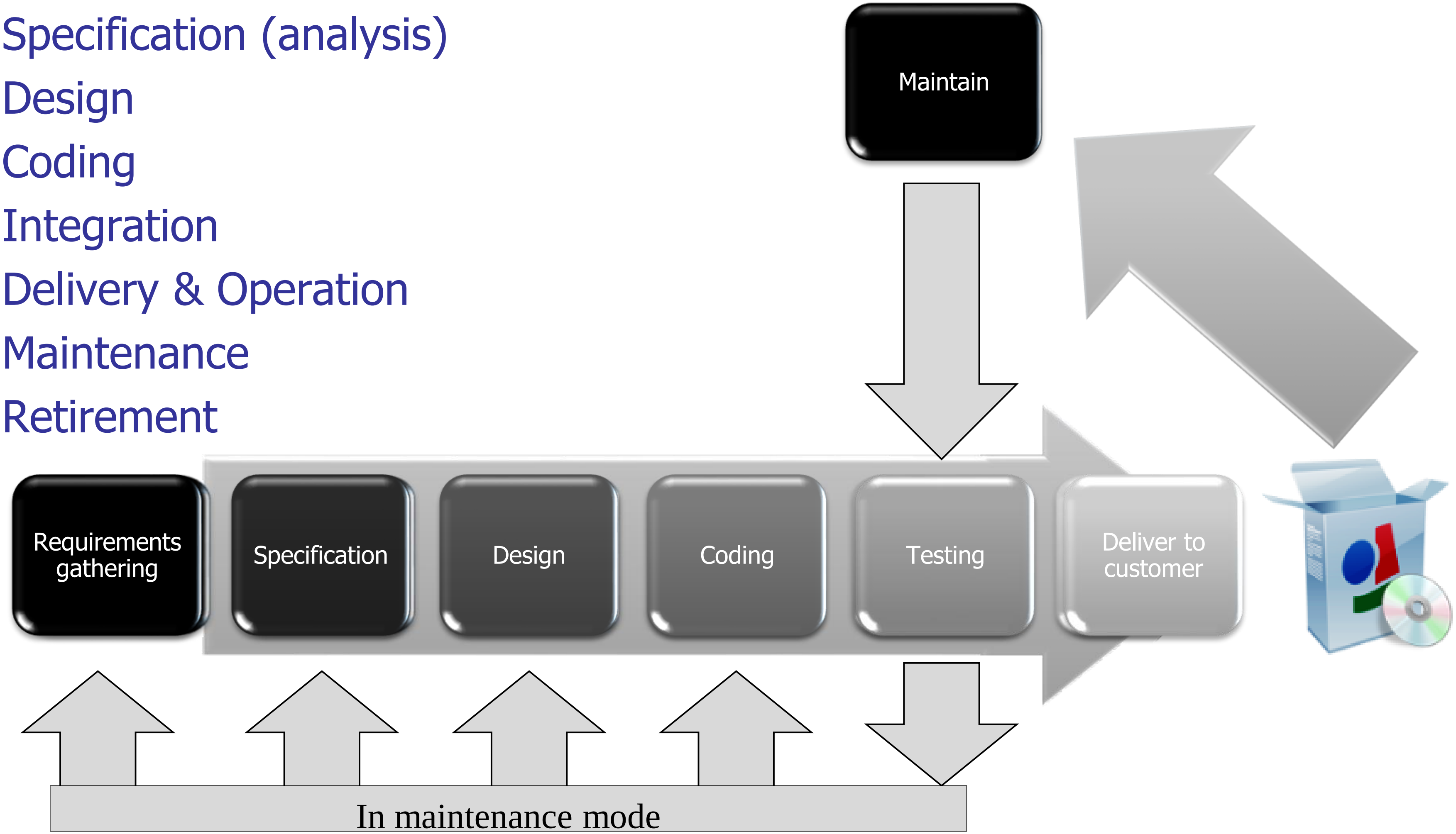
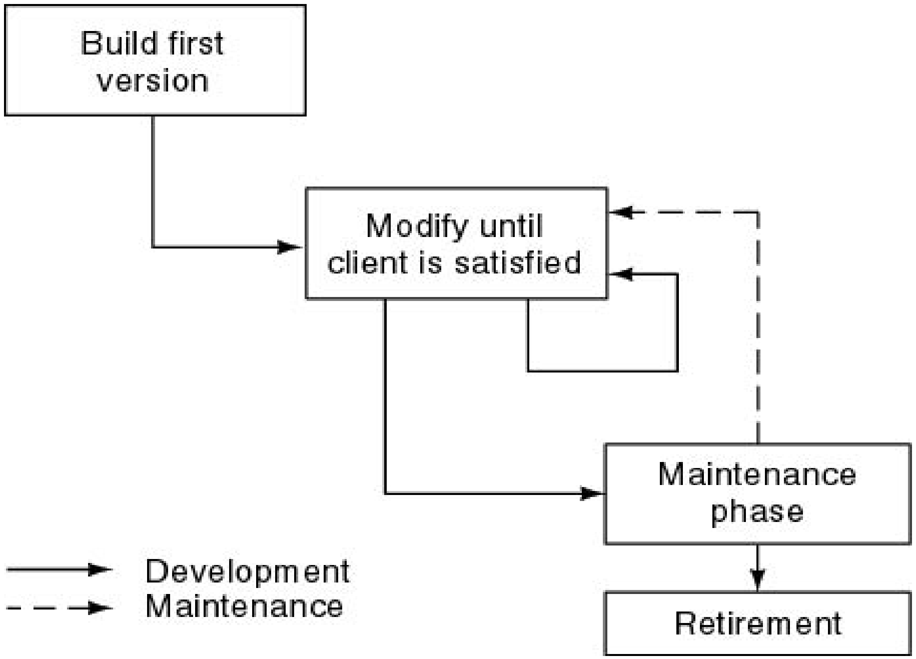
Primary Phases of Software Life-cycle

- Requirements gathering
- Specification (analysis)
- Design
- Coding
- Integration
- Delivery & Operation
- Maintenance
- Retirement



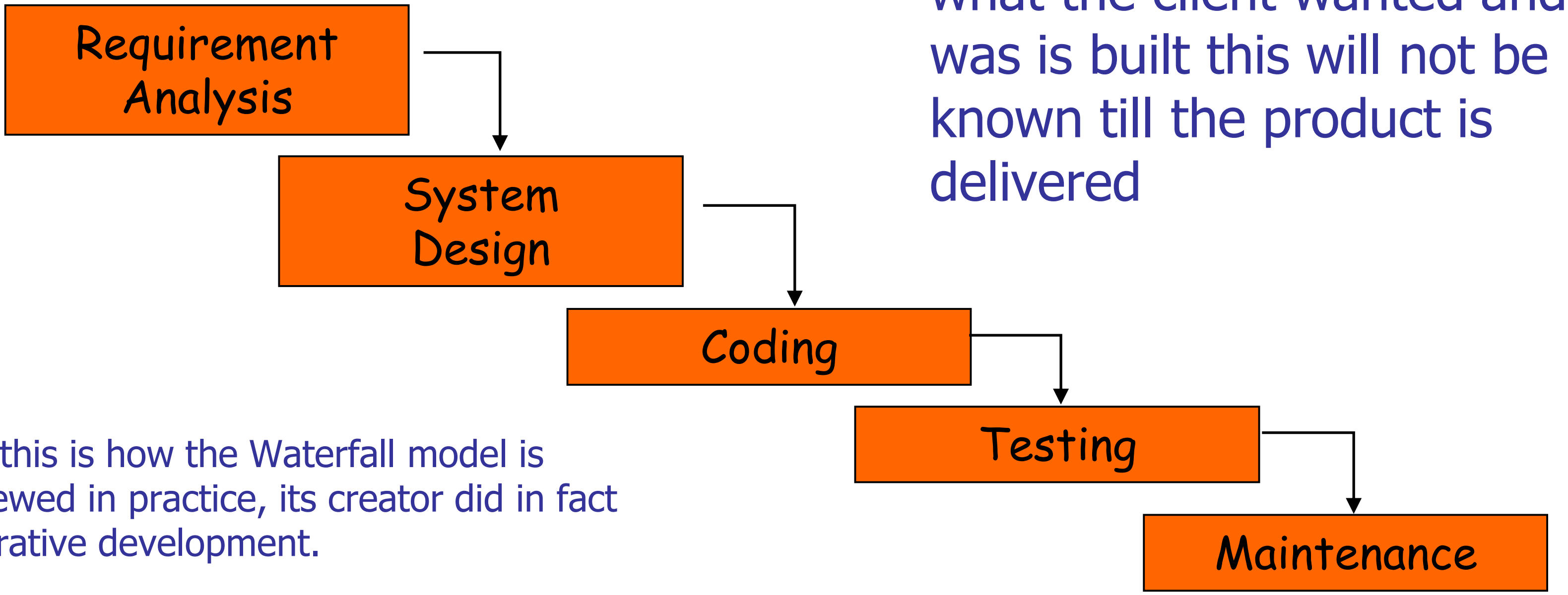
Build and Fix Model



- Lots of software is developed using build-and-fix model
- Basically there is no model.
  - No specifications
  - No design
- This model is completely unsatisfactory and should not be adopted.
- Except possibly for learning purposes?
- **Why?**
- Need life-cycle model
  - "Game plan"
  - Phases
  - Milestones

Waterfall Model

- Output from one phase is fed as input to the next phase.
- One phase is completed, documented and signed-off before the next phase begins.



- Advantages
  - Each phase is well documented.
  - Maintenance easier.
- Disadvantages
  - If there is a mismatch between what the client wanted and was is built this will not be known till the product is delivered

- Note: while this is how the Waterfall model is generally viewed in practice, its creator did in fact allow for iterative development.

Waterfall Strengths

- Easy to understand, easy to use
- Provides structure to inexperienced staff
- Milestones are well understood
- Sets requirements stability
- Good for management control (plan, staff, track)
- Works well when quality is more important than cost or schedule

Waterfall Deficiencies

- All **requirements must be known upfront**
- Deliverables created for each phase are considered frozen – inhibits flexibility
- Can give a false impression of progress
- Does not reflect problem-solving nature of software development – iterations of phases
- Integration is one **big bang** at the end
- Little opportunity for customer to preview the system (until it may be too late)

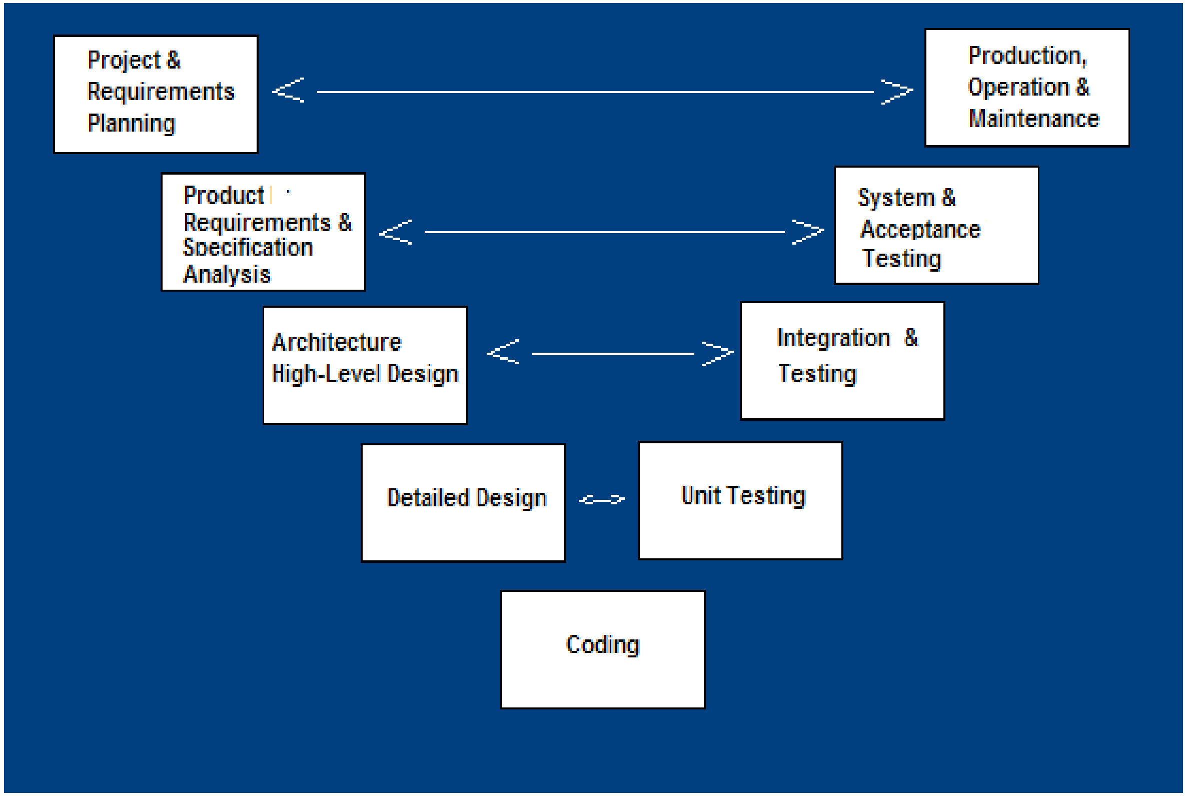
Waterfall Misunderstood in Practice?

- Perhaps.
- Its origins – in essence it is from a different time.
- Hardware costs.
- Critically: it does theoretically advocate iterative development (refer to Royce, 1970).

When to use the Waterfall Model

- Requirements are very well known
- Product definition is stable
- Technology is understood
- New version of an existing product
- Porting an existing product to a new platform.

V-Shaped SDLC Model


Project & Requirements Planning
Production, Operation & Maintenance
Product Requirements & Specification Analysis
System & Acceptance Testing
Architecture High-Level Design
Integration & Testing
Detailed Design
Unit Testing
Coding

- A variant of the Waterfall that emphasizes the verification and validation of the product

- Testing of the product is planned in parallel with a corresponding phase of development

V-Model

- **Verification** – building the product the right way; have we followed our own process in building the product?
- **Validation** – building the right product; is the product valid?

V-Shaped Strengths

- Emphasize planning for verification and validation of the product in early stages of product development
- Each deliverable must be testable
- Project management can track progress by milestones
- Easy to use

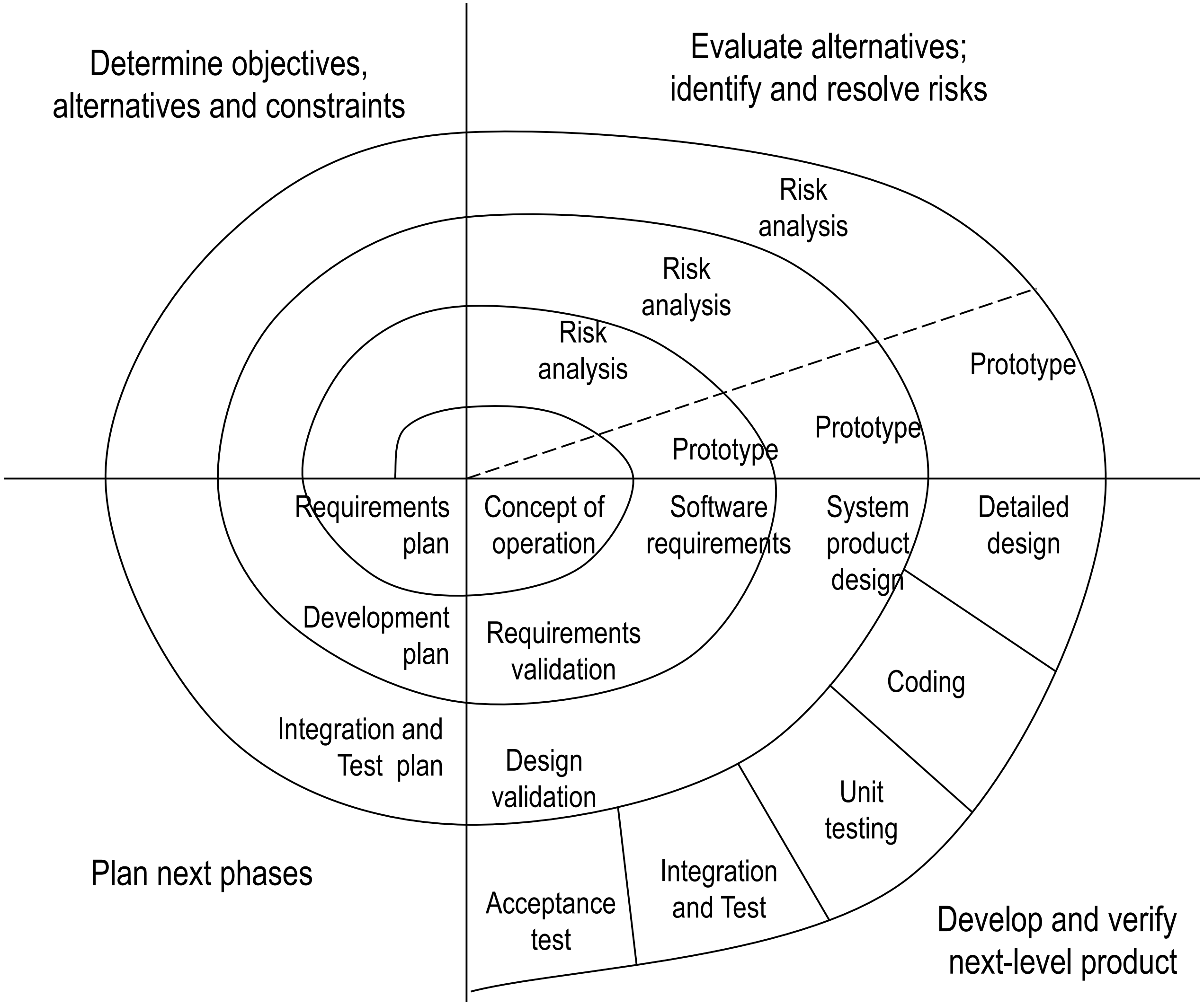
V-Shaped Weaknesses

- Does not easily handle concurrent events
- Does not handle iterations or phases
- Does not easily handle dynamic changes in requirements
- Does not contain risk analysis activities

When to use the V-Shaped Model

- Excellent choice for systems requiring high reliability
- All requirements are known up-front
- When it can be modified to handle changing requirements beyond analysis phase
- Solution and technology are known

Spiral Model



Spiral Quadrants

- **Determine objectives, alternatives and constraints**
  - Objectives: functionality, performance, hardware/software interface, critical success factors, etc.
  - Alternatives: build, reuse, buy, sub-contract, etc.
  - Constraints: cost, schedule, interface, etc.
- **Evaluate alternatives, identify and resolve risks**
  - Study alternatives relative to objectives and constraints
  - Identify risks (lack of experience, new technology, tight schedules, poor process, etc.
  - Resolve risks (evaluate if money could be lost by continuing system development
- **Develop next-level product**
  - Typical activites:
    - Create a design
    - Review design
    - Develop code
    - Inspect code
    - Test product
- **Plan next phase**
  - Typical activities
    - Develop project plan
    - Develop configuration management plan
    - Develop a test plan
    - Develop an installation plan

Spiral Model Strengths

- Provides early indication of insurmountable risks, without much cost
- Users see the system early because of rapid prototyping tools
- Critical high-risk functions are developed first
- The design does not have to be perfect
- Users can be closely tied to all lifecycle steps
- Early and frequent feedback from users
- Cumulative costs assessed frequently

Spiral Model Weaknesses

- Time spent for evaluating risks too large for small or low-risk projects
- Time spent planning, resetting objectives, doing risk analysis and prototyping may be excessive
- The model is complex
- Risk assessment expertise is required
- Spiral may continue indefinitely
- Developers must be reassigned during non-development phase activities
- May be hard to define objective, verifiable milestones that indicate readiness to proceed through the next iteration
- Degree of desirable customer engagement may be difficult to realise

When to use Spiral Model

- When creation of a prototype is appropriate
- When costs and risk evaluation is important
- For medium to high-risk projects
- Long-term project commitment unwise because of potential changes to economic priorities
- Users are unsure of their needs
- Requirements are complex
- New product line
- Significant changes are expected (research and exploration)

Coping with change

- Change is effectively inevitable in software projects.
  - Business changes lead to new and changed system requirements
  - New technologies open up new possibilities for improving implementations
  - Changing platforms require application changes
- Change leads to rework so the costs of change include both rework (e.g. re-analysing requirements) as well as the costs of implementing new functionality

Reducing the costs of rework

- **Change avoidance**, where the software process includes activities that can anticipate possible changes before significant rework is required.
  - For example, a prototype system may be developed to show some key features of the system to customers.
- **Change tolerance**, where the process is designed so that changes can be accommodated at relatively low cost.
  - This normally involves some form of incremental development.
  - Proposed changes may be implemented in increments that have not yet been developed.
  - If this is impossible, then only a single increment (a small part of the system) may have be altered to incorporate the change.

Strategy for Change Management

- Dependent on individual situational contexts
- What is the nature of the changing requirements?
- Frequent, large changes in requirements or small, irregular changes to requirements?
- Depending on this characteristic of the environment, an appropriate change strategy should be adopted.
- Change avoidance may be necessary in certain settings (e.g. where large numbers of suppliers are contributing to a safety critical system), whereas a high change tolerance may be suited to other situations (e.g. where rapid, standalone innovation or R&D is taking place).

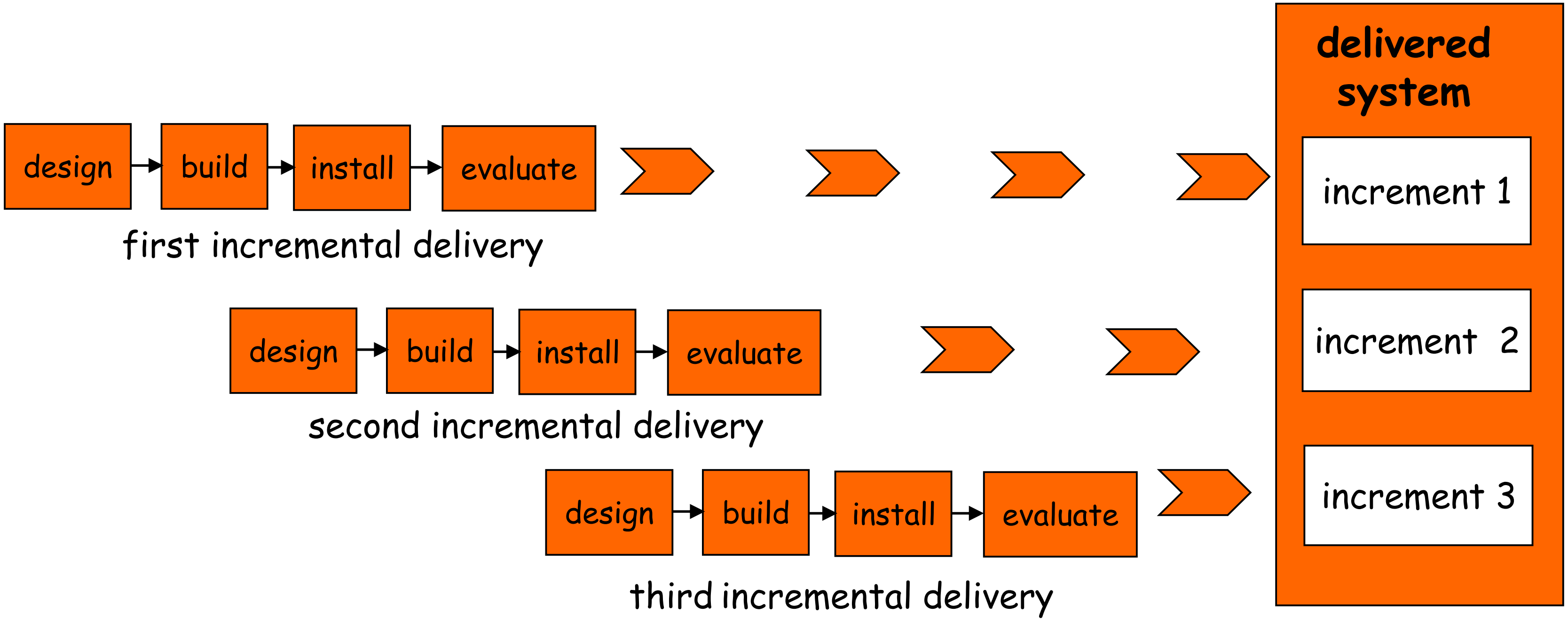
Iterative and Incremental

- **Iterative**
  - repeated execution of the waterfall phases, in whole or in part, resulting in a refinement of the requirements, design and implementation
- **Incremental**
  - operational code produced at the end of an iteration
  - supports a subset of the final product functionality and features

- Artifacts evolve during each phase
- Artifacts considered complete only when software is released
- Reduce cycle time
- Two parallel systems:
  - operational system (Release n)
  - development system (Release n+1)

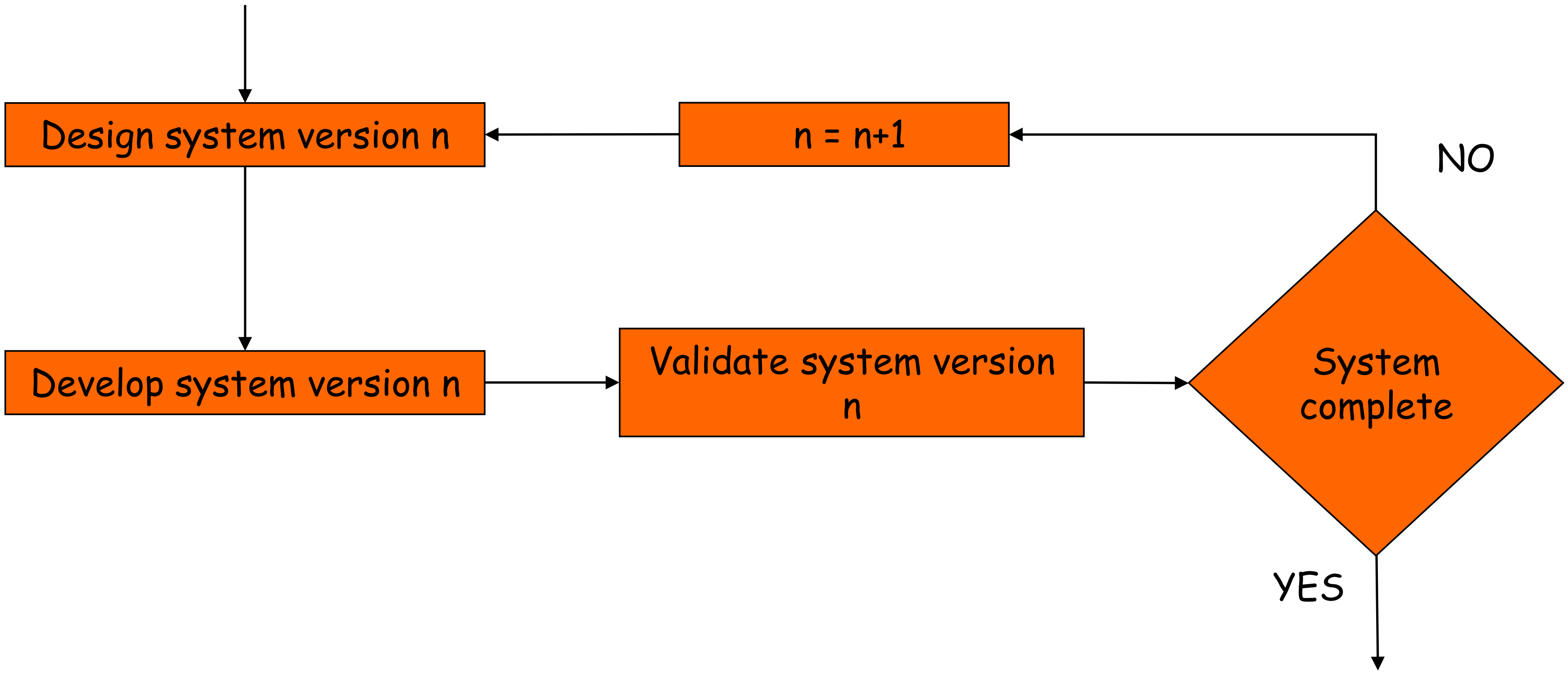
Incremental Model

- Break system into small components
- Construct a partial implementation of a total system
- Then slowly add increased functionality
- The incremental model prioritizes requirements of the system and then implements them in groups.
- Each subsequent release of the system adds function to the previous release, until all designed functionality has been implemented.

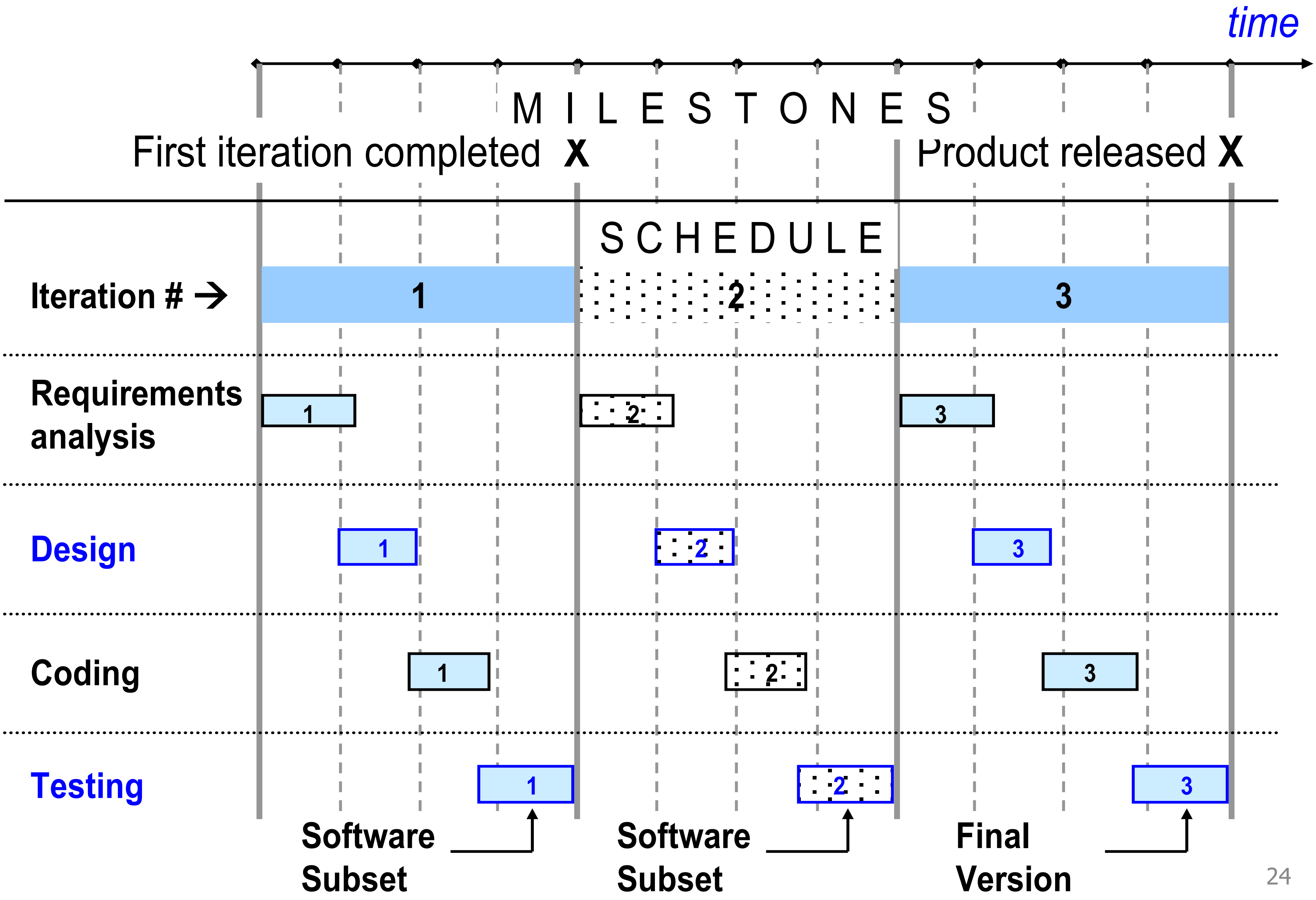


Iterative Model

- Deliver full system shell in the beginning
- Enhance functionality in new releases



Iterative and Incremental Development

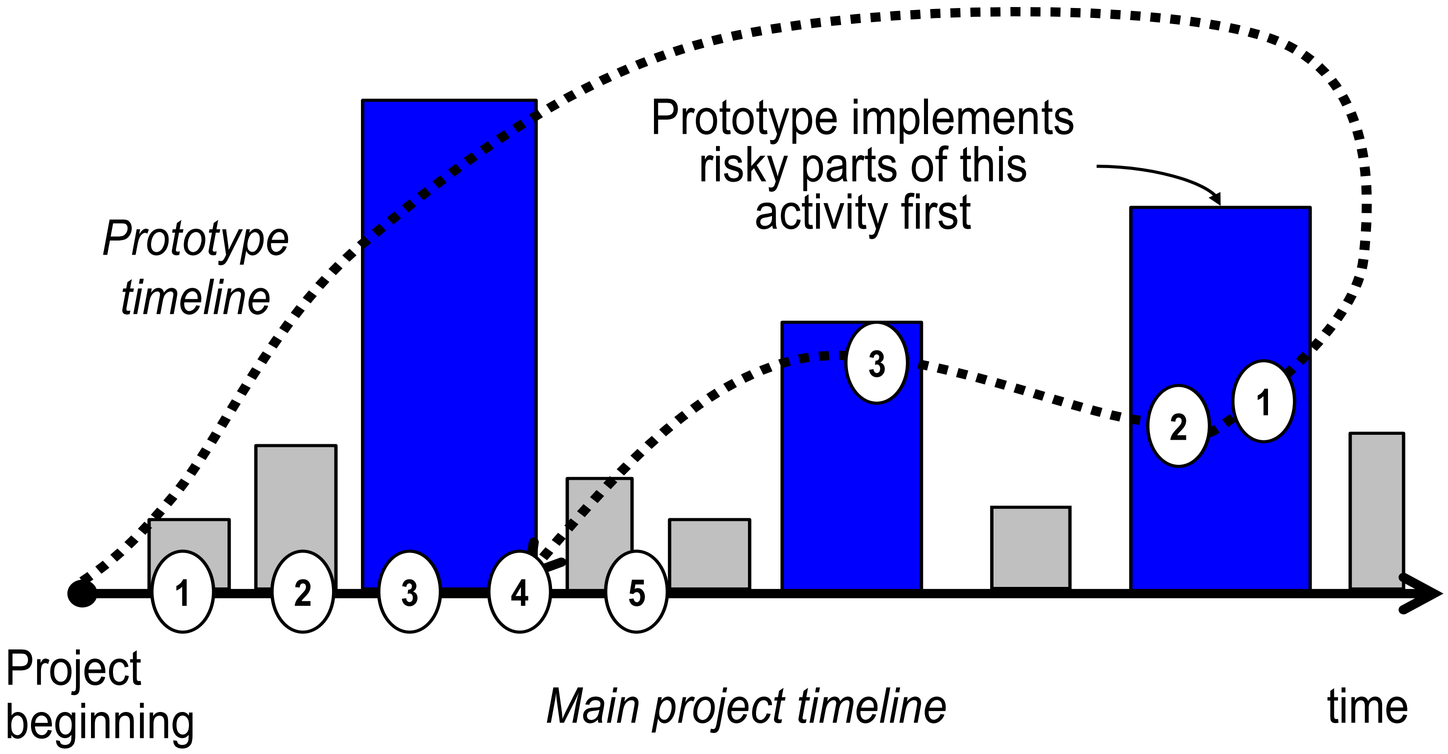


When to use the Incremental Model

- Risk, funding, schedule, program complexity, or need for early realization of benefits.
- Most of the requirements are known up-front but are expected to evolve over time
- A need to get basic functionality to the market early (or to project partners early)
- On projects which have lengthy development schedules
- On a project with new technology

Prototyping

Prototype Rationale



Prototype timeline

Prototype implements risky parts of this activity first

Project beginning

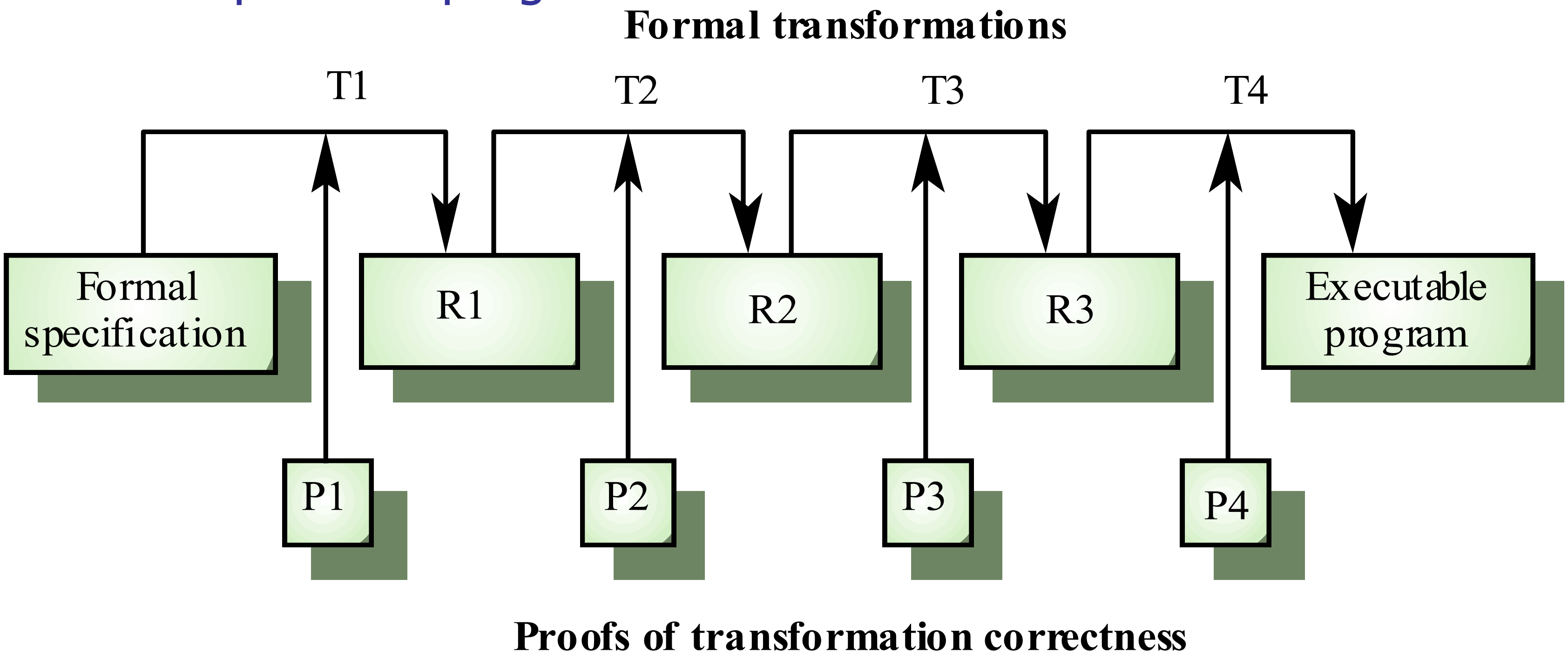
Main project timeline

time

Key: ⓝ = end of a unit of time 🟦 = Activity with risk

Formal Transformations

- Formal mathematical representation of the system is systematically converted into a more detailed, but still mathematically correct, system representation
- Each step adds more detail until the formal specification is converted into an equivalent program



Agile SDLC's

- Speed up or bypass one or more life cycle phases
- Usually less formal and reduced scope
- Used for time-critical applications
- Used in organizations that employ disciplined methods

- Some Agile Methods
  - Adaptive Software Development (ASD)
  - Feature Driven Development (FDD)
  - Crystal Clear
  - Dynamic Software Development Method (DSDM)
  - Rapid Application Development (RAD)
  - Scrum
  - Extreme Programming (XP)
  - Rational Unified Process (RUP)

Tailoring SDLC Models

- Any one model does not fit all projects
- If there is nothing that fits a particular project, pick a model that comes close and modify it for your needs.
- Project should consider risk
  - Is a complete spiral too much?
    - Start with spiral & pare it down
- Project delivered in increments
  - But there could be serious reliability issues – combine incremental model with the V-shaped model
- Each team must pick or customize a SDLC model to fit its project
- Often one model alone is insufficient for any one purpose.

Maintenance or Evolution

- Some observations
  - systems are not built from scratch
  - there is time pressure on maintenance
- The laws of software evolution, Meir (Manny) Lehman
  - law of continuing change - A system that is used will require adaptation and extension as time goes by.
  - law of increasing complexity - As a system evolves it also increases in complexity (unless work is done to reduce the complexity, e.g. architecture/design, refactoring)
  - law of program evolution - the path of evolution for systems is determined by a feedback process (from all stakeholders, but especially from end-users, clients and developers)