

Putting the 'D' in TDD

Or: Between Development And Design

Shai Yallin, Wix.com

Clean Code Days, Munich 2015



tl;dr

- Your job is not to write code
- TDD is not about testing

What's In It For You

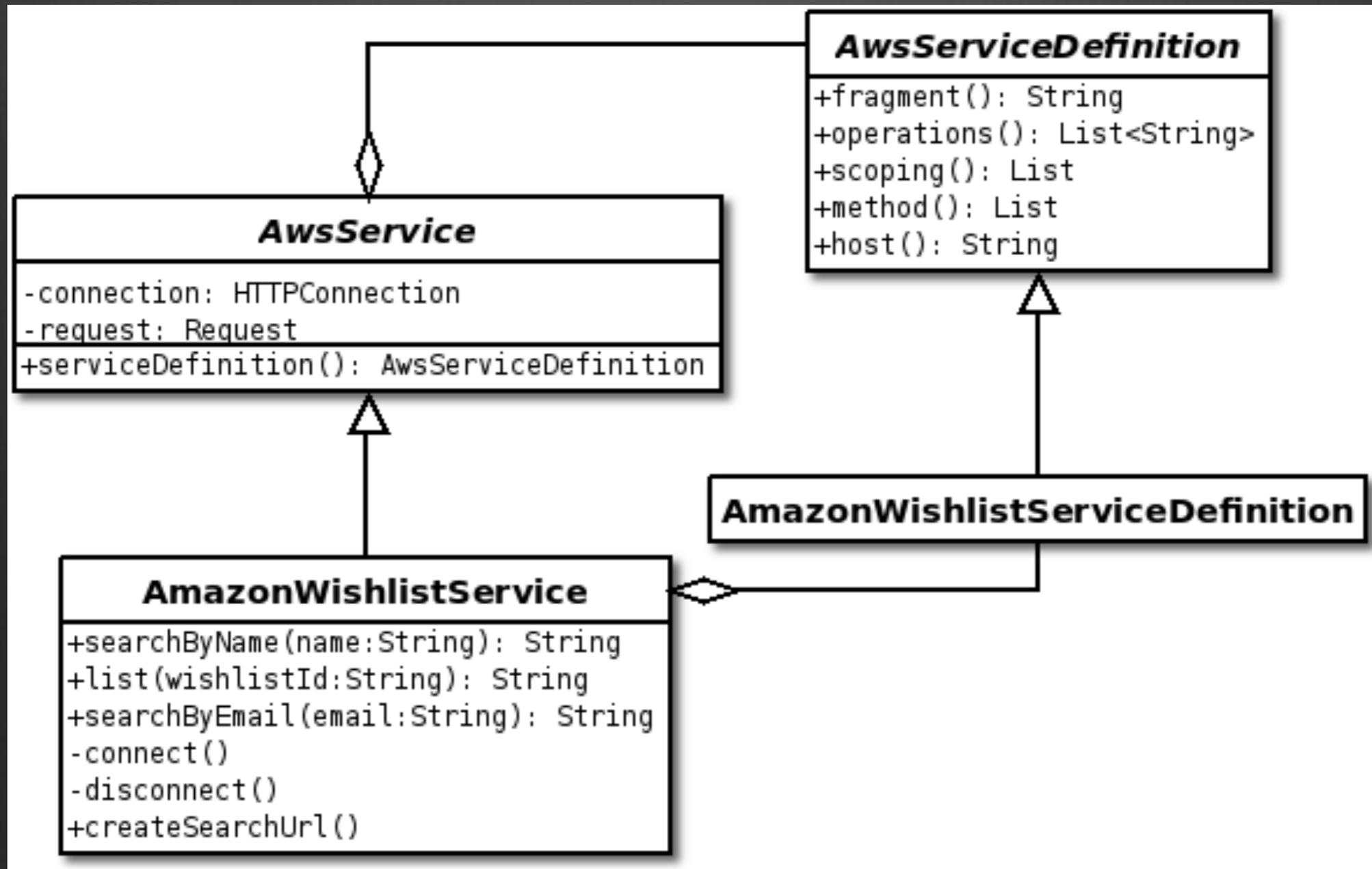
- Lessons learned in 10 years writing software
- Thoughts and beliefs acquired by reading and practicing
- Working style that helped me become a better engineer
- ...and a less frustrated one, too

About Your Host

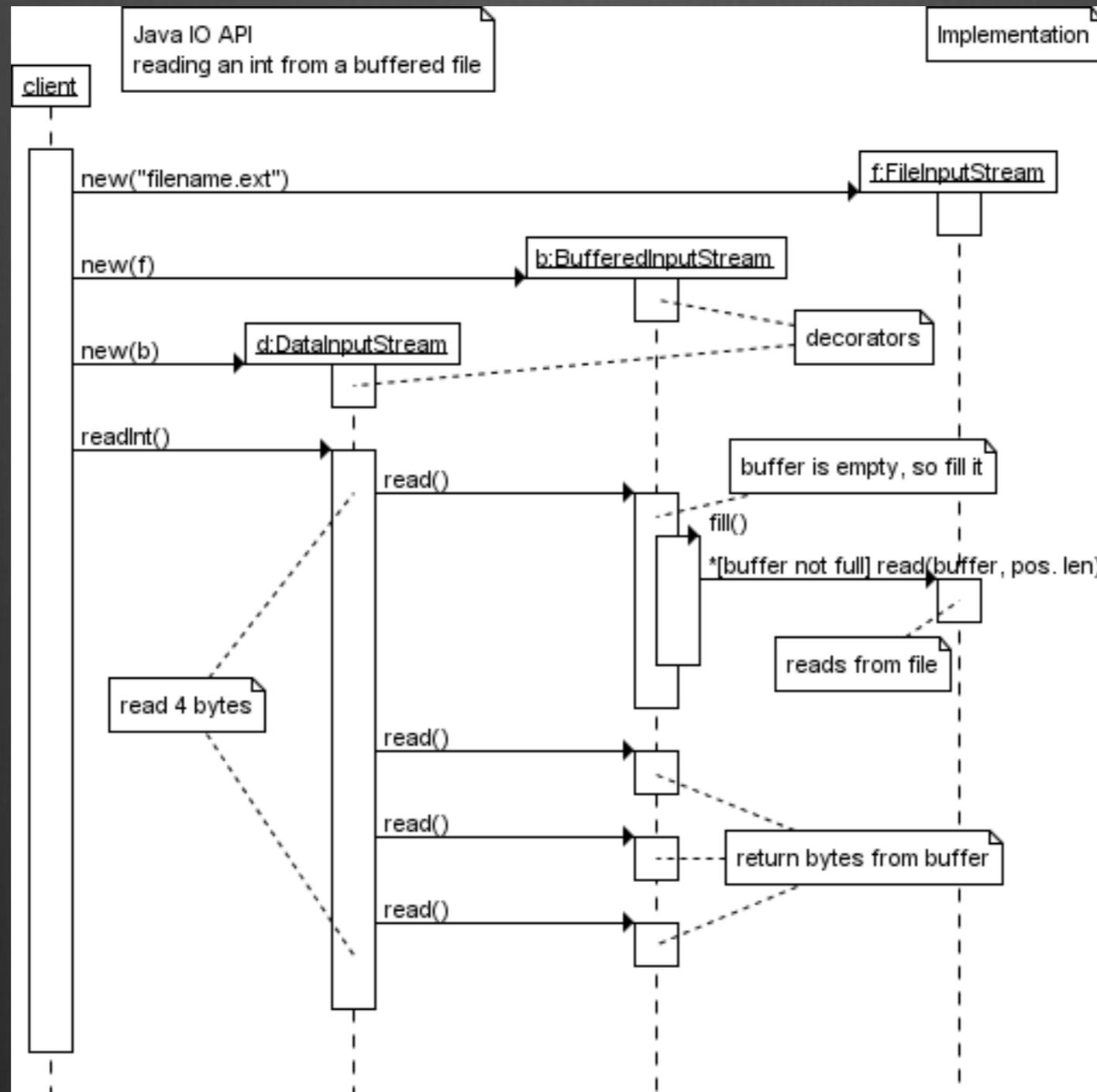
- Coding since 1994
- Engineering software and culture at Wix.com since 2010
- Involved with Israeli Scala and TDD communities
- Rewrote big chunks of Wix backend code

What is design?

Is it this?



Or this?



Wikipedia says:

“Software design is the process of implementing software solutions to one or more set of problems. One of the important parts of software design is the software requirements analysis (SRA). It is a part of the software development process that lists specifications used in software engineering.”

The Addison-Wesley Signature Series



GROWING OBJECT-ORIENTED SOFTWARE, GUIDED BY TESTS

STEVE FREEMAN
NAT PRYCE



Foreword by Kent Beck
Afterword by Tim Mackinnon

Software Gardening

In GOOS¹, Pryce and Freeman liken software engineering to growing a biological entity.

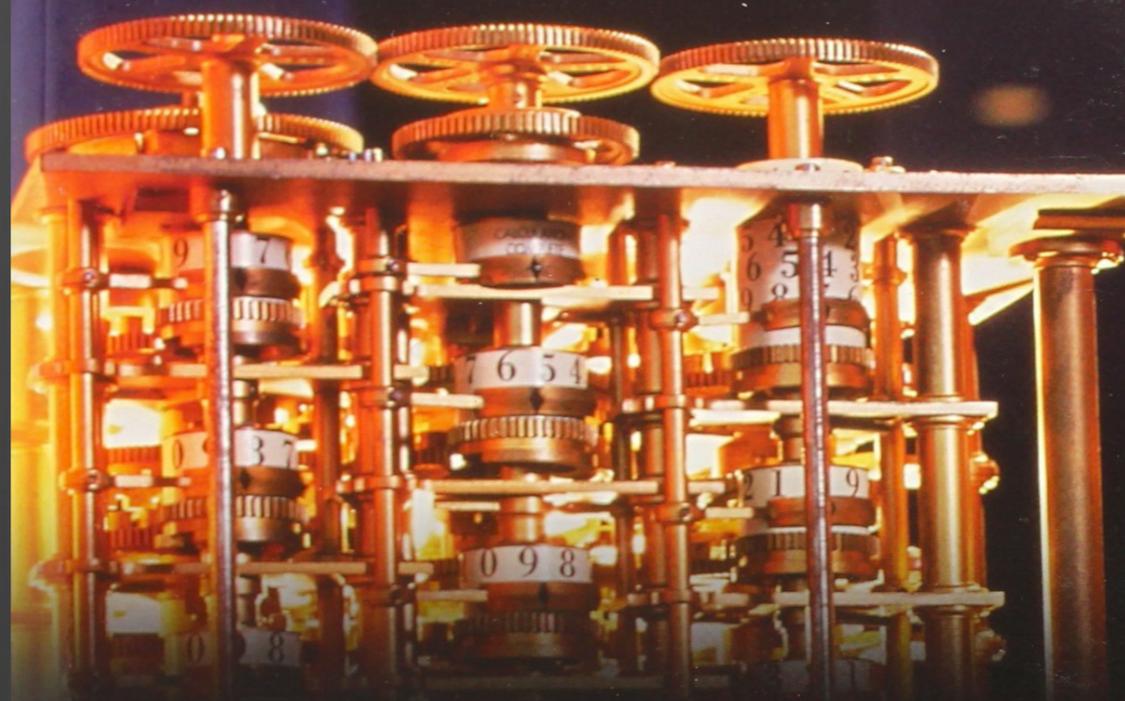
Our job is more like that of a gardener than that of a structural engineer².

If this is the case, where does *design* come into the picture?

¹ *Growing Object-Oriented Software, Guided by Tests*, page XVII

² <http://www.chrisaitchison.com/2011/05/03/you-are-not-a-software-engineer/>

Robert C. Martin Series



**WORKING
EFFECTIVELY
WITH
LEGACY CODE**

Michael C. Feathers

Michael Feathers says:

“When you are programming, you are doing detailed design. The manufacturing team for software is your compiler or interpreter. The source is the only complete specification of what the software will do. The cute boxes in class diagrams are not the design, they are a high level view of the design.”

He goes on saying:

“The artificial line between programming and design causes tremendous waste. Everyone thinks that software is intrinsically different. That it requires a completely different approach from anything else in engineering. It isn't different, we just misunderstood it. We believed that coding and testing were not part of design. They are just like prototyping on a [white]board. Part of design.”

The Implication

- To code *is* to design
- As we grow our software, the design *emerges*
- We code according to product requirement
- Alas, requirements often change as software grows
- Often, this gives rise to *Big Balls Of Mud*^{1 2}

¹ <http://www.laputan.org/mud/>

² <http://effectivesoftwaredesign.com/2013/06/17/the-myth-of-emergent-design-and-the-big-ball-of-mud/>

Or Put Differently:

- Requirements *will* change
- With design choices, earlier == bigger impact
- Thus, focus on *interfaces* and defer the *implementation*

Looking In



Outside In

- When writing a class, we code to interfaces rather than to implementation
- The same approach can – *and should* – be used when approaching systems, sub-systems and modules
- This is achieved when you start writing a new collaborator (be it a method, class, module) *from the call site*

Ain't Gonna Need It¹

- This, in essence is Outside-In Development
- We end up with cohesive, loosely-coupled, SOLID systems
- Promotes the YAGNI principle
- This keeps your code tight and orderly

¹ You actually might need it, but most likely not. See <http://c2.com/cgi/wiki?YagniExceptions>

YAGNI Violated





**THESE ARE NOT THE
ABSTRACTIONS**

**YOU ARE LOOKING
FOR**

On Mechanics and Semantics

- Outside in development also very nicely aims us towards Abstractions rather than Indirections
- Naming matters! i.e. *Librarian* vs *BookService*
- Interfaces over Implementations at system scale → Semantics over Mechanics
- Our job is to introduce these Semantics

They Die Hard

- Engineers grow up on the GoF book, Spring, Hibernate, etc
- Patterns and technologies are cool and exciting
- *And we want to use them*
- We need a paradigm shift to shake off old habits of over-engineering and pattern obsession

TDD To The Rescue



Not TDD

- Writing tests after writing production code
- Writing all tests for a module before writing the module
- Writing lots of E2E tests that are tightly-coupled to the system's inner behavior
- Writing E2Es for the server without including the client in the flows

A Brief History of TDD

- Coined by Kent Beck around the turn of the century
- Began as a part of the Extreme Programming movement but later spun off as its own methodology
- Body of research by Kent Beck, Nat Pryce, Steve Freeman and Martin Fowler

A Brief History of TDD at Wix

- Early 2011 - IT/E2E infrastructure for HTTP servers
- Late 2011 - Major refactoring of BBOM via E2E acceptance tests
- 2012 - Beginnings of TDD adoption
- 2013 - PETRI developed using TDD
- 2014 - TDD major part of Wix Academy

TDD Lifecycle

1. Test
2. Code
3. Refactor
4. GOTO 1



Test

- We start developing a system by writing a failing E2E test that demonstrates the first use case
- This test is in English and does not compile at this phase
- We examine our test to make sure it makes no assumptions about any implementation details

Code

- We start making the test pass, from the outside in
- Write the simplest code possible
- This will tease out any complexity we haven't considered before
- It will take some time - that's OK
- End up with *The Walking Skeleton*

Refactor

- We examine the code we just wrote – and any code that might have been affected – for smells
- We determine solutions to these smells. Often, these solutions are *Design Patterns*
- Iteration by iteration, a design for our system emerges

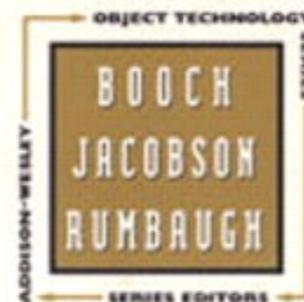
REFACTORING

IMPROVING THE DESIGN
OF EXISTING CODE

MARTIN FOWLER

With Contributions by **Kent Beck, John Brant,
William Opdyke, and Don Roberts**

Foreword by **Erich Gamma**
Object Technology International Inc.



What is Refactoring?

“Refactoring is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. It is a disciplined way to clean up code that minimizes the chances of introducing bugs. In essence when you refactor you are improving the design of the code after it has been written.”

The Forgotten Step

- We sometimes forget about the 'Refactor' step, or don't give it its due respect
- Might be due to time constraints, or misunderstanding of what 'Refactoring' actually means
- Refactoring is not an optional step - it's what aims us towards the required abstractions

Refactoring in the TDD flow

- TDD urges us to code in small increments
- It effectively holds us back from making drastic changes that are not necessary
- As a result, each refactor will be relatively small
- This is done both for production code and test code

Kicking off the TDD process

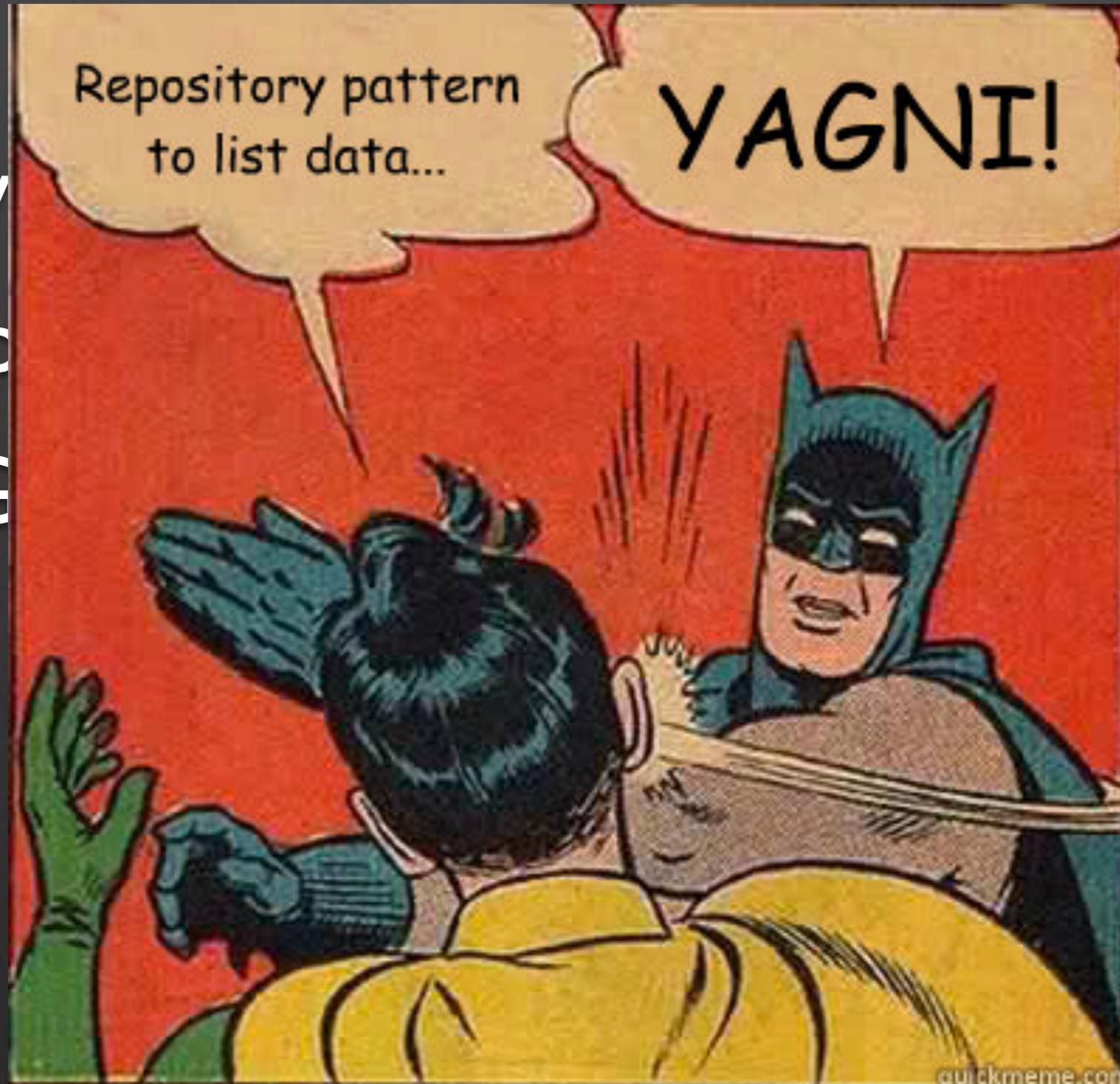
- We sit with the PM, go over the product requirements *as they are currently understood*
- We enumerate and prioritize features
- We pick a first feature and start coding it, building a Walking Skeleton
- We follow the rabbit hole – many hidden issues will now surface

Revised TDD Lifecycle

1. Pick the most important feature
2. Test
3. Code
4. Refactor
5. GOTO 2 - *or* -
6. GOTO 1

It's a Way of Life

- Cultiv
- Let go
- YSAG



Q&A



shaiy@wix.com



<http://www.shaiyallin.com>



@shaiyallin