Introduction to Software Engineering Methods

Introduction & Software LifeCycle

Introduction Content

- Introduction
- Software nature
- Definitions of Software Engineering
- Dealing with complexity and change
  - Abstraction
  - Decomposition
  - Hierarchy
- Software development concepts and activities
- Software development lifecycle

Literature used

- Text book "Object-Oriented Software Engineering: Using UML, Patterns and Java" International Edition, 3/E Bernd Bruegge, Allen H. Dutoit

  Chapters 1, 15, 16

What is Software?

- computer programs
- associated documentation
- associated data that is needed to make the programs operatable

Software’s Nature

- Software is
  - intangible – it is difficult to understand development efforts
  - untrained people can hack something together
  - easily reproducible – cost is not in manufacturing but in development
  - labor-intensive – hard to automate
  - easy to modify
  - a logical rather than physical product – it doesn’t wear out with use

Some known software failures

- **Item:** In the summer of 1991, telephone outages occurred in local telephone systems in California and along the Eastern seaboard. These breakdowns were all the fault of an error in signaling software. Right before the outages, DSC Communications (Plano, TX) introduced a bug when it changed three lines of code in the several-million-line signaling program. After this tiny change, nobody thought it necessary to retest the program.
- **Item:** In 1986, two cancer patients at the East Texas Cancer Center in Tyler received fatal radiation overdoses from the Therac-25, a computer-controlled radiation-therapy machine. There were several errors, among them the failure of the programmer to detect a race condition (i.e., miscoordination between concurrent tasks).
- **Item:** A New Jersey inmate escaped from computer-monitored house arrest in the spring of 1992. He simply removed the rivets holding his electronic anklet together and went off to commit a murder. A computer detected the tampering. However, when it called a second computer to report the incident, the first computer received a busy signal and never called back.

From http://www.byte.com/art/9512/sec6/art1.htm

Example: Space Shuttle Software

- Cost: $10 Billion, millions of dollars more than planned
- Time: 3 years late
- Quality: First launch of Columbia was cancelled because of a synchronization problem with the Shuttle's 5 onboard computers.
  - Error was traced back to a change made 2 years earlier when a programmer changed a delay factor in an interrupt handler from 50 to 80 milliseconds.
  - The likelihood of the error was small enough, that the error caused no harm during thousands of hours of testing.
- Substantial errors still exist.
  - Astronauts are supplied with a book of known software problems "Program Notes and Waivers".

Do you know this product?

Software failures – some common reasons

- Seldom occurring situations are not taken into account
- Users can actively misusing systems
- Unnecessary complexity
- . . .

Software project failures

- Terminated for non-performance of contract.
- Completed but the system is not deployed as users cannot or will not use it.
- Completed but the system does not meet the originally promised
  - cost
  - schedule.
  - quality.
  - capability.
- Completed but the system could not be evolved in a cost-effective manner

Adopted from Charette 1989

Software projects

- The Standish Group delivered “Chaos Report”
  – 365 IT executives in US companies in diverse industry segments.
  – 8,380 projects

- In 1994 only 16.2% of software projects were completed on-time and on-budget (for large and complex systems it is 9%)
  – average time overrun = 222%.
  – average cost overrun = 189%
  – 61% of originally specified features included

- In 2003 , it is 34% of projects completed on-time and on-budget

Reasons for changes (observation from LinkedIn groups)

- the project owner identifies new business opportunities and decides to integrate them into the software being developed;
- due to the technical nature of software development projects, there is a lack of shared understanding of expected outcomes
- original planning was based on specifications that were misinterpreted by the project manager or poorly illustrated by the project owner;
- project team is unable to implement planned functionalities due to lack of expertise or technological limitations;
- the context in which the software is going to be used changes thus generating the need for the software to change;
- new technology or software product is launched on the market

Summarizing Software Nature

- Demand for software is high and rising
- In many cases software has poor design
- “software crisis” in a permanent state
- We have to learn to engineer software

Software Engineering

"The amateur software engineer is always in search of magic, some sensational method or tool whose application promises to render software development trivial. It is the mark of the professional software engineer to know that no such panacea exists."

Grady Booch, in Object-Oriented Analysis and Design

What is Software Engineering?

- *Software Engineering is the establishment and use of <u>sound engineering</u> <u>principles</u> in order to <u>obtain economically</u> software that is <u>reliable</u> and works <u>efficiently on real machines</u>* [from Fritz Bauer in Pressman 1997, pA-1]
- **sound engineering principles** implies:
  - a specification followed by an implementation
  - a demonstration that the solution does what it is supposed to do
  - the adoption of sound project management practices
  - the availability of a range of tools and techniques
- **obtain economically** implies:
  - productivity estimation measures
  - cost estimation and measures
- **reliability and efficiency** imply:
  - performance measures
  - standards
  - quality

What is Software Engineering?

The IEEE describes Software Engineering as *“the application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software”*

- **quantifiable** implies measurement
- the discipline is not only concerned with **development** but also **maintenance** and **operation**

What is Software Engineering?

Brygge&Dutoit:

Software Engineering is a collection of techniques, methodologies and tools that help with the production of

- a high-quality software system
- with a given budget
- before a given deadline
- while change occurs.

What is Software Engineering?

- Software engineering is a modeling activity.
- Software engineering is a problem-solving activity.
- Software engineering is a knowledge acquisition activity.

Scientist, Engineer and Software Engineer

- (Computer) scientist
  - concerns general theories and methods that underlie computer and software systems
  - Has (almost) no time constraints
- Engineer
  - works in application specific domain
  - has time constraints
- Software engineer
  - works in multiple application domains
  - has time constraints
  - changes can occur in requirements and technology

Factors of Software Design

- Complexity
  - The problem domain is difficult because of often we are not experts in it
  - The development process is very difficult to manage
  - Software is extremely flexible – easy to modify
- Changes
  - each implemented change erodes the structure of the system which makes next change more expensive
  - the cost of implementing changes grows in time

How to deal with complexity?

- Abstraction

  -- Ignore non-essential details - modeling

- Decomposition

  -- Break problem into sub-problems

- Hierarchy

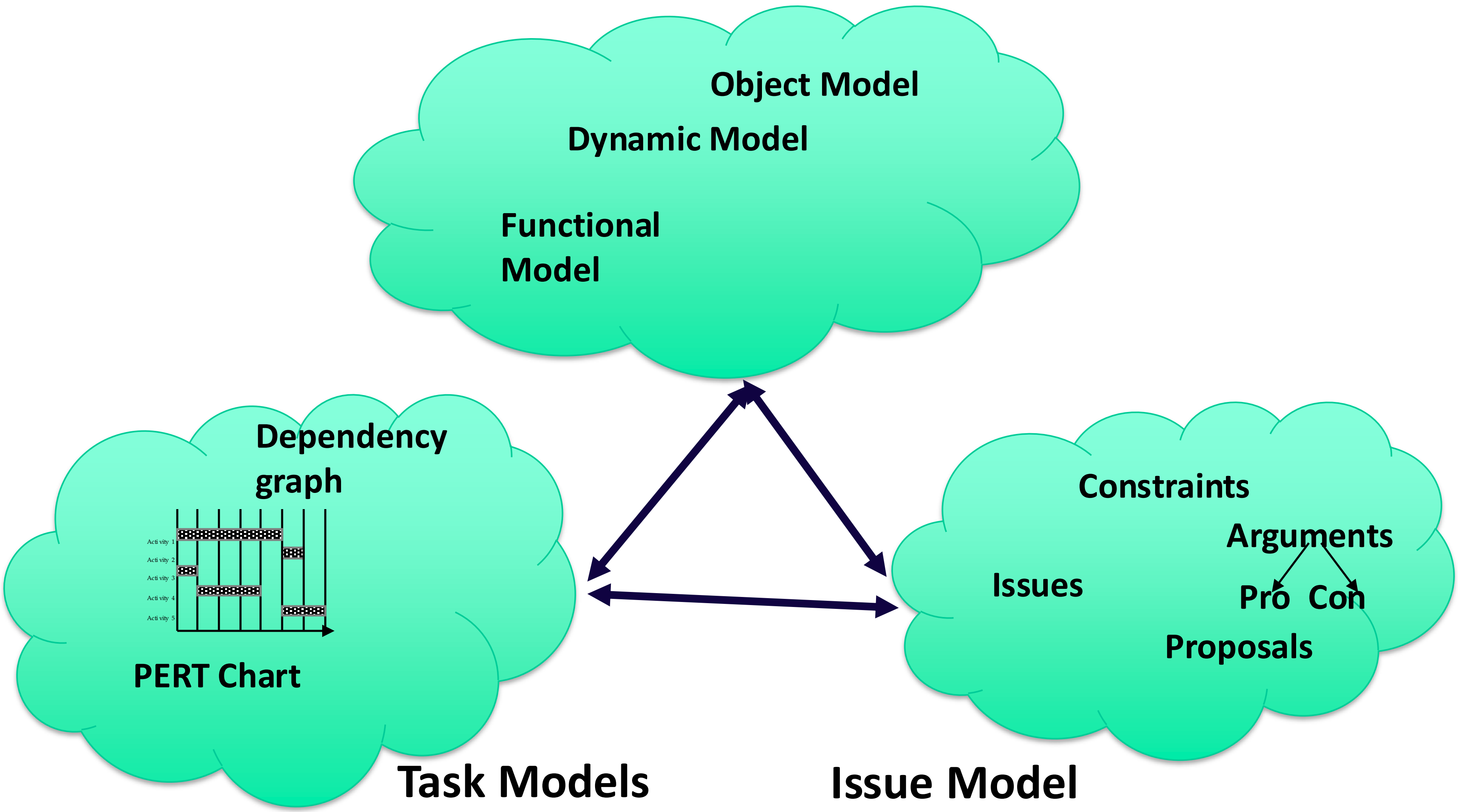
  -- Simple relationship between chunks

Abstraction/Modeling

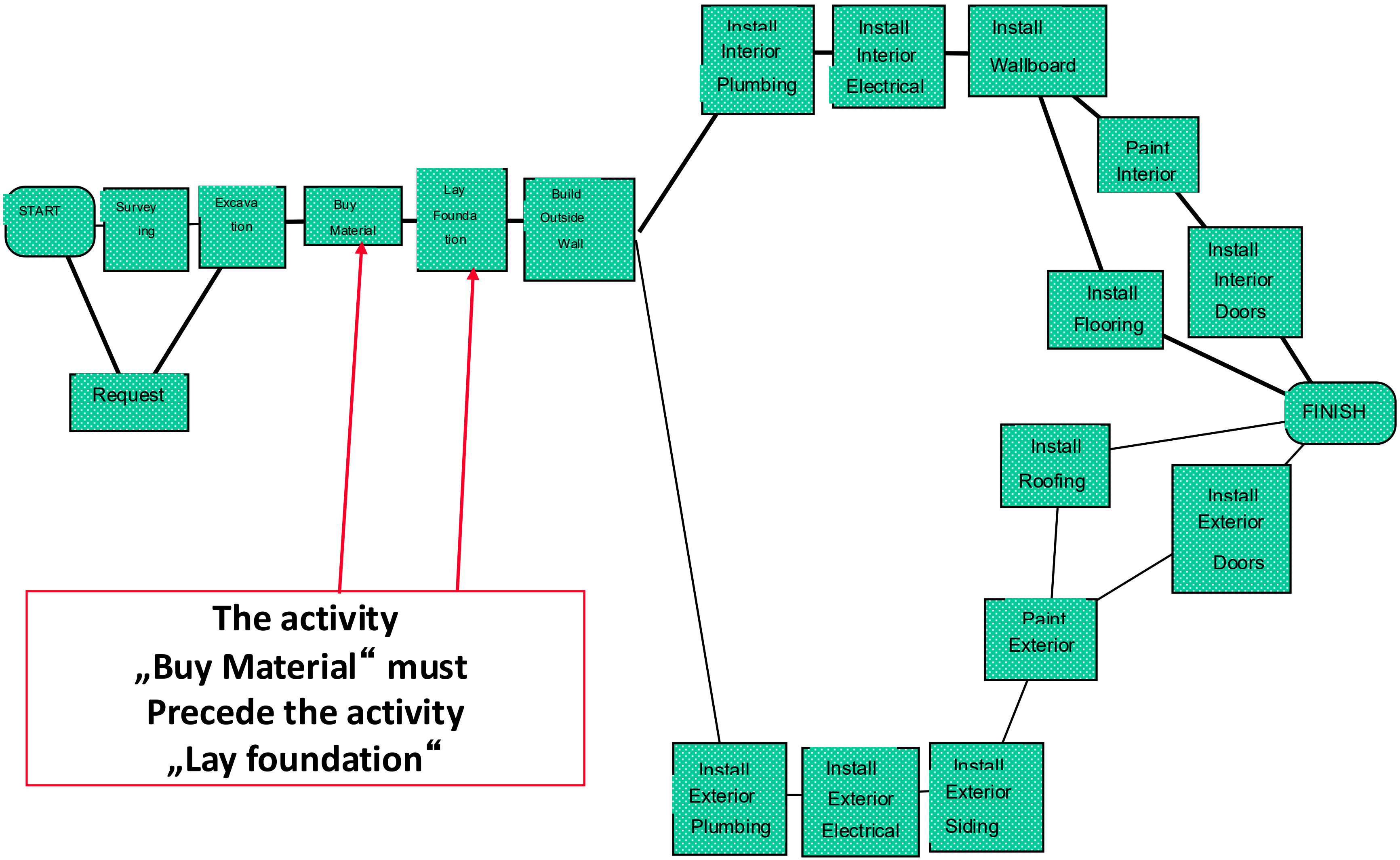
- System Model:
  - Functional model: What are the functions of the system? How is data flowing through the system?
  - Object Model: What is the structure of the system? What are the objects and how are they related?
  - Dynamic model: How does the system react to external events? How is the event flow in the system ?
- Task Model:
  - What are the dependencies between the tasks?
  - How can this be done within the time limit?
  - What are the roles in the project or organization?
- Issues Model:
  - What are the open and closed issues? What constraints were posed by the client? What resolutions were made?

Abstraction/Models

System Models

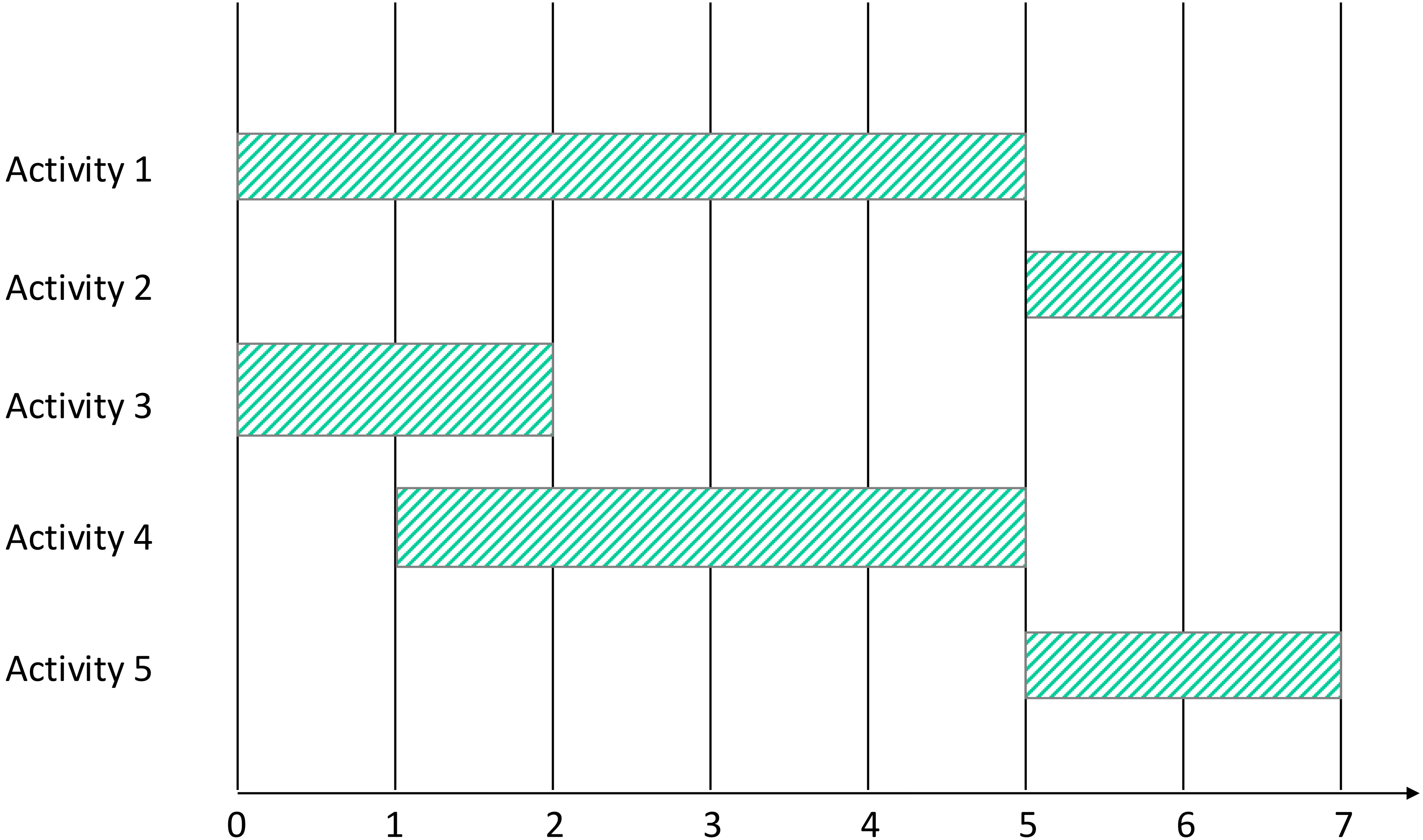


Dependency Graph



The activity „Buy Material“ must Precede the activity „Lay foundation“

Gantt Chart

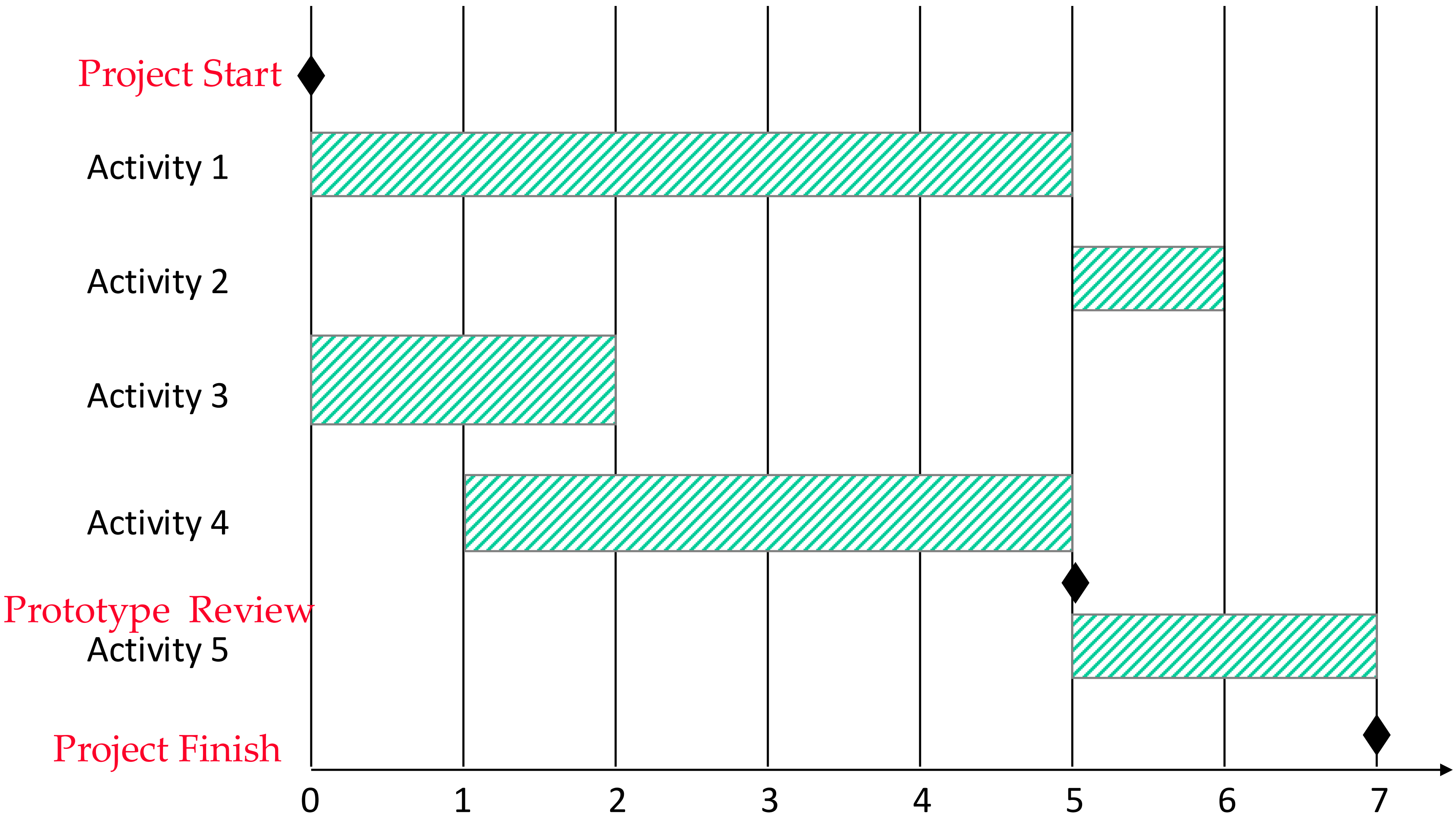


Easy to understand

Time (in weeks after start)

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Gantt Chart

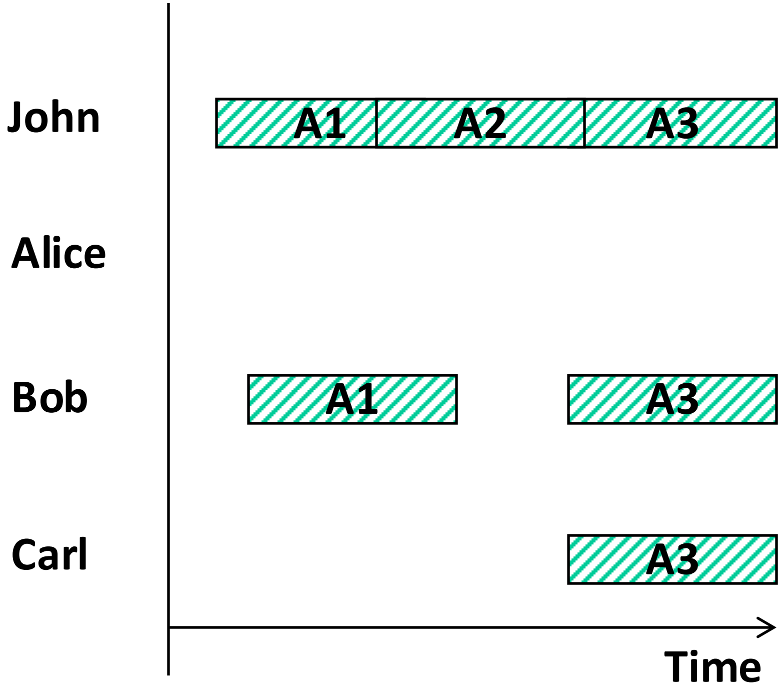


Easy to review

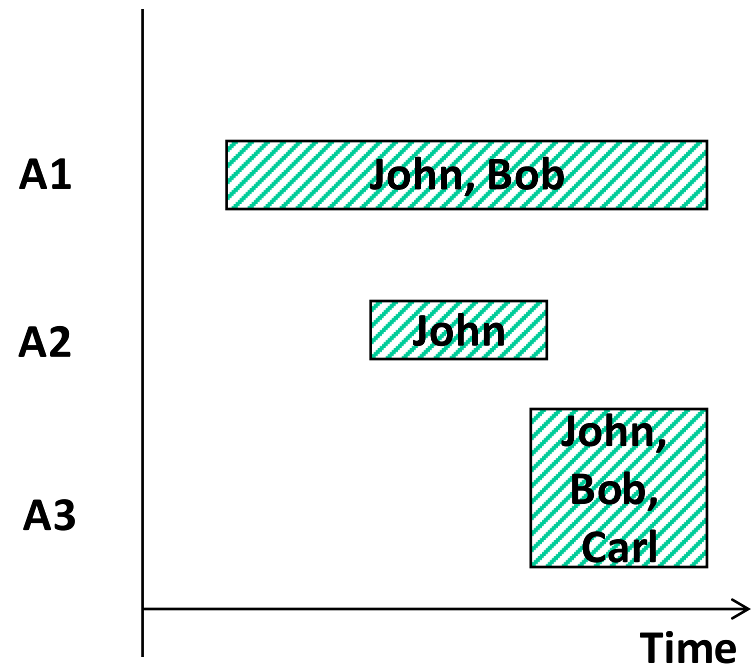
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Types of Gantt Charts

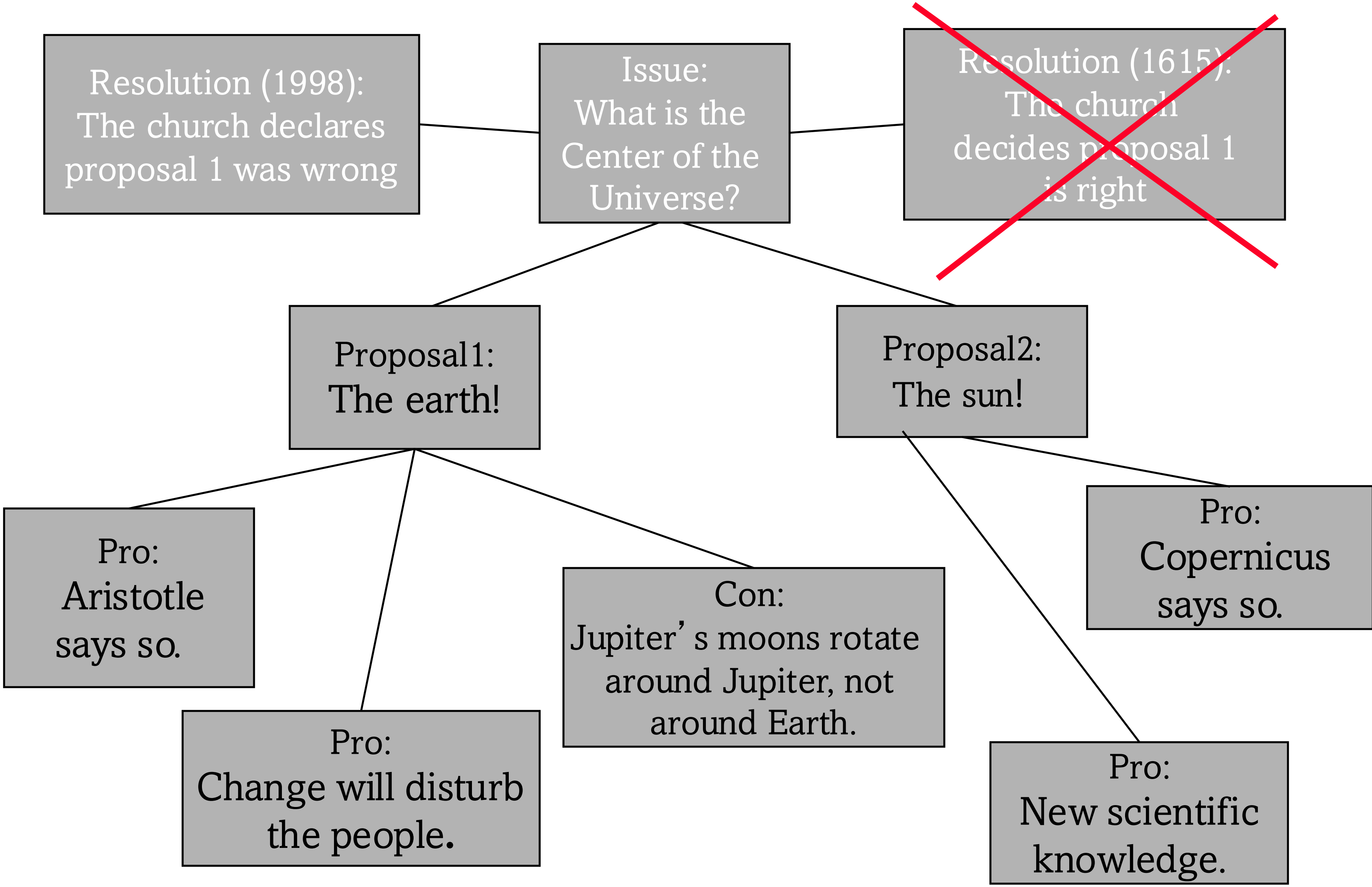
Person-Centered View



Activity-Centered View



Issue Model

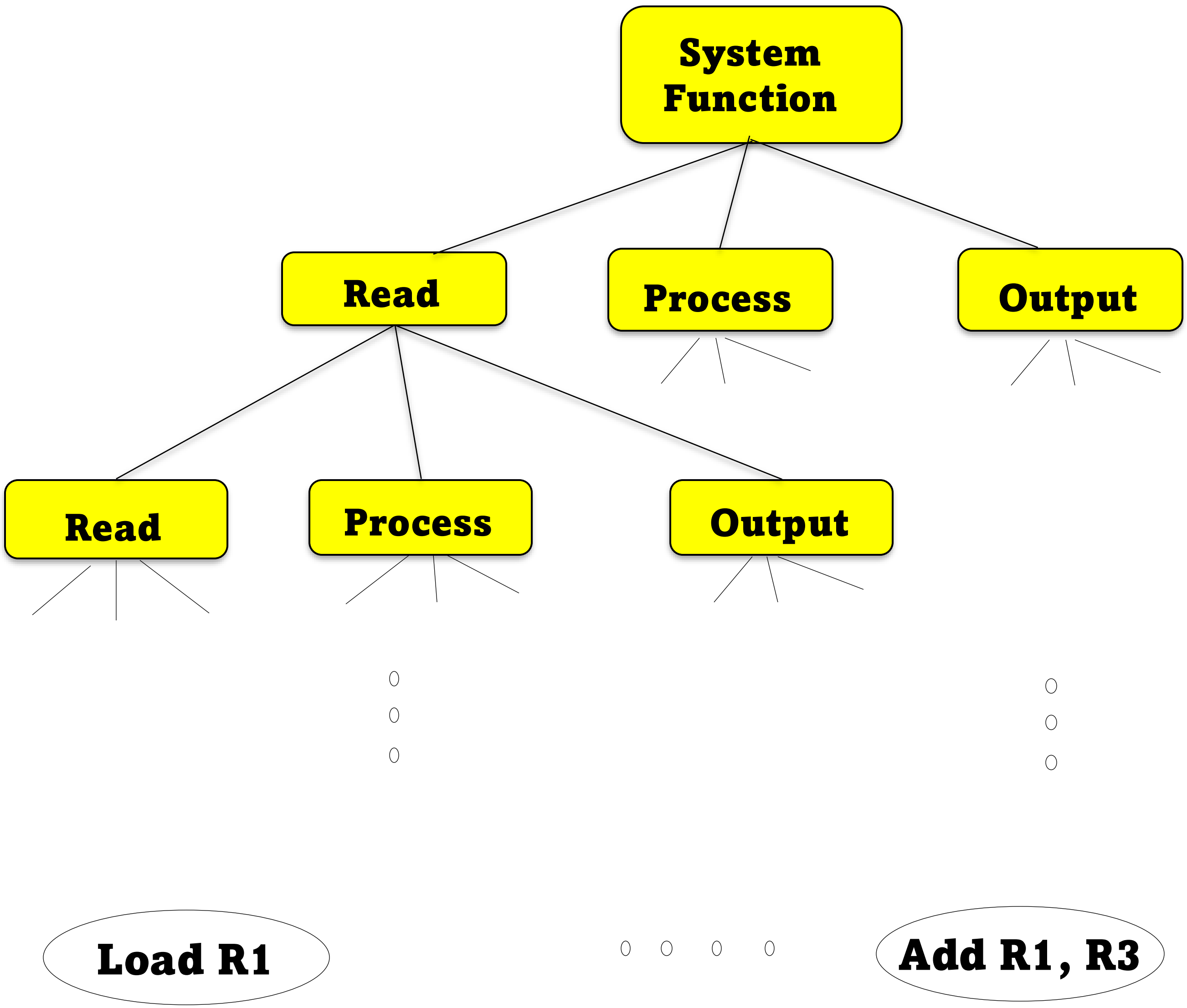


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Decomposition

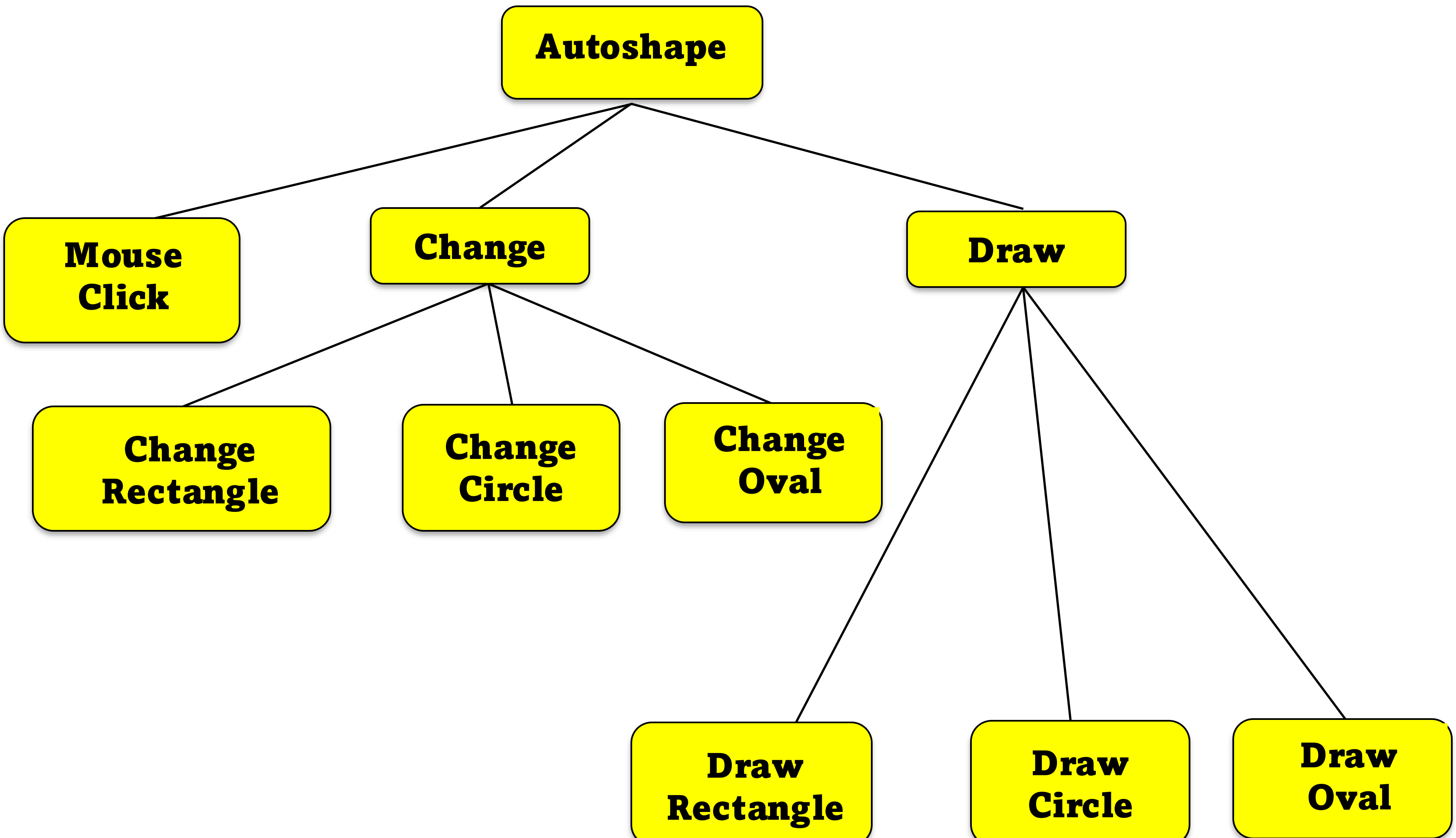
- Functional decomposition - emphasizes the ordering of operations
  – The system is decomposed into functional modules
  – Each module is a processing step (function) in the application domain
  – Modules can be decomposed into smaller modules
- Object-oriented decomposition - emphasizes the agents that cause the operations
  – The system is decomposed into classes (“objects”)
  – Each class is a major abstraction in the application domain
  – Classes can be decomposed into smaller classes

Functional Decomposition



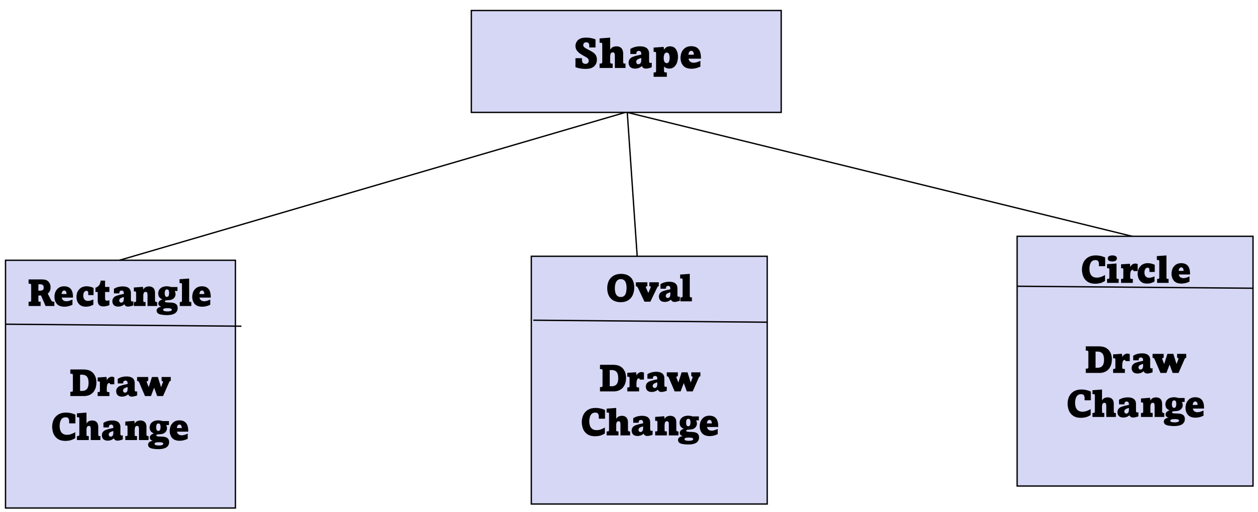
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Functional decomposition

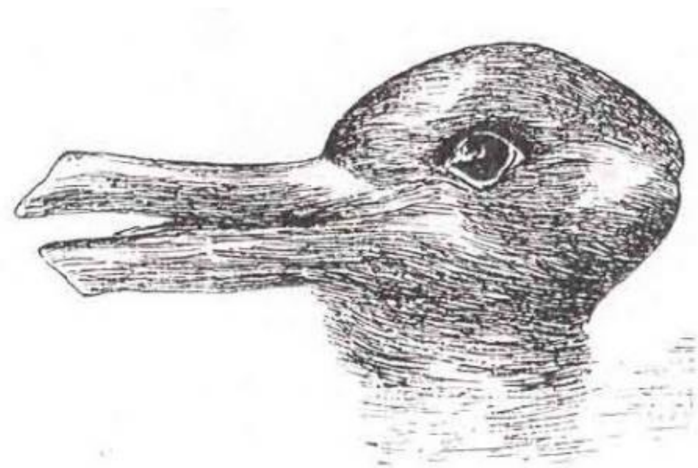


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Object decomposition



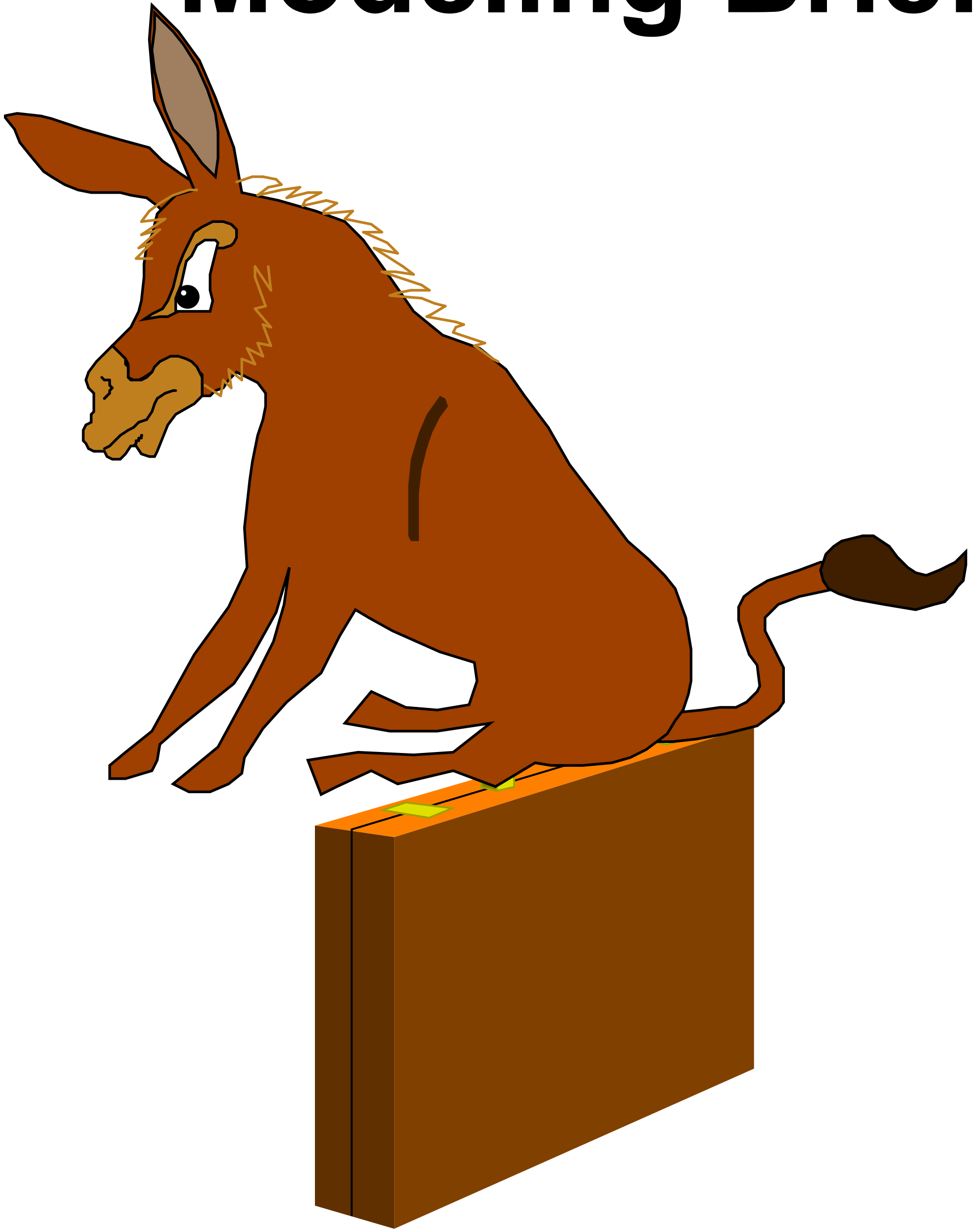
Object-Oriented Decomposition



Rabbit
Hair
Eat() Run()

Duck
Fathers
Swim() Eat() Fly()

Modeling Briefcase



BriefCase
Chair

Capacity: Integer
Weight: Integer

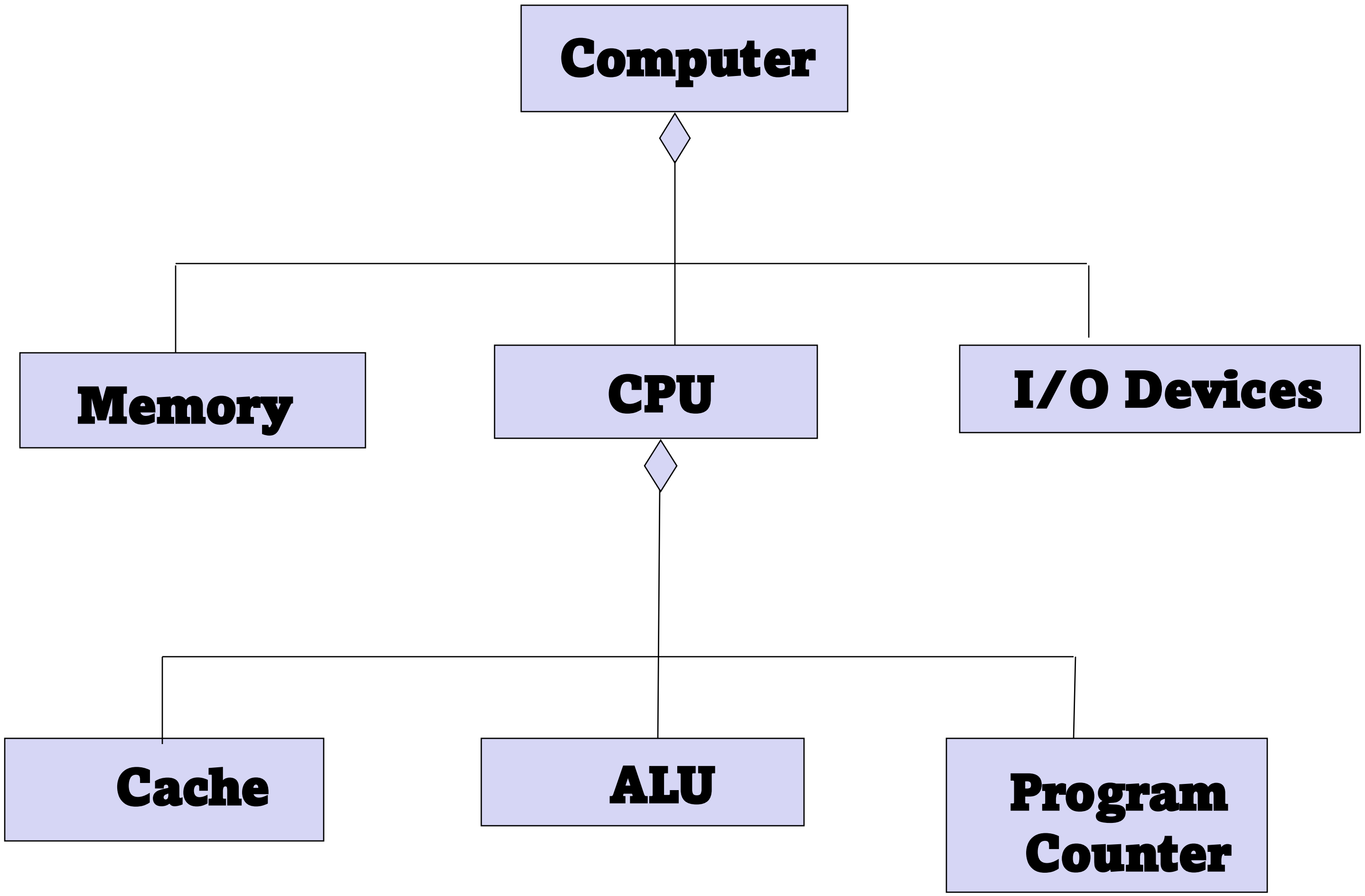
Open()
Close()
Carry()

SitOnIt()

Hierarchy

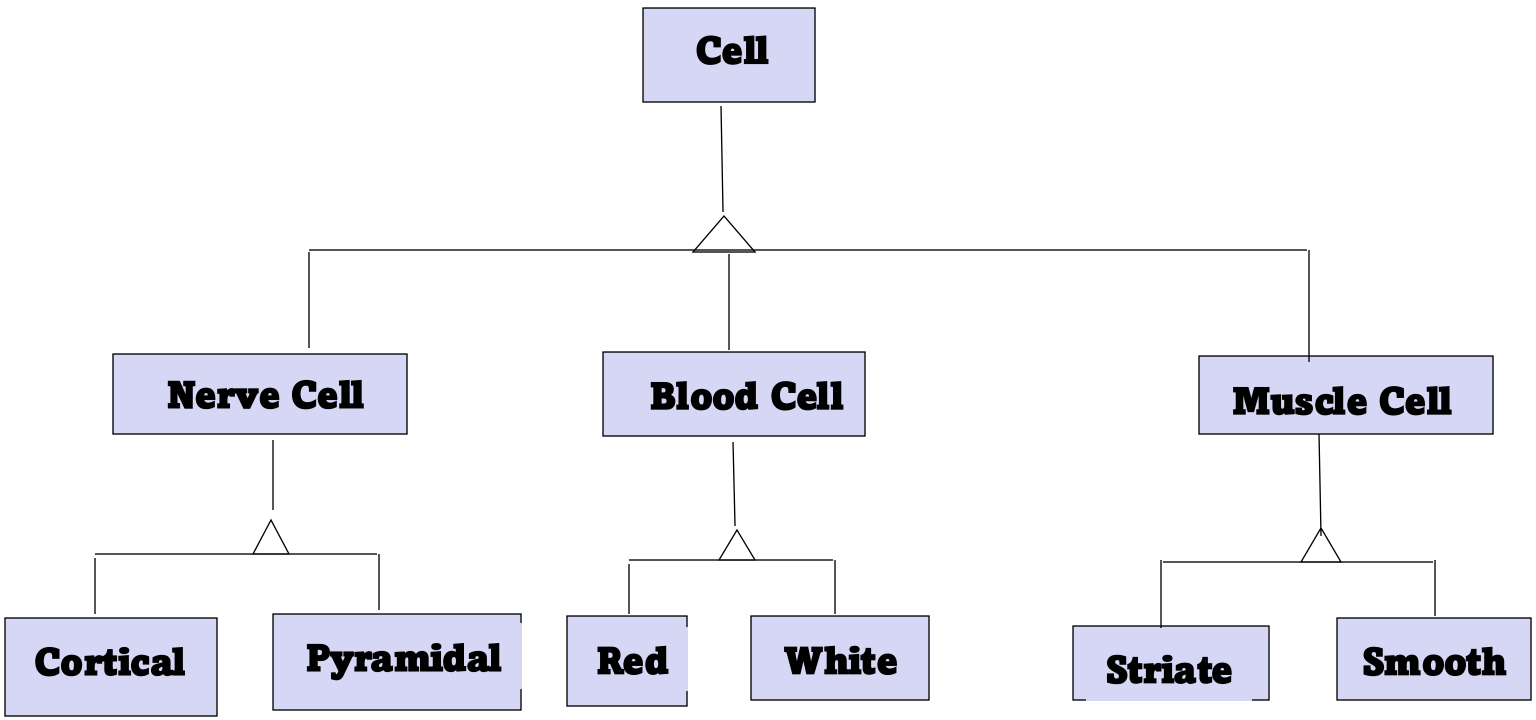
- Relationships between components obtained from abstraction and decomposition
- Hierarchy is one of ways to provide simple relationships
- Part-of-hierarchy
- Is-kind-of hierarchy

Part-of hierarchy



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Is-a-kind-of hierarchy



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Intermediate summary

- Three ways to deal with complexity:
  - Abstraction
  - Decomposition
  - Hierarchy
- Object-oriented decomposition is a good methodology
  - Unfortunately, depending on the purpose of the system, different objects can be found
- How can we do it right?
  - Our current approach:
    - Start with a description of the functionality,
    - then proceed to the object model
  - This leads us to the software lifecycle

Software Engineering Concepts (resources)

- Resources - time, equipment and participants
  - F/E People, Tarif Database
- Participants – all actors involved in the project
  - F/E traveler, train company, John, Alice, Zoe
- Roles – set of responsibilities in the project associated with a set of tasks
  - F/E client, user, manager, developer, technical writer

  and they are assigned to participants
  - F/E Alice – manager, John – technical writer and analyst
  - participant can fill multiple roles

Software Engineering Concepts (Work Products)

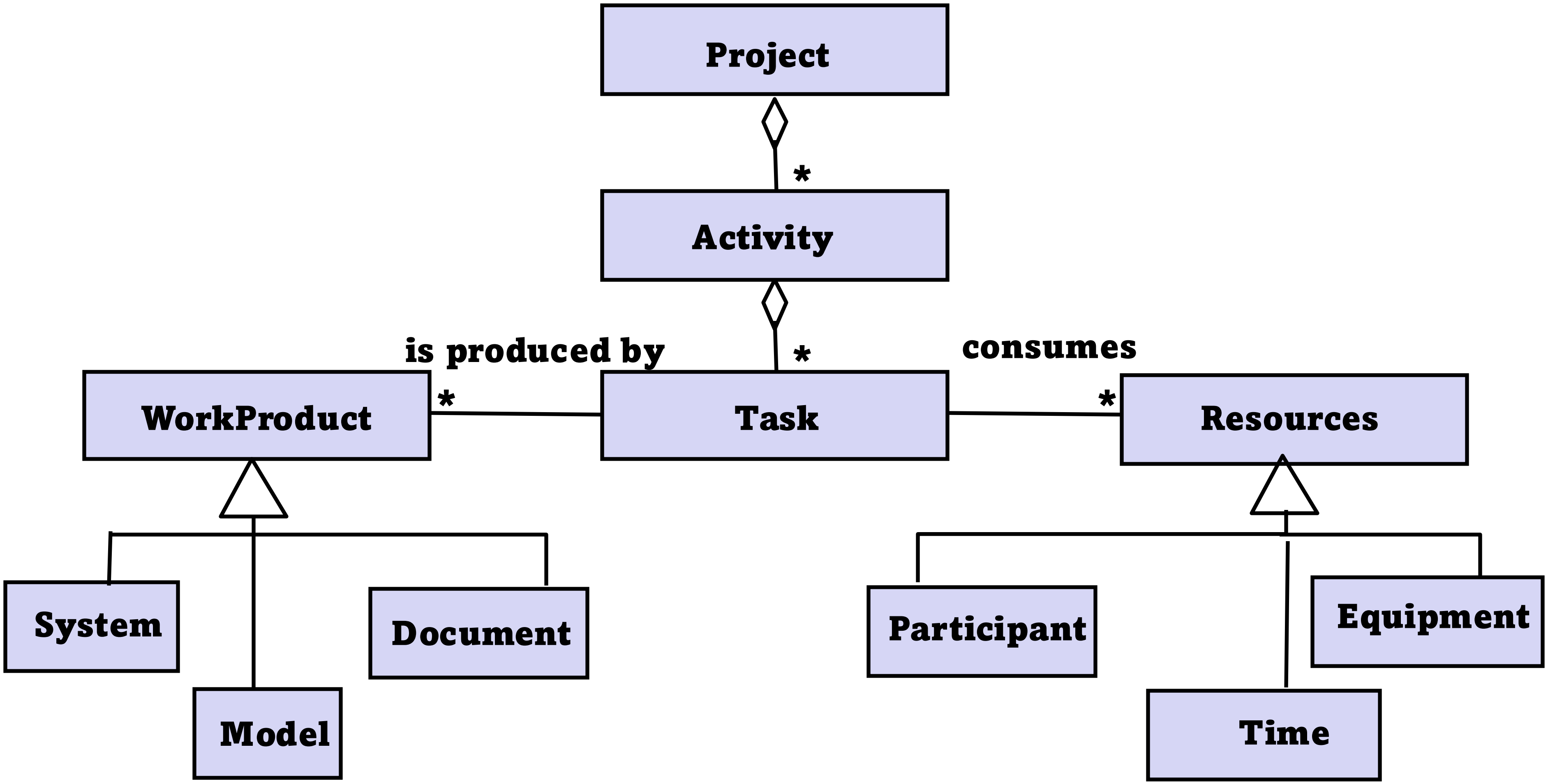
- System – collection of interconnected parts
  - F/E Ticket Distributor system
- Model - abstraction of a system
  - F/E Schematics of electrical wiring, object model
- Document
  - F/E System description

- Work product can be
  - internal - for project consumption
    - F/E test manual, workplan
  - deliverable – delivers to the client
    - F/E specification, operation manual

Software Engineering Concepts (activities and tasks)

- Task – atomic unit of work to be managed
  - consumes Resources
  - produces Work Products
  - depends on other tasks
    - F/E develop "Out of Change" test case
- Activity – set of tasks performed to be managed
  - F/E buy a ticket
- Work on the project is broken into tasks which are assigned to resources

Software Engineering Concepts Hierarchy



Possible SE activities - Simplified view

Problem Domain

Requirements Analysis	What is the problem?
System Design	What is the solution?
Object Design	What is the solution in the context of an existing hardware system?
Implementation	How is the solution constructed?

Implementation Domain

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Software Engineering Development Activities

- ***Analysis*** – concentrates on system requirements – definitions of a system from users' point of view
  - **Requirements elicitation** – determining functionality user needs and a way of its delivering
  - **Requirements analysis** – formalizing determined requirements and ensuring their completeness and consistency
- ***Design*** – constructing the system
  - **System design** – defining a system architecture in terms of design goals and a subsystem decomposition
  - **Object design** – modeling and construction activities related to the solution domain
- ***Implementation*** – translation of the solution domain model into code
- ***Testing*** – the goal is to discover faults in the system to be repaired before the system delivery

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

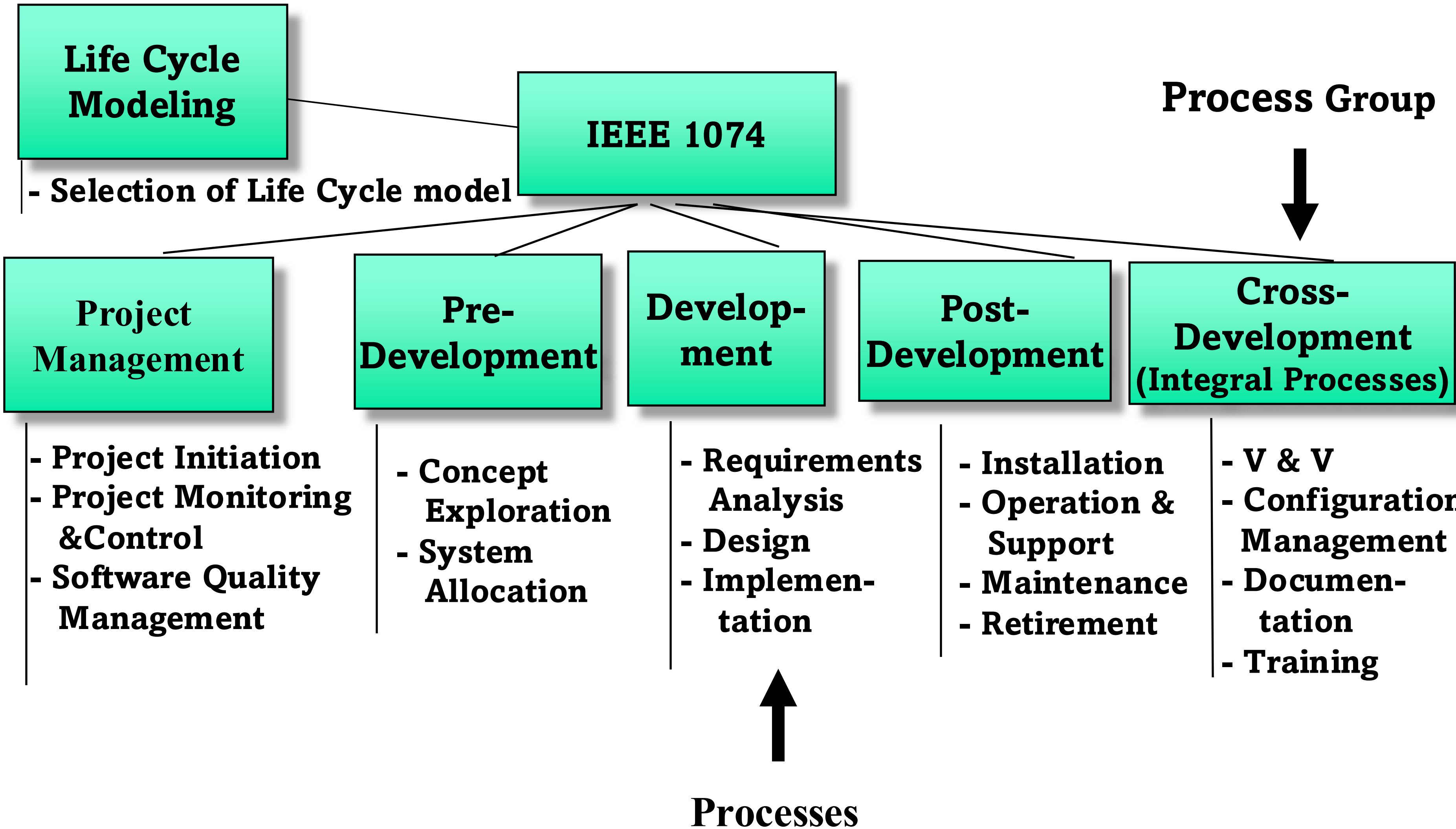
IEEE 1074: Standard for Developing Software Lifecycle Processes

- Describes a set of activities and processes that are **mandatory** for development and maintenance of software
- Establishes a common framework for developing lifecycle models

IEEE 1074: Standard for Developing Software Lifecycle Processes

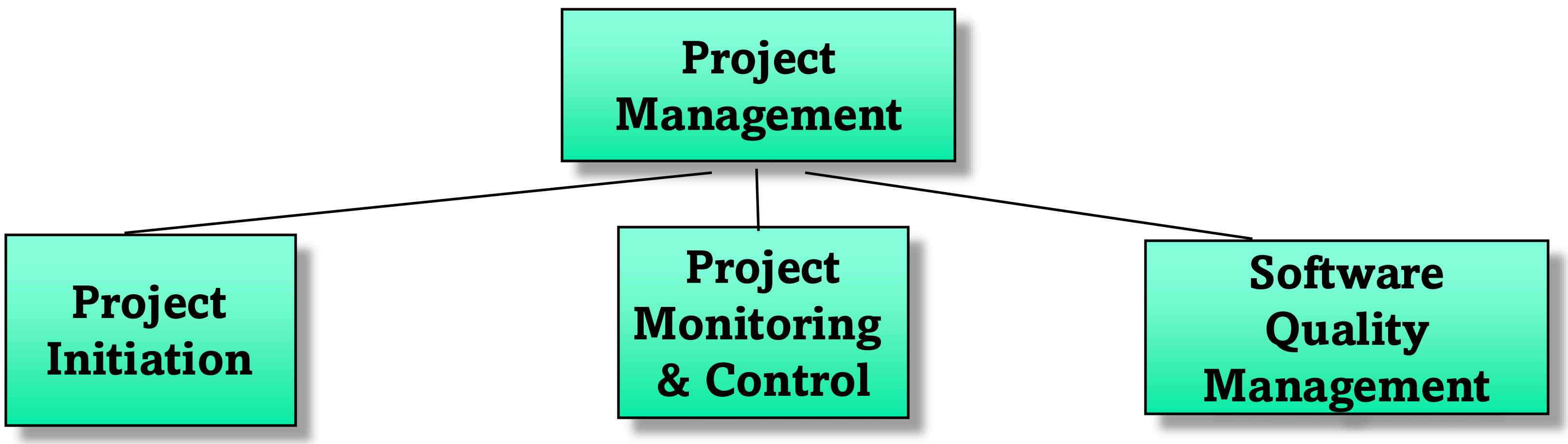
- **Process Group** (F/E Development): Consists of Set of Processes (F/E Design, Implementation)
- **Process** (F/E Design): Consists of Activities (F/E Design database)
- **Activity** (F/E Design Database) : Consists of sub activities and tasks (F/E Make purchase recommendations)

IEEE 1074: Standard for Developing Software Lifecycle Processes



Adopted Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Project Management Process Group



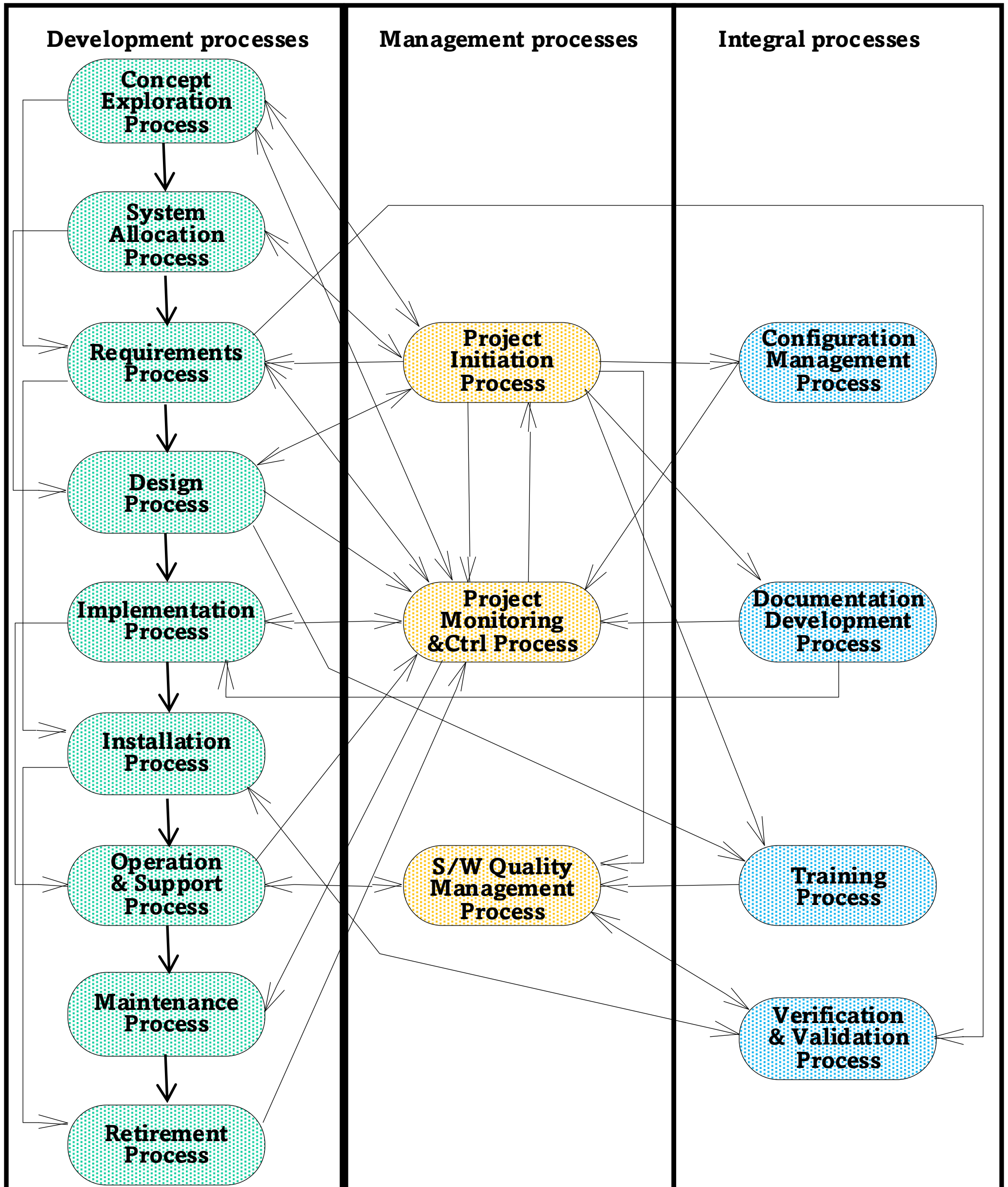
- Map Activities to Software Life Cycle Model
- Allocate Project Resources
- Establish Project Environment
- Plan Project Management

- Analyze Risks
- Perform Contingency Planning
- Manage the Project
- Retain Records
- Implement Problem Reporting Model

- Plan Software Quality Mesurement
- Define Metrics
- Manage Software Quality
- Identify Quality Improvement Needs

Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

IEEE 1074 standard



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java
Adapted from IEEE std. 1074-1997

Capability Maturity Model (CMM)

- **Level 1: Initial**
  - Ad hoc activities applied
  - Success depends on skills of key individuals
  - System is black box for users
  - No interaction with user during the project
- **Level 2: Repeatable**
  - Each project has well-defined software life cycle
  - Models differ from project to project
  - Previous experience in similar projects allows predictability success
  - Interaction with user in defined points
- **Level 3: Defined**
  - Documented software life cycle model is used for all activities
  - Customized version of the model is produced for each project
  - User knows standard model and the model selected for the project
- **Level 4: Managed**
  - Metrics for activities and deliverables defined
  - Collecting data during project duration
  - The software life cycle model can be analyzed
  - User informed about the risks before the project and knows the measures
- **Level 5: Optimized**
  - The measurement data are used as a feedback to improve the software life cycle over lifetime of the organization
  - The user, mangers and developers communicate and work together during the whole project

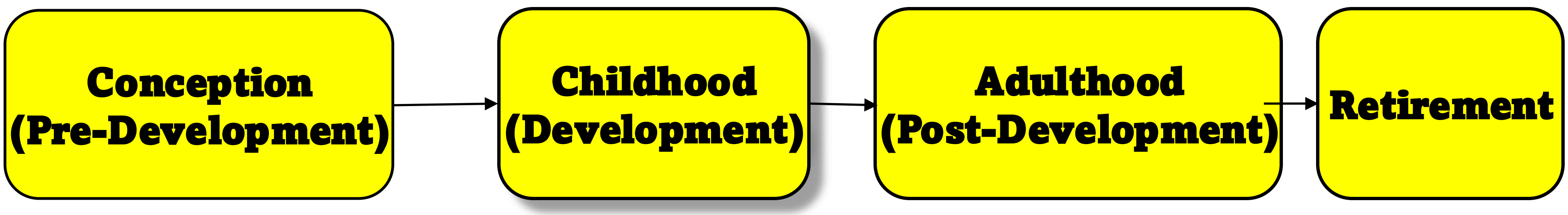
Questions to be asked for a Software Life-Cycle

- Which activities should be selected for the software project?
- What are the dependencies between activities?
  - Does system design depend on analysis? Does analysis depend on design?
- How should the activities be scheduled?
  - Should analysis precede design?
  - Can analysis and design be done in parallel?
  - Should they be done iteratively?

Software Life-cycle Differs Between Projects

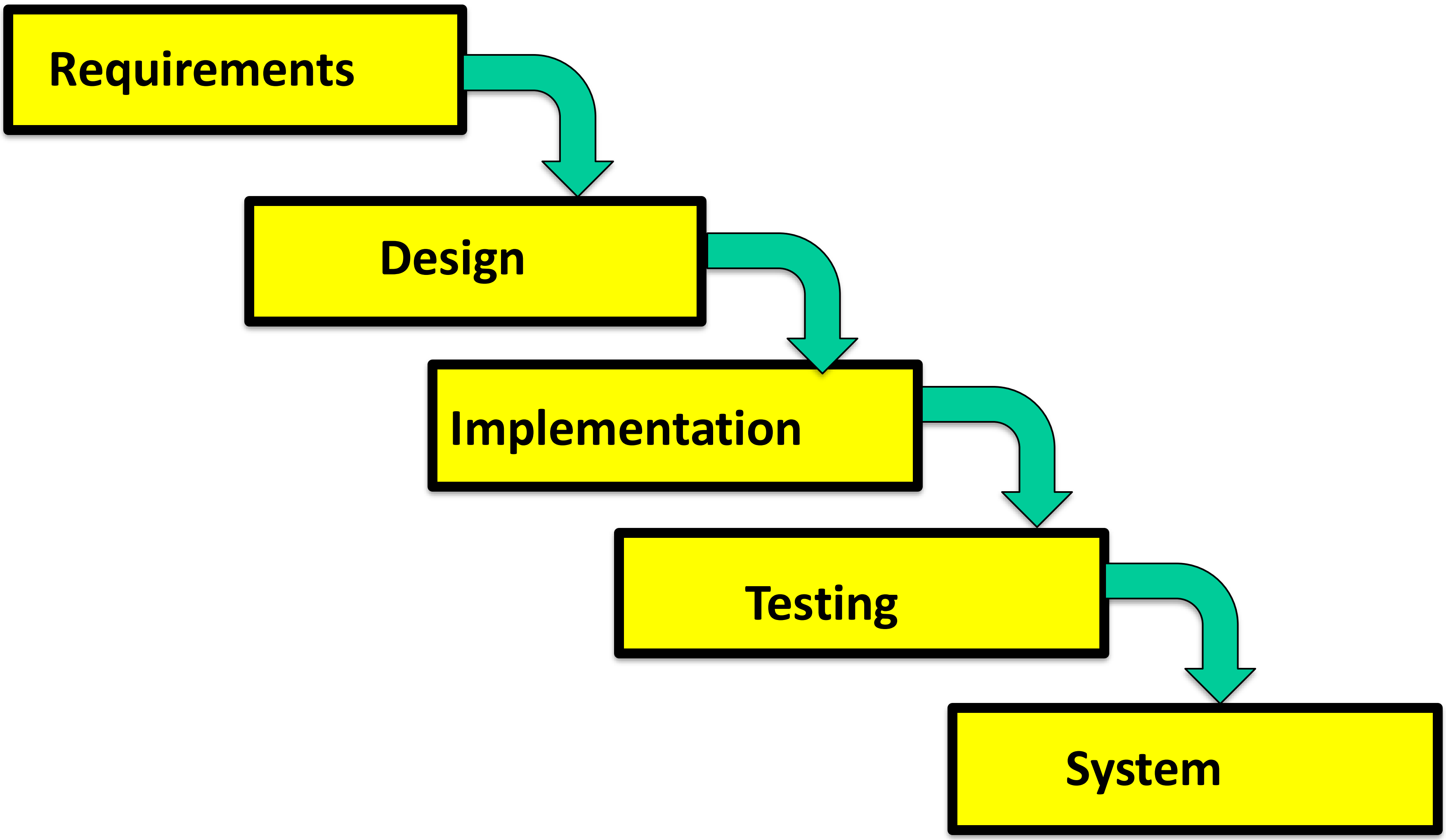
- Different experience and skills
- Different application domains
- Environment changes
- Project size

Stages of a Software Life-Cycle



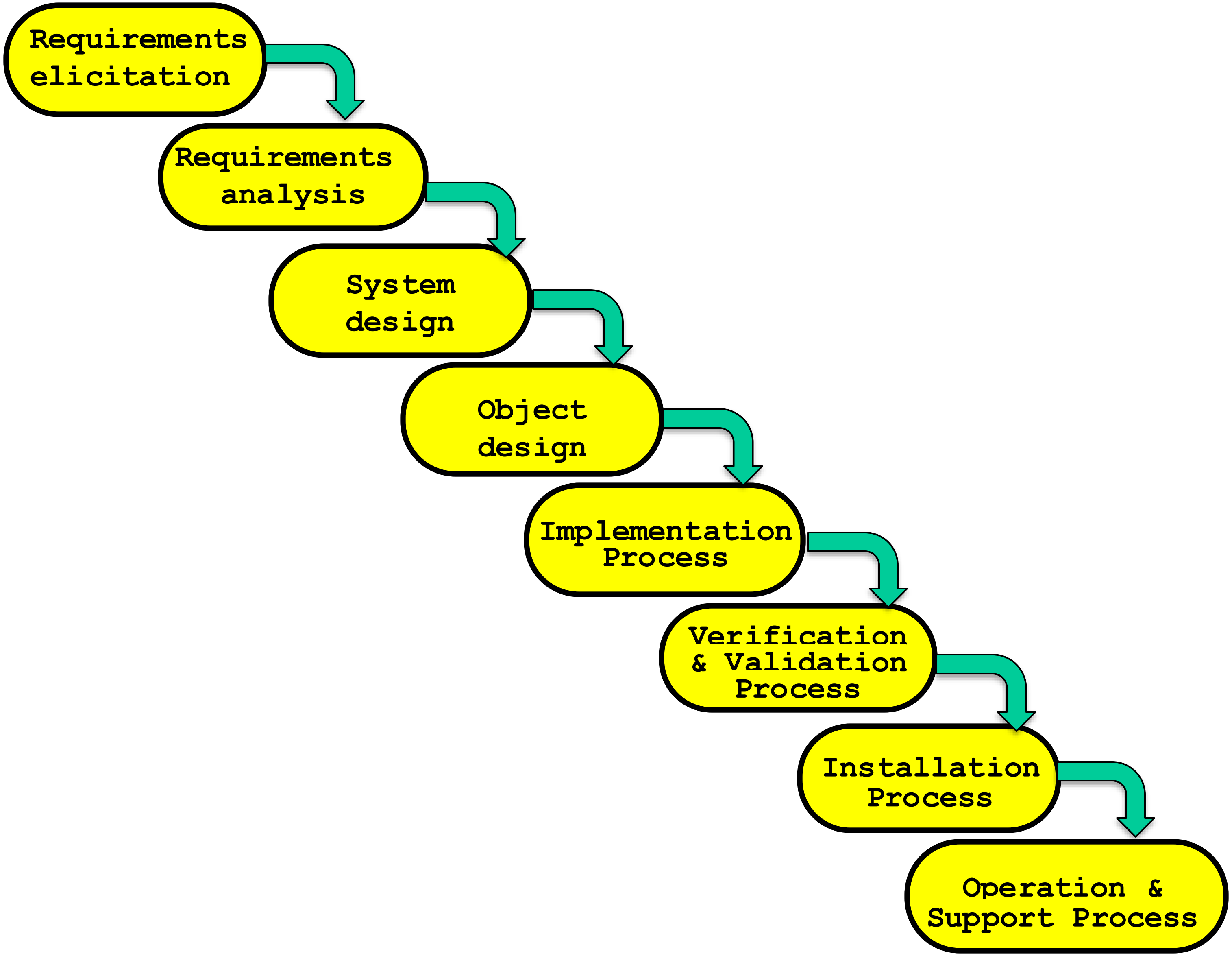
Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

What do we want?



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Waterfall Model

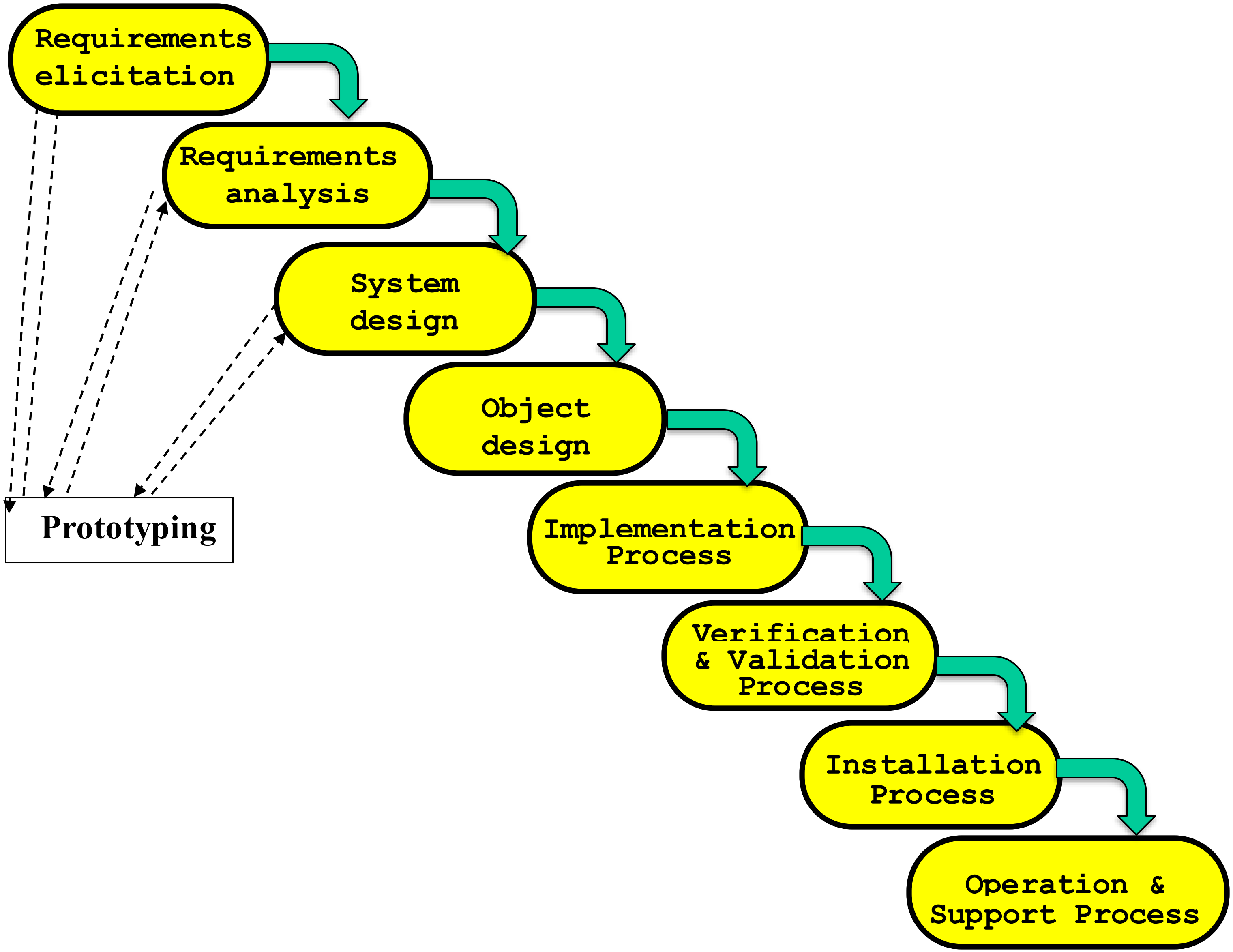


Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java
Adopted from Royce

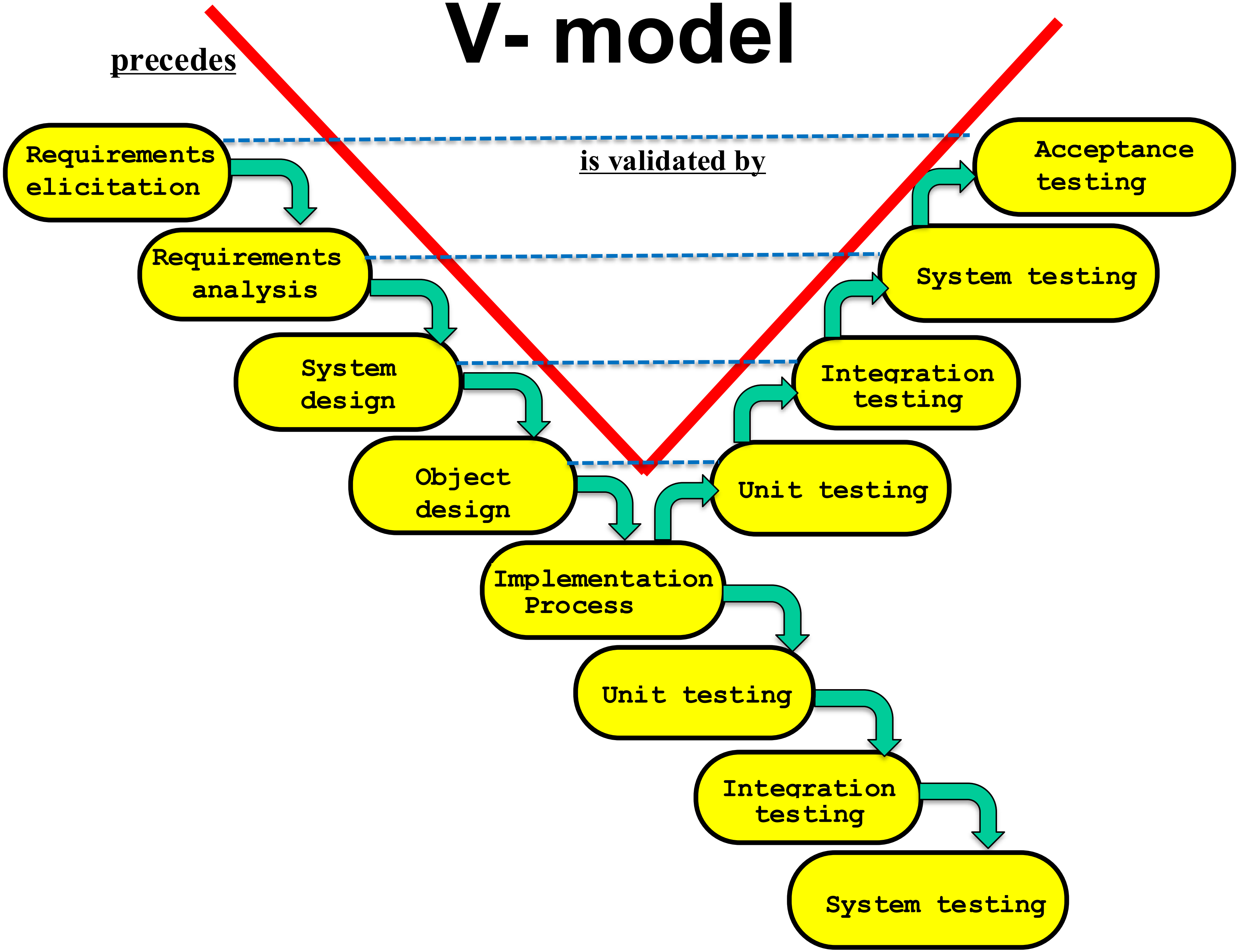
Summarizing Waterfall model

- Managers like waterfall models:
  - Clear milestones
  - No need to look back (linear system), one activity at a time
  - Easy to check progress : 90% coded, 20% tested
- In practice, software development is not sequential
  - The development stages overlap
- Different stakeholders need different abstractions
- System development is a nonlinear activity

Waterfall model with prototyping

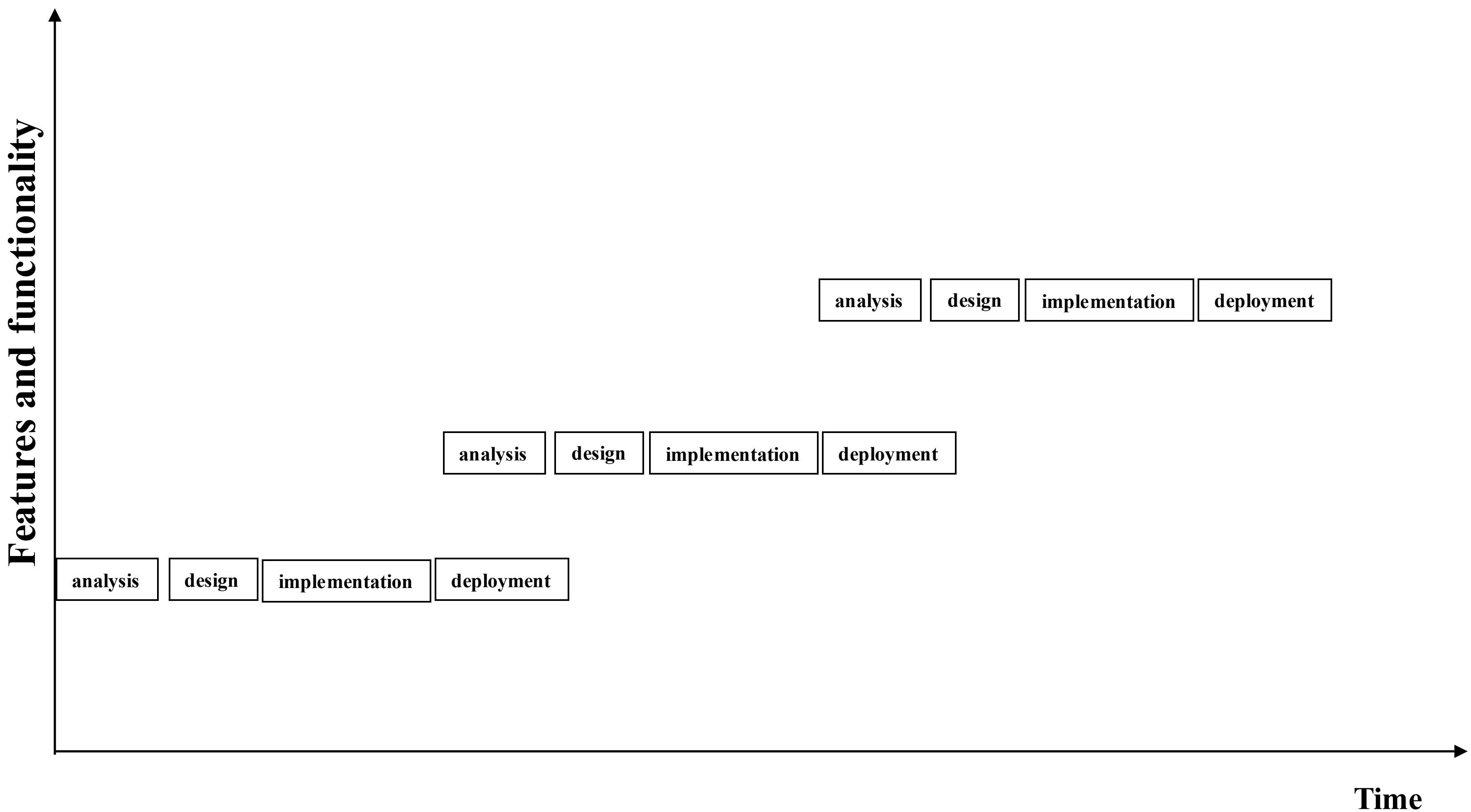


V- model



Adopted from Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java

Incremental model



Iterative models

- Software development is iterative
  - During design problems with requirements are identified
  - During coding, design and requirement problems are found
  - During testing, coding, design & requirement errors are found

  => Spiral Model

Spiral model

- The spiral model proposed by Boehm is an iterative model with the following activities
  - Determine objectives and constraints
  - Evaluate Alternatives
  - Identify risks
  - Resolve risks by assigning priorities to risks
  - Develop a series of prototypes for the identified risks starting with the highest risk.
  - Use a waterfall model for each prototype development ("cycle")
  - If a risk has successfully been resolved, evaluate the results of the "cycle" and plan the next round
  - If a certain risk cannot be resolved, terminate the project immediately

Spiral model

Rounds/cycles

- Concept of Operation,
- Software Requirements,
- Software Product Design,
- Detailed Design,
- Code,
- Unit Test,
- Integration and Test,
- Acceptance

- For each cycle go through these activities
  - Quadrant IV: Define objectives, alternatives, constraints
  - Quadrant I: Evaluate alternative, identify and resolve risks
  - Quadrant II: Develop, verify prototype
  - Quadrant III: Plan next "cycle"

IV Determine objectives

IV Specify constraints

IV Generate alternatives

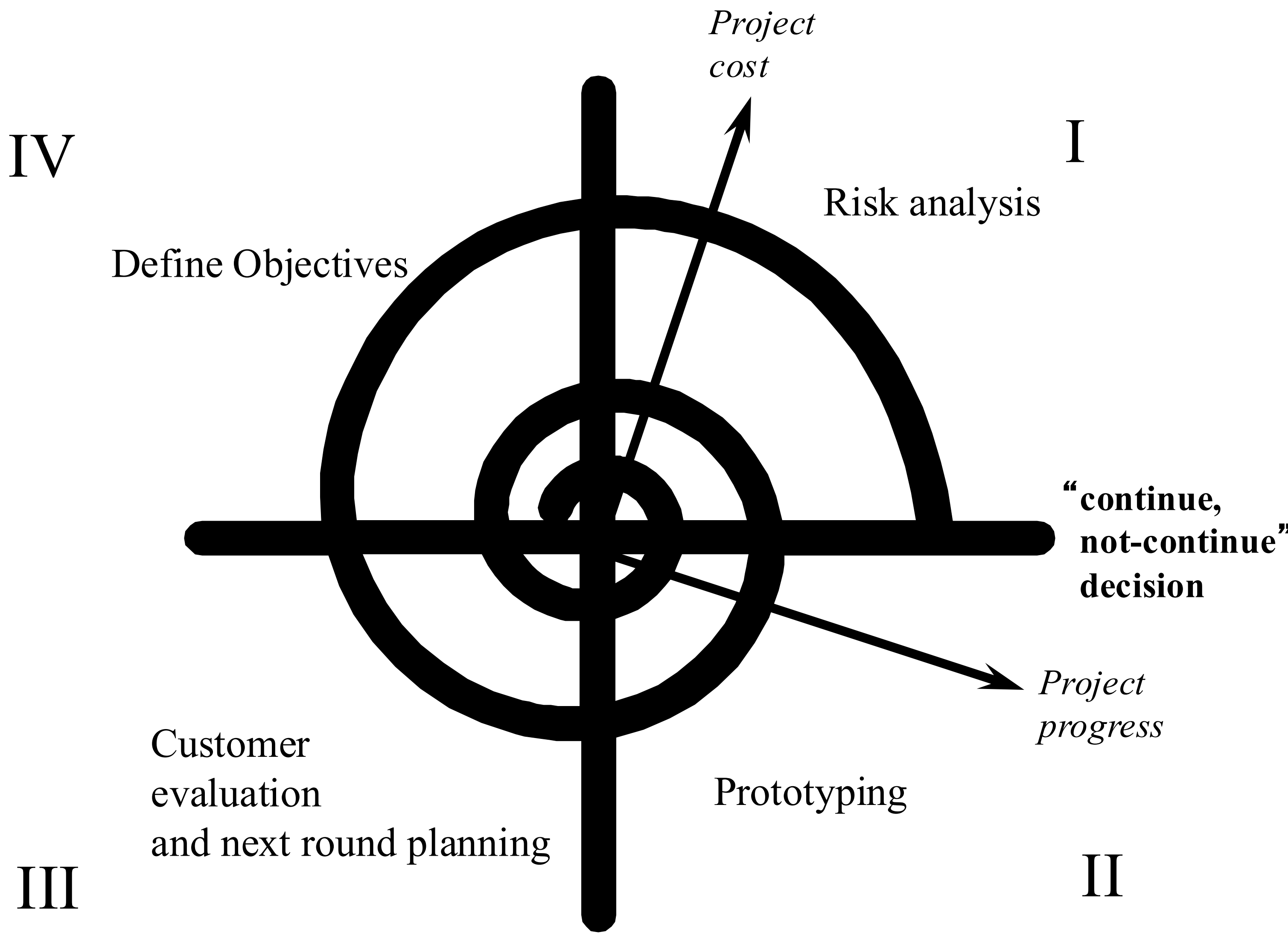
I Identify risks

I Resolve risks

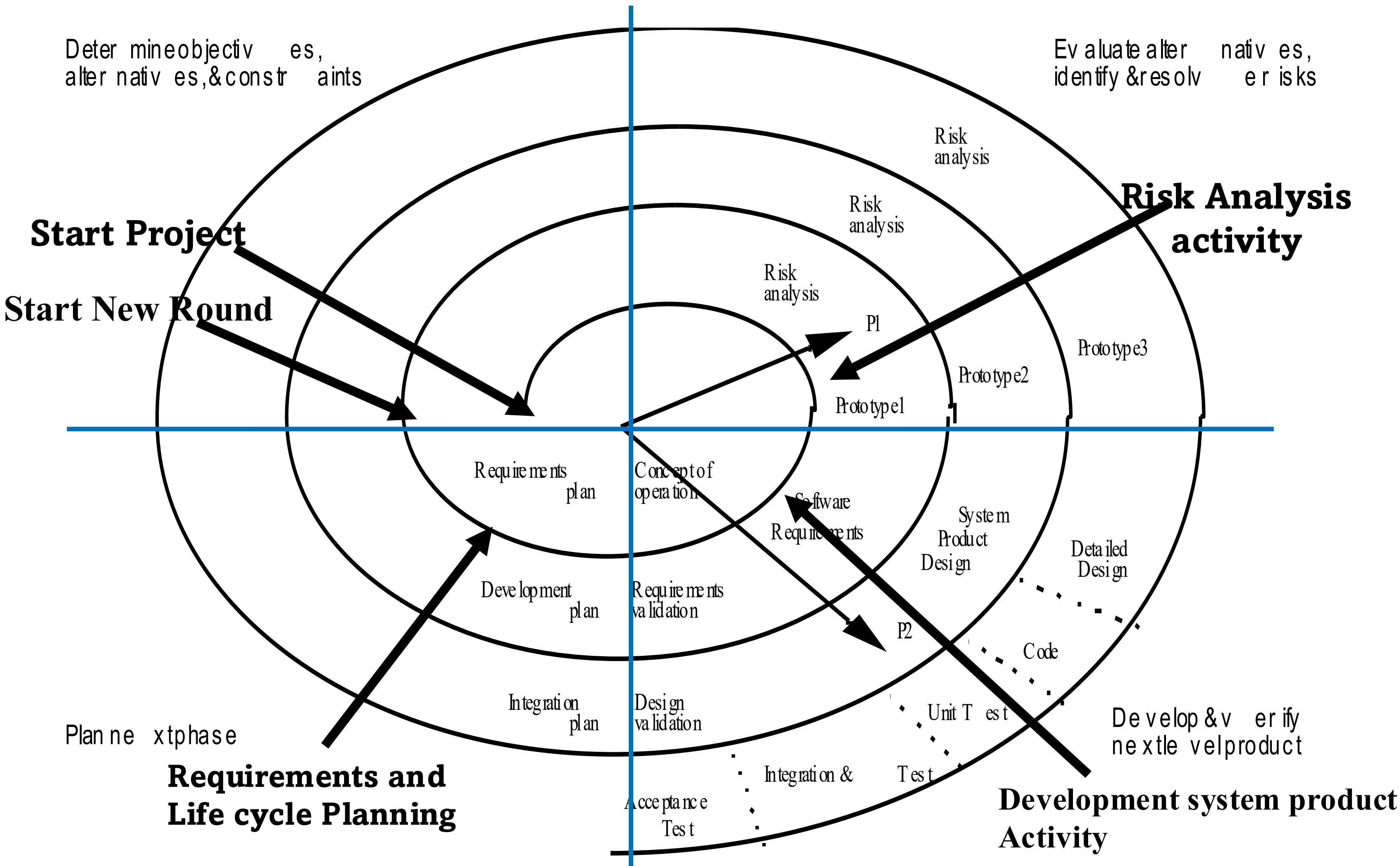
II Develop and verify prototype

III Plan

Spiral model



Spiral Model



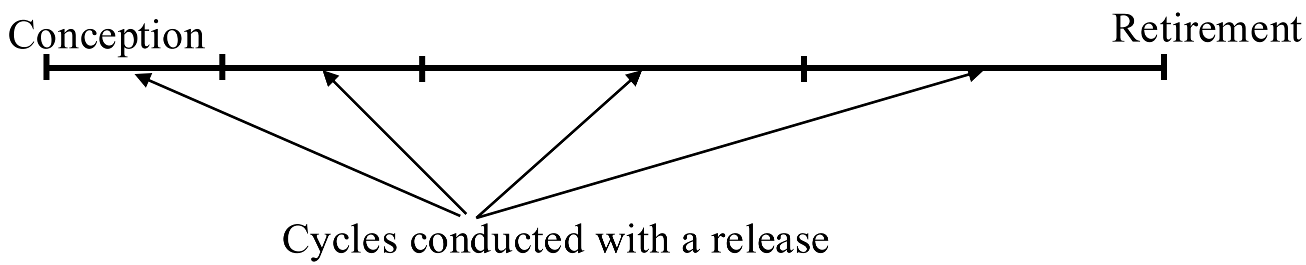
Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java Adopted from Boehm

Limitations of Waterfall and Spiral Models

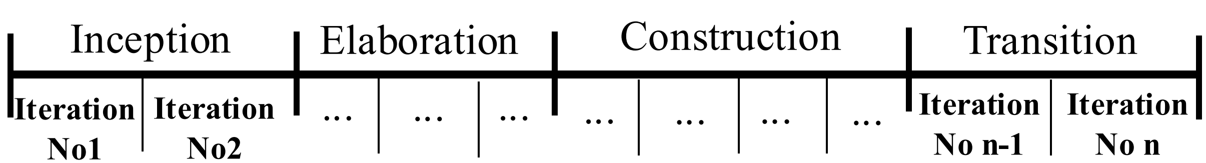
- Neither of these model deals well with frequent change
  - The Waterfall model assume that once you are done with a phase, all issues covered in that phase are closed and cannot be reopened
  - The Spiral model can deal with change between phases, but once inside a phase, no change is allowed

Unified Software Development Process (UP)

- Repeats over a series of cycles (releases)



- Each cycle consists of four phases which are subdivided into iterations



Unified process

- **Inception** – establishes a business case for the system
- **Elaboration** – most of the product cases are specified in details, architecture is designed
- **Construction** – the product is built. The architectural baseline becomes a full-pledged system
- **Transition** – period when product moves to beta release

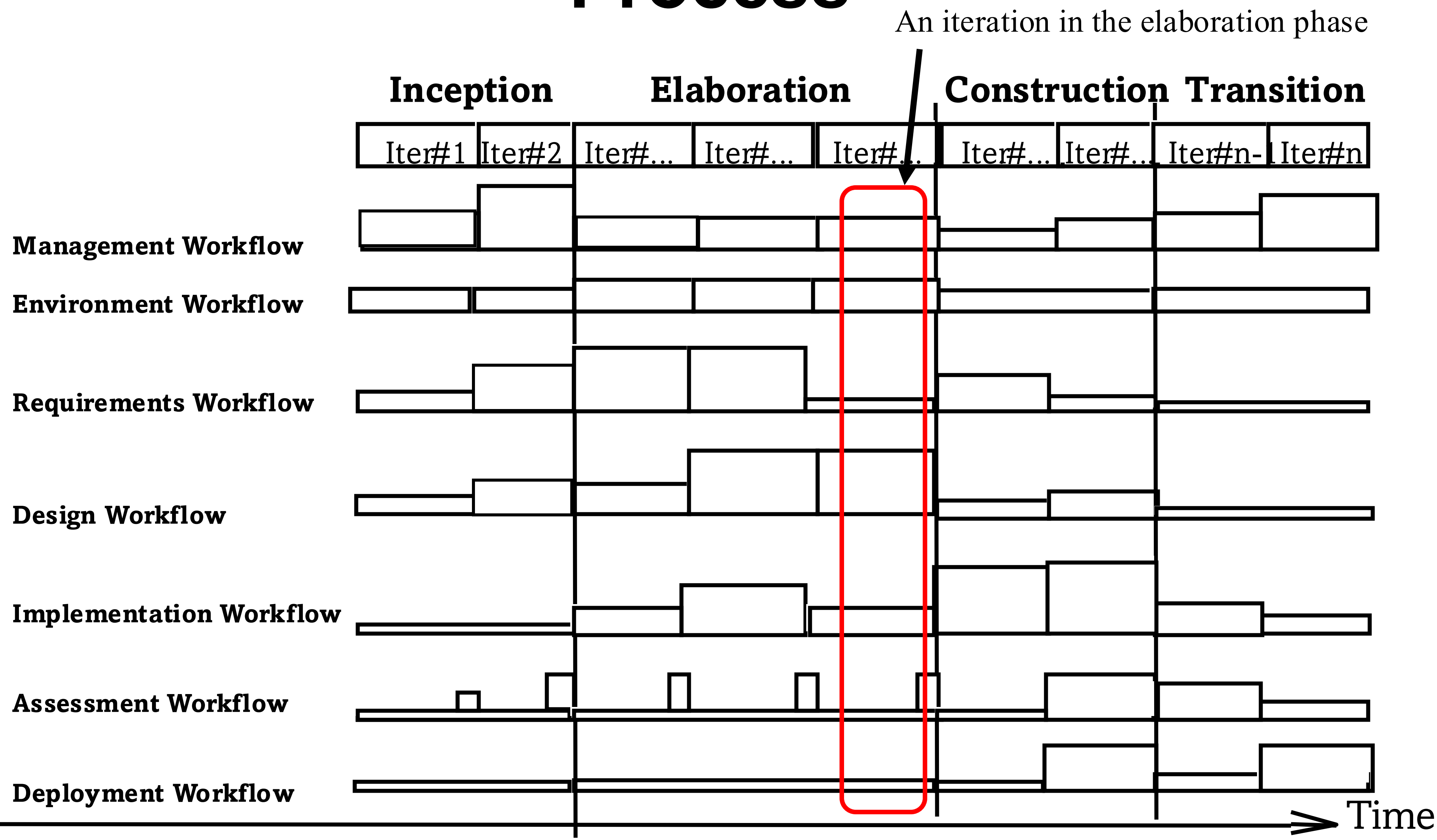
Unified Software Development Process (UP)

- UP organizes projects in two-dimensional terms
- The **horizontal dimension** represents the successive phases of each project iteration:
  - inception,
  - elaboration,
  - construction, and
  - transition.
- The **vertical dimension** represents software development disciplines and supporting activities of configuration and change management, project management, and environment.

Workflows

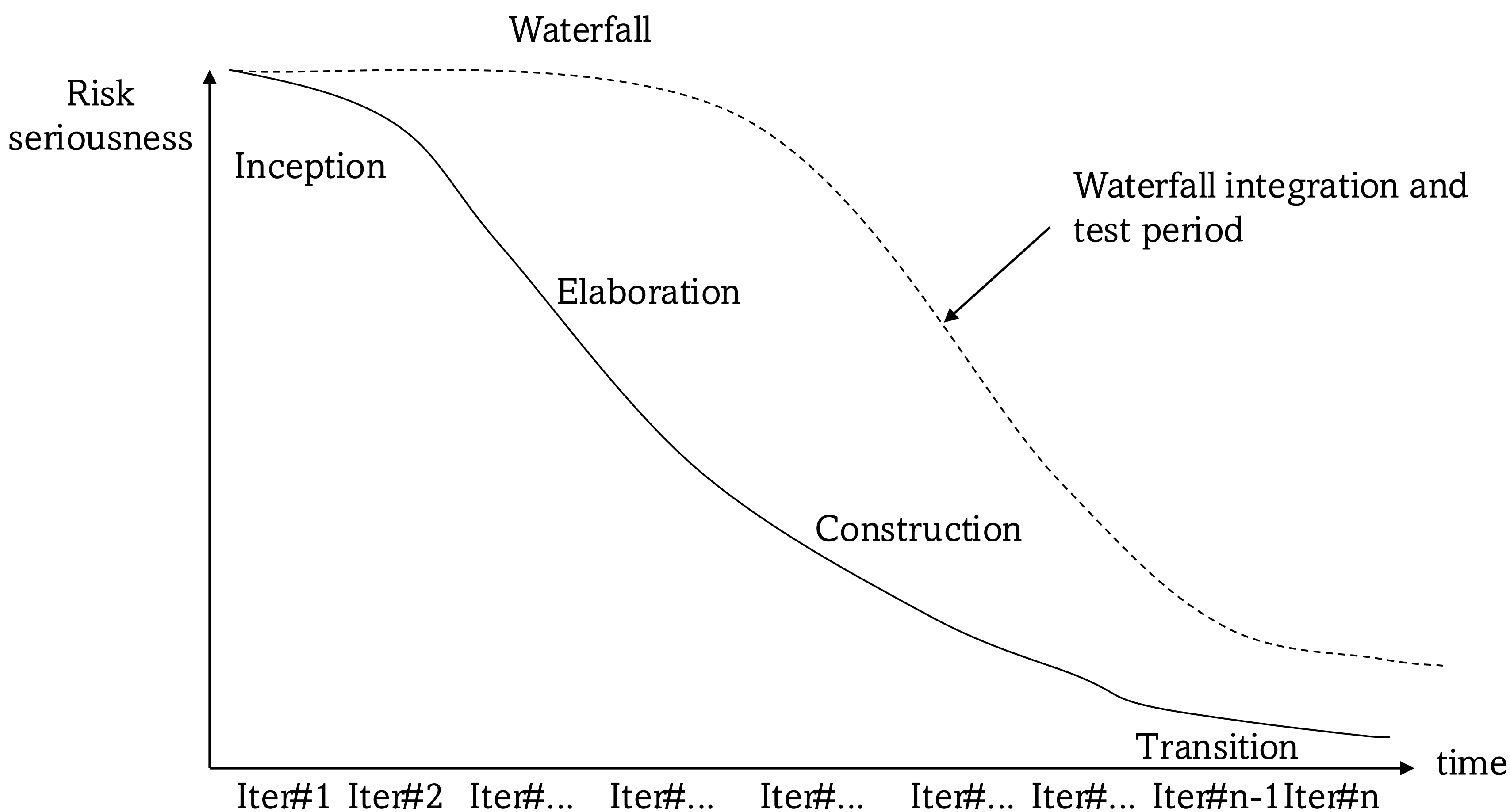
- Cross-functional workflows
  - Management – planning aspects
  - Environment – automation of the process itself
  - Assessment – assesses processes and products needed for reviews
  - Deployment – transition of the system
- Engineering workflows
  - Requirements
  - Design
  - Implementation
  - Testing

Unified Software Development Process



From Bernd Bruegge & Allen H. Dutoit Object-Oriented Software Engineering: Using UML, Patterns, and Java adopted from Jacobson and Royce

UP vs. Waterfall



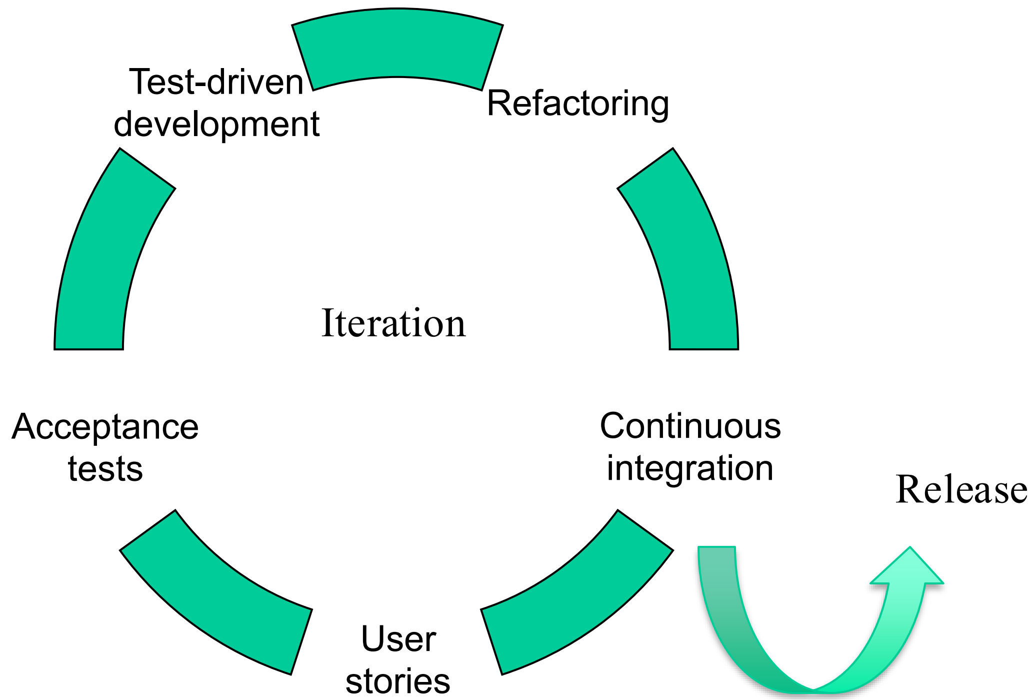
Agile software development

Key points of agility in software production:

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

Adopted from: http://agilemanifesto.org/

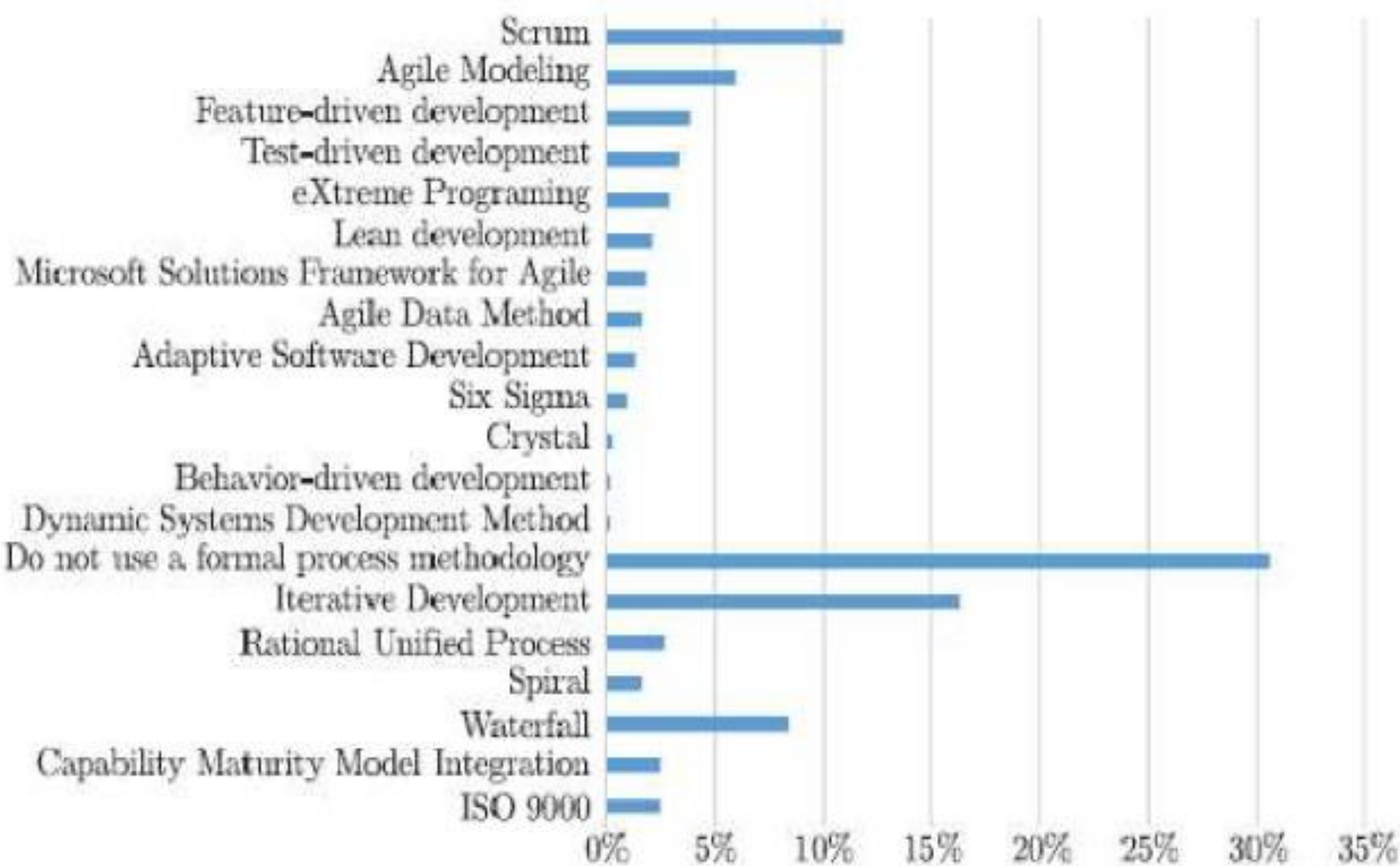
Example Agile Development



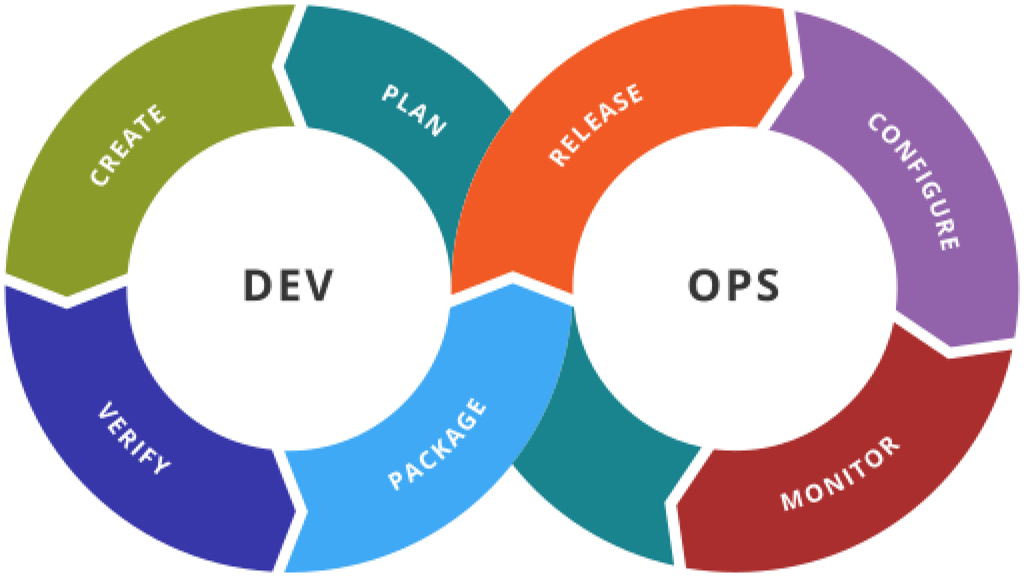
Adopted from XPExplored, by Wake

Forrester survey results

(Open Source Initiative 2015, Available: http://opensource.org/ (online))



DevOps (“DEVelopment” and “OPeration”)



DevOps broadens the scope of ALM to include business owners, customers, and operations as part of the process

Adopted from Wikipedia

Formal software development models

- **inductive methods,**

  where a program is built on the basis of input-output pairs or examples of computations

- **deductive methods**,

  which uses automatic deduction of a proof of solvability of a problem and derives a program from the proof

- **transformational methods**,

  where a program is derived stepwise from a specification by means of transformations

Inductive synthesis - Basic notions (Program specification)

- The expression "programming by examples" is taken to mean the act of specifying a program by means of examples of the program behavior
- Examples consisting of expressions representing an input and output
  - **(A B C)** $\rightarrow$ **C**
  - **(A B)** $\rightarrow$ **B,**
    (A B C) $\rightarrow$ **B,**
    (A B C D) $\rightarrow$ **B**

Program specification

- A single example indicates a specific transformation or computation that must be duplicated by any program defined by the examples
- For any finite set of examples there are an infinite number of programs that behave like the examples.
- Program satisfies or is defined by a set of examples if it produces the examples output when applied to the examples input for all examples in the specifying set.

General methods (search)

- A fundamental method of inductive inference is to search in some systematic way through the space of possible programs (rules)
- Possible program should be agreed with all examples

General methods (search)

- Basic advantage of search is that it is a general method and does not depend much on domain-specific knowledge

- Basic disadvantage of search is that in a general case it can be very inefficient

Machine learning for program synthesis

- Based on great achievements in ML

- Problem Definition (Program synthesis)
  - *Let* L *be the space of all valid programs in the domain-specific language (DSL). Given a training dataset of* {IOK} *for* i = 1,...,K*, where* K *is the size of the training data, compute a* synthesizer Γ*, so that given a test input-output example set* {IOK} test*, the synthesizer* Γ({IOK} test) = P *produces a program* P *, which emulates the program corresponding to* {IOK} test*.*

Some examples of problems FlashFill

Input String	Output String
jacob daniel devlin	Devlin, J.
jonathan uesato	Useato, J
Surya Bhupatiraju	Bhupatiraju S.
Rishabh q. singh	Singh, R.
abdelrahman mohamed	Mohamed, A.
pushmeet kohli	Kohli, P.

Deductive Synthesis

It is based on the observation that constructive proofs are equivalent to programs because each step of a constructive proof can be interpreted as a step of computation.

The way from a problem to program

Problem in source form

↓

Problem represented in a formal theory

↓

Proof of solvability of the problem

↓

Program

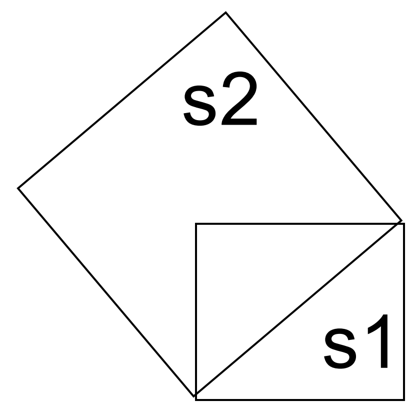
Problem Solving in the SSP Systems (Geometry)

SQUARE

var side, square, diagonal:numeric;

rel area= side^2;

diagonal^2 = 2*side^2;



Figure

var

s1:SQUARE;

s2:SQUARE side = s1.diagonal;

Set of solvable problems:

s1.side $\rightarrow$ s1, s2;   s2.side $\rightarrow$ s1, s2;

s1.diagonal $\rightarrow$ s1, s2; s1.area$\rightarrow$ s1, s2; s2.diagonal$\rightarrow$ s1, s2;

. . .

Deductive synthesis problem formulation

- suppose that program to be derived takes an input *x* and produces an output *y*
- suppose that specification of the program states that a precondition *P(x)* should be true of the input and that a post-condition *Q(x,y)* should be true of the output
- then the theorem to be proven is

$$\forall x \, \exists y [P(x) \rightarrow Q(x,y)]$$

Structural synthesis of programs

The idea of Structural Synthesis of Programs (SSP) is that a proof of solvability of a problem can be built and that the overall structure of a problem can be derived from the proof, knowing very little about properties of the functions used in the program

Logical language of SSP (*LL*-language)

- Propositional variables (begin with capital letters): A, B, C, D, A_i, A^i ....
- Unconditional computability statements:

  $$A_1 \ \& \ ... \ \& \ A_k \rightarrow B$$

  we also use $\underline{A}$ as an abbreviation for $A_1 \ \& \ ... \ \& \ A_k$
- Conditional computability statements

  $(\underline{A}^1 \rightarrow B^1) \ \& \ ... \ \& \ (\underline{A}^n \rightarrow B^n) \rightarrow (\underline{C} \rightarrow D)$

  we also use $\underline{(A \rightarrow B)}$ as an abbreviation for

  $$(A^1 \rightarrow B^1) \ \& \ ... \ \& \ (A^n \rightarrow B^n).$$

Informal meaning

$A_1 \& \ldots \& A_k \to B$ has two meanings

- Logical meaning "$A_1,\ldots, A_k$ implies B", where $A_1,\ldots, A_k, B$ - propositional variables
- Computational meaning "B is computable from $A_1,\ldots, A_k$"

  $A_1,\ldots, A_k, B$ - computational objects

Computational meaning of the LL formulae

- Propositional variables of *LL* correspond to object variables from the source problem description language (specification language).

- An unconditional computability statement $\underline{A} \to B$ expresses a computability of the value of the object variable b corresponding to B from values of $a_1 \ldots a_k$ corresponding to $\underline{A}$.

- A conditional computability statement, e.g.

  $(A \to B) \to (C \to D)$ expresses computability of the object variable d from c depending on other computations, here depending on the computation of the object variable b from a.

Structural synthesis rules (SSR)

- $$\frac{\Sigma \vdash \underline{A} \rightarrow V \qquad \underline{\Gamma \vdash A}}{\Sigma, \underline{\Gamma} \vdash V} \ (\rightarrow -)$$

  where $\underline{\Gamma \vdash A}$ is a set of sequents for all A in $\underline{A}$

- $$\frac{\Gamma, \underline{A} \vdash B}{\Gamma \vdash \underline{A} \rightarrow B} \ (\rightarrow +)$$

- $$\frac{\Sigma \vdash \underline{(\underline{A} \rightarrow B)} \rightarrow (\underline{C} \rightarrow V); \qquad \underline{\Gamma, \underline{A} \vdash B}; \qquad \underline{\Delta \vdash C}}{\Sigma, \underline{\Gamma}, \underline{\Delta} \vdash V} \ (\rightarrow --)$$

where $\underline{\Gamma, \underline{A} \vdash B}$ is a set of sequents for all $\underline{A} \rightarrow B$ in $\underline{(\underline{A} \rightarrow B)}$, and $\underline{\Delta \vdash C}$ is a set of sequents for all C in $\underline{C}$.

SSR1 inference rules

$$\frac{\Sigma \ \vdash \underline{A} \rightarrow_{f} V \ ; \quad \underline{\Gamma \ \vdash A(a)}}{\Sigma, \underline{\Gamma} \vdash V(f(\underline{a}))} \ (\rightarrow -)$$

$$\frac{\Gamma, \underline{A} \ \vdash B(b)}{\Gamma \vdash \ \underline{A} \rightarrow_{\lambda\, \underline{a}.b} \ B} \ (\rightarrow +)$$

$$\frac{\Sigma \vdash \underline{(\underline{A} \rightarrow_{g} B)} \rightarrow (\underline{C} \rightarrow_{F} V); \ \underline{\Gamma, \underline{A} \vdash B(b)}; \ \underline{\Delta \vdash C(c)}}{\Sigma, \underline{\Gamma}, \underline{\Delta} \ \vdash \underline{V(F(}\lambda\, \underline{a}.b, \underline{c}))} \ (\rightarrow - -)$$

Example. Modeling logical circuits

Class **inverter**

```
inverter: (in,out: bool;
            finv: in → out (finv))
```

Corresponding set of formulae

$INVERTER.IN \rightarrow_{finv} INVERTER.OUT$,

$INVERTER.IN \; \& \; INVERTER.OUT \rightarrow_{constr(2)} INVERTER$,

$INVERTER \rightarrow_{select_1} INVERTER.IN$,

$INVERTER \rightarrow_{select2} INVERTER.OUT$

Example. Modeling logical circuits

- Class **and**

```
and:(in1, in2, out:bool;
          fand:in1, in2 → out(fand))
```

Corresponding set of formulae

$AND.IN1\ \&\ AND.IN2 \rightarrow_{fand} AND.OUT$,

$AND.IN1\ \&\ AND.IN2\ \&\ AND.OUT \rightarrow_{constr_3} AND$,

$AND \rightarrow_{select_1} AND.IN1$, $AND \rightarrow_{select_2} AND.IN2$,

$AND \rightarrow_{select_3} AND.OUT$

Example. Modeling logical circuits

- Class **nand**

```
nand: (in1,in2,out: bool;
              a: and in1 = in1, in2 = in2;
              i: inverter in = a.out, out=out;
              fnand: in1,in2 → out);
```

Corresponding set of formulae

$IN1 \rightarrow_{eq1} A.IN1$, $A.IN1 \rightarrow_{eq2} IN1$, $IN2 \rightarrow_{eq3} A.IN2$,

$A.IN2 \rightarrow_{eq4} IN2$, $I.IN \rightarrow_{eq5} A.OUT$, $A.OUT \rightarrow_{eq6} I.IN$,

$I.OUT \rightarrow_{eq7} OUT$, $OUT \rightarrow_{eq8} I.OUT$,

$A.IN1 \,\&\, A.IN2 \rightarrow_{a.fand} A.OUT$, $I.IN \rightarrow_{i.finv} I.OUT$,

$A \rightarrow_{select_1} A.IN1$, $A \rightarrow_{select_2} A.IN2$, $A \rightarrow_{select_3} A.OUT$,

$IN1 \,\&\, IN2 \,\&\, OUT \rightarrow_{constr_3} A$, $I.IN \,\&\, I.OUT \rightarrow_{constr_2} I$,

$I \rightarrow_{select_1} I.IN$, $I \rightarrow_{select_2} I.OUT$, $IN1 \,\&\, IN2 \rightarrow OUT$

Example. Modeling logical circuits (inference)

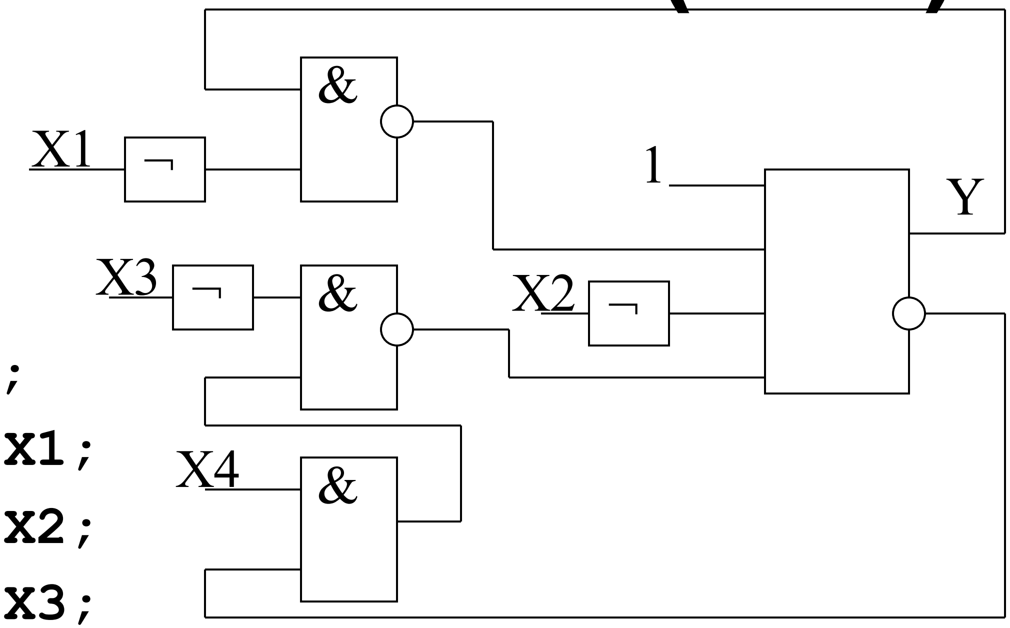
$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{\text{IN1}}{\text{A.IN1}}\,\text{eq1} \qquad \dfrac{\text{IN2}}{\text{A.IN2}}\,\text{eq3}}{\text{A.OUT}}\,\text{a.fand}}{\text{I.IN}}\,\text{eq6}}{\text{I.OUT}}\,\text{i.inv}}{\text{OUT}}\,\text{eq7}$$

Extracted program

fnand: out=eq7(i.finv(eq6(a.fand(eq1(in1), eq3(in2)))))

or fnand: λin1 in2. (i.finv(a.fand(in1, in2)))

Example - Logical circuits (new)



Class **SCHEMA**

```
var  X1, X2, X3, X4: any;
      i1: Inverter in = X1;
      i2: Inverter in = X2;
      i3: Inverter in = X3;
       a1: NAND in1 = t1.Q1, in2 = i1.out;
     a2: NAND in1 = i3.out, in2 = a3.out;
     a3: AND in1 = X4, in2 = t1.Q2;
     t1: D_latch S=1,D=i2.out, C=a2.out,R=a1.out;
alias Y = t1.Q1;
rel problem: X1, X2, X3, X4 -> Y {spec};
```

Problem Solving in the SSP Systems (Geometry)

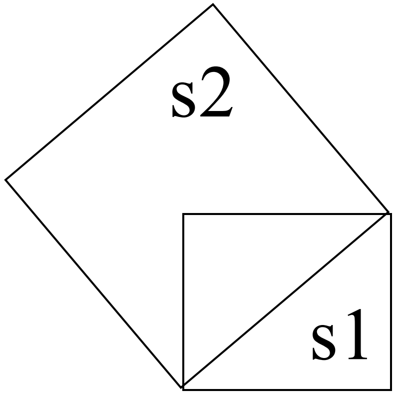
SQUARE

```
   var side, square, diagonal:numeric;
   rel eq1:area= side^2;
        eq2:diagonal^2 = 2*side^2;
```

AREA →eq1 SIDE; SIDE→eq1 AREA;

DIAGONAL→eq2 SIDE; SIDE→eq2 DIAGONAL;

…



Figure

```
   var
       s1:SQUARE;
       s2:SQUARE side = s1.diagonal;
```

S1.AREA →eq1 S1.SIDE; S1.SIDE→eq1 S1.AREA;

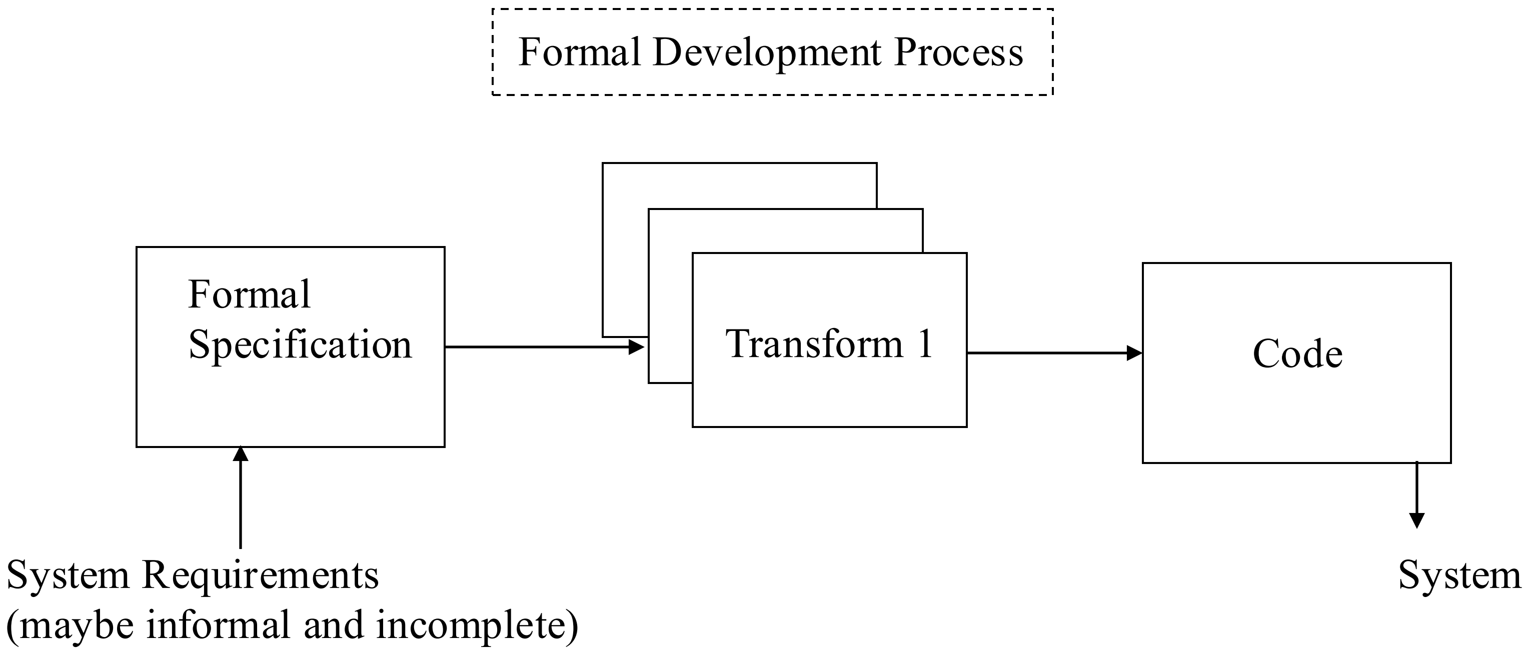
S1.DIAGONAL→eq2 S1.SIDE; S1.SIDE→eq2 S1.DIAGONAL;

S2.AREA →eq1 S2.SIDE; S2.SIDE→eq1 S2.AREA;

S2.DIAGONAL→eq2 S2.SIDE; S2.SIDE→eq2 S2.DIAGONAL;

S2.SIDE →eq S1.DIAGONAL; S1.DIAGONAL→eq S2.SIDE ;

Transformational model



Adopted fromPfleger&Atlee

Deductive and Transformational Synthesis

- Both of them are based on deduction, however, they use different deduction methods.
- Synthesis is usually based on **inference**, whereas transformation is usually based on **replacement**.
- **Inference** means the axioms and rules supplied to the program derivation system are expressed as implications, and the logical operations of the system derive either necessary conclusions or premises of sufficient antecedents of goals.
- **Replacement** means that the axioms supplied to the system are expressed as equations or rewrite rules, and the logical operations replace expressions by other equivalent expressions.

Correctness of transformations

- Let SP0 be a specification of the requirements which the software system is expected to fulfill, expressed in some formal specification language SL.
- The ultimate objective is a program P written in some programming language PL which satisfies the requirements in SP0.
- The main idea is to develop P from SP0 via series of small refinement steps

  $$SP0 \rightarrow SP1 \rightarrow ... \rightarrow P$$

- If each individual refinement step (SP0 $\rightarrow$ SP1, SP1 $\rightarrow$ SP2,...) can be proved to be correct then the resulting program P is guaranteed to be correct.

Correctness of transformations

The notions of the "correctness of transformations" have to be based on suitable relations between programs. Given such a relation φ , the transition from a program P to another program P' then is said to be correct iff P φ P' holds.

- Some relations which are reasonable in connection with program transformations:
  - P and P' have to be "equivalent"
  - Weakening condition for “equivalence”: the equivalence of P and P' is only required if P is defined
  - For nondeterministic programs, it may suffice that P' is included in P, - that the possible results of P' form a subset of possible results of P.

Correctness of transformations

Relation φ has to fulfill the following requirements:

- it has to be reflexive, since the identity should be a valid transformation;
- it has to be transitive in order to allow the successive application of several transformations
- if there is the local application of transformation to a (small) part Q of a program P, then the validity of $Q \varphi Q'$ only implies the validity of $P \varphi P'$, if the relation is monotonic for the constructs of the language (where $P' = P[Q'/Q]$).

Correctness of transformations

The formalization of relations has to refer to the actual semantic definition of the programming language under consideration (for example):

- In denotational semantics the meaning of program is specified with help of a function M mapping programs to semantic object.

  Two programs P1 and P2 then can be defined to be equivalent iff M(P1) = M(P2).

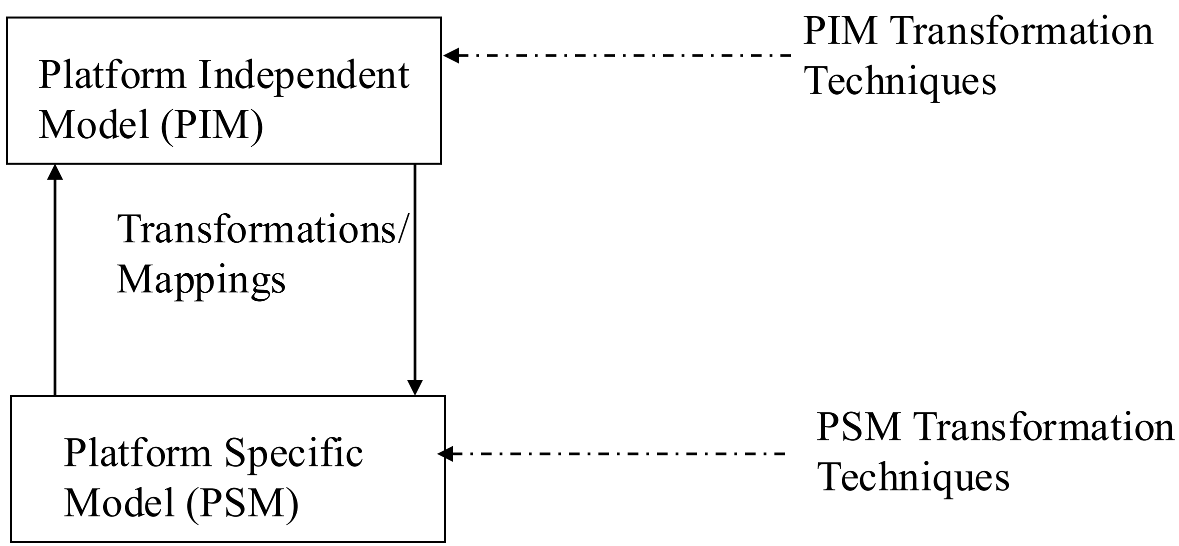
- In operational semantics two programs may be regarded equivalent iff they lead to the same sequences of "elementary" actions.

Model-driven architecture (MDA)

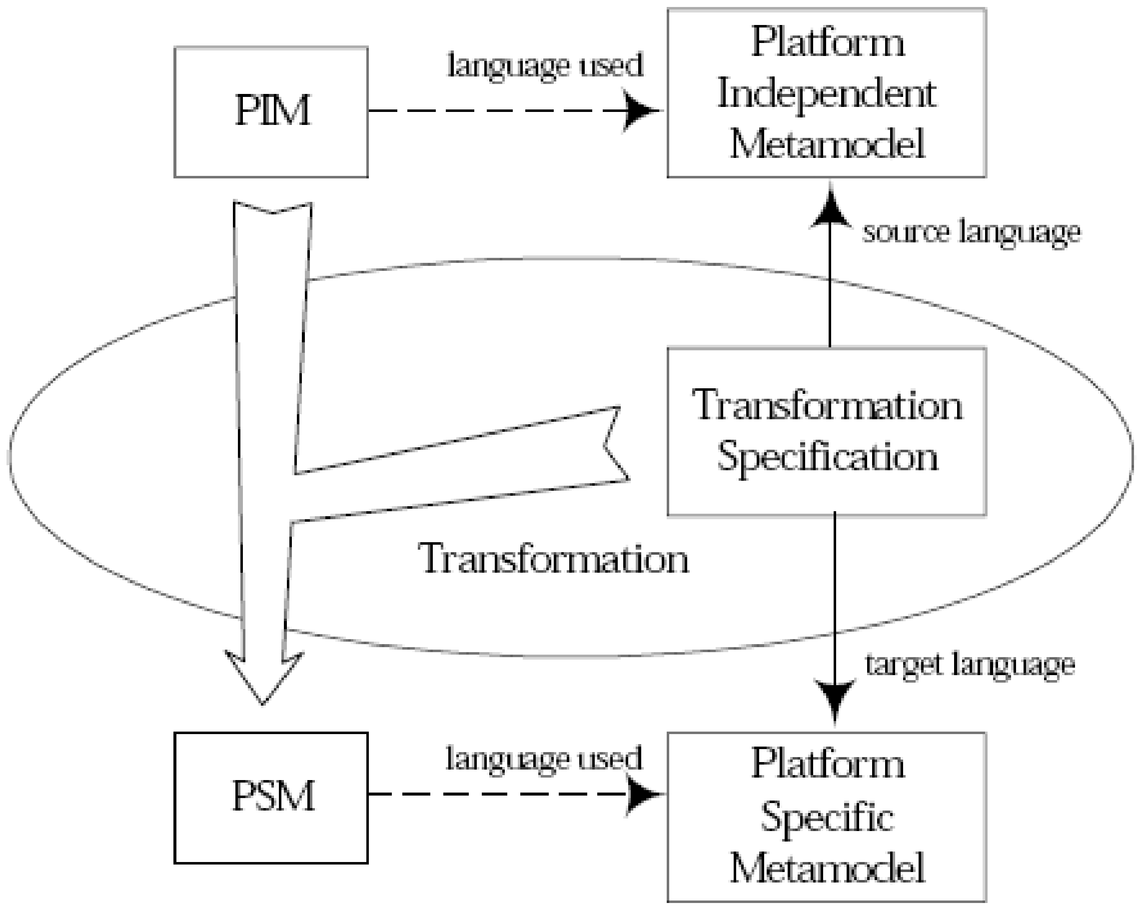
- a software design approach for the development of software systems.
- provides a set of guidelines for the structuring of specifications, which are expressed as models.
- a kind of domain engineering, and supports model-driven engineering of software systems.
- launched by the Object Management Group (OMG) in 2001

Adopted from Alan Brown

Model-Driven Architecture (MDA)

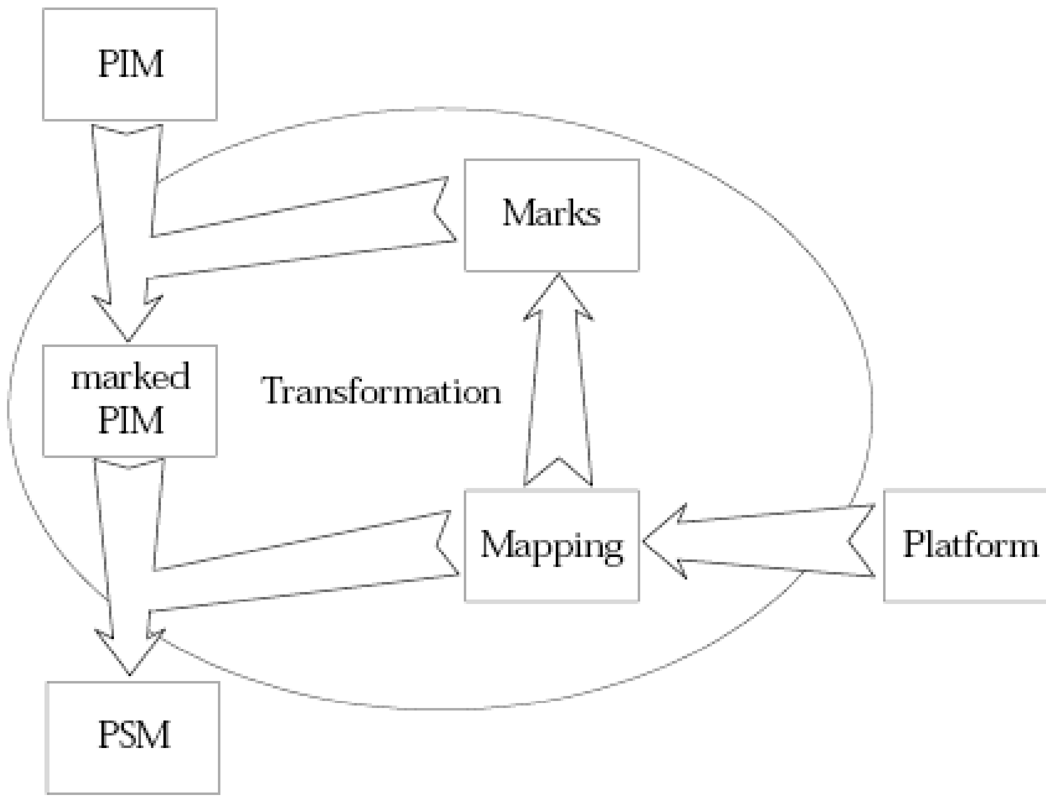


Metamodel tranformation



From MDA Guide Version 1.0.1

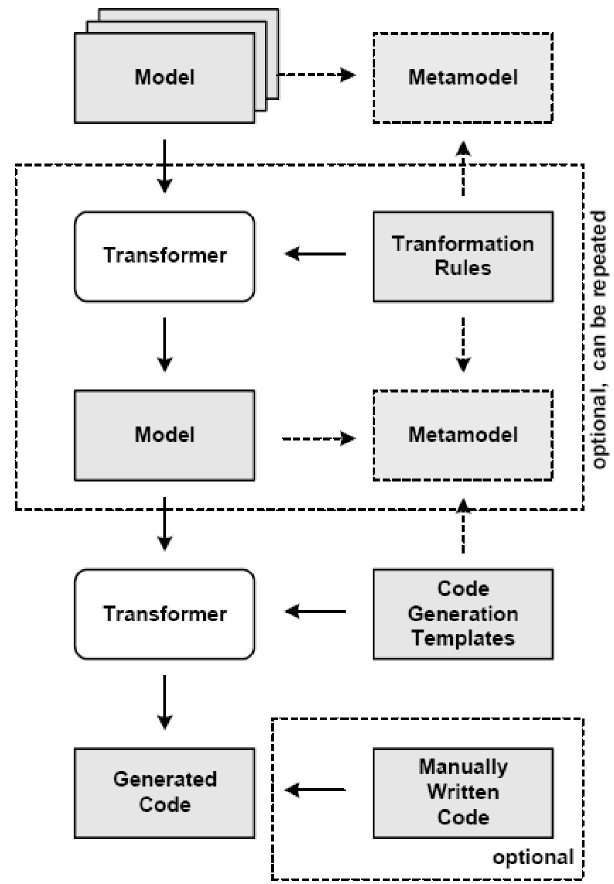
Model tranformation



From MDA Guide Version 1.0.1

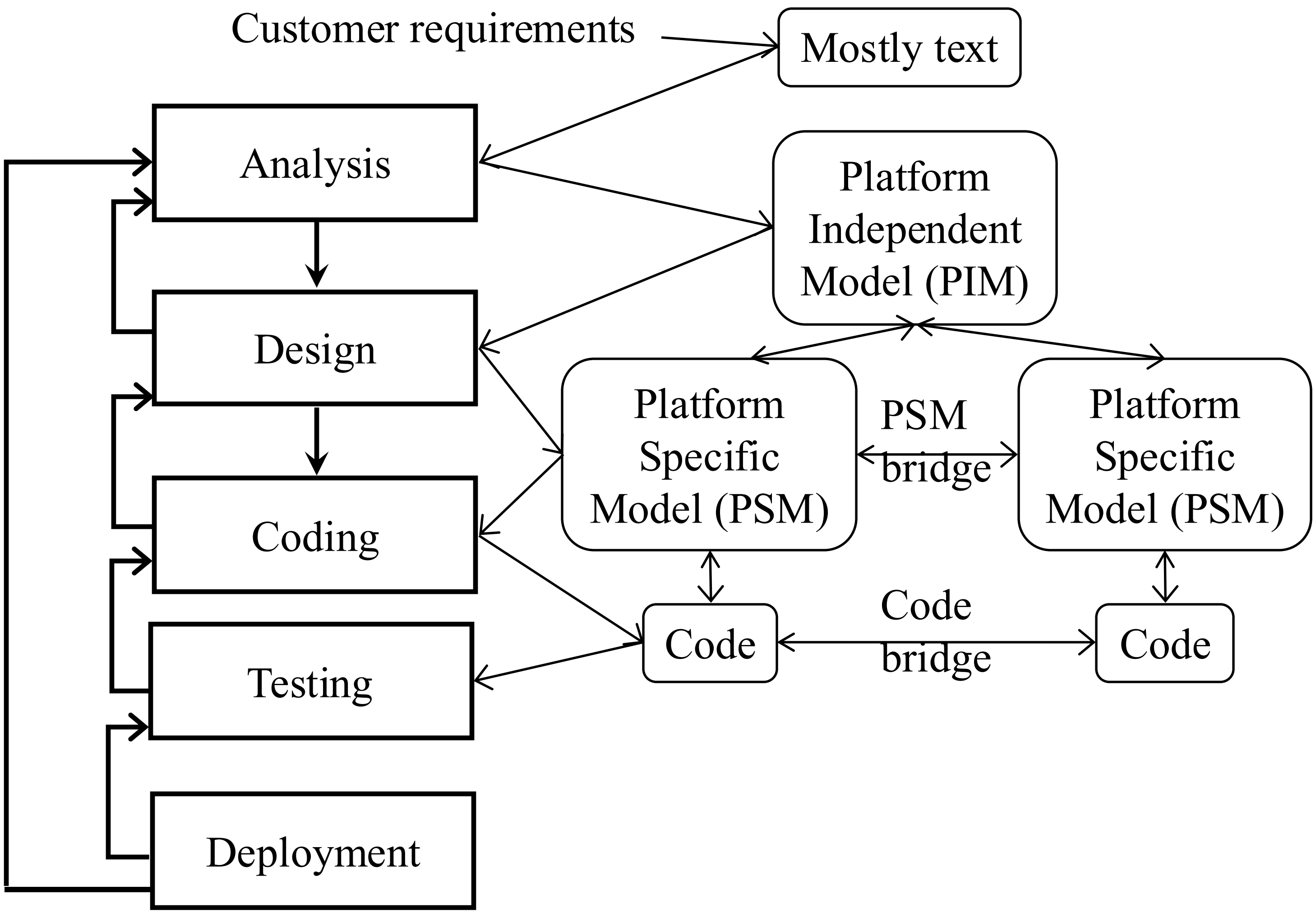
Model Driven Development

- Developer develops **model(s)** based on certain metamodel(s).
- Using **code generation templates**, the model is transformed to executable code.
- Optionally, the **generated code is merged** with manually written code.
- One or more **model-to-model transformation steps** may precede code generation.



Adopted from Alan Brown

Model Driven Architecture



Summary

- Software engineering is a modeling, problem-solving activity, knowledge acquisition activity and rationale-driven activity.
- Dealing with complexity:
  - Abstraction
  - Decomposition
  - Hierarchy
- Software Engineering activities
  - – Analysis
    - **Requirements elicitation**
    - **Requirements analysis**
  - – Design
    - **System design** –
    - **Object design**
  - – Implementation
  - – Testing
- Software Development Life Cycle
  - – Waterfall
  - – Incremental
  - – Spiral/UP
  - – Formal methods
  - – Agile methods

Next lecture

- UML

Chapter 2 in the text-book.