COMP2511

🎨 5.1 - The Unknown

In this lecture

- The unknown; types of unknowns
- The role of the unknown in Software Engineering
  - The role of the unknown in **developing new software**
  - The role of the unknown in **working with existing software**
- Software Longevity
- Collaborative Engineering

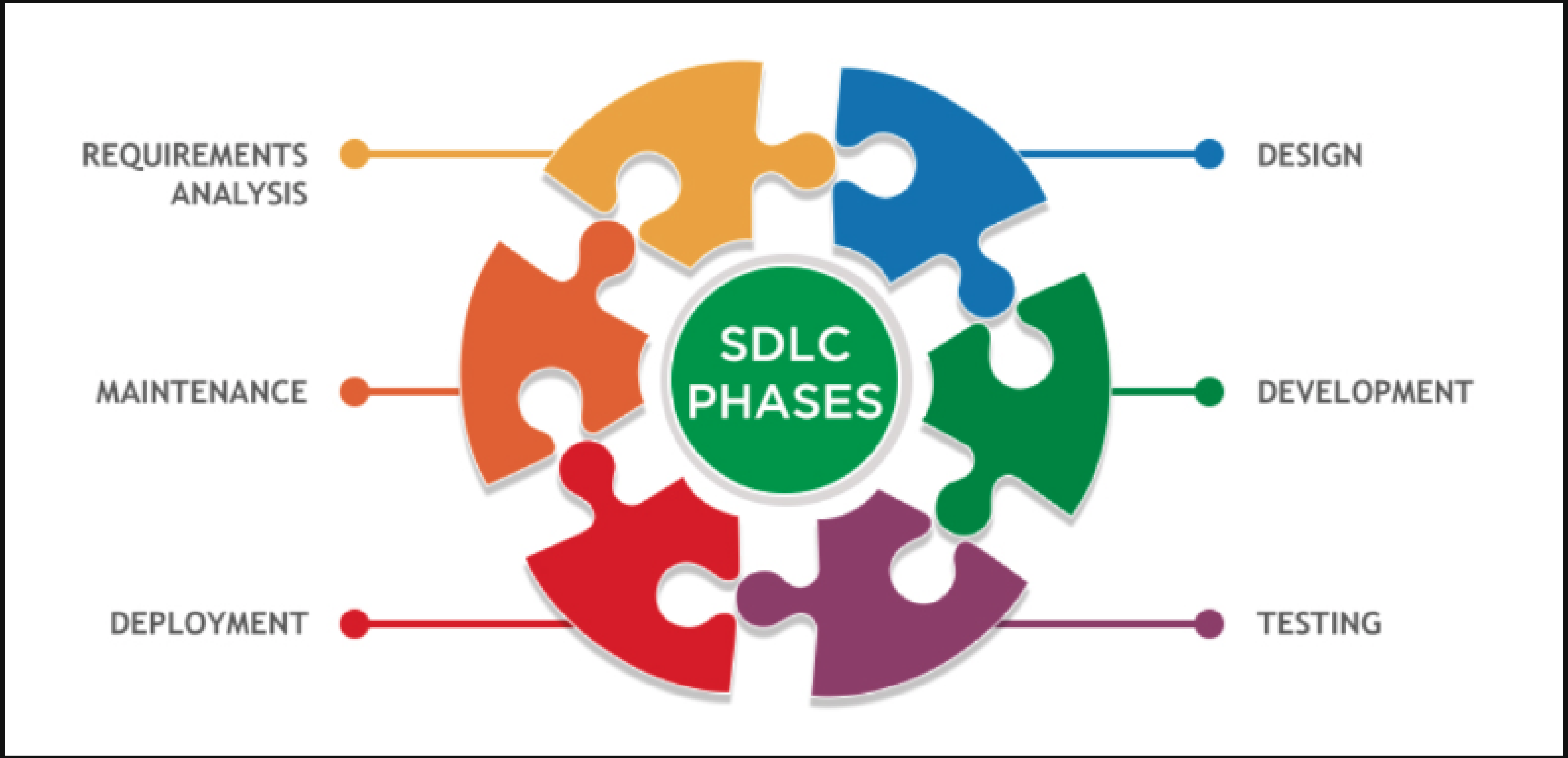
A brief step back - what makes good software?

The Unknown

- Two types of hard problems:
  - Easy to understand, hard to solve
  - Hard to understand, easy to solve
- Two types of unknowns
  - **Known unknowns** - we know that it exists, but we don't know what it is
  - **Unknown unknowns** - that which we had never even thought to consider
- Learn to deal with unknowns gracefully as they arise

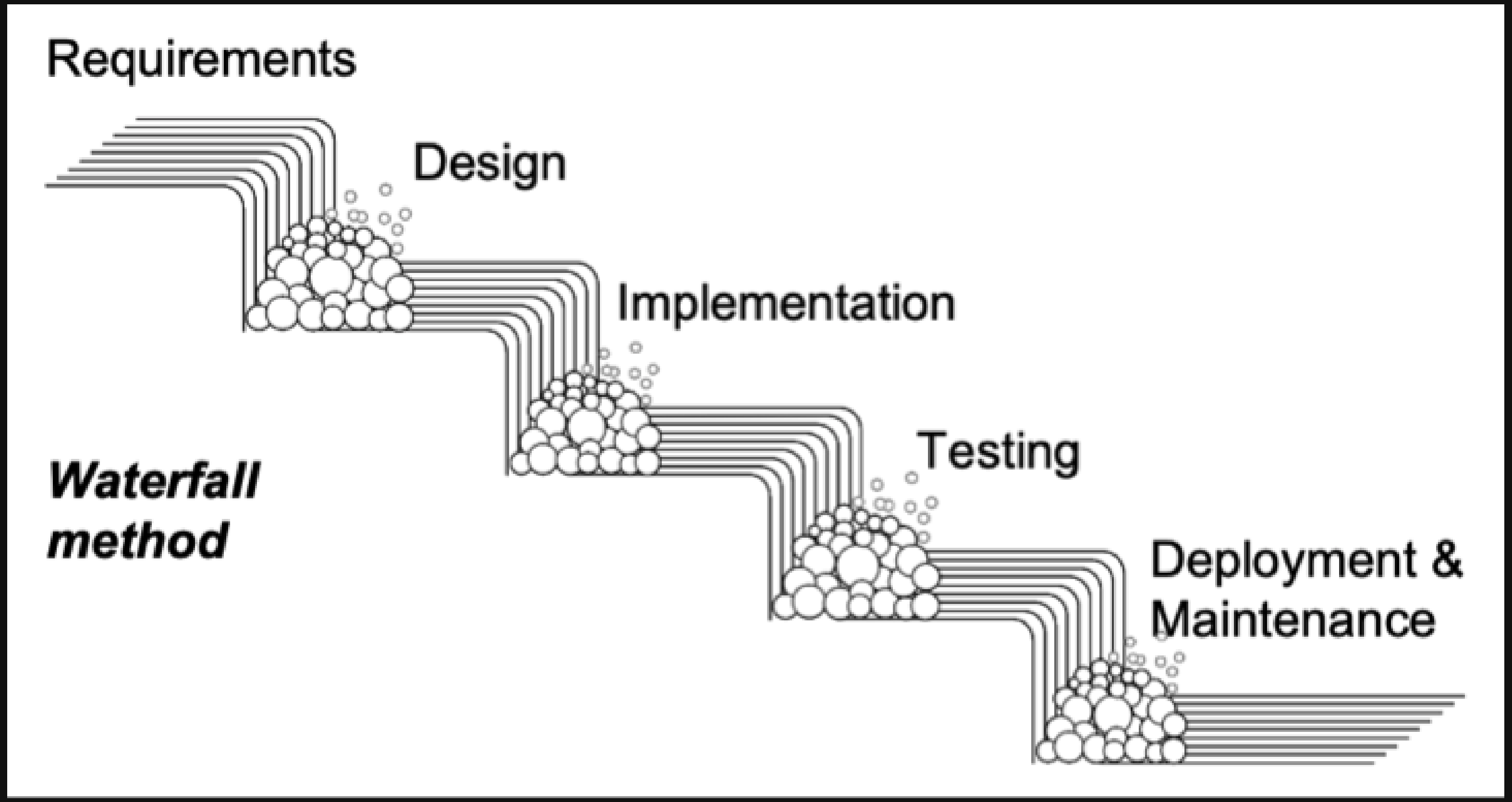
I: The Role of The Unknown in Developing New Software

The Software Development Lifecycle



Traditional Engineering: The Big Design Up Front

- One step at a time
- Ensure the current step is perfect before moving onto the next one
- A big design up front
- Project can take months-years to complete



Difficulties Presented by the Unknown

- The game changes
  - Changing market
  - Changing client expectations
  - Changing technical world
- Evolution of Requirements
- Too many unknown unknowns to be able to **design everything up-front**
- There comes a need to **learn through experience** and **iterate on design**

Iterative Design

- Work in sprints, iterations, milestones
- 'Agile' software development
- Many variants - eXtreme programming, Rapid Application Development, Kanban, Scrum
- Design incrementally
  - Adapt to changes in requirements
  - Discover and deal with problems in design as they arise

Problems with Purely Iterative Design

- No clear sense of direction/trajectory
- In poorly designed systems, adaptations to new requirements become smaller-scale 'workarounds' - limit functionality/decrease maintainability
- Tendency to 'make it up as we go along'

A solution?

- **High Level Design**
  - Design a broad overview up-front
  - A framework to begin development
  - Set the trajectory and boundaries of work at the start
- Adapt and change the design during development as needed
- Design up-front a solution that is open for extension, reusable, etc.
- Complete work in **small increments** and **improve iteratively**
- A brief history of COMP2511

Requirements Engineering

- As systems grow in size and complexity, so do their requirements
- Requirements become harder to:
  - **Communicate**, on the side of the client;
  - **Understand**, on the side of the engineer
- Domains are often highly specific, nuanced and complex
- Requirements *Analysis* might be better termed *Engineering*
- The problem space grows around us with our understanding
- Transformation of the unknown into the known
- Often, this requires us to dive into the deep end and start **actualising solutions** in order to better understand what the problem is
- We have to make **assumptions** to limit the contract to which we program

II: The Role of The Unknown in Working with Existing Software

Now, and Later

- What about later?
- 5 weeks, 5 months, 5 years… 15 years?
- What happens when code grows and systems get bigger?

Complexity

- Code becomes more complex as it grows in size
- Cyclomatic complexity
- More complexity leads to:
  - More scope for errors to creep in;
  - Higher **risk** of software breaking;
  - More unknown unknowns
  - Too many possible combinations to predict up-front

Software Becomes a Beast

- When software reaches a certain level of complexity, it becomes **alive**, and **constantly moving**
- We can no longer informally reason about it with certainty; behaviour can become unpredictable
- Changes in one class/package have far-reaching unpredictable effects due to **coupling**
- Testing becomes essential here
- **Monolith** Repositories
    - Large applications, with all the code inside a single repository
    - Slower build and Continuous Integration times
    - More *inertia* - harder to ship new features
    - More incidents, bugs and defects due to mathematical complexity
    - Industry is moving towards **microservice architecture**

Technical Debt & Trade-offs

- In large-scale software systems, the design decisions you make today will have consequences for years to come.
- Every design decision comes with a cost (termed "technical debt")
- This cost must later be paid back in the form of maintenance, refactoring, or rewriting
- Design decisions are often a matter of picking the deal that is least-worst in the long run - no easy way out
- Opportunity cost - pick one or the other
- All software architecture and design consists of making trade-offs

How to deal with the unknown here?

- **Code Hygiene** - take care of your code, and your code will take care of you
- Rigorous testing is essential
- A Symbiotic Relationship between Product and Engineering
  - Sacrifice time spent shipping new features **now** to maintain software hygiene, meaning we can still ship new features **later**
  - Slower today, faster forever

Software Longevity

- Difficult challenges:
  - Dealing with code that you wrote weeks, months or years ago
  - Dealing with code that *other people* wrote weeks, months and years ago
- In large-scale software systems, many software components are treated as black-boxes since the original author is no longer present and no one understands how it works!
- Can often be a tendency to **completely migrate (move) systems** rather than **maintaining or decomposing existing legacy infrastructure**
- Software has a very fast turnover compared to other forms of engineering

Longevity and the Open-Closed Principle

- There can be a tendency to **build around existing software** rather than going in and making changes
- In some philosophical respects, this is the open-closed principle
- The band-aid problem

Parting Thoughts on Longevity

- Mark of well-written code:
  - It remains intact for a long time
  - *And* new engineers are able to onboard and understand how it functions
- Bugs lying dormant - Human tendency to ignore that which we do not immediately understand or must analyse - leads to problems in legacy codebases remaining unchecked before surfacing

Collaborative Engineering

- Pair assignment - dealing with the unknown together
- Bounce ideas off one another
- Review each others' work - design reviews, test reviews, code reviews
- Pair programming
  - One person codes, one person watches