Domain Primitives

Small Steps Towards Better Software

Albert Attard

albert.attard@thoughtworks.com

Agenda

Data Representation

Ambiguity

Security

Tight Coupling

Shortcomings

Data Representation

Order Number

Take the example of an order number for *Build-to-Order* items, represented as a 10-digit number:

```
0980810031
```

There is an exception for *Build-to-Stock* items, whose orders have a `-` as their second digit:

```
0-21200545
```

Representation

How can we represent this value in our code?

String Representation

We can represent the *Order Number* as a `String`:

```
data class Order(val orderNumber: String)
```

```
object OrderGateway {
  fun fetch(orderNumber: String) : Order { }
}
```

```
object OrderRepository {
  fun delete(orderNumber: String) { }
}
```

Challenges

However, while all *Order Number*s are `String`s, not all `String`s are *Order Number*s.

Only a very small subset of `String`s are *Order Number*s.

So the `String` data type is too generic.

There is something particular about *Order Number*s that we need to capture in our model.

An Example

For example, let's say we pass a random `String` to a function that requires an *Order Number*:

```
val order = Order("any random string will do")
```

```
val order = OrderGateway.fetch("any random string will do")
```

```
OrderRepository.delete("any random string will do")
```

The above code will compile, even though the provided `String` is not an *Order Number*.

A Possible Solution

We could add validation to ensure that the values passed are *Order Number*s.

Take this function for checking whether something is really an *Order Number*:

```
object OrderNumberValidation {
  fun isValid(orderNumber: String): Boolean { }

  @Throws(IllegalArgumentException::class)
  fun check(orderNumber: String): String {
    require (isValid(orderNumber)) { "Invalid order number" }
    return orderNumber
  }
}
```

Validation

We could use `init` block to check an argument before using it, and fail accordingly:

```
data class Order(val orderNumber: String) {
  init { OrderNumberValidation.check(orderNumber) }
}
```

We can use a similar approach for all the other usages.

```
object OrderGateway {
  fun fetch(orderNumber: String): Order {
    OrderNumberValidation.check(orderNumber)
    /* ... */
  }
}
```

Polluted Tests

A test needs to be added for every function that takes an *Order Number* as its input to make sure that these functions are failing as expected.

```
@Test
fun `should throw an exception if passed an invalid order number`() {
  assertFailsWith<IllegalArgumentException> {
    Order("a random string that it is not an Order Number")
  }
}
```

This will pollute the tests, because we need to make sure that inputs are properly checked by each recipient function.

The Forgotten Compiler

We are polluting the tests because we are not taking advantage of the compiler.

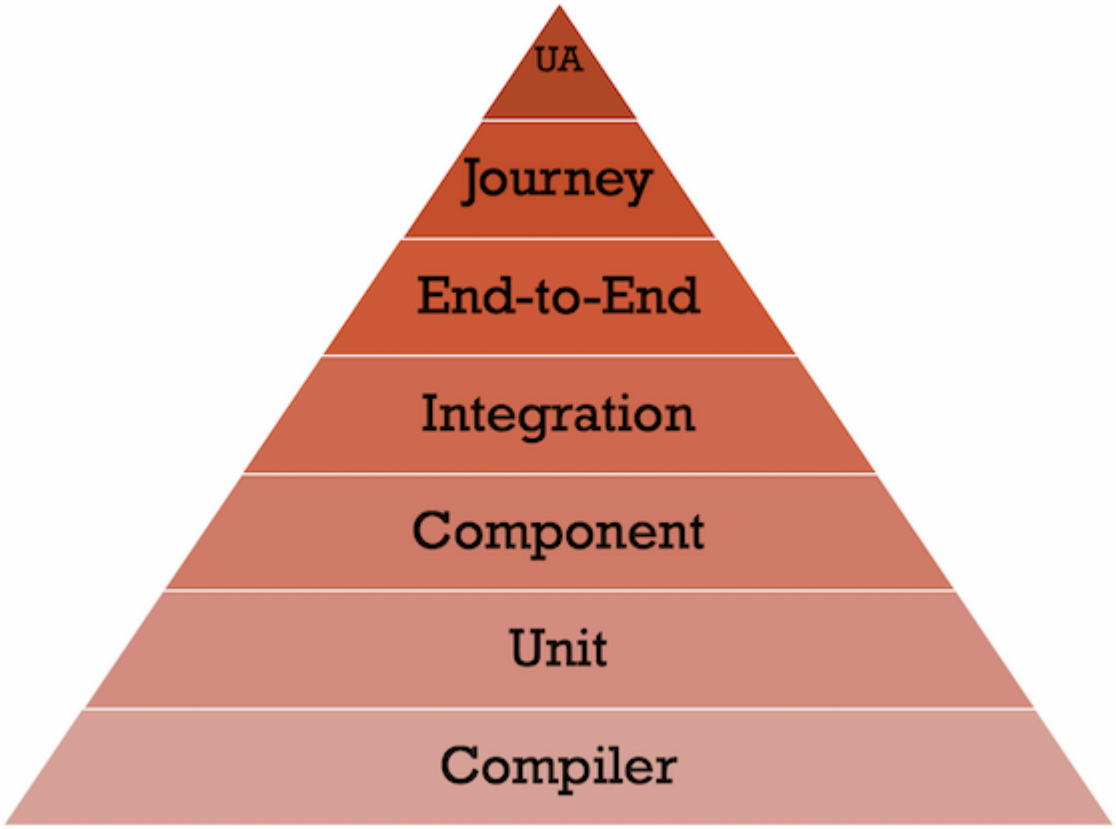
For example, if we pass an `Int` to a function that expects a `String`, the compiler will complain:

```
fun aFunctionThatTakeAString(string: String) {}

aFunctionThatTakeAString(42)
```

We don't need to have tests for these cases. The compiler will handle them.

Test Pyramid



Alternative Approach

Instead of using a *language primitive*, such as `String`, to represent our data types, we can use a *domain primitive*:

```
data class OrderNumber private constructor(val value: String) {
  companion object {
    @Throws(IllegalArgumentException::class)
    operator fun invoke(value: String): OrderNumber {
      require(value.length == 10) { "Invalid order number" }
      return OrderNumber(value)
    }
  }
}
```

Is this a *Value Object*?

Domain Primitives are sometimes referred to as *Value Objects*, but there are some important differences.

Value Objects are usually used to represent types that are not available as a *language primitive*, such as `Money` or `Address`.

While similar, *Domain Primitives* additionally ensure that all instances are valid values of that type, and also that types are not reused, especially between contexts.

For example, the `Name` *Domain Primitive* cannot be used to represent a person's name and a computer's name at the same time. In such a case, we would have two *Domain Primitives*: `PersonName` and `ComputerName`.

Use of Domain Primitives

Instead of `String`, we can now use `OrderNumber`:

```
data class Order(val orderNumber: OrderNumber)
```

```
object OrderGateway {
  fun fetch(orderNumber: OrderNumber) : Order { }
}
```

```
object OrderRepository {
  fun delete(orderNumber: OrderNumber) { }
}
```

Take Advantage of the Compiler

Now we cannot create an `Order` with any random `String`, nor pass it to any function that requires an `OrderNumber`:

```
val order = Order("any random string will not do")
println("$order")
```

The above will not compile.

Less Test Pollution

We now need only to make sure that only valid `OrderNumber`s can be created, and fail accordingly:

```
@Test
fun `should throw an exception when given an invalid order number`() {
  val invalidOrdersNumbers =
        listOf("", "too long to be a valid order number")
  invalidOrdersNumbers.forEach {
    assertFailsWith<IllegalArgumentException> { OrderNumber(it) }
  }
}
```

Sealed Classes

Another approach would be to use sealed classes instead of throwing exceptions:

```
sealed class OrderNumber {

  object Invalid : OrderNumber()

  data class Valid private constructor(val value: String) : OrderNumber(
    companion object {
      operator fun invoke(value: String): OrderNumber {
        return if(value.length != 10) Invalid
               else Valid(value)
      }
    }
  }
}
```

Streamlined Usage

Sealed classes are the preferred option, as these streamline usage:

```
when(OrderNumber("some random string")) {
  is OrderNumber.Invalid -> { /* Handle Invalid */ }
  is OrderNumber.Valid -> { /* Handle Valid */ }
}
```

Note: This example did not include a `companion object` in the `OrderNumber` class due to slide size constraints.

Ambiguity

To Err is Human

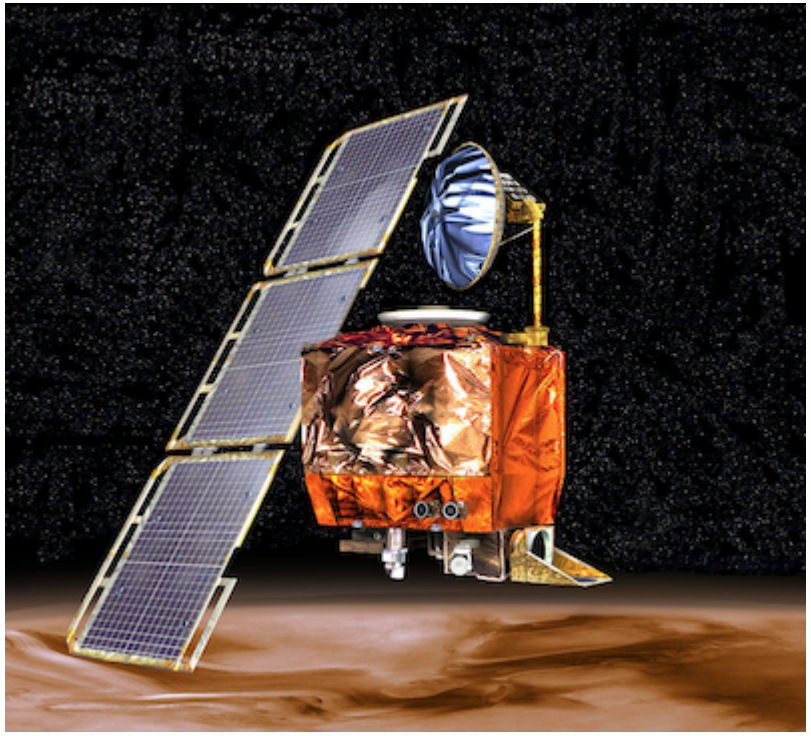
Air Canada Flight 143 ran out of fuel on July 23, 1983, at an altitude of 41,000 feet (12,000 m), midway through the flight.

The use of an incorrect conversion factor led to a total fuel load of only 22,300 pounds (10,100 kg), less than half of the 22,300 kg that was needed.

Fortunately, the crew was able to glide the Boeing 767 safely to an emergency landing.

Bigger Than We Think

NASA's Climate Orbiter was lost on September 23, 1999, due to an metric/imperial mishap.



A Simple Example

Let's say we have an air-conditioning controller that works with Celsius. It has the following function, which is used to control the power of the compressor:

```
fun adjustPower(celsius: Double) {  }
```

Say that the temperature is `18°C`, but by mistake the Fahrenheit equivalent is provided instead (`64.4°F`).

```
adjustPower(64.4) /* by mistake instead of 18 */
```

The controller will think that it's too hot, and will put the air conditioner to full power.

How Can We Mitigate Such Problems?

While we can easily convert between one temperature unit to another, we cannot tell units apart just by looking at the number.

Can we use an `enum` to identify the unit?:

```
enum class TemperatureUnit {
    CELSIUS, FAHRENHEIT
}

fun adjustPower(temperature: Double, unit: TemperatureUnit) {  }
```

Another Approach

Using `enum` will work, but we can do better.

```
sealed class Temperature {

  abstract fun toCelsius(): Celsius

  data class Celsius(val value: Double) : Temperature() { }

  data class Fahrenheit(val value: Double) : Temperature() { }
}
```

How Does This Work?

```
data class Celsius(val value: Double) : Temperature() {
  override fun toCelsius() =
    this
}
```

```
data class Fahrenheit(val value: Double) : Temperature() {
  override fun toCelsius() =
    Celsius((value - 32) * 5.0 / 9.0)
}
```

Improved Controller

The controller can now take any type of temperature unit, and safely convert it to the required type:

```
fun adjustPower(temperature: Temperature) {
  val celsius = temperature.toCelsius()
  /* Work with the proper temperature */
}
```

This will ensure that the controller always works with Celsius, irrespective of the temperature unit provided.

Beyond Conversions

The ambiguity problem goes beyond simple conversions.

Say we have a function that sets an order's delivery date:

```
fun dispatchOrderOn(a: Int, b: Int, c: Int) { }
```

This function takes the day of the month, the month, and the year as its parameters.

Buy just looking at the function's signature, can you tell which is the month parameter, and whether it is zero-based (where `0` represents January) or not?

Security

Leaking Sensitive Information

How many times have we printed a credit-card number, or other sensitive information, by mistake?

```
data class CreditCardNumber(val value: Long)

val number = CreditCardNumber(5555_5555_5555_5555)
println("Paying with: $number")
```

The example above will print the credit card number

```
Paying with: CreditCardNumber(value=5555555555555555)
```

How Can We Prevent This?

We could prevent the credit card number from being printed by overriding the `toString()` function:

```
data class CreditCardNumber(val value: Long) {

  private val maskedValue = "xxxx-xxxx-xxxx-${value % 10000}"

  override fun toString() =
    maskedValue
}
```

This time the credit card number is masked

```
Paying with: xxxx-xxxx-xxxx-5555
```

But What About …

But we can still print the credit card number by getting its value:

```
val number = CreditCardNumber(5555_5555_5555_5555)
println("Paying with: ${number.value}")
```

The above example will still print the credit card number.

```
Paying with: 5555555555555555
```

Can We Address This Somehow?

This is an area where domain primitives shine.

Let's say that **in our context**, the credit card number is only required to be read once, just to make the payment — so if the credit card number is read more than once, we should fail.

How Can We Do That?

```
class CreditCardNumber(value: Long) {

  private val consumed = AtomicBoolean()

  val value: Long = value
    get() =
      if (consumed.compareAndSet(false, true)) field
      else throw IllegalStateException(
            "Credit card number was already consumed"
      )

  /* Other properties and functions removed for brevity */
}
```

How Does This Work?

```
val number = CreditCardNumber(5555_5555_5555_5555)

/* The first time will work */
println("First try: ${number.value}")

/* The second time will throw an exception */
println("Second try: ${number.value}")
```

Any unexpected reads will not go unnoticed.

```
java.lang.IllegalStateException: Credit card number was already consumed
```

Is This a Silver Bullet?

No!!

We could always store the credit card number in a language primitive, such as `Long`, and then print this variable as many times we want to.

```
val number = CreditCardNumber(5555_5555_5555_5555)
val value = number.value

/* Now we can print the credit card number as many times we want */
repeat(100) {
  println("Credit card number: $value")
}
```

Tight Coupling

Coupling with Language Primitives

Say that we need to schedule some tasks to run every so often, like a cron job.

One way to do this would be to use the `java.util.Timer` and `java.util.TimerTask` classes:

```
val task = object : TimerTask() {
  override fun run() {
    println("Running...")
  }
}

val timer = Timer()
timer.scheduleAtFixedRate(task, 1000, 1000)
```

What's Wrong with That?

Without an abstraction layer between the language primitives and the application, swapping the `java.util.Timer` class could be harder than expected.

For example, the `java.util.Timer` will stop running if the `java.util.TimerTask` throws a `RuntimeException`.

That may be unexpected behaviour, and you would like to swap the `java.util.Timer` class to the `java.util.concurrent.ScheduledExecutorService` class.

With tight coupling between the application and the `java.util.Timer`, swapping may prove harder than expected.

Introducing an Abstraction Layer

Domain Primitives can act as an abstraction layer between the *Language Primitives*, such as the `java.util.Timer` class, and the rest of the application.

```
class CronJobTask { }

class CronJob {

  fun runAtFixRate(
      initialDelay: InitialDelay,
      delay: Delay,
      block: () -> Any
  ): CronJobTask { }
}
```

Simplifying Refactoring

Swapping the internals of the `CronJob` and `CronJobTask` classes should not affect any other part of the application.

Tests can ensure that these domain primitives are still behaving as expected, especially after swapping their internals.

Shortcomings

Verbosity

Domain primitives may reduce ambiguity and improve security, but at the cost of verbosity.

Consider a function that takes two integers:

```
original.slice(1, 2)
```

An alternative approach that uses domain primitives is quite a bit more verbose:

```
original.slice(Range(StartIndex(1), Length(2)))
```

Compatibility

Language primitives are compatible to other libraries, while domain primitives are not and need to be converted back and forth.

For example, we cannot save a domain primitive representing a person's name, such as `PersonName`, into a database. We need to get the `String` equivalent.

Class Explosion

The same domain primitive cannot be reused for different purposes, since each domain primitive should serve one purpose.

For example, *name* and *surname* should be represented by two domain primitives, such as `Name` and `Surname`, and not by one, generic, domain primitive, such as `GenericName`.

This may lead to class explosion as many classes are needed.

Thank You

Feedback makes us better

Please send any feedback to: albert.attard@thoughtworks.com


Free to make your mark
ThoughtWorks®
/careers