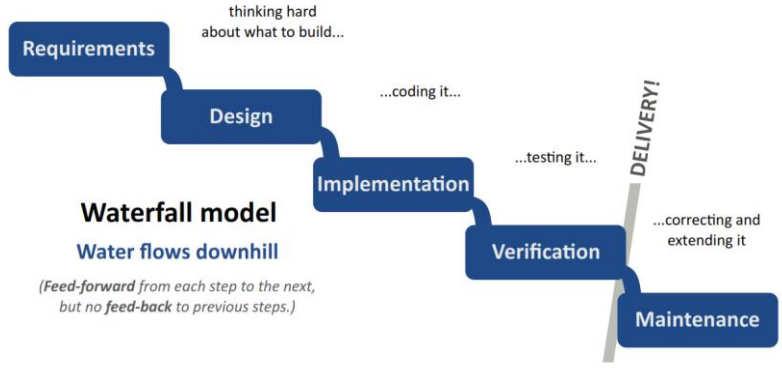
Stepping Back: The Big Picture

Traditional views of testing, and why they're suboptimal

In the (oft-ridiculed) **waterfall model**, we distinguish the following phases:



- Developers hand off a finished product
- Testers (likely another team) check that it works

Traditional views of testing, and why they're suboptimal

The waterfall model comes from the **manufacture** and **construction** industries.

Working with physical artifacts is different than working with software:

- We know that **Construction** and **testing** are obviously two distinct activities.
- A defect might be prohibitively difficult or costly to fix once the product leaves the factory.

Traditional views of testing, and why they're suboptimal

With software, we can use testing as **part of the development**.

The developer can **fix defects** as they are introduced, minimizing feedback cycles and cost.

Automated testing simply means that the activity is reduced to pushing a button.

If we do testing **after** development, we're missing out.

What makes TDD different?

So let's make it an assumption that we should test during development.

Still, knowing ourselves, we have the feeling that testing is going to be de-prioritized and not done at all.

So, this is what we do:

- Write **one** test **first**.
- Then write the implementation code to pass the test.
- Repeat.

What makes TDD different?

This is the "**driven**" part of Test-**Driven** Development. We let each test come first, and drive implementation.

We apply discipline on ourselves not to cheat and write too many tests or extra implementation code.

"*Never write a line of code without a broken test case*."

- Kent Beck (Father of TDD and Junit)

Benefits of TDD: safety

TDD gives us:

- Better design
- Rapid feedback and validation
- Shorter debugging
- Provides documentation
- Provides tests as a byproduct

Ignorance

Benefits of TDD: knowledge crunching

Slowness in a project is often due to us not knowing the domain, the constraints, and the tooling.

Paradoxically, we often attempt to make all the important decisions on **day 1** of a project:

- Implementation language
- Frameworks
- Database backend
- ORM (Object Relational Mapping)

Benefits of TDD: knowledge crunching

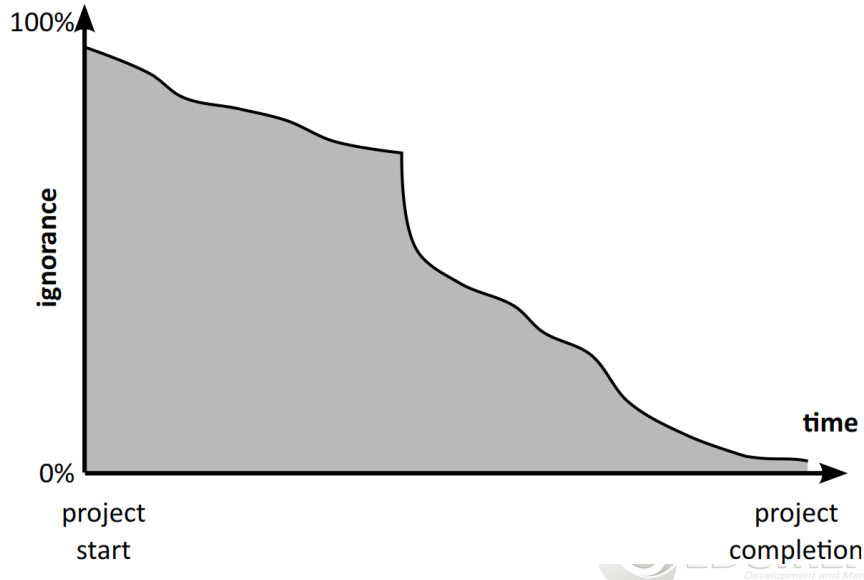
We also have a tendency to **delay those things** that would bring insight into what we are building:

- Throwaway prototypes
- First live demo

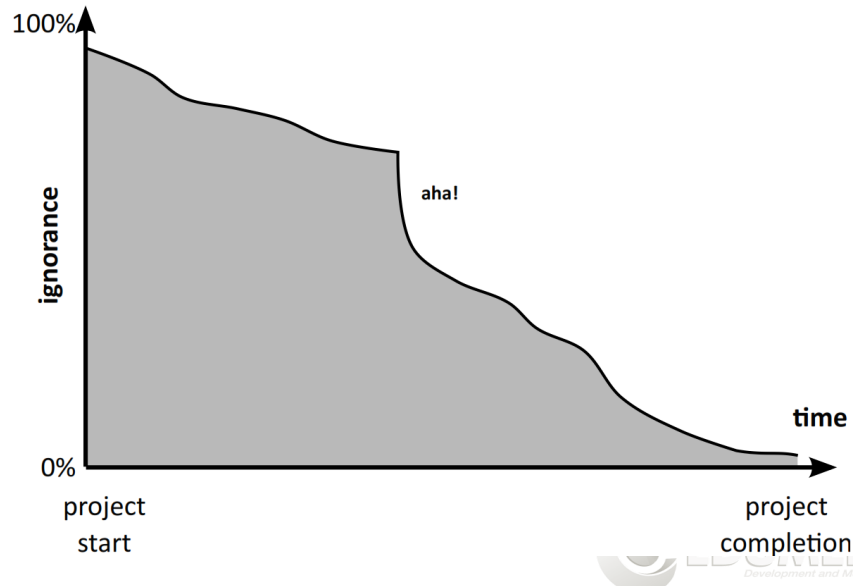
Just for fun, let's treat **ignorance** as an entity and see how it tends to evolve during the course of a project.

Ignorance is the single greatest impediment to throughput.
-- Dan North

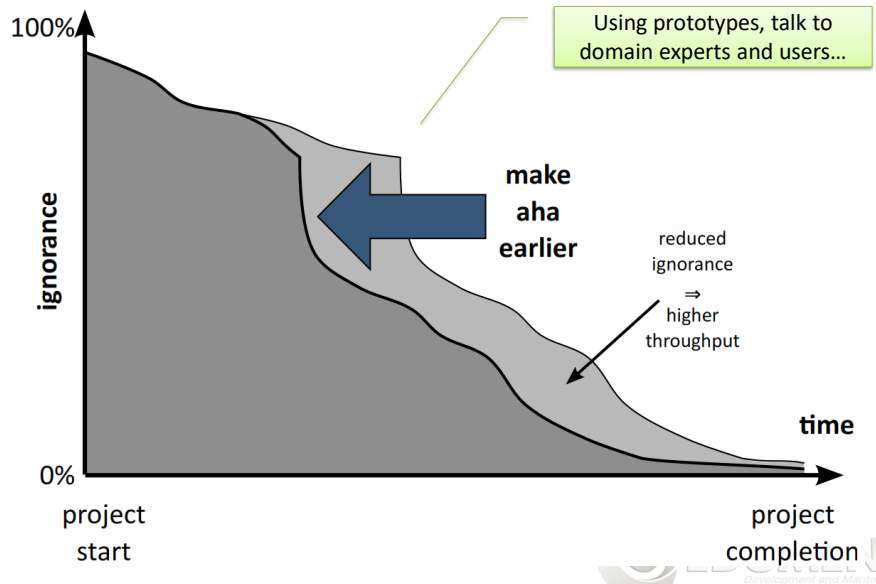
Ignorance curve 1/3



Ignorance curve 2/3



Ignorance curve 3/3



Intent

Intent

Benefits of TDD: intent

Programming is only half telling a computer what instructions to follow. It's also **communication**, explaining through code what a system is meant to do.

without intent

with intent

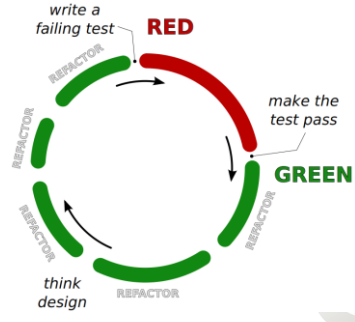
Tests communicate **intent**.

They are more exact than words on paper, because test can be **run**, and observed to pass or fail. In this sense, they act as a kind of **executable specification**.

Why red/green/refactor?

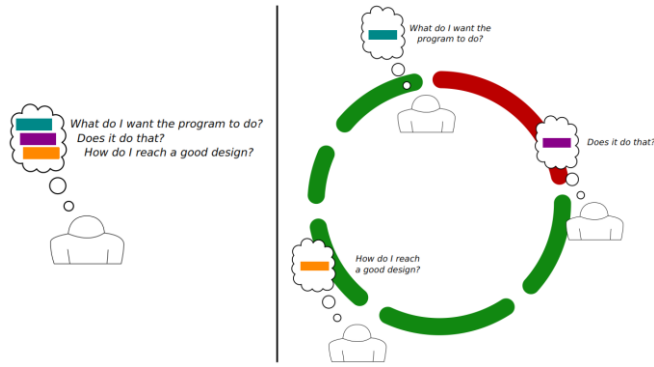
We tend to expand our test/implement model into three steps:

1. Write a failing test (red)
2. Make the test pass (green)
3. Reorganize code while keeping tests passing (refactor)



Why red/green/refactor?

Besides always knowing what to do next, **red/green/refactor** offloads your mind, allowing you to think about one thing at a time.



These are all important questions about the code. It's a relief to handle them separately.

Why not red/green/refactor?

The process on the last slide is sometimes called **Test-First Development** (TFD). TFD is like a "radical TDD". It tells you

"in order to be a true believer, you **must** follow these steps".

In real life, TFD isn't always practical. The red/green/refactor parts of the cycle sometimes gets reordered.

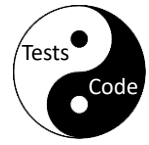
Quis custodiet ipsos custodes?

Some people ask:

- How do I know my tests are correct?
- Do I need to test my tests?

They probably imagine an infinite stack of tests, with diminishing returns quickly setting in.

Tests and implementation are **two aspects** of your software, each supporting and strengthening the other.



Types of test

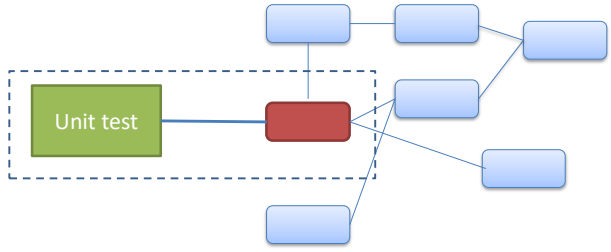
There are lots of types of test:

Test	Description
Unit test	test a small part. What will mostly concern us in this course.
Regression test	Make sure the change to a system doesn't introduce any new bugs.
Non-regression test	Make sure the change to a system is working as expected. Meaning we test the change it self.
Integration test	make sure that components work together.
Acceptance test	check whether requirements are met. When people talk about BDD, they mean these.

It's not unusual for tests to fill several roles, or to go from one role to another in their lifetime

Unit tests

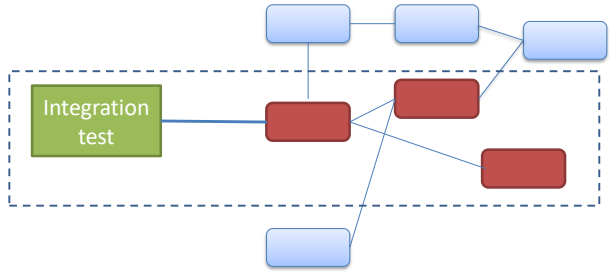
A **unit test** is used to test a small part of the system in isolation.



All **dependencies** should be abstracted away and not interfere with the test.

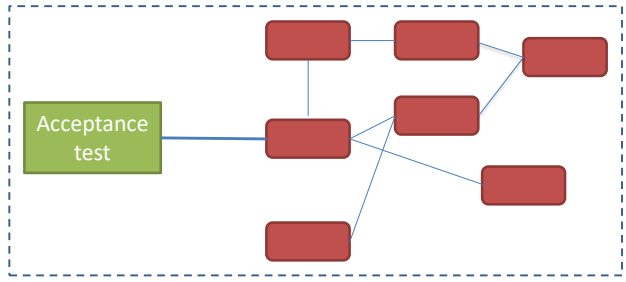
Integration tests

An **integration test** is used to test the interaction and integration between several modules or units in a system.



Integration tests

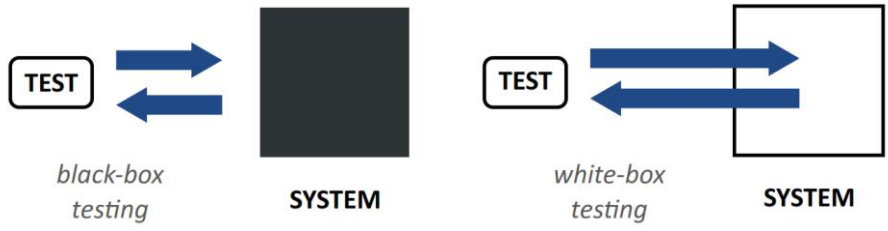
The **acceptance test** is used to test entire system, usually involving the customer.



The **acceptance test** is used to test entire system.

Types of test

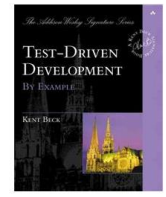
There's also **black-box testing** and **white-box testing**, depending on how much knowledge of the system we mix into the tests.



Book recommendation

Kent Beck is an authority on testing and TDD, if not **the** authority. He wrote a book on TDD that everyone should read, and preferably re-read every two years or so.

- **Test Driven Development: By Example**
  By: Kent Beck
- ISBN: 9780321146533



As books go, we know of no better introduction to TDD than this. It's clear, concrete, and simple.

Book recommendation

- **Working Effectively with Unit Tests**
- By: Jay Fields
  https://leanpub.com/wewut

- **Growing Object-Oriented Software, Guided by Tests**
- By Steve Freeman, Nat Pryce

- **Effective Unit Testing: A guide for Java developers**
- Lasse Koskela