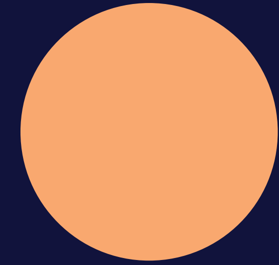
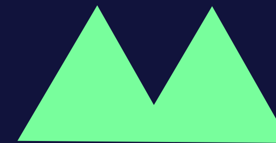
TRANSFORMATION PRIORITY PREMISE

Going through the steps.

Fredrik Wigsnes



bouvet



Uncle Bob

- Aka: Robert C. Martin
- Author of: Clean code, and more.
- Creator of TPP.

bouvet

The process

If we accept the Priority Premise, then we should amend the typical red-green-refactor process of TDD with the following provision:

- When passing a test, prefer higher priority transformations.
- When posing a test choose one that can be passed with higher priority transformations.
- When an implementation seems to require a low priority transformation, backtrack to see if there is a simpler test to pass.

bouvet

The Transformations

So what are these transformations? Perhaps we can make a list of them:

- ({}–>nil) no code at all->code that employs nil
- (nil->constant)
- (constant->constant+) a simple constant to a more complex constant
- (constant->scalar) replacing a constant with a variable or an argument
- (statement->statements) adding more unconditional statements.
- (unconditional->if) splitting the execution path
- (scalar->array)
- (array->container)
- (statement->recursion)
- (if->while)
- (expression->function) replacing an expression with a function or algorithm
- (variable->assignment) replacing the value of a variable.

{} ➔ nil

Tranform no code into something that returns a nil value.

```
function testEmptyCountryCodeReturnsNull() {
    assert(countryFromCode(null), null)
}

function countryFromCode(countryCode: string | null) {
    return null;
}
```

bouvet

nil → Constant

```
function testCountryFromCodeReturnsString() {
    assert(typeof countryFromCode("AU"), "string")
}

function countryFromCode(countryCode: string | null) {
    return "";
}
```

bouvet

Constant ➔ Constant+

A simple constant to a more complex constant.

```
function testCountryFromCodeReturnsAustralia() {
    assert(countryFromCode("AU"), "Australia")
}

function countryFromCode(countryCode: string | null)
{
    return "Australia";
}
```

bouvet

Constant → Scalar

Replacing a constant with a variable or an argument.

```
function countryFromCode(countryCode: string | null) {
    const country = "Australia"
    return country;
}

function getCountry(country: string) {
    return country;
}
```

bouvet

Statement → Statements

Adding more unconditional statements.

```
function testCountryFromCodeReturnsExactlyAustralia() {
    assert(countryFromCode("AU"), "Australia")
}

function countryFromCode(countryCode: string | null) {
    return getCountry("australia");
}

function getCountry(country: string) {
    return country?.substring(0, 1).toUpperCase()
        + country?.substring(1).toLowerCase();
}
```

bouvet

Unconditional → If

Splitting the execution path. It is also acceptable to use a ternary instead.

```
function testCountryFromCodeReturnsSpain() {
    assert(countryFromCode("ES"), "Spain")
}

function countryFromCode(countryCode: string | null) {
    if (countryCode === "ES")
        return "Spain";

    return "Australia";
}
```

bouvet

Scalar ➔ Array

```
function countryFromCode(countryCode: string | null) {
    const CountryCodes = ["AU", "ES"];
    const Countries = ["Australia", "Spain"];

    return Countries[CountryCodes.findIndex((code) => code === countryCode)];
}
```

bouvet

Array ➔ Container

This usually means converting a simple array into a more complex collection like a dictionary or array of objects.

```
function countryFromCode(countryCode: string | null) {
    const Countries = {"AU": "Australia", "ES": "Spain"};

    return countryCode && Countries[countryCode]
}
```

bouvet

Statement → Tail Recursion

Tail recursion means that the last instruction (usually the return value) is a recursive call.

```
function countryFromCode(countryCode: string | null, index=0): string | null {
    const CountryCodes = ["AU", "ES"];
    const Countries = ["Australia", "Spain"];

    if (index === Countries.length) return null;

    if (CountryCodes[index] === countryCode) return Countries[index]!;

    return countryFromCode(countryCode, index+1);
}
```

bouvet

If ➔ While

Convert a binary branching statement (if, unless, ternary, etc) into a loop (while, do, for, etc).

```
function countryFromCode(countryCode: string | null, index=0) {
    const Countries = [["AU", "Australia"], ["ES", "Spain"]];

    Countries.forEach((value) => {
        if (value[0] === countryCode) return value[1];
    });

    return null;
}
```

bouvet

Statement → Recursion

Non-tail recursion means that the last instruction (usually the return value) is not a recursive call.

```
function countryFromCode(countryCode: string | null, index=0): string | null {
    const Countries: string[][] = [["AU", "Australia"], ["ES", "Spain"], ["DK", "Denmark"]];

    let country: string | null = Countries[0]![1] ?? null

    if (Countries[index]![0] !== countryCode) {
        country = countryFromCode(countryCode, index+1)
    }

    return country;
}
```

bouvet

Expression → Function

Replacing an expression with a function or algorithm.

```
function getCountries() {
    return {"AU": "Australia", "ES": "Spain"};
}

function countryFromCode(countryCode: string | null) {
    const Countries = getCountries();

    return countryCode && Countries[countryCode]
}
```

bouvet

Variable ➔ Assignment

Replacing the value of a variable.

```
function testCountryFromCodeReturnsDenmark() {
    assert(countryFromCode("DK"), "Denmark")
}

function countryFromCode(countryCode: string | null) {
    let country = "Australia"

    if (countryCode === "ES") country = "Spain";
    if (countryCode === "DK") country = "Denmark";

    return country;
}
```

bouvet

Issues

There are a number of problems with this.

- Are there other transformations? (almost certainly)
- Are these the right transformations? (probably not)
- Are there better names for the transformations? (almost certainly)
- Is there really a priority? (I think so, but it might be more complicated than a simple ordinal sequence)
- If so, what is the principle behind that priority? (some notion of “complexity”)
- Can it be quantified? (I have no idea)
- Is the priority order presented in this blog correct? (not likely)
- The transformations as described are informal at best. Can they be formalized? (That’s the holy grail!)

Links

- http://blog.cleancoder.com/uncle-bob/2013/05/27/TheTransformationPriorityPremise.html
- https://elliotchance.medium.com/the-transformation-priority-premise-tpp-3e5dc08d445e

bouvet