

Software Engineering

Software Process Models

(with a few reminders from our first two classes)

Common Practices

As you begin your career, you'll find that there are standard ways of organizing software development that are well known throughout the industry – common ways of working.

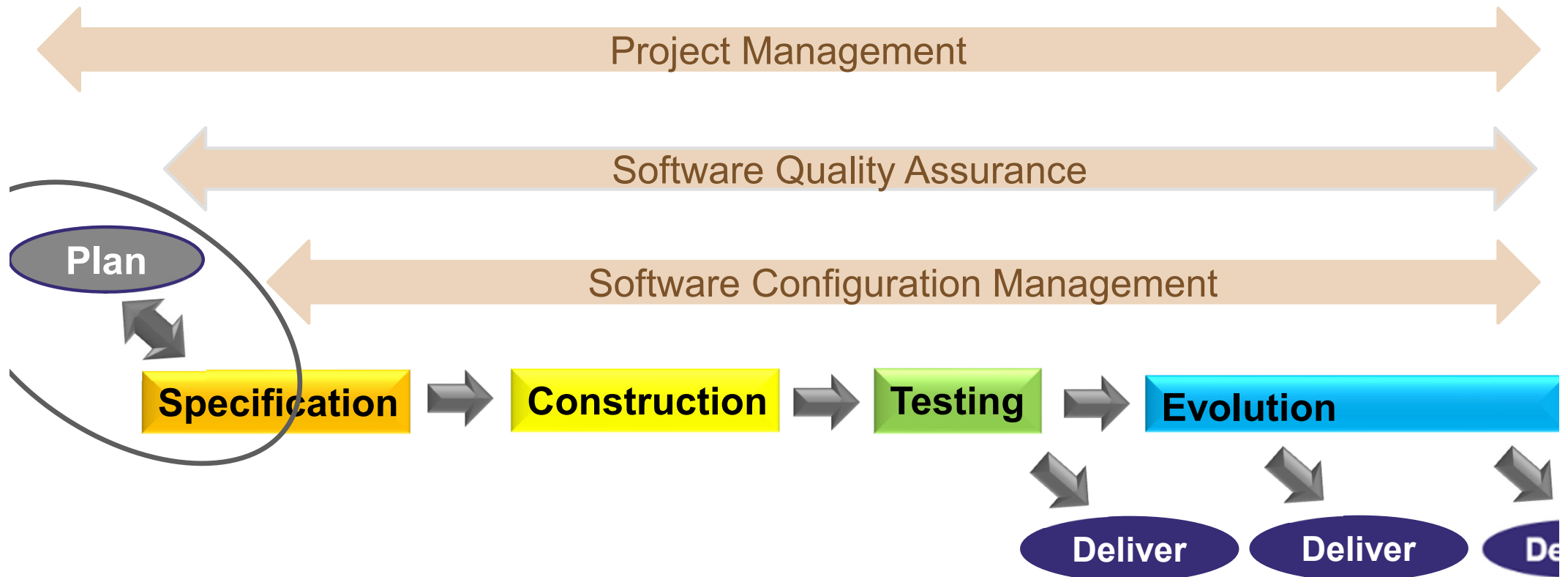
The resulting sequences of development activities and tasks that can be assigned, completed and measured is known as a **Software Development Lifecycle (SDLC)**.

Every company will have different ways of working, but you should be able to recognize and understand the general process model or mix of models that underlies their approach.

Later in your career you need to know how to select the most appropriate model for a product or project that will give you the best chance of success.

The next few slides will contrast two broad categories of models: **Predictive Models** and **Adaptive Models**.

The “obvious” model: Systematic. Fully planned up-front.
A linear sequential model.



This is the *Waterfall Model*

Waterfall is the basis for traditional approaches to development

Waterfall Model

Heavyweight documentation

Flows down from
Phase to Phase

+ formal
plans for all
activities

**Deliverable Artefacts
- Typical Milestones**

Requirements

Requirements Spec.

In its *rigid, strongly
controlled* form:

Design

Design Spec.

A phase *cannot* begin until the
previous phase is complete and
the delivered artefact is *accepted*.

Implementation

Code

A previous phase *cannot* be re-
entered once it has been declared
complete.

Testing

Test Reports

Each phase terminates in a major
milestone and deliverable. The
deliverable is reviewed to
determine whether the phase is
truly complete.

Deployment

Installed
System

Operation &
Maintenance

A major quality goal: make it “impossible”
for defects to leak into later phases.

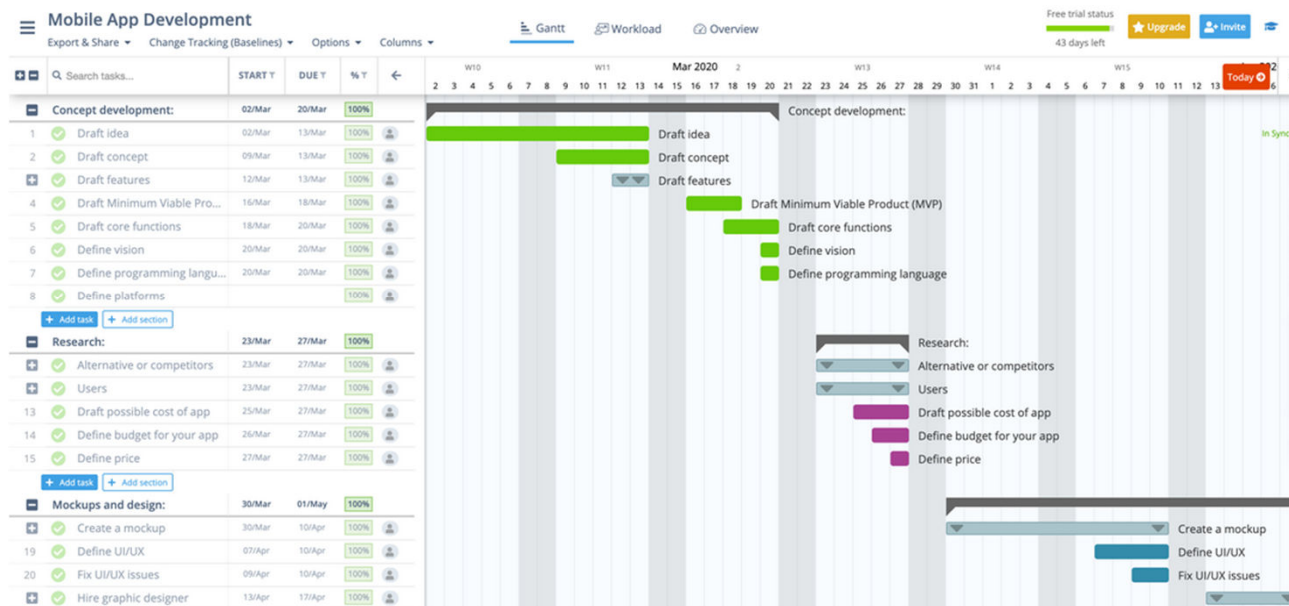
Waterfall is an example of a **predictive model**.

Assumption is that requirements can be known **completely** at the start of the project. And a **solution is clear from the start**.

So, based on past experience, we can **predict exactly what needs to be done** to develop the product and how long it will take.

Then we can produce **big, detailed plans** for every phase of development and drive the project according to those plans.

Hence the big, detailed project Gantt charts produced by project managers:



[Instagantt]

Many regulatory standards reflect this thinking

Example: DO-178C required data (Airborne Software System Certification)

Plan for Software Aspects of Certification	Software Configuration Index
Software Quality Assurance Plan	Software Life Cycle Environment Configuration Index
Software Configuration Management Plan	Tool Qualification Document
Software Development Plan, Software Requirements Standard, Software Design Standard, Software Coding Standard	Software Development Folder
Software Verification Plan	Traceability Matrix
Software Test Plan	Software Accomplishment Summary
Software Requirements Specification	Sources
Tool Requirements Document	Results
Software Design Document	Libraries

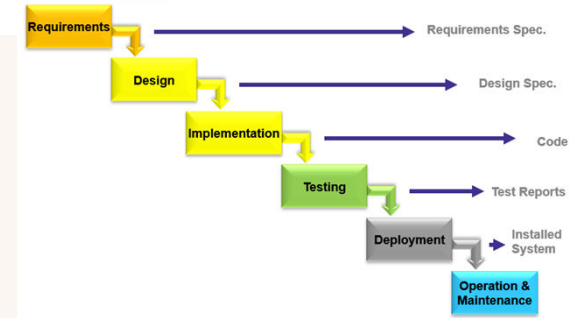
How much do you think this development and certification effort costs?

For example, in HKD per 100 LOC?



Why did predictive models like Waterfall dominate the industry?

Pros:



- Better than nothing! Much better!
- Easy to understand
- Allow a high level of management control. Easy to verify at each stage.
Well-defined deliverables work well in a contracting environment.
- Makes clearly visible exactly *when* different activities take place and when the project will be completed. Separation of concerns makes it easy to schedule specialized teams for different phases.
- Customers know exactly *what* will be delivered – *no surprises*. Important for mission-critical and safety-critical software, which are often specified in extreme detail.
- No need to waste time and effort on experimentation and prototyping.
- In theory, no need to work to prevent the design degrading during development. The design is never modified.

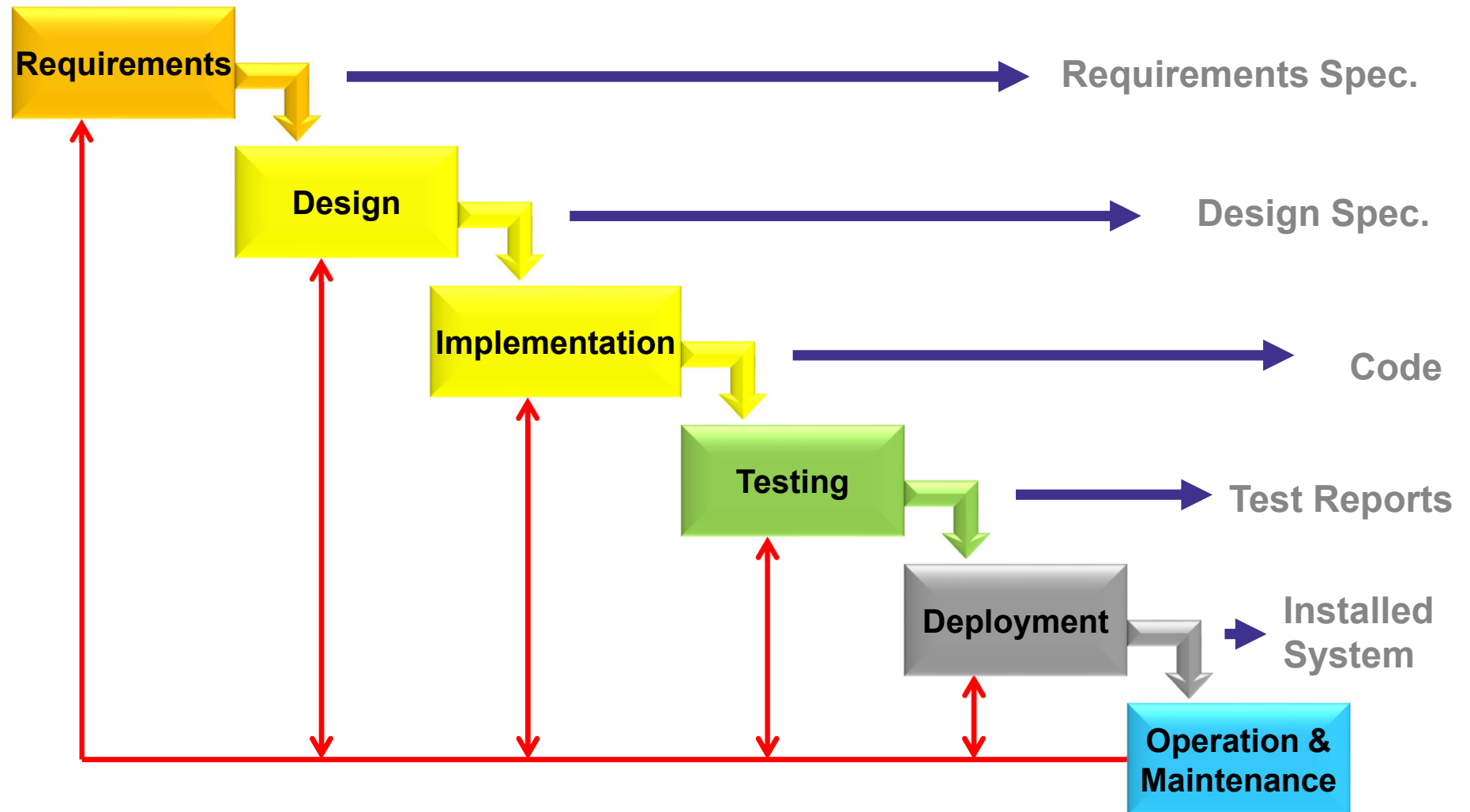
But there are problems with predictive models

Cons

- **They don't work** for innovative, complex software development. They can't handle change or risk well.
 - Can't work if customers' and developers' vision of the product are different or the requirements are incomplete.
 - If you were wrong about requirements change, then it is a **disaster**. Lack of flexibility in the process makes it very difficult to accommodate changes in requirements.
- When you choose a predictive model, you are *gambling* that there will not be major changes.
- Late validation.
 - Actually *does* lead to badly structured systems because of workarounds needed in code resulting from prematurely frozen designs.

An improvement: Waterfall Model with Feedback

In practice, need feedback since problems *are* discovered in later phases.



But this involves so much **expensive rework** of documents and repeated steps that artefacts are typically frozen after **limited feedback**. Still inflexible.

Late validation

Predictive models *do* bring discipline to development. But a **major** problem is *late validation*.

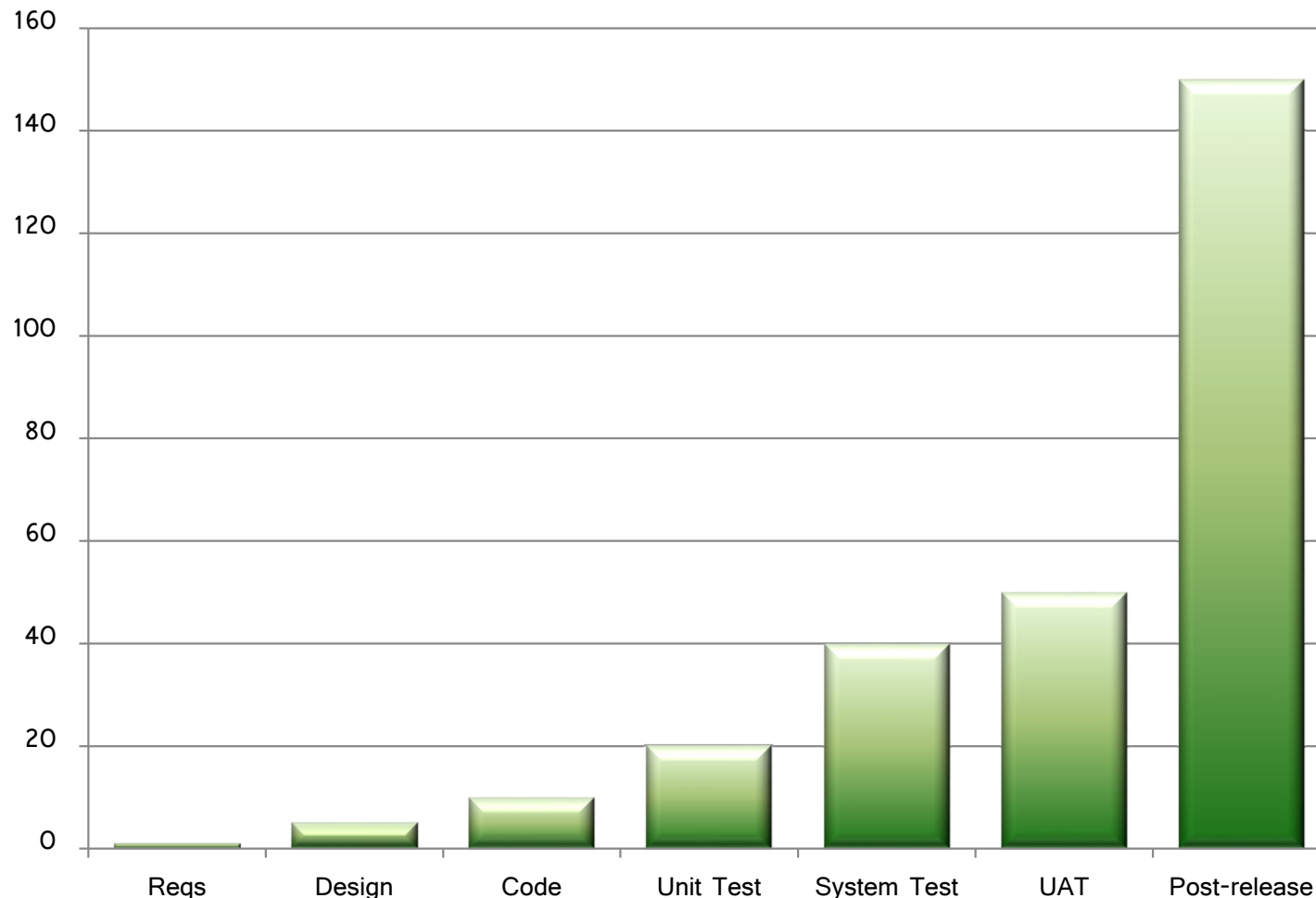
Q: When do customers and end-users see any working features?

A: *Very late in the project.*

No working product is delivered to customer until the *end of the project*. We assume that everything will integrate successfully to produce a system that satisfies users' needs.

A mistake made in the Requirements Phase can stay undetected for a long time. That makes it very *expensive to fix* when it is finally discovered.

Relative cost of fixing a requirements defect at various stages of development



Remember, Waterfall was an attempt to prevent defects reaching the late stages of a project, but it often had the opposite effect and allowed defects to hide for long periods.

When might predictive models be a good choice?

When requirements are well understood and unlikely to change during development, and risks of failing to satisfy them are low.

- When requirements *really are* known in advance
- There *really are* no significant risks in the project
- Team has development experience of this kind of project
- The schedule is known to be long enough to complete all work

So, what sort of project could this be?

Predictive models use Defined Process Control

Models like Waterfall specify *in detail* the steps/tasks/activities, and their ordering, needed to develop the software product.

They are really *manufacturing* models, like those used in industrial mass-production. They treat software development like manufacturing and assume that:

- all knowledge is available up-front, and
- all steps can be fixed and are repeatable with little variation, leading to predictable outcomes.

Defined Process Control for Software?

Defined process control is **open loop**
defined process



- But we do not have perfect knowledge up-front, just assumptions.
- And the **process model** forces us to make decisions based on those assumptions without the benefit of feedback based on delivered, working software.

Cannot work:

- for complex problems where the **solution is not known** in advance,
- for projects where user needs and requirements are changing.

What do we really need?

For types of products that were not a good fit for predictive approaches, real-world teams that were consistently successful used different approaches:

- ❑ self-organize developers into motivated, cooperative teams and collaborate closely rather than working in isolation as individuals or specialized groups
- ❑ produce working increments frequently rather than trying to integrate large increments at longer timescales
- ❑ continuously revise their understanding of requirements
- ❑ quickly respond to change when necessary

These are characteristics of a *lighter* and *adaptable* approach to development compared with the *heavyweight*, *rigidly planned* and tightly managed predictive approaches of Traditional Development.

We saw, this is the **Agile** approach to software development.

Adaptive Models

Software is soft – it can be changed and adapted.
Used properly, that is a strength!

Plan and Document models view change as the enemy and try to *prevent* any need for it.

Adaptive models “*embrace change*” since it will result in a better solution.

Adaptive Models accept that requirements and/or solution *cannot* be known for certain in advance, and that change is extremely likely during development.

Planning continues throughout development allowing the work to adapt to changes when they appear.

That is Empirical Process Control

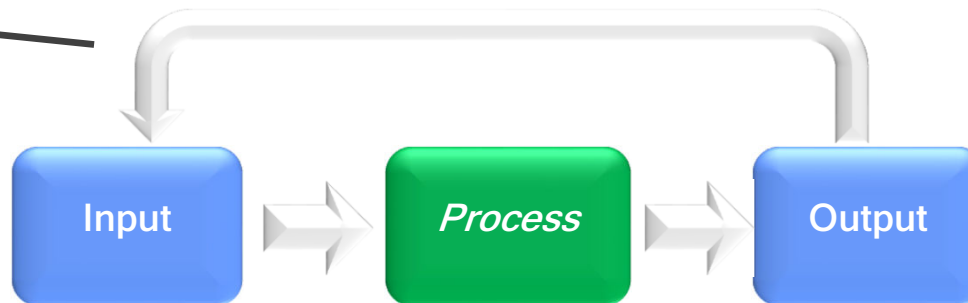
not defined

This form of control is best for complex problems where there is more that we *don't* know than we *do* know.

➡ New knowledge comes from experience.

⬇ ⬆
Then make decisions based on that new knowledge.

Closing the loop



Empirical control is built on three principles which form the basis for Agile frameworks like Scrum:

- Transparency
- Inspection
- Adaptation

Principles

Transparency

All information that is important for developing the product must be available to **everyone** connected to development.

Nothing is **hidden** and there must be common understanding. This enables inspection and, hence, adaptation.

Inspection

Frequently review the **current state** of the work, of progress, and of use of the process framework itself.

Adaptation

On the basis of inspection, make changes where necessary to **adapt** the product and/or process and so bring aspects that have deviated back under control.

Frequency of inspection and adaptation depends on the **amount of risk** we want to take.

So, how to organize technical activities?

Can't escape the Waterfall-like dependencies among the software development activities, but we *do* know that Waterfall can work on small projects with well-defined, stable requirements.

Simple idea: Develop in a sequence of very small, self-contained "mini-projects" – these are *iterations* of development.

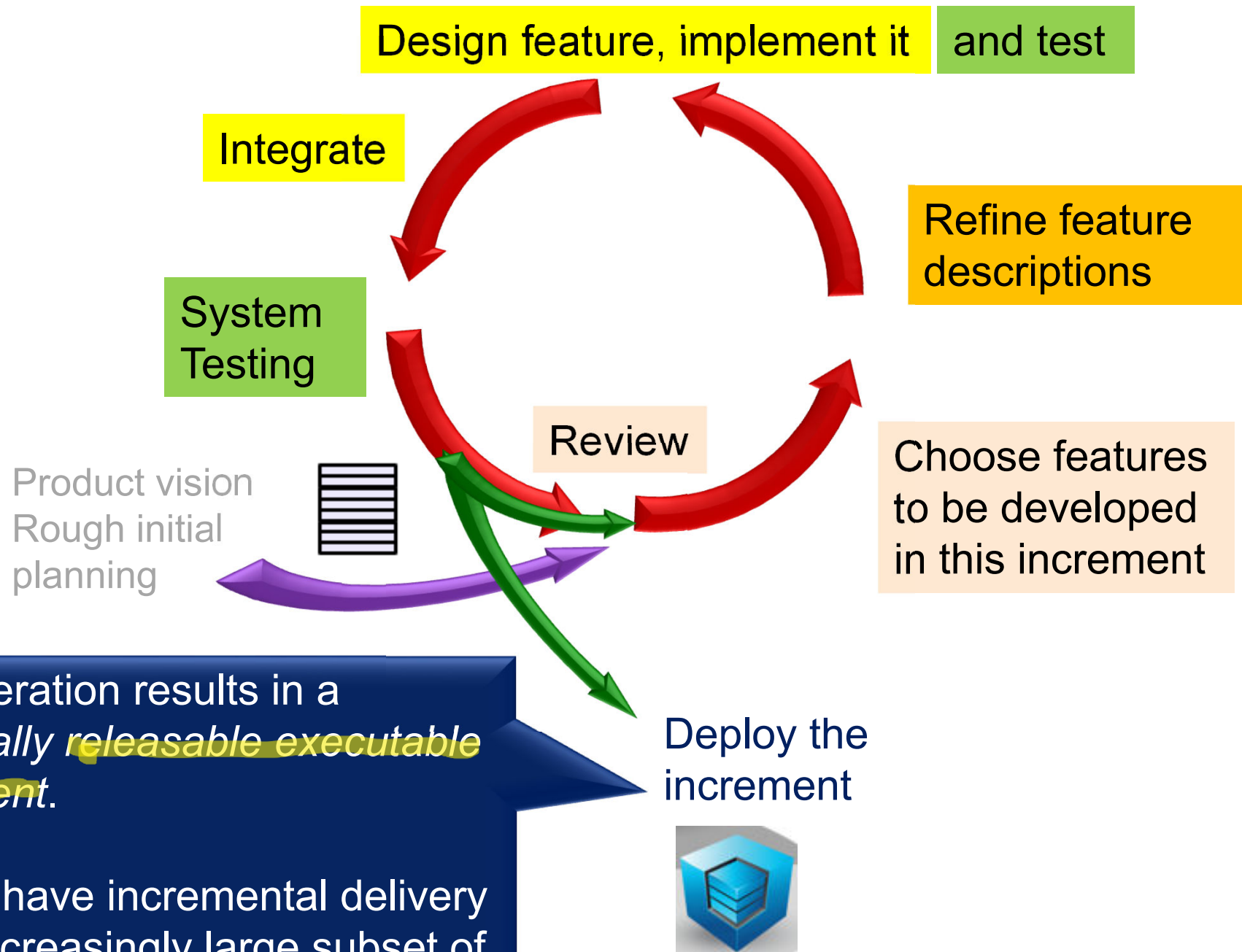
Roughly, each "mini-project" passes through the same set of development activities and builds the piece of the product that is currently of most value and/or highest risk. The overall project *iterates* through that cycle repeatedly.

As we saw previously, each iteration builds on the product of the previous iteration. So, the system grows *incrementally*.

Iterative and Incremental Process

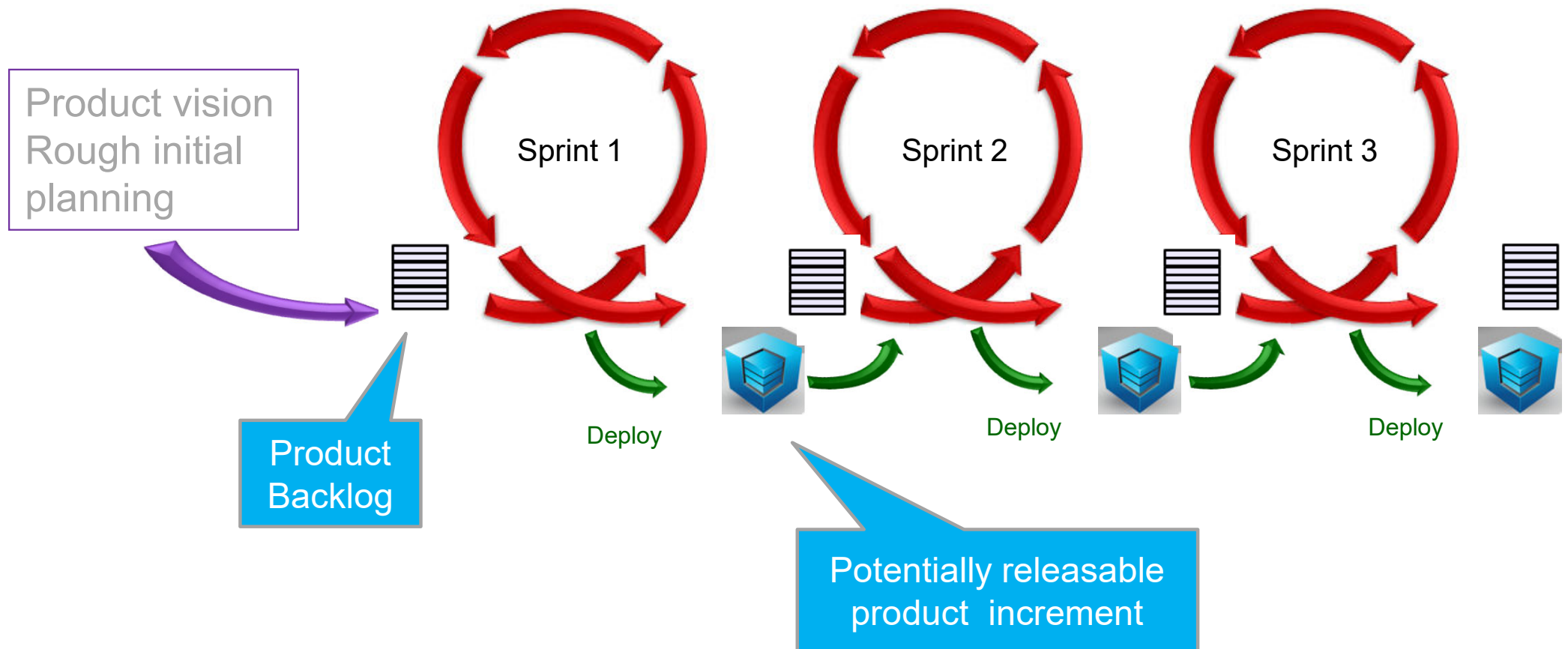


Generic iterative cycle



Development is a series of these iterations


Short timeboxed iterations of the same length.
Called **Sprints** In Scrum, and usually 1-2 weeks long.



Features of Iterative and Incremental Process

- Each iteration has *well-defined goals*. But these are not set until planning *at the start of the iteration*. Helps us handle change.
No massive, *heavyweight up-front planning and commitment*.
- Every iteration delivers a *working release* – a stable, integrated, tested, partially complete *but releasable* system. This is our *proof* that we *achieved the goals of the iteration* and is a basis for involving the customer.
- As the system grows, increment by increment, it is a *working product at every stage*. Potentially, we can release at any time, with the current subset of highest-value features.

Adaptive Process Models handle change and product complexity better than Predictive Models, **BUT**, these models tend to be more complicated to follow and *may not be worth the effort* for *simple, low risk development*.



Product-based vs. Project-based Development

For much of its history, Software Engineering focused on methods and techniques for developing *large, custom software applications*.

Those applications were developed as *projects*. So, Software Engineering's methods and techniques were principally designed to support project-based development.

Can be seen in Waterfall – the “completed” application and its future evolution becomes the responsibility of the customer after delivery.

Current reality:

used to be project
now product

Working software engineers (you!) are most likely to be involved in *software product development*. Requires a shift to product-based approaches.

Custom software vs. Software products

Most professionally-developed software used to be custom software developed and deployed for a specific client – often the government or a large company. Usually, the purchaser would then own the code.

Typical examples (a random selection) would support:

- country-wide health-care record management.
- air traffic control
- online banking
- core banking
- utility billing
- field service management (e.g. MTRC)
- military command and control

May be developed by in-house teams or external custom-software development companies. In both cases there is a “client/customer” who brings the problem.

Custom software vs. Software products

Many customers don't need fully-custom software. **Generic software packages** are sufficient to solve their problems. Possibly with minor customization.

Generic packages are examples of *software products*. Other examples:

- mass-market apps (e.g. Adobe products).
- games
- productivity tools and suites (e.g. MS Word, MS Office)
- social networking services and platforms
- CRM systems

May be developed as a generalization of custom software, but a key difference is:

Software products do not have a **specific client** as the source of requirements.

No *specific* client will determine whether the product represents an acceptable solution to the problem.

Software Products: Process

Time to market is often critical for new products or new feature releases.

You will often be **racing against competitors** or have a **release date** you can't miss.

Traditional:

We saw that for large, safety- or mission-critical projects, the overheads of traditional process models can be accepted. As a result of those overheads, **initial development can take many years.**

Software product development cannot tolerate such overheads. **Speed is essential** and agile process is used almost universally.

Fortunately, agile is a good match – products are usually developed by a single team working in the same location, reducing the need for documentation, and reducing communication and coordination overheads.

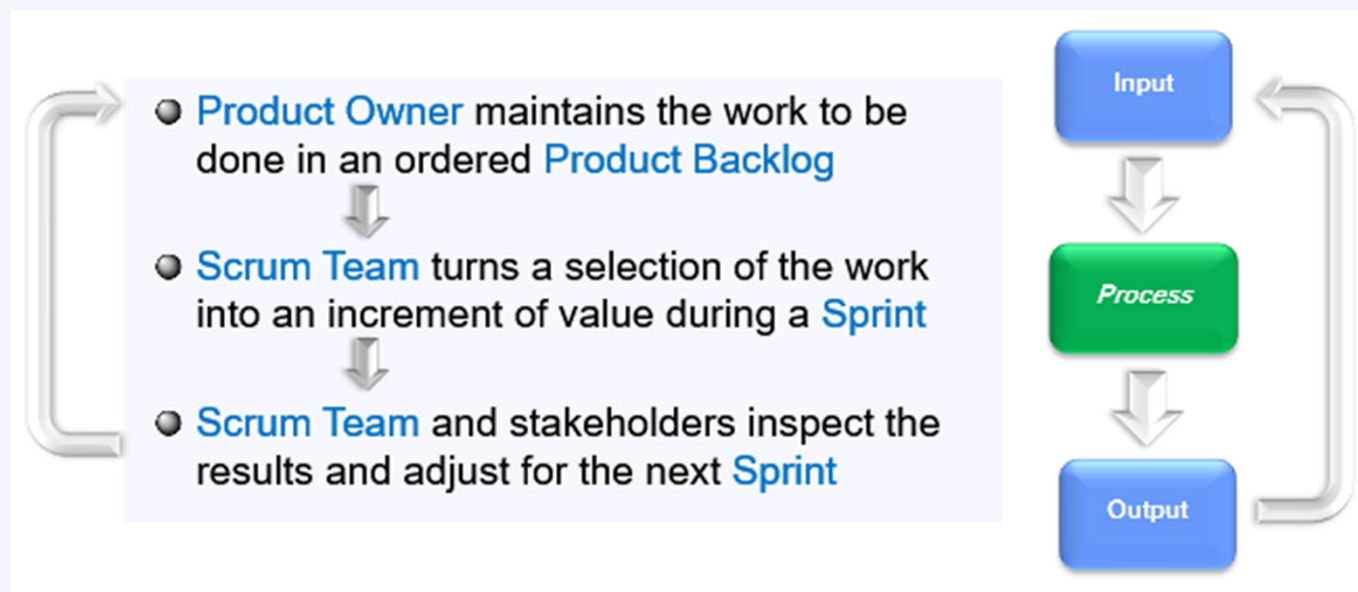
Software Products: Management

Agile teams are **self-managed**. There is no project manager directing the team.

The product backlog is maintained and prioritized by a **particular member** of the team. In Scrum, this is the **Product Owner**. More generally, the responsibility belongs to the **Product Manager**.

If using Scrum, usually the **Product Manager** plays the role of **Product Owner**.

From a future slide set: **The Scrum cycle**

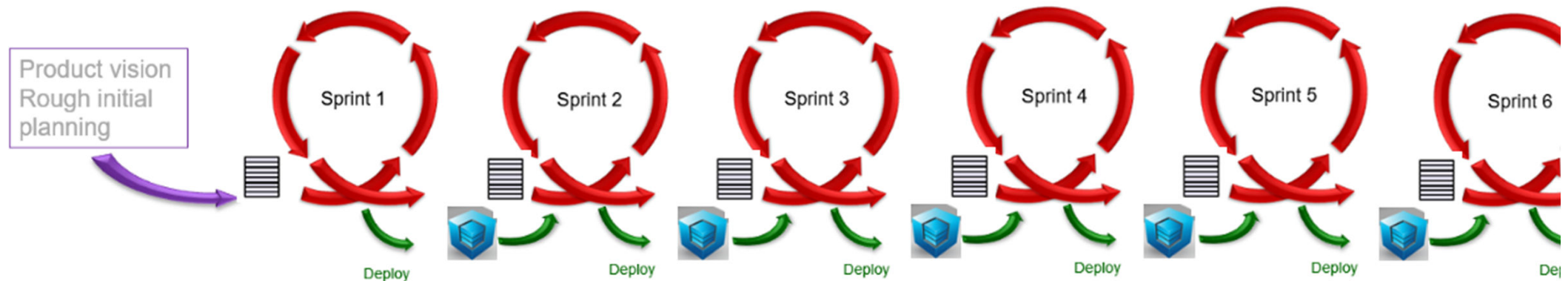


Software Products: Management

So, we can approach development as a **continuous series of agile cycles**, refining and extending the code base.

The concepts of Project and Project Manager lose meaning and relevance.

As we saw:



But, can we really develop effectively without Projects and Project Managers?

Aren't skilled Project managers critical for success?

What did the last Chaos Report indicate?

Resolution by Skill of Project Manager (2020 Report)

Secondary source

Effect of Project Manager on Non-Agile Project Success

Based on 40,000 non-agile projects in the CHAOS database

Skill Level of Project Manager	Successful	Challenged	Failed
Highly Skilled	23%	58%	19%
Skilled	22%	55%	23%
Moderately Skilled	39%	38%	23%
Poorly Skilled or None	58%	33%	9%

“Quality of project manager’s skills has little effect on achieving a positive outcome”

*most unskilled
project manager
created success*

CHAOS Report conclusion for non-agile

“The real key to project manager success is a project manager who can make quick decisions, take risks, and not get bogged down in the project management bureaucracy.”

Resolution by Skill of Project Manager (2020 Report)

Secondary source

Effect of Project Manager on **Agile** Project Success

Based on 10,000 Agile projects in the CHAOS database

But remember: Agile projects tend not to have an active project manager

unskilled project manager seems best here *ALSD*

Skill Level of Project Manager	Successful	Challenged	Failed
Highly Skilled	18%	62%	20%
Skilled	22%	62%	16%
Moderately Skilled	51%	44%	5%
Poorly Skilled or None	91%	8%	1%

If you want to **reduce** your chance of success on an Agile project, use a Project Manager!

CHAOS Report conclusion for Agile

“We believe that the general project management process slows projects down and creates unnatural bureaucracy and long decision intervals.”

Similar conclusion related to use of *traditional* project management tools such as found in EPPM (Enterprise Project Portfolio Management) software packages:

Don't use them.

And a general conclusion:

“Software is infinite, while projects are finite”... “We know we are going to spend money on developing and implementing software. That is a given. What we need to stop doing is artificially breaking software into “projects” with all their overhead and waste of time and money.”