo ya aprendido. El **BDD** fue diseñado con los escenarios Test-First en mente. Punto.

Primero escribimos los tests y a continuación escribimos el código que hace que los tests pasen. Esto es así desde los albores del TDD, y gran parte de su valor radica, precisamente, en esa premisa, ya que, un programador mínimamente experimentado, tiende a diseñar mejores sistemas si antes de escribirlos se pone del lado del código cliente y se ve obligado a usarlo.

Esto es, precisamente, lo que hemos venido haciendo desde el principio de este Estudio, en este estricto orden:

1. Nombramos el contexto (`Context / Specification`) a través de la clase que alberga la **BDD_Spec**: "`When ...`".
2. Escribimos las aserciones (Arrange-Act-ASSERT) que nos permiten tener las especificaciones (`Context / Specification`) a través de los bloques `IT`: "`It should ...`".
3. Ejercitamos el SUT (Arrange-ACT-Assert) a través de un bloque `BECAUSE`: "`Because of ... sut ...`". Es aquí precisamente donde nos ponemos del lado

del código cliente. Nos obligamos a tener que hacer la llamada al método que nos va a proporcionar el comportamiento deseado, cuando ni siquiera existe el nombre del método (no digamos ya el código que lo implementa). De esta forma buscamos la frase (o palabra) que expresa nuestra intención de forma más precisa, así como las dependencias del método.

4. Establecemos las condiciones en las cuales va a tener sentido la ejecución del SUT. Qué es lo que necesitamos (ARRANGE-Act-Assert) para poder contextualizar nuestra **BDD_Spec**. Para ello nos valemos de un bloque `ESTABLISH`: "`Establish context ...`".
5. Buscamos qué es lo que tenemos que hacer para que nuestra **BDD_Spec** compile. Este paso lo hemos ido resolviendo poco a poco, fijándonos en el código de color rojo. Sin embargo es ahora cuando debemos darle importancia.
6. Buscamos que el test falle pero de manera significativa (`RED`-`GREEN`-`REFACTOR`), con sentido. No nos vale cualquier fallo, debe ser el fallo esperado. De esta forma comprobamos, entre otras cosas, que el test no se satisface por un error nuestro, de forma que, por ejemplo, el test sea cierto siempre, incluso antes de que escribamos el código.
7. Escribimos el código que nos permite pasar el test.
8. Comprobamos que el código pasa el test (`RED`-`GREEN`-`REFACTOR`).
9. Cambiamos el código tanto como sea necesario para mejorar la legibilidad (`RED`-`GREEN`-`REFACTOR`) ya sea a través de patrones bien establecidos o simplemente buscando cumplir algunos principio básicos que todo buen programador no debería perder de vista (**S.O.L.I.D.**).

Éste es el proceso que nos permite aplicar **BDD** a nuestro software. Así de simple.

Difícilmente estaremos practicando **BDD** si no seguimos los pasos enumerados anteriormente.

Ahora bien, podemos usar las herramientas que nos proporciona el **BDD**, para proveer al código ya escrito de una red de seguridad que nos ayude a realizar cambios o simplemente evaluar qué lo que hace, lo hace bien.

Y esto es precisamente lo que vamos a intentar describir al implementar el

DDD_Value_Object `Itinerary`.

Para ello, y sin que sirva de precedente, vamos a hacer una traducción prácticamente directa del código del ejemplo original en java, de forma que ese será nuestro **legacy code**, e intentaremos anclar esa red, de la que hablábamos antes, a sus cimientos, para luego poder sugerir cambios y evaluar el diseño del mismo.

> **NOTA**: Por si no ha quedado suficientemente claro, en este capítulo (y sólo en este capítulo) vamos a escribir primero el código (se supone que alguien lo había escrito por nosotros y es eso lo que nos encontramos) y luego los tests. Práctica que, no nos cansaremos de advertir, desvirtúa por completo el sentido original del **BDD**.

Vamos con el itinerario.

3.2 Análisis previo

3.2.1 Abstracto vs Concreto

A la hora de transformar el código java para este ejemplo, nos hemos dado cuenta de que ya es muy tarde para poder simplemente "traducir" o "portar" el código directamente.

La razón es muy simple, mientras el código del ejemplo java depende de clases concretas, nosotros dependemos de abstracciones (interfaces), ya que, además de que nos parece una mejor solución (más flexible, entre otras cosas), nos facilita mucho la vida a la hora de practicar **BDD**. De hecho, es lo que nos permite declarar mocks con tanta alegría y testar comportamiento que incluye dependencias a clases que ni siquiera han sido escritas.

Vamos a poner un ejemplo relevante para el caso que estamos estudiando.

Nosotros ya tenemos definido (que no implementado) el comportamiento del **DDD_Value_Object** `Itinerary`. No hemos escrito ni una sola linea de su "más que posible" implementación (`Itinerary`), ni tampoco hemos escrito ni una sola linea de su "más que probable" contenedor de **BDD_Specs** (`ItinerarySpecs`). Y entonces, ¿de donde hemos sacado ese comportamiento?.

Muy sencillo. De las interacciones que se producen en otras clases que pertenecen al mismo **DDD_Aggregate** y que tienen como dependencia al **DDD_Value_Object** `Itinerary` a través de su interface `IItinerary`.

Es decir, hasta ahora hemos visto como al implementar la funcionalidad del **DDD_Aggregate_Root** `Cargo` y del **DDD_Value_Object** `RouteSpecification`, éstas, para realizar su trabajo, delegaban, a veces, en el **DDD_Value_Object** `Itinerary` (insistimos, a través de su interface `IItinerary`).

Al delegar, nosotros íbamos descubriendo y **diseñando**, cuales eran las responsabilidades del **DDD_Value_Object** `Itinerary`, y lo materializábamos en la

interface `IItinerary`. Repetimos, pero sin escribir una sola linea de código del **DDD_Value_Object** `Itinerary`. Es más, la clase `Itinerary` ni siquiera existe.

Veamos cual es ese comportamiento que ya tenemos definido:

```
namespace dddsample.domain.model.cargo.aggregate
{
    public interface IItinerary : IValueObject<IItinerary>
    {
        ILocation initial_departure_location();
        ILocation final_arrival_location();
        IDate final_arrival_date();
        IList<ILeg> legs();
        int GetHashCode();
    }
}
```

Este hecho, dice mucho de la potencia del **BDD** como herramienta de diseño a través de la cual desarrollar nuestras aplicaciones.

Basándonos en ese comportamiento que ya tenemos definido, no podemos hacer una "traducción" directa, ya que necesitamos cumplir las signaturas de los métodos de `IItinerary`.

Por tanto, ya hemos realizado un gran cambio en el proceso de traducción.

Otro ejemplo que nos afecta directamente, y que se podría englobar en el mismo tipo que el anterior es el de `IValueObject<T>`, definido en nuestro caso como:

```
namespace dddsample.domain.shared
{
    public interface IValueObject<T>
    {
        bool has_the_same_value_as(T the_other_value_object);
    }
}
```

De nuevo, hemos optado por usar una interface que debe ser implementada por cualquier clase que sea un **DDD_Value_Object**, en vez de favorecer la técnica de implementar una **clase base** que nos provea de comportamiento con una implementación por defecto de:

```
bool has_the_same_value_as(T the_other_value_object);
```

que parece que es la versión preferida en el ejemplo java.

Por si esto ya no fuese suficiente diferencia, a mayores hemos implementado `IValueObject<T>` a través de la interface que define al tipo, es decir, podíamos haber optado por algo como:

```
public class VoyageExample : IValueObject<VoyageExample>
```

de forma que ligásemos el tipo genérico `<T>` con un tipo concreto `VoyageExample` (es decir, no abstracto, no interface), pero fuimos mucho más allá, ya que nosotros favorecemos:

```
public interface IVoyageExample : IValueObject<IVoyageExample>
```

De forma que ligamos el tipo genérico `<T>` con un tipo abstracto `IVoyageExample`, que se encadenaría de la siguiente forma:

```
public interface IVoyageExample : IValueObject<IVoyageExample>{}

public class VoyageExample : IVoyageExample
{
    public bool has_the_same_value_as(IVoyageExample the_other_value_object)
    {
        throw new NotImplementedException();
    }
}
```

Por tanto, en ese proceso de "traducción" no nos queda más remedio, que adaptar más de lo que quisiéramos para simular el escenario de aplicar **BDD** a código ya escrito (bendito problema, dirían algunos en este caso).

Una vez hechas estas aclaraciones de base, es importante reseñar que, las clases de las que depende el **DDD_Value_Object** `Itinerary` y que no habían sido creadas hasta ahora (surgirían como una **reacción en cadena** al ir creando e implementado las **BDD_Specs**), tampoco van a ser clases concretas. Preferimos optar por su versión abstracta, vía interfaces, como hemos hecho hasta ahora.

3.2.2 Nuevas Interfaces - IHandlingEvent

Tenemos un nuevo colaborador:

```
public interface IHandlingEvent {}
```

que debemos situar en un nuevo **DDD_Aggregate** `Handling`.

Por lo tanto:

```
namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent {}
}
```

y además sabemos que `IHandlingEvent` va a ser un **DDD_Domain_Event**, por lo que debe implementar la interface que así lo estipula:

```
namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent : IDomainEvent<IHandlingEvent> {}
}
```

Una vez solucionado esto, podemos ir con los nuevos métodos que surgen en `IHandlingEvent`:

```
namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent : IDomainEvent<IHandlingEvent>
    {
        ILocation location();
        IVoyage voyage();
        IHandlingEventType type();
    }
}
```

3.2.2.1 El método location()

Empezamos por `location()`:

Es un ejemplo extraordinario de como se establecen las relaciones entre objetos que pertenecen a distintos **DDD_Aggregates**:

- `IHandlingEvent` pertenece al **DDD_Aggregate** `Handling`, del que por cierto es el **DDD_Aggregate_Root**.
- `ILocation` pertenece al **DDD_Aggregate** `Location`, del que también es **DDD_Aggregate_Root**.

`IHandlingEvent`, si quiere relacionarse con elementos de otro **DDD_Aggregate**, solo puede hacerlo a través de los **DDD_Aggregate_Roots**.

> **NOTA**: En concreto en este contexto, relacionarse, debe entenderse como la posibilidad de almacenar un campo privado interno de forma permanente.

Por lo tanto, en este caso, vemos que se cumple. Así, por ejemplo, para acceder a la funcionalidad encapsulada en el **DDD_Aggregate** `Location`, se accede vía el **DDD_Aggregate_Root** `Location`.

3.2.2.2 El método voyage()

Vamos con el método `voyage()`, pero antes necesitamos arreglar ese código rojo previo donde aparece `IVoyage`:

```
namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent : IDomainEvent<IHandlingEvent>
    {
        ILocation location();
        IVoyage voyage();
        IHandlingEventType type();
    }
}
```

`IVoyage` va a ser una **DDD_Entity** del **DDD_Aggregate** `Voyage` (no existe todavía), en el cual además, desempeñara el rol de **DDD_Aggregate_Root**.

Por lo tanto, para solucionarlo:

```
namespace dddsample.domain.model.voyage.aggregate
{
    public interface IVoyage : IEntity<IVoyage> {}
}
```

Ahora ya no tenemos código rojo pendiendo sobre la cabeza de `voyage()` (en este caso, hemos decidido mostrar los `using` para marcar de donde viene cada una de las interfaces que estamos viendo):

```
using dddsample.domain.model.location.aggregate;
using dddsample.domain.model.voyage.aggregate;
using dddsample.domain.shared;

namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent : IDomainEvent<IHandlingEvent>
    {
        ILocation location();
        IVoyage voyage();
        IHandlingEventType type();
    }
}
```

Podemos retomar, por tanto, la discusión sobre `voyage()`, al que le podemos aplicar exactamente todo lo dicho sobre `location()`, pero intercambiando `ILocation` por `IVoyage`.

3.2.2.3 El método type()

Por último, nos encontramos con `type()`, que sirve para marcar cual es el tipo de evento asociado a `IHandlingEvent`.

Para poder eliminar el código rojo de `IHandlingEvent`, vamos a necesitar cambiar algunas cosas más, tal y como mostramos a continuación en **Métodos nuevos sobre interfaces viejas**.

3.2.3 Métodos nuevos sobre interfaces viejas

3.2.3.1 El método voyage() de ILeg

Otro método que debemos añadir, es `voyage()` pero esta vez al **DDD_Value_Object** `Leg`, a través de su interface `ILeg` (de nuevo mostramos los `using` para facilitar la comprensión de qué **DDD_Aggregates** están en juego):

```
using dddsample.domain.model.location.aggregate;
using dddsample.domain.model.voyage.aggregate;
using dddsample.domain.shared;

namespace dddsample.domain.model.cargo.aggregate
{
    public interface ILeg : IValueObject<ILeg>
    {
        ILocation load_location();
        ILocation unload_location();
        IDate unload_time();
        IVoyage voyage();
    }
}
```

No parece que sea una sorpresa descubrir que `voyage()` devuelve un `IVoyage`.

Con esto se acaban los métodos nuevos sobre interfaces viejas.

3.2.4 Nuevas Interfaces - IHandlingEventType

Las únicas lineas de código en rojo que nos quedan en la interface `IHandlingEvent` antes de poder aplicarle el tratamiento **BDD** a `Itinerary`, son las relativas a `IHandlingEventType`:

```
using dddsample.domain.model.location.aggregate;
using dddsample.domain.model.voyage.aggregate;
using dddsample.domain.shared;

namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent : IDomainEvent<IHandlingEvent>
    {
        ILocation location();
        IVoyage voyage();
```

```
        IHandlingEventType type();
    }
}
```

Así que procedemos a con su definición:

```
namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEventType {}
}
```

Nada misterioso ni digno de mencionar.

Con esto solucionamos, por fin, `IHandlingEvent`:

```
using dddsample.domain.model.location.aggregate;
using dddsample.domain.model.voyage.aggregate;
using dddsample.domain.shared;

namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent : IDomainEvent<IHandlingEvent>
    {
        ILocation location();
        IVoyage voyage();
        IHandlingEventType type();
    }
}
```

Pero en el código fuente de `Itinerary` todavía tenemos que tomar alguna decisión más, y tiene que ver justo con esta interface (`IHandlingEventType`).

3.2.4.1 Definiendo los tipos de eventos

Ésta es una muestra del código que todavía está en rojo:

```
if (handling_event.type() == HandlingEvent.Type.RECEIVE)

(....)

if (handling_event.type() == HandlingEvent.Type.LOAD)

(....)
```

```
if (handling_event.type() == HandlingEvent.Type.UNLOAD)

(....)

if (handling_event.type() == HandlingEvent.Type.CLAIM)
```

Vamos a explicar qué es esto y porqué queremos solucionarlo de otra forma.

`RECEIVE`, `LOAD`, `UNLOAD` y `CLAIM` son tipos aplicables al **DDD_Domain_Event** `HandlingEvent`. Son, por tanto, los tipos de eventos que se pueden producir.

Para implementarlos, en java, utilizan un `enum`. Podríamos hacer lo mismo aquí, en **C#**. Pero no nos entusiasman demasiado los `enums`. No cuando podemos conseguir los mismo con clases e interfaces.

Ésta es la razón por la que hemos definido la interface `IHandlingEventType` de una forma tan explicita (en java se llama `Type`, sin más). Porque le reservamos un papel ligeramente diferente.

La idea es poder escribir:

```
if (handling_event.type() == HandlingEventType.UNLOAD)
```

en vez de:

```
if (handling_event.type() == HandlingEvent.Type.UNLOAD)
```

Para ello, vamos a mostrar un técnica que nos permite tener una alternativa a los `enums`, y además, aplicar un mínimo de testing.

3.2.4.2 Alternativa a los enums

Empezamos:

Primero

Creamos un tipo derivado de nuestra interface:

```
public class HandlingEventType : IHandlingEventType {}
```

Segundo

Añadimos un constructor privado sin parámetros, de forma que para invocarlo, tengamos que estar dentro de la clase:

```
public class HandlingEventType : IHandlingEventType
{
    private HandlingEventType(){}
}
```

Tercero

Creamos tantos campos `public`, `static` y `readonly` como elementos hay en el `enum`. Además hacemos que cada uno de ellos sea del tipo de la interface (`IHandlingEventType`) y que invoquen a nuestro constructor privado.

Por ejemplo:

```
public static readonly IHandlingEventType LOAD = new HandlingEventType();
```

Cuarto

Movemos la clase al `namespace` correcto.

Resultado final

Éste es el resultado final:

```
namespace dddsample.domain.model.handling.aggregate
{
    public class HandlingEventType : IHandlingEventType
    {
        public static readonly IHandlingEventType RECEIVE =
                                                    new HandlingEventType();
        public static readonly IHandlingEventType LOAD =
                                                    new HandlingEventType();
        public static readonly IHandlingEventType UNLOAD =
                                                    new HandlingEventType();
        public static readonly IHandlingEventType CLAIM =
                                                    new HandlingEventType();
        public static readonly IHandlingEventType CUSTOMS =
```

```
                                                    new HandlingEventType();

        private HandlingEventType(){}
    }
}
```

Ahora ya podemos usar la clase tal y como queríamos:

```
if (handling_event.type() == HandlingEventType.RECEIVE)
```

3.2.4.3 Deficiencias

Es importante darse cuenta de que `HandlingEventType` viola la "**O**" de **S.O.L.I.D.** (**OCP**) flagrantemente, ya qué, si queremos añadir un nuevo tipo de evento, tenemos cerrada la extensión y abierta la modificación (justo al contrario de lo que establece el **Open / Closed Principle**, que nos dice que debemos estar abiertos a la extensión, pero cerrados a la modificación).

Una vez hecha la advertencia, podemos continuar, ya que, pese a todo, estamos contentos con la solución.

3.2.5 El último escollo

Hay un último escollo que tenemos que salvar antes de poder ponernos con las **BDD_Specs**, y es que `Itinerary` hace uso de un objeto creado internamente en `Location` (la implementación de `ILocation`):

```
return Location.UNKNOWN;
```

Para ello usa el siguiente código (traducido a **C#**):

```
public class Location : ILocation
{
    /// <summary>
    /// Special Location object that marks an unknown location.
    /// </summary>
    public static readonly Location UNKNOWN = new Location(
                                                    new UnLocode("XXXXX"),
                                                    "Unknown location");
```

```
    (....)
}
```

Vamos a olvidarnos por un segundo de ese código rojo.

Si analizamos detenidamente el código, debería sonarnos, ya que se parece sospechosamente a la solución que adoptamos hace unos instantes para deshacernos del `enum`.

Esto es lo que nos permite que:

```
return Location.UNKNOWN;
```

tenga un color más saludable.

La labor que desempeña desde el punto de vista de `Itinerary`, no es otra que marcar que `Location.UNKNOWN`.

Preferiríamos no tener una implementación de `ILocation` y en cierta forma aplazar la decisión sin que por ello nuestro código no compile, pero ya que, precisamente vamos a tratar con **legacy code**, nos parece que, mantener esta implementación es una buena maldad, ya que convierte el código cliente que hace uso de `Location.UNKNOWN` en código difícilmente testable desde un punto de vista Mockista.

Lo único que vamos a cambiar va a ser el nombre de la clase, de forma que cuando más adelante implementemos la verdadera `ILocation` siguiendo el paradigma del **BDD** (escribiendo el código de la **BDD_Spec** ANTES), no nos moleste.

Para ello hemos decidido ponerle el nombre de `LocationImplementationExample`:

```
namespace dddsample.domain.model.location.aggregate
{
    public class LocationImplementationExample : ILocation
    {
        /// <summary>
        /// Special Location object that marks an unknown location.
        /// </summary>
        public static readonly LocationImplementationExample UNKNOWN =
           new LocationImplementationExample();
```

```
        public bool has_the_same_identity_as(ILocation the_other_entity)
        {
            throw new NotImplementedException();
        }

        public ILocation location_unknown()
        {
            throw new NotImplementedException();
        }
    }
}
```

Comprobamos que todo compila y que nuestras **BDD_Specs** previas siguen funcionando:

```
125 passed, 0 failed, 0 skipped, took 6,04 seconds (Machine.Specifications
0.3.0).
```

Ahora sí que estamos listos para enfrentarnos al código de `Itinerary`.

3.3 La clase base concern_for_itinerary

En vez de mostrar todo el código de `Itinerary` nada más empezar, hemos decidido que, al comienzo de cada **BDD_Spec**, mostraremos el código que intentamos testar.

Vamos ahora con la **clase base abstracta** que solemos usar como base de nuestras **BDD_Specs**.

En este caso, sería algo así como:

```
public abstract class concern_for_itinerary : Observes<IItinerary,Itinerary> {}
```

Al menos, inicialmente, no le vamos a asignar más comportamiento. Su única responsabilidad, por tanto, será la de servir de pasarela para acceder a las extensiones `developwithpassion` de `machine.specifications` vía `Observes<Contract, ClassUnderTest>`.

3.4 ItinerarySpecs - When asked for the itinerary leg collection

Vamos a empezar con una **BDD_Spec** aparentemente sencilla para ir calentando motores.

3.4.1 El legacy code de legs()

El **legacy code** que queremos probar es:

```
/// <summary>
///
/// </summary>
/// <returns>The legs of this itinerary, as a list.</returns>
public IList<ILeg> legs()
{
    // TODO: Return an IEnumerable.
    return this.underlying_leg_collection;
}
```

Para entenderlo un poco mejor, necesitamos verlo en su contexto (omitimos las partes que no consideramos relevantes para el caso que nos ocupa, así como los comentarios):

```
public class Itinerary : IItinerary
{
    IList<ILeg> underlying_leg_collection = new List<ILeg>();

    public Itinerary(List<ILeg> legs)
    {
        this.underlying_leg_collection = legs;
    }

    public IList<ILeg> legs()
    {
        return this.underlying_leg_collection;
    }
}
```

Tanto la clase `Itinerary` como su constructor, tienen mucho más código, pero éste es el que nos afecta de forma directa.

Vamos a analizar qué es lo que ocurre en esta clase, pero parece que todo gira

alrededor de la colección de `ILegs`:

1. Se inyecta una colección de objetos `ILeg` a través del constructor.
2. Esa colección, en principio, permanece sin que nadie la modifique.
3. Necesitamos devolver la colección a través del método `legs()`.

Un par de puntualizaciones.

Hablando con propiedad deberíamos decir una colección de **DDD_Value_Objects** `ILeg`.

Probablemente sería mucho más inteligente, devolver un `IEnumerable`, que no permite que modifiquen los contenidos de la colección, ya que si nos preocupamos de que dentro de nuestro **DDD_Value_Object** no se toque nada de la colección, maldita la gracia que nos hará si nos la modifican desde el exterior. Ésta es la razón por la que hemos añadido un comentario `TODO` en el código.

3.4.2 Obteniendo la BDD_Spec

Tras este breve análisis, estamos listos para abordar la tarea de crear una **BDD_Spec** que cubra este comportamiento.

Una primera idea:

```
public class when_asked_for_the_itinerary_leg_collection :
             concern_for_itinerary
{
    It should_return_its_collection_of_legs = () =>
       result.ShouldEqual(the_collection_of_legs);
}
```

Parece que se aproxima bastante a nuestra intención, así que ya tenemos **BDD_Spec**:

```
When asked for the itinerary leg collection
   It should return its collection of legs.
```

3.4.3 Assert

Vamos con el código rojo, que se soluciona de forma muy simple:

```
public class when_asked_for_the_itinerary_leg_collection :
             concern_for_itinerary
{
    It should_return_its_collection_of_legs = () =>
       result.ShouldEqual(the_collection_of_legs);

    static IList<ILeg> result;
    static IList<ILeg> the_collection_of_legs;
}
```

Ya tenemos cubierta la tercera A del Arrange-Act-Assert.

3.4.4 Act

Vamos ahora con la segunda A.

A través de un bloque `BECAUSE` ejercitamos el SUT, que necesariamente será la llamada al método `legs()`:

```
public class when_asked_for_the_itinerary_leg_collection :
             concern_for_itinerary
{
    Because of = () => result = sut.legs();

    It should_return_its_collection_of_legs = () =>
       result.ShouldEqual(the_collection_of_legs);

    static IList<ILeg> result;
    static IList<ILeg> the_collection_of_legs;
}
```

3.4.5 Arrange

Finalmente necesitamos establecer las condiciones bajo las cuales esperamos que se cumpla el comportamiento especificado.

Si observamos el código que queremos probar, vemos que necesitamos:

1. Llamar al constructor.
2. Haber creado sus dependencias para poder inyectárselas.

Y aquí nos vamos a encontrar con la primera sorpresa, ya que, al contrario de lo que podría parecer, no vamos a crear un mock. En su lugar, crearemos la dependencia de forma real. Sin embargo sí que utilizaremos un mock para crear un elemento `ILeg` de la colección:

```
public class when_asked_for_the_itinerary_leg_collection :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);

        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };

    Because of = () => result = sut.legs();

    It should_return_its_collection_of_legs = () =>
       result.ShouldEqual(the_collection_of_legs);

    static IList<ILeg> result;
    static IList<ILeg> the_collection_of_legs;
    static ILeg the_first_leg;
}
```

Ya tenemos, por tanto, la primera A del Arrange-Act-Assert.

3.4.6 Breve recapitulación del AAA

Revisemos lo que tenemos:

- Creamos la dependencia, esta vez de forma real (no es un mock).
- Invocamos al constructor inyectándole la dependencia creada.
- Invocamos el método que queremos probar (`legs()`).
- Asertamos que dicho método (`legs()`) nos va a devolver la misma colección que le pasamos vía constructor.

Todo parece correcto.

3.4.7 RED

Que hacemos ahora, ¿ejecutamos la **BDD_Spec** y si todo va bien, se acabó?

Aquí viene la segunda sorpresa. No hemos hecho fallar nuestra **BDD_Spec** y ese es un paso imprescindible (RED-GREEN-REFACTOR).

Para ello necesitamos asertar lo contrario a lo que queremos. Es decir:

- Queremos comprobar que la colección devuelta sea la misma que la inyectada a través del constructor.
- Asertaremos que las dos colecciones no van a ser iguales y esperaremos que nuestro fallo sea exactamente ese.

Y eso es tan sencillo como cambiar esta aserción:

```
It should_return_its_collection_of_legs = () =>
   result.ShouldEqual(the_collection_of_legs);
```

por esta otra:

```
It should_return_its_collection_of_legs = () =>
   result.Should_Not_Equal(the_collection_of_legs);
```

Ahora sí que ejecutamos nuestra **BDD_Spec**, esperando ese fallo:

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   when asked for the itinerary legs collection
   » should return its collection of legs (FAIL)

   Test 'should return its collection of legs' failed:
         Machine.Specifications.SpecificationException: Should not equal
   System.Collections.Generic.List`1[dddsample.domain.model.cargo.aggregate.I
   Leg]:
         {
```

```
       }
        but does:
System.Collections.Generic.List`1[dddsample.domain.model.cargo.aggregate.I
Leg]:
       {

       }

       at Machine.Specifications.ShouldExtensionMethods.ShouldNotEqual[T](T
actual, T expected)
       domain\model\cargo.aggregate\ItinerarySpecs.cs(24,0): at
dddsample.specs.domain.model.cargo.aggregate.when_asked_for_the_itinerary_
legs_collection.<.ctor>b__3()
       at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
       at Machine.Specifications.Model.Specification.Verify()

0 passed, 1 failed, 0 skipped, took 0,63 seconds (Machine.Specifications
0.3.0).
```

Exactamente lo que estábamos buscando:

```
Should not equal (....) but does (....)
```

El test nos está diciendo, que esperaba que NO fuesen iguales, pero sí que son iguales.

Ahí va nuestro RED.

3.4.8 GREEN

Cambiamos el código de la **BDD_Spec** para que, esta vez sí, aserte lo que esperamos, que son iguales:

```
It should_return_its_collection_of_legs = () =>
   result.ShouldEqual(the_collection_of_legs);
```

Y ejecutamos la **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for the itinerary legs collection
```

```
» should return its collection of legs


1 passed, 0 failed, 0 skipped, took 0,57 seconds (Machine.Specifications
0.3.0).
```

Objetivo conseguido. Ya tenenos nuestro GREEN.

3.4.9 REFACTOR

Con respecto al tema del REFACTOR, por el momento, nada se puede hacer.

3.5 ItinerarySpecs - When asked for the initial departure load location

3.5.1 Análisis previo

Éste es el **legacy code** que queremos probar:

```
/// <summary>
///
/// </summary>
/// <returns>The initial departure location.</returns>
public ILocation initial_departure_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN ;
    return underlying_leg_collection[0].load_location();
}
```

Para empezar, vemos que hay dos **RAMAS**:

- **RAMA 1**: Si la colección está vacía nos marca `Location.UNKNOWN`.
- **RAMA 2**: Si no está vacía, nos devuelve lo que pedimos.

Por lo tanto queda claro que vamos a necesitar dos **BDD_Specs** para este código.

De estas dos **RAMAS**, parece claro que es la **RAMA 2**, la que representa el Happy Day Scenario, así que vamos primero a probar eso.

3.5.2 Happy Day Scenario

3.5.2.1 El legacy code de initial_departure_location()

Nuestra intención es comprobar esto:

```
public ILocation initial_departure_location()
{
    return underlying_leg_collection[0].load_location();
}
```

Analizando ese código, podemos concluir que:

- Necesitamos que nos devuelvan una `ILocation` concreta (State-Based Testing TDD).
- Necesitamos comprobar que invocamos `load_location()` (Interaction-Based Testing TDD).

Por lo tanto, necesitaremos dos aserciones distintas

3.5.2.2 Assert

Vamos a intentar escribir en código el armazón de la **BDD_Spec**:

```
public class When_asked_for_the_initial_departure_load_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{
    It should_return_the_legs_collection_first_element_load_location = () =>
       result.ShouldEqual(the_initial_departure_load_location);

    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());
}
```

Vamos primero a solucionar el código rojo, y si esta solución nos gusta, escribiremos la **BDD_Spec**:

- `result` tiene que ser una `ILocation`.
- `the_initial_departure_load_location` también será una `ILocation`.
- `the_first_leg` debería ser una `ILeg`.

```
public class When_asked_for_the_initial_departure_load_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{
    It should_return_the_legs_collection_first_element_load_location = () =>
       result.ShouldEqual(the_initial_departure_load_location);

    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());
```

```
    static ILocation result;
    static ILocation the_initial_departure_load_location;
    static ILeg the_first_leg;
}
```

Parece correcta, ya podemos escribir la **BDD_Spec**.

3.5.2.3 Obteniendo la BDD_Spec

```
When asked for the initial departure load location with a non empty leg
collection
   It should return the legs collection first element load location.
   It should leverage the first leg load location.
```

Fijémonos por un instante, con que nivel de detalle, con que expresividad describe la **BDD_Spec** nuestro sistema.

3.5.2.4 Refactorizando la BDD_Spec

Solo encontramos un pero.

Da la impresión de que las aserciones están al revés, se leería mejor así:

```
When asked for the initial departure load location with a non empty leg
collection
   It should leverage the first leg load location.
   It should return the legs collection first element load location.
```

Mucho mejor, ya que ahora sigue el flujo de la ejecución:

- Primero, localizamos el elemento en la colección y sobre él invocamos el método que nos devuelve su `ILocation`.
- Después, devolvemos esa `ILocation`.

3.5.2.5 Refactorizando Assert

Cambiamos el orden en el código también:

```
public class When_asked_for_the_initial_departure_load_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{
    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());

    It should_return_the_legs_collection_first_element_load_location = () =>
       result.ShouldEqual(the_initial_departure_load_location);

    static ILocation result;
    static ILocation the_initial_departure_load_location;
    static ILeg the_first_leg;
}
```

3.5.2.6 Act

Ahora necesitamos ejercitar el SUT invocando al método que queremos probar:

```
public class When_asked_for_the_initial_departure_load_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{
    Because of = () => result = sut.initial_departure_location();

    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());

    It should_return_the_legs_collection_first_element_load_location = () =>
       result.ShouldEqual(the_initial_departure_load_location);

    static ILocation result;
    static ILocation the_initial_departure_load_location;
    static ILeg the_first_leg;
}
```

3.5.2.7 Arrange

Aquí viene la parte más enrevesada de esta **BDD_Spec**, establecer el contexto en la cual se va a desenvolver.

Vamos a hacer un pequeño análisis previo, para identificar los actores:

- Necesitamos una colección de `ILegs` para **inyectar** al constructor. Un escenario creíble incluiría al menos dos `ILegs`:
  - `the_first_leg` (ya creada).
  - `the_last_leg` (sin crear).
- Necesitamos una `ILocation` que devolver, la ya creada `the_initial_departure_load_location`.
- Necesitamos simular el comportamiento de `the_first_leg`, para que cuando le pregunten, por su `load_location()`, devuelva, precisamente `the_initial_departure_load_location`. Esto lo especificaremos en un mock.

Para evitar mayor confusión, creamos inicialmente los elementos que faltan:

```
static ILeg the_last_leg;
static IList<ILeg> the_collection_of_legs;
```

Y éste podría ser el aspecto de nuestro bloque `ESTABLISH`:

```
Establish context = () =>
{
    the_initial_departure_load_location = an<ILocation>();

    the_collection_of_legs = new List<ILeg>();
    the_first_leg = an<ILeg>();
    the_last_leg = an<ILeg>();
    the_collection_of_legs.Add(the_first_leg);
    the_collection_of_legs.Add(the_last_leg);

    the_first_leg
       .Stub(x => x.load_location())
       .Return(the_initial_departure_load_location);
};
```

Empecemos por el bloque central.

Lo único que hace es:

- Crear la colección (una colección real, no un mock).
- Crear dos mocks para los dos elementos que van a ir en la colección.

- Añadir los elementos a la colección.

El bloque inicial declara que la `ILocation` que queremos devolver es un mock.

El bloque final asigna el comportamiento esperado al primer elemento de la colección. Cuando invoquemos a su método `load_location()`, devolverá el `ILocation the_initial_departure_load_location`.

Pero nos falta algo, tenemos que crear el SUT:

```
create_sut_using(()=> new Itinerary(the_collection_of_legs));
```

3.5.2.8 El código completo de la BDD_Spec

Por lo tanto nuestra **BDD_Spec** en código tendría el siguiente aspecto:

```
public class When_asked_for_the_initial_departure_load_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_initial_departure_load_location = an<ILocation>();

        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_last_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);
        the_collection_of_legs.Add(the_last_leg);

        the_first_leg
           .Stub(x => x.load_location())
           .Return(the_initial_departure_load_location);

        create_sut_using(()=> new Itinerary(the_collection_of_legs));
    };

    Because of = () => result = sut.initial_departure_location();

    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());

    It should_return_the_legs_collection_first_element_load_location = () =>
       result.ShouldEqual(the_initial_departure_load_location);
```

```
    static ILocation result;
    static ILocation the_initial_departure_load_location;
    static ILeg the_first_leg;
    static ILeg the_last_leg;
    static IList<ILeg> the_collection_of_legs;
}
```

Si nos fijamos con atención, la mayor parte del ruido proviene de:

- Crear y poblar la colección que queremos inyectar (5 lineas).
- La declaración de los campos que vamos a usar.

De hecho, sustituyendo esos elementos la **BDD_Spec**, queda mucho más clara:

```
public class When_asked_for_the_initial_departure_load_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_initial_departure_load_location = an<ILocation>();

        create_and_populate_the_to_inject_leg_collection();

        the_first_leg
           .Stub(x => x.load_location())
           .Return(the_initial_departure_load_location);

        create_sut_using(()=> new Itinerary(the_collection_of_legs));
    };

    Because of = () => result = sut.initial_departure_location();

    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());

    It should_return_the_legs_collection_first_element_load_location = () =>
       result.ShouldEqual(the_initial_departure_load_location);

    static ILocation result;
    static ILocation the_initial_departure_load_location;
    static ILeg the_first_leg;
    static ILeg the_last_leg;
    static IList<ILeg> the_collection_of_legs;
}
```

Aun así éste es un ejercicio de programación-ficción, ya que nuestra **BDD_Spec** no es ésta.

3.5.2.9 RED

Ahora necesitamos hacer fallar nuestra **BDD_Spec** de forma significativa.

Para ello invertimos el sentido de las aserciones, pasando de:

```
It should_leverage_the_first_leg_load_location = () =>
   the_first_leg.received(x => x.load_location());

It should_return_the_legs_collection_first_element_load_location = () =>
   result.ShouldEqual(the_initial_departure_load_location);
```

a lo contrario:

```
It should_leverage_the_first_leg_load_location = () =>
   the_first_leg.never_received(x => x.load_location());

It should_return_the_legs_collection_first_element_load_location = () =>
   result.ShouldNotEqual(the_initial_departure_load_location);
```

Y ejecutamos esta versión "negada" de nuestra **BDD_Spec** buscando los fallos:

```
   ------ Test started: Assembly: dddsample.specs.dll ------

  When asked for the initial departure load location with a non empty leg
  collection
  » should leverage the first leg load location (FAIL)
  » should return the legs collection first element load location (FAIL)

  Test 'should leverage the first leg load location' failed:
        Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
  ILeg.load_location(); would not be called, but it was found on the actual
  calls made on the mocked object.

  (....)

  Test 'should return the legs collection first element load location'
  failed:
        Machine.Specifications.SpecificationException: Should not equal
  ILocationProxy16d8b14660404a89ade5c191dbcd43a5 but does:
```

```
(....)

0 passed, 2 failed, 0 skipped, took 0,88 seconds (Machine.Specifications
0.3.0).
```

Parece que esto era lo que queríamos, ya que por un lado:

```
Expected that ILeg.load_location(); would not be called, but it was found
on the actual calls made on the mocked object.
```

Y por otro:

```
Should not equal (....) but does (....)
```

3.5.2.10 GREEN

Volvemos a cambiar nuestra **BDD_Spec** para que, ahora sí, haga lo que se supone y la ejecutamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

When asked for the initial departure load location with a non empty leg
collection
» should leverage the first leg load location
» should return the legs collection first element load location


2 passed, 0 failed, 0 skipped, took 0,73 seconds (Machine.Specifications
0.3.0).
```

Fantástico.

Ya tenemos nuestro Happy Day Scenario testado.

3.5.2.11 REFACTOR

Se nos ocurre una refactorización, pero eso ha de ser después de probar el otro escenario.

3.5.3 El escenario complementario

3.5.3.1 Análisis previo y AAA

Recordemos primero que queremos probar el siguiente código:

```
public ILocation initial_departure_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN ;
}
```

Sería completamente irreal crear la **BDD_Spec** desde cero, ya que, tenemos la **BDD_Spec** del Happy Day Scenario, por lo tanto vamos a partir de ella, suprimiendo lo que ahora no necesitamos:

- Poblar la colección.
- Los elementos que antes poblaban la colección.
- La `ILocation` que devolvíamos.

Suprimiendo esos elementos nos encontraríamos con:

```
public class When_asked_for_the_initial_departure_load_location
             _with_an_empty_leg_collection :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_collection_of_legs = new List<ILeg>();
        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };

    Because of = () => result = sut.initial_departure_location();

    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());

    It should_return_the_legs_collection_first_element_load_location = () =>
        result.ShouldEqual(the_initial_departure_load_location);

    static ILocation result;
    static IList<ILeg> the_collection_of_legs;
}
```

Y no, no nos hemos olvidado de que las aserciones no son correctas (más allá del código rojo).

En este escenario tenemos que comprobar dos cosas:

- Que se produce la comprobación de lista vacía.
- Que devolvemos `Location.UNKNOWN`.

Y aquí viene el problema.

La segunda aserción es sencilla:

```
It should_return_a_location_UNKNOWN = () =>
   result.ShouldEqual(LocationImplementationExample.UNKNOWN);
```

Pero la primera es inviable ya que `the_collection_of_legs` no es un mock. No podemos establecer una expectativa sobre ella del estilo a:

```
the_collection_of_legs.received(x => x.Count);
```

Hay formas de saltarse esto, pero se nos presenta la oportunidad perfecta para evaluar si estamos siendo razonables. Es decir, ¿nos parece razonable explicitar una aserción que compruebe que la colección es una colección vacía, sin elementos, cuando nosotros mismos así la creamos en el bloque `ESTABLISH`?

Creemos que la respuesta debería ser que no, que no es razonable. Por lo tanto nuestra **BDD_Spec** debería contener solo una aserción:

```
public class When_asked_for_the_initial_departure_load_location
             _with_an_empty_leg_collection :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_collection_of_legs = new List<ILeg>();
        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };

    Because of = () => result = sut.initial_departure_location();
```

```
    It should_leverage_the_first_leg_load_location = () =>
       the_first_leg.received(x => x.load_location());

    It should_return_the_legs_collection_first_element_load_location = () =>
        result.ShouldEqual(the_initial_departure_load_location);

    It should_return_a_location_UNKNOWN = () =>
       result.ShouldEqual(LocationImplementationExample.UNKNOWN);

    static ILocation result;
    static IList<ILeg> the_collection_of_legs;
}
```

Esto ya se parece más a lo que buscamos. Parece un buen momento para definir formalmente la **BDD_Spec**.

3.5.3.2 Obteniendo la BDD_Spec

Ésta sería la **BDD_Spec**:

```
When asked for the initial departure load location with an empty leg
collection
   It should return a location UNKNOWN.
```

Sin embargo, creemos que puede ser más especifica así:

```
When asked for the itinerary initial departure location with an empty
leg collection
   It should return a location UNKNOWN.
```

Nos quedamos con esta última.

3.5.3.3 RED

Ahora necesitamos nuestro fallo.

Para ello invertimos el sentido de la aserción, pasando de:

```
It should_return_a_location_UNKNOWN = () =>
```

```
   result.ShouldEqual(LocationImplementationExample.UNKNOWN);
```

a:

```
It should_return_a_location_UNKNOWN = () =>
   result.Should_Not_Equal(LocationImplementationExample.UNKNOWN);
```

Ejecutamos la **BDD_Spec**, a la espera de nuestro fallo, algo así como "me dices que no son iguales pero sí que los son":

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   When asked for the initial departure load location with an empty leg
   collection
   » should return a location UNKNOWN (FAIL)

   Test 'should return a location UNKNOWN' failed:
         System.Reflection.TargetInvocationException: Exception has been
   thrown by the target of an invocation. ---> System.ArgumentNullException:
   The list of legs cannot be empty.
         Parameter name: legs
         domain\model\cargo.aggregate\Itinerary.cs(34,0): at
   dddsample.domain.model.cargo.aggregate.Itinerary..ctor(IList`1 legs)

   (....)

   0 passed, 1 failed, 0 skipped, took 0,60 seconds (Machine.Specifications
   0.3.0).
```

3.5.3.4 Errores no esperados

Malas noticias. No es el fallo que esperábamos.

Estamos recibiendo un error con el que no contábamos:

```
   Exception has been thrown by the target of an invocation. --->
   System.ArgumentNullException: The list of legs cannot be empty.
         Parameter name: legs
         domain\model\cargo.aggregate\Itinerary.cs(34,0): at
   dddsample.domain.model.cargo.aggregate.Itinerary..ctor(IList`1 legs)
```

El constructor de `Itinerary` ha lanzado una excepción con el mensaje:

```
   The list of legs cannot be empty.
```

Indicando que el parámetro que provoca la excepción es `legs`.

Vamos a mostrar el constructor al completo (no como en la primera **BDD_Spec** en la que solo mostrábamos lo que nos interesaba), para ver qué es ese `legs` del que nos habla el error:

```
public Itinerary(IList<ILeg> legs)
{
    if (legs.Count == 0)
       throw new ArgumentNullException("legs", "The list of legs cannot be
empty.");

    ILeg null_leg = null;
    if (legs.Contains(null_leg))
       throw new NoNullAllowedException("The legs list cannot contain null
elements.");

    this.underlying_leg_collection = legs;
}
```

Ahí lo tenemos, justo al principio:

```
if (legs.Count == 0)
   throw new ArgumentNullException("legs", "The list of legs cannot be
empty.");
```

Es decir, el constructor no permite construir un `Itinerary` en base a una colección de `ILegs` vacía.

3.5.3.5 Comprobando que estamos en lo cierto

Nuestra **BDD_Spec** era correcta y para demostrarlo, como ejercicio, vamos a hacer lo siguiente:

- Comentamos la linea de código del constructor que está lanzando la excepción.
- Ejecutamos entonces nuestra **BDD_Spec**.

```
------ Test started: Assembly: dddsample.specs.dll ------

When asked for the initial departure load location with an empty leg
collection
» should return a location UNKNOWN (FAIL)

Test 'should return a location UNKNOWN' failed:
       Machine.Specifications.SpecificationException: Should not equal
dddsample.domain.model.location.aggregate.LocationImplementationExample
but does:

(....)

0 passed, 1 failed, 0 skipped, took 0,58 seconds (Machine.Specifications
0.3.0).
```

Justo lo que esperábamos:

```
Should not equal (....) but does (....)
```

Y si revertimos nuestra aserción a lo que de verdad queremos probar, obtendríamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

When asked for the initial departure load location with an empty leg
collection
» should return a location UNKNOWN


1 passed, 0 failed, 0 skipped, took 0,58 seconds (Machine.Specifications
0.3.0).
```

Que es lo que buscábamos.

3.5.3.6 No necesitamos este escenario

Pero dejemos de hacer programación-ficción, y vayamos a la realidad. Y la realidad nos está indicando que la comprobación que se produce en `initial_departure_location()` es redundante. Todavía más si podemos reforzar el hecho de que una vez asignada por el constructor la `underlying_leg_collection` no puede ser modificada, pasando de:

```
IList<ILeg> underlying_leg_collection = new List<ILeg>();
```

a:

```
readonly IList<ILeg> underlying_leg_collection = new List<ILeg>();
```

Recordar que el `private` en **C#** viene implícito.

Si hacemos esto, podemos eliminar esta **BDD_Spec** y el código que prueba, ya que no es necesario.

> **NOTA**: En un entorno real, antes de tomar una decisión como ésta, deberíamos confirmar sin ningún genero de dudas que la lista interna no es modificable.

3.5.3.7 Refactor 1

Por lo tanto simplificaríamos nuestro código pasando de:

```
/// <summary>
///
/// </summary>
/// <returns>The initial departure location.</returns>
public ILocation initial_departure_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN ;
    return underlying_leg_collection[0].load_location();
}
```

a esto:

```
/// <summary>
///
/// </summary>
/// <returns>The initial departure location.</returns>
public ILocation initial_departure_location()
{
    return underlying_leg_collection[0].load_location();
}
```

Que hemos comprobado con la **BDD_Spec** del Happy Day Scenario.

Por lo tanto, podemos estar más que satisfechos, ya que hemos mejorado el **legacy code**, eliminando el código de la **RAMA 1**, que no se ejecuta nunca y que podría darnos una falsa sensación de seguridad, que hemos mitigado convirtiendo en `readonly` la lista interna de `ILegs`.

3.5.3.8 Refactor 2

Pero aun hay algo más que podemos hacer, una posible refactorización de la que hablábamos al final del Happy Day Scenario.

Este código podría ser más expresivo:

```
underlying_leg_collection[0].load_location();
```

De hecho, por nuestra **BDD_Spec**, sabemos que:

```
underlying_leg_collection[0]
```

es en realidad:

```
the first leg
```

Podemos crear un método interno a la clase `Itinerary` que nos permita fortalecer ese concepto.

Vamos a hacerlo usando **ReSharper** (se puede hacer fácilmente "a mano" también):

Primero

Seleccionamos el código que queremos empaquetar, es decir:

```
underlying_leg_collection[0]
```

Segundo

Invocamos "**Extract Method**" en **ReSharper** con:

```
Ctrl + Alt + M(ethod)
```

Tercero

En "**Method Name**", introducimos `the_initial_leg`.

Cuarto

Comprobamos el resultado:

```
public ILocation initial_departure_location()
{
    return the_initial_leg().load_location();
}

ILeg the_initial_leg()
{
    return underlying_leg_collection[0];
}
```

Quinto

Ejecutamos nuestras **BDD_Specs** para comprobar que no hemos roto nada.

3.6 ItinerarySpecs - When asked for the final arrival unload location

3.6.1 Análisis previo

Este caso va a ser similar al anterior:

```
When asked for the initial departure load location
```

El **legacy code** que queremos probar es el siguiente:

```
/// <summary>
///
/// </summary>
/// <returns>The final arrival location.</returns>
public ILocation final_arrival_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN;
    return lastLeg().unload_location();
}
```

y de rebote necesitamos probar también esto:

```
/// <summary>
///
/// </summary>
/// <returns>The last leg on the itinerary.</returns>
ILeg lastLeg()
{
    if (underlying_leg_collection.Count == 0)
       return null;
    return underlying_leg_collection[underlying_leg_collection.Count - 1];
}
```

ya que, desde `final_arrival_location()` llamamos a `lastLeg()`, siendo `lastLeg()` un método privado.

Por lo tanto no podemos testarlo directamente, a menos que lo testemos desde otro método que lo invoque (nuestro caso) o decidamos hacerlo `public`, con lo que tendríamos una **BDD_Spec** específica para él.

La cuestión es que, este último escenario, cambiaría el modelo, desde el punto de vista del cliente que consume a nuestra clase, ya que estaríamos añadiendo nuevos comportamientos.

3.6.1.1 Posibilidades de Refactorización

En un análisis previo, detectamos una pequeña incongruencia, y es que ambos métodos contienen la misma comprobación.

Y no solo eso. Si esa comprobación se cumpliese, cada uno de ellos devolvería algo diferente:

```
if (underlying_leg_collection.Count == 0)
   return LocationImplementationExample.UNKNOWN;
```

frente a:

```
if (underlying_leg_collection.Count == 0)
   return null;
```

Esto es algo relativamente común, cuando nos enfrentamos a **legacy code**, y probablemente requiera algún tipo de solución por nuestra parte. Pero eso es algo a lo que ya llegaremos.

3.6.2 Continuamos con el análisis

Por el momento vamos a hacer un análisis similar al caso anterior.

Vemos que hay dos **RAMAS**:

- **RAMA 1**: Si la colección está vacía nos marca `Location.UNKNOWN`.
- **RAMA 2**: Si no está vacía, nos devuelve la `ILocation` que pedimos.

Exactamente igual que en el caso anterior, vamos a utilizar dos **BDD_Specs** para testar este código, siendo la **RAMA 2**, la que representa el Happy Day Scenario.

Debemos estar atentos ya que, si pasa algo parecido a lo que pasó en el caso anterior, la comprobación de la **RAMA 1** puede que esté de más.

Vamos con el Happy Day Scenario.

3.6.3 Happy Day Scenario

3.6.3.1 El legacy code de final_arrival_location()

Nuestra intención es comprobar esto:

```
public ILocation final_arrival_location()
{
    return lastLeg().unload_location();
}
```

y por lo tanto de rebote:

```
ILeg lastLeg()
{
    if (underlying_leg_collection.Count == 0)
       return null;
    return underlying_leg_collection[underlying_leg_collection.Count - 1];
}
```

3.6.3.2 El camino hacia la primera refactorización

Aquí se nos plantea la primera disyuntiva, y es que sabemos que si tuviésemos una colección de `ILegs` vacía, nunca llegaríamos a invocar este código en `lastLeg()`:

```
if (underlying_leg_collection.Count == 0)
   return null;
```

De hecho por nuestra experiencia con la **BDD_Spec** anterior, sabemos también que tampoco llegaríamos a invocar este otro código en `final_arrival_location()`:

```
if (underlying_leg_collection.Count == 0)
   return LocationImplementationExample.UNKNOWN;
```

La razón. El constructor del **DDD_Value_Object** `Itinerary`, impide la creación de un `Itinerary`, con una colección de `ILegs` vacía, arrojando la siguiente excepción:

```
if (legs.Count == 0)
   throw new ArgumentNullException("legs",
                                   "The list of legs cannot be empty.");
```

Así que, tampoco tiene mucho sentido, hacer la del avestruz, y mandar nuestro sentido común bajo tierra.

Por lo tanto, no vamos a comprobar esta parte de `lastLeg()`:

```
if (underlying_leg_collection.Count == 0)
   return null;
```

ya que sabemos que es imposible que se ejecute.

Sin embargo tampoco podemos borrarla sin más, puesto que necesitaremos analizar qué otros métodos del **DDD_Value_Object** `Itinerary` invocan a `lastLeg()`, ya que en alguno de esos caso podríamos necesitar esa comprobación.

Primero vamos a hacer que nuestra **BDD_Spec** pase y luego nos ocuparemos de este tema.

3.6.3.3 Continuamos con el legacy code de final_arrival_location()

Vamos por el principio, para determinar las características de nuestra **BDD_Spec**:

- Necesitamos que nos devuelvan una `ILocation` concreta (TDD Tradicional).
- Necesitamos comprobar que invocamos `unload_location()` (TDD Mockista).
- No podemos establecer si invocamos o dejamos de invocar `lastLeg()`, ya que es un método privado.

3.6.3.4 Assert

Usaremos dos aserciones, cuyo código podría ser (esta vez con el código rojo ya resuelto):

```
public class when_asked_for_the_final_arrival_unload_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{
    It should_leverage_the_last_leg_unload_location = () =>
       the_last_leg.received(x => x.unload_location());

    It should_return_the_legs_collection_last_element_unload_location = () =>
       result.ShouldEqual(the_final_arrival_unload_location);

    static ILeg the_last_leg;
    static ILocation result;
    static ILocation the_final_arrival_unload_location;
}
```

3.6.3.5 Obteniendo la BDD_Spec

Por lo que podríamos escribir nuestra **BDD_Spec** como:

```
When asked for the final arrival unload location with a non empty leg
collection
   It should leverage the last leg unload location.
   It should return the legs collection last element unload location.
```

Siguiendo el flujo de la ejecución y de nuestra **BDD_Spec**:

- Localizamos el elemento (el último elemento) en la colección y sobre él invocamos el método que nos devuelve su `ILocation`.
- Devolvemos esa `ILocation`.

3.6.3.6 Act

A la hora de ejercitar el SUT, no nos encontramos con ninguna sorpresa:

```
public class when_asked_for_the_final_arrival_unload_location
             _with_a_non_empty_leg_collection :
```

```
            concern_for_itinerary
{
    Because of = () => result = sut.final_arrival_location();

    It should_leverage_the_last_leg_unload_location = () =>
       the_last_leg.received(x => x.unload_location());

    It should_return_the_legs_collection_last_element_unload_location = () =>
       result.ShouldEqual(the_final_arrival_unload_location);

    static ILeg the_last_leg;
    static ILocation result;
    static ILocation the_final_arrival_unload_location;
}
```

3.6.3.7 Arrange

Al igual que con la **BDD_Spec** anterior, la parte más enrevesada es la que se encarga de establecer el contexto en la cual se va a desenvolver.

El análisis es similar al que se hizo entonces, solo que ahora:

- Invocamos `unload_location()` en lugar de `load_location()`.
- El elemento de la colección que nos interesa es el último y no el primero.

Es decir:

```
Establish context = () =>
{
    the_final_arrival_unload_location = an<ILocation>();

    the_collection_of_legs = new List<ILeg>();
    the_first_leg = an<ILeg>();
    the_last_leg = an<ILeg>();
    the_collection_of_legs.Add(the_first_leg);
    the_collection_of_legs.Add(the_last_leg);

    the_last_leg
       .Stub(x => x.unload_location())
       .Return(the_final_arrival_unload_location);

    create_sut_using(() => new Itinerary(the_collection_of_legs));
};

static IList<ILeg> the_collection_of_legs;
```

```
static ILeg the_first_leg;
```

Los dos primeros bloques son exactamente iguales a los de la **BDD_Spec** anterior, con la salvedad de que para aclarar nuestra intención:

- Ahora contamos con `the_final_arrival_collection`.
- Antes contábamos con `the_initial_departure_load_location`.

Pero la diferencia es exclusivamente una cuestión de nombrado del campo, ya que podíamos haber escogido haber nombrado a ambos `the_location`, por poner un ejemplo.

3.6.3.8 Política de nombrado

Consideramos que la especialización en el nombrado es algo positivo desde el punto de vista de la legibilidad.

Nada mejor que un ejemplo.

Considérese:

```
.Return(the_initial_departure_load_location);
.Return(the_final_arrival_unload_location);
```

frente a:

```
.Return(the_location);
.Return(the_location);
```

En el primer caso deberíamos tener claro cual de los dos `.Return()` van con qué **BDD_Spec**.

En el segundo caso, esa consideración deja de tener sentido, ya que es imposible saberlo.

3.6.3.9 Fin del Arrange

En cuanto al tercer bloque, es similar en la intención, solo que ahora necesitamos definir comportamiento en el último elemento de la colección y no en el primero.

Por último, el cuarto bloque es el mismo.

3.6.3.10 El código completo de la BDD_Spec

El aspecto final de nuestra **BDD_Spec** sería:

```
public class when_asked_for_the_final_arrival_unload_location
             _with_a_non_empty_leg_collection :
             concern_for_itinerary
{

    Establish context = () =>
    {
        the_final_arrival_unload_location = an<ILocation>();

        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_last_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);
        the_collection_of_legs.Add(the_last_leg);

        the_last_leg
           .Stub(x => x.unload_location())
           .Return(the_final_arrival_unload_location);

        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };


    Because of = () => result = sut.final_arrival_location();

    It should_leverage_the_last_leg_unload_location = () =>
       the_last_leg.received(x => x.unload_location());

    It should_return_the_legs_collection_last_element_unload_location = () =>
       result.ShouldEqual(the_final_arrival_unload_location);

    static ILeg the_last_leg;
    static ILocation result;
    static ILocation the_final_arrival_unload_location;
    static IList<ILeg> the_collection_of_legs;
    static ILeg the_first_leg;
```

```
}
```

3.6.3.11 RED

El siguiente paso es hacer fallar nuestra **BDD_Spec** de forma significativa.

Por lo tanto invertimos el sentido de las aserciones, pasando de:

```
It should_leverage_the_last_leg_unload_location = () =>
   the_last_leg.received(x => x.unload_location());

It should_return_the_legs_collection_last_element_unload_location = () =>
   result.ShouldEqual(the_final_arrival_unload_location);
```

a lo contrario:

```
It should_leverage_the_last_leg_unload_location = () =>
   the_last_leg.never_received(x => x.unload_location());

It should_return_the_legs_collection_last_element_unload_location = () =>
   result.ShouldNotEqual(the_final_arrival_unload_location);
```

Ejecutamos esta versión negada de nuestra **BDD_Spec** buscando esos fallos:

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   when asked for the final arrival unload location with a non empty leg
   collection
   » should leverage the last leg unload location (FAIL)
   » should return the legs collection last element unload location (FAIL)

   Test 'should leverage the last leg unload location' failed:
         Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
   ILeg.unload_location(); would not be called, but it was found on the
   actual calls made on the mocked object.

   (....)

   Test 'should return the legs collection last element unload location'
   failed:
         Machine.Specifications.SpecificationException: Should not equal
   ILocationProxy548705f5709d47debbe67c487a1e463e but does:

   (....)
```

```
0 passed, 2 failed, 0 skipped, took 1,03 seconds (Machine.Specifications
0.3.0).
```

Que afortunadamente obtenemos.

Ahora ya estamos preparados para volver a cambiar nuestra **BDD_Spec** para que haga lo que se supone.

3.6.3.12 GREEN

Ejecutamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for the final arrival unload location with a non empty leg
collection
» should leverage the last leg unload location
» should return the legs collection last element unload location


2 passed, 0 failed, 0 skipped, took 0,80 seconds (Machine.Specifications
0.3.0).
```

Nuestro Happy Day Scenario funciona.

3.6.3.13 La primera refactorización

Tenemos pendiente una reflexión acerca de `lastLeg()`.

Desde el punto de vista de `final_arrival_location()`, `lastLeg()` hace demasiado, y no en el sentido **SRP**, sino en el sentido de redundancia.

Para solucionarlo, hemos decidido tomar una decisión salomónica, que consiste en que desde `final_arrival_location()` vamos a invocar un nuevo método privado llamado `the_last_leg()`:

```
public ILocation final_arrival_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN;
```

```
    return the_last_leg().unload_location();
}
```

Que va a hacer exclusivamente lo que necesitamos, ni una linea más:

```
ILeg the_last_leg()
{
    return underlying_leg_collection[underlying_leg_collection.Count - 1];
}
```

Si nos quedásemos aquí, estaríamos duplicando código ya que `lastLeg()`, también hace eso, además de esa comprobación que resulta redundante en nuestro contexto.

Se nos plantean tres opciones:

- Eliminar `lastLeg()` y sustituirla por `the_last_leg()`.
- Relacionar `lastLeg()` con `the_last_leg()`.
- No hacer nada y dejar las cosas como están.

La primera opción es temeraria, ya que, hay más llamadas a `lastLeg()`, además de la que se produce desde `final_arrival_location()`, y no hemos analizado si esa comprobación hace falta cuando es llamado desde estos otros lugares. Descartado por el momento.

La tercera opción es mejor que la primera, ya que el riesgo de romper algo es nulo (en la primera el riesgo era muy alto). Sin embargo creemos que podemos hacerlo mejor.

La segunda opción es la escogida ya que, sabemos que no vamos a romper nada y además evitamos parcialmente la duplicación de código. Y decimos parcialmente, porque tener dos métodos privados como `the_last_leg()` y `lastLeg()` dice muy poco de nuestro buen juicio como programadores / analistas / diseñadores.

Sin embargo hay que recordar que estamos simulando el escenario **legacy code**, así que, se supone que llegamos a esta decisión porque no nos queda más remedio (nosotros no hemos escrito el código original), y en el fondo, no es más que una decisión parcial, hasta que nos encontremos con los otros escenarios en los que

`lastLeg()` es invocado. Entonces, probablemente, tomemos una decisión más radical.

Por lo tanto el último cambio es éste:

```
ILeg lastLeg()
{
    if (underlying_leg_collection.Count == 0)
       return null;
    return the_last_leg();
}
```

Muy importante, corremos todas las **BDD_Specs** del **DDD_Value_Object** `Itinerary` creadas hasta ahora para asegurarnos de que no hemos roto nada.

Podemos seguir con la **BDD_Spec** complementaria del Happy Day Scenario.

3.6.4 El escenario complementario

3.6.4.1 Otro escenario no necesario

Gracias a la experiencia ganada a través de la **BDD_Spec**:

> *When asked for the initial departure load location with an empty leg collection*

sabemos que, en cuanto intentemos pasarle al constructor una colección de `ILegs` vacía, nos va a responder con una excepción.

De hecho si intentamos ejecutar la **BDD_Spec**:

```
public class when_asked_for_the_final_arrival_unload_location
             _with_an_empty_leg_collection :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_collection_of_legs = new List<ILeg>();
        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };
```

```
    Because of = () => result = sut.final_arrival_location();

    It should_return_a_location_UNKNOWN =
       () => result.ShouldEqual(LocationImplementationExample.UNKNOWN);

    static ILocation result;
    static IList<ILeg> the_collection_of_legs;
}
```

Que es la que correspondería a este escenario:

> *When asked for the final arrival unload location with an empty leg collection*
>
> *It should return a location UNKNOWN.*

Obtenemos el ya conocido:

```
Exception has been thrown by the target of an invocation. --->
System.ArgumentNullException: The list of legs cannot be empty. Parameter
name: legs
```

Por lo tanto, no vamos a gastar nuestro tiempo y energía en algo que ya sabemos, es redundante.

Tomaremos una decisión similar a la que tomamos en la **BDD_Spec**:

> *When asked for the initial departure load location with an empty leg collection*

Limpiar el código de comprobaciones innecesarias.

3.6.4.2 La segunda refactorización

Nuestro método `final_arrival_location()` pasaría de:

```
/// <summary>
///
/// </summary>
/// <returns>The final arrival location.</returns>
```

```
public ILocation final_arrival_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN;
    return the_last_leg().unload_location();
}

ILeg the_last_leg()
{
    return underlying_leg_collection[underlying_leg_collection.Count - 1];
}
```

a esto otro:

```
/// <summary>
///
/// </summary>
/// <returns>The final arrival location.</returns>
public ILocation final_arrival_location()
{
    return the_last_leg().unload_location();
}

ILeg the_last_leg()
{
    return underlying_leg_collection[underlying_leg_collection.Count - 1];
}
```

De nuevo, hemos limpiado el **legacy code** y además:

- Hemos conseguido mayor claridad en el código.
- Hemos evitado ambigüedad, puesto que si alguien lee una comprobación como la que hemos eliminado, automáticamente asume que ese escenario es viable (cuando hemos demostrado que no lo es).
- Hemos ganado conocimiento sobre el sistema. De que otra forma sino hubiésemos podido depurarlo.

Recordar, por último, que debemos volver a correr las **BDD_Specs**, tras un cambio como éste.

3.6.4.3 La tercera refactorización

Creemos haber descubierto una pequeña incongruencia en el nombre de dos de

nuestros métodos:

- `initial_departure_location()`
- `final_arrival_location()`

Y es que, fijémonos en como definimos esa `ILocation`, que devuelven ambos, en nuestras **BDD_Specs**:

- `the_initial_departure_load_location()`
- `the_final_arrival_unload_location()`

Por tanto, parece que podríamos ser más claros al definir la intención de nuestros dos métodos, si usamos:

- `initial_departure_load_location()`
- `final_arrival_unload_location()`

Para proceder con esta refactorización, usaremos la opción **Rename** de **ReSharper**, evitando así, dejarnos ninguna referencia al método antiguo.

Éstas serán las signaturas de los nuevos métodos, en el contexto de su interface:

```
public interface IItinerary : IValueObject<IItinerary>
{
    ILocation initial_departure_load_location();
    ILocation final_arrival_unload_location();
    IDate final_arrival_date();
    IList<ILeg> legs();
    int GetHashCode();
}
```

Parece que ganamos en expresividad, al incluir los conceptos "**load / unload**" que deberían formar parte del **DDD_Ubiquitous_Language**.

Una advertencia final. Es necesario volver a ejecutar las **BDD_Specs**, para comprobar que todo sigue bien.

3.6.4.4 La cuarta refactorización

Antes de pasar al siguiente escenario, debemos comentar una última refactorización.

En nuestro **legacy code** `Itinerary`, ha desaparecido toda referencia a `Location.UNKNOWN`. Así que, ya no necesitamos todo ese código.

Como siempre, debemos ejecutar la suite de BDD_Specs para comprobar que no existe ningún efecto secundario indeseado.

3.7 ItinerarySpecs - When asked for the final arrival date

3.7.1 Análisis previo

El código que pretendemos testar directamente es éste:

```
/// <summary>
///
/// </summary>
/// <returns>Date when cargo arrives at final destination.</returns>
public IDate final_arrival_date()
{
    ILeg lastLeg = this.lastLeg();

    if (lastLeg == null)
       return END_OF_DAYS;
    return lastLeg.unload_time();
}
```

Y decimos directamente porque nos encontramos con otro escenario que invoca `lastLeg()`, con las posibles connotaciones en cuanto a la redundancia de las comprobaciones.

Recordemos que el método `lastLeg()` ya ha sido refactorizado ligeramente en la **BDD_Spec** anterior:

```
ILeg lastLeg()
{
    if (underlying_leg_collection.Count == 0)
       return null;
    return the_last_leg();
}
```

de forma que, a su vez, llama a nuestro:

```
ILeg the_last_leg()
{
    return underlying_leg_collection[underlying_leg_collection.Count - 1];
}
```

Vamos a hacer una serie de consideraciones previas basadas en un análisis del código:

- La primera comprobación que se va a producir, va a ser la de `lastLeg()`, y por las **BDD_Specs** de los dos escenarios anteriores, sabemos que es un escenario imposible, así que, muy probablemente acabemos usando nuestra versión `the_last_leg()`.
- La segunda comprobación se produce en `final_arrival_date()`, y ésta sí que parece tener sentido, ya que comprueba que esa `ILeg` (en concreto, la última) no sea nula.
- En caso de que las dos comprobaciones anteriores pasen, devolvemos lo que le pedimos (candidato a Happy Day Scenario).

3.7.2 Saltan todas las alarmas

La segunda comprobación, debería hacer saltar todas las alarmas, ya que plantea el siguiente escenario como un escenario posible:

> *En la colección de ILegs que usamos para construir el Itinerary, podría darse el caso de que la última ILeg fuese nula.*

Lo verdaderamente grave es que ese escenario sería posible también en la anterior **BDD_Spec**:

> *When asked for the final arrival unload location*

y el **legacy code** no contenía ninguna comprobación para manejar este supuesto.

Por lo tanto, o el **legacy code** estaba muy mal escrito (y peor analizado / diseñado), o bien, hay algo que no estamos entendiendo correctamente.

Vamos a intentar seguir el flujo de la ejecución, para hacernos una idea de como podemos llegar a ese escenario.

3.7.2.1 Primera posibilidad

Puesto que es imposible que la colección de `ILegs` esté vacía (demostrado dos veces), cuando invocamos `lastLeg()` iremos por:

```
return the_last_leg();
```

que a su vez invoca:

```
return underlying_leg_collection[underlying_leg_collection.Count - 1];
```

De forma que, en realidad, lo que estaríamos haciendo sería:

```
ILeg lastLeg = underlying_leg_collection[underlying_leg_collection.Count - 1];
```

En principio, y sin analizar concienzudamente la situación, da la impresión de que eso es incompatible con que la última `ILeg` sea `null`.

3.7.2.2 Segunda posibilidad

El otro escenario posible sería, que al invocar `lastLeg()`, ejecutásemos:

```
if (underlying_leg_collection.Count == 0)
   return null;
```

Que sabemos positivamente que es imposible, ya que una colección de `ILegs` vacía produciría una excepción en la construcción de `Itinerary`. Pero sigamos con el razonamiento, que quizás tengamos algo aquí.

Por lo tanto iríamos a:

```
ILeg lastLeg = null;
```

que sí produciría el escenario en el cual:

```
if (lastLeg == null)
   return END_OF_DAYS;
```

Sin embargo, sabemos que es imposible, ya que antes hubiese saltado un excepción en el constructor.

3.7.2.3 Objetivos

Toda esta discusión, tiene un objetivo doble:

- Analizar el código y lo que representa.
- Concienciarnos del daño que supone tener comprobaciones de escenarios imposibles.

Por el momento vamos a dejar esto aparcado, ya que el Happy Day Scenario está bastante claro, y es el que pretendemos probar antes de nada.

3.7.3 Continuamos con el análisis

Tendríamos, al menos, tres **RAMAS**:

- **RAMA 1**: Si la colección de `ILegs` está vacía.
- **RAMA 2**: Si la colección de `ILegs` no está vacía y el último elemento es nulo.
- **RAMA 3**: Si la colección de `ILegs` no está vacía y el último elemento no es nulo.

La **RAMA 3** representa el Happy Day Scenario.

Ya sabemos por donde empezar.

3.7.4 Happy Day Scenario

3.7.4.1 El legacy code de final_arrival_date()

Con esta **BDD_Spec** vamos a comprobar lo siguiente:

```
public IDate final_arrival_date()
{
   ILeg lastLeg = this.lastLeg();
```

```
   return lastLeg.unload_time();
}
```

3.7.4.2 Assert

Como siempre, empezamos por las aserciones vía bloques IT (Assert):

```
public class when_asked_for_the_final_arrival_date
             _with_a_non_empty_leg_collection_and_a_not_null_last_leg :
             concern_for_itinerary
{
    It should_leverage_the_last_leg_unload_time = () =>
       the_last_leg.received(x => x.unload_time());

    It should_return_the_legs_collection_last_element_unload_date = () =>
       result.ShouldEqual(the_final_arrival_date);

    static IDate result;
    static IDate the_final_arrival_date;
    static ILeg the_last_leg;
}
```

3.7.4.3 Act

Seguimos con el SUT (Act):

```
Because of = () => result = sut.final_arrival_date();
```

3.7.4.4 Arrange

Finalmente procedemos con la primera de las tres AAA (Arrange):

```
Establish context = () =>
{
    the_final_arrival_date = an<IDate>();

    the_collection_of_legs = new List<ILeg>();
    the_first_leg = an<ILeg>();
    the_last_leg = an<ILeg>();
    the_collection_of_legs.Add(the_first_leg);
    the_collection_of_legs.Add(the_last_leg);

    the_last_leg
```

```
        .Stub(x => x.unload_time())
        .Return(the_final_arrival_date);

    create_sut_using(() => new Itinerary(the_collection_of_legs));
};

static IList<ILeg> the_collection_of_legs;
static ILeg the_first_leg;
```

Que, a estas alturas, no debería suponer desafío alguno.

3.7.4.5 Obteniendo la BDD_Spec

Ahora podemos definir formalmente la **BDD_Spec**, como:

> *When asked for the final arrival date with a non empty leg collection*
> *and a not null last leg*
> *It should leverage the last leg unload time.*
> *It should return the legs collection last element unload date.*

3.7.4.6 El código completo de la BDD_Spec

Que en código sería:

```
public class when_asked_for_the_final_arrival_date
             _with_a_non_empty_leg_collection_and_a_not_null_last_leg :
             concern_for_itinerary
{

    Establish context = () =>
    {
        the_final_arrival_date = an<IDate>();

        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_last_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);
        the_collection_of_legs.Add(the_last_leg);

        the_last_leg
           .Stub(x => x.unload_time())
           .Return(the_final_arrival_date);

        create_sut_using(() => new Itinerary(the_collection_of_legs));
```

```
    };

    Because of = () => result = sut.final_arrival_date();

    It should_leverage_the_last_leg_unload_time = () =>
       the_last_leg.received(x => x.unload_time());

    It should_return_the_legs_collection_last_element_unload_date = () =>
       result.ShouldEqual(the_final_arrival_date);

    static IDate result;
    static IList<ILeg> the_collection_of_legs;
    static IDate the_final_arrival_date;
    static ILeg the_first_leg;
    static ILeg the_last_leg;
}
```

3.7.4.7 RED

Invertimos el sentido de las aserciones para obtener el fallo que buscamos (y tratar de minimizar, por tanto, la posibilidad de un falso positivo):

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for the final arrival date with a non empty leg collection and
a not null last leg
» should leverage the last leg unload time (FAIL)
» should return the legs collection last element unload date (FAIL)

Test 'should leverage the last leg unload time' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
ILeg.unload_time(); would not be called, but it was found on the actual
calls made on the mocked object.

(....)

Test 'should return the legs collection last element unload date' failed:
      Machine.Specifications.SpecificationException: Should not equal
IDateProxy05fb7ba705a844a0ba864f5e3d385f0e but does:

(....)


0 passed, 2 failed, 0 skipped, took 0,79 seconds (Machine.Specifications
0.3.0).
```

3.7.4.8 GREEN

Como los fallos son justamente los esperados, volvemos a revertir las aserciones (dejándolas por tanto en su estado original) y ejecutamos la **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for the final arrival date with a non empty leg collection and
a not null last leg
» should leverage the last leg unload time
» should return the legs collection last element unload date


2 passed, 0 failed, 0 skipped, took 0,93 seconds (Machine.Specifications
0.3.0).
```

Ya deberíamos empezar a acostumbrarnos a acertar a la primera.

Vamos con uno de los escenarios complementarios, para ver si podemos añadir algo a la discusión inicial de las comprobaciones redundantes.

3.7.5 El primer escenario complementario

3.7.5.1 Descubriendo si es un escenario posible (AAA)

Vamos a saltarnos todos nuestros principios y vamos a empezar por la primera A de las tres AAA (Arrange).

Usando el bloque `ESTABLISH` anterior como referencia, podríamos describir nuestro escenario de la siguiente forma (especial atención a la linea en cursiva):

```
Establish context = () =>
{
    the_final_arrival_date = an<IDate>();

    the_collection_of_legs = new List<ILeg>();
    the_first_leg = an<ILeg>();
    the_last_leg = null;
    the_collection_of_legs.Add(the_first_leg);
    the_collection_of_legs.Add(the_last_leg);

    create_sut_using(() => new Itinerary(the_collection_of_legs));
```

```
};
```

Si asociamos este escenario con el siguiente Arrange:

```
Because of = () => result = sut.final_arrival_date();
```

Y con este Assert, que lo único que pretende es comprobar que el escenario es siquiera posible (no hace nada, simplemente dice (`true == true`))

```
It should_assert_true = () => true.ShouldBeTrue();
```

Teóricamente deberíamos obtener un:

```
1 passed, o failed, 0 skipped, (....)
```

Que simplemente validaría que el escenario es factible (el último elemento de la colección es nulo). Pero...

3.7.5.2 RED, escenario imposible

...obtenemos esto:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for the final arrival date with a non empty leg collection and
a null last leg
» should assert true (FAIL)

Test 'should assert true' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.Data.NoNullAllowedException: The legs list cannot contain null
elements.
      domain\model\cargo.aggregate\Itinerary.cs(38,0): at
dddsample.domain.model.cargo.aggregate.Itinerary..ctor(IList`1 legs)

(....)

0 passed, 1 failed, 0 skipped, took 0,67 seconds (Machine.Specifications
0.3.0).
```

Que si nos fijamos con más atención:

```
   Exception has been thrown by the target of an invocation. --->
   System.Data.NoNullAllowedException: The legs list cannot contain null
   elements.
   domain\model\cargo.aggregate\Itinerary.cs(38,0):
```

Es decir, el constructor lanza otra excepción ya que:

```
public Itinerary(IList<ILeg> legs)
{
    ILeg null_leg = null;
    if (legs.Contains(null_leg))
       throw new NoNullAllowedException(
                    "The legs list cannot contain null elements.");

(....)
}
```

3.7.5.3 Pavimentando el camino hacia la primera refactorización

Tal y como sospechábamos (y discutimos en el previo), la:

- **RAMA 2**: Si la colección de `ILegs` no está vacía y el último elemento es nulo.

es un escenario imposible.

El constructor no nos deja construir el **DDD_Value_Object** `Itinerary` con una colección de **DDD_Value_Objects** `ILegs` donde alguno de ellos sea nulo. Por lo tanto, el último no puede ser nulo.

De nuevo, estaríamos ante más comprobaciones redundantes, y esta vez, no son inocuas, ya que hicieron saltar todas las alarmas ante un posible problema de mal diseño del **legacy code**.

Así que nos veremos en la obligación de tomar decisiones.

3.7.6 El segundo escenario complementario

3.7.6.1 Escenario imposible

Sabemos que la:

- **RAMA 1**: Si la colección de `ILegs` está vacía.

representa un escenario imposible.

Lo hemos comprobado hasta la saciedad, observando como lanza la excepción el constructor.

Por lo tanto, actuaremos exactamente igual que en el escenario imposible representado por el intento de **BDD_Spec**:

> *When asked for the final arrival unload location with an empty leg collection*

Es decir, vamos a utilizar nuestro `the_last_leg()`.

3.7.6.2 La segunda refactorización

Pasaríamos de:

```
public IDate final_arrival_date()
{
    ILeg lastLeg = this.lastLeg();

    if (lastLeg == null)
       return END_OF_DAYS;
    return lastLeg.unload_time();
}
```

a:

```
public IDate final_arrival_date()
{
```

```
    ILeg lastLeg = this.the_last_leg();

    if (lastLeg == null)
       return END_OF_DAYS;
    return lastLeg.unload_time();
}
```

Pero podemos ir un poco más allá, utilizando nuestra notación **all_lower** para la variable `lastLeg`:

```
public IDate final_arrival_date
{
    ILeg last_leg = this.the_last_leg();

    if (last_leg == null)
       return END_OF_DAYS;
    return last_leg.unload_time();
}
```

Ejecutamos las **BDD_Specs**, para comprobar que no hemos hecho nada malo.

3.7.6.3 La primera refactorización

Gracias al intento de programar la **BDD_Spec**:

> *When asked for the final arrival date with a non empty leg collection and a null last element*

que corresponde a:

- **RAMA 2**: Si la colección de `ILegs` no está vacía y el último elemento es nulo.

sabemos que es un escenario imposible (también provoca una excepción en el constructor pues no se puede construir un `Itinerary` cuando haya algún elemento de la colección de `ILegs` que sea nulo).

Por lo tanto este código no se ejecutará nunca:

```
if (last_leg == null)
   return END_OF_DAYS;
```

Podemos eliminarlo, de forma que nuestro **legacy code** quedaría así:

```
public IDate final_arrival_date()
{
    ILeg last_leg = this.the_last_leg();

    return last_leg.unload_time();
}
```

Pero podemos mejorarlo y hacer que quede en una sola linea:

```
public IDate final_arrival_date()
{
    return the_last_leg().unload_time();
}
```

Ejecutamos las **BDD_Specs** para comprobar que todo sigue bien.

3.7.7 El antes y el después de nuestro legacy code

Comparemos el antes y el después del **legacy code**:

```
ANTES:

public IDate final_arrival_date()
{
    ILeg lastLeg = this.lastLeg();

    if (lastLeg == null)
       return END_OF_DAYS;
    return lastLeg.unload_time();
}


DESPUÉS:


public IDate final_arrival_date()
{
    return the_last_leg().unload_time();
}
```

Creemos que en este caso concreto sobran las palabras.

Hemos testado un nuevo escenario, hemos mejorado muy mucho nuestro conocimiento del sistema y además hemos optimizado el código de forma radical.

Puesto que ya han surgido las dos excepciones que puede lanzar el constructor, parece un buen momento para formalizar esos dos escenarios.

3.8 Excepciones lanzadas por el constructor

3.8.1 Usando Machine.Specifications sin las extensiones DevelopWithPassion

Para testar este comportamiento (el constructor y las excepciones que lanza), vamos a utilizar una aproximación `machine.specifications` **clásica**, en el sentido de **sin uso de extensiones**.

Por un lado, creemos que es positivo aprender a hacer las cosas de varias maneras y por otro, estamos obteniendo unos errores un tanto extraños si optamos por la versión `developwithpassion` (que ya hemos visto al testar el comportamiento de la **DDD_Factory** `RouteSpecificationFactory`).

En principio hemos decidido escribir a **Jean-Paul Boodhoo** acerca del comportamiento tan extraño que estamos observando. Si obtenemos respuesta a tiempo, intentaremos incluirla en este Estudio.

3.8.2 ItinerarySpecs - When trying to construct the itinerary with an empty leg collection

3.8.2.1 El legacy code

El **legacy code** que intentamos someter al **BDD** es el siguiente:

```
public Itinerary(IList<ILeg> legs)
{
    if (legs.Count == 0)
       throw new ArgumentNullException("legs",
                                       "The list of legs cannot be empty.");

(....)
}
```

3.8.2.2 Assert

Empezamos por las aserciones:

```
It should_throw_a_null_argument_exception = () =>
   exception.ShouldBeAn<ArgumentNullException>();

It should_contain_a_custom_error_message = () =>
   exception.ShouldContainErrorMessage("The list of legs cannot be empty.");

static Exception exception;
```

Lo más destacable es que necesitamos un campo de tipo `Exception` sobre el que basar las aserciones:

```
exception.ShouldBeAn<ArgumentNullException>();
exception.ShouldContainErrorMessage("The list of legs cannot be empty.");
```

3.8.2.3 Act

Vamos con el bloque `BECAUSE`:

```
Because of = () =>
   exception = Catch.Exception(() => new Itinerary(new List<ILeg>()));
```

Es interesante destacar como:

- En este caso no aparece mención alguna al SUT, ya que ese es uno de las extras que vienen con las extensiones `developwithpassion`.
- Para pasarle al constructor una lista vacía basta con invocar al constructor de `List<T>`.
- Usamos `Catch.Exception(() => CODIGO_QUE_PROVOCA_LA_EXCEPCION)`
- Y almacenamos el resultado en el campo `exception`, sobre el que realizamos las aserciones.

3.8.2.4 El código completo de la BDD_Spec

Por lo tanto la **BDD_Spec**, quedaría tal que así:

```
public class when_trying_to_construct_the_itinerary
             _with_an_empty_leg_collection
{
```

```
    Because of = () =>
       exception = Catch.Exception(() => new Itinerary(new List<ILeg>()));

    It should_throw_a_null_argument_exception = () =>
       exception.ShouldBeAn<ArgumentNullException>();

    It should_contain_a_custom_error_message = () =>
       exception.ShouldContainErrorMessage(
          "The list of legs cannot be empty.");

    static Exception exception;
}
```

3.8.2.5 concern_for_itinerary no es aplicable

Del código anterior, debería llamarnos la atención el hecho de que no estamos heredando de la **clase base abstracta** `concern_for_itinerary` ya que al no usar las extensiones `developwithpassion`, no nos hace falta.

Es más, al no utilizar esas extensiones, si heredamos, nuestros test fallan.

3.8.2.6 Obteniendo la BDD_Spec

Vamos a definir la **BDD_Spec** formalmente:

```
When trying to construct the itinerary with an empty leg collection
   It should throw a null argument exception.
   It should contain a custom error message.
```

3.8.2.7 RED

Necesitamos hacer que fallen ambas aserciones de forma significativa, para ello cambiamos en la primera aserción esto:

```
exception.ShouldBeAn<ArgumentNullException>();
```

por esto otro (es un ejemplo, la cuestión es poner un tipo de `Exception` distinto al que esperamos):

```
exception.ShouldBeAn<ArgumentOutOfRangeException>();
```

Y en la segunda aserción, cambiamos el mensaje, de esto:

```
exception.ShouldContainErrorMessage("The list of legs cannot be empty.");
```

a esto:

```
exception.ShouldContainErrorMessage("Testing.....");
```

Ahora ejecutamos la **BDD_Spec**, esperando obtener nuestros fallos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when trying to construct the itinerary with an empty leg collection
» should throw a null argument exception (FAIL)
» should contain a custom error message (FAIL)

Test 'should throw a null argument exception' failed:
      Machine.Specifications.SpecificationException: Should be of type
System.ArgumentOutOfRangeException but is of type
System.ArgumentNullException

(....)

Test 'should contain a custom error message' failed:
      Machine.Specifications.SpecificationException: Should contain
"Testing...." but is "The list of legs cannot be empty.
\nParameter name: legs"

(....)

0 passed, 2 failed, 0 skipped, took 0,71 seconds (Machine.Specifications
0.3.0).
```

Que es justo lo que queríamos.

3.8.2.8 GREEN

Si cambiamos las aserciones a sus valores originales, la **BDD_Spec** pasa:

```
------ Test started: Assembly: dddsample.specs.dll ------

when trying to construct the itinerary with an empty leg collection
```

```
  » should throw a null argument exception
  » should contain a custom error message


  2 passed, 0 failed, 0 skipped, took 0,59 seconds (Machine.Specifications
  0.3.0).
```

3.8.3 ItinerarySpecs - When trying to construct the itinerary with a leg collection which contains null legs

3.8.3.1 El legacy code

Ahora vamos a testar el **legacy code** siguiente:

```
public Itinerary(IList<ILeg> legs)
{
    ILeg null_leg = null;
    if (legs.Contains(null_leg))
       throw new NoNullAllowedException("The legs list cannot contain null
elements.");

(....)
}
```

3.8.3.2 El código completo de la BDD_Spec

La mecánica es la misma que en la anterior **BDD_Spec**, solo que ahora debemos construir en el `ESTABLISH`, la colección de `ILegs`.

Vamos directamente con el código de la **BDD_Spec**, ya que no contiene nada que no hayamos visto hasta ahora:

```
public class when_trying_to_construct_the_itinerary
             _with_a_leg_collection_which_contains_null_legs
{
    Establish context = () =>
    {
        the_collection_of_legs = new List<ILeg>();
        the_first_leg = null;
        the_last_leg = null;
        the_collection_of_legs.Add(the_first_leg);
        the_collection_of_legs.Add(the_last_leg);
    };
```

```
    Because of = () =>
       exception = Catch
                      .Exception(() => new Itinerary(the_collection_of_legs));

    It should_throw_a_null_argument_exception = () =>
       exception.ShouldBeAn<NoNullAllowedException>();

    It should_contain_a_custom_error_message = () =>
       exception.ShouldContainErrorMessage(
                    "The legs list cannot contain null elements.");

    static IList<ILeg> the_collection_of_legs;
    static ILeg the_first_leg;
    static ILeg the_last_leg;
    static Exception exception;
}
```

3.8.3.3 Obteniendo la BDD_Spec

Formalmente, podemos definir la **BDD_Spec** así:

> *When trying to construct the itinerary with a leg collection which contains null legs*
>
> *It should throw a null argument exception.*
>
> *It should contain a custom error message.*

3.8.3.4 RED-GREEN

El proceso es el mismo que hemos estado utilizando para `Itinerary`:

- `RED`: Fallo significativo.
- `GREEN`: Las aserciones se cumplen.

Y el resultado final también:

```
------ Test started: Assembly: dddsample.specs.dll ------

when trying to construct the itinerary with a leg collection which
contains null legs
» should throw a null argument exception
» should contain a custom error message
```

```
2 passed, 0 failed, 0 skipped, took 0,73 seconds (Machine.Specifications
0.3.0).
```

3.8.4 Refactorizaciones

Ahora que por fin tenemos los tests funcionando, podemos hacer un par de refactorizaciones cosméticas al constructor.

De:

```
public Itinerary(IList<ILeg> legs)
{
    if (legs.Count == 0)
       throw new ArgumentNullException("legs",
                                       "The list of legs cannot be empty.");

    ILeg null_leg = null;
    if (legs.Contains(null_leg))
       throw new NoNullAllowedException(
                    "The legs list cannot contain null elements.");

    this.underlying_leg_collection = legs;
}
```

a:

```
public Itinerary(IList<ILeg> the_leg_collection)
{
    if (the_associated_leg_collection.Count == 0)
       throw new ArgumentNullException(
                    "the_leg_collection",
                    "The injected leg collection cannot be empty.");

    ILeg a_null_leg = null;
    if (the_associated_leg_collection.Contains(a_null_leg))
       throw new NoNullAllowedException(
              "The injected leg collection cannot contain null Leg elements.");

     this.underlying_leg_collection = the_associated_leg_collection;
}
```

Puesto que, entre otras cosas, esto:

```
the_associated_leg_collection.Contains(a_null_leg)
```

se lee mucho mejor. Y además, explica mucho mejor su intención que esto otro:

```
legs.Contains(null_leg)
```

De hecho, en nuestra opinión, con estos cambios, los comentarios sobran, ya que no aportan información extra:

```
/// <summary>
/// Constructor.
/// </summary>
/// <param name="the_associated_leg_collection">List of legs for this
itinerary</param>
```

3.8.5 Arreglando los platos rotos

Deberíamos ejecutar la suite de **BDD_Specs** en este momento, para asegurarnos de que todo sigue en su sitio.

De hecho, si lo hacemos, veremos que fallan los relativos a los mensajes que lanzan las excepciones, ya que los acabamos de cambiar en el constructor pero no en los tests:

```
(....)

Test 'should contain a custom error message' failed:
      Machine.Specifications.SpecificationException: Should contain "The
list of legs cannot be empty." but is "The injected leg collection cannot
be empty.
\nParameter name: the_associated_leg_collection"

(....)

Test 'should contain a custom error message' failed:
      Machine.Specifications.SpecificationException: Should contain "The
legs list cannot contain null elements." but is "The injected leg
collection cannot contain null Leg elements."

(...)

123 passed, 2 failed, 0 skipped, took 2,02 seconds (Machine.Specifications
0.3.0).
```

Pero tiene fácil solución.

Esto para el primer fallo:

```
It should_contain_a_custom_error_message = () =>
   exception.ShouldContainErrorMessage(
                "The injected leg collection cannot be empty.");
```

Esto para el segundo:

```
It should_contain_a_custom_error_message = () =>
   exception.ShouldContainErrorMessage(
              "The injected leg collection cannot contain null Leg elements.");
```

Y todo vuelve a funcionar:

```
  125 passed, 0 failed, 0 skipped, took 1,98 seconds (Machine.Specifications
  0.3.0).
```

3.8.6 Imposibilidad de aplicar una DDD_Factory

Por último habría que destacar la imposibilidad de usar una **DDD_Factory** como solución para empaquetar las excepciones (esa fue la solución escogida para construir objetos **DDD_Value_Object** `RouteSpecification`).

La cuestión es que al ser **legacy code**, podría haber código que todavía usase el constructor directamente.

Si se diese ese caso, estaríamos ante un problema enorme, ya que este código antiguo, al crear un **DDD_Value_Object** `Itinerary`, no comprobaría sus invariantes.

Por lo tanto queda descartada la utilización de una **DDD_Factory**, pese a ser una solución mucho más elegante y que además cumpliría el **SRP**.

3.9 Limpiando las BDD_Specs de los escenarios imposibles (Refactor)

Antes de continuar, deberíamos hacer una pequeña limpieza en los nombres de las **BDD_Specs**, el `Context` si usamos la nomenclatura `Context / Specification`, ya que en estos momentos tenemos lo siguiente:

```
when_asked_for_the_final_arrival_date
_with_a_non_empty_leg_collection_and_a_null_last_leg

when_asked_for_the_final_arrival_unload_location
_with_a_non_empty_leg_collection

when_asked_for_the_initial_departure_load_location
_with_a_non_empty_leg_collection
```

Pero para ninguno de esos tres `Contexts` existe una **BDD_Spec** complementaria, ya que al construir las **BDD_Specs** complementarias descubrimos que no hacían falta, pues había comprobaciones de más en los métodos respectivos (escenarios imposibles).

Por otro lado, podríamos considerar que esos escenarios complementarios tienen sus propias **BDD_Specs** escritas. Serían las BDD_Specs que se encargan de cubrir las excepciones que lanza el constructor:

```
When trying to construct the itinerary with an empty leg collection
   It should throw a null argument exception.
   It should contain a custom error message.
```

```
When trying to construct the itinerary with a leg collection which
contains null legs
   It should throw a null argument exception.
   It should contain a custom error message.
```

Así que, nos parece que tiene mucho más sentido usar respectivamente los siguientes `Contexts`:

```
when_asked_for_the_final_arrival_date
when_asked_for_the_final_arrival_unload_location
when_asked_for_the_initial_departure_load_location
```

Por lo tanto, renombramos y ejecutamos la suite de **BDD_Specs** para comprobar que esta refactorización no ha roto nada.

3.10 Sobre la igualdad “DDD Style”

3.10.1 Análisis previo

Empezamos por el **legacy code** que queremos probar:

```
/// <summary>
///
/// </summary>
/// <param name="the_other_itinerary">Itinerary to compare</param>
/// <returns>true if the legs in this and the other itinerary are all
equal.</returns>
public bool has_the_same_value_as(IItinerary the_other_itinerary)
{
    return the_other_itinerary != null &&
           underlying_leg_collection.Equals(the_other_itinerary.legs());
}
```

Por una vez los comentarios resultan interesantes:

```
<returns>true if the legs in this and the other itinerary are all
equal.</returns>
```

A saber, para que dos **DDD_Value_Object** `IItineraries` tengan el mismo valor, necesariamente, la colección interna de **DDD_Value_Objects** `ILegs` debe tener el mismo valor.

Es un caso claro de delegación, pues es la propia igualdad entre las colecciones de **DDD_Value_Objects** `ILegs` la que determina si las dos **DDD_Value_Object** `IItineraries` tienen el mismo valor.

A mayores, y para evitar comportamiento indeseado, si el itinerario que le inyectamos es `null`, evidentemente su colección de `ILegs` es `null` también, así que no pueden tener el mismo valor.

Por lo tanto, parece que estamos ante tres **RAMAS**:

- **RAMA 1**: El `Itinerary` inyectado (`the_other_itinerary`) es `null`.
- **RAMA 2**: El `Itinerary` inyectado cumple la igualdad con respecto a la colección

de `ILegs`.

- **RAMA 3**: El `Itinerary` inyectado no cumple la igualdad con respecto a la colección de `ILegs`.

La **RAMA 2** es el Happy Day Scenario.

Ya sabemos lo que tenemos que hacer.

3.10.2 El Happy Day Scenario

3.10.2.1 El legacy code de has_the_same_value_as()

Pretendemos testar esto:

```
public bool has_the_same_value_as(IItinerary the_other_itinerary)
{
    return underlying_leg_collection.Equals(the_other_itinerary.legs());
}
```

Y de ese código, únicamente cuando el valor devuelto es `true`.

Por lo tanto estamos en la:

- **RAMA 2**: El `Itinerary` inyectado cumple la igualdad con respecto a la colección de `ILegs`.

Parece relativamente sencillo crear una **BDD_Spec**, ya que con dos aserciones bastarían para comprobar que:

- El método `Equals()` es invocado en la colección de `ILegs`.
- El resultado es `true`.

3.10.2.2 Imposibilidad de mockar Equals()

El problema viene a la hora de realizar la aserción Mockista, ya que, pese a lo que podríamos pensar, no es tan fácil mockar la llamada a `Equals()`.

Si nos fijamos, hasta el momento, en todas las **BDD_Specs** de `Itinerary`, hemos creado:

```
the_collection_of_legs = new List<ILeg>();
```

Es decir, la colección de `ILegs`, es real, no está mockada.

Podríamos intentar algo como:

```
the_collection_of_legs = the_dependency<IList<ILeg>>();
```

y a continuación establecer el comportamiento esperado:

```
the_collection_of_legs
   .Stub(x => x.Equals(the_itinerary_with_same_leg_collection.legs()))
   .Return(true);
```

Pero aun así no conseguiríamos lo que buscamos ya que `IList<T>` expone la siguiente funcionalidad propia:

- `IndexOf(T item)`
- `Insert(int index, T item)`
- `RemoveAt(int index)`
- `this[int index] {get; set;}`

Y hereda funcionalidad de:

- `IEnumerable`:
  - `GetEnumerator()`
- `IEnumerable<T>`:
  - `GetEnumerator<T>`
- `ICollection<T>`:
  - `Add(T item)`
  - `Clear()`
  - `Contains(T item)`

- `CopyTo(T[] array, int arrayIndex)`
- `Remove(T item)`
- `Count {get;}`
- `IsReadOnly {get;}`

Ni rastro del `Equals()` (que viene heredado vía la **clase base** `Object`).

Así que un mock, no nos solucionaría nada, ya que no podemos mockar la llamada a `Equals()`.

No nos vamos a extender más en esto y vamos a seguir un paradigma inspirado en el State-Based Testing TDD para comprobar la segunda aserción.

3.10.2.3 AAA

El bloque `IT` (Assert) es trivial, basta con comprobar que el resultado sea `true`:

```
It should_confirm_they_have_the_same_value = () => result.ShouldBeTrue();

static bool result;
```

El bloque `BECAUSE` (Act) tampoco reviste mayor dificultad, simplemente invocamos el método adecuado en el SUT:

```
Because of = () =>
   result = sut.has_the_same_value_as(the_itinerary_with_same_leg_collection);

It should_confirm_they_have_the_same_value = () => result.ShouldBeTrue();

static bool result;
static IItinerary the_itinerary_with_same_leg_collection;
```

El bloque `ESTABLISH` (Arrange) únicamente debe establecer las condiciones en las que se va a desarrollar la **BDD_Spec**.

En este caso, necesita crear dos **DDD_Value_Objects** `Itineraries` que compartan la misma colección de `ILegs`:

```
Establish context = () =>
{
    the_collection_of_legs = new List<ILeg>();
    the_first_leg = an<ILeg>();
    the_last_leg = an<ILeg>();
    the_collection_of_legs.Add(the_first_leg);
    the_collection_of_legs.Add(the_last_leg);

    the_itinerary_with_same_leg_collection =
       new Itinerary(the_collection_of_legs);
    create_sut_using(() => new Itinerary(the_collection_of_legs));
};
```

Es decir:

- La colección de `ILegs` es real.
- Los `ILegs` que conforman la colección son Stubs (no vamos a establecer condiciones sobre ellos, solo queremos que **existan**).
- Construimos ambos `Itineraries` con la misma colección de `ILegs`.

Ya hemos manejado contextos mucho más complejos.

3.10.2.4 Obteniendo la BDD_Spec

Por lo tanto la **BDD_Spec** sería:

```
When asked if two itineraries with the same leg collection have the
same value
   It should confirm they have the same value.
```

3.10.2.5 El código completo de la BDD_Spec

Y el código completo (esta vez más necesario que nunca ya que hemos omitido toda la parte del "código rojo", que ya deberíamos tener clara a estas alturas):

```
public class when_asked_if_two_itineraries_with_the_same_leg_collection
             _have_the_same_value :
             concern_for_itinerary
{
    Establish context = () =>
```

```
    {
        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_last_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);
        the_collection_of_legs.Add(the_last_leg);

        the_itinerary_with_same_leg_collection =
           new Itinerary(the_collection_of_legs);
        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };

    Because of = () =>
       result = sut.has_the_same_value_as(
                       the_itinerary_with_same_leg_collection);

    It should_confirm_they_have_the_same_value = () => result.ShouldBeTrue();

    static bool result;
    static IList<ILeg> the_collection_of_legs;
    static IItinerary the_itinerary_with_same_leg_collection;
    static ILeg the_first_leg;
    static ILeg the_last_leg;
}
```

Destacar que volvemos a heredar de la **clase base abstracta** `concern_for_itinerary` de forma que volvemos a tener el SUT a nuestra disposición.

3.10.2.6 RED

Tenemos que hacer que la aserción falle, primeramente.

Para ello basta con invertir el sentido de la misma:

```
result.ShouldBeFalse();
```

Ejecutamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two itineraries with the same leg collection have the same
value
» should confirm they have the same value (FAIL)

Test 'should confirm they have the same value' failed:
```

```
      Machine.Specifications.SpecificationException: Should be [false] but
is [true]

(....)

0 passed, 1 failed, 0 skipped, took 1,42 seconds (Machine.Specifications
0.3.0).
```

Empezamos con buen pie esta **BDD_Spec**.

Más claro imposible:

```
Should be [false] but is [true]
```

3.10.2.7 GREEN

Volvemos a invertir el sentido de nuestra aserción a su valor original y:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two itineraries with the same leg collection have the same
value
» should confirm they have the same value


1 passed, 0 failed, 0 skipped, took 0,74 seconds (Machine.Specifications
0.3.0).
```

Ya lo tenemos.

Vamos con la siguiente **RAMA**.

3.10.3 El primer escenario complementario

3.10.3.1 El legacy code de has_the_same_value_as()

Correspondería a la:

- **RAMA 3**: El `Itinerary` inyectado no cumple la igualdad con respecto a la colección de `ILegs`.

Probaríamos el mismo código que en la **BDD_Spec** anterior, pero buscando que el valor devuelto sea `false`:

```
public bool has_the_same_value_as(IItinerary the_other_itinerary)
{
    return underlying_leg_collection.Equals(the_other_itinerary.legs());
}
```

El planteamiento es el mismo que en la **BDD_Spec** del Happy Day Scenario.

3.10.3.2 Obteniendo la BDD_Spec

La **BDD_Spec**:

> *When asked if two itineraries with different leg collection have the same value*
>
> *It should confirm they have different value.*

3.10.3.3 El código de la BDD_Spec

Y su código:

```
public class when_asked_if_two_itineraries
             _with_different_leg_collection_have_the_same_value :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_last_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);
        the_collection_of_legs.Add(the_last_leg);

        create_sut_using(() => new Itinerary(the_collection_of_legs));

        the_alternative_collection_of_legs = new List<ILeg>();
        the_alternative_first_leg = an<ILeg>();
        the_alternative_last_leg = an<ILeg>();
        the_alternative_collection_of_legs.Add(the_alternative_first_leg);
        the_alternative_collection_of_legs.Add(the_alternative_last_leg);
```

```
        the_itinerary_with_different_leg_collection =
           new Itinerary(the_alternative_collection_of_legs);
    };

    Because of = () =>
       result = sut.has_the_same_value_as(
                      the_itinerary_with_different_leg_collection);

    It should_confirm_they_have_different_value = () => result.ShouldBeFalse();

    static bool result;
    static IList<ILeg> the_collection_of_legs;
    static ILeg the_first_leg;
    static ILeg the_last_leg;
    static IItinerary the_itinerary_with_different_leg_collection;
    static IList<ILeg> the_alternative_collection_of_legs;
    static ILeg the_alternative_first_leg;
    static ILeg the_alternative_last_leg;
}
```

Diferencias con respecto al Happy Day Scenario:

- En el bloque `ESTABLISH` creamos **dos** listas de `ILegs` y se las inyectamos a los `Itineraries`.
- En el bloque `IT` esperamos que se confirme que son diferentes.

Y ahí se acaban las diferencias.

3.10.3.4 RED-GREEN

Si procedemos con nuestra rutina habitual:

- Invertir el sentido de las aserciones para obtener el fallo significativo.
- Con las aserciones en su sentido original pasamos el test.

Nos encontraremos con:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if two itineraries with different leg collection have the same
value
» should confirm they have different value
```

```
1 passed, 0 failed, 0 skipped, took 0,78 seconds (Machine.Specifications
0.3.0).
```

Vamos con la última **RAMA**.

3.10.4 El segundo escenario complementario

3.10.4.1 El legacy code de has_the_same_value_as()

En esta última **BDD_Spec**, vamos a la:

- **RAMA 1**: El `Itinerary` inyectado (`the_other_itinerary`) es `null`.

Ésta es la parte del código que vamos a testar:

```
public bool has_the_same_value_as(IItinerary the_other_itinerary)
{
    return the_other_itinerary != null;
}
```

3.10.4.2 La BDD_Spec

La **BDD_Spec** es muy similar a las dos anteriores.

En código:

```
public class when_asked_if_a_null_itinerary
             _has_the_same_value_as_any_other_itinerary :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_last_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);
        the_collection_of_legs.Add(the_last_leg);

        a_null_itinerary = null;
        create_sut_using(() => new Itinerary(the_collection_of_legs));
```

```
    };

    Because of = () => result = sut.has_the_same_value_as(a_null_itinerary);

    It should_confirm_they_have_different_value = () => result.ShouldBeFalse();

    static bool result;
    static IList<ILeg> the_collection_of_legs;
    static IItinerary a_null_itinerary;
    static ILeg the_first_leg;
    static ILeg the_last_leg;
}
```

Y definida formalmente:

> *When asked if a null itinerary has the same value as any other itinerary*
>
> *It should confirm they have different value.*

3.10.4.3 RED-GREEN

Si seguimos los pasos ya de sobra conocidos, llegaríamos a:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if a null itinerary has the same value as any other itinerary
» should confirm they have different value


1 passed, 0 failed, 0 skipped, took 0,58 seconds (Machine.Specifications
0.3.0).
```

3.10.5 Consideraciones finales

Esta serie de **BDD_Specs** son muy interesantes ya que el entorno nos obliga a cambiar nuestra habitual perspectiva basada en el Interaction-Based Testing TDD por una basada en el State-Based Testing TDD y comprobamos como las herramientas **BDD** utilizadas, salvan el escollo sin problemas.

3.11 ItinerarySpecs - When asked for the itinerary hash code

Con esta **BDD_Spec** ocurre algo parecido a lo que ocurría en las **BDD_Specs sobre la igualdad** anteriores.

3.11.1 El legacy code GetHashCode()

Consideremos el código que queremos testar:

```
public override int GetHashCode()
{
    return underlying_leg_collection.GetHashCode();
}
```

Lo más sencillo sería, simplemente, establecer un mock para la colección de `ILegs` que supiese si se produce una llamada a `GetHashCode()`. Pero el caso de `GetHashCode()` es el mismo que el de `Equals()` en las **BDD_Specs** anteriores, no pertenece a `IList<T>`, pertenece a `Object` y por tanto no es mockable.

Sin repetirnos en el razonamiento, ya que todo lo que discutimos para las **BDD_Specs sobre la igualdad** es aplicable aquí, vamos a tomar esta vez, dos caminos distintos:

- **Directo**. Comprobamos que el valor devuelto es el que buscábamos (State-Based Testing TDD).
- **Indirecto**. Como no podemos mockar lo que queremos, vamos a mostrar una forma indirecta de testar estos casos, si bien no es la que más sentido tiene aquí. Comprobar que el método devuelve el tipo adecuado.

3.11.2 Assert

Vamos a empezar por la tercera A (Assert):

```
It should_calculate_the_hash_code = () =>
   result.ShouldEqual(the_collection_of_legs_hash_code);

It should_return_a_hash_code = () => result.ShouldBeOfType(typeof(int));
```

```
static int result;
static int the_collection_of_legs_hash_code;
```

Consideraciones:

- La primera aserción es la que identificamos con el State-Based Testing TDD y es autoexplicativa. Simplemente comprueba que el resultado sea el esperado.
- La segunda aserción es la que denominábamos como **camino indirecto**. Si nos fijamos, lo único que estamos asertando es que el resultado va ser de tipo `int`. Esto será muy común a la hora de practicar **BDD** para guiar el desarrollo de los **Controllers** (**MVC**).

3.11.3 Act

La segunda A (Act) es trivial, así que no nos vamos a parar en ella:

```
Because of = () => result = sut.GetHashCode();

It should_calculate_the_hash_code = () =>
    result.ShouldEqual(the_collection_of_legs_hash_code);

It should_return_a_hash_code = () => result.ShouldBeOfType(typeof(int));

static int result;
static int the_collection_of_legs_hash_code;
```

3.11.4 Arrange

La primera A (Arrange) tampoco tiene mayor complicación.

La única novedad nos la encontramos en que almacenamos el valor real del **hash code** de la colección de `ILegs`, para poder usarlo en la primera aserción:

```
Establish context = () =>
{
    the_collection_of_legs = new List<ILeg>();
    the_first_leg = an<ILeg>();
    the_last_leg = an<ILeg>();
    the_collection_of_legs.Add(the_first_leg);
    the_collection_of_legs.Add(the_last_leg);
```

```
    the_collection_of_legs_hash_code = the_collection_of_legs.GetHashCode();

    create_sut_using(() => new Itinerary(the_collection_of_legs));
}
```

3.11.5 Obteniendo la BDD_Spec

Por lo tanto la **BDD_Spec** sería ésta:

```
When asked for the itinerary hash code
   It should return a hash code.
   It should calculate the hash code.
```

3.11.6 RED-GREEN

Siguiendo los pasos habituales (RED-GREEN), las aserciones pasan sin mayor problema:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked for the itinerary hash code
» should return a hash code
» should calculate the hash code


2 passed, 0 failed, 0 skipped, took 1,48 seconds (Machine.Specifications
0.3.0).
```

3.12 ItinerarySpecs - Primera recapitulación

A estas alturas ya tenemos probada toda la funcionalidad del **legacy** `Itinerary` a excepción de `isExpected(IHandlingEvent HandlingEvent)`, que tiene toda la pinta de suponer un verdadero desafío.

Hemos luchado contra la adversidad:

- Dificultad para testar excepciones en el constructor.
- Imposibilidad de mockar las llamadas a `Equals()` y `GetHashCode()`.

Y nos hemos adaptado:

- Usando `machine.specifications` sin extensiones para testar las excepciones.
- Cambiando el paradigma por uno basado en el State-Based Testing TDD para salvar la imposibilidad de mockar según que llamadas.

Al mismo tiempo, hemos adaptado la Práctica de Diseño Agile **BDD**, para poder usarla en casos donde nos veamos en la obligación de testar **legacy code** que no fue precisamente concebido con el **BDD** en mente.

Gracias a esa flexibilidad hemos conseguido mejorar mucho la funcionalidad del **DDD_Value_Object** `Itinerary`, ya que en varios casos hemos logrado simplificar (`REFACTOR`) el código, eliminando comprobaciones innecesarias, que en el mejor de los casos representaba ruido a la hora de entender la intención original (`initial_departure_load_location()` por ejemplo) y en el peor de los casos un serio problema al plantear la existencia de escenarios extremos que podían comprometer la estabilidad del sistema, cuando en realidad eran también escenarios imposibles (Nos viene a la mente el lío que había en `final_arrival_date()`).

También hemos cambiado los nombres de algunos métodos buscando una mayor claridad y mejor definición de intenciones.

Por todo esto parece un buen momento para empaquetar y a **github**, antes de

enfrentarnos a `isExpected(IHandlingEvent HandlingEvent)`.

3.13 ItinerarySpecs - IsExpected()

3.13.1 El legacy code de IsExpected()

Éste es el **legacy code**:

```
/// <summary>
/// Tests if the given handling event is expected when executing this
/// itinerary.
/// </summary>
/// <param name="handling_event">Event to test.</param>
/// <returns><code>true</code> if the event is expected.</returns>
public bool isExpected(IHandlingEvent handling_event)
{
    if (underlying_leg_collection.Count == 0)
    {
        return true;
    }

    if (the_handling_event.type() == HandlingEventType.RECEIVE)
    {
        //Check that the first leg's origin is the event's location
        var leg = underlying_leg_collection[0];
        return leg.load_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    if (the_handling_event.type() == HandlingEventType.LOAD)
    {
        //Check that there is one leg with same load location and voyage
        foreach (var leg in underlying_leg_collection)
        {
            if (leg.load_location()
                   .has_the_same_identity_as(the_handling_event.location()) &&
                leg.voyage()
                   .has_the_same_identity_as(the_handling_event.voyage()))
               return true;
        }
        return false;
    }

    if (the_handling_event.type() == HandlingEventType.UNLOAD)
    {
        //Check that the there is one leg with same unload location and voyage
        foreach (var leg in underlying_leg_collection)
        {
            if(leg.unload_location()
                  .has_the_same_identity_as(the_handling_event.location()) &&
```

```
                leg.voyage()
                   .has_the_same_identity_as(the_handling_event.voyage()))
               return true;
        }
        return false;
    }

    if (the_handling_event.type() == HandlingEventType.CLAIM)
    {
        //Check that the last leg's destination is from the event's location
        var last_leg = lastLeg();
        return last_leg
                  .unload_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    //HandlingEventType.CUSTOMS;
    return true;
}
```

Este tipo de código es muy común y por tanto las posibilidades de encontrarnos algo del estilo son muy altas.

Un primer análisis, muy por encima, nos llevaría a concluir que viola el **SRP** y el **OCP**:

- Viola el **SRP** porque hace demasiadas cosas.
- Viola el **OCP**, porque cada vez que queramos añadir un nuevo `HandlingEventType` vamos a encontrarnos con la situación de que `isExpected()` está abierta a la modificación y cerrada a la extensión. Justo lo opuesto a si cumpliese el **OCP**.

3.13.2 Los comentarios en el código y la violación del SRP

Es bastante curioso observar como la mayor parte de las veces que se viola el **SRP**, hay comentarios de por medio. Vamos a elaborar un poco más este razonamiento:

Si intentásemos definir en lenguaje natural qué es lo que hace este método, probablemente llegaríamos a algo parecido a la siguiente descripción:

> *"Comprueba cual es el tipo de evento recibido y evalúa si era esperado."*

Ese "**y**" es el que marca la división necesaria para cumplir el **SRP**.

`IsExpected()` debería hacer únicamente esto:

> *"Comprueba cual es el tipo de evento recibido."*

Y si nos ponemos muy puntillosos:

> *"Comprueba cual es el tipo de evento recibido y delega."*

Fijémonos en la situación de los comentarios en el código (están sacados tal cual del código java):

```
if (the_handling_event.type() == HandlingEventType.RECEIVE)
    //Check that the first leg's origin is the event's location

if (the_handling_event.type() == HandlingEventType.LOAD)
    //Check that there is one leg with same load location and voyage

if (the_handling_event.type() == HandlingEventType.UNLOAD)
    //Check that the there is one leg with same unload location and voyage

if (the_handling_event.type() == HandlingEventType.CLAIM)
    //Check that the last leg's destination is from the event's location
```

Los propios comentarios nos indican que estamos haciendo demasiadas cosas (violando el **SRP**, por tanto).

Esos comentarios están ahí porque el código no es lo suficientemente expresivo para dejar clara cual es su función.

La mayor parte de las veces que nos encontremos con esto, podemos mejorar nuestro código atendiendo a algunos patrones clásico del **GoF** como por ejemplo **GoF_Strategy**.

Sea como fuere, cada uno de esos comentarios deberían convertirse en un método, o incluso en una clase con un solo método (si podemos identificarlos por ejemplo con un **DDD_Service**), de forma que en ambos casos mejoramos el **OCP**, ya que pese a violarlo

al tener que crear un `if` más, por lo menos, empezaríamos a estar abiertos a la extensión (por ejemplo, una nueva **GoF_Strategy**).

3.13.3 Análisis previo

Una vez hecha esta pequeña reflexión, necesitamos establecer las **RAMAS** para al menos saber cuantas **BDD_Specs** vamos a necesitar.

Vamos por orden según va el código:

- **RAMA 1**: La colección de `ILegs` es un colección vacía.
- **RAMA 2**: Hemos recibido un **DDD_Domain_Event** de tipo `RECEIVED`.
- **RAMA 3**: Hemos recibido un **DDD_Domain_Event** de tipo `LOAD`.
- **RAMA 4**: Hemos recibido un **DDD_Domain_Event** de tipo `UNLOAD`.
- **RAMA 5**: Hemos recibido un **DDD_Domain_Event** de tipo `CLAIM`.
- **RAMA 6**: Hemos recibido un **DDD_Domain_Event** de tipo `CUSTOMS`.

3.13.4 De IsExpected() a was_expecting()

Antes de empezar a implementar las **BDD_Specs** de cada **RAMA**, vamos a cambiar el nombre del método.

Somos conscientes de que ese cambio sería prácticamente inviable en un entorno real **legacy code**, ya que podrían quedar llamadas externas al método con el nombre antiguo.

Sin embargo, una vez hecha la advertencia, debemos notar que no es la primera vez que lo hacemos, y al igual que en las anteriores, la motivación radica en que está bastante mal nombrado y cuando aparezca en las **BDD_Specs** va a sonar bastante antinatural.

Claramente `was_expecting(IHandlingEvent)`, se lee mejor y expresa mejor el significado.

Por lo tanto, vamos a suponer que el método originalmente se llamaba así:

```
public bool was_expecting(IHandlingEvent the_handling_event) { (....) }
```

Desde el punto de vista de un cliente:

```
the_itinerary.was_expecting(a_receive_type_handling_event);
```

Éste último debería mostrar la claridad de la que hablábamos (sin ningún tipo de comentarios queda completamente claro qué es lo que hace).

Ahora sí que vamos con las **BDD_Specs**.

3.13.5 BDD_Spec - was_expecting() - RAMA 1

3.13.5.1 El legacy code

Éste es el **legacy code** que corresponde a la **RAMA 1**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    if (underlying_leg_collection.Count == 0)
    {
        return true;
    }
}
```

3.13.5.2 Escenario imposible

Todo parece normal y correcto, si no fuese por el hecho de que estamos hartos de ver como este escenario es simplemente imposible.

Si se intentase crear un **DDD_Value_Object** `Itinerary` inyectándole una colección de `ILegs` vacía, ya sabemos que el constructor lanzaría una excepción.

Así que, podemos proceder con la eliminación de este código.

La **RAMA 1** representa un escenario imposible.

3.13.6 BDD_Spec - was_expecting() - RAMA 2

3.13.6.1 El legacy code

Éste es el **legacy code** que corresponde a la **RAMA 2**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.RECEIVE)
    {
        //Check that the first leg's origin is the event's location
        var leg = underlying_leg_collection[0];
        return leg.load_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    (....)
}
```

3.13.6.2 Primera refactorización

Antes de empezar a hacer nada, vamos a acometer nuestro primer cambio.

```
var leg = underlying_leg_collection[0];
```

Se parece sospechosamente a un método privado definido por nosotros:

```
ILeg the_initial_leg()
{
    return underlying_leg_collection[0];
}
```

Hacen exactamente lo mismo así que, aunque no se deben hacer cambios sin tener una suite de tests que nos aseguren que no rompemos nada, vamos a ser un poco más flexibles con esta norma y aplicaremos el sentido común.

Por lo tanto:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.RECEIVE)
    {
        //Check that the first leg's origin is the event's location
        var leg = the_initial_leg();
        return leg.load_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    (....)
}
```

Y ya que hemos tomado este camino, no ganamos en expresividad ni en funcionalidad teniendo dos lineas dentro del `if`.

Por lo tanto podemos proceder a compactar:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.RECEIVE)
    {
        //Check that the first leg's origin is the event's location
        return the_initial_leg()
                  .load_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    (....)
}
```

Mucho más expresivo.

Sin leer el comentario queda claro que cargamos la `ILocation` de la `ILeg` inicial y la comparamos con la `ILocation` asociada al evento recibido.

3.13.6.3 Análisis de testabilidad

Vamos con la **BDD_Spec**.

La idea es mantener nuestro paradigma de testar las interacciones entre el SUT y sus colaboradores / dependencias. Por lo tanto vamos a hacer un pequeño análisis previo de qué podemos y qué no podemos testar:

1. La llamada al método `type()` de `the_handling_event`. Parece posible.
2. La llamada a `the_initial_leg()`. Parece imposible de forma directa ya que es un método privado, pero sabemos que ese método privado lo único que hace es devolver el primer `ILeg` de la colección. Y por aquí si que podríamos atacarlo.
3. La llamada a `load_location()` de `the_initial_leg()`. Parece posible si atendemos a lo razonado en el punto anterior.
4. La llamada a `has_the_same_identity_as()` de la `ILocation`. Parece posible.
5. La llamada a `location()` de `the_handling_event`. Parece posible.

3.13.6.4 Assert

Vamos con las aserciones vía bloques `IT`.

En este caso creemos que es mejor ir una a una ya que van a ser unas cuantas y además intentaremos emparejarlas con el resultado del análisis anterior.

El primer punto podría ser algo así:

```
It should_check_that_the_event_is_a_RECEIVE_type_event = () =>
   a_receive_type_handling_event.received(x => x.type());
```

Es una aproximación valida, pero si nos fijamos bien, podemos convenir en que equiparar el que haya recibido un evento del tipo `RECEIVE` con que se haya producido al menos una llamada al método `type()` de `a_receive_type_handling_event`, es cuanto menos, confuso.

Podríamos intentar algo más concreto como:

```
It should_check_that_the_event_is_a_RECEIVE_type_event_v2 = () =>
{
    a_receive_type_handling_event
       .was_told_to(x => x.type().Equals(HandlingEventType.RECEIVE));
```

```
    a_receive_type_handling_event.type()
       .ShouldEqual(HandlingEventType.RECEIVE);
};
```

De esta forma si que parece que estamos más cerca de lo que buscamos, ya que incluso si intentamos leer el código da la impresión de que podemos inferir algo parecido a la aserción en si.

Nos quedamos con esta última versión.

El tercer punto podría parecerse a:

```
It should_load_the_first_leg_location = () =>
   the_first_leg.received(x => x.load_location());
```

Poco más que decir. El código hace exactamente lo que queremos.

El quinto punto podría representarse así:

```
It should_leverage_the_handling_event_location = () =>
   a_receive_type_handling_event.received(x => x.location());
```

Al igual que el tercer punto, el código es transparente.

El cuarto punto sería:

```
It should_compare_the_handling_event_and
   _the_initial_departure_locations = () =>
   the_first_leg
      .load_location()
      .received(x => x
      .has_the_same_identity_as(a_receive_type_handling_event.location()));
```

Que aparentemente es un poco más complejo, pero que también refleja perfectamente la intención.

El segundo punto no lo vamos a testar explícitamente, ya que está testado implícitamente en el tercer y quinto punto.

Éste es un concepto muy importante, el de saber cuando debemos parar de asertar para que nuestras **BDD_Specs** reflejen, perfectamente, el sentido original, evitando que los aspectos importantes se pierdan entre los simples detalles.

Hay una última aserción que queremos hacer:

```
It should_check_that_the_handling_event
   _and_the_initial_departure_locations_are_the_same = () =>
   result.ShouldBeTrue();
```

Ésta se encargaría de comprobar el resultado final, el valor que devuelve el método al finalizar su ejecución.

Por lo tanto, poniendo todo en orden, tendríamos algo parecido a:

```
It should_check_that_the_event_is_a_RECEIVE_type_event = () =>
{
    a_receive_type_handling_event
       .was_told_to(x => x.type().Equals(HandlingEventType.RECEIVE));
    a_receive_type_handling_event.type()
       .ShouldEqual(HandlingEventType.RECEIVE);
};

It should_load_the_first_leg_location = () =>
   the_first_leg.received(x => x.load_location());

It should_leverage_the_handling_event_location = () =>
   a_receive_type_handling_event.received(x => x.location());

It should_compare_the_handling_event_and
   _the_initial_departure_locations = () =>
   the_first_leg
      .load_location()
      .received(x =>x
      .has_the_same_identity_as(a_receive_type_handling_event.location()));

It should_check_that_the_handling_event_and
   _the_initial_departure_locations_are_the_same = () =>
   result.ShouldBeTrue();

static bool result;
static IHandlingEvent a_receive_type_handling_event;
static ILeg the_first_leg;
```

3.13.6.5 Obteniendo la BDD_Spec

Podríamos definir formalmente la **BDD_Spec** de la siguiente forma:

```
When asked if a handling event of type RECEIVE was expected
   It should check that the event is a RECEIVE type event.
   It should load the first leg location.
   It should leverage the handling event location.
   It should compare the handling event and the initial departure
locations.
   It should check that the handling event and the initial departure
locations are the same.
```

3.13.6.6 Act

Vamos con el bloque `BECAUSE`, que contrasta por su simplicidad:

```
Because of = () => result = sut.was_expecting(a_receive_type_handling_event);
```

3.13.6.7 Arrange

Y ahora debemos definir el bloque `ESTABLISH`. Probablemente el más complejo que hayamos realizado hasta ahora. Por eso mismo es mejor ir definiendo un mock a cada paso.

El primer mock se encarga de simular que estamos ante un evento de tipo `RECEIVE`:

```
a_receive_type_handling_event
   .Stub(x => x.type())
   .Return(HandlingEventType.RECEIVE);
```

El segundo mock simula el comportamiento deseado al producirse la llamada `load_location()` en `the_first_leg`, devolviendo una `ILocation` llamada `the_expected_location`:

```
the_first_leg
   .Stub(x => x.load_location())
   .Return(the_expected_location);
```

donde definimos `the_expected_location` como:

```
static ILocation the_expected_location;
```

El tercer mock simula la llamada `location()` en `a_receive_type_handling_event`, devolviendo el recién definido `the_expected_location`:

```
a_receive_type_handling_event
   .Stub(x => x.location())
   .Return(the_expected_location);
```

Llegados a este punto es importante resaltar que cuando se producen las dos llamadas que hay en la comparación a sus métodos respectivos para que devuelvan una `ILocation`, nosotros forzamos que esa `ILocation` sea la misma en ambos casos (de ahí el nombre de `the_expected_location`), para de esta forma, obligar a que se cumpla la igualdad (en este caso `has_the_same_identity_as()`).

Y por último, el cuarto mock se encarga de simular la llamada a `has_the_same_identity_as()`, y además devolver `true`, cuando esa llamada se produzca:

```
the_first_leg.load_location()
   .Stub(x => x.has_the_same_identity_as(
                  a_receive_type_handling_event.location()))
   .Return(true);
```

Ahora deberíamos estar mejor preparados para enfrentarnos al bloque `ESTABLISH` al completo:

```
Establish context = () =>
{
    the_expected_location = an<ILocation>();
    a_receive_type_handling_event = an<IHandlingEvent>();

    a_receive_type_handling_event
       .Stub(x => x.type())
       .Return(HandlingEventType.RECEIVE);
    a_receive_type_handling_event
       .Stub(x => x.location())
       .Return(the_expected_location);

    the_collection_of_legs = new List<ILeg>();
```

```
    the_first_leg = an<ILeg>();
    the_collection_of_legs.Add(the_first_leg);

    the_first_leg
       .Stub(x => x.load_location())
       .Return(the_expected_location);
    the_first_leg.load_location()
       .Stub(x => x.has_the_same_identity_as(
                       a_receive_type_handling_event.location()))
       .Return(true);

    create_sut_using(() => new Itinerary(the_collection_of_legs));
};

static ILocation the_expected_location;
static IList<ILeg> the_collection_of_legs;
```

3.13.6.8 RED

Ha llegado la hora de provocar el fallo a nuestras 5 aserciones.

Para ello deberemos invertir el sentido de lo que asertan:

- No vamos a hacer fallar el primer bloque `IT` ya que no queremos perder el hilo con eso. En la **BDD_Spec** de la **RAMA 3** si que haremos fallar como es debido una aserción equivalente.
- Los cuatro primeros bloques `IT`, deben invocar `.never_received()` en vez de `.received()`.
- El último bloque `IT`, debe invocar `ShouldBeFalse()` en vez de `ShouldBeTrue()`.

Ejecutamos la **BDD_Spec**:

```
when asked if a handling event of type RECEIVE was expected
» should check that the event is a RECEIVE type event
» should leverage the handling event location (FAIL)
» should load the first leg location (FAIL)
» should compare the handling event and the initial departure locations
(FAIL)
» should check that the handling event and the initial departure locations
are the same (FAIL)
```

Nos encontramos con 4 fallos:

```
Test 'should leverage the handling event location' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
IHandlingEvent.location(); would not be called, but it was found on the
actual calls made on the mocked object.


Test 'should load the first leg location' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
ILeg.load_location(); would not be called, but it was found on the actual
calls made on the mocked object.


Test 'should compare the handling event and the initial departure
locations' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
IEntity`1.has_the_same_identity_as(ILocationProxyef0b500048b14f3dbfaed0058
c654fa8); would not be called, but it was found on the actual calls made
on the mocked object.


Test 'should check that the handling event and the initial departure
locations are the same' failed:
      Machine.Specifications.SpecificationException: Should be [false] but
is [true]
```

Que son los que esperábamos.

3.13.6.9 GREEN

Volvemos a invertir el sentido de las aserciones (recuperando su sentido original) y ejecutamos los tests.

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if a handling event of type RECEIVE was expected
» should check that the event is a RECEIVE type event
» should leverage the handling event location
» should load the first leg location
» should compare the handling event and the initial departure locations
» should check that the handling event and the initial departure locations
are the same

5 passed, 0 failed, 0 skipped, took 0,94 seconds (Machine.Specifications
0.3.0).
```

Misión cumplida.

3.13.6.10 Preparando la refactorización de una BDD_Spec ruidosa

Una vez que hemos conseguido que los test pasen, parece un buen momento para analizar qué es lo que estamos asertando. Entender si tiene sentido o por el contrario, estamos asertando de más.

Consideremos nuevamente el código que queremos testar:

```
if (the_handling_event.type() == HandlingEventType.RECEIVE)
{
    //Check that the first leg's origin is the event's location
    return the_initial_leg()
              .load_location()
              .has_the_same_identity_as(the_handling_event.location());
}
```

3.13.6.11 Descripción en lenguaje natural

Intentemos, como ejercicio, describir usando lenguaje natural que es lo que hace este código:

> "*Primero comprueba que hemos recibido un evento de tipo RECEIVE y a continuación comprueba si coinciden las ILocations asociadas al primer ILeg y al evento.*"

3.13.6.12 Refactorizando una BDD_Spec ruidosa

Ya sabemos que somos unos máquinas con el **BDD** y que podemos mockar todo lo que se nos ponga por delante, pero...

...según la descripción, estamos enfocando las aserciones de forma equivocada.

Estamos asertando de más, añadiendo **ruido** a lo que de verdad importa.

Estas dos aserciones están de más:

```
When asked if a handling event of type RECEIVE was expected
```

```
   It should leverage the handling event location.
   It should load the first leg location.
```

No aportan nada relevante, son completamente secundarias y por si eso fuese poco, implícitamente se comprueban en esta aserción:

```
When asked if a handling event of type RECEIVE was expected
   It should compare the handling event and the initial departure
locations.
```

Por lo tanto nos deshacemos de ellas.

Activamente decidimos que es algo que tiene que ocurrir pero que no es relevante desde el punto de vista de la **BDD_Spec**.

Por lo tanto nuestra **BDD_Spec** quedaría tal que así:

```
When asked if a handling event of type RECEIVE was expected
   It should check that the event is a RECEIVE type event.
   It should compare the handling event and the initial departure
locations.
   It should check that the handling event and the initial departure
locations are the same.
```

Si ahora hacemos otra pequeña refactorización de la **BDD_Spec**, obtenemos:

```
When asked if we were expecting a handling event of type RECEIVE
   It should check that the event is a RECEIVE type event.
   It should compare the initial departure location with the handling
event location.
   It should confirm that the locations associated to the handling
event and the initial departure are the same.
```

Que coincide a las mil maravillas con nuestro:

> "*Primero comprueba que hemos recibido un evento de tipo RECEIVE y a continuación comprueba si coinciden las ILocations asociadas al primer ILeg y al evento.*"

Ya tenemos **BDD_Spec** y aun por encima, ya no necesitamos este mock:

```
a_receive_type_handling_event
   .Stub(x => x.location())
   .Return(the_expected_location);
```

Con lo que conseguimos simplificar el superpoblado bloque ESTABLISH.

3.13.6.13 Código definitivo de la BDD_Spec

Para que no queden dudas; tras tanto añadir, borrar, cambiar aserciones; éste sería el código final de la **BDD_Spec**:

```
public class when_asked_if_we_were_expecting_a_handling_event_of_type_RECEIVE :
             concern_for_itinerary
{
    Establish context = () =>
    {
        the_expected_location = an<ILocation>();
        a_receive_type_handling_event = an<IHandlingEvent>();

        a_receive_type_handling_event
           .Stub(x => x.type())
           .Return(HandlingEventType.RECEIVE);

        the_collection_of_legs = new List<ILeg>();
        the_first_leg = an<ILeg>();
        the_collection_of_legs.Add(the_first_leg);

        the_first_leg
           .Stub(x => x.load_location())
           .Return(the_expected_location);
        the_first_leg.load_location()
           .Stub(x => x.has_the_same_identity_as(
                           a_receive_type_handling_event.location()))
           .Return(true);

        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };

    Because of = () =>
       result = sut.was_expecting(a_receive_type_handling_event);

    It should_check_that_the_event_is_a_RECEIVE_type_event = () =>
    {
```

```
        a_receive_type_handling_event
           .was_told_to(x => x
           .type()
           .Equals(HandlingEventType.RECEIVE));
        a_receive_type_handling_event
           .type()
           .ShouldEqual(HandlingEventType.RECEIVE);
    };

    It should_compare_the_initial_departure_location
       _with_the_handling_event_location = () =>
       the_first_leg
          .load_location()
          .received(x => x
          .has_the_same_identity_as(a_receive_type_handling_event.location()));

    It should_confirm_that_the_locations_associated_to_the_handling_event
       _and_the_initial_departure_are_the_same = () =>
       result.ShouldBeTrue();

    static bool result;
    static IList<ILeg> the_collection_of_legs;
    static IHandlingEvent a_receive_type_handling_event;
    static ILeg the_first_leg;
    static ILocation the_expected_location;
}
```

3.13.6.14 Legacy code y SRP

Las aserciones están mucho más claras, pero el contexto es tremendo, señalando tal y como razonábamos en un principio, que hace demasiadas cosas (viola el **SRP**). Por lo tanto es un claro candidato a una refactorización.

Destacar antes de pasar a la siguiente **BDD_Spec**, que la única justificación posible para esto es que estamos hablando de **legacy code**, ya que, si hubiésemos diseñado nosotros este código, realmente necesitaríamos repensarnos las cosas muy mucho.

3.13.7 BDD_Spec – was_expecting() - RAMA 3

3.13.7.1 El legacy code

Éste es el **legacy code** que corresponde a la **RAMA 3**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.LOAD)
    {
        //Check that there is one leg with same load location and voyage
        foreach (var leg in underlying_leg_collection)
        {
            if (leg.load_location()
                    .has_the_same_identity_as(the_handling_event.location()) &&
                leg.voyage()
                    .has_the_same_identity_as(the_handling_event.voyage()))
                return true;
        }
        return false;
    }

    (....)
}
```

Si la anterior **BDD_Spec** ponía de manifiesto que el código hacía demasiadas cosas, en esta **BDD_Spec**, parece que vamos a peor. El bloque `ESTABLISH` promete ser monstruoso.

3.13.7.2 Comparación con la RAMA 2

Vamos a intentar aplicar las lecciones aprendidas en la **RAMA 2** para evitar cometer los mismos errores de inicio.

Hay varias diferencias con respecto a la **BDD_Spec** de la **RAMA 2**:

- Ahora no sabemos cual de las `ILegs` de la colección va a ser la buena. Tenemos que ir recorriéndolas hasta que encontremos alguna que cumpla nuestra condición.
- La condición que hay que cumplir ya no afecta solo a una `ILocation`, sino que además debe cumplir la condición un `IVoyage`.

El resto es igual a la **RAMA 2**.

3.13.7.3 Descripción en lenguaje natural

Podríamos intentar definir en lenguaje natural qué es lo que hace el código:

> "*Primero comprueba que hemos recibido un evento de tipo LOAD y a continuación comprueba si coinciden tanto las ILocations como las IVoyages asociadas a alguna de las ILegs de la colección y al evento.*"

3.13.7.4 Obteniendo la BDD_Spec

Por lo tanto nuestra **BDD_Spec**, inicialmente, podría ser la siguiente:

```
When asked if we were expecting a handling event of type LOAD
   It should check that the event is a LOAD type event.
   It should compare the handling event location with every leg load
location until it finds a match.
   It should compare the handling event voyage with the leg voyage if
there is a previous location match.
   It should confirm that there is one leg with the same load location
and voyage as the ones from the handling event.
```

Ya tenemos algo con lo que empezar a trabajar.

3.13.7.5 Assert

Las cuatro aserciones en código:

```
It should_check_that_the_event_is_a_LOAD_type_event = () =>
{
    a_load_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.LOAD));
    a_load_type_handling_event.type()
       .ShouldEqual(HandlingEventType.LOAD);
};

It should_compare_the_handling_event_location
   _with_every_leg_load_location_until_it_finds_a_match = () =>
{
    the_first_leg.load_location()
       .received(x => x
```

```
        .has_the_same_identity_as(a_load_type_handling_event.location()));
    the_middle_leg.load_location()
        .received(x => x
        .has_the_same_identity_as(a_load_type_handling_event.location()));
};

It should_compare_the_handling_event_voyage
   _with_the_leg_voyage_if_theres_a_previous_location_match = () =>
{
    the_first_leg.voyage()
        .never_received(x => x
        .has_the_same_identity_as(a_load_type_handling_event.voyage()));
    the_middle_leg.voyage()
        .received(x => x
        .has_the_same_identity_as(a_load_type_handling_event.voyage()));
};

It should_confirm_that_there_is_one_leg_with_the_same_load_location
   _and_voyage_as_the_ones_from_the_handling_event = () =>
   result.ShouldBeTrue();
```

Que vuelven a confirmar lo mucho que estamos haciendo.

Todo esto debería ser más sencillo.

> **NOTA**: Las aserciones que estamos observando son el resultado de un proceso de compactación y desechado de las aserciones superfluas. Originalmente, todavía era más complejo, con 7 aserciones en lugar de 4. Aun así, el bloque `ESTABLISH`, revela la complejidad del escenario en toda su dimensión.

3.13.7.6 Act

Como siempre, el bloque `BECAUSE`, es conciso y autoexplicativo:

```
Because of = () => result = sut.was_expecting(a_load_type_handling_event);
```

3.13.7.7 Arrange

Y ahora viene el bloque `ESTABLISH`, que es tremendamente complejo (no nos cansaremos de repetirlo, indicando por tanto que hay una violación del **SRP**).

En él creamos una colección de `ILegs` con tres elementos:

- `the_first_leg`
- `the_middle_leg`
- `the_final_leg`

```
the_collection_of_legs = new List<ILeg>();
the_first_leg = an<ILeg>();
the_middle_leg = an<ILeg>();
the_last_leg = an<ILeg>();

the_collection_of_legs.Add(the_first_leg);
the_collection_of_legs.Add(the_middle_leg);
the_collection_of_legs.Add(the_last_leg);
```

Esperamos que `the_first_leg` no cumpla las condiciones:

```
the_first_leg
   .Stub(x => x.load_location())
   .Return(a_not_matching_location);
the_first_leg.load_location()
   .Stub(x =>
x.has_the_same_identity_as(a_load_type_handling_event.location()))
   .Return(false);
the_first_leg
   .Stub(x => x.voyage())
   .Return(an<IVoyage>());
```

Esperamos que sea `the_middle_leg` el elemento en el que se cumplan las dos condiciones:

- su `load_location()` coincide con la `location()` del evento.
- su `voyage()` coincide con el `voyage()` del evento.

```
the_middle_leg
   .Stub(x => x.load_location())
   .Return(the_expected_location);
the_middle_leg.load_location()
   .Stub(x => x.has_the_same_identity_as(
                  a_load_type_handling_event.location()))
   .Return(true);
the_middle_leg
```

```
   .Stub(x => x.voyage())
   .Return(the_expected_voyage);
the_middle_leg.voyage()
   .Stub(x => x.has_the_same_identity_as(a_load_type_handling_event.voyage()))
   .Return(true);
```

No esperamos nada de `the_last_leg`.

Por último, cubrimos la parte de la comprobación del tipo de evento con:

```
a_load_type_handling_event
   .Stub(x => x.type())
   .Return(HandlingEventType.LOAD);
```

Que es equivalente a lo que habíamos visto en la **BDD_Spec** de la **RAMA 2**.

Con esto completamos el código de la **BDD_Spec**, siendo más que recomendable acudir directamente al código fuente, ya que cuando los escenarios son tan complejos, un documento como éste, no es el mejor lugar para mostrar el código.

3.13.7.8 RED-GREEN para aserciones múltiples

Aplicamos el proceso habitual:

- Invertimos el sentido de las aserciones.
- Ejecutamos la **BDD_Spec** a la espera de obtener los fallos deseados.
- Recuperamos el sentido original de las aserciones.
- Ejecutamos la **BDD_Spec**, que ahora debería pasar.

Solo que ahora tenemos aserciones múltiples, como por ejemplo:

```
It should_check_that_the_event_is_a_LOAD_type_event = () =>
{
    a_load_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.LOAD));
    a_load_type_handling_event.type()
       .ShouldEqual(HandlingEventType.LOAD);
};
```

Y para hacer fallar esta `Specification` (usando la nomenclatura `Context / Specification`) que a su vez se compone de dos aserciones, tenemos que hacer fallar una de cada vez.

> **NOTA**: Ya nos encontramos con una aserción múltiple en la **BDD_Spec** de la **RAMA 2**, pero entonces pasamos de puntillas sobre este aspecto.

Vamos paso a paso.

Cambiamos `was_told_to()` por `never_was_told_to()` y comentamos la linea que contiene la segunda aserción.

Ejecutamos la **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if we were expecting a handling event of type LOAD
» should check that the event is a LOAD type event (FAIL)

Test 'should check that the event is a LOAD type event' failed:
      Rhino.Mocks.Exceptions.ExpectationViolationException: Expected that
IHandlingEvent.type(); would not be called, but it was found on the actual
calls made on the mocked object.

(....)

0 passed, 1 failed, 0 skipped, took 1,08 seconds (Machine.Specifications
0.3.0).
```

Que es el fallo que esperábamos

Ahora necesitamos hacer fallar la otra aserción, pero para ello no puede fallar la primera.

Volvemos a cambiar `never_was_told_to()` por `was_told_to()` (lo dejamos, por tanto, tal y como debe estar) y comentamos esa aserción.

Ahora cambiamos el `ShouldEqual()` por `ShouldNotEqual()` y ejecutamos la **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if we were expecting a handling event of type LOAD
» should check that the event is a LOAD type event (FAIL)

Test 'should check that the event is a LOAD type event' failed:
      Machine.Specifications.SpecificationException: Should not equal
dddsample.domain.model.handling.aggregate.HandlingEventType but does:

(....)

0 passed, 1 failed, 0 skipped, took 1,30 seconds (Machine.Specifications
0.3.0).
```

Ya está.

Podemos descomentar la primera aserción y devolverle el sentido original a la segunda aserción (cambiar `ShouldNotEqual()` por `ShouldEqual()`).

Ejecutamos la **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if we were expecting a handling event of type LOAD
» should check that the event is a LOAD type event


1 passed, 0 failed, 0 skipped, took 1,43 seconds (Machine.Specifications
0.3.0).
```

Fantástico. Funciona.

Si hacemos esto mismo con todas las Specifications de la **BDD_Spec**, deberíamos terminar obteniendo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if we were expecting a handling event of type LOAD
» should check that the event is a LOAD type event
» should compare the handling event location with every leg load location
until it finds a match
» should compare the handling event voyage with the leg voyage if theres a
previous location match
» should confirm that there is one leg with the same load location and
```

```
   voyage as the ones from the handling event


   4 passed, 0 failed, 0 skipped, took 0,89 seconds (Machine.Specifications
   0.3.0).
```

Que es exactamente lo que queríamos.

3.13.8 BDD_Spec – was_expecting() - RAMA 4

3.13.8.1 El legacy code

Éste es el **legacy code** que corresponde a la **RAMA 4**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.UNLOAD)
    {
        //Check that the there is one leg with same unload location and voyage
        foreach (var leg in underlying_leg_collection)
        {
            if (leg.unload_location()
                   .has_the_same_identity_as(the_handling_event.location()) &&
                leg.voyage()
                   .has_the_same_identity_as(the_handling_event.voyage()))
               return true;
        }
        return false;
    }

    (....)
}
```

3.13.8.2 Equivalencia con la RAMA 3

Es exactamente lo mismo que con la **RAMA 3** con la salvedad de:

- El evento ahora es de tipo `UNLOAD`.
- Donde antes teníamos `load_location()`, ahora tenemos `unload_location()`.

Por lo tanto no vamos a ahondar más en ello, ya que no aportaría nada nuevo.

Únicamente dejaremos la **BDD_Spec** definida formalmente recordando que siempre podemos acudir directamente al código fuente.

3.13.8.3 Obteniendo la BDD_Spec

La definición formal de la **BDD_Spec**:

```
When asked if we were expecting a handling event of type UNLOAD
   It should check that the event is an UNLOAD type event.
   It should compare the handling event location with every leg unload
location until it finds a match.
   It should compare the handling event voyage with the leg voyage if
there is a previous location match.
   It should confirm that there is one leg with the same unload
location and voyage as the ones from the handling event.
```

3.13.9 BDD_Spec - was_expecting() - RAMA 5

3.13.9.1 El legacy code

Éste es el **legacy code** que corresponde a la **RAMA 5**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.CLAIM)
    {
        //Check that the last leg's destination is from the event's location
        var last_leg = lastLeg();
        return last_leg
                  .unload_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    (....)
}
```

3.13.9.2 Comparación con la RAMA 2

Conceptualmente, estaríamos en el mismo caso que en la **RAMA 2**, por lo tanto no deberíamos perder más tiempo aquí.

3.13.9.3 Refactorización inesperada

Sin embargo, nos encontramos con una particularidad:

```
var last_leg = lastLeg();
```

Éste es el último sitio desde el que se llama al `lastLeg()` original en lugar de a nuestro `the_last_leg()`, así que necesitamos evaluar si sigue siendo necesario o si podemos eliminarlo y cambiar la llamada por una a `last_leg()`.

3.13.9.4 lastLeg() vs the_last_leg()

Recordemos cual es la implementación de `lastLeg()`:

```
ILeg lastLeg()
{
    if (underlying_leg_collection.Count == 0)
       return null;
    return the_last_leg();
}
```

y de `the_last_leg()`:

```
ILeg the_last_leg()
{
    return underlying_leg_collection[underlying_leg_collection.Count - 1];
}
```

Ambos son métodos privados, pero `lastLeg()` es **legacy code**, mientras que `the_last_leg()` es de nuestra cosecha.

3.13.9.5 La misma pregunta de siempre

La pregunta que tenemos que hacernos es la misma de siempre, ¿es factible un

escenario en el cual la colección de `ILegs` esté vacía?

La respuesta sigue siendo la misma. **NO**.

Ya sabemos que:

- No podemos inyectar una colección vacía al constructor porque se produce una excepción.
- La colección está marcada con el atributo `readonly`, por lo que una vez asignada en el constructor, no puede ser cambiada.

3.13.9.6 La refactorización

Podemos, por lo tanto, eliminar definitivamente `lastLeg()` y cambiar el código de la **RAMA 5**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.CLAIM)
    {
        //Check that the last leg's destination is from the event's location
        var last_leg = the_last_leg();
        return last_leg
                  .unload_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    (....)
}
```

Y al igual que hicimos con el código de la **RAMA 2**, podemos compactar todo lo que ocurre dentro del `if`, en una sola linea:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    if (the_handling_event.type() == HandlingEventType.CLAIM)
    {
        //Check that the last leg's destination is from the event's location
```

```
        return the_last_leg()
                  .unload_location()
                  .has_the_same_identity_as(the_handling_event.location());
    }

    (....)
}
```

3.13.9.7 Obteniendo la BDD_Spec

Ahora sí que estamos listos para definir formalmente nuestra **BDD_Spec**, tomando como modelo la de la **RAMA 2**:

```
When asked if we were expecting a handling event of type CLAIM
   It should check that the event is a CLAIM type event.
   It should compare the final arrival unload location with the
handling event location.
   It should confirm that the locations associated to the handling
event and the final arrival are the same.
```

A partir de aquí es muy similar a la **RAMA 2**, por lo que en caso de querer consultar el código, debemos ir a código fuente.

3.13.10 BDD_Spec – was_expecting() - RAMA 6

3.13.10.1 El legacy code

Éste es el **legacy code** que corresponde a la **RAMA 6**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    (....)

    //HandlingEventType.CUSTOMS;
    return true;
}
```

3.13.10.2 Un gestor de eventos confuso

Éste es un caso extraño ya que, carece de la comprobación asociada al tipo de evento

y da por supuesto que si el evento no es de los tipos anteriores, a saber:

- `RECEIVE`
- `LOAD`
- `UNLOAD`
- `CLAIM`

Debe ser del que falta, `CUSTOMS`.

Esta situación es, cuanto menos, confusa, así que vamos a intentar aclararla.

Los tipos de eventos conocidos por la aplicación son cinco y están definidos a través de un `enum` (en la versión original java) o a través de un campo `public readonly` (en esta versión).

Lo que intentamos poner de manifiesto es que los tipos de eventos **no caen del árbol**. Si tenemos 5 tipos de eventos definidos, solamente podemos recibir alguno de esos 5 tipos de eventos. Y a la hora de procesarlos, tal y como estamos haciendo en `was_expected()`, si queremos ponernos a la defensiva, lo lógico sería que tras comprobar **EXPLÍCITAMENTE** que el evento recibido no es uno de los cinco tipos conocidos, algo ha ido definitivamente mal, por lo que deberíamos declarar el evento como no conocido.

3.13.10.3 Refactorización

Por lo tanto, en vez de acabar el procesado del método con el código que hemos presentado al principio de este epígrafe, sería preferible una alternativa del estilo a:

```
//HandlingEvent.Type.CUSTOMS;
if (the_handling_event.type()== HandlingEventType.CUSTOMS)
{
    return true;
}

return false;
```

De esta forma conseguimos:

- Explicitar y aclarar que hay un tipo de evento llamado `CUSTOMS` que puede ser recibido en cualquier momento y que además se considera correcto siempre, en contraposición a las condiciones que hemos ido estableciendo para el resto de los eventos.
- Poner de manifiesto que los tipos de eventos permitidos son los cinco ya definidos y que cualquier tipo de evento alternativo se considerará como erróneo.

Hemos decidido proceder con este cambio, de forma que será el que sometamos al tratamiento **BDD**.

3.13.10.4 Obteniendo la BDD_Spec

No debería suponer ningún desafío esta **BDD_Spec**, teniendo en cuenta todo por lo que hemos pasado últimamente.

La definición formal de la **BDD_Spec** sería:

```
When asked if we were expecting a handling event of type CUSTOMS
   It should check that the event is a CUSTOM type event.
   It should confirm it was expected.
```

3.13.10.5 El código completo de la BDD_Spec

En código se traduciría como:

```
public class when_asked_if_we_were_expecting_a_handling_event_of_type_CUSTOMS :
             concern_for_itinerary
{
    Establish context = () =>
    {
        a_customs_type_handling_event = an<IHandlingEvent>();
        a_customs_type_handling_event
           .Stub(x => x.type())
           .Return(HandlingEventType.CUSTOMS);

        the_collection_of_legs = new List<ILeg>();
```

```
        a_leg = an<ILeg>();
        the_collection_of_legs.Add(a_leg);

        create_sut_using(() => new Itinerary(the_collection_of_legs));
    };

    Because of = () =>
       result = sut.was_expecting(a_customs_type_handling_event);

    It should_check_that_the_event_is_a_CUSTOMS_type_event = () =>
    {
        a_customs_type_handling_event
           .was_told_to(x => x
           .type()
           .Equals(HandlingEventType.CUSTOMS));
        a_customs_type_handling_event.type()
           .ShouldEqual(HandlingEventType.CUSTOMS);
    };

    It should_confirm_it_was_expected = () => result.ShouldBeTrue();

    static bool result;
    static IList<ILeg> the_collection_of_legs;
    static IHandlingEvent a_customs_type_handling_event;
    static ILeg a_leg;
}
```

Poco hay que comentar, tras haber sufrido las monstruosas **BDD_Specs** anteriores.

3.13.10.6 El aspecto de una BDD_Spec saludable

Sin embargo hay una razón importante por la que mostramos este código, y es precisamente, para poder compararlo con las **BDD_Specs** de las **RAMAS** anteriores.

Éste es el aspecto que deben tener las **BDD_Specs**, cuando no estamos violando, por ejemplo, el **SRP**:

- Sencillas (que no simples).
- Con aserciones directas y comprensibles.
- Con un contexto abarcable.

3.13.10.7 RED-GREEN

Tal y como suponemos, la **BDD_Spec** se cumple (debemos recordar obtener el fallo significativo antes de que pase):

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if we were expecting a handling event of type CUSTOMS
» should check that the event is a CUSTOMS type event
» should confirm it was expected


2 passed, 0 failed, 0 skipped, took 1,73 seconds (Machine.Specifications
0.3.0).
```

3.13.10.8 Consecuencias de nuestra última refactorización

Una última cuestión.

Con nuestra reciente modificación de código, surge una nueva **RAMA**, la **RAMA 7.**

3.13.11 BDD_Spec – was_expecting() - RAMA 7

3.13.11.1 El legacy code

Éste es el **legacy code** que corresponde a la **RAMA 7**:

```
public bool was_expecting(IHandlingEvent the_handling_event)
{
    if (the_handling_event.type() == HandlingEventType.RECEIVE) (....)
    if (the_handling_event.type() == HandlingEventType.LOAD) (....)
    if (the_handling_event.type() == HandlingEventType.UNLOAD) (....)
    if (the_handling_event.type() == HandlingEventType.CLAIM) (....)
    if (the_handling_event.type() == HandlingEventType.CUSTOMS) (....)
    return false;
}
```

Estamos suponiendo que fallan todas las comprobaciones de los `if` y que devolvemos `false`, ya que eso significaría que hemos recibido un evento de tipo desconocido.

3.13.11.2 Assert

Vamos con las aserciones:

```
It should_check_that_the_event_is_an_unknown_type_event = () =>
{
    an_unknown_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.RECEIVE));
    an_unknown_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.LOAD));
    an_unknown_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.UNLOAD));
    an_unknown_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.CLAIM));
    an_unknown_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.CUSTOMS));

    an_unknown_type_handling_event.type()
       .ShouldNotEqual(HandlingEventType.RECEIVE);
    an_unknown_type_handling_event.type()
       .ShouldNotEqual(HandlingEventType.LOAD);
    an_unknown_type_handling_event.type()
       .ShouldNotEqual(HandlingEventType.UNLOAD);
    an_unknown_type_handling_event.type()
       .ShouldNotEqual(HandlingEventType.CLAIM);
    an_unknown_type_handling_event.type()
       .ShouldNotEqual(HandlingEventType.CUSTOMS);
};

It should_confirm_it_was_not_expected = () => result.ShouldBeFalse();
```

Esto sí que es interesante.

3.13.11.3 Las BDD_Specs como documentación

El primer bloque `IT`, probablemente rompa con todas los convenios implícitos asociados al buen uso de las aserciones, ya que, la primera norma, es intentar que

sean del estilo de nuestro segundo bloque `IT`, una sola linea.

Si nos detenemos un instante a reflexionar, la **RAMA 7** se comprueba únicamente con el segundo bloque `IT`, por lo tanto no parece que tenga mucho sentido romper parte del libro de estilo del **BDD** (más en concreto del uso correcto de `machine.specifications`), a no ser que consideremos que las **BDD_Specs**, en si, son **LA DOCUMENTACIÓN** de las clases que estamos testando.

Si tenemos en cuenta este último punto, entonces, el primer bloque `IT` sí que cobra una importancia tremenda, ya que **EXPLICITA**, que el caso que estamos testando es completamente extraordinario y no puede ser englobado en ninguno de los casos conocidos y esperados.

Unas advertencias:

- Usar (o no) este primer bloque `IT` es una cuestión personal, de estilo. Nosotros hemos decidido que en este caso concreto aporta lo que buscamos.
- En cualquier otro caso, usar un bloque del estilo de nuestro primer bloque `IT` debe ser considerado poco menos que una aberración. Es la excepción, no la norma.

3.13.11.4 Act

Vamos a continuar con nuestro bloque `BECAUSE`:

```
Because of = () => result = sut.was_expecting(an_unknown_type_handling_event);
```

No hay sorpresas en este caso.

Con un bloque como nuestro primer bloque `IT` hemos cubierto más que de sobra nuestro cupo de sorpresas.

3.13.11.5 Arrange

Finalmente, el bloque `ESTABLISH`, con los campos asociados, que permiten que todo

funcione:

```
Establish context = () =>
{
    an_unknown_type_handling_event = an<IHandlingEvent>();
    an_unknown_type_handling_event
       .Stub(x => x.type())
       .Return(an<IHandlingEventType>());

    the_collection_of_legs = new List<ILeg>();
    a_leg = an<ILeg>();
    the_collection_of_legs.Add(a_leg);

    create_sut_using(() => new Itinerary(the_collection_of_legs));
};

static bool result;
static IList<ILeg> the_collection_of_legs;
static IHandlingEvent an_unknown_type_handling_event;
static ILeg a_leg;
```

Hay algo que queremos destacar, ya que nos parece muy interesante.

3.13.11.6 La justificación a todas nuestra molestias

Fijémonos en el mock, en concreto en el `.Return()`:

```
an_unknown_type_handling_event
   .Stub(x => x.type())
   .Return(an<IHandlingEventType>());
```

Éste es el ejemplo perfecto que demuestra la potencia del **BDD** al asociarla con el **Interface-Based Programming** y en cierta forma también a la "**I**" de **S.O.L.I.D.** (**Interface Seggregation Principle**) y justifica de por si, toda la deriva que hemos tomado desde el comienzo con respecto al ejemplo original java.

Vamos a intentar explicarlo.

Gracias a que nos protegemos de las implementaciones concretas como `HandlingEventType` a través de la interface `IHandlingEventType`, podemos:

- Crear un mock de tipo `IHandlingEventType` (imposible si se tratase de una clase

concreta).

- Resolver el contexto.

Vamos a mostrar el código de `HandlingEventType` para poder ilustrar todo esto:

```
namespace dddsample.domain.model.handling.aggregate
{
    public class HandlingEventType : IHandlingEventType
    {
        public static readonly IHandlingEventType RECEIVE =
                                                    new HandlingEventType();
        public static readonly IHandlingEventType LOAD =
                                                    new HandlingEventType();
        public static readonly IHandlingEventType UNLOAD=
                                                    new HandlingEventType();
        public static readonly IHandlingEventType CLAIM =
                                                    new HandlingEventType();
        public static readonly IHandlingEventType CUSTOMS =
                                                    new HandlingEventType();

        private HandlingEventType(){}
    }
}
```

Y vamos a retomar el tema de resolver el contexto.

Supongamos que `type()` devolviese el tipo concreto `HandlingEventType`. Es decir, que su signatura fuese la siguiente:

```
namespace dddsample.domain.model.handling.aggregate
{
    public interface IHandlingEvent : IDomainEvent<IHandlingEvent>
    {
        HandlingEventType type();
    }
}
```

y que `HandlingEventType` no implementase la interface `IHandlingEventType`:

```
namespace dddsample.domain.model.handling.aggregate
{
    public class HandlingEventType
{
        public static readonly HandlingEventType RECEIVE =
                                                    new HandlingEventType();
```

```
        public static readonly HandlingEventType LOAD =
                                                    new HandlingEventType();
        public static readonly HandlingEventType UNLOAD =
                                                    new HandlingEventType();
        public static readonly HandlingEventType CLAIM =
                                                    new HandlingEventType();
        public static readonly HandlingEventType CUSTOMS =
                                                    new HandlingEventType();

        private HandlingEventType(){}
    }
}
```

Supongamos que ahora necesitásemos crear el contexto. En concreto, algo equivalente a:

```
a_customs_type_handling_event
   .Stub(x => x.type())
   .Return(an<IHandlingEventType>());
```

Ante esto debemos hacer la siguiente reflexión:

- No podemos crear el mock, así que tendremos que crear el tipo concreto `UNKNOWN`, por poner un ejemplo.
- El constructor de `HandlingEventType` es privado, así que no podemos invocarlo desde nuestro contexto (bloque `ESTABLISH` de nuestra **BDD_Spec**).

Por lo tanto podemos establecer las siguientes conclusiones:

- No podemos simular el escenario.
- **Escenario Intestable**.

Éstas son las pequeñas cosas que hacen que nuestro esfuerzo merezca la pena.

3.13.11.7 RED-GREEN

Necesitamos nuestro fallo significativo.

Para el segundo bloque `IT`, es trivial.

Para el primer bloque `IT`, necesitamos un carro lleno de paciencia. Tenemos que ir **negando** cada una de las aserciones:

- Cambiando `was_told_to()` por `was_never_told_to()`.
- Cambiando `ShouldNotEqual()` por `ShouldEqual()`.

e ir comentando todas las otras, para asegurarnos de que es ésa la que falla.

Una vez terminado este proceso ligeramente tedioso y habiendo devuelto las aserciones a su sentido original, obtendremos nuestra recompensa:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if we were expecting a handling event of type unknown
» should check that the event is an unknown type event
» should confirm it was not expected


2 passed, 0 failed, 0 skipped, took 0,70 seconds (Machine.Specifications
0.3.0).
```

3.13.12 Los Happy Day Scenarios

Si bien no lo hemos especificado, los escenarios descritos por las **RAMAS 2 a 5**, describen los Happy Day Scenarios. Solo tenemos cubierta la parte que devuelve `true`. Por lo tanto, necesitamos cubrir sus complementarios, los que devuelven `false`.

3.13.13 Expresiones Lambda y LINQ

3.13.13.1 Refactor

Antes de acometer estas cuatro **BDD_Specs**, vamos a refactorizar el código de las **RAMAS 3 y 4**:

```
if (the_handling_event.type() == HandlingEventType.LOAD)
{
    foreach (var leg in underlying_leg_collection)
    {
        if (leg.load_location()
```

```
                .has_the_same_identity_as(the_handling_event.location()) &&
             leg.voyage()
                .has_the_same_identity_as(the_handling_event.voyage()))
        return true;
    }
    return false;
}

if (the_handling_event.type() == HandlingEventType.UNLOAD)
{
    foreach (var leg in underlying_leg_collection)
    {
        if (leg.unload_location()
                .has_the_same_identity_as(the_handling_event.location()) &&
             leg.voyage()
                .has_the_same_identity_as(the_handling_event.voyage()))
             return true;
    }
    return false;
}
```

Podemos transformar esos bucles `foreach` en **expresiones Lambda** con **LINQ**.

Por lo tanto:

```
if (the_handling_event.type() == HandlingEventType.LOAD)
{
    return underlying_leg_collection
              .Any(leg => leg
              .load_location()
              .has_the_same_identity_as(the_handling_event.location()) &&
           leg.voyage()
              .has_the_same_identity_as(the_handling_event.voyage()));
}

if (the_handling_event.type() == HandlingEventType.UNLOAD)
{
    return underlying_leg_collection
              .Any(leg => leg
              .unload_location()
              .has_the_same_identity_as(the_handling_event.location()) &&
           leg.voyage()
              .has_the_same_identity_as(the_handling_event.voyage()));
}
```

Estamos muy limitados por culpa del ancho del documento, para mostrar de forma clara este código. Es mucho más sencillo de entender en el código fuente.

Si ejecutamos las **BDD_Specs** comprobaremos que no hemos roto nada.

Vamos ahora con las **BDD_Specs** complementarias.

3.13.14 BDD_Spec – was_expecting() - RAMA 2 COMPLEMENTARIA

3.13.14.1 El legacy code

Recordemos que éste es el **legacy code** que corresponde a la **RAMA 2**:

```
if (the_handling_event.type() == HandlingEventType.RECEIVE)
    return the_initial_leg()
              .load_location()
              .has_the_same_identity_as(the_handling_event.location());
```

3.13.14.2 Obteniendo la BDD_Spec

No tiene ninguna dificultad desarrollar la **BDD_Spec** basándonos en la existente para el Happy Day Scenario, por lo tanto, no lo mostraremos. Se puede consultar el código fuente si se necesita ver la implementación.

Lo único que mostraremos sera la definición formal de la **BDD_Spec**:

```
When asked if we were expecting a handling event of type RECEIVE and we
were not
   It should check that the event is a RECEIVE type event.
   It should compare the initial departure location with the handling
event location.
   It should confirm that the locations associated to the handling
event and the initial departure are different.
```

3.13.15 BDD_Spec – was_expecting() - RAMA 3 COMPLEMENTARIA

3.13.15.1 El legacy code

Recordemos que éste es el **legacy code**, refactorizado, que corresponde a la **RAMA 3**:

```
if (the_handling_event.type() == HandlingEventType.LOAD)
{
    return underlying_leg_collection
              .Any(leg => leg
              .load_location()
              .has_the_same_identity_as(the_handling_event.location()) &&
           leg.voyage()
              .has_the_same_identity_as(the_handling_event.voyage()));
}
```

Necesitamos testar el comportamiento que se produce cuando la consulta **LINQ** no encuentra ningún elemento que cumpla las condiciones y por tanto devuelve `false`.

Para ello vamos a tomar como modelo la **BDD_Spec** del Happy Day Scenario de la **RAMA 3**.

3.13.15.2 Obteniendo la BDD_Spec

Ésta podría ser nuestra **BDD_Spec**:

```
When asked if we were expecting a handling event of type LOAD and we
were not
   It should check that the event is a LOAD type event.
   It should compare the handling event location with every leg load
location.
   It should confirm that we were not expecting this event.
```

Que podríamos refinar ligeramente de forma que:

```
When receiving an unexpected handling event of type LOAD
   It should check that the event is a LOAD type event.
   It should compare the handling event location with every leg load
location.
   It should confirm that the event was unexpected.
```

Mucho más concisa, natural y autoexplicativa.

3.13.15.3 Assert

Una posible implementación de las aserciones sería:

```
It should_check_that_the_event_is_a_LOAD_type_event = () =>
{
    a_load_type_handling_event
       .was_told_to(x => x
       .type()
       .Equals(HandlingEventType.LOAD));
    a_load_type_handling_event.type()
       .ShouldEqual(HandlingEventType.LOAD);
};

It should_compare_the_handling_event_location
   _with_every_leg_load_location = () =>
{
    the_first_leg.load_location()
       .received(x => x
       .has_the_same_identity_as(a_load_type_handling_event.location()));
    the_middle_leg.load_location()
       .received(x => x
       .has_the_same_identity_as(a_load_type_handling_event.location()));
    the_last_leg.load_location()
       .received(x => x
       .has_the_same_identity_as(a_load_type_handling_event.location()));
};

It should_confirm_that_the_event_was_unexpected = () =>
   result.ShouldBeFalse();
```

Centrémonos por un instante en el bloque `IT` intermedio:

```
It should_compare_the_handling_event_location
   _with_every_leg_load_location = () =>
{
    the_first_leg.load_location()
       .received(x => x
       .has_the_same_identity_as(a_load_type_handling_event.location()));
    the_middle_leg.load_location()
       .received(x => x
       .has_the_same_identity_as(a_load_type_handling_event.location()));
    the_last_leg.load_location()
       .received(x => x
       .has_the_same_identity_as(a_load_type_handling_event.location()));
};
```

Lo único que pretendemos es asertar que se ha recorrido la colección y no hemos encontrado ningún elemento que cumpla la condición.

3.13.15.4 Refactorizando con each<T>()

Gracias a **Jean-Paul Boodhoo** y sus extensiones `developwithpassion`, podemos hacer esto mismo de una forma mucho más compacta y elegante, usando:

```
public static void each<T>(this IEnumerable<T> items, Action<T> action)
```

Un **extension method** que se encuentra en:

```
Machine.Specifications.DevelopWithPassion.Extensions.IterationExtensions.
```

Ésta sería la alternativa:

```
It should_compare_the_handling_event_location
   _with_every_leg_load_location_v2 = () =>
   the_collection_of_legs
      .each(leg => leg
      .load_location()
      .received(x => x
      .has_the_same_identity_as(a_load_type_handling_event.location())));
```

Es mucho más clara (todavía más si se consulta el código fuente).

Por lo tanto nos quedaremos con esta nueva versión.

3.13.15.5 Act

El bloque `BECAUSE` es el mismo que en el **RAMA 3**:

```
Because of = () => result = sut.was_expecting(a_load_type_handling_event);
```

3.13.15.6 Arrange

Y el contexto, definitivamente sigue siendo monstruoso:

```
Establish context = () =>
{
    a_load_type_handling_event = an<IHandlingEvent>();
    a_load_type_handling_event
       .Stub(x => x.type())
```

```
        .Return(HandlingEventType.LOAD);

    the_collection_of_legs = new List<ILeg>();
    the_first_leg = an<ILeg>();
    the_middle_leg = an<ILeg>();
    the_last_leg = an<ILeg>();
    the_collection_of_legs.Add(the_first_leg);
    the_collection_of_legs.Add(the_middle_leg);
    the_collection_of_legs.Add(the_last_leg);

    the_first_leg
        .Stub(x => x.load_location())
        .Return(an<ILocation>());
    the_first_leg.load_location()
        .Stub(x => x.has_the_same_identity_as(
                        a_load_type_handling_event.location()))
        .Return(false);
    the_middle_leg
        .Stub(x => x.load_location())
        .Return(an<ILocation>());
    the_middle_leg.load_location()
        .Stub(x => x.has_the_same_identity_as(
                        a_load_type_handling_event.location()))
        .Return(false);
    the_last_leg
        .Stub(x => x.load_location())
        .Return(an<ILocation>());
    the_last_leg.load_location()
        .Stub(x => x.has_the_same_identity_as(
                        a_load_type_handling_event.location()))
        .Return(false);

    create_sut_using(() => new Itinerary(the_collection_of_legs));
};
```

Resumiendo:

- En el **primer bloque** forzamos el tipo de evento que queremos (`LOAD`).
- En el **segundo bloque** creamos y poblamos la colección de `ILegs`.
- En el **tercer bloque** definimos el comportamiento esperado cuando se produce la llamada `load_location()` en cada uno de las `ILegs` de la colección.
- En el **cuarto bloque** creamos el SUT inyectando la colección de `ILegs`.

3.13.15.7 Dos refactorizaciones más gracias a each<T>()

Podemos refactorizar el código en dos puntos distintos:

La clave para la primera refactorización la encontramos en:

> "*...en cada uno de los ILegs de la colección.*"

Ya que nos sugiere la posibilidad de poner en uso el **extension method** recién aprendido `each<T>()` en el **tercer bloque,** para asignar el comportamiento de los mocks:

```
the_collection_of_legs
   .each(leg => leg
   .Stub(x => x.load_location())
   .Return(an<ILocation>()));
the_collection_of_legs
   .each(leg =>leg.load_location()
   .Stub(x => x.has_the_same_identity_as(
                  a_load_type_handling_event.location()))
   .Return(false));
```

Al generalizarse el acceso a los miembros de la colección con esta **refactorización**, podemos prescindir de:

- `the_first_leg`
- `the_middle_leg`
- `the_last_leg`

en el **segundo bloque**, permitiéndonos hacer lo mismo pero de esta forma:

```
the_collection_of_legs = new List<ILeg>();
for (var i = 0; i < 3; i++)
    the_collection_of_legs.Add(an<ILeg>());
```

Por lo tanto, nuestro contexto se convierte en algo mucho más manejable y comprensible:

```
Establish context = () =>
{
    a_load_type_handling_event = an<IHandlingEvent>();
    a_load_type_handling_event
       .Stub(x => x.type())
       .Return(HandlingEventType.LOAD);
```

```
    the_collection_of_legs = new List<ILeg>();
    for (var i = 0; i < 3; i++)
        the_collection_of_legs.Add(an<ILeg>());

    the_collection_of_legs
       .each(leg => leg
       .Stub(x => x.load_location())
       .Return(an<ILocation>()));
    the_collection_of_legs
       .each(leg =>leg.load_location()
       .Stub(x => x.has_the_same_identity_as(
                       a_load_type_handling_event.location()))
       .Return(false));

    create_sut_using(() => new Itinerary(the_collection_of_legs));
};
```

Antes de que se nos olvide. También hemos disminuido la cantidad de campos necesarios:

```
static bool result;
static IList<ILeg> the_collection_of_legs;
static IHandlingEvent a_load_type_handling_event;
```

3.13.15.8 RED-GREEN

Lo único que nos queda, es hacer fallar los tests y a continuación hacerlos pasar (algo que a estas alturas estamos aburridos de hacer).

El resultado final:

```
------ Test started: Assembly: dddsample.specs.dll ------

when receiving an unexpected handling event of type LOAD
» should check that the event is a LOAD type event
» should compare the handling event location with every leg load location
» should confirm that the event was unexpected


3 passed, 0 failed, 0 skipped, took 1,25 seconds (Machine.Specifications
0.3.0).
```

3.13.16 Refactorizando las definiciones de las BDD_Specs

Hemos redefinido la forma en la que nombrábamos nuestras `Specifications` de:

```
When asked if we were expecting a handling event of type LOAD and we
were not
```

a:

```
When receiving an unexpected handling event of type LOAD
```

Por lo tanto deberíamos proceder a cambiar todas las `Specifications` relativas a `was_expecting()`, siguiendo este patrón.

Una vez terminemos de implementar las **BDD_Specs** restantes, mostraremos el resultado.

3.13.17 BDD_Spec – was_expecting() - RAMAS 4 y 5 COMPLEMENTARIAS

3.13.17.1 Obteniendo las BDD_Specs

La **RAMA 4 COMPLEMENTARIA** sería equivalente a la **RAMA 3 COMPLEMENTARIA**, mientras que la **RAMA 5 COMPLEMENTARIA** sería equivalente a la **RAMA 2 COMPLEMENTARIA**.

Puesto que no aportan nada nuevo, no encontramos justificación para repetirnos. Por tanto hemos decidido mostrar únicamente la definición formal de las **BDD_Specs**, de forma que si necesitamos consultar el código, podamos referirnos directamente al código fuente, donde sí están implementadas.

BDD_Spec de la **RAMA 4 COMPLEMENTARIA**:

```
When receiving an unexpected handling event of type UNLOAD
   It should check that the event is an UNLOAD type event.
```

```
   It should compare the handling event location with every leg unload
location.
   It should confirm that the event was unexpected.
```

BDD_Spec de la **RAMA 5 COMPLEMENTARIA**:

```
When receiving an unexpected handling event of type CLAIM
   It should check that the event is a CLAIM type event.
   It should compare the final arrival unload location with the
handling event location.
   It should confirm that the locations associated to the handling
event and the final arrival are different.
```

3.13.18 Las definiciones de las BDD_Specs refactorizadas

Anteriormente aseguramos que en cuanto hubiésemos terminado de implementar las **BDD_Specs** restantes asociadas a `was_expecting()`, mostraríamos como habían quedado tras la refactorización de nombres.

He aquí el resultado, en el contexto de una ejecución de todas las **BDD_Specs** que llevamos escritas hasta ahora:

```
------ Test started: Assembly: dddsample.specs.dll ------
(....)

when receiving an expected handling event of type RECEIVE
» should check that the event is a RECEIVE type event
» should compare the initial departure location with the handling event
location
» should confirm that the locations associated to the handling event and
the initial departure are the same

when receiving an unexpected handling event of type RECEIVE
» should check that the event is a RECEIVE type event
» should compare the initial departure location with the handling event
location
» should confirm that the locations associated to the handling event and
the initial departure are different

when receiving an expected handling event of type LOAD
» should check that the event is a LOAD type event
» should compare the handling event location with every leg load location
until it finds a match
```

```
» should compare the handling event voyage with the leg voyage if theres a
previous location match
» should confirm that there is one leg with the same load location and
voyage as the ones from the handling event

when receiving an unexpected handling event of type LOAD
» should check that the event is a LOAD type event
» should compare the handling event location with every leg load location
» should confirm that the event was unexpected

when receiving an expected handling event of type UNLOAD
» should check that the event is an UNLOAD type event
» should compare the handling event location with every leg unload
location until it finds a match
» should compare the handling event voyage with the leg voyage if theres a
previous location match
» should confirm that there is one leg with the same unload location and
voyage as the ones from the handling event

when receiving an unexpected handling event of type UNLOAD
» should check that the event is an UNLOAD type event
» should compare the handling event location with every leg unload
location
» should confirm that the event was unexpected

when receiving an expected handling event of type CLAIM
» should check that the event is a CLAIM type event
» should compare the final arrival unload location with the handling event
location
» should confirm that the locations associated to the handling event and
the final arrival are the same

when receiving an unexpected handling event of type CLAIM
» should check that the event is a CLAIM type event
» should compare the final arrival unload location with the handling event
location
» should confirm that the locations associated to the handling event and
the final arrival are different

when receiving an expected handling event of type CUSTOMS
» should check that the event is a CUSTOMS type event
» should confirm it was expected

when receiving an unexpected handling event of type unknown
» should check that the event is an unknown type event
» should confirm it was not expected

(...)

160 passed, 0 failed, 0 skipped, took 2,09 seconds (Machine.Specifications
0.3.0).
```

3.14 Limpieza final de Itinerary

Prácticamente hemos finalizado el proceso de **BDDización** del **legacy code** `Itinerary`.

Vamos a aprovechar esta sección para ver qué es lo que nos ha quedado sin testar y decidir que alternativas tenemos.

3.14.1 Problema 1 - EMPTY_ITINERARY

```
public class Itinerary : IItinerary
{
    /// <summary>
    /// NULL OBJECT PATTERN????
    /// </summary>
    public static readonly IItinerary EMPTY_ITINERARY = new Itinerary();
    private Itinerary() {}

    (....)
}
```

`EMPTY_ITINERARY` parece una aproximación al **Null Object Pattern**.

Para implementarlo hace uso del constructor privado sin parámetros.

Hasta el momento no lo hemos usado en ninguna parte, y puesto que es difícilmente testable hemos decidido eliminarlo.

Recordar que hay que ejecutar la suite de **BDD_Specs** para asegurarnos de que no hacemos nada que altere el comportamiento de nuestro sistema:

```
160 passed, 0 failed, 0 skipped, took 3,10 seconds (Machine.Specifications
0.3.0).
```

3.14.2 Problema 2 - END_OF_DAYS

```
public class Itinerary : IItinerary
{
    static readonly IDate END_OF_DAYS = new Date(DateTime.MaxValue);

    (....)
```

```
}
```

La **constante** `END_OF_DAYS` era utilizada por el método `final_arrival_date()` antes de que le aplicásemos la terapia de choque **BDD**:

```
public IDate final_arrival_date()
{
    ILeg lastLeg = this.lastLeg();

    if (lastLeg == null)
       return END_OF_DAYS;
    return lastLeg.unload_time();
}
```

Como en nuestra nueva versión de `final_arrival_date()` no lo utilizamos:

```
public IDate final_arrival_date()
{
    return the_last_leg().unload_time();
}
```

Podemos borrarlo.

No nos cansaremos de recordar que hay que ejecutar la suite de **BDD_Specs** para asegurarnos de que no hacemos nada que altere el comportamiento de nuestro sistema:

```
160 passed, 0 failed, 0 skipped, took 2,80 seconds (Machine.Specifications
0.3.0).
```

3.14.3 Problema 3 - Location.UNKNOWN

Recordemos el siguiente código:

```
public ILocation initial_departure_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN;
    return underlying_leg_collection[0].load_location();
}
```

La nueva versión de ese método no utiliza `LocationImplementationExample.UNKNOWN`,

que recordemos, creamos un poco "de aquella manera", para poder simular completamente nuestro ejercicio **legacy code**.

La nueva versión:

```
public ILocation initial_departure_load_location()
{
    return the_initial_leg().load_location();
}
```

El otro sitio donde se utilizaba era en el método `final_arrival_location()`:

```
public ILocation final_arrival_location()
{
    if (underlying_leg_collection.Count == 0)
       return LocationImplementationExample.UNKNOWN;
    return lastLeg().unload_location();
}
```

Pero la nueva versión tampoco hace uso de ella:

```
public ILocation final_arrival_unload_location()
{
    return the_last_leg().unload_location();
}
```

Por lo tanto todo este código puede desaparecer:

```
using System;

namespace dddsample.domain.model.location.aggregate
{
    public class LocationImplementationExample : ILocation
    {
        /// <summary>
        /// Special Location object that marks an unknown location.
        /// </summary>
        public static readonly LocationImplementationExample UNKNOWN =
           new LocationImplementationExample();

        public bool has_the_same_identity_as(ILocation the_other_entity)
        {
            throw new NotImplementedException();
        }
```

```
        public ILocation location_unknown()
        {
            throw new NotImplementedException();
        }
    }
}
```

Ejecutamos la suite de **BDD_Specs**:

```
160 passed, 0 failed, 0 skipped, took 2,53 seconds (Machine.Specifications
0.3.0).
```

3.14.4 Problema 4 - COMENTARIOS

Hemos ido eliminándolos poco a poco, a medida que ganábamos conocimiento interno del funcionamiento de la clase e íbamos introduciendo una estrategia de nombrado que hiciese que los comentarios fuesen simple y llanamente redundantes.

Sin embargo todavía hay dos excepciones:

3.14.4.1 El metodo legs()

```
/// <summary>
///
/// </summary>
/// <returns>The legs of this itinerary, as a list.</returns>
public IList<ILeg> legs()
{
    // TODO: Return an IEnumerable.
    return this.underlying_leg_collection;
}
```

Vamos a presuponer que tenemos acceso no solo al código de esta clase, sino también al global de la aplicación, de forma que ayudándonos de una herramienta como **ReSharper**, podamos cambiar el nombre de un método y automáticamente asegurarnos que ese renombrado se ha producido coherentemente en todos los puntos del código donde se invocaba.

En ese supuesto, optaríamos por renombrar el método.

Analicemos algo de código cliente de nuestro objetivo, como por ejemplo el propio `Itinerary`:

```
underlying_leg_collection.Equals(the_other_itinerary.legs());
```

Una alternativa podría ser:

```
underlying_leg_collection.Equals(the_other_itinerary.associated_legs());
```

Ya tenemos nuevo nombre, así que, podemos borrar los comentarios ya que ahora no aportan nada nuevo:

```
public IList<ILeg> associated_legs()
{
    // TODO: Return an IEnumerable.
    return this.underlying_leg_collection;
}
```

3.14.4.2 La clase Itinerary

El otro punto conflictivo con respecto a los comentarios es la propia clase `Itinerary`:

```
namespace dddsample.domain.model.cargo.aggregate
{
    /// <summary>
    /// An itinerary.
    /// Originally, a direct port of the java example in order to simulate
    /// legacy code to support the argument presented in the Documentation.
    /// Currently, the refactored version driven by the argument
    /// presented in the Documentation.
    /// </summary>
    public class Itinerary : IItinerary
```

Pero en este caso, los comentarios, sí que nos parecen relevantes, ya que aportan algo que el código no puede aportar.

Por lo tanto estos se quedan.

3.14.5 Problema 5 - IEnumerable<ILeg>

```
public IList<ILeg> associated_legs()
{
    // TODO: Return an IEnumerable.
    return this.underlying_leg_collection;
}
```

Tal y como indica el **TODO**, deberíamos devolver un `IEnumerable<ILeg>` en vez de una `IList<ILeg>`, y probablemente materializaremos este cambio en cuanto empecemos a hacer uso del método `associated_legs()`.

Pero por el momento no vamos a tocarlo.

Eso si, la marca del **TODO**, permanece.

3.15 APENDICE: El Value Object Itinerary ANTES y DESPUES

A modo de documentación asociada, exponemos el código antes de crear una sola **BDD_Spec** y después de haber creado nuestra suite de **BDD_Specs**, de forma que sea más sencillo poder analizar el impacto de la aplicación del **BDD** a un escenario **legacy code**.

3.15.1 La clase original

Ésta sería la clase original, que simulaba el escenario **legacy code**:

```
using System;
using System.Collections.Generic;
using System.Data;
using dddsample.domain.model.handling.aggregate;
using dddsample.domain.model.location.aggregate;

namespace dddsample.domain.model.cargo.aggregate
{
    /// <summary>
    /// An itinerary.
    /// Direct port of the java example in order to simulate legacy code
    /// to support the argument presented in the Documentation.
    /// </summary>
    public class Itinerary : IItinerary
    {
        /// <summary>
        /// NULL OBJECT PATTERN????
        /// </summary>
        public static readonly IItinerary EMPTY_ITINERARY = new Itinerary();
        private Itinerary(){}

        static readonly IDate END_OF_DAYS = new Date(DateTime.MaxValue);


        IList<ILeg> underlying_leg_collection = new List<ILeg>();

        /// <summary>
        /// Constructor.
        /// </summary>
        /// <param name="legs">List of legs for this itinerary.</param>
        public Itinerary(IList<ILeg> legs)
        {
            if (legs.Count == 0)
                throw new ArgumentNullException(
                            "legs",
```

```
                    "The list of legs cannot be empty.");

        ILeg null_leg = null;
        if (legs.Contains(null_leg))
            throw new NoNullAllowedException(
                    "The legs list cannot contain null elements.");

        this.underlying_leg_collection = legs;
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns>The legs of this itinerary, as a list.</returns>
    public IList<ILeg> legs()
    {
        // TODO: Return an IEnumerable.
        return this.underlying_leg_collection;
    }

    /// <summary>
    /// Tests if the given handling event is expected
    /// when executing this itinerary.
    /// </summary>
    /// <param name="handling_event">Event to test.</param>
    /// <returns><code>true</code> if the event is expected.</returns>
    public bool isExpected(IHandlingEvent handling_event)
    {
        if (underlying_leg_collection.Count == 0)
        {
            return true;
        }

        if (handling_event.type() == HandlingEventType.RECEIVE)
        {
            //Check that the first leg's origin is the event's location
            var leg = underlying_leg_collection[0];
            return leg.load_location()
                      .has_the_same_identity_as(handling_event.location());
        }

        if (handling_event.type() == HandlingEventType.LOAD)
        {
            //Check that there's one leg with same load location and voyage
            foreach (var leg in underlying_leg_collection)
            {
                if (leg.load_location()
                       .has_the_same_identity_as(
                          handling_event.location()) &&
                    leg.voyage()
                       .has_the_same_identity_as(handling_event.voyage()))
                    return true;
```

```
            }
            return false;
        }

        if (handling_event.type() == HandlingEventType.UNLOAD)
        {
            //Check that the there's one leg with
            //same unload location and voyage
            foreach (var leg in underlying_leg_collection)
            {
                if (leg.unload_location()
                       .has_the_same_identity_as(
                          handling_event.location()) &&
                    leg.voyage()
                      .has_the_same_identity_as(handling_event.voyage()))
                    return true;
            }
            return false;
        }

        if (handling_event.type() == HandlingEventType.CLAIM)
        {
            //Check that the last leg's destination
            //is from the event's location
            var last_leg = lastLeg();
            return last_leg.unload_location()
                           .has_the_same_identity_as(
                              handling_event.location());
        }

        //HandlingEvent.Type.CUSTOMS;
        return true;
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns>The initial departure location.</returns>
    public ILocation initial_departure_location()
    {
        if (underlying_leg_collection.Count == 0)
            return LocationImplementationExample.UNKNOWN;
        return underlying_leg_collection[0].load_location();
    }

    /// <summary>
    ///
    /// </summary>
    /// <returns>The final arrival location.</returns>
    public ILocation final_arrival_location()
    {
        if (underlying_leg_collection.Count == 0)
```

```
                return LocationImplementationExample.UNKNOWN;
            return lastLeg().unload_location();
        }

        /// <summary>
        ///
        /// </summary>
        /// <returns>Date when cargo arrives at final destination.</returns>
        public IDate final_arrival_date()
        {
            ILeg lastLeg = this.lastLeg();

            if (lastLeg == null)
                return END_OF_DAYS;
            return lastLeg.unload_time();
        }

        /// <summary>
        ///
        /// </summary>
        /// <returns>The last leg on the itinerary.</returns>
        ILeg lastLeg()
        {
            if (underlying_leg_collection.Count == 0)
                return null;
            return underlying_leg_collection
                      [underlying_leg_collection.Count - 1];
        }

        /// <summary>
        ///
        /// </summary>
        /// <param name="the_other_itinerary">Itinerary to compare.</param>
        /// <returns><code>true</code> if the legs in this
        /// and the other itinerary are all equal.</returns>
        public bool has_the_same_value_as(IItinerary the_other_itinerary)
        {
            return the_other_itinerary != null &&
                underlying_leg_collection.Equals(the_other_itinerary.legs());
        }

        public override int GetHashCode()
        {
            return underlying_leg_collection.GetHashCode();
        }
    }
}
```

3.15.2 La clase post-BDD

Ésta sería la clase tras la aplicación de todas las **BDD_Specs** y limpiezas varias que hemos ido detallando exhaustivamente en este capítulo. Eso sí, sin el `namespace` ya que nuestra habitual práctica de nombrado de variables, métodos y demás, hace muy difícil la lectura del código limitados por los margenes de un documento de estas características:

```
/// <summary>
/// An itinerary.
/// Originally, a direct port of the java example in order to simulate
/// legacy code to support the argument presented in the Documentation.
/// Currently, the refactored version driven by the argument
/// presented in the Documentation.
/// </summary>
public class Itinerary : IItinerary
{
   readonly IList<ILeg> underlying_leg_collection = new List<ILeg>();

   public Itinerary(IList<ILeg> the_associated_leg_collection)
   {
      if (the_associated_leg_collection.Count == 0)
         throw new ArgumentNullException(
                     "the_associated_leg_collection",
                     "The injected leg collection cannot be empty.");

      ILeg a_null_leg = null;
      if (the_associated_leg_collection.Contains(a_null_leg))
         throw new NoNullAllowedException(
                     "The injected leg collection
                      cannot contain null Leg elements.");

      this.underlying_leg_collection = the_associated_leg_collection;
   }

   public IList<ILeg> associated_legs()
   {
      // TODO: Return an IEnumerable.
      return this.underlying_leg_collection;
   }

   public bool was_expecting(IHandlingEvent the_handling_event)
   {
      if (the_handling_event.type() == HandlingEventType.RECEIVE)
         return the_initial_leg()
                   .load_location()
                   .has_the_same_identity_as(the_handling_event.location());

      if (the_handling_event.type() == HandlingEventType.LOAD)
```

```
            return underlying_leg_collection
                       .Any(leg => leg
                       .load_location()
                       .has_the_same_identity_as(the_handling_event.location()) &&
                     leg.voyage()
                       .has_the_same_identity_as(the_handling_event.voyage()));

         if (the_handling_event.type() == HandlingEventType.UNLOAD)
            return underlying_leg_collection
                       .Any(leg => leg
                       .unload_location()
                       .has_the_same_identity_as(the_handling_event.location()) &&
                     leg.voyage()
                       .has_the_same_identity_as(the_handling_event.voyage()));

         if (the_handling_event.type() == HandlingEventType.CLAIM)
            return the_last_leg()
                       .unload_location()
                       .has_the_same_identity_as(the_handling_event.location());

         if (the_handling_event.type() == HandlingEventType.CUSTOMS)
            return true;

         return false;
      }

      public ILocation initial_departure_load_location()
      {
         return the_initial_leg().load_location();
      }

      ILeg the_initial_leg()
      {
         return underlying_leg_collection[0];
      }

      public ILocation final_arrival_unload_location()
      {
         return the_last_leg().unload_location();
      }

      ILeg the_last_leg()
      {
         return underlying_leg_collection[underlying_leg_collection.Count - 1];
      }

      public IDate final_arrival_date()
      {
         return the_last_leg().unload_time();
      }

      public bool has_the_same_value_as(IItinerary the_other_itinerary)
```

```
   {
      return the_other_itinerary != null &&
             underlying_leg_collection
                .Equals(the_other_itinerary.associated_legs());
   }

   public override int GetHashCode()
   {
      return underlying_leg_collection.GetHashCode();
   }
}
```

Capítulo 4: Value Object RouteStatus

4.1 Introducción

4.1.1 Los enums y el BDD

Ya hemos comentado con anterioridad que no nos gustaba demasiado la idea de usar objetos de tipo `enum`, siempre y cuando pudiésemos usar clases.

También sugerimos una solución para este problema, y dejamos la puerta abierta a algún tipo de mejora que nos permitiese diseñar este tipo de objetos usando **BDD**, aunque tengamos que tener presente en todo momento, las particularidades asociadas a los mismos.

Pues esa es precisamente la tarea a la que se enfrentará este capítulo.

4.1.2 Breve Análisis

A través del desarrollo del **DDD_Value_Object** `RouteStatus`, intentaremos utilizar todo lo aprendido hasta ahora.

La idea es tratar a este **DDD_Value_Object** de la misma forma que trataríamos a cualquier otro **DDD_Value_Object**, a saber:

- Implementar la interface `IValueObject<T>`.
- Implementar el método `has_the_same_value_as()`.

Para ello, debemos tener en cuenta que los requisitos nos indican que `RouteStatus` puede tomar tres valores:

- `NOT_ROUTED`.
- `ROUTED`.
- `MISROUTED`.

Valores utilizados únicamente por el **DDD_Value_Object** `IDelivery`.

> **NOTA**: Todavía no hemos desarrollado su implementación, `Delivery`.

Ésta es nuestra idea inicial.

Con esta información deberíamos tener suficiente para empezar con el código.

4.2 La clase base concern_for_route_status

Como siempre empezamos con la **clase base abstracta** que nos permitirá canalizar el uso de `Observes`.

4.2.1 Observes<T> en detalle

Dentro de:

```
Machine.Specifications.DevelopWithPassion.Rhino
```

podemos encontrarnos, entre otros, con los siguientes métodos:

```
public class Observes<ClassUnderTest> :
             Observes<ClassUnderTest, ClassUnderTest>
```

Que no hemos usado hasta ahora, ya que siempre nos hemos decantado por la versión que incluye no solo la `ClassUnderTest`, sino también el `Contract`:

```
public class Observes<Contract, ClassUnderTest> :
             InstanceObservations<Contract, ClassUnderTest,
                                  RhinoMocksMockFactory>
             where ClassUnderTest : Contract
```

Y que obliga a que el `Contract` esté implementado por la `ClassUnderTest`.

Esto es lo que nos encontramos cuando vemos detrás de la cortina del número de magia que parece ser SUT (System Under Test).

4.2.2 concern_for_route_status

Vamos con nuestra tarea:

```
namespace dddsample.specs.domain.model.cargo.aggregate
{
    public abstract class concern_for_route_status :
                          Observes<IRouteStatus, RouteStatus> {}
}
```

Que nos obliga a definir tanto el `Contract` como la `ClassUnderTest`:

```
public class RouteStatus : IRouteStatus {}
public interface IRouteStatus {}
```

Y aquí vamos a tomar nuestra primera decisión en cuanto al diseño de `IRouteStatus`, y es que, pretendemos tratar a `IRouteStatus` como un ciudadano de primera clase de la Villa **DDD_Value_Object**.

Así que, para poder decir con propiedad aquello de **DDD_Value_Object** `IRouteStatus`:

```
public interface IRouteStatus : IValueObject<IRouteStatus> {}
```

Y esto nos obliga a implementar el método `has_the_same_value_as()` en `RouteStatus`:

```
public class RouteStatus : IRouteStatus
{
    public bool has_the_same_value_as(IRouteStatus the_other_value_object)
    {
        throw new NotImplementedException();
    }
}
```

Ya tenemos todos los elementos y obligaciones necesarios para poder empezar a desarrollar nuestras **BDD_Specs**.

4.3 RouteStatusSpecs - When asked about the misrouted value

Ésta sería la primera **BDD_Spec**, que nos debería permitir intuir por donde vamos a movernos:

```
When asked about the misrouted value
   It should return the misrouted display name.
```

Por lo tanto, parece que cada uno de los valores que puede tomar `RouteStatus`, va a poder devolvernos cual es su valor.

4.3.1 Primer intento

4.3.1.1 Aclarando los términos Context / Specification y Assert en AAA

Vamos con el `Context / Specification` en código.

Primero el `Context`:

```
public class when_asked_about_the_misrouted_value : concern_for_route_status
```

La clase que contiene la **BDD_Spec**, es la responsable del `Context`.

Es importante entender esto y más importante todavía, evitar la confusión con el bloque `ESTABLISH` del que muchas veces decimos "**establece el contexto**".

Si bien es cierto que el bloque `ESTABLISH` "**establece el contexto**", no debemos perder de vista que cuando hablamos de `Context / Specification`, el `Context` viene **representado por** el nombre de la clase y **se establece** con el bloque `ESTABLISH`.

Ahora, la única `Specification` de nuestra **BDD_Spec**:

```
It should_return_the_misrouted_display_name = () =>
   result.ShouldEqual("MISROUTED");

static string result;
```

Nos hemos referido muchas veces a las `Specifications` como "**aserciónes**", ya que dentro del esquema AAA (Arrange-Act-Assert), se ocupan precisamente de la parte de las aserciones (Assert).

Sin embargo, ya hemos visto que esta correspondencia no siempre es exacta, ya que, aunque no es una práctica demasiado común (y si empieza a ser demasiado común, deberíamos empezar a plantearnos si no estaremos tomando decisiones de diseño incorrectas), una `Specification` puede estar formada por varias aserciones.

Sobre la aserción en si, ya podemos concluir que ese:

> *(...) display name.*

del que hablábamos va a ser un `string`.

4.3.1.2 Act

Vamos con el bloque `BECAUSE` (Act):

```
Because of = () => sut.display_name();
```

Nueva decisión de diseño.

Hemos decidido que para acceder al valor:

> *(...) display name.*

proporcionaremos un método llamado, ¡oh, sorpresa!, `display_name()`.

Por lo tanto para eliminar ese código rojo y facilitar la compilación del código, debemos añadir ese método al `Contract IRouteStatus`:

```
public interface IRouteStatus : IValueObject<IRouteStatus>
{
```

```
    string display_name();
}
```

Con lo que necesitaremos implementarlo en la `ClassUnderTest`:

```
public class RouteStatus : IRouteStatus
{
    public bool has_the_same_value_as(IRouteStatus the_other_value_object)
    {
        throw new NotImplementedException();
    }

    public string display_name()
    {
        throw new NotImplementedException();
    }
}
```

4.3.1.3 Arrange muestra el error

Podemos continuar con el bloque `ESTABLISH` (Arrange), donde establecemos el contexto, y aquí es donde nos encontramos con el problema.

Nuestra intención es tener un constructor privado en `RouteStatus`. Así que por aquí no vamos bien.

4.3.2 Segundo intento, prescindiendo de las extensiones develowithpassion

4.3.2.1 La BBD_Spec en código

Otro forma de abordar esto, sería prescindir de las extensiones `developwithpassion` y simplemente implementar nuestra **BDD_Spec** de la siguiente forma:

```
public class when_asked_about_the_misrouted_value_v2
{
    Because of = () => result = RouteStatus.MISROUTED.display_name();

    It should_return_the_misrouted_display_name = () =>
       result.ShouldEqual("MISROUTED");

    static string result;
```

```
}
```

4.3.2.2 Nada cambia en el Context / Specification

Si nos fijamos, nuestro `Context / Specification` **no** ha cambiado, tenemos la misma **BDD_Spec**:

```
When asked about the misrouted value
   It should return the misrouted display name.
```

Solo que ahora seguimos una estrategia diferente, que nos va a llevar a un diseño mejor (todo lo que hemos hecho, al final, viene motivado por el diseño o condiciona al mismo).

4.3.2.3 Act

Prestemos atención ahora al bloque `BECAUSE` y en especial al código rojo:

```
Because of = () => result = RouteStatus.MISROUTED.display_name();
```

Claramente estamos poniendo de manifiesto que:

- Queremos usar la clase `RouteStatus` sin instanciar.
- Necesitamos un campo llamado `MISROUTED`.
- Ese campo, sí que tiene que ser una instancia, ya que podemos invocar el método `display_name()`.

Por lo tanto, nos vamos dirigiendo poco a poco hacia el diseño que tenemos en mente.

De hecho, para empezar, ya hemos salvado el problema del constructor privado. Al no necesitar un constructor como clientes, podemos usar el constructor privado internamente.

Vamos paso a paso y sin hacer trampas.

Lo primero, como siempre, arreglar el código rojo para que al menos compile.

Con esto conseguimos el objetivo:

```
public class RouteStatus : IRouteStatus
{
    public static readonly IRouteStatus MISROUTED =
                                            new RouteStatus("MISROUTED");


    public bool has_the_same_value_as(IRouteStatus the_other_value_object)
    {
        throw new NotImplementedException();
    }

    public string display_name()
    {
        throw new NotImplementedException();
    }
}
```

Es decir, ahora, en nuestra **BDD_Spec**, tenemos que:

```
Because of = () => result = RouteStatus.MISROUTED.display_name();
```

Nada de rojo. Todo correcto.

Pero para corregir el bloque `BECAUSE`, nos hemos metido en otro problema:

```
public class RouteStatus : IRouteStatus
{
    public static readonly IRouteStatus MISROUTED =
                                            new RouteStatus("MISROUTED");

    (....)
}
```

Fijémonos en ese código rojo que aparece ahora y que implica que el código no compila.

> **NOTA**: En el código real, en este caso, no aparece de color rojo, sino con subrayado ondulado rojo, pero la única forma que teníamos de incluirlo en

> este documento era con una captura de pantalla, que decidimos descartar. Por esa razón manipulamos el color y hemos pasado el rojo del subrayado al propio código.

La razón que se oculta tras él, es que no existe ningún constructor para `RouteStatus` que admita un argumento de tipo `string`.

Por lo tanto para poder eliminar ese nuevo rojo, debemos crear el constructor (y ahora debería quedar claro que ésto nos lleva justo en la dirección que queremos, el constructor privado para uso interno):

```
public class RouteStatus : IRouteStatus
{
    public static readonly IRouteStatus MISROUTED =
                                            new RouteStatus("MISROUTED");


    RouteStatus(string the_injected_value)
    {
        throw new NotImplementedException();
    }

    public bool has_the_same_value_as(IRouteStatus the_other_value_object)
    {
        throw new NotImplementedException();
    }

    public string display_name()
    {
        throw new NotImplementedException();
    }
}
```

Fantástico. Ya compila.

Y aun por encima hemos llegado a donde queríamos.

4.3.2.4 RED-GREEN

Seguimos con nuestro proceso de siempre.

Necesitamos correr la **BDD_Spec** para verla fallar por los motivos adecuados:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about the misrouted value v2
» should return the misrouted display name (FAIL)

Test 'should return the misrouted display name' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.TypeInitializationException: The type initializer for
'dddsample.specs.domain.model.cargo.aggregate.RouteStatus' threw an
exception. ---> System.NotImplementedException: The method or operation is
not implemented.

(....)


0 passed, 1 failed, 0 skipped, took 0,62 seconds (Machine.Specifications
0.3.0).
```

Por el momento todo va según lo esperado.

Vamos a hacer que pase el test.

Para ello necesitamos que el valor **inyectado** a través del constructor de `RouteStatus`, se almacene internamente, y posteriormente, sea devuelto a través del método `display_name()`.

Es decir:

```
public class RouteStatus : IRouteStatus
{
    public static readonly IRouteStatus MISROUTED =
                                          new RouteStatus("MISROUTED");

    readonly string underlying_display_name;

    RouteStatus(string the_injected_value)
    {
        underlying_display_name = the_injected_value;;
    }

    public string display_name()
    {
        return underlying_display_name;
```

```
    }

    public bool has_the_same_value_as(IRouteStatus the_other_value_object)
    {
        throw new NotImplementedException();
    }
}
```

Probamos de nuevo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about the misrouted value v2
» should return the misrouted display name


1 passed, 0 failed, 0 skipped, took 0,44 seconds (Machine.Specifications
0.3.0).
```

Lo hemos conseguido. Ahora solo necesitamos borrar lo de la "**v2**" en:

```
public class when_asked_about_the_misrouted_value_v2
```

ya que ésta es la versión que queremos.

Vamos a repetir el proceso para el siguiente valor, pero sin repetir los mismos errores.

4.4 RouteStatusSpecs - When asked about the routed value

No debería suponer ninguna sorpresa saber ésta sería la **BDD_Spec**:

```
When asked about the routed value
   It should return the routed display name.
```

4.4.1 Definición del Context / Specification en código

Vamos con la clase y el bloque `IT`:

```
public class when_asked_about_the_routed_value
{
    It should_return_the_routed_display_name = () =>
       result.ShouldEqual("ROUTED");

    static string result;
}
```

Igual que en el caso anterior. Solo que ahora esperamos recibir `ROUTED` en lugar de `MISROUTED`.

4.4.2 Act

El bloque `BECAUSE` para completar la **BDD_Spec** sería éste:

```
Because of = () => result = RouteStatus.ROUTED.display_name();
```

Como siempre, tenemos que solucionar el código rojo para que compile. Pero esta vez necesitamos menos que en la **BDD_Spec** anterior, puesto que toda la infraestructura del constructor privado, etc, ya lo tenemos.

Así que basta con añadir esta linea a la clase `RouteStatus`:

```
public class RouteStatus : IRouteStatus
{
    public static readonly IRouteStatus ROUTED = new RouteStatus("ROUTED");
    public static readonly IRouteStatus MISROUTED =
```

```
                                        new RouteStatus("MISROUTED");

    (....)
}
```

4.4.3 RED-GREEN

Ya podemos ejecutar la **BDD_Spec**, esperando nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about the routed value
» should return the routed display name


1 passed, 0 failed, 0 skipped, took 0,45 seconds (Machine.Specifications
0.3.0).
```

Pues no hay fallo...

Tenemos bastante claro que no es un falso positivo, pero necesitamos alguna prueba un poco más contundente que nuestra intuición. Así que, vamos a hacer fallar el test.

Para ello, cambiamos esto que teníamos en nuestro bloque `IT`:

```
It should_return_the_routed_display_name = () => result.ShouldEqual("ROUTED");
```

por esto otro:

```
It should_return_the_routed_display_name = () => result.ShouldEqual("Hello");
```

y ejecutamos la **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked about the routed value
» should return the routed display name (FAIL)

Test 'should return the routed display name' failed:
       Machine.Specifications.SpecificationException: Should equal "Hello"
but is "ROUTED"
       at Machine.Specifications.ShouldExtensionMethods.ShouldEqual[T](T
```

```
   actual, T expected)
         domain\model\cargo.aggregate\RouteStatusSpecs.cs(23,0): at
   dddsample.specs.domain.model.cargo.aggregate.when_asked_about_the_routed_v
   alue.<.ctor>b__1()
         at
   Machine.Specifications.Model.Specification.InvokeSpecificationField()
         at Machine.Specifications.Model.Specification.Verify()

   0 passed, 1 failed, 0 skipped, took 0,61 seconds (Machine.Specifications
   0.3.0).
```

Justo lo que esperábamos:

```
   Should equal "Hello" but is "ROUTED"
```

Por lo tanto nuestro código está bien.

Así que podemos recuperar el estado original de nuestra **BDD_Spec**:

```
public class when_asked_about_the_routed_value
{
    Because of = () => result = RouteStatus.ROUTED.display_name();

    It should_return_the_routed_display_name = () =>
       result.ShouldEqual("ROUTED");

    static string result;
}
```

Al volver a ejecutarla:

```
   1 passed, 0 failed, 0 skipped, took 0,65 seconds (Machine.Specifications
   0.3.0).
```

Mejor imposible.

4.5 RouteStatusSpecs – When asked about the not routed value

4.5.1 Similitud con la BDD_Spec anterior

El proceso para añadir `NOT_ROUTED` es exactamente el mismo que hemos utilizado en la **BDD_Spec** anterior para añadir `ROUTED`, por lo tanto no nos vamos a repetir. El código que lo implementa puede ser consultado en el código fuente.

4.5.2 Definición de la BDD_Spec

Dejamos la definición de la **BDD_Spec**:

```
When asked about the not routed value
   It should return the not routed display name.
```

4.6 RouteStatusSpecs - Implementando el comportamiento IValueObject<T>

4.6.1 Implementación del Happy Day Scenario

Vamos ahora con esta **BDD_Spec**:

```
When comparing two equal route status
   It should confirm that both have the same value.
```

Con ella esperamos empezar a definir el comportamiento que debe tener la igualdad cuando hablamos de cualquier **DDD_Value_Object**.

La implementación es muy sencilla, así que la mostramos ya completa:

```
public class when_comparing_two_equal_route_status
{
    Because of = () =>
       result = RouteStatus.MISROUTED
                           .has_the_same_value_as(RouteStatus.MISROUTED);

    It should_confirm_that_both_have_the_same_value = () =>
       result.ShouldBeTrue();

    static bool result;
}
```

Podemos destacar que en el bloque `BECAUSE` hemos escogido el valor `MISROUTED` para probar la igualdad, pero podríamos haber escogido cualquiera de los otros dos:

- `ROUTED`.
- `NOT_ROUTED`.

Ejecutamos la **BDD_Spec** esperando nuestro fallo por no estar implementado:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two equal route status
» should confirm that both have the same value (FAIL)
```

```
Test 'should confirm that both have the same value' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)


0 passed, 1 failed, 0 skipped, took 0,70 seconds (Machine.Specifications
0.3.0).
```

Tal y como deseábamos.

Ahora tenemos que implementar la funcionalidad de `has_the_same_value_as()`.

Lo único que necesitamos hacer es comparar los valores de ambos **display_names**, tal que así:

```
public bool has_the_same_value_as(IRouteStatus the_other_route_status)
{
    return underlying_display_name
              .Equals(the_other_route_status.display_name());
}
```

Con eso debería bastar.

De todas formas es interesante ver como usamos `Equals()` en lugar de nuestros `has_the_same_value_as()` o `has_the_same_identity_as()`.

La razón es que estamos tratando con `strings` y no con un tipo definido por nosotros:

```
readonly string underlying_display_name;
```

En caso de haber sido un tipo definido por nosotros, con total seguridad hubiésemos utilizado `has_the_same_value_as()` o `has_the_same_identity_as()` (el que estuviese disponible dependiendo del contexto).

Comprobamos:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two equal route status
» should confirm that both have the same value


1 passed, 0 failed, 0 skipped, took 0,70 seconds (Machine.Specifications
0.3.0).
```

4.6.2 Implementación del escenario complementario

Esta **BDD_Spec** sería la complementaria a la anterior, el Happy Day Scenario.

Aquí va la definición:

```
When comparing two different route status
   It should confirm they have different value.
```

Para la implementación solo necesitamos escoger dos de los valores que puede tomar `RouteStatus`.

En este caso nos hemos decantado por los que no usamos en la **BDD_Spec** anterior.

Por lo tanto, el código es el siguiente:

```
public class when_comparing_two_different_route_status
{
    Because of = () =>
       result = RouteStatus.ROUTED
                                .has_the_same_value_as(RouteStatus.NOT_ROUTED);

    It should_confirm_they_have_different_value = () => result.ShouldBeFalse();

    static bool result;
}
```

Y no debería sorprendernos que pase el test directamente:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two different route status
» should confirm they have different value
```

```
1 passed, 0 failed, 0 skipped, took 0,41 seconds (Machine.Specifications
0.3.0).
```

Ya sabemos como hacer para comprobar que éste no es un falso positivo. Podríamos hacerlo fallar primero, asertando justo lo opuesto de lo que esperamos. Pero no vamos a detallarlo.

4.7 Recapitulación

Con las tres primeras **BDD_Specs**, lo que hicimos fue probar de manera indirecta que existía el `RouteStatus` concreto que estábamos testando:

- `NOT_ROUTED`.
- `ROUTED`.
- `MISROUTED`.

Al tener tres valores distintos para `RouteStatus`, hicimos, por lo tanto, tres **BDD_Specs** al respecto.

Cuando nos encontremos en una situación en la que no podamos testar lo que nos gustaría de forma directa, primero debemos evaluar si no estamos metiendo la pata con el diseño, y si, efectivamente, estamos diseñando correctamente, entonces debemos buscar alguna forma indirecta para testar lo que queremos. Aun así, la mayor parte de las veces, suele tratarse de un error de diseño.

De forma directa testamos el `display_name`, como es evidente.

Con las dos últimas **BDD_Specs**, aseguramos que nuestros `RouteStatus` se pueden considerar propiamente **DDD_Value_Objects**.

Por si acaso se nos quedó algo por el camino, aquí tenemos la clase completa:

```
namespace dddsample.domain.model.cargo.aggregate
{
    public class RouteStatus : IRouteStatus
    {
        public static readonly IRouteStatus ROUTED =
                                            new RouteStatus("ROUTED");
        public static readonly IRouteStatus NOT_ROUTED =
                                            new RouteStatus("NOT_ROUTED");
        public static readonly IRouteStatus MISROUTED =
                                            new RouteStatus("MISROUTED");

        readonly string underlying_display_name;

        RouteStatus(string the_injected_value)
```

```
        {
            underlying_display_name = the_injected_value;
        }

        public string display_name()
        {
            return underlying_display_name;
        }

        public bool has_the_same_value_as(IRouteStatus the_other_route_status)
        {
            return underlying_display_name
                       .Equals(the_other_route_status.display_name());
        }
    }
}
```

4.8 Una última mejora a posteriori

4.8.1 Contextualizando la situación

Al vernos en la obligación de desarrollar el **DDD_Value_Object** `TransportStatus`, objeto equivalente al **DDD_Value_Object** `RouteStatus`, nos hemos dado cuenta de que hemos empezado a repetir código.

> **NOTA**: El desarrollo del **DDD_Value_Object** `TransportStatus` es análogo al del **DDD_Value_Object** `RouteStatus`. Por esa razón no ha sido documentado. Sin embargo sí que se encuentra en el código fuente.

Comparemos `ITransportStatus` con `IRouteStatus`:

```
public interface ITransportStatus : IValueObject<ITransportStatus>
{
    string display_name();
}
```

```
public interface IRouteStatus : IValueObject<IRouteStatus>
{
    string display_name();
}
```

La semejanza no solo la encontramos en el nombre sino también en la intención, ya que `display_name()` representa exactamente el mismo concepto en ambos casos, sin ningún matiz que los diferencie.

4.8.2 Nace IEnumeration

Por lo tanto, hemos decidido que el método `display_name()` debería tener su propia interface llamada `IEnumeration`.

Hay que tener en cuenta que es precisamente el concepto de enumeración el que estamos tratando de simular con clases e interfaces como alternativa al uso de `enums`.

Por lo tanto, ésta será la definición y la cadena de implementaciones entre las distintas interfaces.

Empezamos por `IEnumeration`:

```
namespace dddsample.domain.shared
{
    public interface IEnumeration
    {
        string display_name();
    }
}
```

En este caso cobra una especial relevancia el `namespace` al que pertenece, ya que, implica que consideramos que se trata de una interface que puede ser implementada por cualquier **DDD_Aggregate**.

De ahí que la situemos en:

```
dddample.domain.shared
```

Seguimos con `IRouteStatus` e `ITransportStatus`:

```
public interface ITransportStatus : IValueObject<ITransportStatus>,
                                    IEnumeration {}

public interface IRouteStatus : IValueObject<IRouteStatus>,
                                IEnumeration {}
```

Es importante señalar como tanto `ITransportStatus` como `IRouteStatus` se encargan de señalar que sus implementaciones (en nuestro caso `TransportStatus` y `RouteStatus`) incluirán:

- El comportamiento asociado a `IEnumeration`.
- El comportamiento asociado a ser un **DDD_Value_Object**.

Recordemos que ambas pertenecen a:

```
namespace dddsample.domain.model.cargo.aggregate
```

Finalmente vemos como las clases que implementan a ambas, no cambian:

```
public class TransportStatus : ITransportStatus { (....) }

public class RouteStatus : IRouteStatus { (....) }
```

Si ejecutamos la suite de **BDD_Specs**, observaremos que no hemos roto nada:

```
217 passed, 0 failed, 0 skipped, took 2,70 seconds (Machine.Specifications
0.3.0).
```

Hay que destacar que este tipo de cambios menores en pos de un mejor sistema, se producen de forma habitual.

Al final, tal y como hemos recalcado hasta la saciedad, todo es **DISEÑO**.

Capítulo 5: Refactorizando RouteSpecificationFactory

5.1 Introducción

A medida que íbamos desarrollando las **BDD_Specs** que se hallan en `LegSpecs` y que definen el comportamiento que debería tener el **DDD_Value_Object** `Leg`, se ha puesto de manifiesto la necesidad de contar también con una **DDD_Factory** `LegFactory` ya que, nuevamente debemos comprobar en el momento de la creación del objeto que éste cumple una serie de invariantes, de forma que si alguna de ellas es violada se lance una excepción.

Ya tenemos una **DDD_Factory** que se encarga de crear **DDD_Value_Objects** `IRouteSpecification` y como ya prevemos que este patrón (nos referimos a la necesidad de comprobar las invariantes en el momento de la construcción) se va a seguir repitiendo, parece un buen momento para refactorizar la **DDD_Factory** existente de forma que se adecue a nuestras nuevas necesidades.

5.1.1 CargoAggregateFactory

La idea consiste en contar con una **DDD_Factory** `CargoAggregateFactory` que pueda crear cualquier instancia abstracta (es decir, que devuelva interfaces) relativa al **DDD_Aggregate** `Cargo`.

Al hacerlo de esta forma, violamos el **OCP**, ya que estaríamos cerrados a la extensión y abiertos a la modificación (justo al contrario de lo que te dice el **OCP**), sin embargo, en este caso nos parece la mejor solución ya que para cumplir con el **OCP**, necesitaríamos crear una **DDD_Factory** por cada objeto. Es decir:

- `LegFactory`.
- `RouteSpecificationFactory`.
- `ItineraryFactory`.
- ...

Y en nuestra humilde opinión, complicaría en exceso el diseño (**Complejidad Innecesaria o Needless Complexity**)

Aun así, al ser conscientes de que estamos violando el **OCP**, podremos mantenernos alerta durante el proceso de codificación de esta **DDD_Factory**, por si acaso encontramos una solución mejor.

> **NOTA**: Si en estos momentos estamos pensando en revisar este documento en busca de las `LegSpecs`, es mejor que no lo hagamos, ya que no se encuentran en el mismo.
>
> La razón es que no hay nada en ellas que sea digno de ser comentado, más allá de como al necesitar un proceso de construcción más complejo a la hora de crear un **DDD_Value_Object** `Leg` nos ha influido en la decisión de establecer una **DDD_Factory**, que es precisamente de lo que trata este capítulo.
>
> No hay nada en `LegSpecs` que no hayamos tratado con anterioridad en alguna de las `xxxSpecs` que sí hemos documentado profusamente.
>
> De todas formas, debemos recordar que sí que podemos consultar tanto `LegSpecs` como, `ILeg` o `Leg` en el código fuente.
>
> Es más, lo recomendamos encarecidamente, antes de continuar con esta sección.

Vamos por tanto con la refactorización de `RouteSpecificationFactory` en `CargoAggregateFactory`, primeramente.

5.2 Refactorizando RouteSpecificationFactory en CargoAggregateFactory

5.2.1 RouteSpecificationFactory

Recordemos por un instante qué es lo que tenemos hasta el momento.

```
using System;
using dddsample.domain.model.location.aggregate;

namespace dddsample.domain.model.cargo.aggregate
{
    public class RouteSpecificationFactory
    {
        public IRouteSpecification create_route_specification_using(
                                    ILocation the_origin_location,
                                    ILocation the_destination_location,
                                    IDate the_arrival_deadline)
        {
            (....)

            return new RouteSpecification(the_origin_location,
                                          the_destination_location,
                                          the_arrival_deadline);
        }
    }
}
```

Ésta sería la parte más significativa de la **DDD_Factory** `RouteSpecificationFactory` donde "**(....)**" sustituye a las diferentes comprobaciones de las invariantes (que no son relevantes para la refactorización que vamos a acometer).

También contamos con las siguientes **BDD_Specs**, relativas a su comportamiento, mostradas aquí como **log** de salida de **TestDriven.NET**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when attempting to inject a null origin location into the route
specification factory
» should throw a null argument exception
» should throw an invariant violated exception message

when attempting to inject a null destination location into the route
specification factory
```

```
» should throw a null argument exception
» should throw an invariant violated exception message

when attempting to inject a null arrival deadline into the route
specification factory
» should throw a null argument exception
» should throw an invariant violated exception message

when attempting to inject the same origin and destination locations into
the route specification factory
» should throw an argument exception
» should throw an invariant violated exception message


8 passed, 0 failed, 0 skipped, took 0,89 seconds (Machine.Specifications
0.3.0).
```

5.2.2 El camino hacia CargoAggregateFactory

Vamos a empezar con el cambio de nombre, ya que, habíamos quedado en que queremos que se llamase `CargoAggregateFactory`, en vez de `RouteSpecificationFactory`.

5.2.2.1 Haciendo uso de ReSharper

Para ello usaremos **ReSharper**:

Nos situamos encima de `RouteSpecificationFactory`:

```
public class RouteSpecificationFactory
```

y pulsamos:

```
F2
```

de forma que nos permita refactorizar el nombre.

Cambiamos el nombre a `CargoAggregateFactory` y nos aseguramos de que la opción:

```
Synchronize files names accordingly to changes.
```

esté marcada y pulsamos:

```
NEXT
```

En nuestro caso, incluso encuentra los siguientes símbolos como posibles para proceder con el cambio de nombre:

```
the_route_specification_factory = new RouteSpecificationFactory();
```

en `RouteSpecificationSpecs`.

No queremos que cambie `the_route_specification_factory` en este contexto por `the_cargo_aggregate_factory`, por lo tanto, desmarcamos la opción y pulsamos:

```
NEXT
```

Sorprendentemente, no todo ha ido bien.

Si intentamos ejecutar las **BDD_Specs** ahora obtenemos el siguiente error:

```
Error 1     The type or namespace name 'RouteSpecificationFactory' could
not be found (are you missing a using directive or an assembly reference?)

c:\dev\dddsample\source\dddsample.specs\domain\model\cargo.aggregate\Route
SpecificationSpecs.cs     26     26     dddsample.specs
```

Por lo tanto, no ha cambiado todo lo que debía cambiar.

5.2.2.2 Arreglando el error de ReSharper

Vamos a `RouteSpecificationFactory` y nos encontramos con:

```
public abstract class concern_for_route_specification :
                      Observes<IRouteSpecification, RouteSpecification>
{
    Establish context = () =>
    {
        the_origin_location = an<ILocation>();
        the_destination_location = an<ILocation>();
        the_arrival_deadline = an<IDate>();
        the_route_specification_factory = new RouteSpecificationFactory();
```

```
        create_sut_using(() =>
           the_route_specification_factory
              .create_route_specification_using(the_origin_location,
                                                the_destination_location,
                                                the_arrival_deadline));
    };

    protected static ILocation the_origin_location;
    protected static ILocation the_destination_location;
    protected static IDate the_arrival_deadline;
    protected static RouteSpecificationFactory the_route_specification_factory;
}
```

Para solucionar el código rojo, lo único que tenemos que hacer es cambiar manualmente `RouteSpecificationFactory` por `CargoAggregateFactory`, y volvemos a ejecutar las **BDD_Specs**:

```
198 passed, 0 failed, 0 skipped, took 2,51 seconds (Machine.Specifications
0.3.0).
```

Justo lo que deseábamos

Ya hemos completado nuestro primer objetivo.

5.2.2.3 Los nombres de los ficheros

Ahora necesitamos inspeccionar que los nombres de los ficheros se adecuan a lo que queremos y ahí nos encontramos con que tenemos:

- `CargoAggregateFactory.cs`
- `RouteSpecificationFactorySpecs.cs`

Evidentemente el primero está bien, pero el segundo no nos gusta tanto.

Ahora buscamos tener un `CargoAggregateFactorySpecs.cs` que englobe todas las **BDD_Specs** relativas a cualquier creación de un objeto por parte de `CargoAggregateFactory`.

Por lo tanto procedemos a cambiar el nombre y ejecutamos la suite de **BDD_Specs**:

```
198 passed, 0 failed, 0 skipped, took 2,27 seconds (Machine.Specifications
0.3.0).
```

Ahora sí que podemos pasar al siguiente punto.

5.3 Añadiendo LegFactory a CargoAggregateFactory

5.3.1 Definición formal de las BDD_Specs de LegFactory

Ya tenemos `CargoAggregateFactory`, así que necesitamos definir las **BDD_Specs** relativas a la creación de **DDD_Value_Objects** `ILeg`.

Tomando como modelo las **BDD_Specs** relativas a `RouteSpecificationFactory`, necesitamos implementar:

```
When attempting to inject a null voyage into the leg factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject a null load location into the leg factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject a null unload location into the leg factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject a null load time into the leg factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject a null unload time into the leg factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.
```

Como podemos apreciar, hay un patrón muy claro y similar en todas ellas, así que únicamente documentaremos la implementación de la primera. Recordemos que siempre podemos ir al código fuente para ver las restantes.

5.3.2 LegFactorySpecs - when attempting to inject a null voyage into the leg factory

5.3.2.1 La BDD_Spec

Ésta es la **BDD_Spec**:

```
When attempting to inject a null voyage into the leg factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.
```

5.3.2.2 Assert

Vamos con la tercera A, Assert, a través de dos bloques `IT`:

```
It should_throw_a_null_argument_exception = () =>
   exception_thrown_by_the_sut.ShouldBeAn<ArgumentNullException>();

It should_throw_an_invariant_violated_exception_message = () =>
   exception_thrown_by_the_sut
      .ShouldContainErrorMessage(
         "Invariant Violated: a valid voyage is required.");
```

La equivalencia entre la definición formal de la **BDD_Spec** y su implementación en código es completa.

5.3.2.3 Act

Vamos con la segunda A, Act, a través de un bloque `BECAUSE`:

```
Because of = () => catch_exception(() =>
   sut.create_leg_using(an_empty_voyage,
                        the_injected_load_location,
                        the_injected_unload_location,
                        the_injected_load_time,
                        the_injected_unload_time));
```

En este caso le estamos indicando que capture la excepción que se produzca al invocar el `create_leg_using()` de la **DDD_Factory**.

5.3.2.4 Solucionando el código rojo

Tenemos que solucionar el código rojo. Lo haremos en dos partes.

Primeramente atenderemos a los argumentos del método y a continuación iremos con

la **reacción en cadena** del nombre del método invocado.

La primera parte es muy sencilla, lo hemos hecho un montón de veces:

```
Because of = () => catch_exception(() =>
   sut.create_leg_using(an_empty_voyage,
                        the_injected_load_location,
                        the_injected_unload_location,
                        the_injected_load_time,
                        the_injected_unload_time));

static IVoyage an_empty_voyage;
static ILocation the_injected_load_location;
static ILocation the_injected_unload_location;
static IDate the_injected_load_time;
static IDate the_injected_unload_time;
```

5.3.2.5 Reacción en cadena

Ahora vamos con la parte del **diseño** (la **reacción en cadena**).

Lo que estamos indicando con:

```
sut.create_leg_using(....)
```

es que queremos un nuevo método en la **DDD_Factory** que nos permita crear `ILegs`.

Por lo tanto:

```
using System;
using dddsample.domain.model.location.aggregate;
using dddsample.domain.model.voyage.aggregate;

namespace dddsample.domain.model.cargo.aggregate
{
    public class CargoAggregateFactory
    {
        public IRouteSpecification create_route_specification_using(
                                    ILocation the_origin_location,
                                    ILocation the_destination_location,
                                    IDate the_arrival_deadline)
        { (....) }

        public ILeg create_leg_using(
```

```
                        IVoyage the_voyage,
                        ILocation the_load_location,
                        ILocation the_unload_location,
                        IDate the_load_time,
                        IDate the_unload_time)
        {
            throw new NotImplementedException();
        }
    }
}
```

5.3.2.6 El sentido de CargoAggregateFactory

Al haber creado `CargoAggregateFactory` estamos intentando obligar a que cualquier creación compleja (léase, por ejemplo, "**que compruebe invariantes**") de un objeto que se encuentre dentro del **DDD_Aggregate** `Cargo`, siempre pase por la **DDD_Factory**.

Una apreciación interesante es que, al contrario que con el resto de los objetos, no hemos creado una interface para `CargoAggregateFactory`. Por el momento no lo hemos necesitado a la hora de mockar nuestras dependencias. Sin embargo, hay otra razón que trataremos al final de este capítulo, que sí que nos parece relevante como para crear interfaces (nótese el plural) para obligar a `CargoAggregateFactory` a implementarlas. Como pequeño adelanto nombrar el **DDD_Intention-Revealing_Interfaces**.

5.3.2.7 Arrange

Ya tenemos las dos últimas AAA cubiertas, así que vamos con la primera A, Arrange, a través de un bloque `ESTABLISH`:

```
Establish context = () =>
{
    an_empty_voyage = null;
    the_injected_load_location = an<ILocation>();
    the_injected_unload_location = an<ILocation>();
    the_injected_load_time = an<IDate>();
    the_injected_unload_time = an<IDate>();
};
```

Nada nuevo.

Simplemente establecemos los valores iniciales de los elementos que entran en juego para que se produzca la excepción deseada.

5.3.2.8 El código completo de la BDD_Spec

Por lo tanto éste sería el código de la **BDD_Spec** resultante, con un pequeño cambio en el mensaje de error:

```
public class when_attempting_to_inject_a_null_voyage_into_the_leg_factory :
             concern_for_route_specification_factory
{
    Establish context = () =>
    {
        an_empty_voyage = null;
        the_injected_load_location = an<ILocation>();
        the_injected_unload_location = an<ILocation>();
        the_injected_load_time = an<IDate>();
        the_injected_unload_time = an<IDate>();
    };

    Because of = () => catch_exception(() =>
       sut.create_leg_using(an_empty_voyage,
                            the_injected_load_location,
                            the_injected_unload_location,
                            the_injected_load_time,
                            the_injected_unload_time));

    It should_throw_a_null_argument_exception = () =>
       exception_thrown_by_the_sut.ShouldBeAn<ArgumentNullException>();

    It should_throw_an_invariant_violated_exception_message = () =>
       exception_thrown_by_the_sut.ShouldContainErrorMessage(
          "Invariant Violated:
           a valid voyage is required in order to construct a leg.");

    static IVoyage an_empty_voyage;
    static ILocation the_injected_load_location;
    static ILocation the_injected_unload_location;
    static IDate the_injected_load_time;
    static IDate the_injected_unload_time;
}
```

Todo parece correcto.

5.3.2.9 Análisis de concern_for_route_specification_factory

Sin embargo hay algo que chirría. Se encuentra en la primera linea:

```
public class when_attempting_to_inject_a_null_voyage_into_the_leg_factory :
             concern_for_route_specification_factory
```

Ese `concern_for_route_specification_factory` es un desastre desde un punto de vista semántico, ya que, recordemos que estamos construyendo un **DDD_Value_Object** `Leg` y no un **DDD_Value_Object** `RouteSpecification`.

Sin embargo el `Observes` es el correcto:

```
public abstract class concern_for_route_specification_factory :
                      Observes<CargoAggregateFactory> { }
```

5.3.2.10 Creando concern_for_leg_factory

Por lo tanto, tenemos dos opciones:

1. Cambiar `concern_for_route_specification_factory` por `concern_for_cargo_aggregate_factory`.
2. Crear una nueva `concern_for_leg_factory`.

Con la primera opción, en nuestra opinión, perdemos expresividad (recordemos que somos conscientes de que estamos violando el **OCP**).

Si no violásemos el **OCP**, tendríamos múltiples objetos como:

- `RouteSpecificationFactory`.
- `LegFactory`.
- ...

con sus respectivos:

- `concern_for_route_specification_factory`.

- `concern_for_leg_factory`.
- ...

Por lo tanto la segunda parece la mejor opción, así que:

```
public abstract class concern_for_leg_factory : Observes<CargoAggregateFactory>
{ }

public class when_attempting_to_inject_a_null_voyage_into_the_leg_factory :
             concern_for_leg_factory { (....) }
```

5.3.2.11 RED-GREEN

Una vez resuelto este problema, deberíamos hacer fallar nuestra **BDD_Spec**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when attempting to inject a null voyage into the leg factory
» should throw a null argument exception (FAIL)
» should throw an invariant violated exception message (FAIL)

Test 'should throw a null argument exception' failed:
      Machine.Specifications.SpecificationException: Should be of type
System.ArgumentNullException but is of type System.NotImplementedException
      at
Machine.Specifications.ShouldExtensionMethods.ShouldBeOfType(Object
actual, Type expected)

(....)

Test 'should throw an invariant violated exception message' failed:
      Machine.Specifications.SpecificationException: Should contain
"Invariant Violated: a valid voyage is required in order to construct a
leg." but is "The method or operation is not implemented."

(....)

0 passed, 2 failed, 0 skipped, took 0,78 seconds (Machine.Specifications
0.3.0).
```

Es lo que esperábamos.

Para hacer que nuestra **BDD_Spec** se cumpla, basta con:

```
public ILeg create_leg_using(
               IVoyage the_voyage,
               ILocation the_load_location,
               ILocation the_unload_location,
               IDate the_load_time,
               IDate the_unload_time)
{
    if (the_voyage == null)
       throw new ArgumentNullException(
                    "the_voyage",
                    "Invariant Violated:
                     a valid voyage is required in order to construct a leg.");

    return new Leg(the_voyage,
                   the_load_location,
                   the_unload_location,
                   the_load_time,
                   the_unload_time);
}
```

Ejecutamos la **BDD_Spec** ahora:

```
------ Test started: Assembly: dddsample.specs.dll ------

when attempting to inject a null voyage into the leg factory
» should throw a null argument exception
» should throw an invariant violated exception message


2 passed, 0 failed, 0 skipped, took 0,79 seconds (Machine.Specifications
0.3.0).
```

Podemos darnos por satisfechos.

Las restantes **BDD_Specs** son similares a ésta.

5.3.3 create_leg_using() al completo

En cuanto hemos satisfecho todas las **BDD_Specs** de las que hablábamos al principio del capítulo, nos encontramos con que `create_leg_using()` tiene el siguiente aspecto:

```
public ILeg create_leg_using(
               IVoyage the_voyage,
               ILocation the_load_location,
               ILocation the_unload_location,
```

```
                IDate the_load_time,
                IDate the_unload_time)
{
    if (the_voyage == null)
       throw new ArgumentNullException(
                    "the_voyage",
                    "Invariant Violated:
                     a valid voyage is required in order to construct a leg.");

    if (the_load_location == null)
       throw new ArgumentNullException(
                    "the_load_location",
                    "Invariant Violated:
                     a valid load location is required
                     in order to construct a leg.");

    if (the_unload_location == null)
       throw new ArgumentNullException(
                    "the_unload_location",
                    "Invariant Violated:
                     a valid unload location is required
                     in order to construct a leg.");

    if (the_load_time == null)
       throw new ArgumentNullException(
                    "the_load_time",
                    "Invariant Violated:
                     a valid load time is required
                     in order to construct a leg.");

    if (the_unload_time == null)
       throw new ArgumentNullException(
                    "the_unload_time",
                    "Invariant Violated:
                     a valid unload time is required
                     in order to construct a leg.");

    return new Leg(the_voyage,
                   the_load_location,
                   the_unload_location,
                   the_load_time,
                   the_unload_time);
}
```

5.3.3.1 Necesidad de la DDD_Factory

La razón por la que mostramos el código anterior, cuando en realidad no ofrece casi ninguna novedad a lo que ya hemos visto al implementar la última **BDD_Spec**, es mostrar porqué creemos que sería imposible defender que ese código debería ir en el

constructor.

Un constructor con todo eso pide a gritos una **DDD_Factory**.

5.3.4 Usando LegFactory

Ahora que ya tenemos nuestra factoría funcionando, debemos hacer los cambios necesarios para que sea usada donde proceda.

Por lo tanto necesitamos que:

1. El constructor de `Leg` pase a ser `internal`.
2. `LegSpecs` use la **DDD_Factory** en vez del constructor para crear los **DDD_Value_Objects** `Leg`.

El primer punto es trivial:

```
internal Leg(IVoyage the_voyage,
             ILocation the_load_location,
             ILocation the_unload_location,
             IDate the_load_time,
             IDate the_unload_time)
{ (....) }
```

Pero ahora las llamadas al constructor desde `LegSpecs` no se encuentran en el ámbito correcto debido al `internal`, por lo que nuestro código no compila. Esto nos lleva al punto dos.

Ésta es la versión que tenemos ahora (la que no compila) de `concern_for_leg`:

```
public abstract class concern_for_leg : Observes<ILeg, Leg>
{
    Establish context = () =>
    {
        the_injected_voyage = an<IVoyage>();
        the_injected_load_location = an<ILocation>();
        the_injected_unload_location = an<ILocation>();
        the_injected_load_time = an<IDate>();
        the_injected_unload_time = an<IDate>();
```

```
        create_sut_using(() =>
           new Leg(the_injected_voyage,
                   the_injected_load_location,
                   the_injected_unload_location,
                   the_injected_load_time,
                   the_injected_unload_time));
    };

    protected static IVoyage the_injected_voyage;
    protected static ILocation the_injected_load_location;
    protected static ILocation the_injected_unload_location;
    protected static IDate the_injected_load_time;
    protected static IDate the_injected_unload_time;
}
```

El problema está en la llamada al constructor, que ahora debemos sustituir por la llamada a la **DDD_Factory**.

Por lo tanto basta con:

```
public abstract class concern_for_leg : Observes<ILeg, Leg>
{
    Establish context = () =>
    {
        the_injected_voyage = an<IVoyage>();
        the_injected_load_location = an<ILocation>();
        the_injected_unload_location = an<ILocation>();
        the_injected_load_time = an<IDate>();
        the_injected_unload_time = an<IDate>();
        the_leg_factory = new CargoAggregateFactory();

        create_sut_using(() =>
           the_leg_factory.create_leg_using(
                           the_injected_voyage,
                           the_injected_load_location,
                           the_injected_unload_location,
                           the_injected_load_time,
                           the_injected_unload_time));
    };

    protected static CargoAggregateFactory the_leg_factory;
    protected static IVoyage the_injected_voyage;
    protected static ILocation the_injected_load_location;
    protected static ILocation the_injected_unload_location;
    protected static IDate the_injected_load_time;
    protected static IDate the_injected_unload_time;
}
```

Pero con esto no es suficiente, ya que en algunas de las **BDD_Specs** de `LegSpecs`,

creamos otros objetos de tipo `ILeg` (para comparaciones, ...).

Por lo tanto también debemos sustituir esas llamadas (en concreto, hay cuatro) al constructor por llamadas a la factoría.

Puesto que son todas iguales, mostramos a modo de ejemplo el antes y después de una de ellas:

ANTES

```
public class when_comparing_two_legs_with_the_same_attributes : concern_for_leg
{
    Establish context = () =>
    {
      the_other_leg = new Leg(the_injected_voyage,
                              the_injected_load_location,
                              the_injected_unload_location,
                              the_injected_load_time,
                              the_injected_unload_time);
        (....)
    }
}
```

DESPUES

```
public class when_comparing_two_legs_with_the_same_attributes : concern_for_leg
{
    Establish context = () =>
    {
      the_other_leg = the_leg_factory.create_leg_using(
                                          the_injected_voyage,
                                          the_injected_load_location,
                                          the_injected_unload_location,
                                          the_injected_load_time,
                                          the_injected_unload_time);
        (....)
    }
}
```

Si ahora ejecutamos la suite de **BDD_Specs** obtenemos:

```
210 passed, 0 failed, 0 skipped, took 2,11 seconds (Machine.Specifications
0.3.0).
```

Por lo tanto podemos dar por finalizado este epígrafe.

5.3.5 Refactorizando LegFactory

5.3.5.1 Responsabilidad e intención

Hay un par de cosas que todavía no nos convencen del todo.

Las **BDD_Specs** se cumplen, pero la **DDD_Factory** `CargoAggregateFactory` podría ser más expresiva, podría reflejar mejor nuestra intención a través de **DDD_Intention-Revealing_Interfaces**.

Vamos al asunto.

¿Cual es la responsabilidad de la **DDD_Factory** `CargoAggregateFactory`?

La creación de cualquier objeto que pertenezca al **DDD_Aggregate** `Cargo` a través de su correspondiente interface. Siempre y cuando, el proceso de creación de dicho objeto sea demasiado complejo para ir en un constructor común.

¿La **DDD_Factory** `CargoAggregateFactory` expresa claramente su intención?

No. Ya que recordemos que su definición es la siguiente:

```
namespace dddsample.domain.model.cargo.aggregate
{
    public class CargoAggregateFactory { (....) }
}
```

Es demasiado oscura.

Atendiendo únicamente a su definición, no acaba de quedar clara cual es su responsabilidad.

Aquí se nos plantearía la opción de añadir una interface como `ICargoAggregateFactory`, de forma que tuviésemos:

```
namespace dddsample.domain.model.cargo.aggregate
{
```

```
    public class CargoAggregateFactory : ICargoAggregateFactory { (....) }
}
```

Pero tampoco aclara demasiado su responsabilidad.

Al menos ahora sabríamos que debe cumplir un contrato, pero no es exactamente lo que estamos buscando.

5.3.5.2 DDD_Intention-Revealing_Interfaces e ISP

Nos parece que hemos llegado a la situación perfecta para introducir una aplicación práctica del principio **DDD** conocido como **DDD_Intention-Revealing_Interfaces** y de la **I** de los principios **S.O.L.I.D.** (**Interface Seggregation Principle**)

Para ello obligaremos a `CargoAggregateFactory` a que implemente las dos interfaces siguientes:

```
public class CargoAggregateFactory : IRouteSpecificationFactory, ILegFactory
{ (....) }
```

De esta forma, la intención es muchísimo más evidente y además, estamos diciendo indirectamente, que cualquier objeto del **DDD_Aggregate** `Cargo`:

- Debe crearse aquí.
- Debe tener su propia interface que defina cuales son sus métodos de creación.

5.3.5.3 ILegFactory e IRouteSpecificationFactory

Necesitamos arreglar el código rojo, para ello basta con:

```
public interface ILegFactory {}

public interface IRouteSpecificationFactory {}
```

Ahora debemos llenarlas de contenido. Por lo tanto:

```
public interface ILegFactory
```

```
{
    ILeg create_leg_using(IVoyage the_voyage,
                          ILocation the_load_location,
                          ILocation the_unload_location,
                          IDate the_load_time,
                          IDate the_unload_time);
}

public interface IRouteSpecificationFactory
{
    IRouteSpecification create_route_specification_using(
                           ILocation the_origin_location,
                           ILocation the_destination_location,
                           IDate the_arrival_deadline);
}
```

Y esto sí que ya cumple **ISP** en oposición a una posible:

```
public interface ICargoAggregateFactory
{
    ILeg create_leg_using(IVoyage the_voyage,
                          ILocation the_load_location,
                          ILocation the_unload_location,
                          IDate the_load_time,
                          IDate the_unload_time);

    IRouteSpecification create_route_specification_using(
                           ILocation the_origin_location,
                           ILocation the_destination_location,
                           IDate the_arrival_deadline);
}
```

Vamos por el buen camino.

Nuestro sistema es más flexible y más expresivo.

Sin embargo, aun hay algo que no nos acaba de gustar y es que, si bien queda claro a través de nuestras **BDD_Specs**, cuales son las excepciones (violación de invariantes) posibles cuando cumplimos tanto `ILegFactory` como `IRouteSpecification`, no queda tan claro (no de forma explicita al menos), como debemos crear una instancia correctamente.

Por esa razón, vamos a introducir una nueva **BDD_Spec** que creemos, debería ayudar a paliar esta falla.

La definición formal de la **BDD_Spec** es la siguiente:

```
When creating a leg through the factory
   It should return an object that conforms the leg interface.
```

Y éste sería el código de la misma:

```
public class when_creating_a_leg_through_the_factory : concern_for_leg_factory
{
    Establish context = () =>
    {
        the_injected_voyage = an<IVoyage>();
        the_injected_load_location = an<ILocation>();
        the_injected_unload_location = an<ILocation>();
        the_injected_load_time = an<IDate>();
        the_injected_unload_time = an<IDate>();
    };

    Because of = () => result = sut.create_leg_using(
                                        the_injected_voyage,
                                        the_injected_load_location,
                                        the_injected_unload_location,
                                        the_injected_load_time,
                                        the_injected_unload_time);

    It should_return_an_object_that_conforms_the_leg_interface = () =>
       result.ShouldBeAn<ILeg>();

    static IVoyage the_injected_voyage;
    static ILocation the_injected_load_location;
    static ILocation the_injected_unload_location;
    static IDate the_injected_load_time;
    static IDate the_injected_unload_time;
    static ILeg result;
}
```

Esta vez la parte interesante está en el bloque `IT`:

```
result.ShouldBeAn<ILeg>();
```

Pues es ahí precisamente donde explicitamos algo que ya sabíamos, pero que creemos debe ser subrayado; que estamos creando un **DDD_Value_Object** `ILeg`.

Como con toda **BDD_Spec**, necesitamos un fallo significativo.

Hay varias formas de lograr esto, pero una de ellas sería sustituir en la linea anterior el tipo esperado por cualquier otro como por ejemplo una `ILocation`:

```
result.Should_Not_BeOfType(typeof(ILocation));
```

Si ejecutamos la **BDD_Spec** ahora, obtendremos nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when creating a leg through the factory
» should return an object that conforms the leg interface (FAIL)

Test 'should return an object that conforms the leg interface' failed:
      Machine.Specifications.SpecificationException: Should be of type
dddsample.domain.model.location.aggregate.ILocation but is of type
dddsample.domain.model.cargo.aggregate.Leg

(....)

0 passed, 1 failed, 0 skipped, took 0,82 seconds (Machine.Specifications
0.3.0).
```

Que es lo que buscábamos, o casi, ya que debemos fijarnos que el tipo esperado es:

```
dddsample.domain.model.cargo.aggregate.Leg
```

y no:

```
dddsample.domain.model.cargo.aggregate.ILeg
```

Es decir, el tipo concreto es `Leg`, no `ILeg`.

Pese a este nimio contratiempo, creemos firmemente que esta **BDD_Spec** sigue siendo necesaria.

Necesitamos que pase, así que si ejecutamos la **BDD_Spec** con el código original obtenemos:

```
------ Test started: Assembly: dddsample.specs.dll ------
```

```
when creating a leg through the factory
» should return an object that conforms the leg interface


1 passed, 0 failed, 0 skipped, took 1,12 seconds (Machine.Specifications
0.3.0).
```

Que era lo que esperábamos

Por lo tanto ahora ya hemos explicitado como se crea una `ILeg` a través de la **DDD_Factory**.

El proceso para crear una `RouteSpecification` a través de la **DDD_Factory** es análogo por lo que no lo mostraremos aquí, pero recordemos que el código fuente es nuestro amigo.

5.3.5.4 Una última mejora

Reflexionemos por un instante sobre el código siguiente:

```
public abstract class concern_for_leg : Observes<ILeg, Leg>
{
    Establish context = () =>
    {
        the_injected_voyage = an<IVoyage>();
        the_injected_load_location = an<ILocation>();
        the_injected_unload_location = an<ILocation>();
        the_injected_load_time = an<IDate>();
        the_injected_unload_time = an<IDate>();
        the_leg_factory = new CargoAggregateFactory();

        create_sut_using(() => the_leg_factory.create_leg_using(
                                                    the_injected_voyage,
                                                    the_injected_load_location,
                                                    the_injected_unload_location,
                                                    the_injected_load_time,
                                                    the_injected_unload_time));
    };

    protected static CargoAggregateFactory the_leg_factory;
    protected static IVoyage the_injected_voyage;
    protected static ILocation the_injected_load_location;
    protected static ILocation the_injected_unload_location;
    protected static IDate the_injected_load_time;
```

```
    protected static IDate the_injected_unload_time;
}
```

En concreto estamos interesados en esta linea:

```
protected static CargoAggregateFactory the_leg_factory;
```

Ésta era la única solución cuando `CargoAggregateFactory` estaba definida de la siguiente forma:

```
namespace dddsample.domain.model.cargo.aggregate
{
    public class CargoAggregateFactory { (....) }
}
```

Pero recordemos que con las mejoras que hemos realizado a través del **ISP** y el **DDD_Intention-Revealing_Interfaces**, ahora está definida de la siguiente forma:

```
public class CargoAggregateFactory : IRouteSpecificationFactory, ILegFactory
{ (....) }
```

Por lo tanto ya no tenemos la obligación de declarar `the_leg_factory` como un tipo concreto (`CargoAggregateFactory`), sino que podemos definirlo como la interface que cumple.

Es decir:

```
protected static ILegFactory the_leg_factory;
```

Recordemos que esta otra linea no cambia y sigue siendo valida:

```
the_leg_factory = new CargoAggregateFactory();
```

Si ejecutamos las **BDD_Specs**, tras este cambio, comprobaremos que no hemos roto nada:

```
246 passed, 0 failed, 0 skipped, took 3,62 seconds (Machine.Specifications
0.3.0).
```

Por lo tanto, este pequeño cambio hace que nuestro código sea más flexible y además ayuda a la comprensión del mismo puesto que nos indica que `CargoAggregateFactory` implementa `ILegFactory`.

Capítulo 6: El Aggregate Root Cargo (Parte II)

6.1 Implementando Equals()

Teníamos pendiente la implementación de `Equals()` en el **DDD_Aggregate_Root** `Cargo`.

6.1.1 El Happy Day Scenario

6.1.1.1 La BBD_Spec

Partimos de la siguiente definición formal:

> *When comparing two cargoes with the same identity using equals*
> *It should leverage the cargo entity comparer.*
> *It should confirm they are equal.*

Las dos aserciones en código son:

6.1.1.2 Assert

```
public class when_comparing_two_cargoes_with_the_same_identity_using_equals :
             concern_for_cargo
{
    It should_leverage_the_cargo_entity_comparer = () =>
       tracking_id.received(x => x.has_the_same_value_as(tracking_id));

    It should_confirm_they_are_equal = () => result.ShouldBeTrue();

    static bool result;
}
```

Observamos como para poder comprobar que estamos llamando a `has_the_same_value_as()` necesitamos asertar de forma indirecta.

Si el SUT fuese un mock podríamos asertar directamente que esperábamos que se produjese esa llamada, pero el SUT no es un mock.

Sin embargo en la implementación de `has_the_same_value_as()`, nos encontramos con que:

```
public bool has_the_same_identity_as(ICargo the_other_entity)
{
   return the_other_entity != null &&
          underlying_tracking_id.has_the_same_value_as(
                                    the_other_entity.tracking_id());
}
```

Por lo tanto podemos hacer esa misma comprobación asertando sobre `underlying_tracking_id`, que sí es un mock.

Eso es exactamente lo que haremos. Otra prueba más de lo importante que es usar **Dependency Injection** si pretendemos seguir los preceptos del **BDD**.

6.1.1.3 Act

El bloque `BECAUSE`, como siempre, es el más sencillo:

```
Because of = () => result = sut.Equals(the_other_cargo);

static ICargo the_other_cargo;
```

Simplemente llamamos a `Equals()`.

6.1.1.4 Arrange

Para establecer el contexto en el que se desarrollará la **BDD_Spec**, usamos un bloque `ESTABLISH`:

```
Establish context = () =>
{
    the_other_cargo = an<ICargo>();
    the_other_cargo
       .Stub(x => x.tracking_id())
       .Return(tracking_id);

    tracking_id
       .Stub(x => x.has_the_same_value_as(tracking_id))
       .Return(true);
};
```

No supone un desafío técnico, pero sin embargo es muy interesante observar como

establecemos las condiciones para que cuando se produzca la llamada a `has_the_same_value_as()` podamos simular que ambos `ICargo` tengan la misma identidad.

Este tipo de flexibilidad es lo que convierten el **BDD** en una herramienta tan poderosa.

6.1.1.5 RED

Vamos con el `RED`:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two cargoes with the same identity using equals
» should leverage the cargo entity comparer (FAIL)
» should confirm they are equal (FAIL)

Test 'should leverage the cargo entity comparer' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

Test 'should confirm they are equal' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)


0 passed, 2 failed, 0 skipped, took 0,98 seconds (Machine.Specifications
0.3.0).
```

Justo lo que esperábamos:

```
The method or operation is not implemented.
```

6.1.1.6 GREEN

Necesitamos implementarlo, y tal y como ya hemos visto anteriormente, con una sola

linea vamos a tener suficiente para todas las **BDD_Specs** relativas a `Equals()`:

```
public override bool Equals(object the_to_compare_object)
{
    return this.has_the_same_identity_as(the_to_compare_object as ICargo);
}
```

Delegamos la responsabilidad a `has_the_same_identity_as()`.

Con esto conseguimos que pase:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two cargoes with the same identity using equals
» should leverage the cargo entity comparer
» should confirm they are equal


2 passed, 0 failed, 0 skipped, took 0,89 seconds (Machine.Specifications
0.3.0).
```

6.1.2 El primer escenario complementario

Muy poco hay que comentar de esta segunda **BDD_Spec**, ya que es análoga a la anterior pero donde pone `true` ahora ponemos `false`.

Formalmente, la **BDD_Spec** es la siguiente:

```
When comparing two cargoes with different identity using equals
   It should leverage the cargo entity comparer.
   It should confirm they are different.
```

Vemos como con respecto a la **BDD_Spec** del Happy Day Scenario, solo cambia la segunda aserción, ya que ahora comprobamos que son diferentes:

```
It should_confirm_they_are_different = () => result.ShouldBeFalse();
```

Lo que provoca un cambio mínimo también a la hora de establecer el contexto, al forzar que cuando se produzca la comparación ahora sea `false`:

```
Establish context = () =>
{
    the_other_cargo = an<ICargo>();
    the_other_cargo
       .Stub(x => x.tracking_id())
       .Return(tracking_id);

    tracking_id
       .Stub(x => x.has_the_same_value_as(tracking_id))
       .Return(false);
};
```

Con solo esos dos cambios, la **BDD_Spec** ya pasa:

```
------ Test started: Assembly: dddsample.specs.dll ------

when comparing two cargoes with different identity using equals
» should leverage the cargo entity comparer
» should confirm they are different


2 passed, 0 failed, 0 skipped, took 0,72 seconds (Machine.Specifications
0.3.0).
```

6.1.3 Los dos escenarios complementarios restantes

En estas dos **BDD_Specs** comprobamos casos extremos. Qué pasa si comparamos un `ICargo`:

1. Con otro `ICargo` nulo.
2. Con un objeto cualquiera que no sea un `ICargo`.

```
When comparing any cargo with a null cargo using equals
   It should leverage the cargo entity comparer.
   It should confirm they are different.

When comparing any cargo with a different type object using equals
   It should leverage the cargo entity comparer.
   It should confirm they are different.
```

Ambas comparten bloques `IT`:

```
It should_leverage_the_cargo_entity_comparer = () =>
```

```
   tracking_id.never_received(x => x.has_the_same_value_as(tracking_id));

It should_confirm_they_are_different = () => result.ShouldBeFalse();
```

Lo más interesante es ver como comprobamos que se produce la llamada a `has_the_same_value_as()`.

Sabemos de las anteriores **BDD_Specs** que la llamada en sí, se produce, por lo tanto ahora lo que comprobamos es que cuando se produce, sea la primera condición la que se cumpla:

```
public bool has_the_same_identity_as(ICargo the_other_entity)
{
   return the_other_entity != null &&    (CONDICION 1)
          underlying_tracking_id.has_the_same_value_as(
                                    the_other_entity.tracking_id());
}
```

Y para ello asertamos que no llega a la segunda, por lo que, efectivamente se tuvo que cumplir la primera.

Probablemente estemos rizando el rizo en este caso.

De hecho, teniendo en cuenta las dos **BDD_Specs** anteriores, también sería correcto que las dos **BDD_Specs** que nos ocupan viniesen definidas como:

```
When comparing any cargo with a null cargo using equals
   It should confirm they are different.

When comparing any cargo with a different type object using equals
   It should confirm they are different.
```

Nuestra justificación para haber tomado el camino más complejo, es que, por un lado, desvelábamos la manera de comprobar cosas indirectamente, y por otro, tampoco está mal hacerlo así.

El único cambio a mayores, lo encontramos en los bloques `ESTABLISH`, muchísimo más sencillos.

En el caso de la comparación con otro `ICargo` nulo:

```
Establish context = () =>
{
    the_other_cargo = null;
};
```

Mientras que en el caso de un objeto con diferente tipo, hemos decidido que una `ILeg` nos sirve para establecer el contexto:

```
Establish context = () =>
{
    not_a_cargo = an<ILeg>();
};

static ILeg not_a_cargo;
```

En cualquier caso, las dos **BDD_Specs** pasan.

Capítulo 7: Aggregate Root Location - Llevando el BDD y el DDD hasta sus últimas consecuencias

7.1 Introducción

Antes de nada intentemos situar el contexto en el que nos vamos a mover para poder entender mejor la importancia de lo que intentamos poner de manifiesto con este capítulo.

7.1.1 Breve descripción del DDD_Aggregate Location

El **DDD_Aggregate** `Location` es muy sencillo, puesto que originalmente constaría de:

- `Location`: una **DDD_Entity** que además es el **DDD_Aggregate_Root**.
- `UnitedNationsLocationCode`: un **DDD_Value_Object**.
- `ILocationRepository`: el **DDD_Repository** en forma de interface.

Un **DDD_Aggregate_Root** `Location` en nuestro modelo, sería una parada en un viaje, como por ejemplo el origen desde donde se envía el paquete (ya hemos trabajado con ese concepto a través de **OriginLocation**) o el destino del mismo (conocido como **DestinationLocation**). También deberían resultarnos familiares los conceptos de `load_locations` y `unload_locations`.

Por lo tanto, intuitivamente deberíamos tener claro el concepto de **DDD_Aggregate_Root** `Location`.

Para poder construir un **DDD_Aggregate_Root** `Location`, inyectamos a través del constructor, un **DDD_Value_Object** `UnitedNationsLocationCode`, que a su vez es el responsable de proveer a nuestro **DDD_Aggregate_Root** de una **identidad** única.

No vamos a extendernos con el **DDD_Value_Object** `UnitedNationsLocationCode` ya que únicamente serviría para desviarnos de nuestro objetivo y entrar en una serie de consideraciones que nada aportarían a nuestra discusión.

De todas formas, podemos consultar más información en las siguientes direcciones web:

- http://www.unece.org/cefact/locode/
- http://www.unece.org/cefact/locode/DocColumnDescription.htm#LOCODE

El **DDD_Value_Object** `UnitedNationsLocationCode` no es el único objeto que inyectamos a nuestro **DDD_Aggregate_Root** `Location`, ya que necesitamos también un **LocationName**, que en el ejemplo original java es un `string`.

La intención es clara, poder disponer de un nombre real que nos permita asociar nuestro **DDD_Aggregate_Root** `Location` con la población que representa.

Adelantándonos un poco a nuestro desarrollo, decir que, aquí es donde vamos a abrir la **Caja de Pandora**.

El desarrollo de las **BDD_Specs** y su implementación, apenas suponen un desafío, ya que, se repiten conceptos que hemos tratado ampliamente. Como muestran estas dos **BDD_Specs**:

```
When returning the associated united nations location code
   It should return the underlying united nations location code.

When returning the actual location name
   It should return the underlying location name.
```

También se repiten patrones ya conocidos cuando nos referimos a las **BDD_Specs** responsables de proveer el comportamiento asociado a cualquier **DDD_Entity**:

```
When comparing two locations with the same identity
   It should leverage the underlying united nations location code value
comparer.
   It should confirm they have the same identity.

When comparing two locations with the different identity
   It should leverage the underlying united nations location code value
comparer.
   It should confirm they have different identity.

When comparing any location with null location
   It should confirm they have different identity.
```

Evidentemente estas últimas tres **BDD_Specs** definen el comportamiento de `has_the_same_identity_as()`, método que toda **DDD_Entity** debe implementar al implementar la interface `IEntity<T>` y que en nuestro caso delega la responsabilidad en el `UnitedNationsLocationCode`, tal y como habíamos expuesto una lineas más arriba.

A mayores del diseño original, hemos incorporado una **DDD_Factory** llamada `LocationAggregateFactory` y que por el momento solo implementa una interface, `ILocationFactory`, ya que, no nos parece una muy buena práctica hacer que el constructor lance excepciones en caso de que los argumentos inyectados no sean validos. Preferimos tener una **DDD_Factory** que se encargue de ello para evitar violar el **SRP**:

```
public interface ILocationFactory
{
    ILocation create_location_using(
                IUnitedNationsLocationCode the_united_nations_location_code,
                string the_location_name);
}
```

Con estos datos deberíamos estar listos para poder entender lo que vamos a plantear a continuación.

7.2 Implementando GetHashCode()

7.2.1 El porqué de este escenario

Hay varias formas de introducir, o incluso provocar, la siguiente discusión. Una de ellas, la que nos sirvió a nosotros para confirmar que había un problema, si bien ya intuíamos desde el principio que algo no iba del todo bien, sería plantear la **BDD_Spec** responsable del comportamiento asociado al calculo del **hash code** del **DDD_Aggregate_Root** `Location`:

```
When calculating the location hash code
   It should leverage the underlying united nations location code hash
code.
   It should leverage the underlying location name hash code.
   It should return the hash code.
```

La intención es cristalina, para poder calcular el **hash code** del **DDD_Aggregate_Root** `Location`, delegamos una parte de dicho calculo a los objetos inyectado en la construcción, a saber:

- **DDD_Value_Object** `UnitedNationsLocationCode`.
- (Aquí está el problema) **LocationName**.

7.2.2 El primer síntoma

Para empezar no sabemos que "**nombre**", dentro de la terminología **DDD**, usar para referirnos a **LocationName**. Parece que se trata de un **DDD_Value_Object**. Sin embargo, al ser implementado directamente a través de un `string`, no podemos obligarlo a implementar la interface `IValueObject<T>` que, además de "**marcar**" al objeto como un **DDD_Value_Object**, nos proporcionaría el método `has_the_same_value_as()`.

Pero ya nos estamos adelantando a los acontecimientos. Vamos primeramente a ver los problemas que se nos plantean al intentar traducir a código la **BDD_Spec**.

7.2.3 La clase base concern_for location

Antes de nada, mostraremos la **clase base abstracta** de la que heredan todas nuestras **BDD_Specs** relativas al **DDD_Aggregate_Root** `Location`, ya que sin ella, complicaríamos la comprensión de la **BDD_Spec** que queremos desarrollar:

```
public abstract class concern_for_location : Observes<ILocation, Location>
{
    Establish context = () =>
    {
        the_injected_united_nations_location_code =
           an<IUnitedNationsLocationCode>();
        the_injected_location_name = "Location Name Test";
        the_location_factory = new LocationAggregateFactory();

        create_sut_using(() =>
           the_location_factory.create_location_using(
                                    the_injected_united_nations_location_code,
                                    the_injected_location_name));
    };

    static protected IUnitedNationsLocationCode
                         the_injected_united_nations_location_code;
    static protected string the_injected_location_name;
    static ILocationFactory the_location_factory;
}
```

Se confirma por tanto lo esperado:

- La interface de la que heredamos es `ILocation`.
- La implementación es `Location`.
- Contamos con una instancia de la **DDD_Factory** `ILocationFactory` a través de `LocationAggregateFactory`.
- Declaramos como campos los dos argumentos necesarios para construir una `ILocation`:
  - `the_injected_united_nations_location_code`.
  - `the_injected_location_name`.
- Para la creación de una `ILocation` usamos la **DDD_Factory**.

7.2.4 Definición del Context / Specification en código

Ahora podemos ir con la definición del esqueleto de los bloques `IT` de nuestra **BDD_Spec**:

```
public class when_calculating_the_location_hash_code : concern_for_location
{
   It should_leverage_the_underlying_united_nations_location_code_hash_code =
      () => {};
   It should_leverage_the_underlying_location_name_hash_code = () => {};
   It should_return_the_hash_code = () => {};
}
```

Vamos a ir paso a paso para que la parte técnica no nos despiste del problema real.

Empezamos con la implementación del primer bloque `IT`:

```
It should_leverage_the_
   underlying_united_nations_location_code_hash_code = () =>
   the_injected_united_nations_location_code
      .received(x => x
      .GetHashCode());
```

Hemos hecho cosas parecidas unas cuantas veces.

Con esa linea de código, en realidad, estamos diseñando, ya que explicitamos la naturaleza de la relación (o interacción) existente entre nuestro SUT (el **DDD_Aggregate_Root** `Location`) y el argumento inyectado (el **DDD_Value_Object** `UnitedNationsLocationCode`).

Por lo tanto, ninguna sorpresa.

Vamos ahora con el tercer bloque `IT`:

```
It should_return_the_hash_code = () => result.ShouldEqual(new int());
```

Directo y sin complicaciones. Hemos supuesto que el resultado va a ser un entero cualquiera porque, de antemano, es difícil calcular un **hash code** sin apenas datos. Sabemos que esto va a fallar y ya lo cambiaremos por un valor más adecuado a la realidad cuando llegue el momento.

Vamos con el segundo bloque `IT`:

```
It should_leverage_the_underlying_location_name_hash_code = () =>
   the_injected_location_name
      .received(x => x
      .GetHashCode());
```

Usamos la misma lógica que para el primer bloque `IT`.

7.2.5 El segundo síntoma

Sin embargo, hay una señal inequívoca de que nos vamos a estrellar.

El problema es que `the_injected_location_name` es un `string` y por tanto no podemos ejecutar:

```
the_injected_location_name.received(x => x.GetHashCode());
```

ya que `the_injected_location_name` no es mockable.

Ésto desencadenaría toda la reflexión que queremos plantear. Pero supongamos que no nos damos cuenta de este detalle y que seguimos adelante, algo que puede ocurrir perfectamente y nos va a servir para demostrar la red de seguridad que es disponer de un framework **BDD** tan fabuloso como `machine.specifications`.

Hemos terminado con los bloques `IT`.

7.2.6 Act

Seguimos con el bloque `BECAUSE`:

```
Because of = () => result = sut.GetHashCode();
```

Nada que comentar. Simplemente hacemos la llamada al método que queremos que provoque el comportamiento especificado.

7.2.7 Arrange

Por último vamos con el bloque `ESTABLISH`:

```
Establish context = () =>
{
    the_injected_united_nations_location_code
       .Stub(x => x.GetHashCode())
       .Return(1);

    the_injected_location_name
       .Stub(x => x.GetHashCode())
       .Return(2);
};
```

Tenemos dos mocks para simular el comportamiento de las llamadas a `GetHashCode()` de los argumentos que usamos para construir el **DDD_Aggregate_Root** `Location`.

7.2.8 El tercer síntoma

De nuevo, deberíamos darnos cuenta de que el segundo mock va a dar problemas, por las mismas razones expuestas al analizar el segundo bloque `IT`. Pero igual que entonces, vamos a suponer que también esta bandera de aviso pasa desapercibida.

7.2.9 El código completo de la BDD_Spec

Antes de ejecutar la **BDD_Spec**, mostraremos el código completo de la misma.

La razón para hacerlo es que esta **BDD_Spec** no se encuentra así en el código fuente, puesto que nos va a dar un error no esperado. La versión del código fuente es la que no da el error.

Por lo tanto, aquí va el código de la **BDD_Spec**:

```
public class when_calculating_the_location_hash_code : concern_for_location
{
    Establish context = () =>
    {
```

```
        the_injected_united_nations_location_code
            .Stub(x => x.GetHashCode())
            .Return(1);

        the_injected_location_name
            .Stub(x => x.GetHashCode())
            .Return(2);
    };

    Because of = () => result = sut.GetHashCode();

    It should_leverage_the
       _underlying_united_nations_location_code_hash_code = () =>
       the_injected_united_nations_location_code
          .received(x => x
          .GetHashCode());

    It should_leverage_the_underlying_location_name_hash_code = () =>
       the_injected_location_name
          .received(x => x
          .GetHashCode());

    It should_return_the_hash_code = () => result.ShouldEqual(new int());

    static int result;
}
```

7.2.10 RED

Para ejecutar la **BDD_Spec** correctamente, necesitamos varias cosas.

Primero, la interface `ILocation` debe contener el método `GetHashCode()`:

```
namespace dddsample.domain.model.location.aggregate.interfaces
{
    public interface ILocation : IEntity<ILocation>
    {
        int GetHashCode();
        IUnitedNationsLocationCode associated_united_nations_location_code();
        string actual_location_name();
    }
}
```

Segundo, la implementación `Location` debe lanzar una excepción cuando `GetHashCode()` es invocada:

```
namespace dddsample.domain.model.location.aggregate
{
    public class Location : ILocation
    {
        readonly IUnitedNationsLocationCode
                    underlying_united_nations_location_code;
        readonly string underlying_location_name;

        internal Location(
                   IUnitedNationsLocationCode the_united_nations_location_code,
                   string the_injected_location_name)
        {
            underlying_united_nations_location_code =
               the_united_nations_location_code;
            underlying_location_name = the_injected_location_name;
        }

        public override int GetHashCode()
        {
            throw new NotImplementedException();
        }

        (....)

    }
}
```

Tercero, la interface `IUnitedNationsLocationCode` debe contener el método `GetHashCode()`:

```
namespace dddsample.domain.model.location.aggregate.interfaces
{
    public interface IUnitedNationsLocationCode :
                     IValueObject<IUnitedNationsLocationCode>
    {
        int GetHashCode();
    }
}
```

Con esto estamos listos para ejecutar la **BDD_Spec** a la espera de nuestro fallo deseado:

```
  Method Not Implemented
```

que no se va a producir:

```
  ------ Test started: Assembly: dddsample.specs.dll ------
```

```
when calculating the location hash code
» should leverage the underlying united nations location code hash code
(FAIL)
» should leverage the underlying location name hash code (FAIL)
» should return the hash code (FAIL)

Test 'should leverage the underlying united nations location code hash
code' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.InvalidOperationException: The object 'Location Name Test' is not a
mocked object.

(....)

Test 'should leverage the underlying location name hash code' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.InvalidOperationException: The object 'Location Name Test' is not a
mocked object.

(....)

Test 'should return the hash code' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.InvalidOperationException: The object 'Location Name Test' is not a
mocked object.

(....)

0 passed, 3 failed, 0 skipped, took 1,55 seconds (Machine.Specifications
0.3.0).
```

7.2.11 La enfermedad

Y efectivamente no se produce, ya que el error que obtenemos en los tres casos es el mismo, pero no el esperado:

```
The object 'Location Name Test' is not a mocked object.
```

Teniendo en cuenta que "**Location Name Test**" es el valor de nuestro `the_injected_location_name`, el mensaje que recibimos es alto y claro.

Ya no podemos seguir ignorando los problemas que íbamos indicando a medida que

desarrollábamos el código de la **BDD_Spec**.

Como `the_injected_location_name` es un `string`, no podemos mockarlo. Por lo tanto nuestra segunda aserción (el segundo bloque `IT`) es difícil de comprobar:

```
It should leverage the underlying location name hash code.
```

Hay varias opciones, pero nosotros vamos a resumirlas en dos.

7.3 Primera solución

7.3.1 Descripción

Una primera solución, consistiría en intentar cambiar el código de la **BDD_Spec** para que se adapte al inconveniente descubierto con anterioridad.

Podríamos modificar el segundo bloque `IT` para evitar la necesidad de un mock.

Intentaríamos, por tanto, comprobar la aserción de forma indirecta:

```
It should_leverage_the_underlying_location_name_hash_code = () =>
   the_injected_location_name.GetHashCode().ShouldEqual(new int());
```

De esta forma nos bastaría con comprobar que el `GetHashCode()` de `the_injected_location_name` tiene un valor determinado (lo de "`new int()`" es transitorio, deberíamos cambiarlo por un valor concreto una vez obtengamos el fallo esperado).

Además, tendríamos que eliminar la parte errónea de la asignación de comportamiento del mock en el bloque `ESTABLISH`, de forma que ahora solo necesitaríamos:

```
Establish context = () =>
{
    the_injected_united_nations_location_code
       .Stub(x => x.GetHashCode())
       .Return(1);
}
```

Desaparece por tanto:

```
the_injected_location_name
   .Stub(x => x.GetHashCode())
   .Return(2);
```

Que también nos causaba problemas.

7.3.2 RED

Antes de nada vamos a ejecutar la **BDD_Spec** para ver si obtenemos el fallo deseado:

```
------ Test started: Assembly: dddsample.specs.dll ------

when calculating the location hash code
» should leverage the underlying united nations location code hash code
(FAIL)
» should leverage the underlying location name hash code (FAIL)
» should return the hash code (FAIL)

Test 'should leverage the underlying united nations location code hash
code' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

Test 'should leverage the underlying location name hash code' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

Test 'should return the hash code' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)

0 passed, 3 failed, 0 skipped, took 0,88 seconds (Machine.Specifications
0.3.0).
```

Efectivamente, ahora obtenemos el fallo deseado:

```
System.NotImplementedException: The method or operation is not
implemented.
```

Por lo tanto hasta aquí todo va bien.

7.3.3 GREEN

Ahora implementamos la funcionalidad que deseamos en el método `GetHashCode()`:

```
public override int GetHashCode()
{
    var result = 37;
    result = result * 19 + underlying_united_nations_location_code
                               .GetHashCode();
    result = result * 19 + underlying_location_name.GetHashCode();
    return result;
}
```

El código no se diferencia en nada de otros **hash codes** que hemos implementado con anterioridad.

Si ejecutamos la **BDD_Spec** ahora:

```
------ Test started: Assembly: dddsample.specs.dll ------

when calculating the location hash code
» should leverage the underlying united nations location code hash code
» should leverage the underlying location name hash code (FAIL)
» should return the hash code (FAIL)


Test 'should leverage the underlying location name hash code' failed:
       Machine.Specifications.SpecificationException: Should equal [0] but
is [341572917]
       at Machine.Specifications.ShouldExtensionMethods.ShouldEqual[T](T
actual, T expected)
       domain\model\location.aggregate\LocationSpecs.cs(132,0): at
dddsample.specs.domain.model.location.aggregate.when_calculating_the_locat
ion_hash_code.<.ctor>b__5()
       at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
       at Machine.Specifications.Model.Specification.Verify()

Test 'should return the hash code' failed:
       Machine.Specifications.SpecificationException: Should equal [0] but
is [341586293]
       at Machine.Specifications.ShouldExtensionMethods.ShouldEqual[T](T
actual, T expected)
       domain\model\location.aggregate\LocationSpecs.cs(137,0): at
dddsample.specs.domain.model.location.aggregate.when_calculating_the_locat
ion_hash_code.<.ctor>b__6()
       at
```

```
Machine.Specifications.Model.Specification.InvokeSpecificationField()
      at Machine.Specifications.Model.Specification.Verify()

1 passed, 2 failed, 0 skipped, took 0,94 seconds (Machine.Specifications
0.3.0).
```

Que en cierta forma es lo esperado, ya que si analizamos lo que ocurre, tenemos que:

- La **primera aserción** pasa.
- La **segunda aserción** falla.
- La **tercera aserción** falla.

7.3.3.1 Analizando la corrección de la primera aserción

La primera aserción pasa:

```
» should leverage the underlying united nations location code hash code
```

porque se produce la llamada al mock:

```
the_injected_united_nations_location_code.received(x => x.GetHashCode());
```

7.3.3.2 Analizando y corrigiendo el fallo de la segunda aserción

La segunda aserción falla:

```
Should equal [0] but is [341572917]
```

porque no hemos definido el valor que queremos comprobar (tenemos "`new int()`" que fija el valor a cero):

```
the_injected_location_name.GetHashCode().ShouldEqual(new int());
```

Por lo tanto debemos hacer caso al fallo y cambiar el "`new int()`" por:

```
the_injected_location_name.GetHashCode().ShouldEqual(341572917);
```

A continuación ejecutamos de nuevo esa aserción (solo esa) para ver si ahora es

correcto:

```
------ Test started: Assembly: dddsample.specs.dll ------

when calculating the location hash code
» should leverage the underlying location name hash code


1 passed, 0 failed, 0 skipped, took 0,73 seconds (Machine.Specifications
0.3.0).
```

Por lo tanto, la segunda aserción también pasa.

7.3.3.3 Analizando y corrigiendo el fallo de la tercera aserción

La tercera aserción falla por la misma razón que la segunda, si bien el valor que vamos a necesitar es distinto:

```
Should equal [0] but is [341586293]
```

Así que necesitamos cambiar esta linea de la aserción:

```
result.ShouldEqual(new int());
```

por esta otra:

```
result.ShouldEqual(341586293);
```

Ejecutamos de nuevo esta última aserción:

```
------ Test started: Assembly: dddsample.specs.dll ------

when calculating the location hash code
» should return the hash code


1 passed, 0 failed, 0 skipped, took 0,70 seconds (Machine.Specifications
0.3.0).
```

Y ahora pasa.

7.3.4 La solución funciona

Por lo tanto habríamos solucionado el problema:

```
------ Test started: Assembly: dddsample.specs.dll ------

when calculating the location hash code
» should leverage the underlying united nations location code hash code
» should leverage the underlying location name hash code
» should return the hash code


3 passed, 0 failed, 0 skipped, took 0,65 seconds (Machine.Specifications
0.3.0).
```

Con esta **BDD_Spec**:

```
public class when_calculating_the_location_hash_code : concern_for_location
{
    Establish context = () =>
    {
        the_injected_united_nations_location_code
           .Stub(x => x.GetHashCode())
           .Return(1);
    };

    Because of = () => result = sut.GetHashCode();

    It should_leverage_the
       _underlying_united_nations_location_code_hash_code = () =>
       the_injected_united_nations_location_code
          .received(x => x
          .GetHashCode());

    It should_leverage_the_underlying_location_name_hash_code = () =>
       the_injected_location_name.GetHashCode().ShouldEqual(341572917);

    It should_return_the_hash_code = () => result.ShouldEqual(341586293);

    static int result;
}
```

7.3.5 Mejoras

Ya hemos visto la mecánica para solucionar el problema, pero habría que estar seguros de que en realidad hacemos lo que queremos.

Es decir, asegurarnos de que con este bloque `IT`:

```
It should_leverage_the_underlying_location_name_hash_code = () =>
   the_injected_location_name.GetHashCode().ShouldEqual(341572917);
```

estamos verdaderamente comprobando que:

```
When calculating the location hash code
   It should leverage the underlying location name hash code.
```

Ya sabemos que de forma directa no es posible comprobarlo, de ahí toda esta discusión. Pero debemos estar seguros de que la forma indirecta escogida, comprueba la aserción, y es precisamente aquí donde nos asaltan las dudas.

Por el momento, lo que estamos comprobando es que el **hash code** de `the_injected_location_name` tiene un valor concreto. Pero eso, no supone que estemos usando ese valor en el cálculo del **hash code** del **DDD_Aggregate_Root** `Location`.

Si lo dejásemos tal cual está, lo único que estaríamos potenciando es la parte de "las **BDD_Specs** como documentación", ya que se leen tal cual queremos:

```
when calculating the location hash code
» should leverage the underlying united nations location code hash code
» should leverage the underlying location name hash code
» should return the hash code
```

Pero nos quedamos muy cortos a la hora de comprobar esa segunda aserción.

Continuando con la misma filosofía expuesta en la **primera solución**, una mejora sería considerar el siguiente bloque `IT`:

```
It should_leverage_the_underlying_location_name_hash_code = () =>
   result
      .ShouldEqual((37*19 + 1)*19 + the_injected_location_name.GetHashCode());
```

En este caso perdemos algo de claridad, ya que podríamos preguntarnos que narices pintan todos eso números:

```
(37*19 + 1)*19
```

Pero debemos recordar que estamos buscando una forma indirecta de comprobar que se produce una llamada a un método concreto de un objeto que está imposibilitado para ser un mock.

Viéndolo por el lado positivo, de esta forma, sí que comprobamos que esa llamada se produce, y al mismo tiempo, que el resultado de la misma es tenido en cuenta en el calculo general.

Por lo tanto, si escogemos ir por el camino de la primera solución, debemos utilizar este código.

Una forma de refinar un poco la aserción, sería calcular de antemano el valor numérico:

```
(37*19 + 1)*19
```

Y además invertir los sumandos:

```
(37*19 + 1)*19 + the_injected_location_name.GetHashCode()
```

De forma que el algoritmo mediante el cual lo calculamos no introduzca ruido a la aserción:

ANTES

```
It should_leverage_the_underlying_location_name_hash_code = () =>
   result
      .ShouldEqual((37*19 + 1)*19 + the_injected_location_name.GetHashCode());
```

DESPUÉS

```
It should_leverage_the_underlying_location_name_hash_code = () =>
   result
      .ShouldEqual(the_injected_location_name.GetHashCode() + 13376);
```

En nuestra opinión, el **ANTES** se lee como:

"**El resultado debe ser igual a**":

- Unos números que no entiendo
- MULTIPLICADOS POR
- Diecinueve
- SUMADOS A
- El **hash code** del nombre (**LO QUE NOS INTERESA**).

Mientra que el **DESPUÉS** es mucho más claro ya que se lee como:

"**El resultado debe ser igual a**":

- El **hash code** del nombre (**LO QUE NOS INTERESA**)
- + un numero.

Ésta es una solución pragmática para un problema grave y que por desgracia nos encontraremos muchísimas veces.

La ventaja es que no incurrimos en ningún tipo de **Complejidad Innecesaria aka Needless Complexity**, uno de los grandes males de la programación.

La desventaja es que no estamos siendo fieles al **BDD**, ya que, si algo hemos repetido a lo largo de este documento es que el **BDD**, en su **núcleo central**, está formado por **DISEÑO**. Y es el **DISEÑO** el que nos está gritando que hay algo que estamos haciendo mal y como respuesta, parece que hemos decidido taparnos los oídos.

7.4 Segunda solución

7.4.1 Cuando el código habla y no estamos escuchando

Antes de acometer esta segunda solución, vamos a glosar todos los avisos que nos iba proporcionando la traducción a código de nuestra **BDD_Spec**:

Primero. Mientras convertíamos a código la segunda aserción a través del segundo bloque `IT`:

```
It should_leverage_the_underlying_location_name_hash_code = () =>
   the_injected_location_name
      .received(x => x
      .GetHashCode());
```

PROBLEMA:`the_injected_location_name` es un `string` y por tanto no podemos tratarlo como un mock.

Segundo. En el bloque `ESTABLISH`, el siguiente mock va a fallar:

```
Establish context = () =>
{
    (....)

    the_injected_location_name
       .Stub(x => x.GetHashCode())
       .Return(2);
};
```

PROBLEMA:`the_injected_location_name` es un `string` y por tanto no podemos tratarlo como un mock.

Tercero. Al ejecutar la **BDD_Spec** a la espera de nuestro fallo deseado, nos llevamos la siguiente sorpresa:

```
System.InvalidOperationException: The object 'Location Name Test' is not a
mocked object.
```

PROBLEMA: Teniendo en cuenta que "**Location Name Test**" es el valor de nuestro

`the_injected_location_name`, el mensaje que recibimos podría ser interpretado como que `the_injected_location_name` es un `string` y por tanto no podemos tratarlo como un mock.

Es decir, el mismo problema en los tres avisos.

Y ahora viene las pregunta:

- ¿Que nos impide solucionar el problema desde la base?
- ¿Acaso el **BDD** no permite diseñar el sistema tal y como uno quiere?

Las respuesta a estas preguntas conforman la segunda solución.

7.4.2 Preparando el camino para la segunda solución

7.4.2.1 La interface ILocationName

Empezaremos alejándonos todavía más de la solución propuesta por el código original java, y tal y como ya hicimos cuando creamos `IDate`, vamos a crear el **DDD_Value_Object** `ILocationName`.

Para ello, crearemos la siguiente interface:

```
public interface ILocationName : IValueObject<ILocationName>{}
```

Ya sabemos que no nos hace falta, por el momento, crear la implementación de esta interface, pero si decidiésemos hacerlo, éste sería el esqueleto:

```
public class LocationName : ILocationName
{
    public bool has_the_same_value_as(ILocationName the_other_value_object)
    {
        throw new NotImplementedException();
    }
}
```

7.4.2.2 Retomando la BDD_Spec original

Ahora vamos a retomar la **BDD_Spec** original (anterior a la primera solución):

```
public class when_calculating_the_location_hash_code : concern_for_location
{
    Establish context = () =>
    {
        the_injected_united_nations_location_code
           .Stub(x => x.GetHashCode())
           .Return(1);

        the_injected_location_name
           .Stub(x => x.GetHashCode())
           .Return(2);
    };

    Because of = () => result = sut.GetHashCode();

    It should_leverage_the
       _underlying_united_nations_location_code_hash_code = () =>
       the_injected_united_nations_location_code
          .received(x => x
          .GetHashCode());

    It should_leverage_the_underlying_location_name_hash_code = () =>
        the_injected_location_name
           .received(x => x
           .GetHashCode());

    It should_return_the_hash_code = () => result.ShouldEqual(new int());

    static int result;
}
```

7.4.2.3 Aplicando ILocationName a la BDD_Spec

Necesitamos que cuando invocamos `GetHashCode()` en `the_injected_location_name`, invoquemos el `GetHashCode()` de la interface `ILocationName`, por lo tanto añadimos el método a la interface:

```
public interface ILocationName : IValueObject<ILocationName>
{
    int GetHashCode();
}
```

Pero con esto no basta ya que en `concern_for_location` al crear

`the_injected_location_name` debemos usar el tipo `ILocationName` y no el tipo `string` con todo lo que implica.

Podemos apreciar esto último en el **ANTES / DESPUÉS** con las lineas que cambian en cursiva:

ANTES

```
public abstract class concern_for_location : Observes<ILocation, Location>
{
    Establish context = () =>
    {
        the_injected_united_nations_location_code =
           an<IUnitedNationsLocationCode>();
        the_injected_location_name = "Location Name Test";
        the_location_factory = new LocationAggregateFactory();

        create_sut_using(() =>
           the_location_factory.create_location_using(
                                the_injected_united_nations_location_code,
                                the_injected_location_name));
    };

    static protected IUnitedNationsLocationCode
                        the_injected_united_nations_location_code;
    static protected string the_injected_location_name;
    static ILocationFactory the_location_factory;
}
```

DESPUÉS

```
public abstract class concern_for_location : Observes<ILocation, Location>
{
    Establish context = () =>
    {
        the_injected_united_nations_location_code =
           an<IUnitedNationsLocationCode>();
        the_injected_location_name = an<ILocationName>();
        the_location_factory = new LocationAggregateFactory();

        create_sut_using(() =>
           the_location_factory.create_location_using(
                                the_injected_united_nations_location_code,
                                the_injected_location_name));
    };
```

```
    static protected IUnitedNationsLocationCode
                        the_injected_united_nations_location_code;
    static protected ILocationName the_injected_location_name;
    static ILocationFactory the_location_factory;
}
```

Pero esto nos genera el siguiente problema. La **DDD_Factory** `ILocationFactory` espera los siguientes argumentos:

```
public interface ILocationFactory
{
    ILocation create_location_using(
                IUnitedNationsLocationCode the_united_nations_location_code,
                string the_location_name);
}
```

Por lo tanto debemos cambiarlos por:

```
public interface ILocationFactory
{
    ILocation create_location_using(
                IUnitedNationsLocationCode the_united_nations_location_code,
                ILocationName the_location_name);
}
```

Lo que nos obliga a cambiar su implementación a:

```
public class LocationAggregateFactory : ILocationFactory
{
    public ILocation create_location_using(
                IUnitedNationsLocationCode the_united_nations_location_code,
                ILocationName the_location_name)
    { (....) }
}
```

Todavía necesitamos hacer algunos cambios más, que poco aportan a la comprensión de la segunda solución. Se trata de cambios concretos en la implementación de las clases afectadas para acomodarlas a los nuevos requerimientos.

La única razón por la que hemos decidido documentarlos es para poder ilustrar que antes de ejecutar / crear nuevas **BDD_Specs** (en concreto, la relativa a la segunda solución), debemos acondicionar el sistema para el cambio y también debemos

cerciorarnos de que todas nuestras **BDD_Specs** funcionan tras esos cambios, ya que las **BDD_Specs** son nuestra red de seguridad que confirman que las nuevas características no afectan a todo lo que hemos programado hasta ahora.

Continuamos con este proceso.

7.4.2.4 Primer cambio en create_locations_using()

En la implementación del método `create_locations_using()` de la **DDD_Factory** `ILocationFactory`, necesitamos hacer dos cambios más:

El primer cambio lo podemos ver en el siguiente código:

```
if (String.IsNullOrEmpty(the_location_name.name()))
   throw new ArgumentNullException(
                "the_location_name",
                "Invariant Violated: ....");
```

Con esto, devolveríamos la representación del nombre (el `string`).

Qué esté en rojo, no debería suponer un problema. Hemos basado toda nuestra metodología de diseño en ir solucionando el código rojo.

Sin embargo, en este caso, no nos convence.

Hemos creado un nuevo **DDD_Value_Object**, `ILocationName`. Por lo tanto, creemos que ya no tiene sentido hacer esa comprobación en la **DDD_Factory**. La comprobación que deberíamos hacer, en cambio, sería:

```
if (the_location_name == null)
   throw new ArgumentNullException(
                "the_location_name",
                "Invariant Violated: ....");
```

Fijémonos por un instante en la parte que cambia:

```
ANTES
```

```
if (String.IsNullOrEmpty(the_location_name.name()))
```

```
DESPUÉS
```

```
if (the_location_name == null)
```

En el **ANTES** comprobábamos que un método de una instancia de un **DDD_Value_Object** `ILocationName` que devuelve un `string`, no sea ni nulo ni sea la cadena vacía. Esta comprobación hay que hacerla, pero no en ésta **DDD_Factory** sino que deberíamos tener unas nuevas **BDD_Specs** relativas al recién creado **DDD_Value_Object** `ILocationName` que se encargasen de ello, de forma que probablemente necesitemos otra **DDD_Factory** que se podría llamar `ILocationNameFactory` donde se comprueben las invariantes relativas al proceso de construcción de un **DDD_Value_Object** `ILocationName`. Ese es su sitio. Ahí es donde debe residir ese comportamiento / responsabilidad.

En la parte del **DESPUÉS** comprobamos que la instancia del **DDD_Value_Object** `ILocationName` sea correcta. Ese es el comportamiento / responsabilidad de la que debe encargarse la **DDD_Factory** `ILocationFactory`.

Por lo tanto no nos queda otra que modificar las **BDD_Specs** ya existentes.

Es decir, estas dos **BDD_Specs**:

```
When attempting to inject a null location name into the location
factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.

When attempting to inject an empty location name into the location
factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.
```

ahora carecen de sentido en este contexto. Serían las **BDD_Specs** del **ANTES**, y de alguna forma se transformarán en dos **BDD_Specs** que definirán el comportamiento del **DDD_Value_Object** `ILocationName`.

Pero lo que más nos interesa en estos momentos es saber cual sería la **BDD_Spec** del **DESPUÉS**. Pues bien, aquí la tenemos:

```
When attempting to inject a null location name into the location
factory
   It should throw a null argument exception.
   It should throw an invariant violated exception message.
```

Ésta si que define perfectamente el comportamiento que queremos obtener.

Antes de explicar cual es el segundo cambio necesario en la implementación de la **DDD_Factory** `ILocationFactory`, no podemos evitar volver a destacar, que todo este razonamiento, provocado por el uso del **BDD**, responde única y exclusivamente al **DISEÑO**. Lo repetiremos hasta la saciedad, cuando practicamos **BDD**, estamos **DISEÑANDO** y **REDISEÑANDO** constantemente.

7.4.2.5 Segundo cambio en create_locations_using()

El segundo cambio necesario se produce en esta linea de código:

```
return new Location(the_united_nations_location_code, the_location_name);
```

El constructor que invoca espera que `the_location_name` sea un `string`, en lugar de un `ILocationName` que es lo que nosotros pretendemos.

Por lo tanto, necesitamos cambiar la signatura del constructor de la clase `Location`, para que admita un **DDD_Value_Object** `ILocationName` en vez de un tipo primitivo `string`.

El cambio es muy sencillo, tal y como se ilustra aquí:

```
internal Location(IUnitedNationsLocationCode the_united_nations_location_code,
                  ILocationName the_injected_location_name)
{
    underlying_united_nations_location_code = the_united_nations_location_code;
    underlying_location_name = the_injected_location_name;
}
```

Al hacer este cambio, nos encontramos con el problema de que `underlying_location_name` es de tipo `string`. Por lo tanto también debemos cambiar su tipo a `ILocationName`:

```
public class Location : ILocation
{
    readonly IUnitedNationsLocationCode
                underlying_united_nations_location_code;
    readonly ILocationName underlying_location_name;

    internal Location(
                IUnitedNationsLocationCode the_united_nations_location_code,
                ILocationName the_injected_location_name)
    {
        underlying_united_nations_location_code =
           the_united_nations_location_code;
        underlying_location_name = the_injected_location_name;
    }

    (....)
}
```

Pero aquí no se acaban las consecuencias, ya que en el método `actual_location_name()` también tenemos problemas.

Como anteriormente `underlying_location_name` era un `string`, devolvíamos directamente `underlying_location_name`. Pero ahora no es un `string`, sino que es un `ILocationName`. Por lo tanto la manera más sencilla de resolver esto es crear en la interface que representa al **DDD_Value_Object** `ILocationName` un método llamado `name()` que devuelva ese `string` que necesitamos (de nuevo más **DISEÑO**):

```
public interface ILocationName : IValueObject<ILocationName>
{
    int GetHashCode();
    string name();
}
```

Por lo que ahora simplemente devolvemos ese `string`, resultado de la invocación del método `name()` en el **DDD_Value_Object** `ILocationName`:

```
public class Location : ILocation
{
```

```
    readonly IUnitedNationsLocationCode
       underlying_united_nations_location_code;
    readonly ILocationName underlying_location_name;

    internal Location(
                IUnitedNationsLocationCode the_united_nations_location_code,
                ILocationName the_injected_location_name)
    {
        underlying_united_nations_location_code =
           the_united_nations_location_code;
        underlying_location_name = the_injected_location_name;
    }

    public string actual_location_name()
    {
        return underlying_location_name.name();
    }

    (....)

}
```

Esto lanza una última pregunta y es que la **BDD_Spec** que se ocupaba de definir el comportamiento de `actual_location_name()` ya no sirve, puesto que lo que afirma no es cierto:

```
When returning the actual location name
   It should return the underlying location name.
```

Era cierto antes, pero ahora deberíamos considerar algo así como:

```
When returning the actual location name
   It should leverage the underlying location name representation.
```

La diferencia existente entre ambas **BDD_Specs** al traducirlas a código, debería ayudar a apuntalar lo expuesto:

ANTES:

```
It should_return_the_underlying_location_name = () =>
   result.ShouldEqual("boo");
```

DESPUÉS:

```
It should_leverage_underlying_location_name_representation = () =>
   the_injected_location_name.received(x=>x.name());
```

De esta forma, delegamos la responsabilidad a donde corresponde.

Con esto finalizamos los cambios necesarios.

7.4.2.6 Comprobando la magnitud de la hecatombe

Para evaluar la magnitud de la hecatombe que hemos provocado, debemos ejecutar la suite de **BDD_Specs**. Sin embargo, ni siquiera podemos ejecutarla porque el código no compila. Hemos roto el código en más de 5 puntos distintos.

Procedemos, con paciencia, a evaluar cada uno de los errores y observamos que todos ellos se producen en **BDD_Specs** que dependían directamente del hecho de que `ILocationName` antes era un `string`.

Uno de los ejemplos que se repiten es que:

ANTES:

```
static string the_injected_location_name;
the_injected_location_name = "Test Location Name";
```

DESPUÉS:

```
static ILocationName the_injected_location_name;
the_injected_location_name = an<ILocationName>();
```

Cuando ya hemos resuelto los problemas de compilación, ejecutamos la suite de **BDD_Specs** y comprobamos que hay varias aserciones que ya no se cumplen. Revisamos cada uno de los errores y los corregimos. Todos ellos apuntaban a la **DDD_Factory** `ILocationFactory`.

Finalmente obtenemos:

```
294 passed, 0 failed, 0 skipped, took 3,31 seconds (Machine.Specifications
0.3.0).
```

7.4.2.7 El sentido de todo el proceso

Como ya advertimos con anterioridad, lo importante de todo este proceso, no es entender todos y cada uno de los paso y sus implicaciones, sino más bien darnos cuenta de que al hacer un cambio como éste, antes de ponernos con ninguna otra cosa, debemos arreglar todo lo que hemos roto.

Ahora sí que por fin estamos listos para comprobar si la segunda solución funciona.

7.4.3 Comprobando la viabilidad de la segunda solución

Si ejecutamos, directamente, la **BDD_Spec** obtenemos que:

```
------ Test started: Assembly: dddsample.specs.dll ------

when calculating the location hash code v2
» should leverage the underlying united nations location code hash code
» should leverage the underlying location name hash code
» should return the hash code (FAIL)


Test 'should return the hash code' failed:
       Machine.Specifications.SpecificationException: Should equal [0] but
is [13378]
       at Machine.Specifications.ShouldExtensionMethods.ShouldEqual[T](T
actual, T expected)
       domain\model\location.aggregate\LocationSpecs.cs(161,0): at
dddsample.specs.domain.model.location.aggregate.when_calculating_the_locat
ion_hash_code_v2.<.ctor>b__8()
       at
Machine.Specifications.Model.Specification.InvokeSpecificationField()
       at Machine.Specifications.Model.Specification.Verify()

2 passed, 1 failed, 0 skipped, took 0,81 seconds (Machine.Specifications
0.3.0).
```

Es decir, tan solo necesitamos cambiar ese ”`new int()`” por el `13378` que es el

resultado real para que se cumpla:

```
ANTES:

It should_return_the_hash_code = () => result.ShouldEqual(new int());

DESPUÉS:

It should_return_the_hash_code = () => result.ShouldEqual(13378);
```

Y aquí nuestra **BDD_Spec** en funcionamiento:

```
------ Test started: Assembly: dddsample.specs.dll ------

when calculating the location hash code v2
» should leverage the underlying united nations location code hash code
» should leverage the underlying location name hash code
» should return the hash code


3 passed, 0 failed, 0 skipped, took 0,77 seconds (Machine.Specifications
0.3.0).
```

Voilà. Ya lo tenemos.

Nuestra segunda solución ya funciona.

7.5 La búsqueda del equilibrio: Embrace Change

Llegamos al final, pero no sin antes poner en valor las dos soluciones que planteamos a lo largo de este capítulo:

- La **primera solución** viola el espíritu del **BDD**, pero cumple con lo pactado a través de las **BDD_Specs** (que se supone salen de una análisis de requisitos).
- La **segunda solución** es un claro ejemplo del espíritu del **BDD**, pero dinamita (o refina) el diseño original al añadir un nuevo **DDD_Value_Object**.

Las implicaciones de esto son mucho más importantes de lo que parece a simple vista, ya que implican al núcleo mismo del **DDD**.

Se supone que las **BDD_Specs** están supeditadas a lo que necesita el cliente y por tanto serían un fiel reflejo de nuestro diseño **DDD**.

A su vez el diseño **DDD** se apoya en un concepto básico como es el **DDD_Ubiquitous_Language**.

Este **DDD_Ubiquitous_Language** es el que se pone en cuestión con la **segunda solución**, al crear el nuevo **DDD_Value_Object** `ILocationName` por nuestra cuenta y riesgo.

Ésto es completamente inaceptable. No podemos tomar una decisión de este calibre sin consultar con el equipo y probablemente con el cliente, ya que el término "**Location Name**" debe ser incorporado al **DDD_Ubiquitous_Language** y tanto el equipo como el cliente (representado por los expertos en el Dominio) deben estar de acuerdo.

Con todo esto, queremos resaltar que más allá de que la **segunda solución** sea más correcta que la **primera solución** desde un punto de vista estrictamente **BDDdiano**, el decidirnos por alguna de las dos, vendrá motivado por una reunión extra entre el equipo y el cliente en la que puedan aclarar el **DDD_Ubiquitous_Language**.

Suponiendo que a raíz de esa reunión optemos por la **segunda solución** y ampliemos

nuestro **DDD_Ubiquitous_Language**, probablemente se planteen otras preguntas como:

¿Debemos seguir devolviendo el `string` en `actual_location_name()` o por el contrario deberíamos devolver un `ILocationName`?

Recordemos que en estos momentos nuestro código es el siguiente:

```
public string actual_location_name()
{
    return underlying_location_name.name();
}
```

Y estamos planteándonos si no deberíamos hacer esto otro:

```
public ILocationName actual_location_name()
{
    return underlying_location_name;
}
```

Es muy posible que esta última pregunta debiera plantearse desde el punto de las **BDD_Specs**, de la siguiente forma:

AHORA:

```
When returning the actual location name
   It should leverage the underlying location name representation.
```

POSIBILIDAD QUE SE ABRE TRAS LA PREGUNTA:

```
When returning the actual location name
   It should return the underlying location name.
```

Con todo esto, solo pretendemos ilustrar como para encontrar el equilibrio entre dos Prácticas de Diseño con tanta personalidad como el **BDD** y el **DDD** no siempre hay un camino correcto y otro incorrecto. Según queramos valorar a la una o la otra, deberemos sucumbir a los compromisos, y lo que está claro es que esos compromisos deben hacerse en el contexto del equipo.

En nuestra opinión (y aquí deberíamos resaltar aquello de **humilde y personalísima**) la **segunda solución** es la buena, ya que, no vemos que estemos **destrozando** el diseño original, sino que por el contrario, tal y como esperamos haber demostrado a lo largo de este documento, no existe un diseño original correcto. El diseño es algo que va evolucionando a cada instante. Por lo tanto, mientras los cambios que hagamos vengan motivados por las razones correctas seremos siempre partidarios de poner en juego de forma activa aquello de **Embrace Change**.

Y con esto llegamos al final.

Apéndice I: Introducción al BDD con Machine Specifications

AI.1 Clonación, compilación

Necesitamos abrir una ventana de comandos. En nuestro caso hemos decidido ejecutar la ventana de comandos de **powershell**.

Para empezar, debemos **clonar** el **repositorio github** de `machine.specifications`.

Podemos **clonar** el original, o bien ir directamente a alguno de los **forks** existentes con distintas mejoras.

Para este proyecto, **clonaremos** el **fork** realizado por **Jean-Paul Boodhoo**:

```
git clone http://github.com/developwithpassion/machine.specifications.git
```

Si todo va bien, deberíamos obtener una salida parecida a ésta, en la ventana de comandos:

```
Initialized empty Git repository in
c:/development/machine.specifications/.git/
remote: Counting objects: 8771, done.
remote: Compressing objects: 100% (2509/2509), done.
remote: Total 8771 (delta 6165), reused 8607 (delta 6004)/s
Receiving objects: 100% (8771/8771), 66.66 MiB | 1.02 MiB/s, done.
Resolving deltas: 100% (6165/6165), done.
Checking out files: 100% (682/682), done.
```

De esta forma habríamos creado una copia local del proyecto.

Para proceder a compilarlo y obtener los binarios necesarios para utilizar `machine.specifications` en otro proyecto, basta con ejecutar bajo **powershell**:

```
.\build.cmd package
```

Si hemos realizado este último paso correctamente, las últimas lineas de la salida de la ventana de comandos deberían ser tal que así:

```
7-Zip (A) 4.64  Copyright (c) 1999-2009 Igor Pavlov  2009-01-03
Scanning
```

```
Creating archive
C:/development/machine.specifications/Distribution/Machine.Specifications-
net-3.5-Debug.zip

Compressing  Castle.Core.dll
Compressing  Castle.Core.pdb
Compressing  Castle.Core.xml
Compressing  Castle.DynamicProxy2.dll
Compressing  Castle.DynamicProxy2.pdb
Compressing  Castle.DynamicProxy2.xml
Compressing  CommandLine.dll
Compressing  CommandLine.xml
Compressing  InstallResharperRunner.4.1.bat
Compressing  InstallResharperRunner.4.5.bat
Compressing  InstallResharperRunner.5.0 - VS2008.bat
Compressing  InstallResharperRunner.5.0 - VS2010.bat
Compressing  InstallTDNetRunner.bat
Compressing  InstallTDNetRunnerSilent.bat
Compressing  License.txt
Compressing  Machine.Specifications.DevelopWithPassion.dll
Compressing  Machine.Specifications.DevelopWithPassion.pdb
Compressing  Machine.Specifications.DevelopWithPassion.Rhino.dll
Compressing  Machine.Specifications.DevelopWithPassion.Rhino.pdb
Compressing  Machine.Specifications.dll
Compressing  Machine.Specifications.dll.tdnet
Compressing  Machine.Specifications.GallioAdapter.3.1.dll
Compressing  Machine.Specifications.GallioAdapter.3.1.pdb
Compressing  Machine.Specifications.GallioAdapter.plugin
Compressing  Machine.Specifications.pdb
Compressing  Machine.Specifications.Reporting.dll
Compressing  Machine.Specifications.Reporting.pdb
Compressing  Machine.Specifications.Reporting.Templates.dll
Compressing  Machine.Specifications.ReSharperRunner.4.1.dll
Compressing  Machine.Specifications.ReSharperRunner.4.1.pdb
Compressing  Machine.Specifications.ReSharperRunner.4.5.dll
Compressing  Machine.Specifications.ReSharperRunner.4.5.pdb
Compressing  Machine.Specifications.ReSharperRunner.5.0.dll
Compressing  Machine.Specifications.ReSharperRunner.5.0.pdb
Compressing  Machine.Specifications.SeleniumSupport.dll
Compressing  Machine.Specifications.SeleniumSupport.pdb
Compressing  Machine.Specifications.TDNetRunner.dll
Compressing  Machine.Specifications.TDNetRunner.pdb
Compressing  mspec.exe
Compressing  mspec.pdb
Compressing  Newtonsoft.Json.dll
Compressing  Rhino.Mocks.dll
Compressing  Rhino.Mocks.xml
Compressing  Spark.dll
Compressing  Spark.pdb
Compressing  TestDriven.Framework.dll
Compressing  ThoughtWorks.Selenium.Core.dll
Compressing  ThoughtWorks.Selenium.Core.pdb
```

```
Everything is Ok
(Q)uit, (Enter) runs the build again
```

Si vamos al directorio especificado unas lineas más arriba, podremos obtener el fichero comprimido que contiene todos los binarios necesarios para empezar a utilizar `machine.specifications`:

```
C:/development/machine.specifications/Distribution/Machine.Specifications-
net-3.5-Debug.zip
```

NOTA ACTUALIZADA a 2 de junio de 2010

El proyecto `machine.specifications` (el **fork** de **Jean-Paul Boodhoo**) contiene un **bug** que imposibilita la correcta compilación y generación de binarios.

Una forma de evitar estos problemas consiste en:

Primero.

En:

```
(...)\machine.specifications\Source\
```

abrir:

```
Machine.Specifications-2010.sln
```

Segundo.

Añadir en la raíz de la **solución** de **Visual Studio** la:

```
Solution Folder "DevelopWithPassion"
```

Tercero.

Añadir a la **Solution Folder** "**DevelopWithPassion**" los 3 **proyectos** siguientes:

```
Machine.Specifications.DevelopWithPassion
Machine.Specifications.DevelopWithPassion.Rhino
Machine.Specifications.DevelopWithPassion.Specs
```

Cuarto.

Ejecutar bajo **powershell**:

```
.\build.cmd package
```

AI.2 Nuestra primera BDD_Spec con machine.specifications

> **NOTA**: Se supone que hemos creado previamente una **Solution** y un **Project** en **Visual Studio**. Asimismo, suponemos que hemos creado una clase que pueda albergar las **BDD_Specs**.

AI.2.1 Mecánica y orden

Empezaremos primero por la clase (el contexto) (`Context`):

```
public class when_adding_two_numbers
```

A continuación el bloque `IT` (la aserción) (Assert):

```
It should_return_the_sum_of_the_two_numbers = () => result.ShouldEqual(5);
```

que nos obliga a crear el campo `result`:

```
static int result;
```

Es decir, en este momento, esto es lo que tenemos:

```
public class when_adding_two_numbers
{
    It should_return_the_sum_of_the_two_numbers = () => result.ShouldEqual(5);

    static int result;
}
```

Simplemente leyendo lo que hemos escrito, podemos intuir la definición formal de la **BDD_Spec** (o constatación del `Context / Specification`):

> *When adding two numbers,*
> *It should return the sum of the two numbers.*

Ahora vendría la pregunta:

`WHY`?

que tiene que ser respondida con un bloque `BECAUSE` (Act):

```
Because of = () => result = Calculator.Add(2, 3);
```

Pero por el momento `Calculator` está en rojo, ya que ni siquiera existe la clase `Calculator`.

Vamos a crearla usando los `namespaces` correctos:

```
namespace machinesample
{
    public class Calculator
    {
        static public int Add(int first, int second)
        {
            throw new NotImplementedException();
        }
    }
}
```

Una vez hecho esto, los `Calculator` y `Add()` del bloque `BECAUSE`, ya no deberían estar en rojo, ya que ahora existen y por tanto debería compilar:

```
Because of = () => result = Calculator.Add(2, 3);
```

Si corremos el test no debería pasar (de hecho debería lanzar una excepción).

Para hacer pasar el test podemos sustituir:

```
throw new NotImplementedException();
```

por:

```
return first + second;
```

Y con eso debería ser suficiente.

AI.2.2 Breve recapitulación

El código de nuestra primera **BDD_Spec** quedaría tal que así:

```
namespace machinesample.tests
{
    public class when_adding_two_numbers
    {
        Because of = () => result = Calculator.Add(2, 3);

        It should_return_the_sum_of_the_two_numbers = () =>
           result.ShouldEqual(5);

        static int result;
    }
}

namespace machinesample
{
    public class Calculator
    {
        static public int Add(int first, int second)
        {
            return first + second;
        }
    }
}
```

AI.3 Algunas normas básicas

Éstas son algunas de las normas básicas que debemos seguir a la hora de crear nuestras **BDD_Specs** (`Context / Specification`):

- Solo podemos tener **un bloque** `BECAUSE` por cada **BDD_Spec**.
- Podemos tener **tantos bloques** `IT` como queramos por cada **BDD_Spec**.
- Se considera una "**buena práctica**", utilizar **bloques** `IT` **de una sola linea**, ya que ayuda a que nos centremos en validar una sola aserción cada vez (a través de una **expresión Lambda**).
- Una razón para violar la norma anterior, sería, que intentásemos validar una aserción lógica, ya que puede ocurrir que ésta se componga de varias lineas.

AI.4 Mejorando y ampliando nuestra primera BDD_Spec

Hay un par de detalles de nuestra primera **BDD_Spec** que podrían ser mejorados.

Supongamos que queremos evitar toda la parte `static`, es decir, que queremos instanciar un objeto de tipo `Calculator` y llamar al método **NO estático** `Add()`.

Podríamos usar un bloque `ESTABLISH` (Arrange).

Empezamos definiendo un nuevo campo de tipo `Calculator`:

```
static Calculator calculator;
```

Convertimos el método `Add()` en **NO estático**:

```
public int Add(int first, int second)
```

Y usamos, a continuación, el bloque `ESTABLISH`:

```
Establish context = () =>
{
    calculator = new Calculator();
};
```

En este caso hemos usado uno con un **Anonymous Method** para definir el comportamiento asociado al bloque `ESTABLISH`, en previsión de que necesitemos más lineas de código que nos ayuden a establecer el contexto. Sin embargo, al tratarse una sola linea, también valdría una **expresión Lambda**:

```
Establish context = () =>  calculator = new Calculator();
```

Con cualquiera de las dos opciones:

- **Anonymous Method**.
- **Expresión Lambda**.

si corremos ahora el test, debería pasar.

El código resultante sería:

```
namespace machinesample.tests
{
    public class when_adding_two_numbers
    {
        Establish context = () =>
        {
            calculator = new Calculator();
        };

        Because of = () => result = calculator.Add(2, 3);

        It should_return_the_sum_of_the_two_numbers = () =>
           result.ShouldEqual(5);

        static int result;
        static Calculator calculator;
    }
}

namespace machinesample
{
    public class Calculator
    {
        public int Add(int first, int second)
        {
            return first + second;
        }
    }
}
```

Al haber introducido el bloque `ESTABLISH` en la ecuación, explicitaríamos aun más el carácter AAA (Arrange-Act-Assert) de nuestra **BDD_Spec**:

```
// ARRANGE
Establish context = () => {};

// ACT
Because of = () => {};

// ASSERT
It should_return_the_sum_of_the_two_numbers = () => {};
```

AI.5 El propósito de las variables "context" y "of"

Es interesante detenerse por unos instantes a analizar el código resultante de nuestra **BDD_Spec**, para poder entender mejor el sentido de todos los elementos que entran en juego.

En concreto hay dos variables de las que no hemos explicado su propósito:

- `context`.
- `of`.

La única razón para que se llamen así, es que ayudan a que la **BDD_Spec** se pueda leer como un texto, en vez de código propiamente dicho.

Consideremos una de las múltiples alternativas:

```
Establish c = () => calculator = new Calculator();
Because o = () => result = calculator.Add(2, 3);
```

La intención no queda tan clara en este último ejemplo, comparado con:

```
Establish context = () => calculator = new Calculator();
Because of = () => result = calculator.Add(2, 3);
```

Se trata, por tanto, de una preferencia personal. Al igual que cualquier variable, podemos nombrarla según nos apetezca.

Una ventaja de utilizar esta nomenclatura, sin embargo, es que ayudamos a revelar la intención de nuestro código, y ese es un tema ampliamente tratado por **Eric Evans** en su "**Biblia Azul**" **Domain-Driven Design: Tackling Complexity in the Heart of Software** (**DDD_Intention-Revealing_Interfaces**), pero en un contexto un tanto diferente.

Éste sería un ejemplo perfecto de como el **BDD** puede ayudarnos a aplicar conceptos propios del **DDD** y viceversa, como conceptos del **DDD** son aplicables, a distintos niveles, en el **BDD**.

AI.6 Capturando excepciones en una BDD_Spec

Vamos a preparar una **BDD_Spec** que nos permita comprobar que una excepción concreta ha sido lanzada.

Para empezar, crearemos una nueva **BDD_Spec**, con un bloque `IT` que compruebe que la excepción ha sido lanzada.

Para ello necesitaremos también un campo en el cual almacenar dicha excepción:

```
public class when_attempting_to_add_and_either_of_the_numbers_is_negative
{
    It should_throw_an_argument_exception = () =>
       exception.ShouldBeAn<ArgumentException>();

    static Exception exception;
}
```

Poco más podemos comentar. La intención es clara y no hay nuevos elementos que destacar, a no ser por el **Extension Method** `ShouldBeAn<T>()`, que nos sirve para comprobar tipos (de la misma forma que la anteriormente utilizada `ShouldEqual()` nos servía para comprobar valores).

A continuación deberíamos ejecutar la llamada que provoque la excepción, es decir, llamar al método `Add()` con dos números (uno de los cuales debería ser negativo).

Eso es exactamente lo que hacemos a continuación, empaquetado por un bloque `BECAUSE`:

```
Because of = () => exception = Catch.Exception(() => calculator.Add(-2, 3));
```

Es interesante subrayar el hecho de que estamos comprobando exactamente el mismo método que en la **BDD_Spec** anterior (`Add()` en `Calculator`), pero en un contexto muy diferente, ya que ahora estamos interesados en las posibles excepciones que lance y no en el resultado de la operación suma.

Lo más curioso, como veremos a continuación, es que pese a la diferencia de

contextos tan clara, el bloque `ESTABLISH` (donde se establece el contexto) es el mismo que el de la anterior **BDD_Spec**.

¿Donde radica entonces la diferencia?

En la expectativa. En el bloque `IT` que establecimos unas lineas más arriba.

Continuamos con la **BDD_Spec**.

Tal y como ya habíamos adelantado, establecemos el mismo contexto que en la **BDD_Spec** anterior:

```
Establish context = () => calculator = new Calculator();
```

Cabe destacar el uso de la **Expresión Lambda** en vez del **Anonymous Method**.

Éste sería el código completo de nuestra **BDD_Spec**:

```
public class when_attempting_to_add_and_either_of_the_numbers_is_negative
{
    Establish context = () => calculator = new Calculator();

    Because of = () =>
       exception = Catch.Exception(() => calculator.Add(-2, 3));

    It should_should_throw_an_argument_exception = () =>
       exception.ShouldBeAn<ArgumentException>();

    static Calculator calculator;
    static Exception exception;
}
```

Que correspondería a la siguiente definición formal:

```
When attempting to add and either of the numbers is negative
   It should throw an argument exception
```

Si corremos ahora el test, nos encontraremos con que no pasa, ya que todavía no hemos modificado el método `Add()` de la `Calculator` para que compruebe que ninguno

de los dos operandos sea negativo, y en caso de serlo, lance la excepción adecuada.

Una forma de resolver esto sería:

```
public class Calculator
{
    public int Add(int first, int second)
    {
        ensure_both_numbers_are_positive(first, second);
        return first + second;
    }

    void ensure_both_numbers_are_positive(int first, int second)
    {
        if (first < 0 || second < 0)
        {
            throw new ArgumentException("Negative number detected.");
        }
    }
}
```

De hecho, con esta implementación el test pasa.

Con estas dos **BDD_Specs** hemos cubierto lo básico de `machine.specifications`. A continuación nos centraremos en las extensiones que ofrece el **fork** de **Jean-Paul Boodhoo**, `developwithpassion`.

AI.7 Machine Specifications DevelopWithPassion

Vamos a ver ahora como podríamos desarrollar las **BDD_Specs** anteriores haciendo uso de las extensiones `developwithpassion` de `machine.specifications`.

Antes de empezar, recordar que necesitamos referenciar dos **dlls**:

```
Machine.Specifications.DevelopWithPassion.dll
Machine.Specifications.DevelopWithPassion.Rhino.dll
```

AI.7.1 Redefiniendo el proceso con las extensiones DevelopWithPassion

Vamos a seguimos con el mismo **Workflow**:

- Nombre **BDD_Spec**.
- Bloque `IT`.
- Campo `result`.

Es decir:

```
public class when_adding_two_numbers
{
    It should_return_the_sum_of_two_numbers = () => result.ShouldEqual(5);

    static object result;
}
```

Por el momento nada ha cambiado.

Seguimos con el **Workflow** ya aprendido:

- Bloque `BECAUSE`.
- Resolvemos dependencias del bloque **BECAUSE**:
  - Creamos clase `SecondCalculator`.
  - Creamos método `Add()`.

En código.

El bloque `BECAUSE`:

```
Because of = () => result = SecondCalculator.Add(3, 2);
```

Y la clase `SecondCalculator`:

```
public class SecondCalculator
{
    static public int Add(int first, int second)
    {
        throw new NotImplementedException();
    }
}
```

Corremos el test para asegurarnos que no pasa, para a continuación escribir el código necesario para que pase:

```
static public int Add(int first, int second)
{
    return first + second;
}
```

Corremos nuevamente el test y comprobamos que todo está correcto.

Hasta aquí, todo es exactamente igual que antes, de no ser porque hemos añadido dos nuevas **dlls** al proyecto (que por cierto, todavía no hemos usado).

Vamos ahora con el primer cambio, `Observes`.

AI.7.2 Usando Machine.Specifications.DevelopWithPassion.Rhino.Observes

Creamos una nueva clase llamada `concern` que herede de `Observes`:

```
public class concern : Observes {}
```

y hacemos que nuestra **BDD_Spec** herede a su vez de `concern`:

```
public class when_adding_two_numbers : concern
```

Volvemos a correr el test para asegurarnos de que no hemos roto nada.

Ésta es la mecánica, pero, ¿que sentido tiene usar esta infraestructura?.

La respuesta es sencilla. Con esta infraestructura compartimos un contenedor en el que poder arrojar código que sea común a todas las **BDD_Specs**.

A continuación veremos un ejemplo.

Supongamos que cambiamos esta **BDD_Spec** para que use una instancia de `SecondCalculator` y `Add()` deje de ser `static`.

Creamos una segunda **BDD_Spec** exactamente igual a la del ejemplo anterior pero heredando de `concern`:

```
public class when_attempting_to_add_and_either_of_the_numbers_is_negative :
             concern
```

Finalmente implementamos en `SecondCalculator` la lógica necesaria para que esta segunda **BDD_Spec** también pase.

En código el resultado sería:

```
namespace machinesample.tests
{
    public class concern : Observes {}
    public class when_adding_two_numbers : concern
    {
        Establish context = () => calculator = new SecondCalculator();

        Because of = () => result = calculator.Add(3, 2);

        It should_return_the_sum_of_two_numbers = () => result.ShouldEqual(5);

        static int result;
        static SecondCalculator calculator;
    }
```

```
    public class when_attempting_to_add_and_either_of_the_numbers_is_negative :
                 concern
    {
        Establish context = () => calculator = new SecondCalculator();

        Because of = () =>
           exception = Catch.Exception(() => calculator.Add(-2, 3));

        It should_should_throw_an_argument_exception = () =>
           exception.ShouldBeAn<ArgumentException>();

        static SecondCalculator calculator;
        static Exception exception;
    }
}

namespace machinesample
{
    public class SecondCalculator
    {
        public int Add(int first, int second)
        {
            ensure_both_numbers_are_positive(first, second);
            return first + second;
        }

        void ensure_both_numbers_are_positive(int first, int second)
        {
            if (first < 0 || second < 0)
            {
                throw new ArgumentException("Negative number detected.");
            }
        }
    }
}
```

Vemos que en ambas **BDD_Specs** el Bloque `ESTABLISH` es el mismo, así que podríamos refactorizar este código haciendo uso de `concern`:

```
public class concern : Observes
{
    Establish context = () => calculator = new SecondCalculator();

    protected static SecondCalculator calculator;
}

public class when_adding_two_numbers : concern
{
    Because of = () => result = calculator.Add(3, 2);
```

```
    It should_return_the_sum_of_two_numbers = () => result.ShouldEqual(5);

    static int result;
}

public class when_attempting_to_add_and_either_of_the_numbers_is_negative :
             concern
{
    Because of = () =>
       exception = Catch.Exception(() => calculator.Add(-2, 3));

    It should_should_throw_an_argument_exception = () =>
       exception.ShouldBeAn<ArgumentException>();

    static Exception exception;
}
```

Es importante realizar una advertencia aquí.

No todos los practicantes de **BDD** están a favor de tener código compartido entre las **BDD_Specs**, ya que se supone que cada **BDD_Spec** debe ser autoexplicativa y por lo tanto cualquiera que lea una **BDD_Spec** debe entender perfectamente lo que se establece en ella.

Puesto que en nuestro ejemplo solo tenemos 2 **BDD_Specs**, éste puede parecer un problema menor, ya que al haber tan poco código, "levantar la vista" y leer la clase que hereda de `Observes`, no presenta mayor problema.

Sin embargo, la cosa cambia si tuviésemos, por ejemplo, 25 **BDD_Specs** relativas a un mismo `concern`.

Honestamente, es muy difícil sugerir siquiera una norma o regla general sobre esta práctica. Debe ser el desarrollador el que en cada caso decida si debe o no usar esta técnica. Probablemente si el bloque `ESTABLISH` estuviese formado por 15 lineas que poco aportan a las **BDD_Specs**, sería interesante plantearse su uso. También podemos favorecer esta técnica, en casos en los que la comprensión de la **BDD_Spec** no se vea alterada.

En nuestro caso, procuramos evitarla siempre que nos es posible.

Volviendo a nuestro ejemplo, destacar que la comprensión y significado de las **BDD_Specs** no parece verse alterado por el uso de esta técnica.

Tómese como ejemplo:

```
public class when_adding_two_numbers : concern
{
    Because of = () => result = calculator.Add(3, 2);

    It should_return_the_sum_of_two_numbers = () => result.ShouldEqual(5);

    static int result;
}
```

Totalmente aislada y sin saber que contiene `concern`, la intención no parece perderse por el camino.

De todas formas, `Observes` todavía tiene mucho más que ofrecernos.

AI.7.3 Usando Machine.Specifications.DevelopWithPassion.Rhino.Observes<T> en conjuncion con Machine.Specifications.DevelopWithPassion.Observation SUT

Teniendo en cuenta que el **DDD** proclama a los cuatro vientos que nunca debemos olvidar que nuestro código debe revelar su intención, hay otra forma de usar `Observes` que se adhiere mejor a este principio.

En vez de explicar de antemano lo que hace el código, pondremos primero el código y después haremos algún comentario:

```
public class concern : Observes<SecondCalculator>
```

Resulta difícil incluso explicarlo mejor que el propio código.

Entre las ventajas de usar esta aproximación, nos encontramos con la posibilidad de usar el campo "`sut`", acrónimo de System Under Test (SUT), que, ¡oh sorpresa!, es del tipo `T` especificado por `Observes<T>` (en este caso `SecondCalculator`).

No pretendemos entrar en una explicación más extensa de la historia del termino SUT, para ello uno puede referirse a, por ejemplo, el gran libro de **Gerard Meszaros** sobre **Tests Patterns** (**xUnit Test Patterns – Refactoring Test Code**).

En un apartado más personal, debemos decir que la utilización del termino SUT, en nuestra modesta opinión, clarifica mucho la intención de las **BDD_Specs**, ya que ayuda a leerlas mejor y a tener un elemento más de anclaje al que poder agarrarnos a la hora de leer **BDD_Specs** no escritas por nosotros (`sut` ejercitaría el System Under Test).

Nuestro código ahora sería:

```
public class concern : Observes<SecondCalculator> {}

public class when_adding_two_numbers : concern
{
    Because of = () => result = sut.Add(3, 2);

    It should_return_the_sum_of_two_numbers = () => result.ShouldEqual(5);

    static int result;
}

public class when_attempting_to_add_and_either_of_the_numbers_is_negative :
             concern
{
    Because of = () => exception = Catch.Exception(() => sut.Add(-2, 3));

    It should_should_throw_an_argument_exception = () =>
       exception.ShouldBeAn<ArgumentException>();

    static Exception exception;
}
```

Las ventajas son obvias, al acceder al campo `sut`, evitamos tener que crear nuestro System Under Test, y aun por encima, ganamos en el apartado de **revelar intención**, estableciendo de forma clara que nuestro:

```
"Concern" "Observes SecondCalculator".
```

Ahora sí que cobra sentido el uso de `Observes<T>` (o casi podríamos decir `Observes<SUT>`). Incluso aunque no seamos grandes partidarios de la técnica de la **clase base** `concern`, podríamos usar `Observes<SUT>` directamente, tal y como se puede apreciar en las siguientes declaraciones de **BDD_Specs**:

```
public class when_adding_two_numbers : Observes<SecondCalculator> {}
public class when_attempting_to_add_and_either_of_the_numbers_is_negative :
             Observes<SecondCalculator> {}
```

Así, seguiríamos teniendo la capacidad de acceder al campo `sut`.

Pero ésta no es la única ventaja que nos proporciona...

...hay más.

AI.7.4 Usando Observes<T> en conjunción con exception_thrown_by_the_sut() y catch_exception()

En vez de usar:

```
public class when_attempting_to_add_and_either_of_the_numbers_is_negative :
             concern
{
    Because of = () => exception = Catch.Exception(() => sut.Add(-2, 3));

    It should_should_throw_an_argument_exception = () =>
       exception.ShouldBeAn<ArgumentException>();

    static Exception exception;
}
```

las ampliaciones de `developwithpassion`, nos permiten usar:

```
public class when_attempting_to_add_and_either_of_the_numbers_is_negative :
             concern
{
    Because of = () => catch_exception(() => sut.Add(-2, 3));

    It should_should_throw_an_argument_exception = () =>
       exception_thrown_by_the_sut.ShouldBeAn<ArgumentException>();
}
```

Al igual que con `Observes<SUT>` o con el propio `sut`, la intención es mucho más clara de esta última forma. Y además, evitamos tener que crear el campo `exception`.

Vamos, a continuación, con mas cambios entre:

- `machine.specifications` y
- `machine.specifications.developwithpassion`.

AI.7.5 Usando Dependency Injection (DI) en el System Under Test (SUT)

Supongamos que nuestro SUT (nuestra calculadora), necesitase en su constructor, una serie de dependencias.

Supongamos, por ejemplo, que nuestra calculadora se comunicase con una base de datos.

> **NOTA**: El hecho de que nuestro SUT sepa siquiera que existe una BD (y no digamos que se comunique con ella), es una muy mala práctica en **DDD**, ya que nuestra calculadora debería comunicarse con un **DDD_Repository** y éste, a su vez, sería el encargado de realizar las llamadas al origen de datos. Pero vamos a utilizar la aproximación directa para poder explicar el uso de las extensiones `machine.specifications.developwithpassion`.

Para no perder el **WorkFlow** aprendido.

En:

```
public class when_adding_two_numbers : concern
```

añadimos:

```
It should_open_a_connection_to_the_database = () =>
   connection.received(x => x.Open());
```

donde:

```
static IDbConnection connection;
```

Vemos por tanto, que estamos usando una nueva extensión, `.received()`, más o menos autoexplicativa (tengamos en cuenta que existe otra extensión llamada `.never_received()`).

`.received()`, nos permite asertar que el objeto que lo invoca (en nuestro caso, `connection`) ha recibido una llamada al método especificado en la **Expresión Lambda** (en nuestro caso `Open()`).

Para aquellos que estén familiarizados con el Interaction-Based Testing TDD o TDD Mockista (hemos de confesar que nos encontramos en ese grupo), ésto empezará a sonar agradablemente familiar. Para aquellos que no lo estén, quizás sea más complejo entender la utilidad de un **Extension Method** que nos permite comprobar si se ha realizado, o no, una llamada a un determinado método de una clase.

Continuamos estableciendo el contexto:

```
Establish context = () => connection = the_dependency<IDbConnection>();
```

Aquí introducimos otra nueva extensión, `the_dependency<T>`, que nos permite:

- Crear un mock del tipo `T` (en nuestro caso `IDBConnection`).
- Inyectárselo al constructor del SUT (en nuestro caso `SecondCalculator`).

Evidentemente, lo que estamos haciendo es automatizar la gestión de la **Inyección de Dependencias** (**DI – Dependency Injection**).

Debemos tener en cuenta un aspecto muy importante. Todas estas extensiones vienen respaldadas por distintos conceptos que han ido evolucionando con el tiempo (entre otros, el Interaction-Based Testing TDD). Si entendemos los conceptos originales, el **BDD** a través de `machine.specifications.developwithpassion`, no es más que un nuevo paso hacia el objetivo de diseñar y testar nuestro código con la menor fricción posible. De todas formas, aun entendiendo la dificultad que estos conceptos puedan suponerle al neófito, ésto no significa que no se pueda empezar desde aquí (hay

incluso quienes defienden que el **BDD** no es más que el "**TDD como Dios manda**"), e ir incorporando algunos conocimientos "**históricos**" según avanzamos. Como siempre, la ruta de aprendizaje que cada uno escoge es algo terriblemente personal.

Continuando con nuestro ejemplo, si corremos ahora el test, veremos que falla, ya que nos dice que:

```
should open a connection to the database : Failed
IDbConnection.Open(); Expected #1..2147483647, Actual #0.
```

Es decir, la extensión `.received()` invocada en la `IDBConnection` esperaba una llamada al método `Open()`, pero esa llamada no se ha producido.

Vamos a crear el código de `SecondCalculator` que nos permita pasar el test:

```
public class SecondCalculator
{
    IDbConnection connection;

    public SecondCalculator(IDbConnection connection)
    {
        this.connection = connection;
    }

    public int Add(int first, int second)
    {
        ensure_both_numbers_are_positive(first, second);
        connection.Open();
        return first + second;
    }
    (...)
}
```

Ahora el test pasa, gracias a que:

- Con la extensión `.received()` espera una llamada a `Open()`.
- Con la extensión `the_dependency<T>()` inyectamos al constructor de nuestro SUT el mock de `IDbConnection`.

Supongamos ahora, que establecemos el siguiente bloque `IT`:

```
It should_run_a_command = () => command.received(x => x.ExecuteNonQuery());
```

donde `command` sería un campo:

```
static IDbCommand command;
```

Si quisiésemos inyectárselo al constructor del SUT, ya sabemos que podemos usar la extensión `the_dependency<T>()`, pero si simplemente queremos crear un mock que no sea inyectado en el SUT, tenemos otra extensión más que podemos usar:

```
command = an<IDbCommand>();
```

La extensión `an<T>()`, crea un mock que cumple con el contrato de la interface `T`.

Continuado con el bloque `ESTABLISH`:

```
Establish context = () =>
{
    connection = the_dependency<IDbConnection>();
    command = an<IDbCommand>();
};
```

vemos que tenemos:

- Las dependencias necesarias para crear el SUT (`the_dependency<T>()`).
- Los mocks necesarios para ejercitar el SUT (`an<T>()`).

Y todo esto completamente aislado de su entorno, es decir, sin implementaciones reales de los objetos con los que interaccionan (usando fakes, mocks o stubs).

Nuevamente, aquellos familiarizados con el Interaction-Based Testing TDD deberían ser capaces de asimilar ésto sin ningún esfuerzo, ya que, la idea principal que subyace detrás de este ejemplo, es la misma que se encuentra detrás del Interaction-Based Testing TDD.

Una vez creadas las dependencias, necesitamos permitir que interactuen. En concreto, cuando hablamos de `IDbConnections` e `IDbCommands` en .NET, lo habitual es encontrarnos con la siguiente relación:

```
command = connection.CreateCommand();
```

Para ejercitar algo así dentro de nuestro bloque `ESTABLISH`, podemos recurrir a Rhino.Mocks (necesitamos añadir una referencia a su **dll** en nuestro proyecto) que nos provee del método `.Stub<T>` y `.Return()`, de forma que podemos hacer lo siguiente:

```
Establish context = () =>
{
    connection = the_dependency<IDbConnection>();
    command = an<IDbCommand>();

    connection
       .Stub(x => x.CreateCommand())
       .Return(command);
};
```

Es decir, una vez creados (e inyectados en caso de ser necesario) nuestros fakes, le decimos al sistema que cuando el campo `connection` reciba una llamada a su método `CreateCommand()` devuelva el fake `command` que acabamos de crear. Éste es el significado de la nueva linea de código.

Si corremos ahora el test, no va a pasar.

Añadimos el código que nos permite hacer pasar el test:

```
public int Add(int first, int second)
{
    ensure_both_numbers_are_positive(first, second);
    connection.Open();
    connection.CreateCommand().ExecuteNonQuery();

    return first + second;
}
```

Con esto satisfacemos la nueva **BDD_Spec**, ya que `connection` crea el comando a través del método `CreateCommand()` y éste, a su vez, ejecuta el método `ExecuteNonQuery()`, tal y como le pedíamos en nuestro bloque `IT`.

Hay una sutileza que no debemos pasar por alto.

Al tener esta linea en nuestro bloque `ESTABLISH`:

```
connection
   .Stub(x => x.CreateCommand())
   .Return(command);
```

estamos diciendo que no nos importa demasiado el **COMO** de esa linea.

Es decir, mientras que encapsulamos la llamada a `ExecuteNonQuery()` en un bloque `IT`:

```
It should_run_a_command = () => command.received(x => x.ExecuteNonQuery());
```

introduciendo una expectativa que debe cumplirse, no hacemos lo mismo para la llamada a `CreateCommand()`.

Ésto puede deberse a que así es como lo queremos diseñar, pero es muy importante entenderlo, ya que de lo contrario nos podemos llevar sorpresas desagradables más adelante si suponemos que tenemos cubierta la llamada a `CreateCommand()` con un test, cuando no es el caso.

Si en vez de ser una decisión consciente de diseño, fuese un error, o quizás de forma menos dramática, algo que hemos pasado por alto, podríamos solucionarlo de la siguiente forma. Creando un bloque `IT` que encapsule las expectativas que tenemos con respecto a `CreateCommand()`:

```
It should_use_the_connection_to_create_the_command = () =>
   connection.received(x => x.CreateCommand());
```

Ya está. Eso es todo.

Una cuestión diferente es la capacidad que tengamos para separar lo que simplemente es **preparación** frente a la que son la **expectativas**.

Las primeras van dentro de un bloque `ESTABLISH`, mientras que las últimas tienen un bloque `IT` propio.

Pero esa es ya una cuestión de diseño que no afecta a como podemos usar `machine.specifications`.

AI.7.6 Creación manual del SUT

¿Como haríamos para crear el SUT de formar manual (suponiendo que no queremos o podemos usar la extensión `the_dependency<T>()`)?.

Si queremos **inyectar** manualmente una (o varias) **dependencias** al SUT pero no queremos crearlo nosotros, podemos usar la extensión `provide_a_basic_constructor_argument()`.

Supongamos que queremos **inyectar** manualmente una instancia concreta de una `SqlConnection`:

```
Establish context = () =>
{
    connection = new SqlConnection();
    provide_a_basic_sut_constructor_argument(connection);
}
```

De esta forma el SUT recibiría como parámetro en su constructor la instancia concreta (en oposición a faked / mocked) de `connection`.

Incluso podemos hacer algo como:

```
provide_a_basic_sut_constructor_argument<IDbConnection>(connection);
```

donde supongamos que:

```
var connection = new SqlConnection();
```

De esta forma, si el SUT espera una `IDbConnection`, nos permitiría hacer la construcción.

Si quisiéramos, no ya **inyectar dependencias** manualmente, sino, crear nosotros mismos el SUT haciendo la llamada al constructor de forma explicita, podríamos usar

la extensión `create_sut_using()`.

Siguiendo nuestro ejemplo:

```
create_sut_using(() => new SecondCalculator(connection));
```

> **NOTA**: Hay problemas con `provide_a_basic_sut_constructor_argument<T>()` si el constructor necesita dos argumentos del mismo tipo.
>
> De hecho hemos encontrado ese mismo tipo de problemas usando `the_dependency<T>()`.
>
> Parece que lo más seguro en caso de que tengamos parámetros en el constructor del mismo tipo es usar `create_sut_using()`.

Y con esto, finalizamos este apéndice, en cuanto a **BDD** + `machine.specifications` se refiere.

AI.8 Algunos atajos de teclado útiles para el Jetbrains ReSharper

Añadir una referencia:

```
Alt + P(roject) + R(eference)
```

Goto Type:

```
Ctrl + N
```

Goto Symbol:

```
Ctrl + Shift + Alt + N
```

Goto File:

```
Ctrl + Alt + N
```

CleanUp Code:

```
Ctrl + Alt + F(ix)
```

Crear una nueva clase:

```
Alt + R(eshaper) + E(dit) + N(ew) + C(lass)
```

Como mover ficheros entre proyectos / carpetas / ... y ajustar los `namespaces` automáticamente:

```
Alt + Shift + L(ocate) para abrir solution explorer.
Navegamos con los cursores hasta situarnos encima del Project.
Alt + P(roject) + (New Fol)D(er) para crear una carpeta nueva.
Cortamos y pegamos el *.cs sobre la carpeta que queramos
Sobre la Carpeta invocamos Refactor This: Ctrl + Shift + R(efactor)
```

Generate:

```
Alt + Ins
```

Apéndice II: Interfaces DDD

AII.1 IValueObject<T>

Empezamos por las interfaces que definen los tipos de objetos existentes más representativos del dominio. Nos referimos a los **DDD_Value_Objects** y las **DDD_Entities**.

Una posible implementación de la interface de `IValueObject<T>`, siendo fieles a su versión java, sería:

```
namespace dddsample.domain.shared
{
    public interface IValueObject<T>
    {
        bool SameValueAs(T other);
    }
}
```

De forma qué, una clase que quiera implementar esta interface, tendría esta apariencia:

```
public class PruebaValueObject : IValueObject<PruebaValueObject>
{
    public bool SameValueAs(PruebaValueObject other)
    {
        throw new NotImplementedException();
    }
}
```

Hasta aquí todo parece ir bien, pero sin embargo, hay algo que no acaba de encajar del todo.

Ese algo es el nombre del método `SameValueAs()`.

Pero mejor vamos paso a paso para poder entender el porqué de las decisiones que estamos a punto de tomar.

La clase `PruebaValueObject` no sale de la nada. Si nos planteamos seguir los preceptos del **BDD**, es comprensible que a la hora de escribir el código de `IValueObject<T>` no necesitemos una **BDD_Spec** que nos guíe hacia el código que queremos escribir. Al fin

y al cabo, esa interface es realmente un contrato y una declaración de intenciones. Es más, sería la primera de muchas interfaces que se rigen por el principio del **DDD** enunciado por **Evans** en su **Biblia Azul** llamado **DDD_Intention-Revealing_Interfaces**.

Pero, ¿podemos hacer ese mismo razonamiento para `PruebaValueObject`?

La respuesta es un **NO** rotundo.

Hace falta esa **BDD_Spec** que justifique el comportamiento de `PruebaValueObject`.

Así que, vamos a escribir esa **BDD_Spec**:

```
public class concern : Observes<IValueObject<PruebaValueObject>,
                                PruebaValueObject>{}

public class when_asked_to_compare_two_pruebavalueobjects
             _that_have_the_same_value :
             concern
{
    Establish context = () =>
    {
        theOther = new PruebaValueObject();
        //TODO: le damos los valores adecuados.
    };

    Because of = () => result = sut.SameValueAs(theOther);

    It should_recognize_the_equality = () => result.ShouldEqual(true);

    static bool result;
    static PruebaValueObject theOther;
}
```

Sin entrar en demasiados detalles que no son importantes para nuestro razonamiento (en concreto los relativos al incompleto bloque `ESTABLISH`), debemos fijarnos en:

```
Because of = () => result = sut.SameValueAs(theOther);
```

que es donde ejecutamos el SUT.

Ahora viene la parte verdaderamente importante.

Puesto que debimos haber escrito la clase `PruebaValueObject` después de escribir esta **BDD_Spec** (más que **después**, parece más correcto afirmar, **como consecuencia de**), tendríamos la oportunidad de observar como se comportaría nuestro código desde el punto de vista de los clientes que van a tener que usarlo. De nuevo:

```
result = sut.SameValueAs(theOther);
```

Y es precisamente aquí, donde no hubiésemos escrito este código de esta manera, ya que no se acaba de leer correctamente, sobre todo cuando lo comparamos con algo como:

```
result = sut.HasTheSameValueAs(theOther);
```

que sí se lee como una frase bien construida.

AII.1.1 Notación all_lower

Aun así, podemos ir unos cuantos pasos más lejos, al hacer uso de la notación / convención propuesta por **Jean-Paul Boodhoo** y utilizar para:

- **métodos**
- **propiedades**
- **eventos**
- **variables locales**
- **constantes locales**
- **parámetros**
- **campos (no privados)**

un estilo **all_lower** en vez de **UpperCamelCase** o **lowerCamelCase**.

En nuestro caso, el resultado sería:

```
Because of = () => result = sut.has_the_same_value_as(the_other);
```

Esta versión es más legible, en nuestra humilde opinión.

En el contexto de la **BDD_Spec**:

```
public class concern : Observes<IValueObject<PruebaValueObject>,
                                PruebaValueObject>{}

public class when_asked_to_compare_two_pruebavalueobjects
             _that_have_the_same_value :
             concern
{
    Establish context = () =>
    {
        the_other = new PruebaValueObject();
        //TODO: le damos los valores adecuados.
    };

    Because of = () => result = sut.has_the_same_value_as(the_other);

    It should_recognize_the_equality = () => result.ShouldEqual(true);

    static bool result;
    static PruebaValueObject the_other;
}
```

Por lo tanto, retomando el argumento original que nos llevó a cuestionarnos el nombre del método `SameValueAs()` de la interface `IValueObject<T>`, el resultado final sería algo así:

```
namespace dddsample.domain.shared
{
    public interface IValueObject<T>
    {
        bool has_the_same_value_as(T the_other);
    }
}
```

Una cuestión distinta es que en .NET ya tengamos, por convención, un mecanismo similar al propuesto por `has_the_same_value_as()`, y que no es otro que:

```
public bool Equals(object obj) {}
```

Con esto finalizamos el código necesario para definir un **DDD_Value_Object**, al haber satisfecho la necesidad de que cualquier **DDD_Value_Object** debe proveernos de un mecanismo para averiguar si dos **DDD_Value_Objects** son iguales. Y lo son si

almacenan el mismo valor.

AII.1.2 Los comentarios y el código

Una cuestión que creemos importante abordar, es la inserción de comentarios en el código.

Desde un punto de vista estrictamente personal, no consideramos una buena práctica la inserción de los mismos ya que dañan seriamente la legibilidad del código. Es más, una de las razones para usar **all_lower** y utilizar esa convención donde los **métodos**, **campos**, **variables** son tan **redundantes**, es precisamente, evitar la necesidad del uso de comentarios en el código, en la medida de lo posible.

Sin embargo, en este caso, teniendo en cuenta la vocación y motivación pedagógica de este proyecto, vamos a transigir y hemos decidido, incluir los comentarios del proyecto original en java referentes a las interfaces que definen los contratos del **DDD**.

Aun así, que cada cual juzgue la legibilidad de estos dos fragmentos de código, que, opinamos, ilustran a la perfección nuestra reticencia hacia los comentarios (o al menos hacia el exceso de comentarios):

```
namespace dddsample.domain.shared
{
    public interface IValueObject<T>
    {
        /// <summary>
        /// Value objects compare by the values of their attributes,
        /// they don't have an identity.
        /// </summary>
        /// <param name="the_other">The other value object.</param>
        /// <returns><code>true</code> if the given value object's and
        /// this value object's attributes are the same.</returns>
        bool has_the_same_value_as(T the_other);
    }
}

namespace dddsample.domain.shared
{
    public interface IValueObject<T>
```

```
    {
        bool has_the_same_value_as(T the_other);
    }
}
```

En caso de no haber dejado suficientemente clara nuestra postura, decir que el primer fragmento de código es el que figura en el código fuente (en nuestra opinión, los comentarios dañan la legibilidad del código), mientras que el segundo fragmento de código es el que hubiésemos deseado escribir (mucho más legible al no tener comentarios pero si nombrar los métodos de una manera adecuada).

AII.2 IEntity<T>

De la misma forma que un concepto clave en **DDD** como **DDD_Value_Object** disponía de su propio **contrato** (léase **interface**), otro de los pilares del **DDD** como es **DDD_Entity** requiere del suyo.

No creemos oportuno volver a hacer los mismos comentarios que hemos realizado en las sección anterior dedicada a la interface `IValueObject<T>`, si bien, sería aplicable aquí en su totalidad.

El código es el siguiente:

```
namespace dddsample.domain.shared
{
    public interface IEntity<T>
    {
        /// <summary>
        /// Entities compare by identity, not by attributes.
        /// </summary>
        /// <param name="the_other">The other entity.</param>
        /// <returns><code>true</code> if the identities are the same,
        /// regardless of other attributes.</returns>
        bool has_the_same_identity_as(T the_other);
    }
}
```

AII.3 IValueObject<T> vs IEntity<T>

Una **DDD_Entity** requiere de una identidad única al contrario que un **DDD_Value_Object** que carece de la misma.

Otro aspecto destacable es ver como, pese a imponer la necesidad de tener un método como `has_the_same_identity_as()`, mientras en un **DDD_Value_Object** esta igualdad se basa en el valor de sus atributos, en el caso de una **DDD_Entity** viene dada por su identidad propiamente dicha, independientemente del valor de sus otros atributos, incluso aunque estos no coincidan.

Recordar brevemente que **Evans** en su **Biblia Azul**, pone como ejemplo de **DDD_Value_Object** un lápiz de color en manos de un niño haciendo un dibujo.

La igualdad / equivalencia de ese lápiz de color, vendrá determinado por su color (azul, verde, rojo, ...) y por características como si está afilado, etc.

Si cambiamos un lápiz por otro al niño que está haciendo el dibujo, y mantenemos estos atributos, podrá terminar de colorearlo sin ningún tipo de problema.

El lápiz no tiene identidad basada en su ocurrencia en el mundo real.

No hay unicidad. Hay atributos.

Por otro lado el ejemplo escogido por **Evans** para explicar una **DDD_Entity** es la confusión de identidad que asegura se produjo, cuando otro **Eric Evans** destrozó el piso en el que vivía y le llego él la demanda.

Al final consiguió aclararlo y la demanda no fue a mayores.

De esta forma puso de manifiesto como, en el caso de las personas, la identidad es algo básico, ya que puede haber dos personas distintas llamadas **Eric Evans** (coincidiría el atributo Nombre), pero, su identidad no es la misma, con lo cual no podemos hablar de igualdad en base a los atributos.

> **NOTA**: **Eric Evans**, realmente cuenta esta anécdota en su libro. Si es o deja de ser real es algo que desconocemos, pero ilustra magníficamente el concepto.

AII.4 IDomainEvent<T>

Éste es un concepto nuevo, que **Evans** no trata en su libro. Sin embargo es de vital importancia. Tanta, que en su charla en el **QCon 2009** de Londres y en el encuentro de Mayo de 2009 en el evento **DDD NYC** (con un titulo tan sugerente como "**Eric Evans: What I've learned about DDD since the book**"), habla largo y tendido sobre ellos.

El código que implementa la interface es el siguiente:

```
namespace dddsample.domain.shared
{
    /// <summary>
    /// A domain event is something that is unique,
    /// but does not have a lifecycle.
    /// The identity may be explicit,
    /// for example the sequence number of a payment,
    /// or it could be derived from various aspects of the event such as
    /// where, when and what has happened.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public interface IDomainEvent<T>
    {
        /// <summary>
        ///
        /// </summary>
        /// <param name="the_other_event">The other domain event.</param>
        /// <returns><code>true</code> if the given domain event
        /// and this event are regarded as being the same event.</returns>
        bool is_the_same_event_as(T the_other_event);
    }
}
```

Apéndice III: Implementando el patrón DDD_Specification (Versión BDD)

AIII.1 La interface ISpecification<T>

AIII.1.1 Objetivo inicial

El objetivo inicial que perseguimos es disponer de una **interface**, a partir de la cual, podamos implementar una versión abstracta del patrón **DDD_Specification**, que nos proporcione una implementación por defecto de las opciones / operaciones más usuales, a saber:

- **AND**.
- **OR**.
- **NOT**.

De forma que, el método que se encarga de verificar si se satisface la **DDD_Specification** se mantiene abstracto, obligando a cualquier clase que herede de nuestra **clase base abstracta** a proveer una implementación propia de dicho método.

Vamos con algo de código para aclarar toda esta idea.

AIII.1.2 Definición de ISpecification<T>

Ésta es precisamente la interface de la que hablábamos:

```
namespace dddsample.domain.shared
{
    public interface ISpecification<T>
    {
        bool is_satisfied_by(T item);
        ISpecification<T> and(ISpecification<T> spec);
        ISpecification<T> or(ISpecification<T> spec);
        ISpecification<T> not();
    }
}
```

Aquí con toda la ceremonia implícita de los comentarios:

```
namespace dddsample.domain.shared
{
```

```
    /// <summary>
    /// Specification interface.
    /// Use AbstractSpecification as base for creating specifications,
    /// and only the method is_satisfied_by(Object) must be implemented.
    /// </summary>
    /// <typeparam name="T"></typeparam>
    public interface ISpecification<T>
    {
        /// <summary>
        /// Check if item is satisfied by the specification.
        /// </summary>
        /// <param name="item">Object to test.</param>
        /// <returns>true if item
        /// satisfies the specification.</returns>
        bool is_satisfied_by(T item);

        /// <summary>
        /// Creates a new specification that is the AND operation
        /// of this specification and another specification.
        /// </summary>
        /// <param name="spec">Specification to AND.</param>
        /// <returns>A new specification.</returns>
        ISpecification<T> and(ISpecification<T> spec);

        /// <summary>
        /// Creates a new specification that is the OR operation
        /// of this specification and another specification.
        /// </summary>
        /// <param name="spec">Specification to OR.</param>
        /// <returns>A new specification.</returns>
        ISpecification<T> or(ISpecification<T> spec);

        /// <summary>
        /// Creates a new specification that is the NOT operation
        /// of this specification.
        /// </summary>
        /// <returns>A new specification.</returns>
        ISpecification<T> not();
    }
}
```

AIII.1.3 Error en la implementación java

Creemos haber descubierto un error en el código java que tomamos como punto de partida para el diseño de nuestro Domain Model, ya que la signatura propuesta para el método `not()` es la siguiente:

```
ISpecificationJava<T> not(ISpecificationJava<T> spec);
```

y esto es algo que no tiene ningún tipo de sentido.

Para implementar una operación **NOT** solo hace falta un único operando (en este caso, la propia clase que invoque **NOT**), al contrario que en las operaciones **AND** y **OR** donde son necesarios dos operandos.

De hecho, un ejemplo donde se muestra su invocación podría ser algo así como:

```
var not_spec = new ThisIsYourSpecification(33).not();
```

Estamos suponiendo que tenemos una **DDD_Specification** como:

```
ThisIsYourSpecification
```

que admite números enteros:

```
ThisIsYourSpecification(33)
```

y es negable:

```
ThisIsYourSpecification(33).not()
```

En ese caso, podemos ver claramente que:

```
not()
```

no admite ningún parámetro.

AIII.2 La clase abstracta Specification<T>

Vamos a intentar llegar a la implementación de la **clase abstracta** `Specification<T>` a través del **BDD**.

Tal y como está implementada en el código java hace que sea imposible de testar y mucho menos, de llegar a ella a través del **BDD**.

De una de las razones que convierten ese código en enemigo del **BDD** trata nuestra siguiente sección.

AIII.2.1 "Inyectar o no Inyectar, he aquí la cuestión"

El problema de base es que la implementación de la **clase abstracta** en java oculta sus dependencias.

Vamos a explicar este concepto.

AIII.2.1.1 Dependencias ocultas y la imposibilidad de mockarlas

Una clase que oculta sus dependencias es aquella que de forma opaca crea otros objetos de los que depende y sin lo cuales no puede realizar el trabajo que se le ha encomendado.

Traduciendo este concepto (casi **anti-pattern**) a código, obtendríamos algo como:

```
public abstract class Specification<T> : ISpecification<T>
{
    public ISpecification<T> and(ISpecification<T> the_other)
    {
        return new AndSpecification<T>(this, the_other);
    }
}
```

El método `and()`, internamente, crea una instancia de la clase `AndSpecification<T>`, de forma que nadie que vea desde fuera esta clase pueda saber que la clase `Specification<T>` tiene esa dependencia.

Uno de los problemas con este tipo de **prácticas** desde el punto de vista del **BDD** (o Interaction-Based Testing TDD), es que es imposible testar esta clase y más concretamente este método, pues no hay una forma clara de sustituir esa llamada a la construcción de la clase `AndSpecification<T>`.

Lo que estamos diciendo es que no podemos crear un objeto falso (un mock) que nos permita definir una aserción, si bien, en la actualidad existe algún Mocking Framework que permite hacer cosas parecidas.

Sin dudar del valor que esto representa, sobre todo cuando nos enfrentamos a código fuente heredado (**Legacy Code** o **Brownfield Development**) y con poca capacidad de cambio, no nos parece una práctica saludable, ya que, en la mayoría de los casos se pueden optar por alternativas más **testables**, y probablemente, más solidas y fáciles de mantener a medio y largo plazo.

AIII.2.1.2 Constructor Dependecy Injection

Para entender mejor el concepto de dependencia oculta, es interesante considerar la alternativa.

Una de las alternativa sería reconocer de antemano todas las dependencias de nuestra clase e intentar inyectárselas a través del constructor (**Constructor Dependency Injection**).

Es decir, aplicado a nuestro caso, tendríamos algo como:

```
public abstract class Specification<T> : ISpecification<T>
{
    AndSpecification<T> and_specification;

    public Specification(AndSpecification<T> and_specification)
    {
        this.and_specification = and_specification;
    }

    public ISpecification<T> and(ISpecification<T> the_other_specification)
    {
        return and_specification(this, the_other_specification);
```

```
    }
}
```

Y enfatizamos el **algo como** ya que hemos hecho trampas.

`Specification<T>` es una **clase abstracta**, por lo que al declararle un constructor nos vemos en la obligación de heredar la signatura del mismo en cualquier clase que herede de `Specification<T>`, y en el caso de `AndSpecification<T>`, la signatura del constructor de `Specification<T>` simple y llanamente carece de sentido, ya que el constructor que buscamos para `AndSpecification<T>` sería algo así como:

```
public AndSpecification(ISpecification<T> left_side,
                        ISpecification<T> right_side)
```

Llegados a este punto, en el que claramente se pone de manifiesto que la aproximación escogida no es la correcta (para empezar ni siquiera compila) por muy buenos y correctos principios que estábamos intentando aplicar (**Constructor Dependency Injection**), podemos ir todavía más allá en la utopía y proponer la solución completa (con las operaciones **AND**, **OR** y **NOT**):

```
public abstract class Specification<T> : ISpecification<T>
{
    AndSpecification<T> and_specification;
    OrSpecification<T> or_specification;
    NotSpecification<T> not_specification;

    public Specification(AndSpecification<T> and_specification,
                         OrSpecification<T> or_specification,
                         NotSpecification<T> not_specification)
    {
        this.and_specification = and_specification;
        this.or_specification = or_specification;
        this.not_specification = not_specification;
    }

    public abstract bool is_satisfied_by(T item);

    public ISpecification<T> and(ISpecification<T> the_other_specification)
    {
        return and_specification(this, the_other_specification);
    }

    public ISpecification<T> or(ISpecification<T> the_other_specification)
    {
```

```
        return or_specification(this, the_other_specification);
    }

    public ISpecification<T> not()
    {
       return not_specification(this);
    }
}

public class NotSpecification<T> : Specification<T>
{
    ISpecification<T> to_negate;

    public NotSpecification(AndSpecification<T> and_specification,
                            OrSpecification<T> or_specification,
                            NotSpecification<T> not_specification) :
           base(and_specification, or_specification, not_specification) {}

    public NotSpecification(ISpecification<T> to_negate)
    {
        this.to_negate = to_negate;
    }

    public override bool is_satisfied_by(T item)
    {
        return !this.to_negate.is_satisfied_by(item);
    }
}

public class OrSpecification<T> : Specification<T>
{
    ISpecification<T> left_side;
    ISpecification<T> right_side;

    public OrSpecification(AndSpecification<T> and_specification,
                           OrSpecification<T> or_specification,
                           NotSpecification<T> not_specification) :
           base(and_specification, or_specification, not_specification) {}

    public OrSpecification(ISpecification<T> left_side,
                           ISpecification<T> right_side)
    {
        this.left_side = left_side;
        this.right_side = right_side;
    }

    public override bool is_satisfied_by(T item)
    {
        return this.left_side.is_satisfied_by(item) ||
               this.right_side.is_satisfied_by(item);
    }
}
```

```
public class AndSpecification<T> : Specification<T>
{
    ISpecification<T> left_side;
    ISpecification<T> right_side;

    public AndSpecification(AndSpecification<T> and_specification,
                            OrSpecification<T> or_specification,
                            NotSpecification<T> not_specification) :
           base(and_specification, or_specification, not_specification) {}

    public OrSpecification(ISpecification<T> left_side,
                           ISpecification<T> right_side)
    {
        this.left_side = left_side;
        this.right_side = right_side;
    }

    public override bool is_satisfied_by(T item)
    {
        return this.left_side.is_satisfied_by(item) ||
               this.right_side.is_satisfied_by(item);
    }
}
```

Podemos listar los errores de esta solución.

Primero, los tres `return` serían erróneos, ya que no devuelven el tipo esperado:

```
public ISpecification<T> and(ISpecification<T> the_other_specification)
{
    return and_specification(this, the_other_specification);
}

public ISpecification<T> or(ISpecification<T> the_other_specification)
{
    return or_specification(this, the_other_specification);
}

public ISpecification<T> not()
{
    return not_specification(this);
}
```

Segundo, ninguno de los tres constructores compilaría ya que no existe ningún constructor sin parámetros en la **clase abstracta** `Specification<T>` que nos permita tener la signatura deseada (podríamos alargar este fútil ejercicio hasta el infinito, pero no por ello estaríamos más cerca de la solución):

```
public NotSpecification(ISpecification<T> to_negate)
{
    this.to_negate = to_negate;
}

public OrSpecification(ISpecification<T> left_side,
                       ISpecification<T> right_side)
{
    this.left_side = left_side;
    this.right_side = right_side;
}

public AndSpecification(ISpecification<T> left_side,
                        ISpecification<T> right_side)
{
    this.left_side = left_side;
    this.right_side = right_side;
}
```

Con esto, creemos que ha quedado demostrado que, en cuanto optemos por implementar una **clase base abstracta** `Specification<T>` nos vemos inmersos en un problema de testabilidad. Dicho problema se deriva de que no parece que haya una forma transparente de tratar las dependencias e inyectarlas a través del constructor. Por lo tanto, hemos decidido asumir que nuestra solución para la **DDD_Specification** puede que no vaya a ser 100% testable, pero al menos vamos a intentar contener el impacto que esto nos produce.

AIII.2.2 Limitando Daños

Puesto que la solución ideal no es posible, tenemos que llegar a un compromiso que nos permita solucionar el problema y limitar al máximo los posibles daños o consecuencias.

AIII.2.2.1 Eliminando la clase abstracta Specification<T>

Una de las cuestiones que menos nos convencía era todo el asunto de la **clase base abstracta** `Specification<T>`. Si bien es excelente desde un punto de vista pedagógico, creemos que deberíamos intentar eliminarla.

La pregunta entonces sería, ¿es factible?.

Por suerte, la respuesta es afirmativa.

Para empezar, partiríamos de una interface de `ISpecification<T>` mucho más clara (expresa mejor la intención) y minimalista, sin que por ello vayamos a perder funcionalidad:

```
public interface ISpecification<T>
{
    bool is_satisfied_by(T item);
}
```

Con respecto a las tres operaciones; a saber, **AND**, **OR** y **NOT**; vamos a sacarlas fuera de la **clase base** `Specification<T>` (que deseamos eliminar) y vamos a hacer que implementen la interface `ISpecification<T>` en lugar de la **clase base abstracta** `Specification<T>`.

AIII.2.2.2 AndSpecification<T> sin Specification<T>

Usando como ejemplo la clase `AndSpecification<T>`:

```
public class AndSpecification<T> : ISpecification<T>
{
    public bool is_satisfied_by(T item)
    {
        throw new NotImplementedException();
    }
}
```

Para poder implementar el método `is_satisfied_by(T item)`, vamos a inyectar, a través del constructor, sus dos dependencias:

```
public AndSpecification(ISpecification<T> left_side,
                        ISpecification<T> right_side)
```

de forma que pueda realizar su trabajo con un simple:

```
return left_side.is_satisfied_by(item) &&
       right_side.is_satisfied_by(item);
```

El código completo es éste:

```
public class AndSpecification<T> : ISpecification<T>
{
    ISpecification<T> left_side;
    ISpecification<T> right_side;

    public AndSpecification(ISpecification<T> left_side,
                            ISpecification<T> right_side)
    {
        this.left_side = left_side;
        this.right_side = right_side;
    }

    public bool is_satisfied_by(T item)
    {
        return left_side.is_satisfied_by(item) &&
               right_side.is_satisfied_by(item);
    }
}
```

Es decir, hemos eliminado completamente la necesidad de la **clase base abstracta** `Specification<T>` a la hora de implementar las 3 operaciones que necesitamos (**AND**, **OR** y **NOT**) y por el momento nos mantenemos 100% testables al inyectar las dependencias a través del constructor.

Una vez visto el código, debería resultar mucho más asequible el entender como se llegó a él a través de un proceso **BDD**.

En vez de empezar por `AndSpecification<T>`, empezaremos por `OrSpecification<T>`.

AIII.3 OrSpecification<T> vía BDD

AIII.3.1 Definiendo la BDD_Spec

Nuestra **BDD_Spec** para la clase `OrSpecification<T>` sería:

```
When asked if an item that meets both conditions satisfies the or
operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should not evaluate the right side condition.

When asked if an item that meets only the left side condition satisfies
the or operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should not evaluate the right side condition.

When asked if an item that meets only the right side condition
satisfies the or operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.

When asked if an item that does not meet any of the conditions
satisfies the or operator specification
   It should confirm that the or operator specification was not
satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.
```

Con esta **BDD_Spec** definiríamos perfectamente el comportamiento que esperamos alcanzar con la clase `OrSpecification<T>`.

AIII.3.2 Introduciendo el concepto del Happy Day Scenario

La primera condición:

```
When asked if an item that meets both conditions satisfies the or
operator specification
```

correspondería al Happy Day Scenario, usando una expresión atribuida a **Jean-Paul Boodhoo**.

La definición de la expresión es muy sencilla. El Happy Day Scenario es aquel escenario representado por la **BDD_Spec** en la que todo va bien (se cumplen todas las condiciones). Algo así como el **caso base**.

El sentido de intentar identificar siempre el Happy Day Scenario viene de que, sin lugar a dudas, es una práctica positiva empezar la codificación por él, para luego ir centrándonos en escenarios más concretos donde distintos elementos empiezan a fallar.

AIII.3.3 Implementando el Happy Day Scenario

Mantengamos la **BDD_Spec** como referencia:

```
When asked if an item that meets both conditions satisfies the or
operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should not evaluate the right side condition.
```

Su implementación en código usando `machine.specifications` con las extensiones `developwithpassion` sería:

```
public class concern_for_the_or_specification :
             Observes<ISpecification<IWhateverType>,
                      OrSpecification<IWhateverType>> {}

public class when_asked_if_an_item_that_meets_both_conditions
             _satisfies_the_or_operator_specification :
             concern_for_the_or_specification
{
    It should_confirm_that_the_or_operator_specification_was_satisfied = () =>
       result.ShouldBeTrue();

    It should_evaluate_the_left_side_condition = () =>
       left_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_both_conditions));
```

```
    It should_not_evaluate_the_right_side_condition = () =>
       right_side_specification
          .never_received(x => x
          .is_satisfied_by(the_item_that_meets_both_conditions));
}
```

La equivalencia entre la definición formal de la **BDD_Spec** y el código que la implementa es absoluta.

AIII.3.4 Anatomía de una BDD_Spec

Hemos nombrado un par de veces eso de **BDD_Spec**, pero todavía no hemos explicado en que consiste, si bien, la mejor manera de entender su propósito es ver como se usa en dos o tres casos distintos.

Vamos a valernos de la **BDD_Spec** del Happy Day Scenario para intentar explicar un poco las cosas.

Las **BDD_Specs** definen el comportamiento que esperamos de nuestro código.

Nosotros las usaremos como abreviatura de `Context / Specification`, que es el término real que se usa en **BDD**.

> **NOTA**: El término **BDD_Spec** es una invención nuestra, para evitar la confusión que se podría producir al usar `Context / Specification` en su versión abreviada (**Specification** o **Spec**), teniendo en cuenta que también hay **Specifications** en **DDD** y tienen un sentido completamente diferente. Por esa razón cuando nos refiramos a **Spec** en el sentido **BDD** usaremos **BDD_Spec** y cuando nos refiramos a **Specification** en sentido **DDD** usaremos **DDD_Specification**.

Partiendo de la **BDD_Spec** del Happy Day Scenario:

```
When asked if an item that meets both conditions satisfies the or
operator specification
```

```
It should confirm that the or operator specification was satisfied.
It should evaluate the left side condition.
It should not evaluate the right side condition.
```

Podemos identificar su `Context`:

```
When asked if an item that meets both conditions satisfies the or
operator specification
```

y sus `Specifications`:

```
It should confirm that the or operator specification was satisfied.
It should evaluate the left side condition.
It should not evaluate the right side condition.
```

AIII.3.5 Equivalencia entre BDD_Spec formal y código que la implementa

Recordemos que definimos las **BDD_Specs** usando la vertiente `Context / Specification`. Por lo tanto la equivalencia entre las partes que la conforman son las siguientes.

`Context` de la **BDD_Spec**:

```
When asked if an item that meets both conditions satisfies the or
operator specification
```

`Context` del código que implementa la **BDD_Spec**:

```
public class when_asked_if_an_item_that_meets_both_conditions
             _satisfies_the_or_operator_specification :
             concern_for_the_or_specification
{ (....) }
```

`Specifications` de la **BDD_Spec**:

```
It should confirm that the or operator specification was satisfied.
It should evaluate the left side condition.
It should not evaluate the right side condition.
```

`Specifications` del código que implementa la **BDD_Spec**:

```
It should_confirm_that_the_or_operator_specification_was_satisfied = () => {};

It should_evaluate_the_left_side_condition = () => {};

It should_not_evaluate_the_right_side_condition = () => {};
```

Una vez entendido esto, podemos pasar a explicar más detalladamente qué es lo que hace el código de la **BDD_Spec**.

AIII.3.6 Bloque IT: It should confirm that the or operator specification was satisfied

Éste es el código que tratamos de explicar:

```
It should_confirm_that_the_or_operator_specification_was_satisfied = () =>
   result.ShouldBeTrue();
```

En este caso estamos diciendo que `result` debería ser `true`.

De esta forma confirmamos que la condición se ha cumplido.

AIII.3.7 Bloque IT: It should evaluate the left side condition

Éste es el código que tratamos de explicar:

```
It should_evaluate_the_left_side_condition = () =>
   left_side_specification
      .received(x => x
      .is_satisfied_by(the_item_that_meets_both_conditions));
```

En este caso estamos diciendo que `left_side_specification` debería recibir una llamada a su método `is_satisfied_by ()` con el argumento `the_item_that_meets_both_conditions`.

De esta forma confirmamos que los requisitos del `ISpecification<T>` del lado izquierdo se han comprobado.

AIII.3.8 Bloque IT: It should not evaluate the right side condition

Éste es el código que tratamos de explicar:

```
It should_not_evaluate_the_right_side_condition = () =>
   right_side_specification
      .never_received(x => x
      .is_satisfied_by(the_item_that_meets_both_conditions));
```

En este caso estamos diciendo que `right_side_specification` no debería recibir una llamada a su método `is_satisfied_by()` con el argumento `the_item_that_meets_both_conditions`.

Debemos entender que como el operando izquierdo (`left_side_specification`) es `true`, independientemente del valor del operando derecho (`right_side_specification`), la operación **OR** devuelve `true`.

La parte que nos interesa es la de que para confirmar la operación **OR**, al ser el operando izquierdo `true`, el derecho no debe evaluarse:

```
right_side_specification..never_received(...);
```

De ahí que usemos el método `.never_received()`, que sería el opuesto al usado en el anterior bloque `IT`, `.received()`.

AIII.3.9 Bloque BECAUSE

En este bloque es en el que vamos a detallar la razón por la cual se produce el comportamiento testado, es decir, es donde invocamos el System Under Test (SUT):

```
Because of = () =>
   result = sut.is_satisfied_by(the_item_that_meets_both_conditions);
```

Lo que estamos haciendo es guardar en `result` el resultado de invocar el método que queremos testar (`sut.is_satisfied_by()`) con sus parámetros adecuados (`the_item_that_meets_both_requirements`).

AIII.3.10 Bloque ESTABLISH

En este bloque es donde se encuentra toda la fontanería (creación de objetos, creación de mocks, establecimiento de los requisitos de los mocks,...). Es decir, toda la preparación de los objetos que juegan un papel en la **BDD_Spec**.

Vamos a dar dos versiones distintas.

AIII.3.10.1 Versión 1

Vamos primero con la **versión 1**:

```
VERSIÓN 1 - Errores Inconsistentes


Establish context = () =>
{
    left_side_specification = the_dependency<ISpecification<IWhateverType>>();
    right_side_specification = the_dependency<ISpecification<IWhateverType>>();
    the_item_that_meets_both_conditions = an<IWhateverType>();

    left_side_specification
        .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
        .Return(true);
    right_side_specification
        .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
        .Return(true);
};
```

Vemos que tenemos dos bloques:

En el **primer bloque** simplemente creamos todos los objetos que necesitamos a través de las factorías que nos provee las extensiones `developwithpassion` de `machine.specifications`:

- `the_dependency<T>()`: Crea un mock del tipo especificado por `T` y además lo

inyecta al constructor cuando se crea el System Under Test (SUT).

- `an<T>()`: Crea un stub / mock del tipo especificado por `T`.

Por lo tanto, en nuestro caso, creamos dos mocks para inyectar en el constructor del SUT:

- `left_side_specification`
- `right_side_specification`

y un stub de propósito general:

- `the_item_that_meets_both_conditions`.

Es decir:

```
left_side_specification = the_dependency<ISpecification<IWhateverType>>();
right_side_specification = the_dependency<ISpecification<IWhateverType>>();
the_item_that_meets_both_conditions = an<IWhateverType>();
```

En el **segundo bloque** establecemos el comportamiento esperado de los dos mocks creados anteriormente:

- `left_side_specification`
- `right_side_specification`

obligándolos a que, cuando reciban una llamada al método `is_satisfied_by()` con el argumento `the_item_that_meets_both_conditions`, devuelvan el valor `true`:

```
left_side_specification
   .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
   .Return(true);
right_side_specification
    .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
    .Return(true);
```

AIII.3.10.2 Problemas de la versión 1

El problema que hay con esta versión, es un problema de uso de `machine.specifications.developwithpassion` y solamente es aplicable a escenarios en los que el constructor del SUT recibe dos o más argumentos del mismo tipo.

En muestro caso, recibe dos argumentos del tipo `ISpecification<T>`:

```
public OrSpecification(ISpecification<T> left_side_specification,
                       ISpecification<T> right_side_specification)
```

La razón por la cual esto es un problema, parece venir derivada de la implementación interna de `the_dependency<T>()` que funciona tal cual se espera mientras no se de esa condición de los argumentos del mismo tipo.

El error que produce es realmente complejo de trazar, ya que, en ningún momento lanza una excepción o un mensaje de aviso, sino que de vez en cuando los tests fallan.

Este tipo de comportamiento es algo de lo que no nos debemos sorprender y es un pequeño precio que debemos pagar por usar (y muchas veces, abusar de) herramientas tan experimentales como ésta.

Hemos decidido mantener en este documento las dos versiones ya que creemos que desde el punto de vista didáctico, deberían ayudar a entender mejor los conceptos aquí explicados, además de servir de advertencia en caso de encontrar errores de este estilo (inconsistencia en el resultado de los tests), pues no debemos descartar que estemos usando mal la herramienta.

A continuación explicamos la Versión 2, que es la correcta.

AIII.3.10.3 Versión 2

Vamos ahora con la **versión 2**:

```
VERSIÓN 2 - Correcta
```

```
Establish context = () =>
{
    left_side_specification = an<ISpecification<IWhateverType>>();
    right_side_specification = an<ISpecification<IWhateverType>>();
    the_item_that_meets_both_conditions = an<IWhateverType>();

    left_side_specification
        .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
        .Return(false);
    right_side_specification
        .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
        .Return(true);

    create_sut_using(() => new OrSpecification<IWhateverType>(
                                left_side_specification,
                                right_side_specification));
};
```

En este caso vemos que está dividido en tres bloques:

Al igual que en la **versión 1**, en el **primer bloque** simplemente creamos todos los objetos que necesitamos a través de las factorías que nos provee las extensiones `developwithpassion` de `machine.specifications`:

- `an<T>()`: Crea un stub / mock del tipo especificado por `T`.

Por lo tanto creamos tres stubs / mocks:

- `left_side_specification`
- `right_side_specification`
- `the_item_that_meets_both_conditions`

En el código:

```
left_side_specification = an<ISpecification<IWhateverType>>();
right_side_specification = an<ISpecification<IWhateverType>>();
the_item_that_meets_both_conditions = an<IWhateverType>();
```

En el **segundo bloque** establecemos el comportamiento esperado de dos de los mocks creados anteriormente:

- `left_side_specification`
- `right_side_specification`

ya que:

- `the_item_that_meets_both_conditions`

no va a tener comportamiento asociado (no se va a producir ninguna llamada a ninguno de sus métodos o propiedades, comportándose por tanto como un stub).

Por lo tanto estamos obligando a los dos mocks a que, cuando reciban una llamada al método `is_satisfied_by()` con el argumento `the_item_that_meets_both_conditions`, devuelva el valor `true`:

```
left_side_specification
   .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
   .Return(true);
right_side_specification
    .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
    .Return(true);
```

Este **segundo bloque** es exactamente igual al de la **versión 1**.

La única diferencia existente es que en el **primer bloque** hemos creado estos elementos a través de un método diferente.

- En la **versión 1**:
  - Dos llamadas a `the_dependency<T>()`
  - Una llamada a `an<T>()`

- En la **versión 2**:
  - Tres llamadas a `an<T>()`

En el **tercer bloque** (inexistente en la **versión 1**) es donde enjugamos las diferencias del **primer bloque** en ambas versiones.

En la **versión 1**, al hacer uso de `the_dependency<T>()`, no solo creamos el mock sino que además le indicamos al framework **BDD** que inyecte estos campos al constructor del SUT (con los problemas asociados detallados anteriormente).

En la **versión 2** necesitamos, todavía, inyectarle esos parámetros al constructor del SUT, y eso es precisamente lo que hacemos. Usamos `create_sut_using()` y como parámetro le pasamos la invocación explicita del constructor (pero esta vez sin los problemas a los que hacíamos referencia unas lineas más arriba):

```
create_sut_using(() => new OrSpecification<IWhateverType>(
                               left_side_specification,
                               right_side_specification));
```

AIII.3.11 El código completo de la BDD_Spec

Ésta sería la **BDD_Spec** completa con todos los elementos que entran en juego:

```
public class concern_for_the_or_specification :
             Observes<ISpecification<IWhateverType>,
                      OrSpecification<IWhateverType>> {}

public class when_asked_if_an_item_that_meets_both_conditions
             _satisfies_the_or_operator_specification :
             concern_for_the_or_specification
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_both_conditions = an<IWhateverType>();

        left_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
           .Return(true);
        right_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
           .Return(true);

        create_sut_using(() => new OrSpecification<IWhateverType>(
                                      left_side_specification,
                                      right_side_specification));
    };

    Because of = () =>
       result = sut.is_satisfied_by(the_item_that_meets_both_conditions);
```

```
    It should_confirm_that_the_or_operator_specification_was_satisfied = () =>
       result.ShouldBeTrue();

    It should_evaluate_the_left_side_condition = () =>
       left_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_both_conditions));

    It should_not_evaluate_the_right_side_condition = () =>
       right_side_specification
          .never_received(x => x
          .is_satisfied_by(the_item_that_meets_both_conditions));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_both_conditions;
}
```

AIII.3.11.1 La interface IWhateverType

A mayores hemos creado la interface `IWhateverType`, que simula el comportamiento de cualquier interface que definamos (algo así como `Object` pero en versión interface):

```
namespace dddsample.specs.domain.shared
{
    public interface IWhateverType {}
}
```

AIII.3.11.2 La clase base concern_for_the_or_specification

La **clase abstracta** `concern_for_the_or_specification`, proporciona a nuestras **BDD_Specs** un contenedor donde poder alojar el código común a todas ellas.

Hereda de `Observes<Contract, ClassUnderTest>`, que es una clase que nos proporciona las extensiones `developwithpassion` de `machine.specifications`.

Si atendemos al código:

```
public class concern_for_the_or_specification :
             Observes<ISpecification<IWhateverType>,
                      OrSpecification<IWhateverType>> {}
```

podemos identificar:

- `Contract` como `ISpecification<IWhateverType>`.
- `ClassUnderTest` como `OrSpecification<IWhateverType>`.

Por lo tanto:

- Nuestro SUT va a ser de tipo `ISpecification<IWhateverType>`.
- La implementación de nuestro SUT es `OrSpecification<IWhateverType>`.

AIII.3.12 Escribiendo el código que nos permite satisfacer la BDD_Spec

Una vez escrito el código que describe lo que esperamos del escenario especificado por la **BDD_Spec**, debemos escribir la aplicación en sí, es decir, la clase `OrSpecification<T>`.

Es importante destacar que el orden en el que hacemos las cosas es importante.

Así, primeramente debemos escribir el código de la **BDD_Spec** (tal y como hemos hecho), para a continuación escribir el código de la aplicación (que es lo que vamos a hacer ahora).

Si nos centramos primeramente en los dos últimos bloque `IT`:

```
It should_evaluate_the_left_side_condition = () =>
   left_side_specification
      .received(x => x
      .is_satisfied_by(the_item_that_meets_both_conditions));

It should_not_evaluate_the_right_side_condition = () =>
   right_side_specification
      .never_received(x =>x
      .is_satisfied_by(the_item_that_meets_both_conditions));
```

vemos que necesitamos que ocurra:

```
left_side_specification.is_satisfied_by(item);
```

pero que no ocurra:

```
right_side_specification.is_satisfied_by(item);
```

y que además tanto `left_side_specification` como `right_side_specification` deben provenir del constructor de la clase (vía **Constructor Dependency Injection**):

```
public OrSpecification(ISpecification<T> left_side_specification,
                       ISpecification<T> right_side_specification)
```

Es decir, con este código que se muestra a continuación conseguiríamos que los dos bloques `IT` compilasen y además cumpliesen los requisitos:

```
public class OrSpecification<T> : ISpecification<T>
{
    ISpecification<T> left_side_specification;
    ISpecification<T> right_side_specification;

    public OrSpecification(ISpecification<T> left_side_specification,
                           ISpecification<T> right_side_specification)
    {
        this.left_side_specification = left_side_specification;
        this.right_side_specification = right_side_specification;
    }

    public bool is_satisfied_by(T item)
    {
        this.left_side_specification.is_satisfied_by(item);
        return false;
    }
}
```

Este paso es importante desde un punto de vista pedagógico, ya que, claramente la clase no hace lo que queremos (devuelve siempre `false`, sean cuales sean las dos **DDD_Specifications** y el `item` que le pasamos y además no evalúa jamas la expresión de la derecha), pero sin embargo cumplimos dos de los tres bloques `IT`.

Esto debería ayudarnos a que abramos los ojos y empecemos a entender un poco mejor todo lo que hay por debajo del **BDD**.

Para conseguir satisfacer las 3 condiciones expuestas en los bloques `IT`, nos falta:

```
It should_confirm_that_the_or_operator_specification_was_satisfied = () =>
   result.ShouldBeTrue();
```

Pero tiene fácil arreglo, basta con devolver `true`, en vez de `false`:

```
public bool is_satisfied_by(T item)
{
    this.left_side_specification.is_satisfied_by(item);
    return true;
}
```

De esta forma satisfacemos las 3 condiciones de los bloques `IT`.

AIII.3.12.1 La evaluación del operando lógico OR

La pregunta ahora sería la siguiente, ¿el método `is_satisfied_by(T item)` de la clase `OrSpecification<T>` hace lo que queremos que haga?

La respuesta es que no, ya que queremos que haga una evaluación del operando lógico **OR**, y claramente no lo hace ya que evalúa únicamente uno de los operandos:

```
this.left_side_specification.is_satisfied_by(item);
```

y aun por encima, ni siquiera lo tiene en cuenta a la hora de devolver el valor correcto, pues siempre devuelve `true`:

```
return true;
```

independientemente de cual haya sido el resultado de la evaluación previa.

La cuestión aquí, es que sí que cumple los requisitos de nuestra **BDD_Spec** (los 3 bloques `IT`).

¿Que es lo que ocurre entonces?

Pues que necesitamos programar las otras **BDD_Specs** que quedan:

```
When asked if an item that meets only the left side condition satisfies
the or operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should not evaluate the right side condition.

When asked if an item that meets only the right side condition
satisfies the or operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.

When asked if an item that does not meet any of the conditions
satisfies the or operator specification
   It should confirm that the or operator specification was not
satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.
```

para obligar a nuestro código a hacer lo que queremos (que evalúe la operación lógica **OR**).

AIII.3.12.2 El sentido común

En un escenario de programación normal, lo lógico sería haber escrito ya en este punto:

```
public bool is_satisfied_by(T item)
{
    return this.left_side_specification.is_satisfied_by(item) ||
           this.right_side_specification.is_satisfied_by(item);
}
```

que hace lo que queremos, y a continuación escribir el código de las **BDD_Specs** restantes y comprobar que efectivamente funciona.

Sin embargo es muy útil la filosofía que hay detrás de hacer las cosas tal y como hemos ido detallando hasta ahora, y es que el principio

> "*escribir el mínimo código necesario para satisfacer las condiciones establecidas por nuestro test*"

del TDD, es muy útil cuando nos enfrentamos a escenarios mucho más complejos que el tratado aquí.

AIII.3.13 Implementando los tres escenarios restantes

Sin entrar en el nivel de detalle de la explicación del Happy Day Scenario, los tres escenarios restantes que complementan al mismo, quedan definidos a través de las **BDD_Specs**:

```
When asked if an item that meets only the left side condition satisfies
the or operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should not evaluate the right side condition.

When asked if an item that meets only the right side condition
satisfies the or operator specification
   It should confirm that the or operator specification was satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.

When asked if an item that does not meet any of the conditions
satisfies the or operator specification
   It should confirm that the or operator specification was not
satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.
```

Siendo éste el código que las implementa:

```
public class when_asked_if_an_item_that_meets_only_the_left_side_condition
             _satisfies_the_or_operator_specification :
             concern_for_the_or_specification
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_the_left_side_condition = an<IWhateverType>();

        left_side_specification
           .Stub(x => x.is_satisfied_by(
                          the_item_that_meets_the_left_side_condition))
```

```
            .Return(true);
        right_side_specification
            .Stub(x => x.is_satisfied_by(
                            the_item_that_meets_the_left_side_condition))
            .Return(false);

        create_sut_using(() => new OrSpecification<IWhateverType>(
                                        left_side_specification,
                                        right_side_specification));
    };

    Because of = () =>
        result = sut.is_satisfied_by(
                        the_item_that_meets_the_left_side_condition);

    It should_confirm_that_the_or_operator_specification_was_satisfied = () =>
        result.ShouldBeTrue();

    It should_evaluate_the_left_side_condition = () =>
        left_side_specification
            .received(x => x
            .is_satisfied_by(the_item_that_meets_the_left_side_condition));

    It should_not_evaluate_the_right_side_condition = () =>
        right_side_specification
            .never_received(x => x
            .is_satisfied_by(the_item_that_meets_the_left_side_condition));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_the_left_side_condition;
}

public class when_asked_if_an_item_that_meets_only_the_right_side_condition
             _satisfies_the_or_operator_specification :
             concern_for_the_or_specification
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_the_right_side_condition = an<IWhateverType>();

        left_side_specification
            .Stub(x => x.is_satisfied_by(
                            the_item_that_meets_the_right_side_condition))
            .Return(false);
        right_side_specification
            .Stub(x => x.is_satisfied_by(
                            the_item_that_meets_the_right_side_condition))
            .Return(true);
```

```
        create_sut_using(() => new OrSpecification<IWhateverType>(
                                    left_side_specification,
                                    right_side_specification));
    };

    Because of = () =>
       result = sut.is_satisfied_by(
                       the_item_that_meets_the_right_side_condition);

    It should_confirm_that_the_or_operator_specification_was_satisfied = () =>
       result.ShouldBeTrue();

    It should_evaluate_the_left_side_condition = () =>
       left_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_the_right_side_condition));

    It should_evaluate_the_right_side_condition = () =>
       right_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_the_right_side_condition));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_the_right_side_condition;
}

public class when_asked_if_an_item_that_does_not_meet_any_of_the_conditions
             _satisfies_the_or_operator_specification :
             concern_for_the_or_specification
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_no_condition = an<IWhateverType>();

        left_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_no_condition))
           .Return(false);
        right_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_no_condition))
           .Return(false);

        create_sut_using(() => new OrSpecification<IWhateverType>(
                                    left_side_specification,
                                    right_side_specification));
    };

    Because of = () =>
```

```
        result = sut.is_satisfied_by(the_item_that_meets_no_condition);

    It should_confirm_that_the_or_operator_specification
        _was_not_satisfied = () =>
        result.ShouldBeFalse();

    It should_evaluate_the_left_side_condition = () =>
        left_side_specification
            .received(x => x
            .is_satisfied_by(the_item_that_meets_no_condition));

    It should_evaluate_the_right_side_condition = () =>
        right_side_specification
            .received(x => x
            .is_satisfied_by(the_item_that_meets_no_condition));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_no_condition;
}
```

Lo que obligaría a nuestra clase `OrSpecification` a hacer lo correcto:

```
public class OrSpecification<T> : ISpecification<T>
{
    ISpecification<T> left_side_specification;
    ISpecification<T> right_side_specification;

    public OrSpecification(ISpecification<T> left_side_specification,
                           ISpecification<T> right_side_specification)
    {
        this.left_side_specification = left_side_specification;
        this.right_side_specification = right_side_specification;
    }

    public bool is_satisfied_by(T item)
    {
        return this.left_side_specification.is_satisfied_by(item) ||
               this.right_side_specification.is_satisfied_by(item);
    }
}
```

Como lo demuestra la salida de haber ejecutado las **BDD_Specs** con **TestDriven.NET**:

```
  ------ Test started: Assembly: dddsample.specs.dll ------

  when asked if an item that meets both conditions satisfies the or operator
  specification
```

```
» should confirm that the or operator specification was satisfied
» should evaluate the left side condition
» should not evaluate the right side condition

when asked if an item that meets only the left side condition satisfies
the or operator specification
» should confirm that the or operator specification was satisfied
» should evaluate the left side condition
» should not evaluate the right side condition

when asked if an item that meets only the right side condition satisfies
the or operator specification
» should confirm that the or operator specification was satisfied
» should evaluate the left side condition
» should evaluate the right side condition

when asked if an item that does not meet any of the conditions satisfies
the or operator specification
» should confirm that the or operator specification was not satisfied
» should evaluate the left side condition
» should evaluate the right side condition


12 passed, 0 failed, 0 skipped, took 0,65 seconds (Machine.Specifications
0.3.0).
```

Y con esto habríamos completado nuestro objetivo ya que quedaría implementada la clase `OrSpecification<T>`, guiados por **BDD**.

Recordemos que hemos eliminando la necesidad de usar la **clase base abstracta** `Specification<T>` y que además, gracias a la aplicación de conceptos como **Dependency Injection**, hemos podido reemplazar las dependencias de una clase con mocks, facilitándonos la tarea de testar la clase `OrSpecification<T>` completamente aislada de sus dependencias y colaboradores.

AIII.4 NotSpecification<T> vía BDD

Para este caso no nos detendremos ni entraremos en el mismo nivel de detalle que en el de `OrSpecification<T>`, puesto que, muchos de los conceptos son similares.

Aun así, demostraremos como, efectivamente, la interface de la versión java era incorrecta.

AIII.4.1 Las BDD_Specs

Nuestras **BDD_Specs** para la clase `NotSpecification<T>` serían:

```
When asked if an item that meets the condition satisfies the not
operation specification
   It should confirm that the original condition has been negated.
   It should evaluate the condition.

When asked if an item that does not meet the condition satisfies the
not operation specification
   It should confirm that the original condition has been negated.
   It should evaluate the condition.
```

El código que implementa ambas **BDD_Specs**:

```
public class concern_for_the_not_specification :
             Observes<ISpecification<IWhateverType>,
                      NotSpecification<IWhateverType>> {}

public class when_asked_if_an_item_that_meets_the_condition
             _satisfies_the_not_operation_specification :
             concern_for_the_not_specification
{
    Establish context = () =>
    {
        the_to_negate_specification =
           the_dependency<ISpecification<IWhateverType>>();
        the_item_that_meets_the_condition = an<IWhateverType>();

        the_to_negate_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_the_condition))
           .Return(true);
    };
```

```
    Because of = () =>
       result = sut.is_satisfied_by(the_item_that_meets_the_condition);

    It should_confirm_that_the_original_condition_has_been_negated = () =>
       result.ShouldBeFalse();

    It should_evaluate_the_condition = () =>
       the_to_negate_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_the_condition));

    static bool result;
    static IWhateverType the_item_that_meets_the_condition;
    static ISpecification<IWhateverType> the_to_negate_specification;
}

public class when_asked_if_an_item_that_does_not_meet_the_condition
             _satisfies_the_not_operation_specification :
             concern_for_the_not_specification
{
    Establish context = () =>
    {
        the_to_negate_specification =
           the_dependency<ISpecification<IWhateverType>>();
        the_item_that_does_not_meet_the_condition = an<IWhateverType>();

        the_to_negate_specification
           .Stub(x => x.is_satisfied_by(
                           the_item_that_does_not_meet_the_condition))
           .Return(false);
    };

    Because of = () =>
       result = sut.is_satisfied_by(the_item_that_does_not_meet_the_condition);

    It should_confirm_that_the_original_condition_has_been_negated = () =>
       result.ShouldBeTrue();

    It should_evaluate_the_condition = () =>
       the_to_negate_specification
          .received(x => x
          .is_satisfied_by(the_item_that_does_not_meet_the_condition));

    static bool result;
    static IWhateverType the_item_that_does_not_meet_the_condition;
    static ISpecification<IWhateverType> the_to_negate_specification;
}
```

La mecánica vuelve a ser la misma que habíamos empleado para la **BDD_Spec** de `OrSpecification<T>`, solo que en este caso solo entra en juego una sola `ISpecification<T>` y no dos (queda probado por tanto que la interface de la versión

java probablemente sea errónea), y además lo que debemos probar es que convierte `true` en `false` y viceversa.

Nuestra clase `NotSpecification<T>` asociada a esta **BDD_Spec** es:

```
public class NotSpecification<T> : ISpecification<T>
{
    ISpecification<T> to_negate_specification;

    public NotSpecification(ISpecification<T> to_negate_specification)
    {
        this.to_negate_specification = to_negate_specification;
    }

    public bool is_satisfied_by(T item)
    {
        return !this.to_negate_specification.is_satisfied_by(item);
    }
}
```

Y todo funciona según lo previsto tal y como demuestra el log de **TestDriven.NET**:

```
------ Test started: Assembly: dddsample.specs.dll ------

when asked if an item that meets the condition satisfies the not operation
specification
» should confirm that the original condition has been negated
» should evaluate the condition

when asked if an item that does not meet the condition satisfies the not
operation specification
» should confirm that the original condition has been negated
» should evaluate the condition


4 passed, 0 failed, 0 skipped, took 0,62 seconds (Machine.Specifications
0.3.0).
```

AIII.5 AndSpecification<T> vía BDD

Finalmente reproducimos las **BDD_Specs** para la clase `AndSpecification<T>`, para de esta forma completar la primera parte del patrón **DDD_Specification** y poder proceder con la segunda parte.

Es un calco de `OrSpecification<T>` pero realizando la función lógica **AND** en vez de **OR**.

A estas alturas, después de haber navegado por los entresijos de la implementación de `NotSpecification<T>` y sobre todo de `OrSpecification<T>`, no debería plantear ningún reto entenderlo.

Nuestras **BDD_Specs** para la clase `AndSpecification<T>` serían:

```
When asked if an item that meets both conditions satisfies the and
operator specification
   It should confirm that the and operator specification was satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.

When asked if an item that meets only the left side condition satisfies
the and operator specification
   It should confirm that the and operator specification was not
satisfied.
   It should evaluate the left side condition.
   It should evaluate the right side condition.

When asked if an item that meets only the right side condition
satisfies the and operator specification
   It should confirm that the and operator specification was not
satisfied.
   It should evaluate the left side condition.
   It should not evaluate the right side condition.

When asked if an item that does not meet any of the conditions
satisfies the and operator specification
   It should confirm that the and operator specification was not
satisfied.
   It should evaluate the left side condition.
   It should not evaluate the right side condition.
```

Es interesante fijarse, que con respecto a `OrSpecification<T>` cambian los casos en los

que no se evalúa el operando derecho, ya que:

```
False AND CualquierCosa = False
```

La implementación de las **BDD_Specs**:

```
public class concern_for_the_and_specification :
             Observes<ISpecification<IWhateverType>,
                      AndSpecification<IWhateverType>> {}

public class when_asked_if_an_item_that_meets_both_conditions
             _satisfies_the_and_operator_specification :
             concern_for_the_and_specification
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_both_conditions = an<IWhateverType>();

        left_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
           .Return(true);
        right_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_both_conditions))
           .Return(true);

        create_sut_using(() => new AndSpecification<IWhateverType>(
                                     left_side_specification,
                                     right_side_specification));
    };

    Because of = () => result =
       sut.is_satisfied_by(the_item_that_meets_both_conditions);

    It should_confirm_that_the_and_operator_specification_was_satisfied = () =>
       result.ShouldBeTrue();

    It should_evaluate_the_left_side_condition = () =>
       left_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_both_conditions));

    It should_evaluate_the_right_side_condition = () =>
       right_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_both_conditions));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
```

```
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_both_conditions;
}

public class when_asked_if_an_item_that_meets_only_the_left_side_condition
             _satisfies_the_and_operator_specification :
             concern_for_the_and_specification
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_the_left_side_condition = an<IWhateverType>();

        left_side_specification
           .Stub(x => x.is_satisfied_by(
                           the_item_that_meets_the_left_side_condition))
           .Return(true);
        right_side_specification
           .Stub(x => x.is_satisfied_by(
                           the_item_that_meets_the_left_side_condition))
           .Return(false);

        create_sut_using(() => new AndSpecification<IWhateverType>(
                                      left_side_specification,
                                      right_side_specification));
    };

    Because of = () =>
        result = sut.is_satisfied_by(
                        the_item_that_meets_the_left_side_condition);

    It should_confirm_that_the_and_operator_specification
       _was_not_satisfied = () =>
       result.ShouldBeFalse();

    It should_evaluate_the_left_side_condition = () =>
       left_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_the_left_side_condition));

    It should_evaluate_the_right_side_condition = () =>
       right_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_the_left_side_condition));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_the_left_side_condition;
}
```

```
public class when_asked_if_an_item_that_meets_only_the_right_side_condition
             _satisfies_the_and_operator_specification :
             concern_for_the_and_specification
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_the_right_side_condition = an<IWhateverType>();

        left_side_specification
           .Stub(x => x.is_satisfied_by(
                           the_item_that_meets_the_right_side_condition))
           .Return(false);
        right_side_specification
           .Stub(x => x.is_satisfied_by(
                           the_item_that_meets_the_right_side_condition))
           .Return(true);

        create_sut_using(() => new AndSpecification<IWhateverType>(
                                      left_side_specification,
                                      right_side_specification));
    };

    Because of = () =>
       result = sut.is_satisfied_by(
                       the_item_that_meets_the_right_side_condition);

    It should_confirm_that_the_and_operator_specification
       _was_not_satisfied = () =>
       result.ShouldBeFalse();

    It should_evaluate_the_left_side_condition = () =>
       left_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_the_right_side_condition));

    It should_not_evaluate_the_right_side_condition = () =>
       right_side_specification
          .never_received(x => x
          .is_satisfied_by(the_item_that_meets_the_right_side_condition));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_the_right_side_condition;
}

public class when_asked_if_an_item_that_does_not_meet_any_of_the_conditions
             _satisfies_the_and_operator_specification :
             concern_for_the_and_specification
{
```

```
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
        the_item_that_meets_no_condition = an<IWhateverType>();

        left_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_no_condition))
           .Return(false);
        right_side_specification
           .Stub(x => x.is_satisfied_by(the_item_that_meets_no_condition))
           .Return(false);

        create_sut_using(() => new AndSpecification<IWhateverType>(
                                    left_side_specification,
                                    right_side_specification));
    };

    Because of = () =>
       result = sut.is_satisfied_by(the_item_that_meets_no_condition);

    It should_confirm_that_the_and_operator_specification
       _was_not_satisfied = () =>
       result.ShouldBeFalse();

    It should_evaluate_the_left_side_condition = () =>
       left_side_specification
          .received(x => x
          .is_satisfied_by(the_item_that_meets_no_condition));

    It should_not_evaluate_the_right_side_condition = () =>
       right_side_specification
          .never_received(x => x
          .is_satisfied_by(the_item_that_meets_no_condition));

    static bool result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
    static IWhateverType the_item_that_meets_no_condition;
}
```

La clase que hace que se cumplan las **BDD_Specs**:

```
public class AndSpecification<T> : ISpecification<T>
{
    ISpecification<T> left_side_specification;
    ISpecification<T> right_side_specification;

    public AndSpecification(ISpecification<T> left_side_specification,
                            ISpecification<T> right_side_specification)
    {
```

```
        this.left_side_specification = left_side_specification;
        this.right_side_specification = right_side_specification;
    }

    public bool is_satisfied_by(T item)
    {
        return this.left_side_specification.is_satisfied_by(item) &&
               this.right_side_specification.is_satisfied_by(item);
    }
}
```

AIII.6 Recapitulación de lo conseguido.

Vamos a listar a continuación lo que ya hemos hecho, para de esa forma poder entender mejor las consecuencias y además poder tener una imagen más clara de que es lo que nos falta para poder finalizar la implementación del **DDD_Specification**.

AIII.6.1 BDD y diseño

Hemos conseguido eliminar completamente la necesidad de la **clase base abstracta** `Specification<T>`, que dificultaba la aplicación de la Práctica de Diseño Agile **BDD**.

Hemos simplificado la interface `ISpecification<T>`, ganando en expresividad e intención, y sin perder por ello funcionalidad.

Hemos aislado la implementación de las clases `AndSpecification<T>`, `OrSpecification<T>` y `NotSpecification<T>` y les hemos aplicado **BDD**.

AIII.6.2 S.O.L.I.D.

Hemos aplicado tanto el **SRP** (**Single Responsability Principle**) como el **OCP** (**Open-Closed Principle**) al desarrollo realizado hasta ahora (aunque no hayamos hecho especial hincapié en ello).

El **SRP** lo hemos aplicado, por ejemplo, al haber despojado de responsabilidades extra a la interface `ISpecification<T>`, de forma que ahora, `ISpecification<T>` se ocupa de una sola cosa (**tiene una sola razón para cambiar**), saber si un determinado `item` cumple o no una `ISpecification<T>` (`is_satisfied_by(T item)`).

El **OCP** lo hemos aplicado al haber sacado de dentro de la **clase abstracta** `Specification<T>` las clases `AndSpecification<T>`, `OrSpecification<T>` y `NotSpecification<T>`, ya que antes si queríamos añadir nuevas clases como estas tres nombradas, nos veíamos en la obligación de editar la **clase abstracta** `Specification<T>`. Esta clase no cumplía el **OCP** ya que era muy difícil que cumpliese lo de **Abierta a la Extensión / Cerrada a la Modificación**, pues para extender nos veíamos en la obligación

de modificar.

La situación ahora es bien distinta ya que al haber prescindido de la **clase abstracta** `Specification<T>`, y haber extraído las clases `AndSpecification<T>`, `OrSpecification<T>` y `NotSpecification<T>`, podemos extender (creando una nueva clase que implemente `ISpecification<T>`) sin necesidad de editar una clase ya existente.

AIII.7 Lo que todavia nos queda por hacer

Una vez listados los logros debemos ver lo que nos falta.

AIII.7.1 Encapsular la construcción

Todavía necesitamos encapsular la creación de las clases `AndSpecification<T>`, `OrSpecification<T>`, `NotSpecification<T>` para disponer de los métodos asociados a `ISpecification<T>`:

```
public ISpecification<T> and(ISpecification<T> the_other)
public ISpecification<T> or(ISpecification<T> the_other)
public ISpecification<T> not()
```

ya que antes se ocupaba de ello la **clase abstracta** `Specification<T>`, y en algún sitio necesitamos poner los:

```
new AndSpecification<T>(this, the_other);
new OrSpecification<T>(this, the_other);
new NotSpecification<T>(this);
```

AIII.8 La clase SpecificationExtensions

AIII.8.1 Introducción a la implementación de and<T>()

Nos guste más o menos, en algún sitio tenemos que proceder con la creación de la clase `AndSpecification<T>` para así poder "empaquetarla" a través de la operación / método `and<T>()`.

Recordemos que precisamente esta creación era la que hacía que nuestras dependencias fuesen opacas en la difunta **clase abstracta** `Specification<T>`.

La mala noticia es que vamos a tener que usar una **dependencia opaca** en oposición a la **Constructor Dependency Injection**, con lo que ya podemos despedirnos del tratamiento **BDD** en su vertiente Mockista en ese caso concreto.

La buena noticia es que vamos a limitar el rango de este pequeño desaguisado desde el punto de vista del **BDD** (hay miles de programadores que utilizan **dependencias opacas** todos los días sin inmutarse) a una sola linea de código, y aun por encima, haremos un poco de magia para que el método `and()` **se materialice**, como si tuviésemos una **clase base**.

AIII.8.2 BDD_Spec para and<T>()

AIII.8.2.1 Creando SpecificationExtensions

Puesto que sabemos que a continuación vamos a crear también los métodos `or<T>()` y `not<T>()`, y que para ello vamos a utilizar **Extension Methods** (misterio resuelto, estos eran los polvos mágicos), hemos decidido crear una clase llamada `SpecificationExtensions`.

De esta forma, dotamos a la solución de un contenedor para nuestros **Extension Methods**, referentes a la implementación del **DDD_Specification**:

```
static public class SpecificationExtensions
```

```
{
    static public ISpecification<T> and<T>(this ISpecification<T> left_side,
                                           ISpecification<T> right_side){}
}
```

AIII.8.2.2 La BDD_Spec

Vamos con la **BDD_Spec** que debería resolver una buena parte de las dudas:

```
When using the and specification extension method
   It should return an and specification type.
```

Somos perfectamente conscientes de que no podemos comprobar directamente que estamos creando una `AndSpecification<T>`, pero sin embargo, nos las vamos a arreglar para al menos saber que nuestro **Extension Method** devuelve una instancia del tipo que queremos.

Aun así este planteamiento **indirecto** no es la panacea, ya que podremos consultar el tipo de la instancia, pero no, el orden de los argumentos con los que llamamos al constructor de `AndSpecification<T>`.

Pero eso es algo que veremos en unos minutos.

AIII.8.2.3 El bloque IT

La versión en código de la **BDD_Spec** relativa al bloque `IT` es la siguiente:

```
public class when_using_the_and_specification_extension_method :
             concern_for_the_specification_extensions
{
    It should_return_an_and_specification_type = () =>
       result.ShouldBeAn<AndSpecification<IWhateverType>>();

    static ISpecification<IWhateverType> result;
}
```

Sencillo de entender, ya que lo único que asertamos es que nuestro campo `result`, definido como `ISpecification<IWhateverType>`, de todas las posibilidades que tiene como `ISpecification`, a saber:

- `AndSpecification`
- `OrSpecification`
- `NotSpecification`

Va a ser del tipo `AndSpecification<IWhateverType>`.

Eso es exactamente lo que estamos asertando.

AIII.8.2.4 El bloque BECAUSE

El bloque `BECAUSE` ejercita lo que queremos testar, en este caso, la llamada al **Extension Method**.

Podemos hacerlo de tres formas distintas.

Podemos invocar la clase `static` con todos sus argumentos:

```
Because of = () =>
   result = SpecificationExtensions
               .and<IWhateverType>(left_side_specification,
                                   right_side_specification);
```

Podemos invocar la clase `static` con el mínimo numero de argumentos:

```
Because of = () =>
   result = SpecificationExtensions
               .and(left_side_specification,
                    right_side_specification);
```

Podemos usar el **Extension Method**:

```
Because of = () =>
   result = left_side_specification
               .and(right_side_specification);
```

Cualquiera de estas formas sería valida.

Sin embargo, nos parece que si estamos tomándonos todas las molestias para crear los **Extension Methods**, deberíamos usarlos.

Por tanto, nos declaramos a favor de la tercera versión:

Así quedaría el bloque `BECAUSE` con los campos necesarios para que compile:

```
Because of = () =>
   result = left_side_specification
               .and(right_side_specification);

static ISpecification<IWhateverType> result;
static ISpecification<IWhateverType> left_side_specification;
static ISpecification<IWhateverType> right_side_specification;
```

AIII.8.2.5 El bloque ESTABLISH

Para preparar el contexto usamos el bloque `ESTABLISH`, que en este caso se limita a declarar dos mocks para:

- `left_side_specification`
- `right_side_specification`

```
Establish context = () =>
{
    left_side_specification = an<ISpecification<IWhateverType>>();
    right_side_specification = an<ISpecification<IWhateverType>>();
};
```

AIII.8.2.6 El código completo de la BDD_Spec

Tendríamos entonces la **BDD_Spec** completa:

```
public class when_using_the_and_specification_extension_method :
             concern_for_the_specification_extensions
{
    Establish context = () =>
    {
        left_side_specification = an<ISpecification<IWhateverType>>();
        right_side_specification = an<ISpecification<IWhateverType>>();
    };
```

```
    Because of = () =>
       result = left_side_specification.and(right_side_specification);

    It should_return_an_and_specification_type = () =>
       result.ShouldBeAn<AndSpecification<IWhateverType>>();

    static ISpecification<IWhateverType> result;
    static ISpecification<IWhateverType> left_side_specification;
    static ISpecification<IWhateverType> right_side_specification;
}
```

Que creemos muy sencilla de entender y muy clara en su intención.

AIII.8.2.7 Haciendo que falle la BDD_Spec

El siguiente paso sería ejecutar la **BDD_Spec** y hacer que falle de una forma controlada. Para ello necesitamos que al menos el **esqueleto** de la clase `SpecificationExtensions` se encuentre resuelto.

Con esto llegaría:

```
static public class SpecificationExtensions
{
    static public ISpecification<T> and<T>(this ISpecification<T> left_side,
                                           ISpecification<T> right_side)
    {
        throw new NotImplementedException();
    }
}
```

Ahora ya podemos ejecutar la **BDD_Spec**:

```
   ------ Test started: Assembly: dddsample.specs.dll ------

   when using the and specification extension method
   » should return an and specification type (FAIL)

   Test 'should return an and specification type' failed:
         System.Reflection.TargetInvocationException: Exception has been
   thrown by the target of an invocation. --->
   System.NotImplementedException: The method or operation is not
   implemented.

   (....)
```

```
0 passed, 1 failed, 0 skipped, took 0,77 seconds (Machine.Specifications
0.3.0).
```

Ha ocurrido lo que esperábamos, que ha sido lanzada una excepción del tipo:

```
Exception has been thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.
```

AIII.8.2.8 Haciendo que pase la BDD_Spec

Ahora es cuando escribiríamos el código necesario para que la **BDD_Spec** pase.

En este caso es necesaria una sola linea:

```
static public class SpecificationExtensions
{
    static public ISpecification<T> and<T>(this ISpecification<T> left_side,
                                           ISpecification<T> right_side)
    {
        return new AndSpecification<T>(left_side, right_side);
    }
}
```

Podemos ejecutar de nuevo la **BDD_Spec**. Pero esta vez esperamos que no de errores:

```
------ Test started: Assembly: dddsample.specs.dll ------

when using the and specification extension method
» should return an and specification type


1 passed, 0 failed, 0 skipped, took 0,62 seconds (Machine.Specifications
0.3.0).
```

Todo correcto.

AIII.8.2.9 Reafirmando el fallo significativo de la BDD_Spec

Podemos comprobar que obtenemos un error significativo, si en vez de asertar que

esperamos una `AndSpecification`, esperamos una `OrSpecification`.

Para ello basta con cambiar una linea del bloque `IT`.

Ésta:

```
It should_return_an_and_specification_type = () =>
      result.ShouldBeAn<AndSpecification<IWhateverType>>();
```

Por ésta otra:

```
It should_return_an_and_specification_type = () =>
      result.ShouldBeAn<OrSpecification<IWhateverType>>();
```

Si ejecutamos la **BDD_Spec** ahora, deberíamos obtener un error que nos diga algo así como que:

```
"...esperaba una OrSpecification pero en realidad me he encontrado con una
AndSpecification"
```

Vamos a comprobarlo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when using the and specification extension method
» should return an and specification type (FAIL)

Test 'should return an and specification type' failed:
       Machine.Specifications.SpecificationException: Should be of type
dddsample.domain.shared.OrSpecification`1[dddsample.specs.domain.shared.IW
hateverType] but is of type
dddsample.domain.shared.AndSpecification`1[dddsample.specs.domain.shared.I
WhateverType]
       at
Machine.Specifications.ShouldExtensionMethods.ShouldBeOfType(Object
actual, Type expected)
       at Machine.Specifications.ShouldExtensionMethods.ShouldBeOfType[T]
(Object actual)
       at Machine.Specifications.ShouldExtensionMethods.ShouldBeAn[T]
(Object item)
       domain\shared\SpecificationExtensionsSpecs.cs(19,0): at
dddsample.specs.domain.shared.when_using_the_and_specification_extension_m
ethod.<.ctor>b__2()
       at
```

```
   Machine.Specifications.Model.Specification.InvokeSpecificationField()
         at Machine.Specifications.Model.Specification.Verify()

   0 passed, 1 failed, 0 skipped, took 0,64 seconds (Machine.Specifications
   0.3.0).
```

Ahí lo tenemos (en versión compacta):

```
   Should be of type OrSpecification`1[IWhateverType]
   but is of type AndSpecification`1[IWhateverType]
```

Volvemos a cambiar nuestro bloque `IT` a su versión original:

```
It should_return_an_and_specification_type = () =>
      result.ShouldBeAn<AndSpecification<IWhateverType>>();
```

AIII.8.2.10 Limitaciones

Ya hemos visto que nuestro test es capaz de distinguir entre las distintas implementaciones de `ISpecification<T>`, pero todavía no hemos visualizado la limitación acerca del orden con el que invocamos los argumentos.

Para ello vamos a cambiar la siguiente linea en la implementación de `and<T>()`:

```
static public ISpecification<T> and<T>(this ISpecification<T> left_side,
                                       ISpecification<T> right_side)
{
    return new AndSpecification<T>(left_side, right_side);
}
```

por esta otra:

```
static public ISpecification<T> and<T>(this ISpecification<T> left_side,
                                       ISpecification<T> right_side)
{
    return new AndSpecification<T>(right_side, left_side);
}
```

Y ejecutamos la **BDD_Spec**:

```
   ------ Test started: Assembly: dddsample.specs.dll ------
```

```
when using the and specification extension method
» should return an and specification type


1 passed, 0 failed, 0 skipped, took 0,62 seconds (Machine.Specifications
0.3.0).
```

Comprobamos, por lo tanto, que no hace distinción.

Ésta era la limitación de la que hablábamos.

Sin embargo, siendo pragmáticos, esta limitación es mínima, ya que al menos estamos comprobando que devuelve el tipo correcto.

Antes de finalizar, debemos volver a cambiar el orden de los argumentos para que haga lo que deseamos. Así que:

```
return new AndSpecification<T>(left_side, right_side);
```

Y ejecutamos nuevamente la **BDD_Spec**, para comprobar que todo sigue bien.

AIII.8.3 BDD_Spec para or<T>()

Es similar a nuestra anterior **BDD_Spec** para `and<T>()`.

Así que únicamente dejaremos definida formalmente la **BDD_Spec** que define su comportamiento y mostraremos el código al que llegamos.

En caso de necesitar consultar el código de la **BDD_Spec**, podemos dirigirnos directamente al código fuente.

La **BDD_Spec**:

```
When using the or specification extension method
   It should return an or specification type.
```

Y el código que nos permite pasar el test:

```
static public class SpecificationExtensions
{
    static public ISpecification<T> or<T>(this ISpecification<T> left_side,
                                          ISpecification<T> right_side)
    {
        return new OrSpecification<T>(left_side, right_side);
    }
}
```

AIII.8.4 BDD_Spec para not<T>()

La **BDD_Spec** que define el comportamiento de `not<T>()` es muy sencilla:

```
When using the not specification extension method
   It should return a not specification type.
```

Su implementación en código es similar a las dos anteriores, pero con la particularidad de que en este caso solo hay un argumento en lugar de dos (`and<T>()` y `or<T>()` necesitan dos cada uno):

```
public class when_using_the_not_specification_extension_method :
             concern_for_the_specification_extensions
{
    Establish context = () =>
       to_negate_specification = an<ISpecification<IWhateverType>>();

    Because of = () => result = to_negate_specification.not();

    It should_return_a_not_specification_type = () =>
       result.ShouldBeAn<NotSpecification<IWhateverType>>();

    static ISpecification<IWhateverType> result;
    static ISpecification<IWhateverType> to_negate_specification;
}
```

Destacar que ahora para invocar al **Extension Method**, necesitamos únicamente un solo argumento (el que lo invoca).

Si ejecutamos la **BDD_Spec** con el **esqueleto** de la implementación de `not<T>()`:

```
static public class SpecificationExtensions
```

```
{
    static public ISpecification<T> not<T>(this ISpecification<T> to_negate)
    {
        throw new NotImplementedException();
    }
}
```

Obtendremos nuestro fallo:

```
------ Test started: Assembly: dddsample.specs.dll ------

when using the not specification extension method
» should return a not specification type (FAIL)

Test 'should return a not specification type' failed:
      System.Reflection.TargetInvocationException: Exception has been
thrown by the target of an invocation. --->
System.NotImplementedException: The method or operation is not
implemented.

(....)


0 passed, 1 failed, 0 skipped, took 0,70 seconds (Machine.Specifications
0.3.0).
```

Y con la siguiente linea:

```
static public class SpecificationExtensions
{
    static public ISpecification<T> not<T>(this ISpecification<T> to_negate)
    {
        return new NotSpecification<T>(to_negate);
    }
}
```

Haremos que pase:

```
------ Test started: Assembly: dddsample.specs.dll ------

when using the not specification extension method
» should return a not specification type


1 passed, 0 failed, 0 skipped, took 0,68 seconds (Machine.Specifications
0.3.0).
```

Y con esto, habremos finalizado la implementación del patrón **DDD_Specification** guiados por **BDD**.

AIII.8.5 SpecificationExtensions al completo

```
static public class SpecificationExtensions
{
    static public ISpecification<T> and<T>(this ISpecification<T> left_side,
                                           ISpecification<T> right_side)
    {
        return new AndSpecification<T>(left_side, right_side);
    }

    static public ISpecification<T> or<T>(this ISpecification<T> left_side,
                                          ISpecification<T> right_side)
    {
        return new OrSpecification<T>(left_side, right_side);
    }

    static public ISpecification<T> not<T>(this ISpecification<T> to_negate)
    {
        return new NotSpecification<T>(to_negate);
    }
}
```

Apéndice IV: Primeros pasos con GIT

AIV.1 Configuración inicial

Una vez hemos instalado nuestro cliente **git**, necesitaremos abrir una **ventana de comandos git** y teclear las siguientes órdenes.

AIV.1.1 Configuración inicial

Empezaremos por definir nuestro **nombre de usuario** y nuestro **email** de forma global, afectando, por tanto, a todos nuestros futuros proyectos **git**:

```
git config --global user.name "tu nombre"
git config --global user.email "tu-email@algo.com"
```

AIV.1.2 Inicializando el repositorio

Para **inicializar el repositorio** en nuestra máquina, creamos el directorio donde queremos que vaya nuestro proyecto y a continuación le damos la configuración básica en cuanto a **git** se refiere:

```
mkdir dddsample
git init
```

AIV.1.3 Haciendo nuestro primer commit añadiendo el fichero .gitignore

Vamos a añadir nuestro primer fichero al **repositorio git**.

Lo lógico es que ese primer fichero sea el **.gitignore**:

```
*resharper.user
[Dd]ebug/
[Rr]elease/
build/
[Bb]in/
[Oo]bj/
*.suo
*.sln.cache
_ReSharper.*/
*.user
```

que como vemos contiene los filtros de los tipos de ficheros que no queremos que **git** añada.

Para ello, necesitamos realizar un proceso de dos pasos.

Primero añadimos el fichero al sistema de **tracking** de **git**:

```
git add -A
```

Una vez que **git** sepa que el fichero existe, podemos realizar nuestro primer **commit** con su comentario asociado:

```
git commit -m "First Commit – Added the .gitignore file"
```

Suele ser útil comprobar que todo ha ido bien con un:

```
git status
```

AIV.1.4 Preparando la comunicación con el repositorio remoto en github

En cuanto hayamos creado una cuenta en **github**, ya dispondremos de un usuario (en nuestro caso **3j**).

A continuación crearemos el **repositorio remoto** de forma que obtengamos una dirección **git** para el mismo (en nuestro caso **dddsample.git**).

Ahora ya podemos configurar en nuestra máquina dichos datos asignándole un **alias** (**origin**):

```
git remote add origin git@github.com:3j/dddsample.git
git remote
```

AIV.1.5 Enviando los datos al repositorio remoto en github

Para sincronizar los datos de nuestra máquina con los del **repositorio github**, debemos

enviar los datos:

```
git push origin master
```

Ahora el **repositorio de github** ya contiene el fichero **.gitignore** que añadimos anteriormente a nuestro **repositorio git local**.

AIV.2 El Check-In Dance

Ya tenemos nuestro repositorio funcionando en **github**. A partir de aquí, todo se limita a bailar correctamente el **Check-In Dance** y hacer uso de las **ramas** (**branches**) de una forma inteligente.

> **NOTA**: Nos referiremos a este proceso como **empaquetado** a lo largo de todo el documento, de forma que encontraremos multitud de veces la expresión "**empaquetamos y a github**". Una traducción más directa del término hubiese dado con la expresión "**bailamos y a github**", sin embargo, consideramos que en castellano suena mejor la primera que la segunda.

Detallamos a continuación este proceso.

> **NOTA**: Hay que tener en cuenta que al no ser un proyecto colaborativo, los pasos del **Check-In Dance** se reducen.

El **primer paso** consiste en comprobar el estado de nuestro **repositorio local** y a continuación bajarnos desde **github** la última versión de nuestro proyecto:

```
git status
git pull origin master
```

El **segundo paso** consiste en crear la **rama** en la que queremos trabajar.

```
git checkout -b dev
```

Hay que tener en cuenta, que al contrario que con Sistemas de Versiones No Distribuidos como **Subversion**, trabajar directamente en la **rama master**, no se considera una buena práctica. De ahí que hayamos creado la **rama dev**.

No viene mal consultar el estado del **repositorio** para ver que todo va bien:

```
git status
```

Antes de ir con el tercer paso, necesitamos generar nuevo contenido en nuestro proyecto.

Por ejemplo, podemos añadir la estructura inicial de carpetas así como sus ficheros principales:

- source
  - dddsample
    - dddsample.csproj
  - dddsample.specs
    - dddsample.specs.csproj
  - dddsample.sln

- thirdparty
  - machine.specifications
    - Machine.Specifications.DevelopWithPassion.dll
    - Machine.Specifications.DevelopWithPassion.Rhino.dll
    - Machine.Specifications.dll
    - Rhino.Mocks.dll

El **tercer paso** consiste en hacerle saber a **git** cuales son los ficheros nuevos que queremos que siga. Se puede hacer de forma individual, pero nosotros preferimos hacerlo de forma colectiva para todos los ficheros nuevos:

```
git add -A
```

Consultamos si el fichero **.gitignore** está realizando bien su trabajo con un:

```
git status
```

Funciona. Ninguno de los ficheros / carpetas del **ReSharper** se han añadido al **repositorio** (entre otros ficheros).

El **cuarto paso** consiste en salvar los cambios localmente.

Para ello:

```
git commit -m "Added source and thirdparty filders, sln file, a couple of
csproj files (dddsample and dddsample.specs) and finally the
machine.specifications files in order to start developing."
```

El **quinto paso** consiste en pasar los cambios del paso anterior de la **rama dev** a la **rama master**:

```
git checkout master
git merge dev
```

El **sexto paso** es **opcional**, y consiste en borrar la **rama dev**, ya que no la necesitamos más (y siempre podemos volver a crearla):

```
git branch -D dev
```

Comprobamos que efectivamente la **rama** ya no existe con:

```
git branch
```

El **séptimo y último paso**, consiste en enviar los cambios a nuestro **repositorio remoto** en **github**:

```
git push origin master
```

Con esto habríamos completado un ciclo del **Check-In Dance**.