Agile Processes

In the past two decades there has been a move towards lightweight software development processes, collectively known as **Agile** processes.

Waterfall tries to **prevent** new requirements by negotiating a contract upfront.

Waterfall tries to **prevent** error by adding more process.

Agile **accepts** new requirements and errors occur, and remains flexible (agile) so as to be able to deal with them.

Agile Manifesto

The **Agile Manifesto** was proposed in 2001 by a number of people, including well-known names like **Kent Beck, Robert Martin, Ward Cunningham, Dave Thomas**.

These were real software people, trying to express how software was actually being developed.

In a sense, Agile was attempt to create a scalable methodology from code&fix.

Agile Manifesto

We are uncovering better ways of developing software by doing it and helping others do it. Through this work we have come to value:

individuals and interactions *over processes and tools*
working software *over comprehensive documentation*
customer collaboration *over contract negotiation*
responding to change *over following a plan*

That is, while there is value in the items on the right, we value the items on the left more.

12 Principles of Agile

1. Our highest priority is to satisfy the customer through **early and continuous delivery of valuable software**.

2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.

3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

12 Principles of Agile

4. Business people and developers must work together daily throughout the project.

5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.

Note: Some of notions are dated, e.g. "face-to-face conversations" have been very successfully replaced by online interactions in physically distributed teams.

12 Principles of Agile

7. Working software is the primary measure of progress.

8. Agile processes promote **sustainable development**. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to **technical excellence** and **good design** enhances agility.

12 Principles of Agile

10. Simplicity – the art of maximising the amount of work not done – is essential.

11. The best architectures, requirements, and designs emerge from **self-organising teams**.

12. At regular intervals, the team **reflects on how to become more effective**, then tunes and adjusts its behaviour accordingly.

Agile Principles Summarised

Deliver valuable software regularly.

Be open always to new requirements.

No barriers between product team and developers.

Motivated developers.

Communication.

Well-designed, working software.

Little documentation.

Sustainable development.

Self-organising teams.

Reflection.

Agile Processes

There are many agile processes. Three of the best known are:

Scrum: This is a pure agile process, not only for software We'll look at it in a bit more detail.

Extreme Programming: Very much focused on software. We'll look at some of its widely-accepted engineering practices in more detail.

Kanban: Very lightweight. More suitable for "endless" processes like services, also suits better when there is little customer involvement.

Scrum

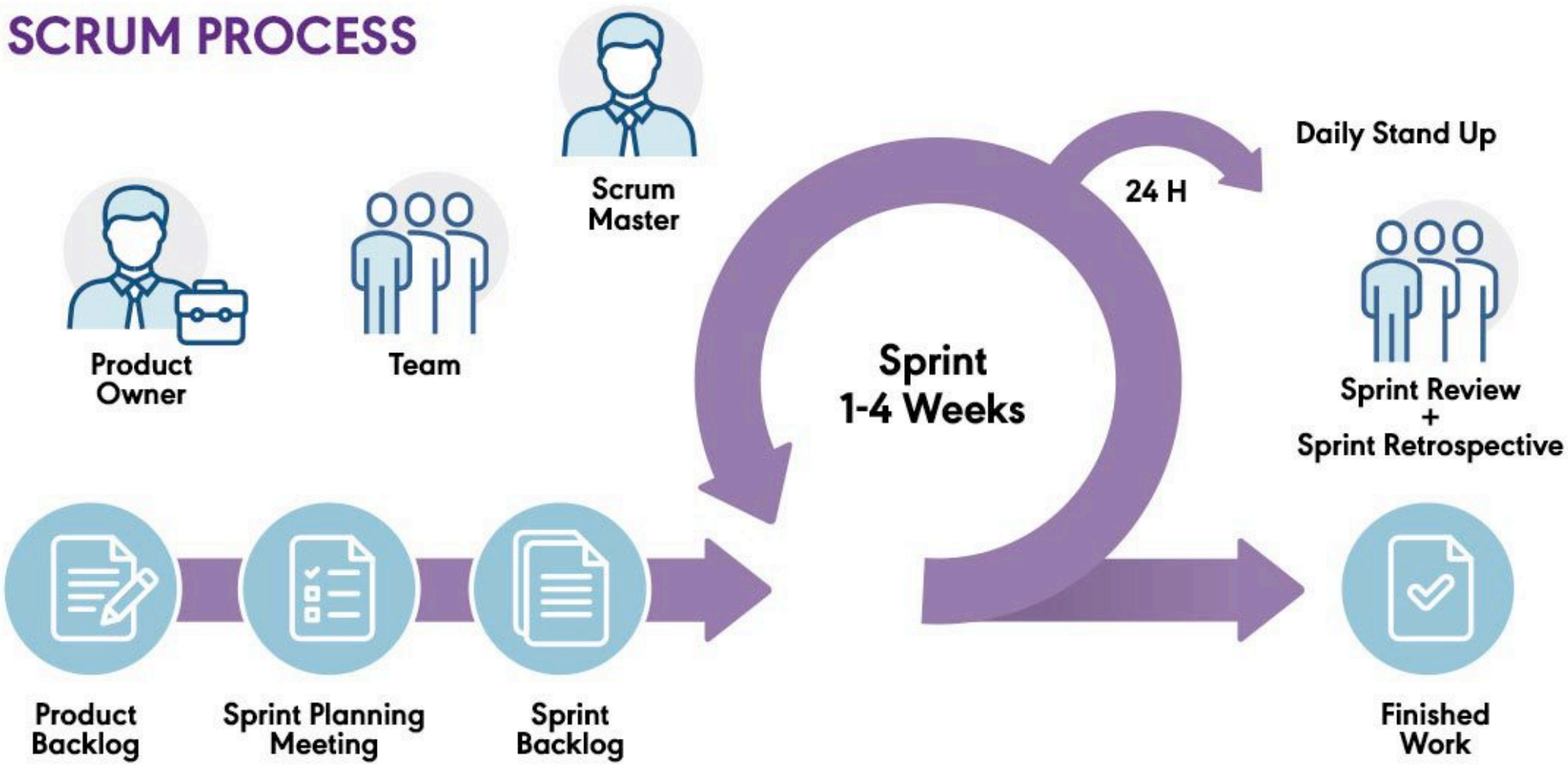
First proposed in 1995.

Scrum is very commercialised and the area is heavily trademarked.

Scrum is unrelated to software -- you can use it to build non-software artefacts as well.

It's become the most popular agile process.

Overview of Scrum



Thanks to pm-partners for the image.

Scrum Roles

Product Owner: Non-technical role. Has a clear **vision** for the product being developed. Writes user stories, answers team's product-related questions.

Scrum Master: A Scrum expert. Ensures that the team adheres to the Scrum framework. May also be a developer.

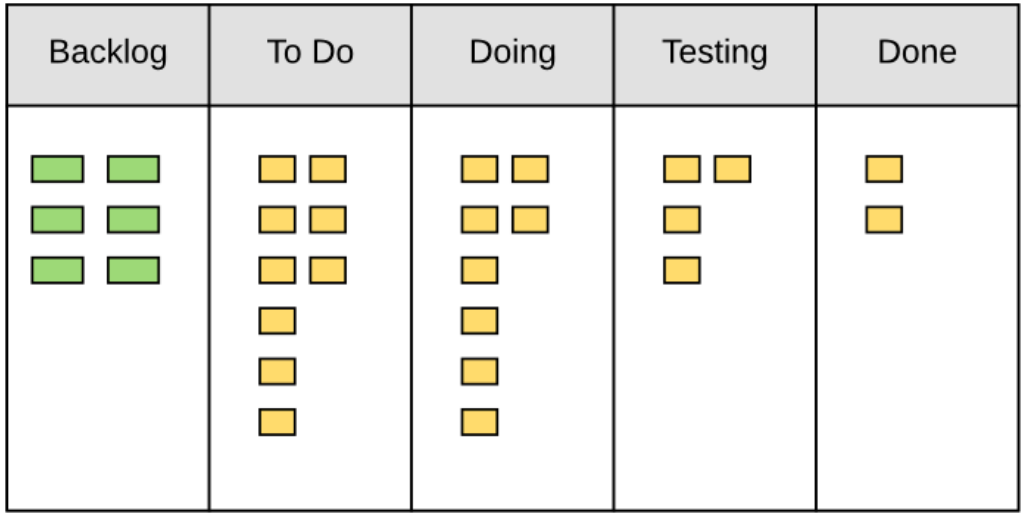
Software Developer: A software expert focussed on implementing the user stories written by the Product Owner.

Scrum Artifacts

Product Backlog: Prioritised list of user stories to be implemented, possibly other tasks as well.

Sprint Backlog: A subset of the product backlog to be implemented on this sprint, decomposed into a set of tasks.

Scrum Board: A visual representation of the current state of the sprint, e.g.



Scrum Events

Sprint: A time-boxed iteration (~2 weeks) leading to potentially a new software release. Aka **iteration**.

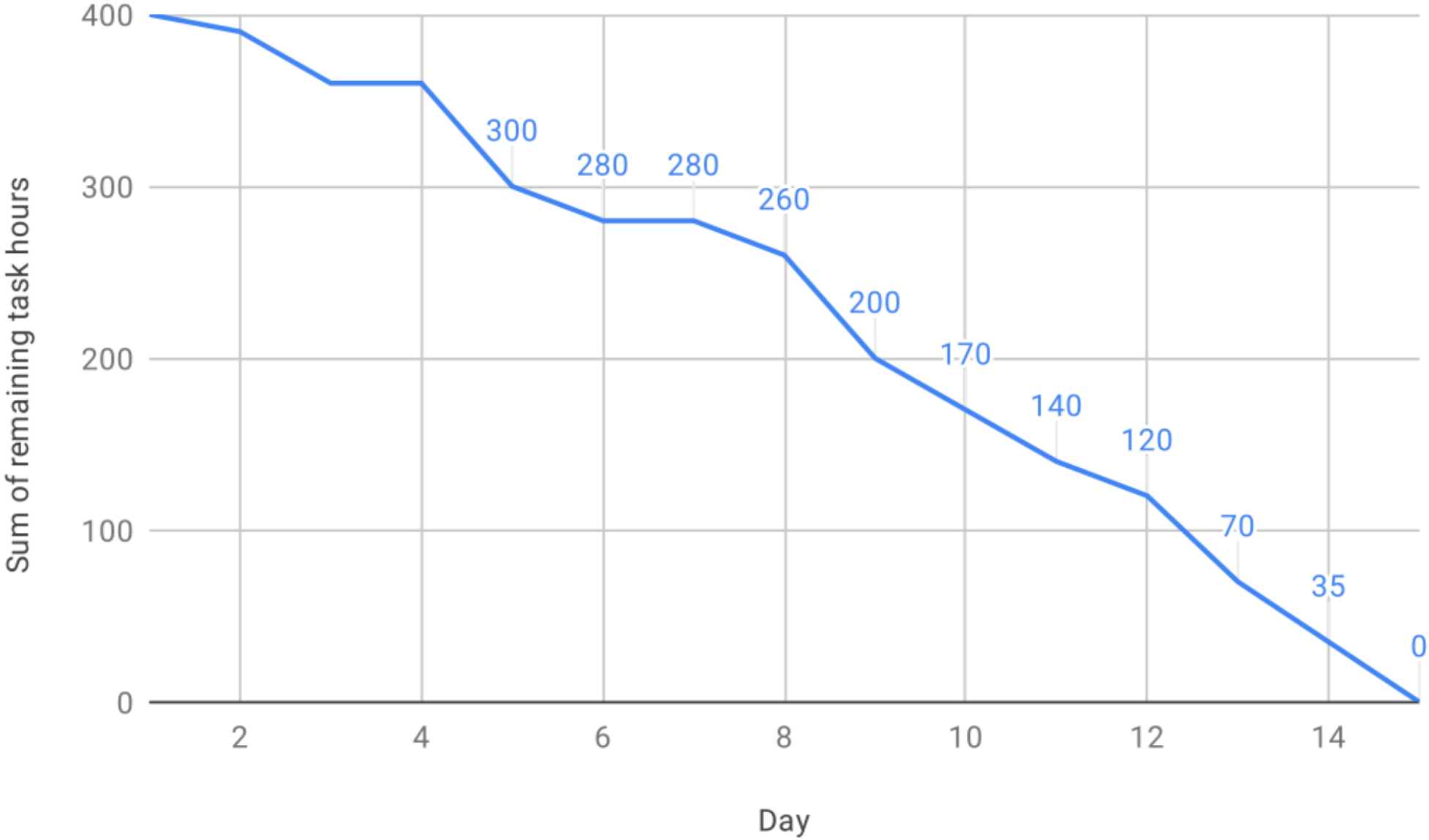
Sprint Planning: Meeting at start of sprint to decide on stories to be implemented, and their decomposition into tasks.

Stand-Up: *Short* daily meeting to start the day. Everyone summarises what they achieved yesterday, plans for today, and obstacles to progress.

Retrospective: Meeting to reflect on how the sprint went, how the team could do better etc.

Burndown Chart

Shows how much work is left to do to complete the tasks of the sprint, e.g.



Sample User Story

SWITCH WEAPONS WHILE SPRINTING

2 POINTS

AS A PLAYER,

I WANT TO SWITCH BETWEEN MY EQUIPPED WEAPONS WHILE SPRINTING

SO THAT I DON'T HAVE TO STOP TO SEE MY INVENTORY

Velocity

Each story has an associated number of points representing the difficulty in implementing it.

The number of points is assigned by the developers during sprint planning.

The **velocity** of the team is the number of story points it can complete during a sprint.

Obviously this is not an exact measure, but it gives a rough idea of how the team is doing.

More importantly, it informs the Product Owner how many stories they can expect the team to implement in a sprint.

Other Aspect of Scrum/Agile

Cross-functional Teams: Team must have all the expertise required to deliver functionality.

"done": The notion of "done" must be defined. Implementation must at least pass all tests, may also be other QA/UI/UX etc. issues

Servant Manager: Managers are not traditional bosses. Their main role is to facilitate the dev team.

Framework: Scrum is really a tailorable framework, not a fully-specified process.

But how about the software?

Scrum is often used on software projects, but is not intended just for software.

This is why it has very little to say about critical issues like testing, quality, usability, security etc.

Extreme Programming (XP) is very much about software.

We now look at some engineering practices, mostly from XP, which are mainstream in all agile processes.

Let's consider what happens within a sprint...

No BDUF

Little or no upfront design is performed.

UML diagrams may be drawn, but only for the purpose of immediate communication. Design documentation is not created or maintained.

Everything is kept as simple as can be. Several Agile acronyms reflect this philosophy:

YAGNI: You Ain't Gonna Need It

KISS: Keep it Simple Stupid

TSTTCPW: The Simplest Thing That Could Possibly Work

Testing

Testing is essential -- it's how we know that our software works.

Our focus here is on **unit testing** -- the tests a developer creates as they work.

The test suite can be run at any time to provide assurance that the existing code is still correct.

(XP goes a step further and demands that tests be written first. More on this later.)

We'll come back to **testing** later in the module.

Unit tests for correctness

The test suite is actually an **executable specification** of what the system is to do.

Contrast this with a complex design where it's impossible to assess if a given program is a correct implementation or not.

The promotion of testing to this level of importance has a vital characteristic that can be exploited:

> Whenever the code is updated, it is easy to test if a new bug has been introduced.

How is the software designed?

So far we have a Scrum process, with developers coding and testing.

How can we build a large (~1M LOCs) software system by just coding and never considering software architecture and design?

We use an approach called **emergent design** or **pay-as-you-go design**.

Emergent Design

When a developer tackles a new story, they check if it can be implemented easily in the current program.

If not, they **change the design** of the software so that:

(1) it becomes easy to implement the new user story.

(2) the behaviour of the software remains unchanged

(1) makes the task at hand easy

(2) ensures that no new bugs have been created

This process is called **refactoring**.

Running the test suite after a refactoring is likely to detect any new bugs.

We'll come back to **refactoring** later in the module.

Simple Example

You're tasked with implementing a story.

You figure out how to do it. It involves adding new functionality to a method that is already quite long.

Code&Fix: Just do it and move on!

Agile Development:

(1) Refactor: Split the long method into several shorter more cohesive methods.

(2) Test: Run your existing unit tests. This reassures you that nothing is broken.

(3) Implement: Code up the new story, writing unit tests for it of course.

We'll come back to **code smells** (incl. Long Method) software properties like **cohesion** later in the module.

Floss and *Radical* Refactoring

Ideally the software design emerges in baby steps as described on the previous slide.

Sometimes deeper problems with the software design are uncovered, requiring a more radical approach.

Radical refactoring occurs (rarely!) when the whole program design is overhauled.

This process may take a whole sprint and involve the entire development team.

Note: **Radical Refactoring** is aka **Root Canal** refactoring.

Agile Code

The code is the essential artefact produced in an agile process.

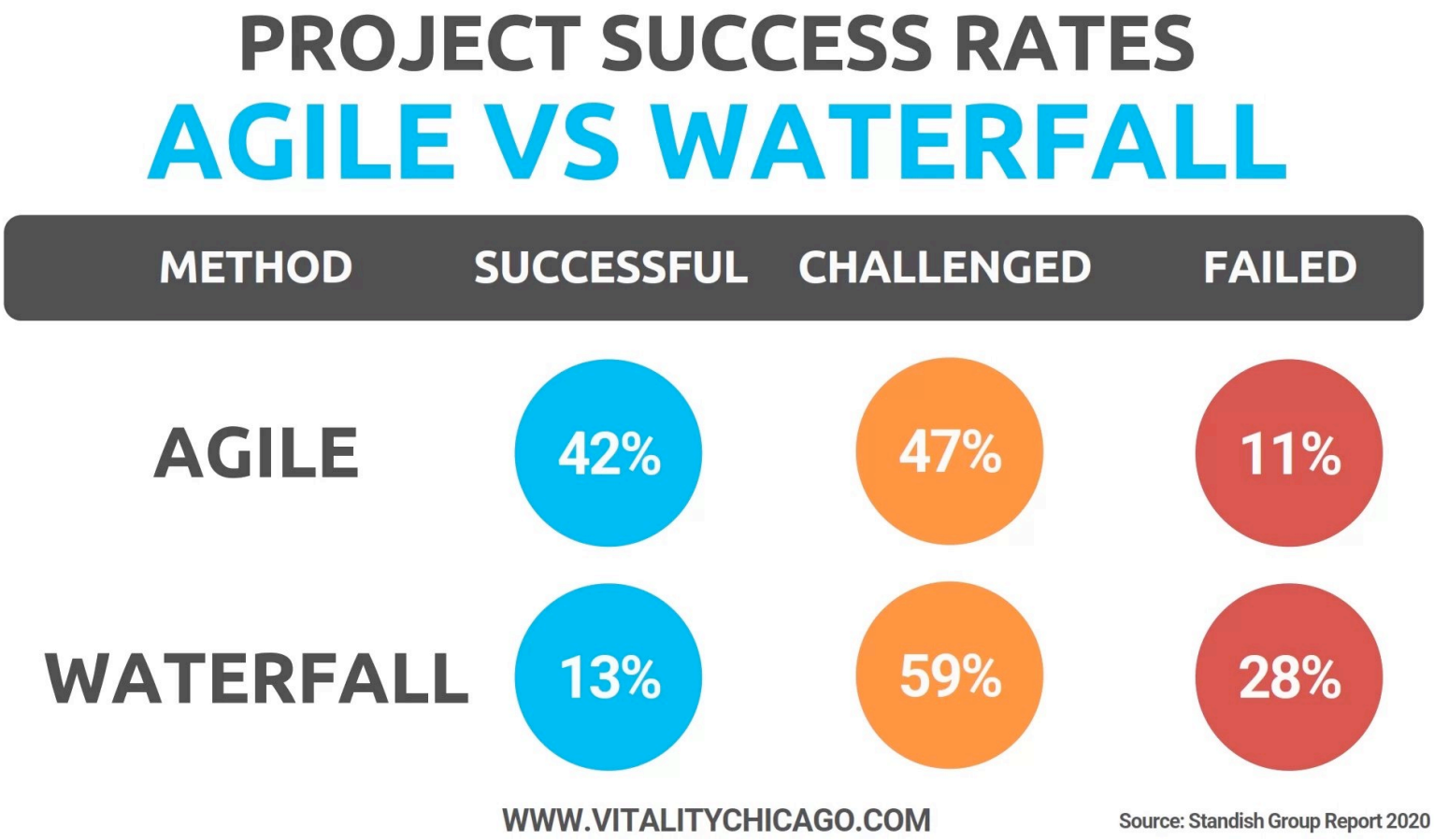
It has to be of high quality.

The **code is elegant and clear**. It has to be because developers work with it continually.

The **design is simple**. The design is only for the current system, not for possible futures.

The code works. This can be easily verified by running the test suite.

Does Agile work well?



From the so-called *CHAOS report*, an annual survey of IT projects in the US. To be taken with a large grain of salt.

Why does Agile work for software?

You can't build a house in an Agile way

"OMG this is so nice!"

"Just rotate it 30 degrees to catch the morning sun"

Software vs. Physical Construction

Waterfall

- Requirements
- Analysis
- Design
- Implementation
- Testing
- Release

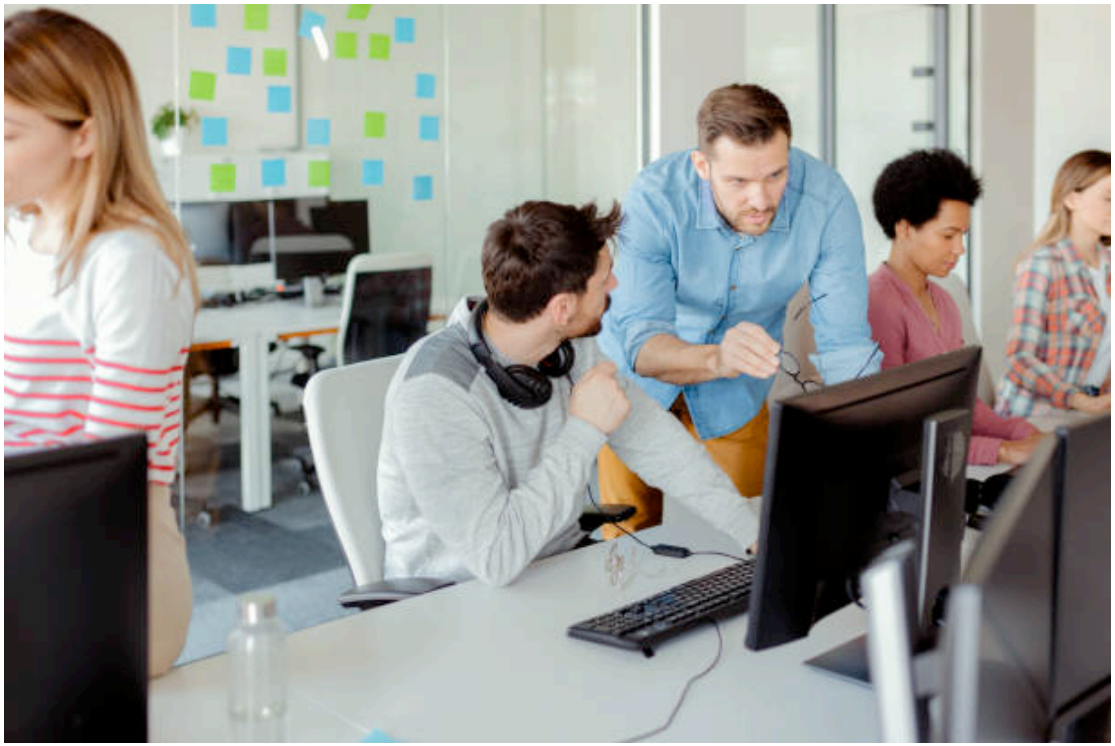
Physical Construction

- Strategic definition
- Prep. and brief
- Concept design
- Developed design
- Technical design
- Construction
- Handover and close

Yellow: Creating models (blueprints) on paper
Orange: Actually building the system
Green: Releasing the system to customer

The modelling & blueprint phases look similar!

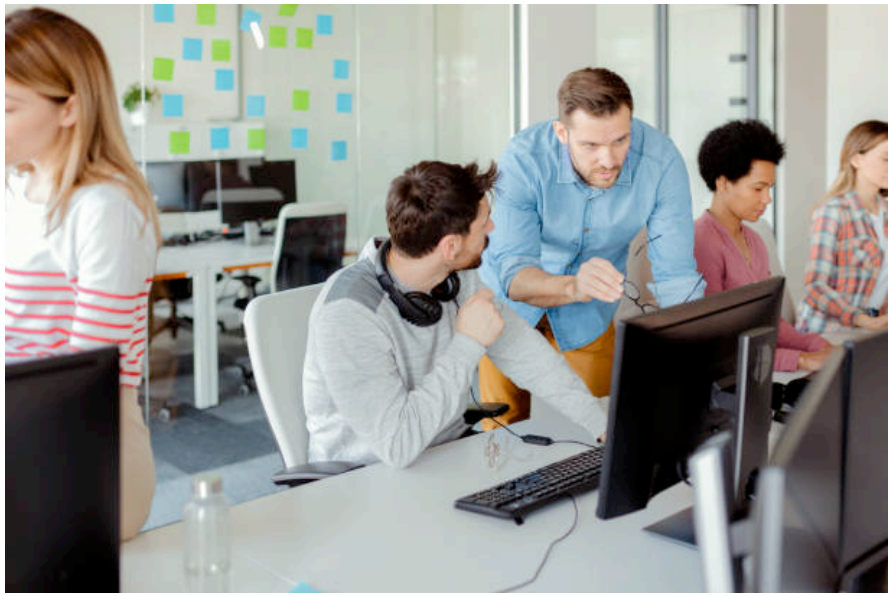
The modelling phases of both software and physical construction processes look similar



How about the construction and implementation phases?

Software Implementation vs. Physical Construction

Software Implementation



Physical Construction

In essence, the process of transforming our detailed blueprints (software) into what the customer wants (a running system) is fully automated!

This is what makes an agile approach possible in software.

Resources

Read the relevant parts of the recommended textbook: *Software Engineering: A Modern Approach* by Marco Tulio Valente.

http://www.agilelearninglabs.com/resources/scrum-introduction
is a clear introduction to Scrum.

https://www.atlassian.com/agile
Atlassian provide clear "no nonsense" tutorials on Agile.

Summary

Agile most suited to small teams (<=12 people) on projects where the requirements are likely to change frequently.

In ways, it is the antithesis of the Waterfall process.

Agile has worked well in practice, and no doubt this field will continue to evolve.