Behaviour-Driven Development

Writing software that matters

Dan North – DRW

My name is Dan

I am a developer

I am a coach

I am your guide

Introduction: Software that doesn't matter

Failure modes – a field guide

The project comes in late

...or costs too much to finish

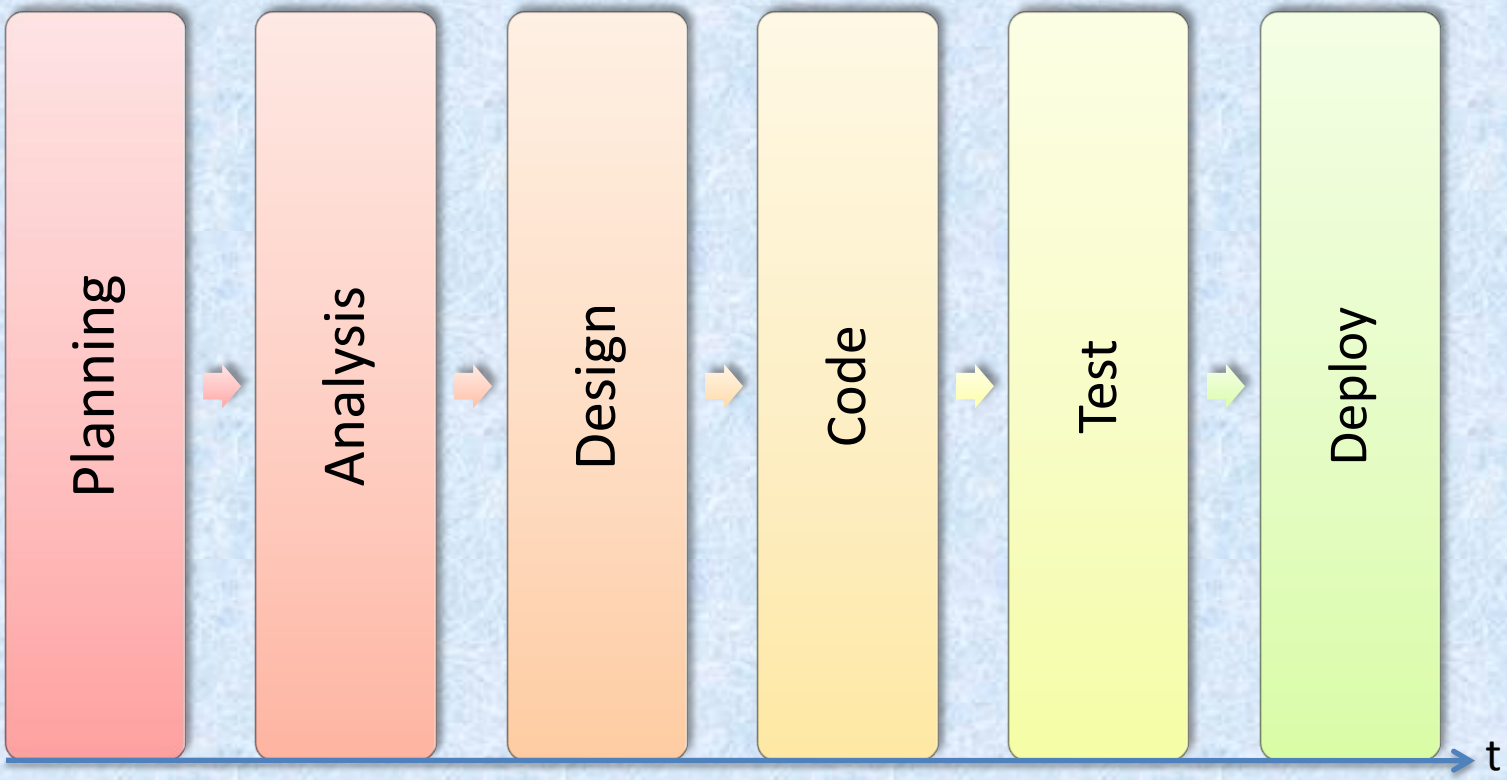
The application does the wrong thing

It is unstable in production

It breaks the rules

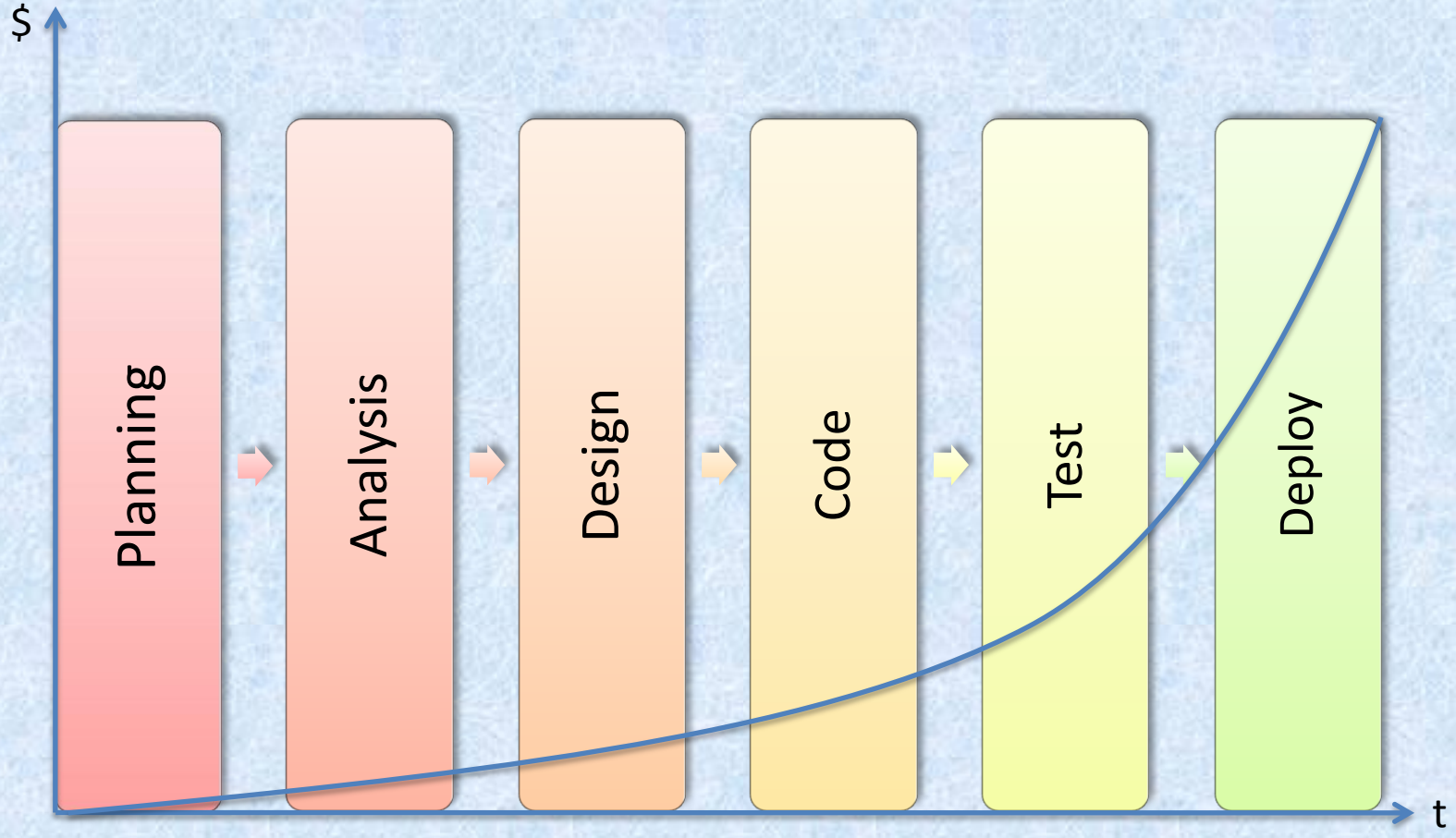
The code is impossible to work with

How we deliver software

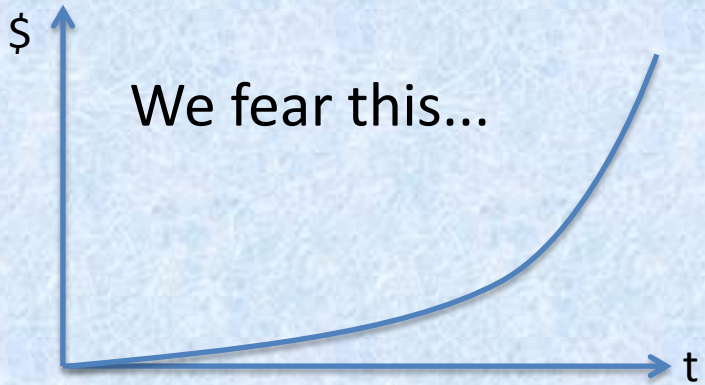


Why do we do this?

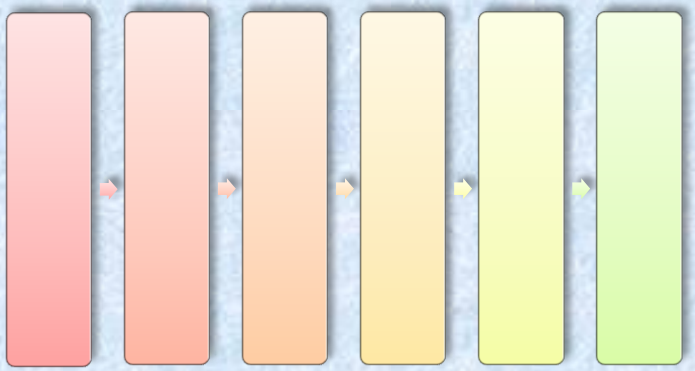
The exponential change curve



The exponential change curve



so we do this...



which reinforces this!

If only we could deliver better...

Deliver features rather than modules

Prioritise often, change often

Only focus on high-value features

Flatten the cost of change

Adapt to feedback

Learn!

What we would need

Adaptive planning

Streaming requirements

Evolving design

Code we can change

Frequent code integration

Run all the regression tests often

Frequent deployments

Part 1: Defining BDD

A loose definition of BDD

"Behaviour-driven development is about implementing an application by describing its behaviour from the perspective of its stakeholders"

- Me ☺

A more formal definition of BDD

"BDD is a second-generation, outside-in, pull-based, multiple-stakeholder, multiple-scale, high-automation, agile methodology.

"It describes a cycle of interactions with well-defined outputs, resulting in the delivery of working, tested software."

BDD is derivative

Derives from:

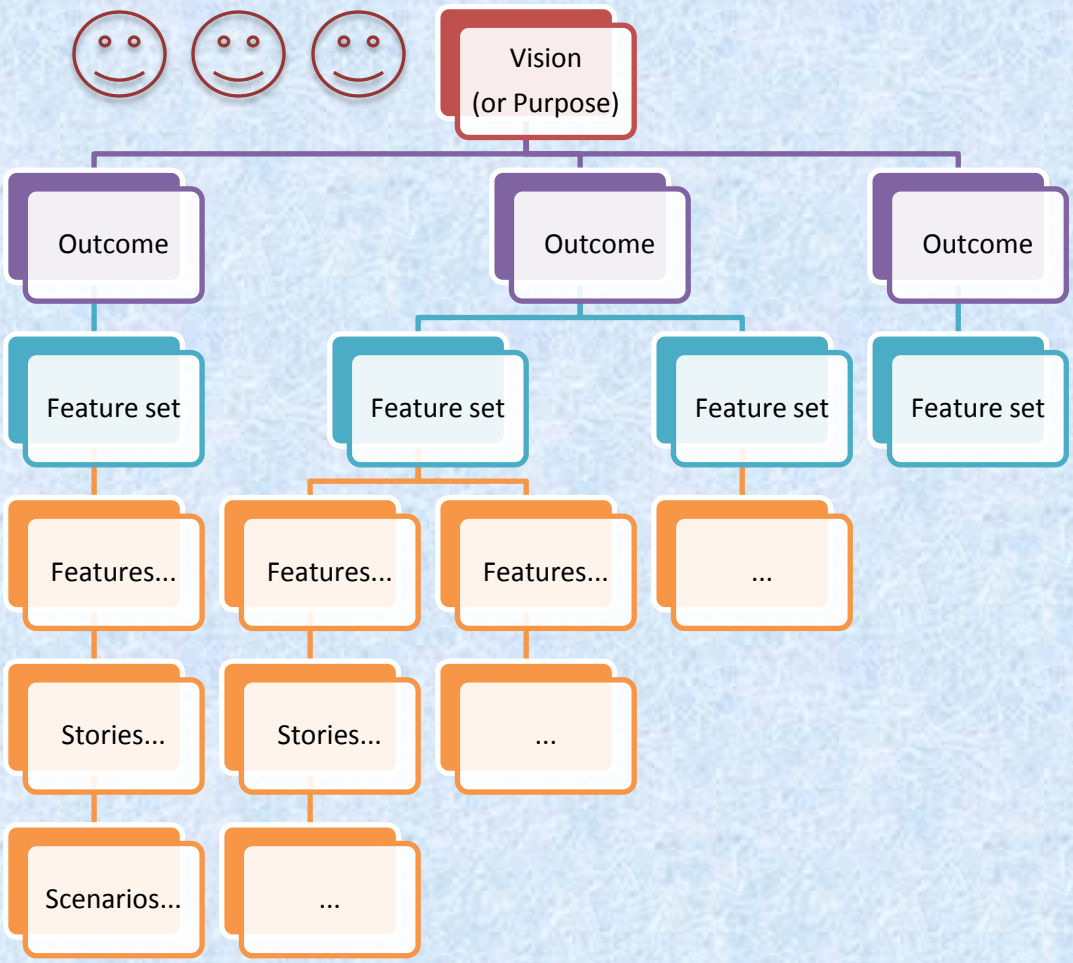
- XP, especially TDD and CI
- Acceptance Test-Driven Planning
- Lean principles
- Domain-Driven Design

Influenced by:

- Neurolinguistic Programming (NLP)
- Systems Thinking

...outside-in...

Who is this application *for*?



Everything has a diminishing return

Don’t create more detail than we can consume

- Analysis, design, estimation, planning, process

Or more technology than we need

- Don’t solve a problem we don’t have yet

Any more detail is waste, any less is risk!

- Focus on *deliberate discovery*

Principle 1: Enough is enough

Who is the stakeholder?

Anyone who cares!

…about how much the application costs
…about what it does and how to use it

…about whether it hammers the network
…about whether it is secure
…about whether it complies with the law
…about how easy it is to deploy and diagnose
…about how well it is written and architected
…and how easy it is to change

Two flavours of stakeholder

Core stakeholders

- the people with the vision

Incidental stakeholders

- the “non-functional” stakeholders
- the people working to achieve the outcomes

Principle 2: Deliver stakeholder value

BDD works on multiple levels

Stories and scenarios describe *application-level* behaviour

Code examples describe *code-level* behaviour

Wider scope is possible

e.g. Behaviour-driven “guerrilla” SOA

Principle 3: It’s all behaviour

Automation creates rapid feedback

CI ensures the application is always releasable

- Requires *comprehensive* automated acceptance tests
- And that your CI environment is similar to the real one

CI and SCM principles apply elsewhere too

- You can version your database changes
- And automate the roll-forward and roll-back

Some features require on-going monitoring

- e.g. Performance testing or penetration testing

The Agile Manifesto

People and interactions *over* process and tools

Collaboration *over* contract negotiation

Working software *over* documentation

Adapting to change *over* following a plan

The XP values

Communication

Simplicity

Feedback

Courage

Respect

It doesn't end at "dev complete"

Use your build process for release

- Then the path to production is tested

Aim for "deterministically boring"

Engage the downstream stakeholders

- A release shouldn't come as a surprise

*Software has **zero value** until it is live!*

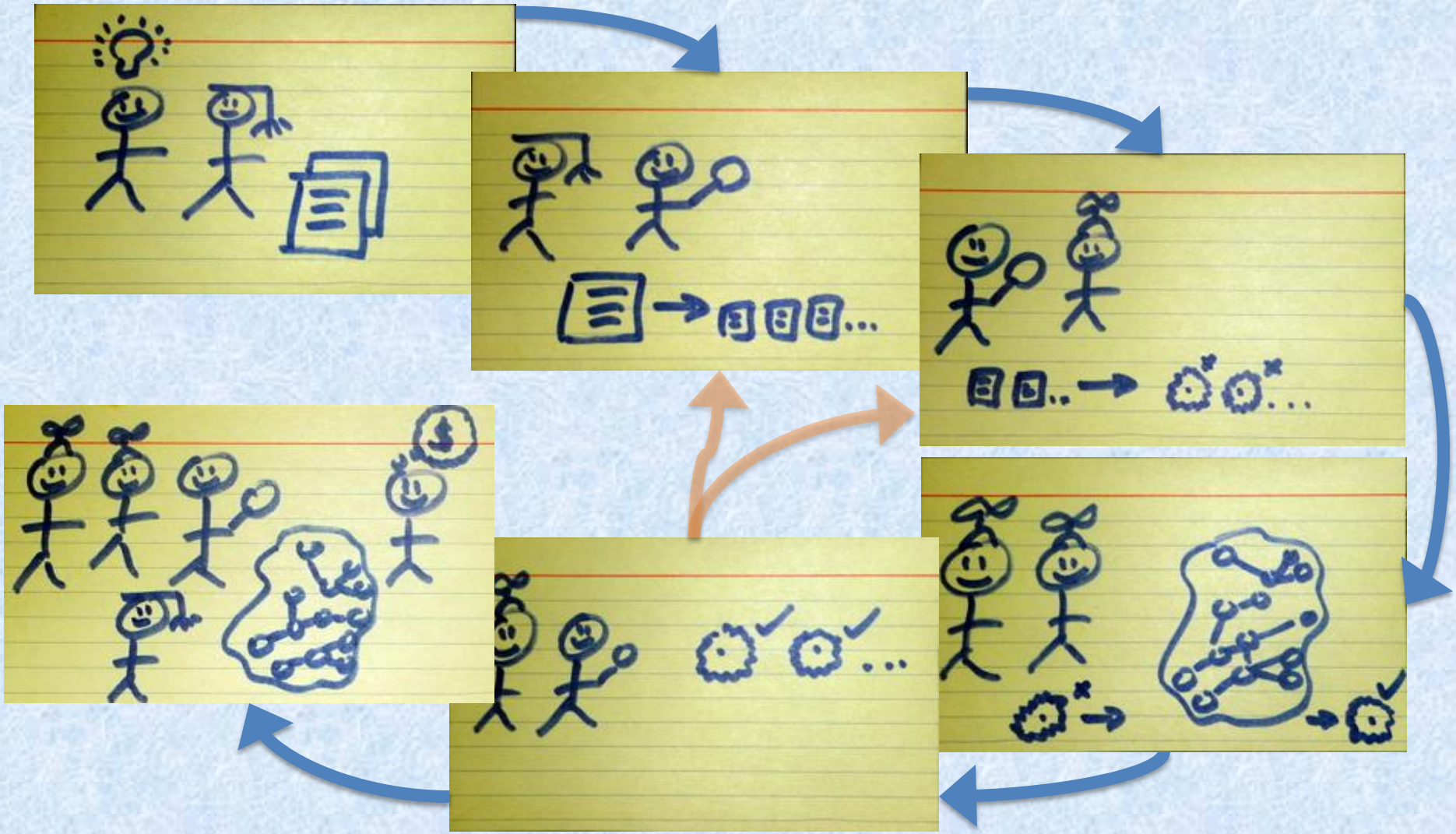
Recap – three principles of BDD

1. Enough is enough
2. Deliver stakeholder value
3. It's all behaviour

Part 2: How BDD works

...cycle of interactions...

BDD in six pictures



The roles in a BDD team

The core stakeholders

The incidental stakeholders

The analysts (or BAs)

The testers (or QAs)

The developers

The project manager (or Boss)

What's in a story?

A story is a unit of delivery

Story 28 - View patient details
As an Anaesthetist
I want to view the Patient's surgical history
So that I can choose the most suitable gas

...clearly-defined outputs...

Focus on the value

Story 28 - View patient details

In order to choose the most suitable gas

an Anaesthetist

wants to view the Patient's surgical history

...clearly-defined outputs...

Focus on the value

Story 29 - Log patient details

In order to choose the most suitable gas

an Anaesthetist

wants other Anaesthetists to log the

Patient's surgical history for later retrieval

...clearly-defined outputs...

Agree on "done"

Define scope using scenarios

Scenario - existing patient with history
Given we have a patient on file
And the patient has had previous surgery
When I request the Patient's surgical history
Then I see all the previous treatments

Automate the scenarios

Make each step executable

Given we have a patient on file

In Ruby:

```
Given “we have a patient on file” do
    # ...
end
```

In Java:

```
@Given(“we have a patient on file”)
public void createPatientOnFile() {
    // ...
}
```

Code-by-example to implement

Also known as TDD

Start with the edges, with what you know

Implement outermost objects and operations

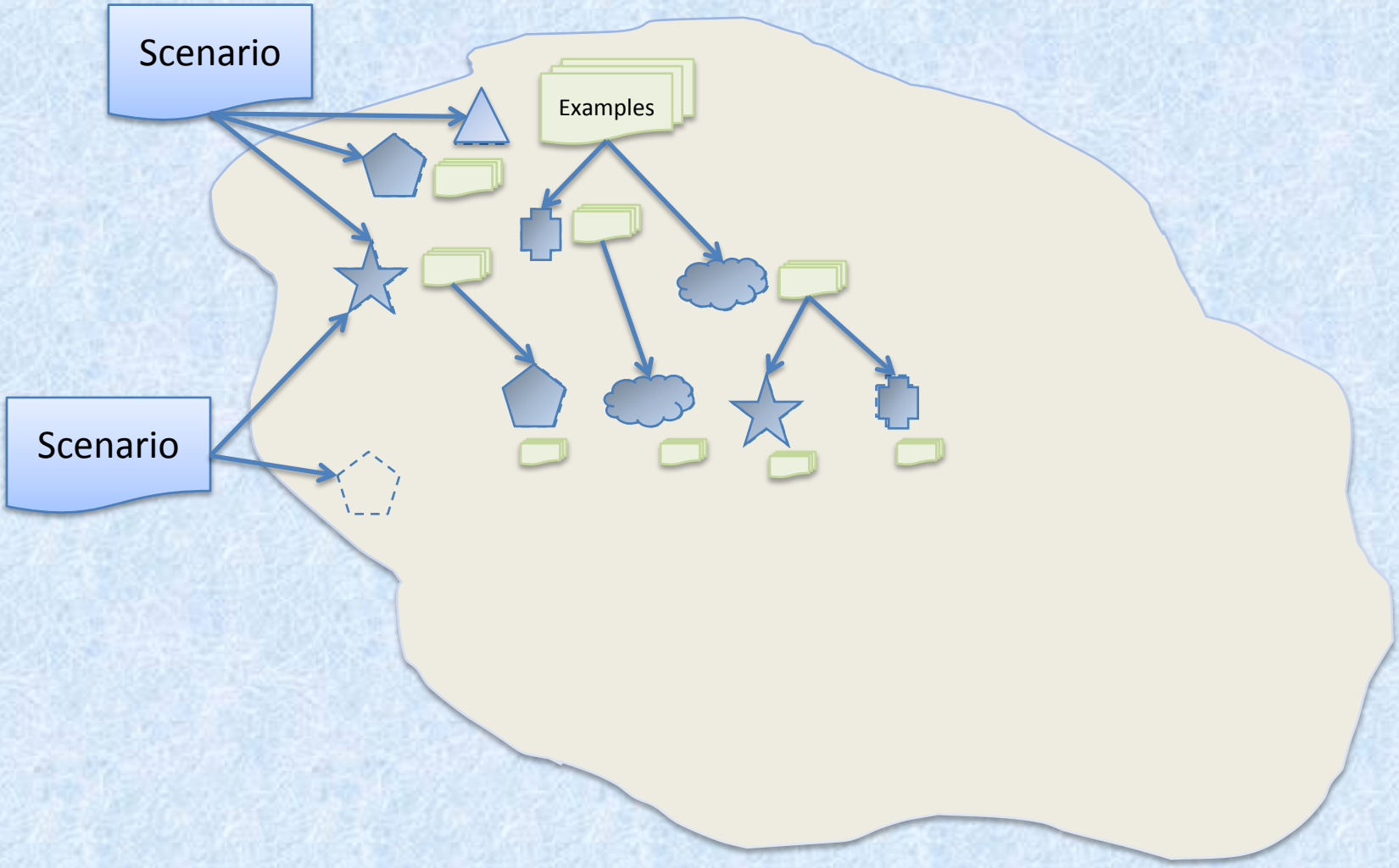
Discover collaborators, working inwards
and mock them out for now

Repeat until “Done”

If the model doesn’t “feel” right, experiment!

...clearly-defined outputs...

Code-by-example example



Good tools can help here

Cucumber or JBehave for stories

RSpec, XUnit for code examples

Mockito, Mocha, Moq for mocking

Be opinionated rather than dogmatic with the tooling!

We keep the development artifacts

Examples become code tests

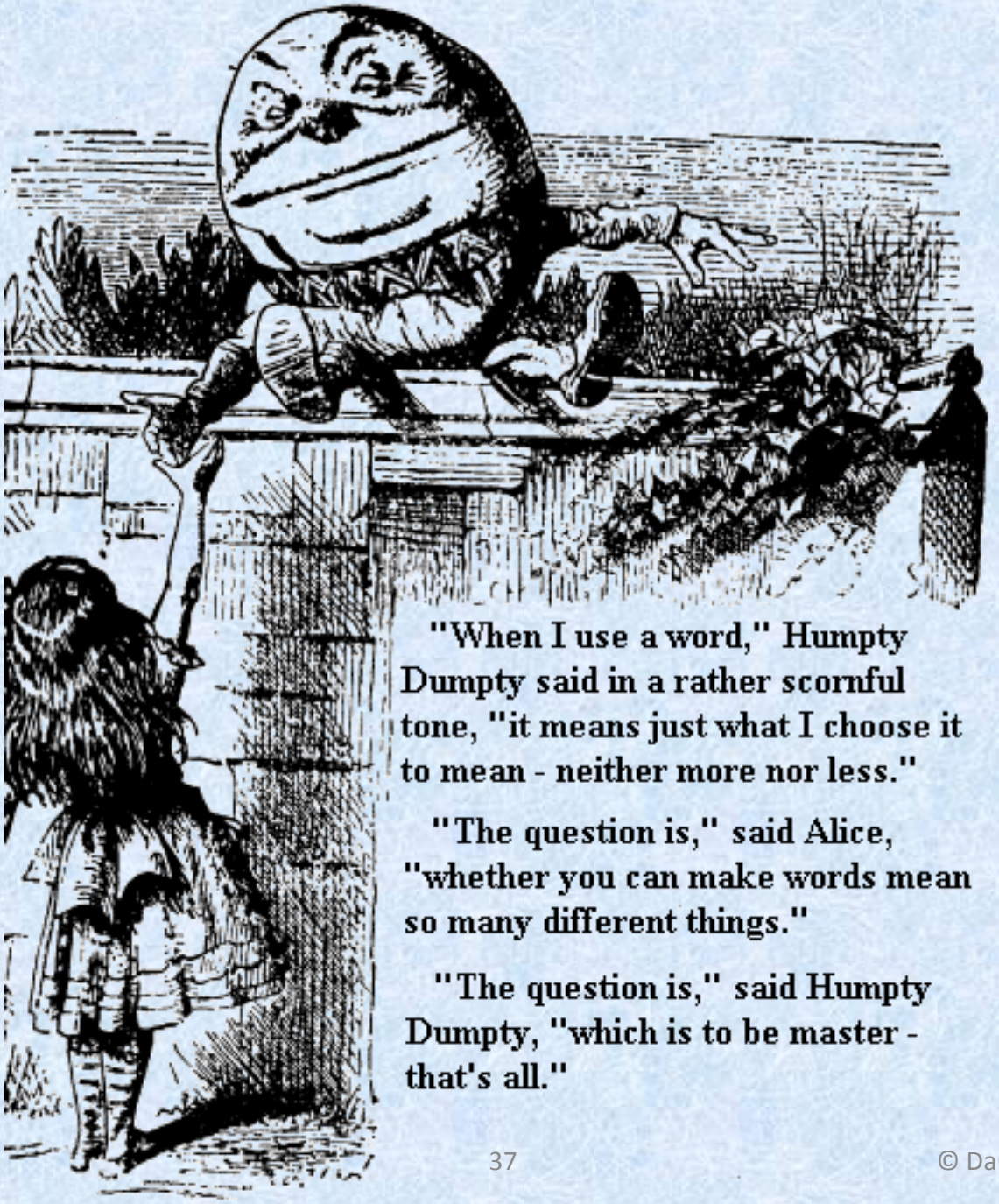
…and documentation

Scenarios become acceptance tests

which become regression tests

Automation is key

Part 3: Getting the words right



"When I use a word," Humpty Dumpty said in a rather scornful tone, "it means just what I choose it to mean - neither more nor less."

"The question is," said Alice, "whether you can make words mean so many different things."

"The question is," said Humpty Dumpty, "which is to be master - that's all."

Domain-driven design 101

Model your domain

...and identify the core domain

Create a shared language

...and make it ubiquitous

Determine the model's bounded context

...and think about what happens at the edges

The map is not the territory

There are many kinds of model

- Each is useful in different contexts

There is no “perfect” domain model

- So don’t try to create one!

Domain modelling takes practice

A legacy domain modelling example

```
Map<int, Map<int, int>>
    portfolioIdsByTraderId;

if (portfolioIdsByTraderId.get(trader.getId())
    .containsKey(portfolio.getId())) {...}
```

becomes:

```
if (trader.canView(portfolio)) {...}
```

We often manage multiple domains

You want to retrieve patient records

in Java, using Hibernate

so you define

```
class HibernatePatientRecordRepository {
```

What if your IDE did domain-specific fonts?

Writing effective stories

Each story represents (part of) a feature

- and each feature belongs to a stakeholder

Each stakeholder represents a domain

- even the incidental stakeholders

Mixing domains *within a scenario* leads to brittle tests

- What exactly is the scenario verifying?
- What does it mean when things change?

"What does the stakeholder want in this story?"

Part 4: Other topics

Getting started with BDD

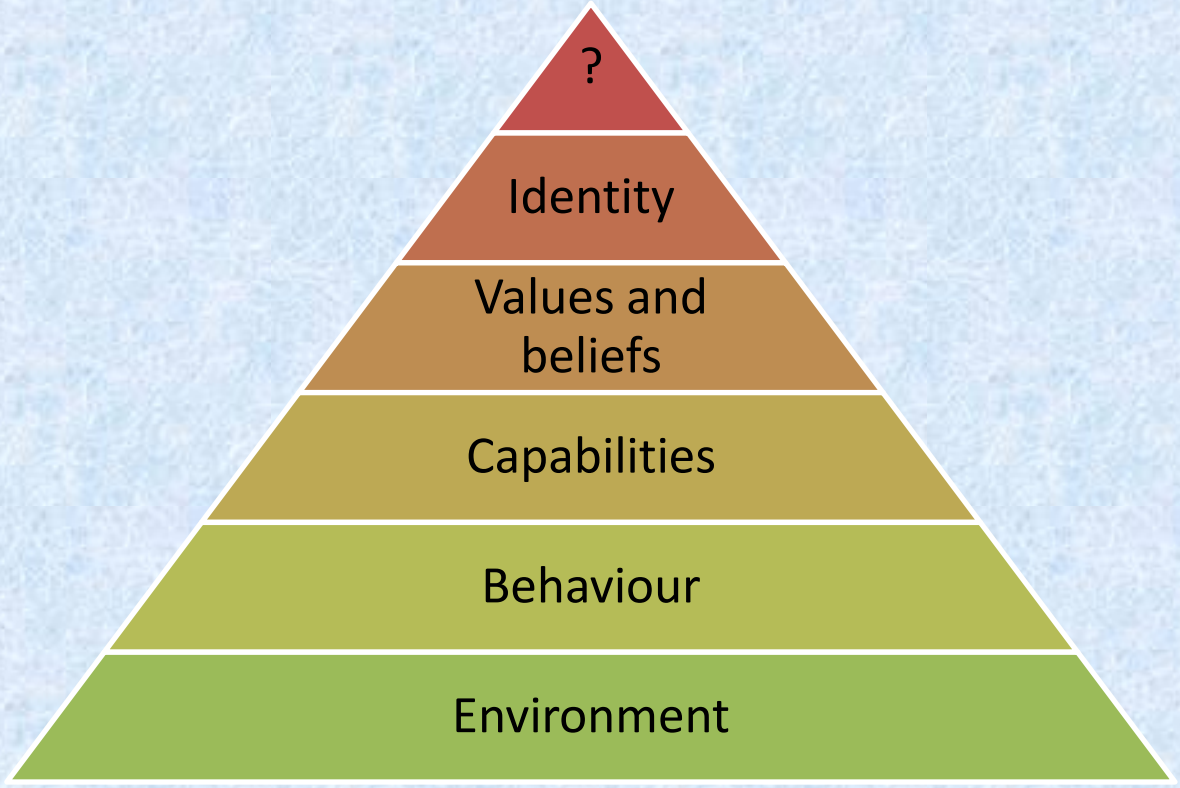
Lasting change involves values and beliefs

Introducing any change is disruptive

We need to understand how this works

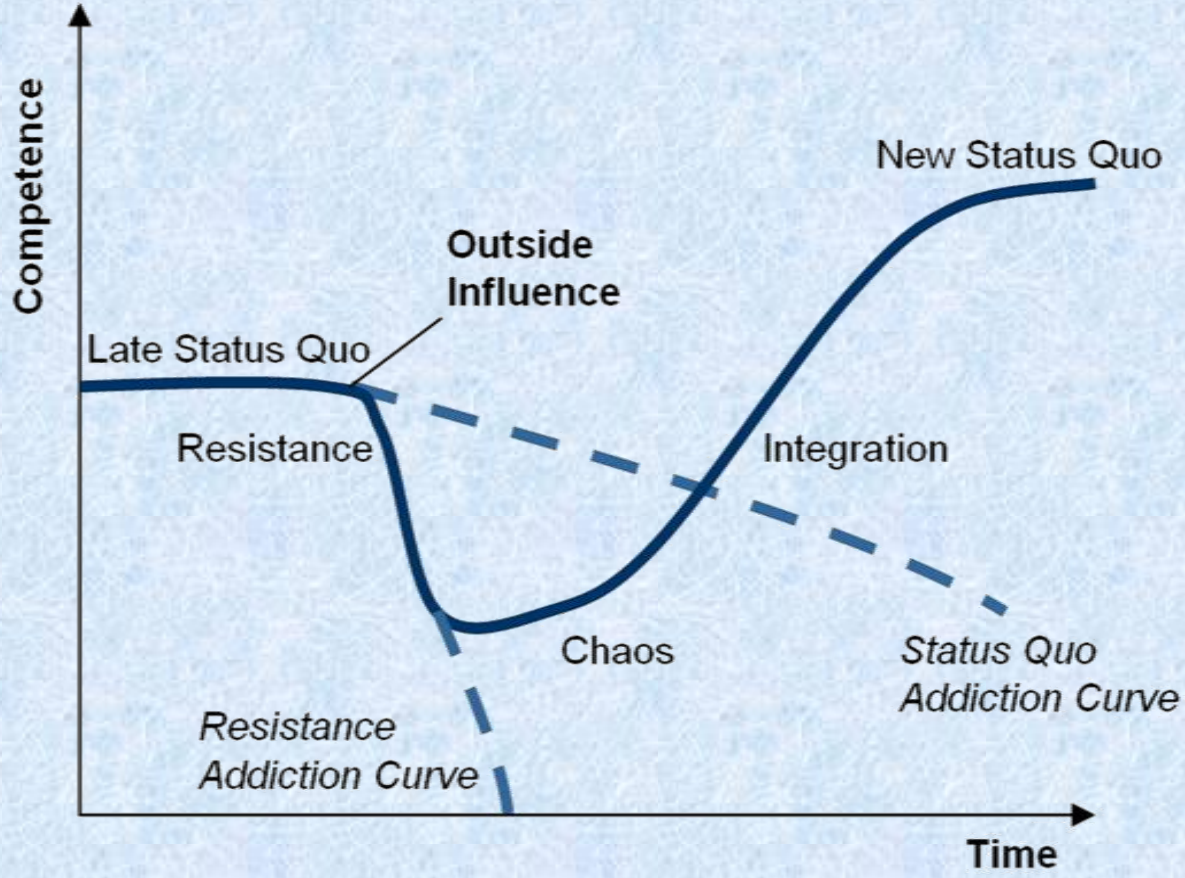
Getting started with BDD

Values and beliefs: the Dilts model



Getting started with BDD

Introducing change: the Satir model



Getting started with BDD

Lasting change involves values and beliefs

Introducing any change is disruptive

Use small increments, find quick wins

Identify suitable pilots, nurture them

BDD is ideally suited to this

BDD on legacy systems

"Working effectively with legacy code" – Michael Feathers

Introduce automation early

- SCM is vital, build is vital, CI is critical

Test your assumptions

- Automated tests will give you confidence
- Use the tests to build out a domain model

Especially around integration points

- Triple benefit: assurance, stub and regression

BDD in the large

"No more than 10"

(with thanks to Linda Rising)

Partition work by functional areas

- with clear interfaces and boundaries

Enable each team to be fully autonomous

- avoid the "Testing Centre of Excellence"

Have a single codebase and a single build

Distributed BDD

Same as for large teams, plus...

Have multiple stand-ups to “pass the baton”

Use technology to shorten the distance

- video-conferencing, digital whiteboard, Skype

Be aware of cultural disconnects

Exaggerated Collaboration

Conclusion

Software that matters

...has tangible stakeholder value

...is delivered on time, incrementally

...is easy to deploy and manage

...is robust in production

...is easy to understand and communicate

BDD is a step in that direction

Thank you

dnorth@drwuk.com

@tastapod

http://dannorth.net
http://jbehave.org
http://rspec.info

Bibliography

Extreme Programming explained (2nd edition)
- Kent Beck

Domain-Driven Design - Eric Evans

The Art of Systems Thinking
and
The Way of NLP - Joseph O'Connor