Software Lifecycle Models

A software lifecycle model is a standardised format for

- planning
- organising, and
- running

a new development project.

Hundreds of different kinds of models are known and used.

Many are minor variations on just a small number of basic models. In this section we:

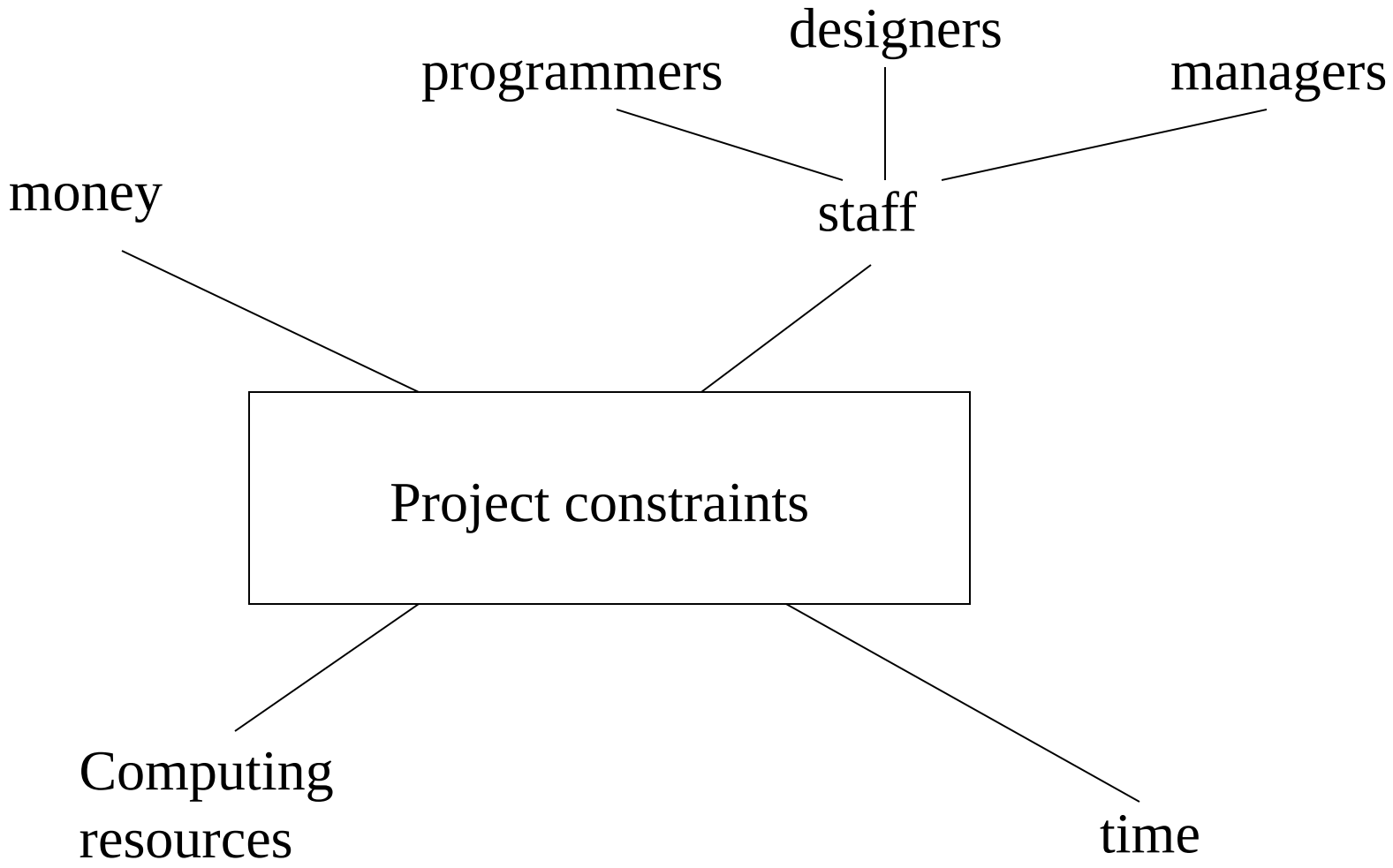
- **survey** the main types of model, and
- consider **how to choose** between them.

6.1. Planning with Models

SE projects usually live with a **fixed financial budget**. (An exception is maintainance?)

Additionally, time-to-market places a strong **time constraint**.

There will be other **project constraints** such as staff.



Examples of Project Constraints

Project planning is the art of scheduling the **necessary activities**, in time, space and across

staff in order to optimise:

- project risk [low] (see later)
- profit [high]
- customer satisfaction [high]
- worker satisfaction [high]
- long-term company goals

Questions:

1. What are these necessary activities? (besides programming)

2. Are there good patterns of organisation that we could copy?

A project plan contains much information,
but must at least describe:

- resources needed
  (***people, money, equipment, etc***)
- dependency & timing of work
  (***flow graph***, ***work packages***)
- rate of delivery (***reports, code, etc***)

It is impossible to measure **rate of progress**
except with reference to a plan.

In addition to project members, the following may need access to parts of the project plan:

- Management,
- Customers
- Subcontractors
- Suppliers
- Investors
- Banks

6.2. Project Visibility

Unlike other engineers
(e.g. civil, electronic, chemical … etc.)
software engineers do not produce anything ***physical***.

It is inherently difficult to monitor an SE project due to **lack of visibility**.

This means that SE projects must produce

additional deliverables (***artifacts***)

which **are visible**, such as:

- ***Design documents/ prototypes***
- ***Reports***
- ***Project/status meetings***
- ***Client surveys* (*e.g. satisfaction level*)**

6.3. What is a Lifecycle Model?

Definition.

A (software/system) *lifecycle model* is a description of the sequence of activities carried out in an SE project, and the relative order of these activities.

It provides a fixed **generic framework** that can be tailored to a specific project.

Project specific ***parameters*** will include:

- Size, (person-years)
- Budget,
- Duration.

project plan =

lifecycle model + project parameters

There are hundreds of different lifecycle models to choose from, e.g:

- ***waterfall,***
- ***code-and-fix***
- ***spiral***
- ***rapid prototyping***
- ***unified process (UP)***
- ***agile methods, extreme programming (XP)***
- ***COTS …***

but many are minor variations on a smaller number of basic models.

By changing the lifecycle model, we can **improve and/or tradeoff**:

- ***Development speed (time to market)***
- ***Product quality***
- ***Project visibility***
- ***Administrative overhead***
- ***Risk exposure***
- ***Customer relations, etc, etc.***

Normally, a lifecycle model covers the entire **lifetime of a product**.

From ***birth of a commercial idea***
to ***final de-installation of last release***

i.e. The three main phases:

- ***design***,
- ***build***,
- ***maintain***.

Note that we can sometimes **combine** lifecycle models,

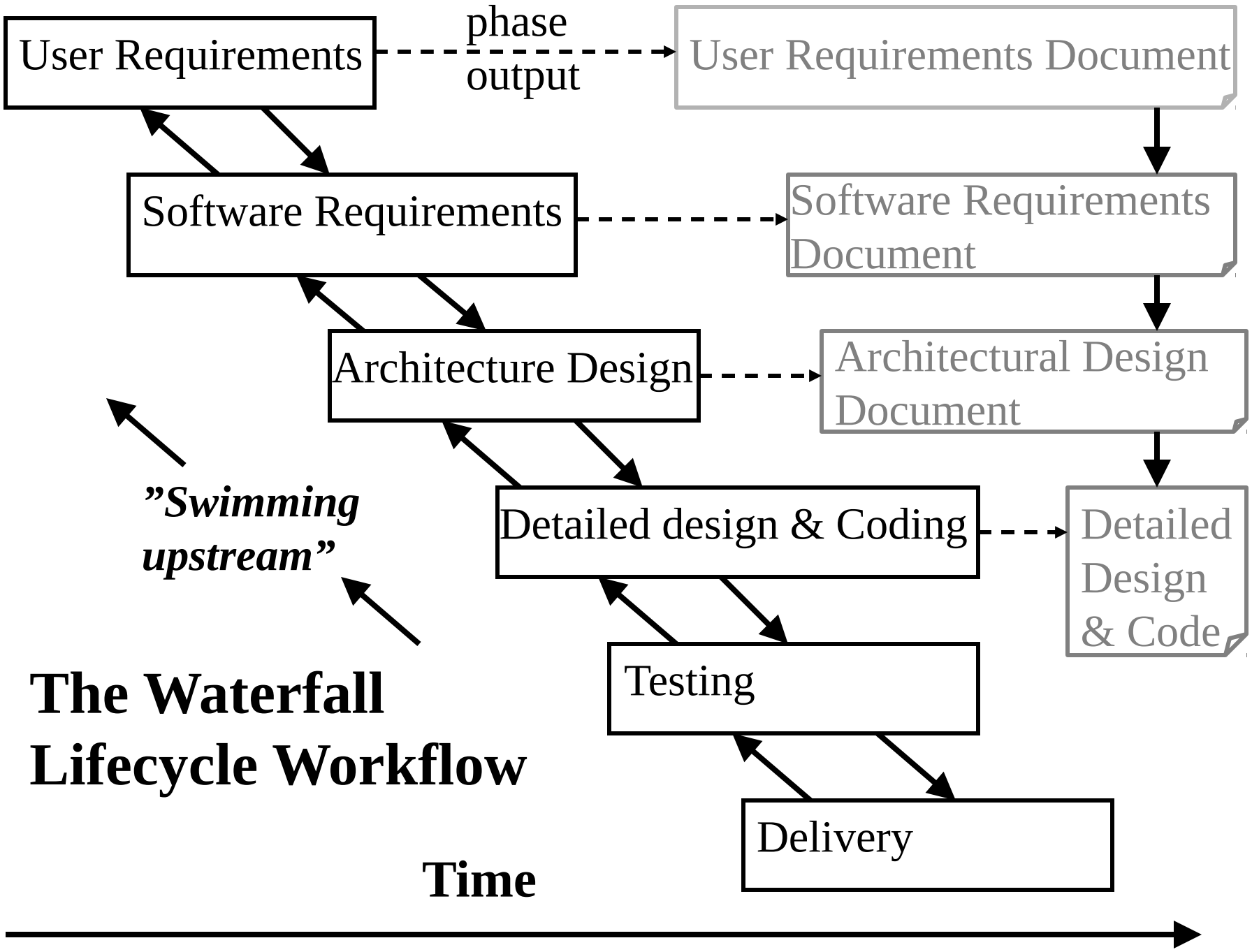
e.g. ***waterfall inside evolutionary*** – onboard shuttle software

We can also **change** lifecycle model between releases as a product matures,

e.g. ***rapid prototyping → waterfall***

6.4. The Waterfall Model

- The waterfall model is the classic lifecycle model – it is widely known, understood and (commonly?) used.
- In some respect, waterfall is the ”common sense” approach.
- Introduced by Royce 1970.


User Requirements
phase output
User Requirements Document
Software Requirements
Software Requirements Document
Architecture Design
Architectural Design Document
Detailed design & Coding
Detailed Design & Code
Testing
Delivery
”Swimming upstream”
The Waterfall Lifecycle Workflow
Time

Advantages

1. Easy to understand and implement.
2. Widely used and known (in theory!)
3. Reinforces good habits: define-before- design, design-before-code
4. Identifies deliverables and milestones
5. Document driven, URD, SRD, … etc. Published documentation standards, e.g. PSS-05.
6. Works well on mature products and weak teams.

Disadvantages I

1. Idealised, doesn’t match reality well.
2. Doesn’t reflect iterative nature of exploratory development.
3. Unrealistic to expect accurate requirements so early in project
4. Software is delivered late in project, delays discovery of serious errors.

Disadvantages II

5. Difficult to integrate risk management
6. Difficult and expensive to make changes to documents, ”swimming upstream”.
7. Significant administrative overhead, costly for small teams and projects.

6.5. Code-and-Fix

This model starts with an informal general product idea and just develops code until a product is ”ready” (or money or time runs out). Work is in random order.

Corresponds with no plan! (**Hacking**!)

Advantages

1. No administrative overhead
2. Signs of progress (code) early.
3. Low expertise, anyone can use it!
4. Useful for small "***proof of concept***" projects, e.g. as part of risk reduction.

Disadvantages

1. Dangerous!
    1. No visibility/control
    2. No resource planning
    3. No deadlines
    4. Mistakes hard to detect/correct
2. Impossible for large projects, communication breakdown, chaos.

6.6. Spiral Model

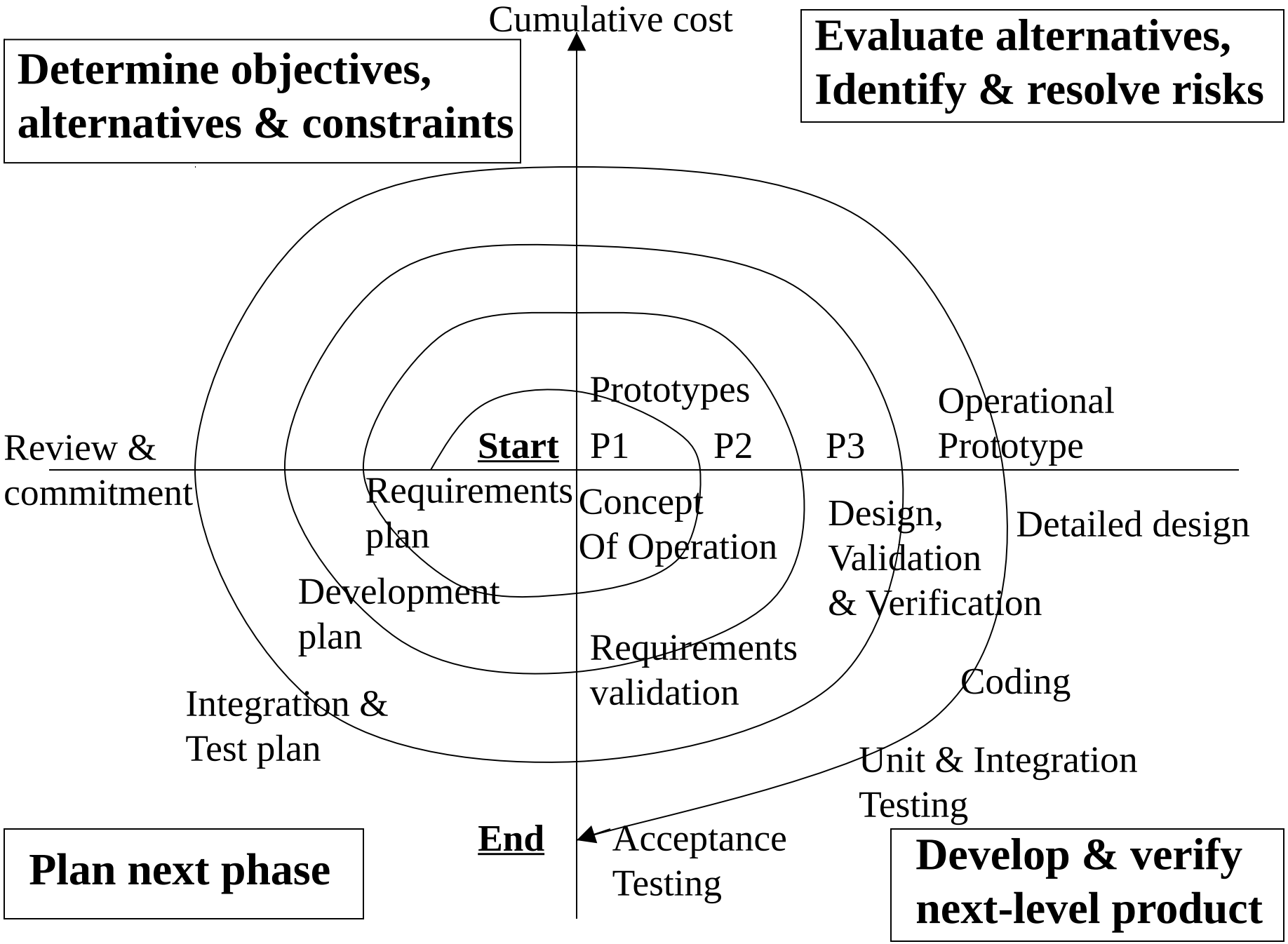
Since end-user requirements are hard to obtain/define, it is natural to develop software in an ***experimental*** way: e.g.

1. Build some software
2. See if it meets customer requirements
3. If no goto 1 else stop.

This loop approach gives rise to structured **iterative lifecycle models.**

In 1988 Boehm developed the spiral model as an iterative model which includes ***risk analysis*** and ***risk management***.

Key idea: on each iteration identify and solve the sub-problems with the ***highest risk***.


Cumulative cost
Determine objectives, alternatives & constraints
Evaluate alternatives, Identify & resolve risks
Review & commitment
Start
Prototypes
P1
P2
P3
Operational Prototype
Requirements plan
Concept Of Operation
Design, Validation & Verification
Detailed design
Development plan
Requirements validation
Coding
Integration & Test plan
Unit & Integration Testing
End
Acceptance Testing
Plan next phase
Develop & verify next-level product

Each cycle follows a waterfall model by:

1. Determining objectives
2. Specifying constraints
3. Generating alternatives
4. Identifying risks
5. Resolving risks
6. Developing next-level product
7. Planning next cycle

Advantages

1. Realism: the model accurately reflects the iterative nature of software development on projects with unclear requirements
2. Flexible: incoporates the advantages of the waterfal and rapid prototyping methods
3. Comprehensive model decreases risk
4. Good project visibility.

Disadvantages

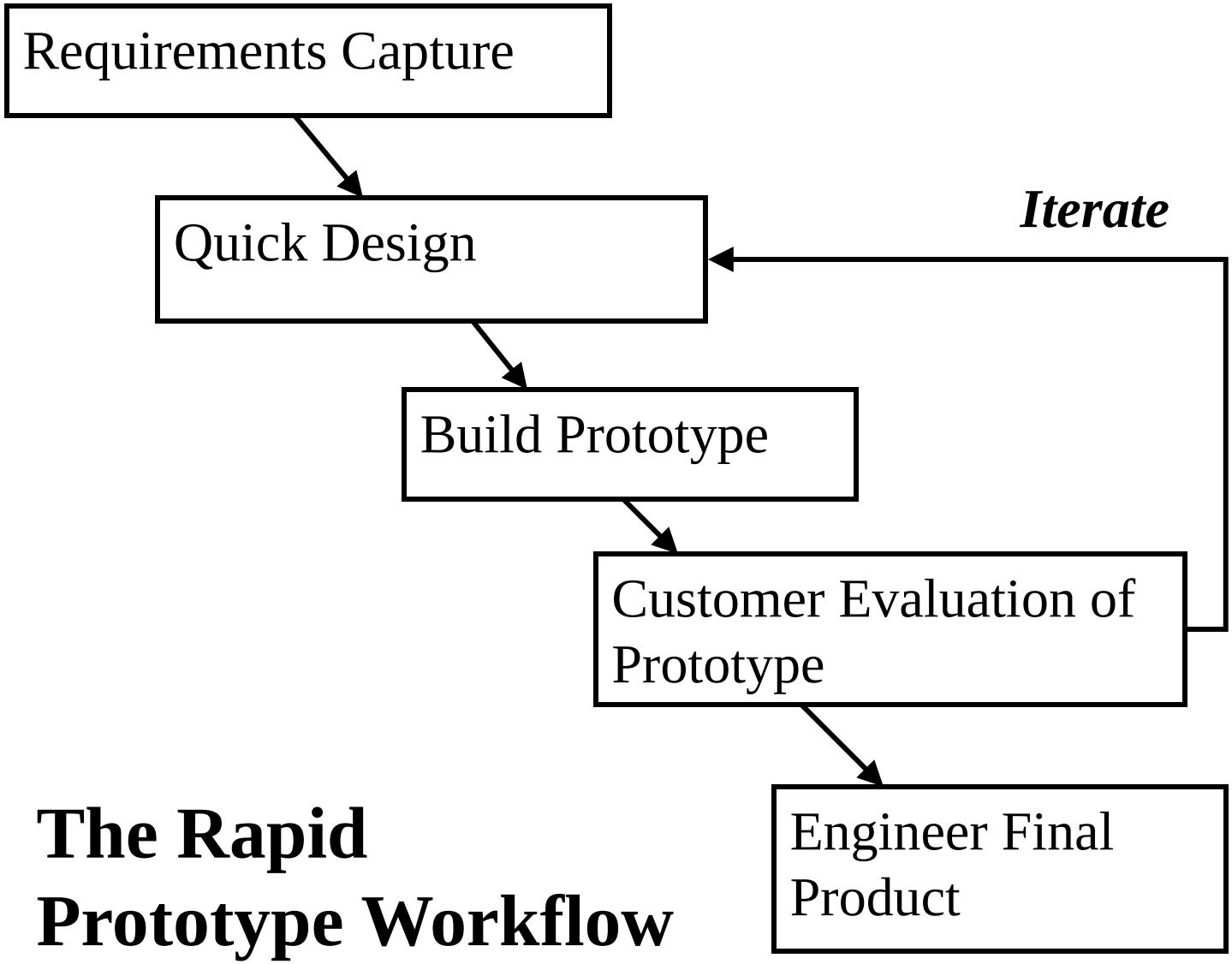
- Needs technical expertise in risk analysis to really work
- Model is poorly understood by non-technical management, hence not so widely used
- Complicated model, needs competent professional management. High administrative overhead.

6.7. Rapid Prototyping

Key idea: Customers are non-technical and usually don’t know what they want/can have.

Rapid prototyping emphasises requirements analysis and validation, also called:

- ***customer oriented development***,
- ***evolutionary prototyping***



The Rapid Prototype Workflow

Advantages

1. Reduces risk of incorrect user requirements
2. Good where requirements are changing/uncommitted
3. Regular visible progress aids management
4. Supports early product marketing

Disadvantages I

1. An unstable/badly implemented prototype often becomes the final product.
2. Requires extensive customer collaboration
   - Costs customers money
   - Needs committed customers
   - Difficult to finish if customer withdraws
   - May be too customer specific, no broad market

Disadvantages II

3. Difficult to know how long project will last
4. Easy to fall back into code-and-fix without proper requirements analysis, design, customer evaluation and feedback.

6.8. Agile (XP) Manifesto

XP = Extreme Programming emphasises:

- Individuals and interactions
  - **Over processes and tools**
- Working software
  - **Over documentation**
- Customer collaboration
  - **Over contract negotiation**
- Responding to change
  - **Over following a plan**

6.8.1. Agile Principles (Summary)

- Continuous delivery of software
- Continuous collaboration with customer
- Continuous update according to changes
- Value participants and their interaction
- Simplicity in code, satisfy the spec

6.9. XP Practices (Summary)

- Programming in pairs
- Test driven development
- Continuous planning, change , delivery
- Shared project metaphors, coding standards and ownership of code
- No overtime! (Yeah right!)

Advantages

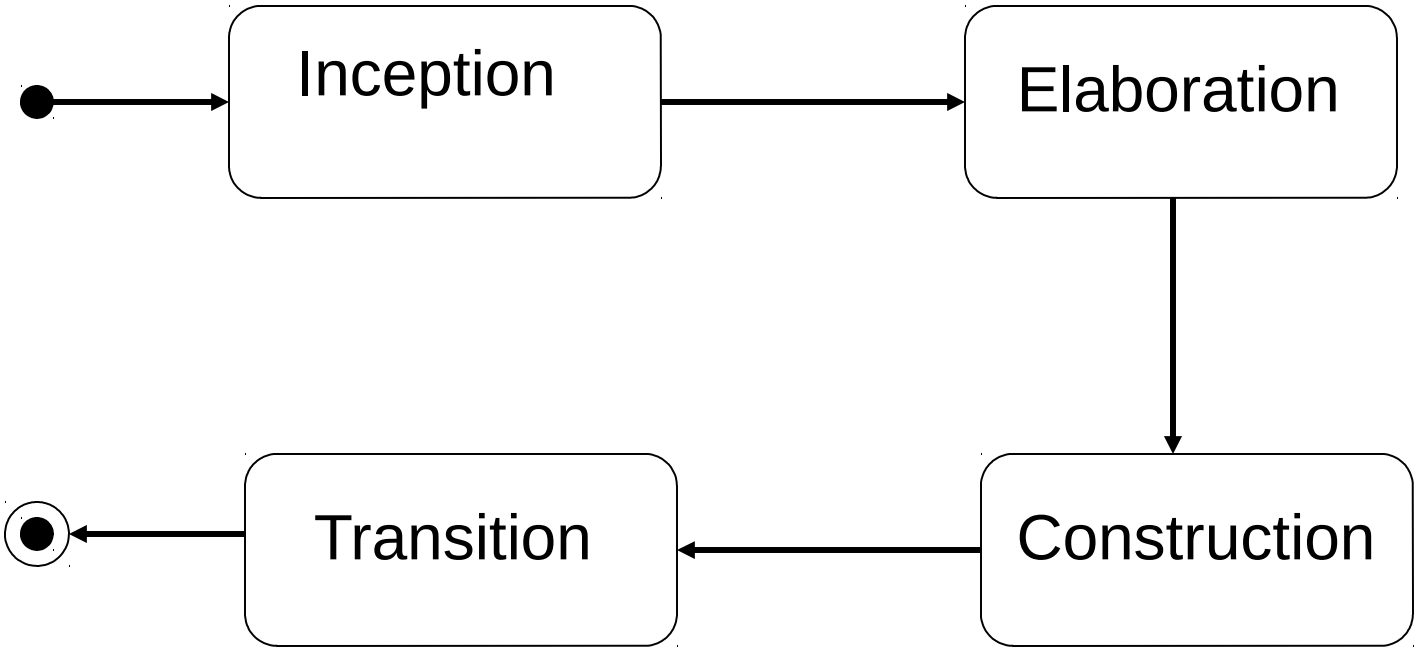
- Lightweight methods suit small-medium size projects
- Produces good team cohesion
- Emphasises final product
- Iterative
- Test based approach to requirements and quality assurance

Disadvantages

- Difficult to scale up to large projects where documentation is essential
- Needs experience and skill if not to degenerate into code-and-fix
- Programming pairs is costly
- Test case construction is a difficult and specialised skill.

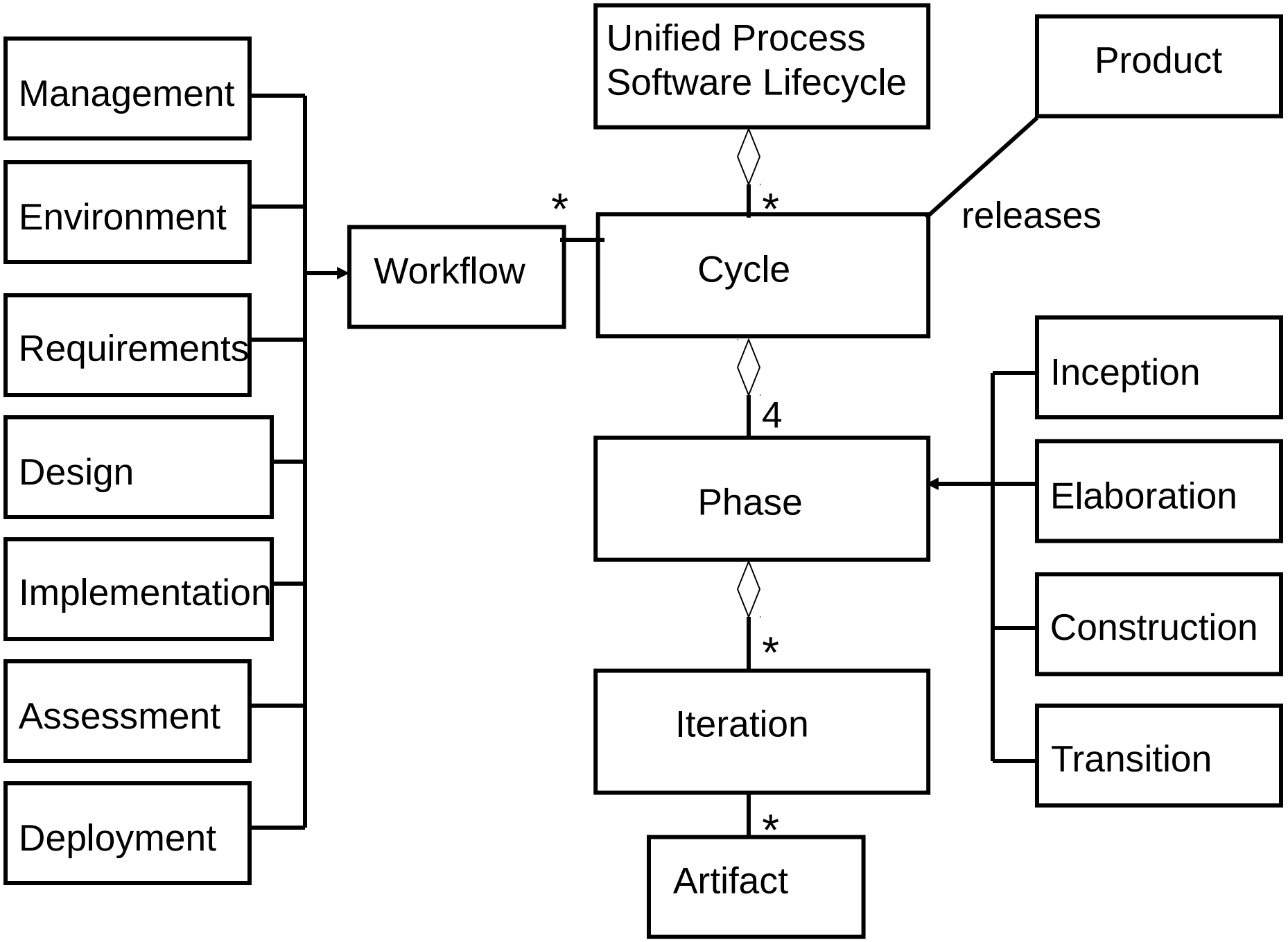
6.10. Unified Process (UP)

- Booch, Jacobson, Rumbaugh 1999.
- Lifetime of a software product in **cycles**:
- ***Birth, childhood, adulthood, old-age, death***.
- Product maturity stages
- Each cycle has phases, culiminating in a new release (c.f. Spiral model)

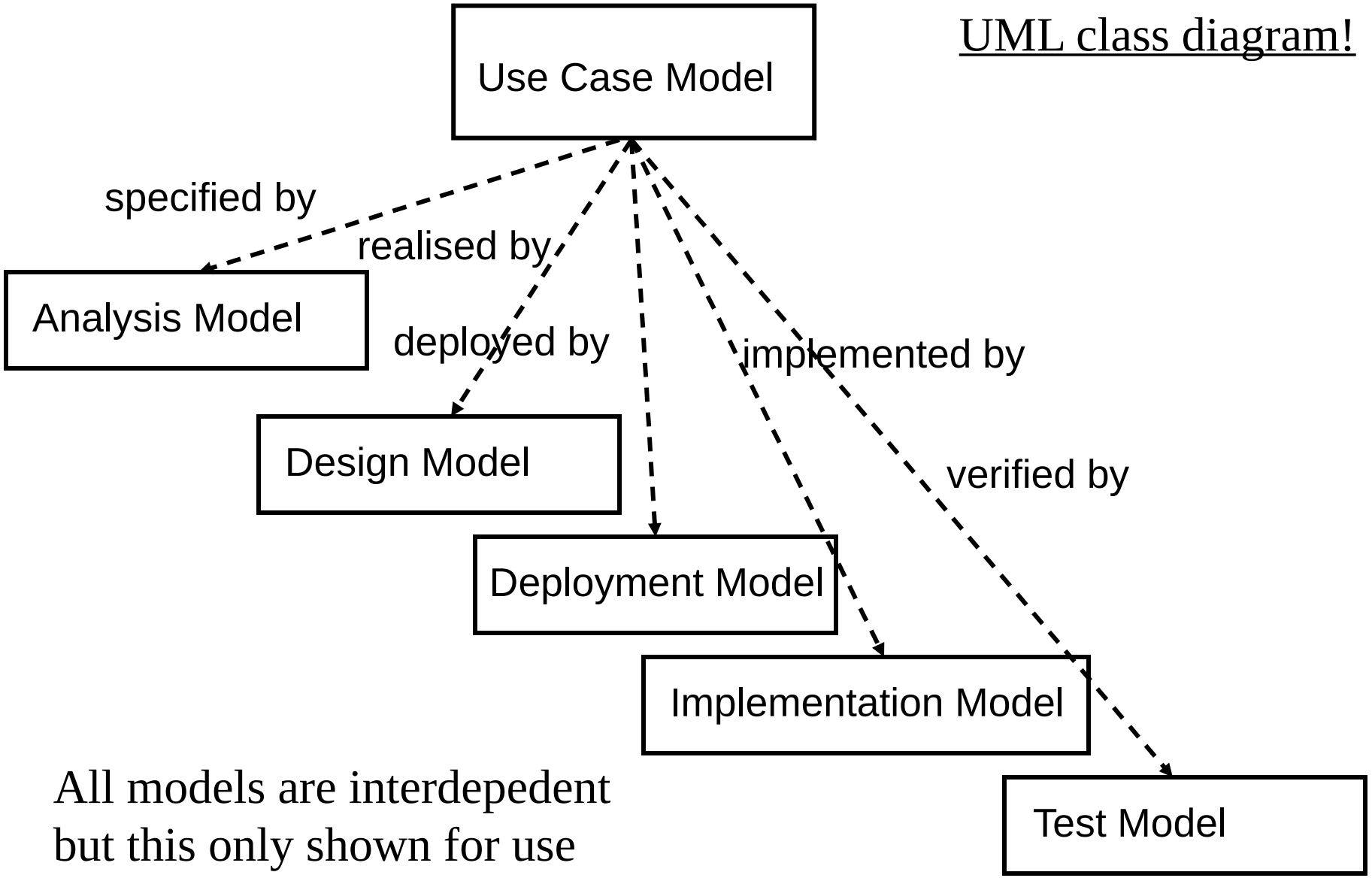


UP Lifecycle – single phase workflow (drawn as a UML Statechart!)

- **Inception** – identify core use cases, and use to make architecture and design tradeoffs. Estimate and schedule project from derived knowledge.
- **Elaboration** – capture detailed user requirements. Make detailed design, decide on build vs. buy.
- **Construction** – components are bought or built, and integrated.
- **Transition** – release a mature version that satisfies acceptance criteria.


Unified Process
Software Lifecycle
Product
releases
Management
Environment
Requirements
Design
Implementation
Assessment
Deployment
Workflow
Cycle
*
*
4
Phase
Inception
Elaboration
Construction
Transition
*
Iteration
*
Artifact

UML class diagram!



All models are interdepedent but this only shown for use case model

6.11. COTS

- **COTS =**

  Commercial Off-The-Shelf software
- Engineer together a solution from existing commercial software packages using minimal software ***”glue”*.**
- E.g. using databases, spread sheets, word proccessors, graphics software, web browsers, etc.

Advantages

- Fast, cheap solution
- May give all the basic functionality
- Well defined project, easy to run

Disadvantages

- Limited functionality
- Licensing problems, freeware, shareware, etc.
- License fees, maintainance fees, upgrade compatibility problems