Growing Object-Oriented Software, Guided by Tests

Part 1

Martin Stolle, 04.07.2018

Über das Buch

- Autoren Steve Freeman und Nat Pryce
- UK
- Software Berater
- Gründer London XP Days
- 2010 Veröffentlicht

1. Theorie ← Wir sind hier
2. Anwendung in der Praxis anhand eines Beispiels
3. Testbarkeit erhalten im Projektverlauf

Inhalt

- Implementing TDD effectively: getting started, and maintaining your momentum throughout the project
- Creating cleaner, more expressive, more sustainable code
- Using tests to stay relentlessly focused on sustaining quality
- Understanding how TDD, Mock Objects, and Object-Oriented Design come together in the context of a real software development project
- Using Mock Objects to guide object-oriented designs
- Succeeding where TDD is difficult: managing complex test data, and testing persistence and concurrency

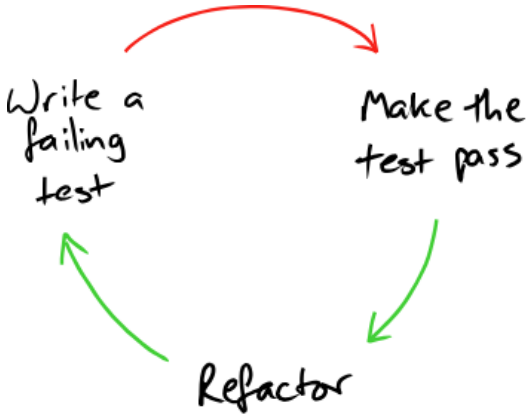
Quelle: http://www.growing-object-oriented-software.com/index.html, 25.06.2018

Warum TDD?

- Learn how components work whilst completing the project
- System that combines many components will be too complex to understand
- Anticipate unanticipated changes
- Empirical feedback to learn and the system and its use
- Team needs repeated cycle of activity
- Deployment is critical at each cycle
- Deployment is to check assumptions against reality
- Project organized with loops is incremental and iterative
    - Incremental: build system feature by feature
    - iterative : refine implementation in response to feedback
- “You have nothing to lose but your bugs”

TDD in a nutshell

- 2 foundations for growing code
  - Constant testing
  - Keep code as simple as possible
    - Easier to understand and modify
    - Devs read more code that write
- Refactor: micro technique, small scale improvements without changing the behaviour

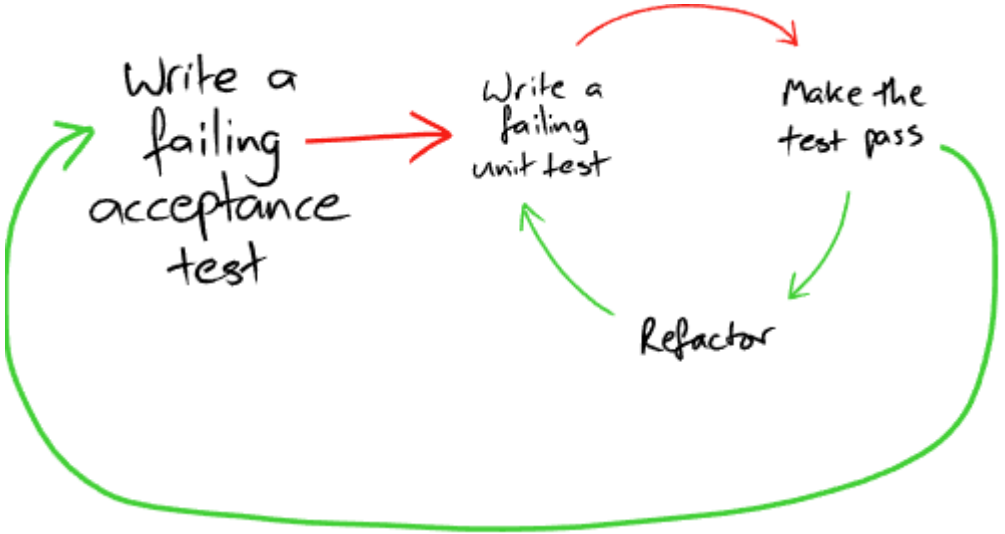


Quelle: http://www.growing-object-oriented-software.com/figures/tdd-simple.svg, 25.06.2018

TDD in a nutshell

- Start with acceptance test
  - When it passes feature is done
- Outer cycle: Acceptance test
- Inner cycle: Unit test
- End-to-end: interact with the system only from the outside
- Acceptance test: Does the system work?
- Integration test: Does our code work against code we can’t change?
- Unit test: Do our objects do the right thing, are they convenient to work with?

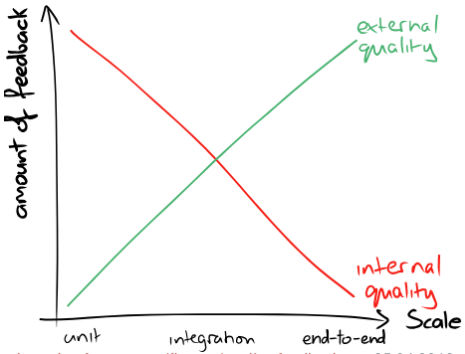
TDD in a nutshell



Quelle: http://www.growing-object-oriented-software.com/figures/tdd-simple.svg, 25.06.2018

TDD in a nutshell

- External quality: how well system meets the needs of its customers and users
- Internal quality: how well it meets the needs of its developers and admins
- It allows us to modify the system behavior safely and predictably
- End-to-end tests = external quality
- Unit tests = internal quality, feedback for design
  - Loosely coupled and highly cohesive
- Coupled:
  - if a change in one force a change in the other
- Cohesion:
  - Is a measure of weather its responsibilities form a meaningful unit



Quelle: http://www.growing-object-oriented-software.com/figures/quality-feedback.svg, 25.06.2018

TDD with objects

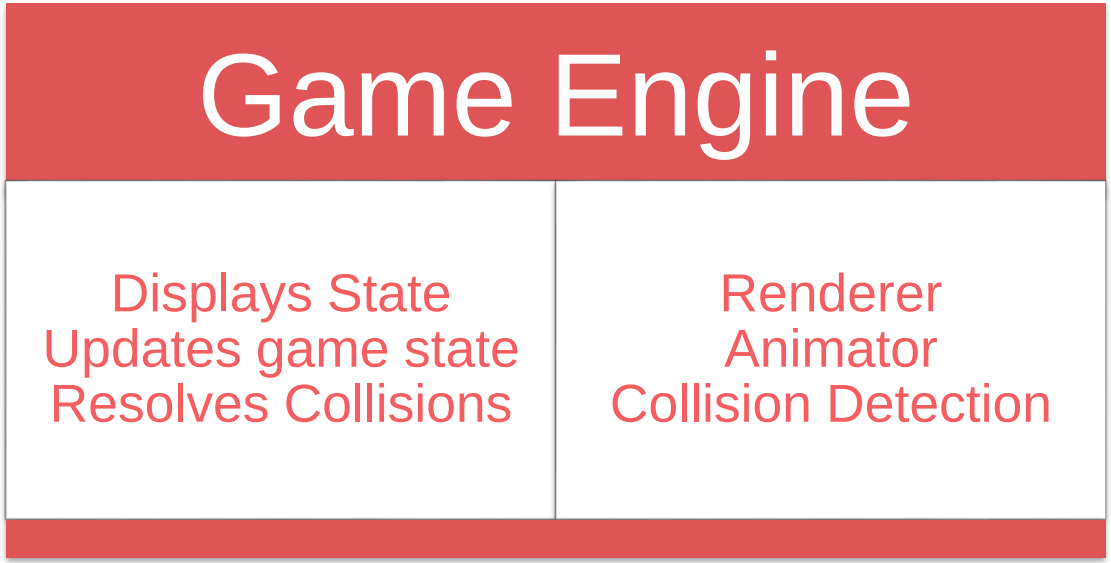
- Object:
  - receives & send messages (more important than internal properties or behaviours)
  - identity might change depending on the message they receive
- Method
  - handles message it understands
  - Encapsulates internal state to coordinate communication
- System:
  - Composition of objects
  - Web of collaborating objects
- Change behaviour by changing composition
- Values: Model unchanging quantities and measurements

TDD with objects

- Communication pattern is a set of rules that govern how a group of objects talk to each other
- The roles they play, what messages they can send
  - Define roles with abstract classes
- Object
  - Implementation of one or more roles
- Role
  - Set of related responsibilities
- Responsibility
  - Obligation to perform a task or know information

CRC-Cards

- By Wirfs-Brock and McKean, Object Design: Roles, Responsibilities, and Collaborators, Addison-Wesley, 2003
- Index cards to explore the potential object structure of an application



TDD Tools

- The calling object should describe what it wants in terms of the role that its neighbour plays “Tell, Don’t ask”-Law of Demeter
  - Asking still allowed occasionally
- Objects make decision based on the info they hold internally or that came with the message
- Avoid navigation to other objects
- Use mock to specify how we expect the target object to communicate with its mock neighbours

```
master.getModelisable().getDockablePanel().getCustomizer().setEnabled(true);
master.allowSavingOfCustomisations();
```

TDD Tools

- Mockery
  - Holds context of a test, creates mock objects and manages expectations and stubbing for the test

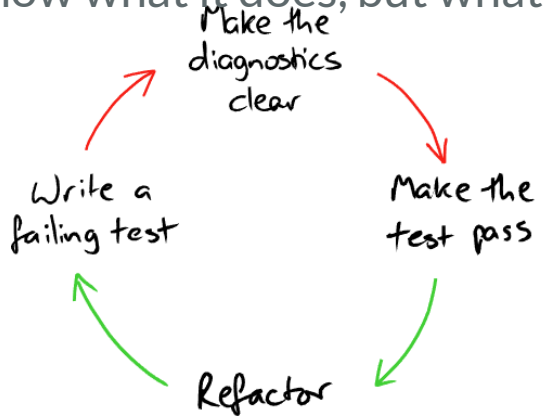
1. Create required mocks
2. Create any real objects
3. Specify how you expect the mock objects to be called
4. Call target
5. Assert resulting values are valid and all calls are made

TDD - Kickstart

- Start with build, test deploy
  - Walking Skeleton
  - Flushing out technical and organizational risk
- After that: acceptance test
- Deploy
  - Understanding the process while automating it
- Draw the walking skeleton on a whiteboard, understand the problem
  - Broad brush design
- Front load stress!

TDD Cycle revisited

- Write failure test cases down, later implement them
- Write test that you'd want to read
  - TestDox convention
- **Error messages should guide us, express indent of the test**
- Develop from the inputs to the outputs
- Test should not just show what it does, but what the method is for



Quelle: http://www.growing-object-oriented-software.com/figures/quality-feedback.svg, 25.06.2018

TDD Cycle revisited

- Write acceptance test using only terminology from the application domain
  - Not from underlying technology
  - Focus on implementing the feature
  - Look at the system from users point of view not implementers
- Unit test exercise objects in isolation
  - Helps with design and confidence
- Unit and integration test support the dev team
  - Quick and always pass
- Acceptance tests for completed features should always pass
- Don’t start with failure cases, start with the simplest success not that simplest case
  - Starting with failure bad for morale not with the simplest

OOP Style

- Difficult to test? Don’t ask how to test but why it is difficult
- Fear kills progress
- How well is our TDD strategy
  - Reflect regularly, identify weaknesses
- Value code that is easy to maintain than easier to write
- Separation of concerns
  - Change as little code as possible to change behavior of a system
- Higher levels of abstraction
  - Avoid complexity, program by combining components rather than manipulating variables and control flow
- Encapsulation: behavior of object can only be affected through its API
- Information hiding: coneals how an objects implements its functional behind abstraction of its API

OOP Style

- Composite simpler than the sum of its parts
  - New objects encapsulates behaviour for easier use
- 3 types of relationships
  1. Dependencies: Object cannot function without these services (Canvas)
  2. Notifications: Object notifies interested peers whenever it changes state, fire and forget (Button)
  3. Adjustments: Adjust objects behaviour to the wider needs of the system (Renderer)
- Write helper methods to improve readability even if they are very small
- Name these methods to make the calling code more readable
- Functional programming style within an object, message passing style between objects
- Single responsibility principle
  - Describe what objects does without “and/or”

OOP Style

- The API of a composite object should not be more complicated than that of any of its components
  - Rule helps us decide whether and object hides enough information
- Context independent: object has no knowledge where it executes
- Relationship might be permanent passed in construction or transient, passed in method
- Paternalistic approach: each object is just told enough to do its job
- Context independence simplifies objects
  - Don't need to manage their own relationships
  - Managing relationship easier
    - Created and composed together in same place

Archiving Design

- Starting with a test means we have to describe what we want before we consider how
- Keep unit tests understandable by limiting their scope
- Encourages context independence because we have to set up dependencies
- System is a web of communication objects
- Interfaces to define available messages
  - Dependency Inversion Principle / Liskov Substitution Principle
- An interface describes whether two components will fit together, while a protocol describes whether they will work together

Achieving design

- More important than the available messages (interfaces) are the patterns of communication - their communication protocols
  - If we receive message A, do we execute B?
- Mock objects make communication protocols visible
  - See the objects peers
- Mocks encourages information hiding
  - The less you mock the better
- Mock dependencies, notifications, adjustments but NOT internals
- Types to represent value concepts in the domain
  - Create a consistent domain model

Achieving design

- Value
  - immutable, simple, no meaningful identity
- Objects:
  - have state, identity and relationships
- Feet and meters and numbers but different things
- Helps us find relevant code
- 3 basic techniques for introducing value types
  1. Breaking out
     i. code to complex > break out coherent units of behaviour into helper types
  2. Budding off
     i. Placeholder type for new domain concept, as code grows fill in more details
  3. Bundling up
     i. commonly group of values always used together, new type that highlights the missing concept

Achieving design

- If code gets too complex to understand → clean up
- Breaking out - Splitting large objects into a group of collaborating objects
  - Pulling out cohesive units of functionality into smaller collaborating objects
  - New objects can be tested
  - Forces us to look at the dependencies
- Unit test say
  - Break up an object if it becomes too large to test easily, or if its test failures become difficult to interpret. Then unit-test the new parts separately

- Treat code as spike
  - once we know what to do roll back and re-implement cleanly

Achieving design

- Budding off – defining a new service that an objects needs and adding a new object to provide it
  - When implementing an object we discover that it needs a service
  - Give service a name, mock it in the unit tests to clarify relationship between the two
  - Then write an object to provide that service, repeat

- Unit test say
  - When writing a test, we ask ourselves: „if this worked, who would know?“ If the right answer to that question is not in the target objects, its probably time to introduce a new collaborator

Achieving design

- Bundling up: Hiding related objects into a containing objects
  - Composite is simpler that the sum of its parts
  1. Name helps us understand the domain better
  2. Scope dependencies more clearly
  3. We can test the new composite object directly

- Unit test say
  - When a test for an object becomes too complicated to set up – when there are too many moving parts to get the code into the relevant state – consider bundling up some of the collaborating objects
  - “Bloated constructor”

Achieving design

- Interfaces reflect relationships between objects as defined by their communication protocols
- Interfaces to name the roles that objects can play and to describe the messages they'll accept
- To express intent in testing code use 2 layers
  - Implementation layer: graph of objects, behavior is combined result of how its objects respond to events
  - Declarative layer: builds up objects of implementation layer using small “sugar” methods to describe the purpose of each
- Achieve more with less code
- Raise ourselves from programming in therm of control flow and data manipulation to composing programs from smaller program where objects from the smallest unit of behavior

Walking Skeleton

- Learn how to slice up the functionality so it can be built with little time
- Each slice should be concrete that the team can tell when it is done
- Iteration zero: setting up the infrastructure
- Programming by wishful thinking
  - Write tests as if the application already exists
- Avoid mocking external libraries
  - It may work different in another version
- Write a thin layer of adapter objects that uses third party API
- We test the adapter

Walking Skeleton

- Name methods that trigger events in imperative mood
  - `openConnection()`
- Names of assertions in indicative mood
  - `isConnectionOpen()`
- Names should be descriptive
- Failure defines target for our next coding episode
- Outside-in development

Ende