# Introduction to Software Methodology

How do we build software systems?


The **specification** may be in many forms: just an idea in our head, a set of requirements written in English or even a formal specification.

The **implementation** will be a software system.

The ⇢ represents the approach we use to system building, often loosely called a **software methodology**.

# Is Methodology Important?

Ever since we started building more complicated software systems (c. 1960s), there has been an interest in **software methodology**.

The holy grail in this area is an algorithm/process that transforms a specification into an implementation.

**Formal Methods** solve this problem for simple systems that are amenable to formal specification and derivation, but this doesn't address real systems building issues.

Software methodology remains a very fluid area, largely because we aren't (ever) satisfied with its success rate.

# Software Failure

It is a popular complaint (perhaps misconception?) that software projects typically

- run overtime,
- run over budget
- fail to meet their requirements,

...if indeed they manage to produce a working product at all.

Some software projects fail very dramatically, and there are many examples. We'll look at just one briefly.

# AT&T Network Crash

AT&T's long distance network crashed and remained down for 9 hours on January 15, 1990.


A failure in one switch caused a propagation of error to all the other switches.

75,000,000 phone calls failed and 200,000 airline reservations were lost.

The failure was traced back to the C coding error on the next slide.

AT&T's 5ESS Switch contains several million lines of code. (The famous bug was in the previous switch, 4ESS).

# Source of the AT&T Network Crash

```
while (expression){
  switch expression{
    ...
    case (value):{
      if (logical) {
        sequence of statements
        break
      }
      else {
        other statements
      }
      statements after if...else statement
    }
  }
  statements after switch statement
}
statements after while statement
```

What is the effect of this break statement?

Code reported differently in different sources. Programmer meant to break from while loop, but break only exits

# Some figures about failure

The CHAOS report, produced initially in 1994, examines the state of IT projects in the US. Results are as follows:

| | 1994 | 2000 | 2004 | 2009 | 2012 | 2015 |
|---|---|---|---|---|---|---|
| Project completed on time within budget | 16% | 28% | 29% | 32% | 39% | 29% |
| Project late, exceeding budget and with limited functionality ("challenged") | 54% | 49% | 53% | 44% | 43% | 52% |
| Total failure (cancelled etc.) | 31% | 23% | 18% | 24% | 18% | 19% |

- Take this report with a **large grain of salt!**
- Nevertheless, failure is a feature of the software industry

By comparison with other engineering disciplines, software systems seem to fail more often.

Perhaps we're not using the right **methodology**?

# Aside: Software Engineering is difficult

Software engineering is arguably harder than other types of engineering:

Software systems are among the most complicated artifacts created by humankind

"Software Engineering" covers an enormous range of systems. The equivalent "Physical Engineer" does not exist.

Software is especially unforgiving of error.

Back to methodology…

# What is a Methodology?

We can view a methodology as having three components:

A **set of models** to be produced during development.

A **notation** for describing these models.

A **process** to be followed during development

Also, underpinning the set of models we produce is the **metamodel**.

In an object-oriented context, what are the elements of the metamodel?

We now explore the issue of process in more detail.

# Roadmap of this Section

We start by looking at two "traditional" methodologies, both of which have considerable limitations:

**Code and Fix**
**The Waterfall process**

We then consider the key aspects of all more recent methodologies:

**Incremental**
**Iterative**

Finally we examine two examples of different, modern methodologies:

**Unified Process**
**Extreme Programming**

# “Code and Fix”

“Code and Fix” (aka hacking) involves starting coding as soon as you have an idea of what to do. When you encounter a bug/new requirement, you just fix it or code it up.

This isn’t really a process at all, just the idea of writing code that should be part of the final program and working towards the final goal in that way.

A bit like e.g. trying to get to Galway by walking due West.

# Can “code and fix” work?

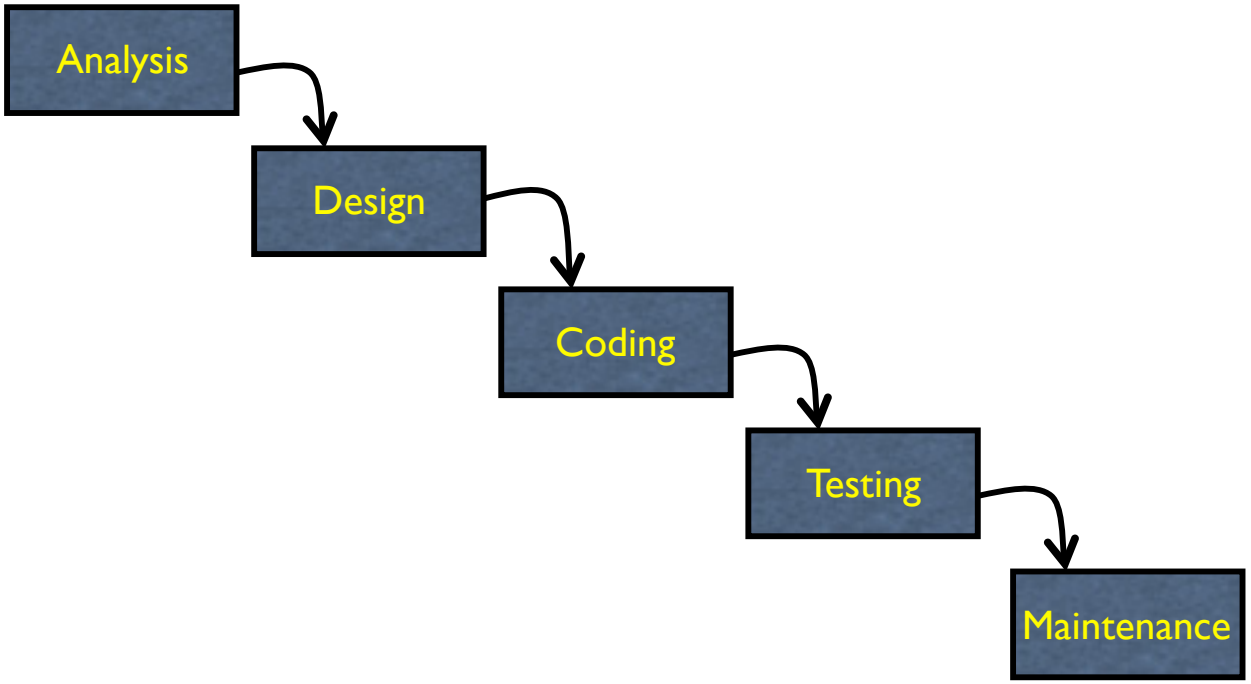
This approach can work for a resourceful individual, but is impossible to apply when a large team is involved.

Even when a working product is achieved, **maintenance** is frequently impossible.

However, successful systems have been developed by individual programmers working on their own, or by very small teams.

# The Waterfall Model

Initial attempts (~1960s) to define a real process for developing software were based on an approach used in other areas.




What other process does this bring to mind?

What are the key properties of the Waterfall process?

# Properties of the Waterfall Model

Development is broken into a number of **distinct phases**.

The phases are completed **in order**: analysis, then design, then implementation.

The output of each phase serves as input to the next phase (this is the "**waterfall**").

Mimics a typical, sequential construction process.

Sounds great, but…

> Although very appealing in a certain way, and much loved by managers, the Waterfall model has severe shortcomings.

One of the first descriptions of the Waterfall Process [Royce, 1970] already identified many of its shortcomings.

# Weaknesses of the Waterfall Model

The essential problem with the Waterfall model is that it takes a naïve approach to **risk management**.

There are two types of threat involved:

1. The designers/developers make a mistake.
2. The requirements change during development.

See: "The Demise of the Waterfall Model Is Imminent" and Other Urban Myths by Phillip A. Laplante and Colin J. Neill, ACM Queue, February 2004
Volume 1, issue 10.

(1) cannot be avoided of course.

Neither can (2) in reality.

Any methodology that ignores these threats is doomed to failure.

Another issue is **late feedback**. Customers see the product only very late in the process.

# Error Detection

The best technique we have for discovering errors in software is...

**testing**

Testing an **analysis** document is possible:

run through various scenarios with the customer

Testing a **design** is possible:

- "desk checks"
- role play with CRC cards
- perhaps automated support...

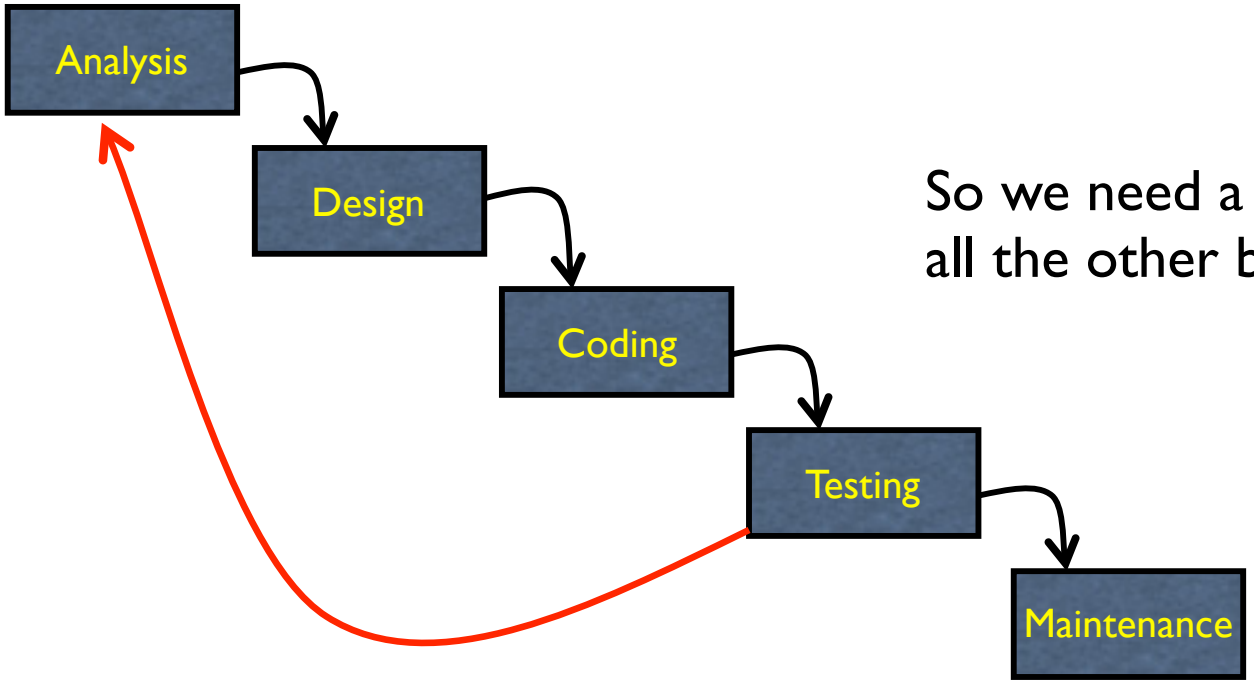
But testing is never more conclusive than when testing **executable code**.

Also, performance problems can usually only be reliably detected on the executable code.

# Designer/Developer Error

In the Waterfall Model, when will an error be discovered?
Say the initial analysis is wrong, when will this be found?

It **may** be uncovered during design, but it is only likely to be found during testing (and even that is not guaranteed).




So we need a backarrow, and in fact all the other backarrows as well.

Uncovering analysis/design errors months or years after they were made is very expensive! Problem is: **we're testing too late**

# Handling New Requirements

In many engineering environments, it not acceptable for a customer to introduce new requirements after the project has been started.

However, most software systems have to fit into a dynamic environment where unstable requirements are an unavoidable reality.

Fixed requirements are a myth.

In a pure waterfall process, a new requirement is handled in a similar way to how an analysis error is handled:

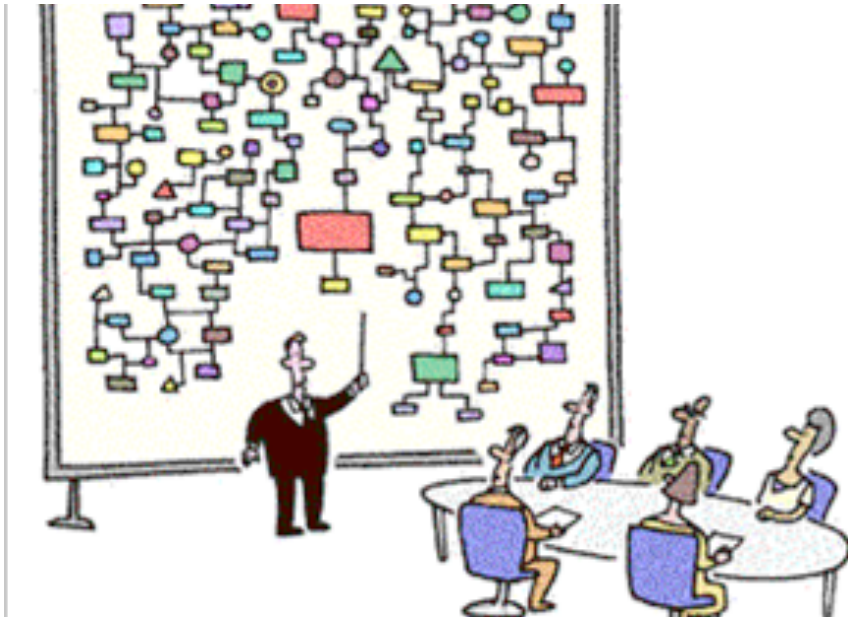
1. analysis is revisited, then
2. design is revisited, then
3. implementation is revisited

**Very costly!**

# Analysis Paralysis

The problems just outlined often leads to waterfall projects suffering from **analysis paralysis**.

The Analysis has to be perfect before it can be signed off. Perfection is impossible, so completion is impossible.



With the project deadline a few months way, the analysis is typically ditched and the developers just start coding.

The results in a process that is in reality not much of an improvement on Code and Fix.

# Improving on Waterfall

Current process models invariably aim to manage risk better.

Two of the key features of any current methodology is that they are:

**Incremental**
**Iterative**

We'll look at these properties separately before considering concrete methodologies that are based on them.

(Although they are distinct properties, they go very much hand-in-hand)

# Incremental Development

The key property of an **incremental**, or **evolutionary** process is that a simple working prototype is initially produced and this is gradually extended (evolved) to the final system

Advantages of incremental development:

- Early testing and feedback from customers
- Easier integration of new requirements

Disadvantages of incremental development:

- Hard to manage -- will the increments ever result in the final system?
- How is the overall system design developed?

# Aside: this is nothing new...

In 1981/82, Jonathan Sachs and Mitch Kapor developed the the first "killer application" spreadsheet, Lotus 1-2-3.

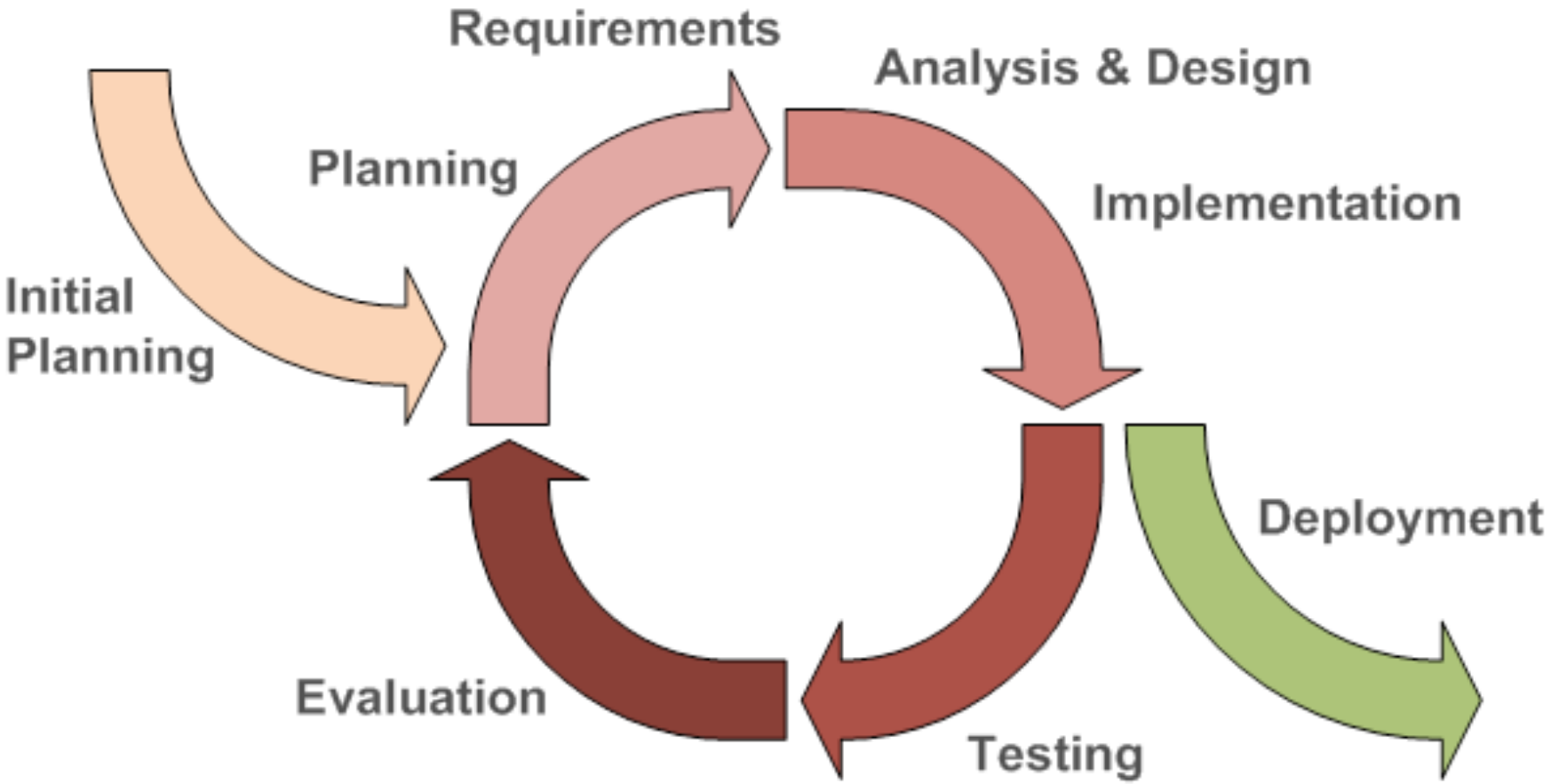
Kapor and Sachs added feature after feature, finally creating the then best-of-breed spreadsheet application.

Sachs:

> *"It wasn't like we sat down and designed everything and then built it. There was no point where we sat in a room and drew this thing on a whiteboard that said: spreadsheet, graphing, database, macro language"*

# Iterative Development

The key property of an **iterative** model is that several passes through the various stages of the methodology are made, e.g.



# Iterative vs. Incremental Development

'Iterative' and 'incremental' are distinct properties. It is possible, though not common, to have one without the other.

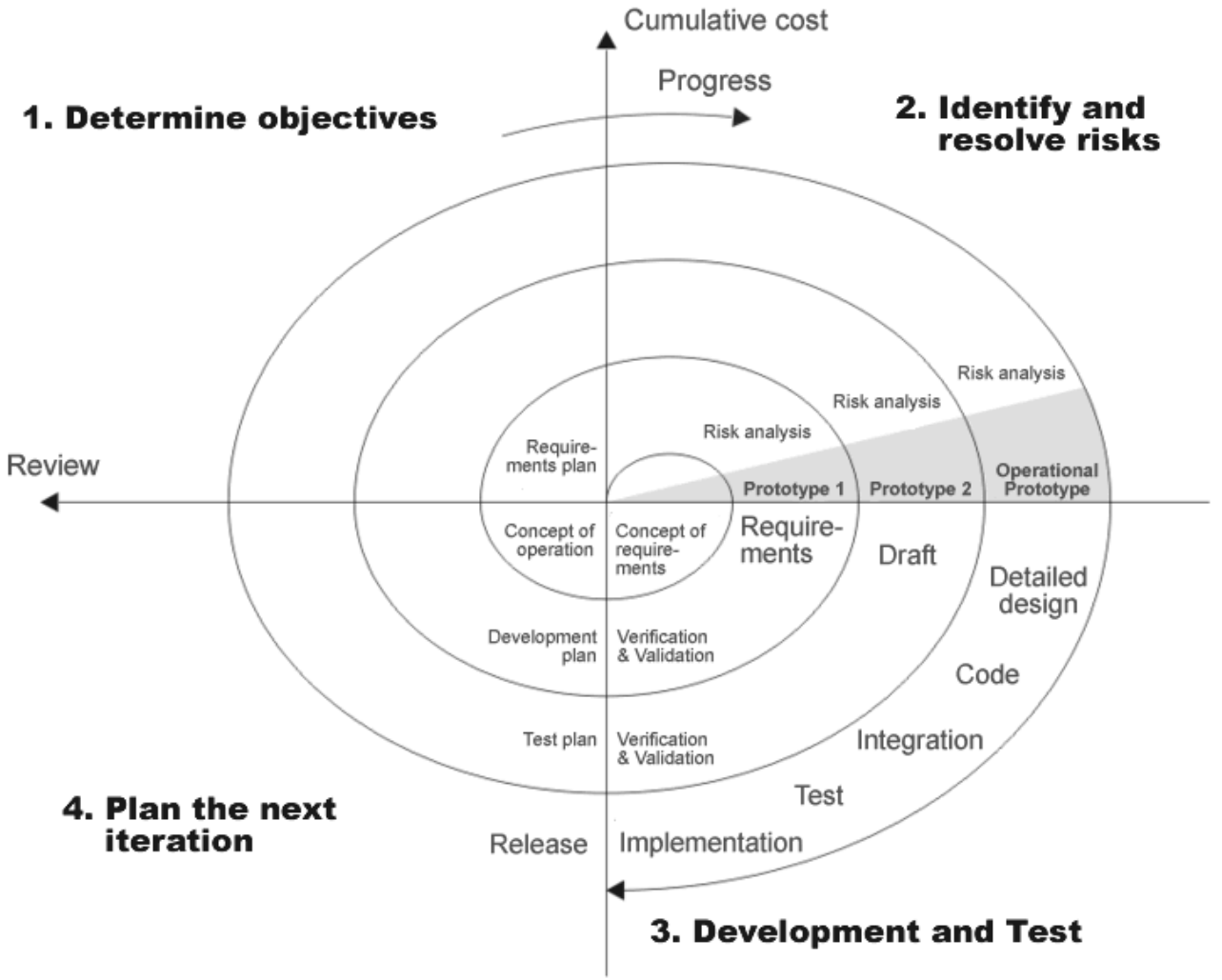

# Iterative Development

Combining the waterfall metaphor with iteration could lead to a whirlpool…

However, no-one wants to be stuck in a whirlpool, so the best known early process model that involves iteration is called the **spiral model** (Boehm, 1988).

Diagram on next slide.

# Boehm's Spiral Model



# Boehm's Spiral Model

Advantages of the Spiral Model:

- Risk assessment is part of the process
- Flexible – acknowledges that the same process does not suit all projects

…

Disadvantages of the Spiral Model:

- Complex
- Hard to manage
- Overkill for small projects

# The Unified Process

The **Unified Process** is a state-of-the-art, tailorable **framework** that is both iterative and incremental, and borrows a lot from the spiral model.

Initially developed by designers of UML

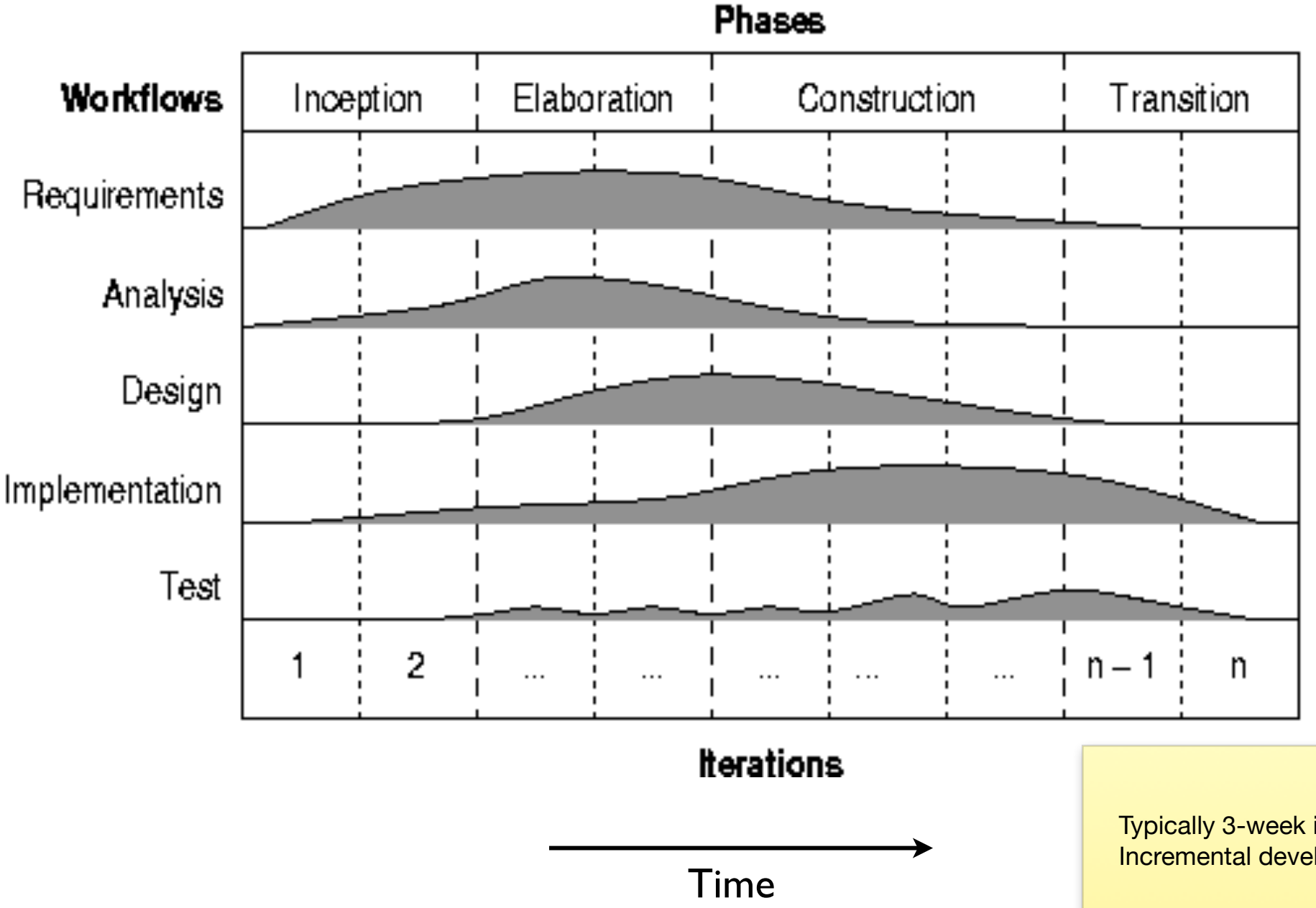
Structures project as a number of *phases*

Each phase contains several *iterations*

Different *workflows* (activities) are performed in each iteration (notice the traditional set of activities)

Diagram on next slide

# Unified Process Outline




Typically 3-week iterations.
Incremental development.

# 4 Phases of the Unified Process

1. **Inception**
2. **Elaboration**
3. **Construction**
4. **Transition**

# Phase 1: Inception

Make business case for the project. Create preliminary project plan and costings.

Establish project scope, decide what will be part of the project and what will not.

Create basic **Use Case Model** (model of **what** the system does, see UML section of this module).

Propose possible system architecture; several architectures may be proposed.

Identify possible risks to the project.

Should be short phase, may involve just one iteration.

A long inception phase suggests that either the project has problems or that too much analysis/design is taking place.

# Phase 2: Elaboration

Perform **Domain Analysis**. Involves identifying the key classes in the domain (see UML section of module).

Design and implement overall System Architecture, leading to an **Executable Architecture Baseline**). This is a partial implementation of the entire system with its complete overall design and architecture.

Plan construction phase (time and costs).

# Phase 3: Construction

The bulk of the actual coding takes place in this phase. Low-level modelling may also take place (e.g., UML Sequence or Collaboration models).

Multiple, timeboxed iterations are performed, each resulting in a new release of the executable system.

Development is driven by the Use Cases, and these can be used to define test cases as well.

The phase finishes when the complete system has been implemented.

# Phase 4: Transition

System is deployed to its users (this can be simple or complicated).

Beta testing happens here, along with subsequent system refinements.

Training of maintainers and end users.

# Unified Process Pros and Cons

**Pros:**

* Includes all aspects of software development
* Flexible and tailorable
* Mature

**Cons:**

* Suits large projects best
* Possible overkill for smaller projects

# UML and the Unified Process are closely related

UML is a modelling language.

It defines diagrams that represent aspects of software systems

UML diagrams can be used in conjunction with many different processes (or even in the absence of a formal process)

Because of their history, there is a close fit between UML and the UP.

Both were developed by three well-known figures in software methodology: **Booch**, **Rumbaugh** and **Jacobson**.

# The Unified Process is really a Framework

The UP is not a prescriptive methodology – it's really a tailorable framework that you can specialise to fit your specific project.

Some specialisations are well known in themselves, especially the **Rational Unified Process** (IBM trademark).

Other specialisations include:

- **Enterprise UP**: an extension of RUP for enterprise systems
- **Agile UP**: a lightweight, Agile version of UP
- **Open UP**: part of the Eclipse Process Framework, suitable for open source development

# Agile Processes

See separate set of slides for this.

# Snapshot of Irish Software Industry

| Company# | Product Area/Market | Lifecycle | Product Maturity (>) | Co. Size (>) |
|---|---|---|---|---|
| 1 | hardware/security | outsourcing | 10 | 30 |
| 2 | healthcare | unstructured | 2 | 20 |
| 3 | web services | iterative | 8 | 60 |
| 4 | financial services | dsdm/iterative | 4 | 200 |
| 5 | public services | waterfall | 2 | 600 |
| 6 | telecoms | waterfall | 10 | 10000 |
| 7 | telecoms | waterfall | 1 | 300 |
| 8 | financial services | agile | 2 | 700 |
| 9 | ERP | JIT | 5 | 25 |
| 10 | telecoms | RUP | 9 | 10000 |
| 11 | financial services | waterfall | 10 | 25 |
| 12 | entertainment | XP | 1 | 20 |
| 13 | financial services | waterfall | 3 | 80 |
| 14 | travel services | iterative | 10 | 700 |
| 15 | financial services | waterfall | 15 | 300 |
| 16 | public services | waterfall | 10 | 1000 |
| 17 | financial services | spiral | 10 | 200 |
| 18 | software services | scrum | 1 | 10000 |
| 19 | software services | RUP/waterfall | 10 | 10000 |
| 20 | financial services | scrum | 15 | 30 |
| 21 | logistics | iterative | 8 | 50 |
| 23 | web services | agile | 5 | 10000 |
| 22 | public services | waterfall | 10 | 1000 |
| 24 | financial services | waterfall | 5 | 50 |
| 25 | entertainment | scrum | 5 | 500 |
| 26 | CRM | waterfall | 3 | 10000 |
| 27 | telecoms | iterative | 10 | 20 |
| 28 | financial services | unstructured | 1 | 1000 |
| 29 | CRM | waterfall | 4 | 500 |

See web resources for more detail

# Summary

The process we use for developing software is a critical issue, and there is no definitive best approach.

There has been an evolution both in terms of the *metamodel* used and *methodology* used.

The object-oriented model is dominant in the software industry.

Current methodologies are invariably **iterative** and **incremental**, ranging from the sophisticated Unified Process to lightweight processes like Extreme Programming.