Overview

- Why Mock Objects?
  - “Mockist” **vs.** “Classic” TDD
  - “Mockist” **and** “Classic” TDD
- Mocks and Smalltalk:
  - The Mocketry framework introduction
- Examples

Why Mock Objects?

Do we need Mocks at all (in Smalltalk)?

Public Opinion

- Smalltalk **vs.** Mock Objects?
  - Few/rare special cases
  - With mock you don’t test real thing
  - Use mocks for external objects only
  - Use other means to involve complex external objects
  - Speed up by other means

Public Opinion

…seems to be about **testing**

Smalltalk **and** Mock Objects

- “Mock Objects” is a TDD technique
  - … about **developing** systems
  - … **not** just testing
  - … useful in **all languages**
- **Smalltalk** makes mocks
  - much easier to use

Why Mock Objects?

- **Cut off dependencies** in tests
- Test-Driven **Decomposition**
  - Discover Responsibility (for collaborators)
  - Thinking vs. Speculating/Fantasizing

Seamless TDD

What Is The Problem?

- Dependencies
  - How to cut them off?
- Novel Collaborators
  - Where to cut?

What Is The problem?

- **Dependencies**
  - **How to cut them off?**
- Novel Collaborators
  - Where to cut?

Dependencies

We have:

- System Under Development (**SUD**)
- Collaborators
- Collaborators' collaborators …

Complex Test

So What?

We have to implement collaborators

- … without tests
- … loosing focus on SUD

Digression

Dependencies: Example

- **Mocks Aren't Stubs** by **Martin Fowler**
- Filling orders from warehouse

Filling Order

```
OrderTests >>
   testIsFilledIfEnoughInWarehouse
| order |
order:= Order on: 50 of: #product.
order fillFrom: warehouse.
self assert: order isFilled
```

Filling Order

```
OrderTests >>
testIsFilledIfEnoughInWarehouse
| order warehouse |
```

```
warehouse := Warehouse new.
“Put #product there”
“…but how?!”

order := Order on: 50 of: #product.
order fillFrom: warehouse.
```

…

Digression Detected!

- I develop **Order**
- I **don't want** to think about **Warehouse**

Filling Order

```
OrderTests >>
testIsFilledIfEnoughInWarehouse
| order warehouse |
warehouse := Warehouse new.
warehouse add: 50 of: #product.

order := Order on: 50 of: #prod.
order fillFrom: warehouse.
…
```

…And Even More **Digression**

Reduce amount of #product at the warehouse

```
test…
…
self assert:
   (warehouse
      amountOf: #product)
                        isZero
```

… And Even More Digression

Another test case:

If **there isn't enough** #product in the warehouse,

- **do not fill** order
- **do not remove** #product from warehouse

… Much More Digression

More complex test cases

Collaborators' logic becomes more and more complex…

This can engulf

Not-So-Seamless TDD

- SUD is **Order**
- Warehouse blures SUD
  - #add:of:
  - #amountOf:
- No explicit tests for Warehouse

Mocking Warehouse

```
OrderTests >>
  testIsFilledIfEnoughInWarehouse
    | order |
      order := Order on: 50 of: #product.
    [
       :warehouse|
       [order fillFrom: warehouse]
         should satisfy:
                    [“expectations”]
    ] runScenario.
    self assert: order isFilled
```

Mocking Warehouse

```
…
[:warehouse|
[order fillFrom: warehouse]
  should satisfy:
    [(warehouse
        has: 50 of: #product)
            willReturn: true.
     warehouse
        remove: 50 of: #product]
] runScenario.
…
```

The **Mocketry** Framework

Mock Objects in Smalltalk World

Behavior Expectations

When you do this with SUD,
expect that to happen
with collaborators

Collaborators are **mocked**

Behavior Expectations

Do this

Exercise

Expect that

Verify

Scenario

Mocketry Scenario Pattern

```
SomeTestCases >> testCase
[

    testScenario

] runScenario
```

Mocketry Scenario Pattern

```
SomeTestCases >> testCase
[
    [exercise]
        should strictly satisfy:
            [behaviorExpectations]
] runScenario
```

Behavior Expectations

- Just send mock objects the messages they should receive

```
warehouse
   has: 50 of: #product
```

- Specify their reaction

```
(warehouse
   has: 50 of: #product)
                  willReturn: true
```

Mocketry Scenario Pattern

```
SomeTestCases >> testCase
[
    [exercise] should strictly satisfy: [behaviorExpectations]

    [exercise] should strictly satisfy: [behaviorExpectations]

    … do anything
] runScenario
```

Mocketry Scenario Pattern

```
SomeTestCases >> testCase
[ :mock |
    [exercise] should strictly satisfy: [behaviorExpectations]
    [exercise] should strictly satisfy: [behaviorExpectations]
    … do anything
] runScenario
```

Mocketry Scenario Pattern

```
SomeTestCases >> testCase

[ :mock |

    [exercise] should strictly satisfy: [behaviorExpectations]

    [exercise] should strictly satisfy: [behaviorExpectations]

    … do anything

] runScenario
```

Mocketry Scenario Pattern

```
SomeTestCases >> testCase
[ :mock1 :mock2 :mock3 |
    [exercise] should strictly satisfy: [behaviorExpectations]
    [exercise] should strictly satisfy: [behaviorExpectations]
    … do anything
] runScenario
```

Trivial Example 1

```
TrueTests >>
        testDoesNotExecuteIfFalseBlock
[ :block |
        [true ifFalse: block ]
                shouldsatisfiy:
                        [“nothing expected”]
] runScenario
```

Trivial Example 2

```
TrueTests >>
        testExecutesIfTrueBlock
[ :block |
        [true ifTrue: block]
                should satisfiy:
                        [block value]
] runScenario
```

State Specification DSL

- resultObject **should** <expectation>
  - result should **be:** anotherObject
  - result should **equal:** anotherObject
  - …

Mocketry

- There is much more…
- Ask me
- …or **Dennis Kudryashov** (the Author)

Mocking Warehouse

```
OrderTests >>
testIsFilledIfEnoughInWarehouse
| order |
order := Order on: 50 of: #product.

[:warehouse|
   [order fillFrom: warehouse]
     should satisfy:
    [(warehouse has: 50 of: #product)
            willReturn: true.
     warehouse remove: 50 of: #product]
] runScenario.

self assert: order isFilled
```

Mocking Warehouse

```
OrderTests >>
testIsNotFilledIfNotEnoughInWarehouse
| order |
order := Order on: #amount of: #product.
[:warehouse|
   [order fillFrom: warehouse]
     should satisfy:
    [(warehouse has: 50 of: #product)
            willReturn: false.
     “Nothing else is expected” ]
] runScenario.
self assert: order isFilled
```

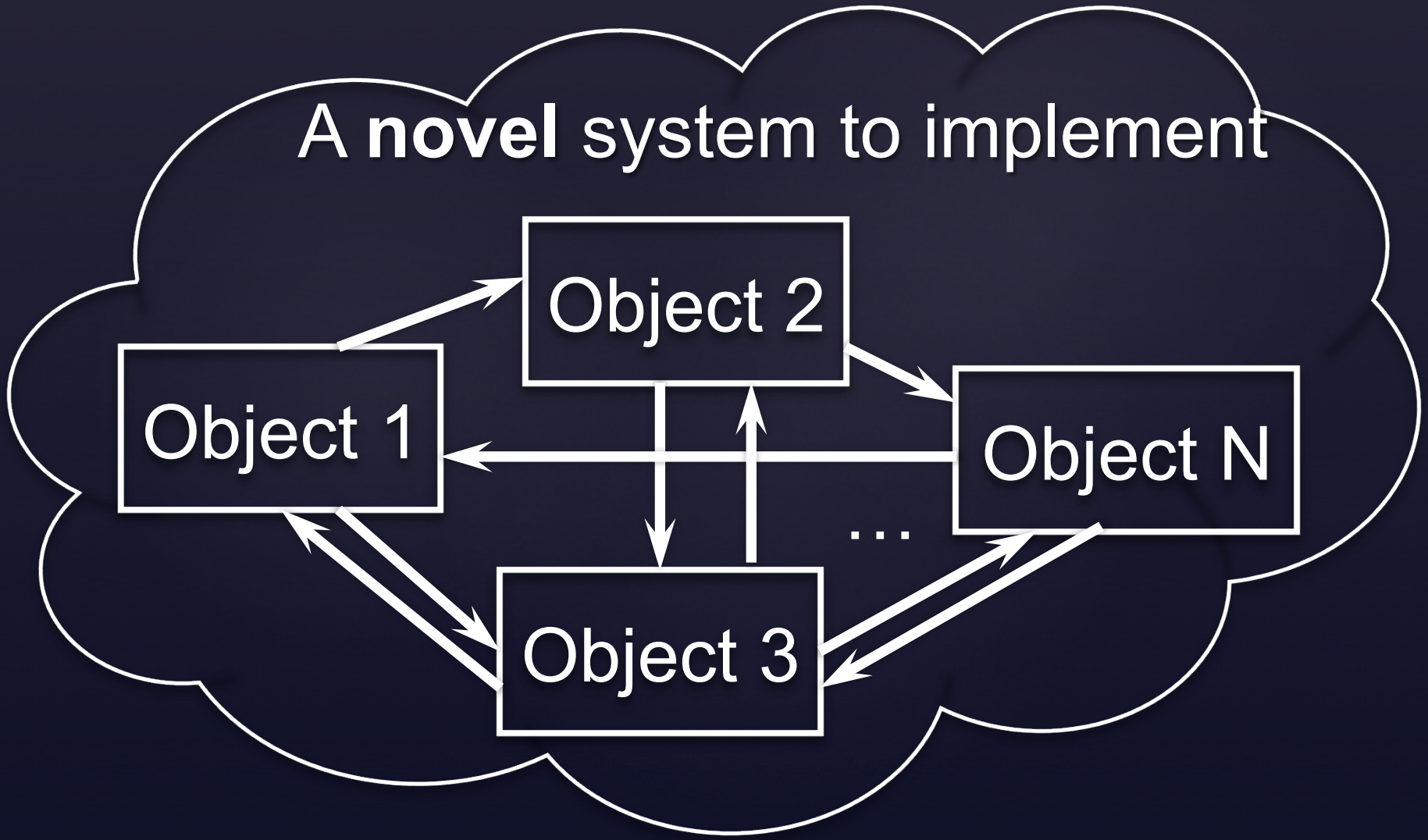
What We Get

- No need to implement Warehouse
- Just specify expectations
- … right in the test
- Focus on **the SUD**

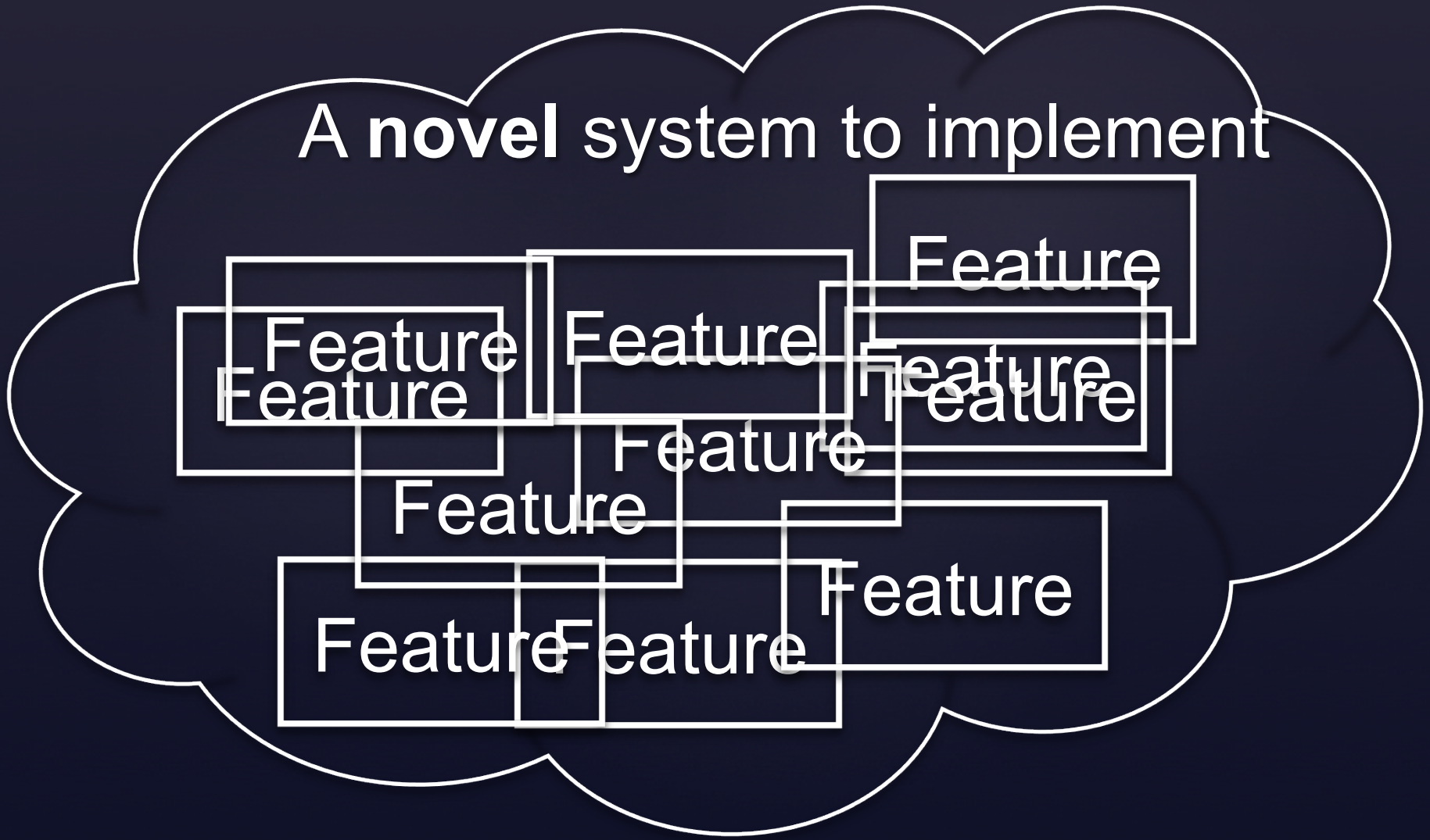
What's the problem?

- Dependencies
- **Novel Collaborators**

Where to Start?



Where to Cut?


A **novel** system to implement
Feature
Feature
Feature
Feature
Feature
Feature
Feature
Feature
Feature
Feature
Feature

Where to Start?

- Try to guess
  - … and be ready to abandon test(s)
  - … or get a mess

Or

- **Analyze** thoroughly
  - … up-front decomposition
  - … **without** tests — just a fantasy

Novel Collaborators: Example

Bulls and Cows Game

- Computer generates a secret key
  - e.g., a 4-digit number
- Human player tries to disclose it

Bulls and Cows Game

Scenario:

- User creates a game object
- **User starts the game**
  - **Game should generate a key**
- …

testGeneratesKeyOnStart

```
self assert: key …?
```

“How to represent key?!”

What Can I Do?

- Spontaneous representation
  - Do you feel lucky?
- Analyze thoroughly
  - Give up TDD
- Postpone the test
  - Not a solution

What Can I Do?

- …
- Create a new class for key
  - Unnecessary complexity?

What Can I Do?

That was a **Digression**!

testGeneratesKeyOnStart

```
|key|
game start.
key := game key.
self assert:
    key isKindOf: Code
```

testGeneratesKeyOnStart

```
[  :keyGen |
 game keyGenerator: keyGen.
 [ game start ]
   should satisfy:
     [keyGen createKey
        willReturn: #key]
 game key should be: #key
] runScenario.
```

What We Get

- Key generation **functionality**
  - is revealed
  - moved to another object
- Dependency Injection
  - fake key can be created for tests
  - KeyGenerator refactored to Turn
- No risk of incorrect decision

What We Get

Seamless TDD:

- No digression
- No up-front decomposition
- No up-front design
- No speculating / fantasizing

Complete Example

Mock Objects For Top-Down Design by Bulls-and-Cows Example

Just ask!

Classic vs. Mockist TDD

State vs. Behavior?
Result vs. Intention?

No contradiction!

Mockist approach
complements “classic” TDD

Classic and Mockist TDD

- Top-Down with Mockist TDD
  - Analysis and Decomposition
- Bottom-Up with Classic TDD
  - Synthesis and “real-object” testing