

MODEL-DRIVEN SOFTWARE DEVELOPMENT

Jörg Kienzle

OVERVIEW

- What is Software Engineering?
- Software Development Processes
- Software Development Phases
- Model-Driven Engineering
- UML
- Overview of Process we will use
 - Background on Fondue, Fusion, etc...
 - Overview of Models that we will build

WHAT IS SOFTWARE ENGINEERING

- **Software:**

- Computer programs, procedures, and possibly associated documentation and data pertaining to the operation of a computer system

According to
the IEEE

- **Software Engineering:**

- The application of a **systematic, disciplined, quantifiable approach** to the **development, operation, and maintenance** of software
- In other words: the application of engineering to software

INCREASING COMPLEXITY



- Scope, complexity and pervasiveness of computer-based and controlled systems continue to increase
- Software assumes more and more responsibility

CHALLENGES FOR SE

- 1960's: Cope with software correctness
 - Milestone: Floyd 'assigning meaning to programs'
- 1970's: Cope with project size
 - Milestone: Parnas, Yourdon: modularity & structure
- 1980's: Cope with variability in requirements
 - Milestone: Jackson, Meyer: modeling, object orientation
- 1990's: Cope with distributed systems and mass deployment:
 - Milestone: Szyperski: product-lines & components
- 2000's: pervasive software integration, system of systems, accelerating technological changes
 - Milestone: ?

LONG-TERM AVAILABILITY

- AIRBUS A300 Life Cycle

- Program began in 1972, production stopped in 2007
- $2007 - 1972 = 35$ years...
- Support will last until 2050
 - $2050 - 1972 = 78$ years !!



DEPENDABILITY

- Consequences of systems failing
 - Annoying to catastrophic
 - Opportunities lost, businesses failed, security breaches, systems destroyed, lives lost



On June 4, 1996 an Ariane V rocket launched by the European Space Agency exploded just forty seconds after lift-off

SOFTWARE DEVELOPMENT PROCESS

- A well-defined and well-managed software engineering process
 - **Provides guidance** as to the order of a team's activities,
 - **Specifies** what **artifacts** should be developed,
 - **Directs** the **tasks** of individual developers and the team as a whole, and
 - **Offers criteria for monitoring and measuring** a project's products and activities.



Are we on Track?

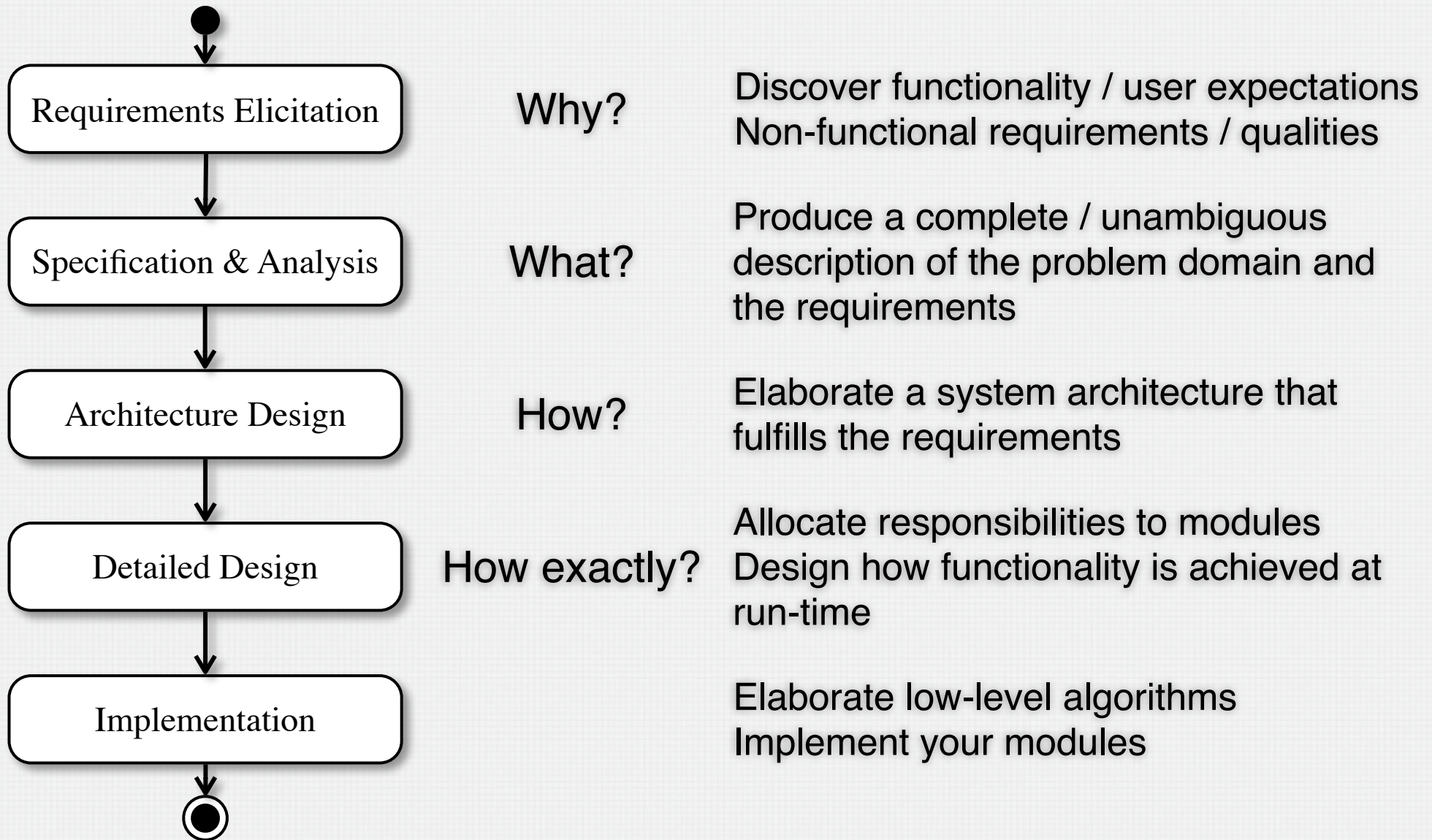
SOFTWARE PROCESS ACTIVITIES

- Primary activities
 - Development
 - Operation
 - Maintenance
- Supporting activities
 - Documentation
 - Configuration management
 - Quality assurance
 - Verification and validation
 - Training
- Process-related activities
 - Management
 - Infrastructure
 - Tailoring
 - Process assessment

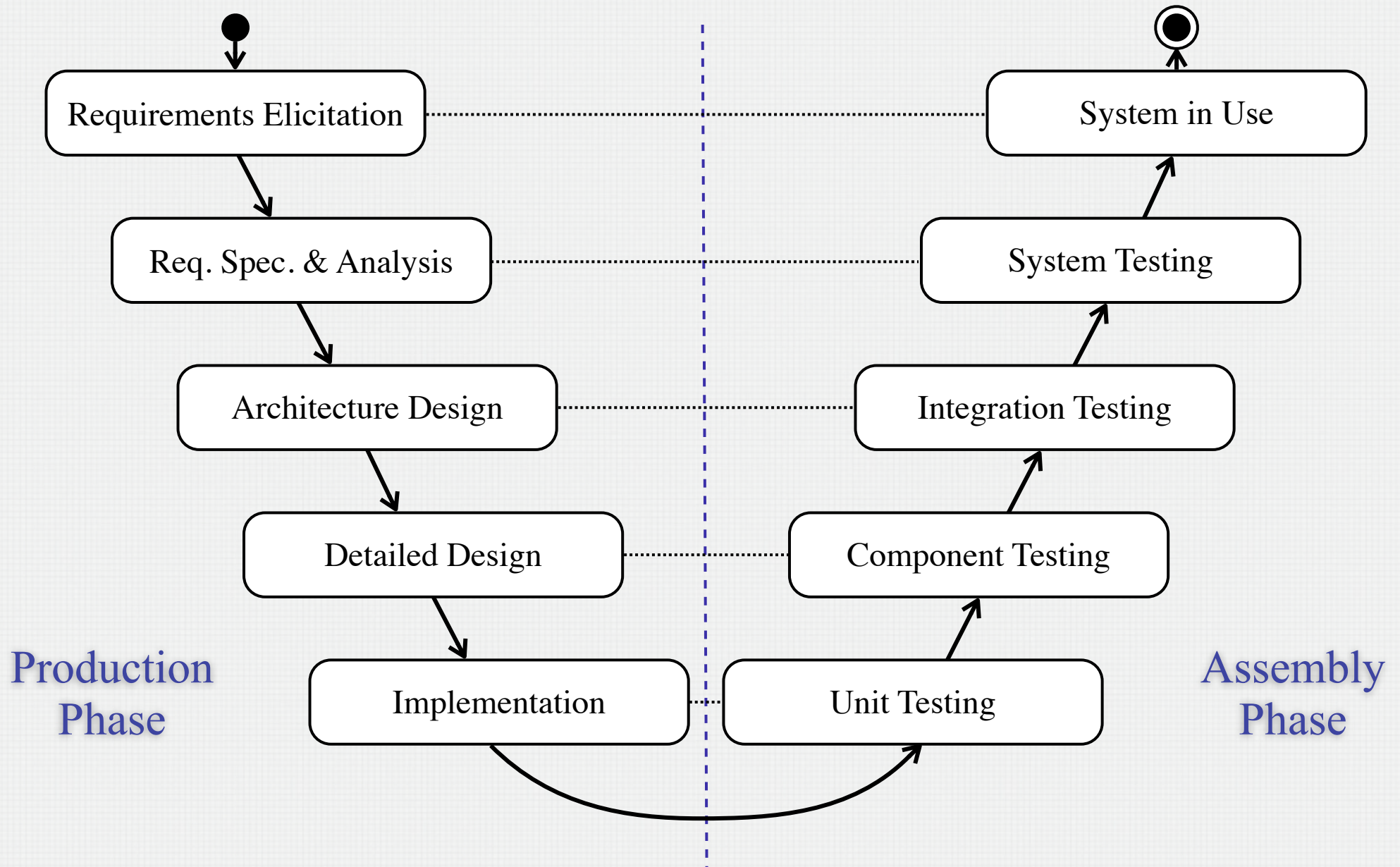
Systematic guidance on how to do this is called a **development method**

We will focus on this!

CLASSIC WATERFALL MODEL

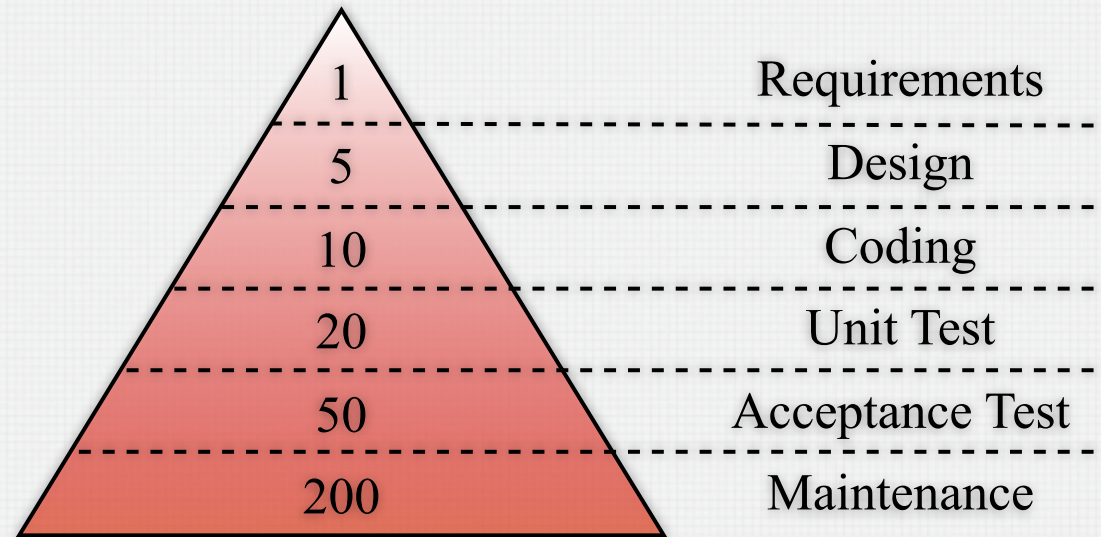


SOFTWARE DEVELOPMENT V MODEL



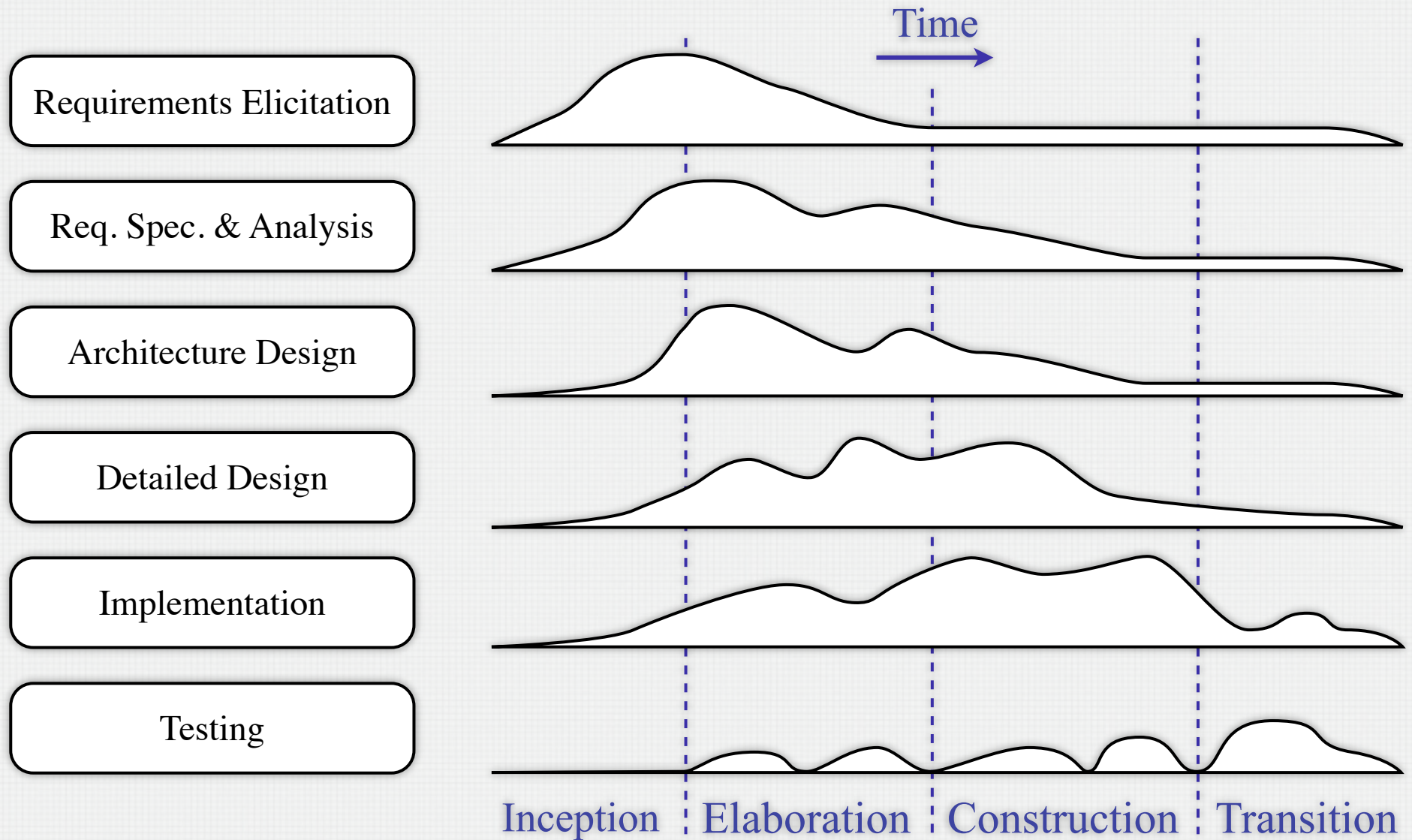
IMPORTANCE OF “GOOD” REQUIREMENTS

- **Faults / omissions made at the requirements stage are expensive** to fix later
 - Stated requirements might be implemented, but the system is not one that the customer wants
- Need to determine and establish the precise expectations of the customer!

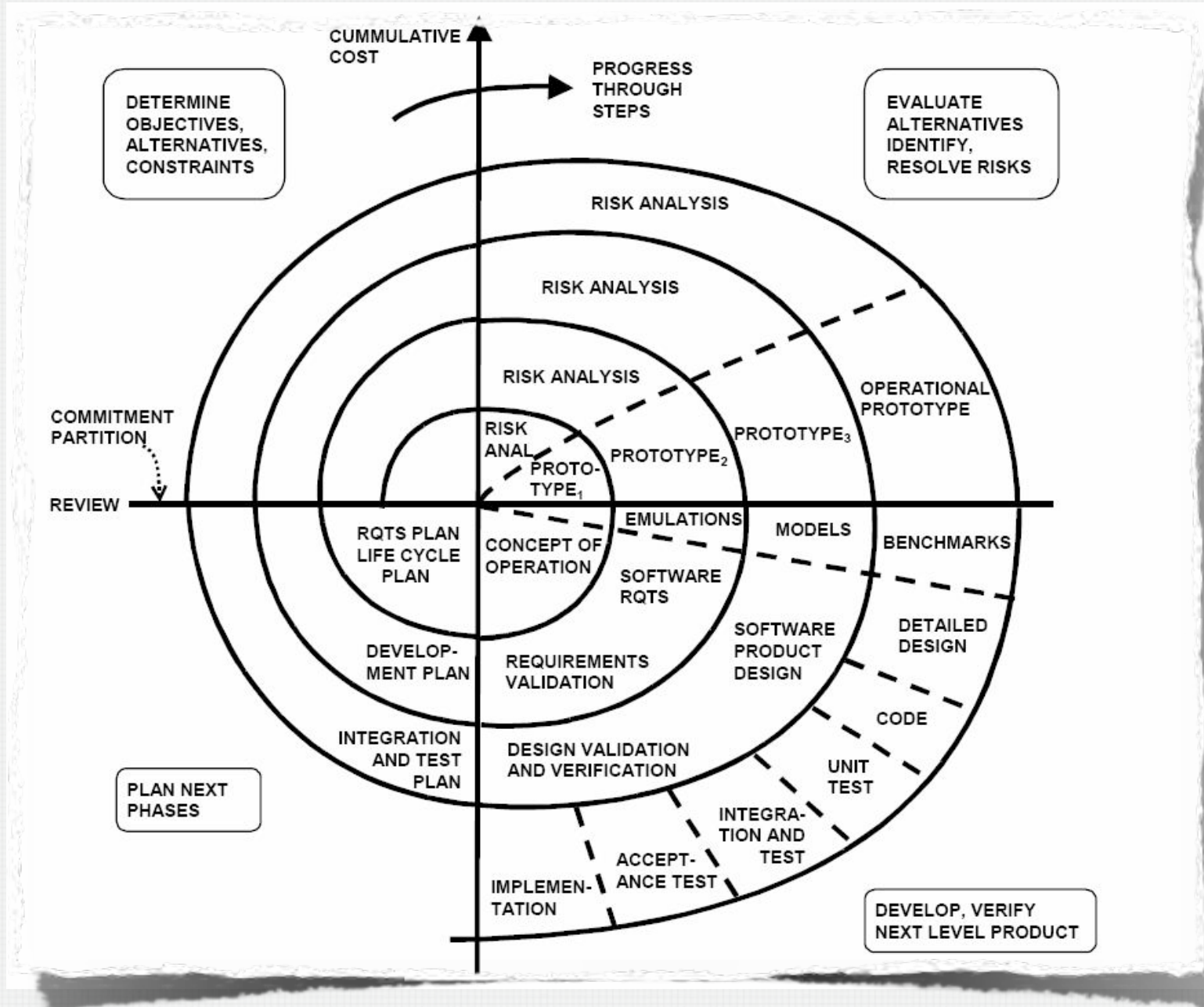


Relative Cost to Repair a Defect
at Different Lifecycle Phases [Davis 93]

ITERATIVE SOFTWARE DEVELOPMENT



SPIRAL METHODOLOGY



WHY MODELLING?

“Modeling, in the broadest sense, is the cost-effective **use of something in place of something else for some cognitive purpose**. It allows us to use something that is **simpler, safer or cheaper** than reality instead of reality for some purpose.”



Jeff Rothenberg
The Nature of Modeling
John Wiley & Sons, August 1989

WHY MODELLING?

“A model represents reality for a given purpose; the model is **an abstraction of reality** in the sense that it cannot represent all aspects of reality. This allows us to deal with the world in a simplified manner, **avoiding the complexity, danger and irreversibility** of reality.”

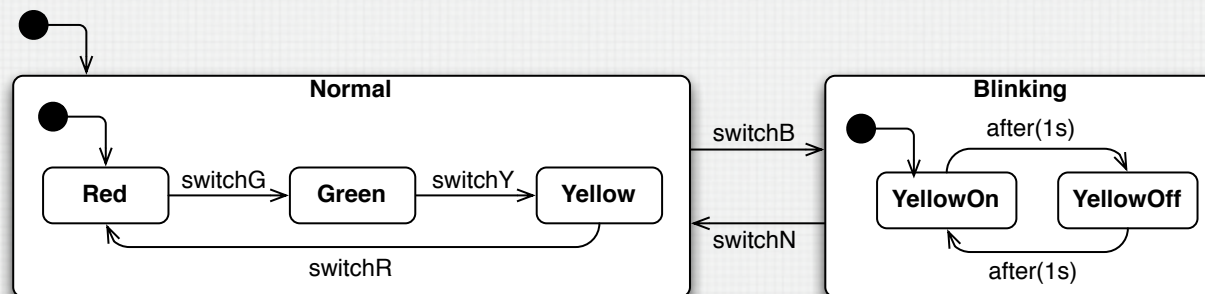


Jeff Rothenberg
The Nature of Modeling
John Wiley & Sons, August 1989

MODELLING AND SOFTWARE DEVELOPMENT

- A Model is a simplified representation of an aspect of the world for a specific purpose
- We use models to better understand a system
 - For an observer A, M is a model of an object O, if M helps A to answer questions about O. (Minsky)
- A model **helps to understand, communicate** and **build**
- Modelling and engineering: model something not yet existing!

M1 - Modelling Space



M0 - The World



MODEL-DRIVEN ENGINEERING

- A unified conceptual framework in which software development is seen as a process of model production, refinement and integration
- **Models are at the centre of the development activities**
 - Models are built representing **different views** of a software system using **different formalisms**, i.e., modelling languages, at **different levels of abstraction**, for **different purpose**
 - **Models are connected** by **model transformations**
 - High-level specification models are **refined** / **combined** / **transformed** to include more solution details until the produced models can be executed
- **Tools** are of major importance to effectively create, manipulate and transform models

EXAMPLE MODELS

Requirements Elicitation

Why?

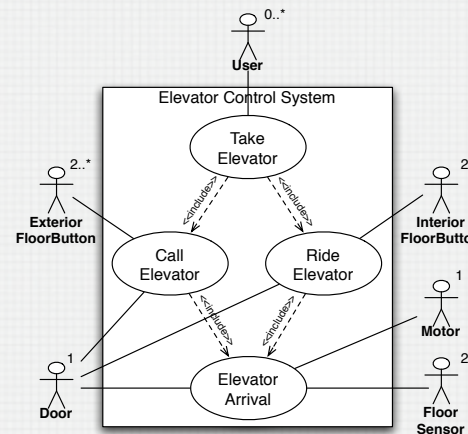
Generated Artifacts



Text Documents

Class name:	Superclass:	Subclasses:
Game		
Responsibilities:	Collaborations:	
Define and Initialize values	Board, Player	
Play TicTacToe	Board, Player	

CRC Cards

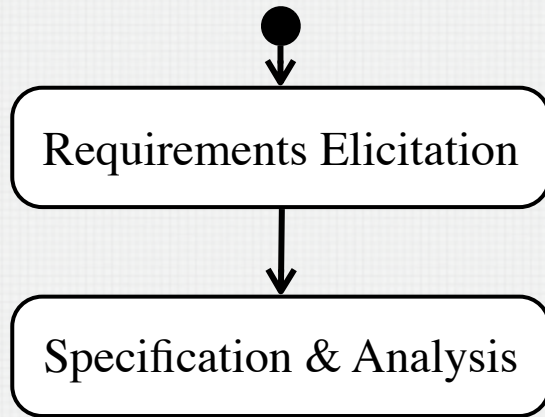


Use Case Diagram

Use Case: TakeElevator
Scope: Elevator Control System
Primary Actor: User
Intention: The intention of the User is to take the elevator to go to a destination floor.
Level: User Goal
Main Success Scenario:
 1. User **Call[s]Elevator**
 2. User **Ride[s]Elevator**
Extensions:
 1a. Cabin is already at User's floor...
 1b. User is already inside...

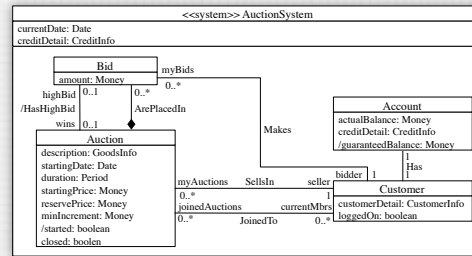
Use Cases

EXAMPLE MODELS

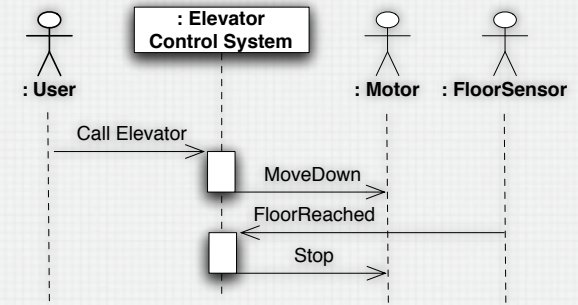


What?

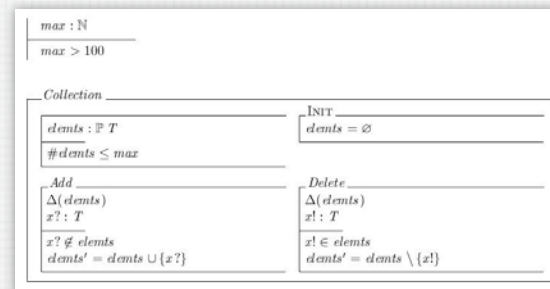
Generated Artifacts



Concept Model

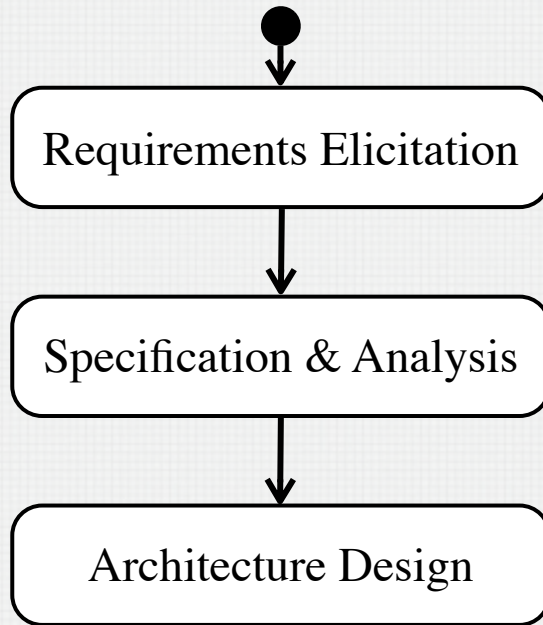


Environment Model



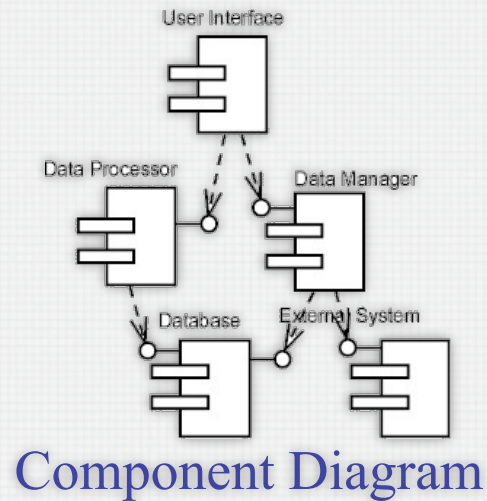
Z or B or OCL Specification

EXAMPLE MODELS

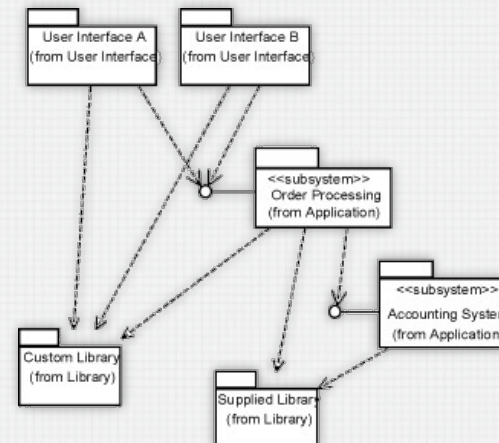


How?

Generated Artifacts



Component Diagram

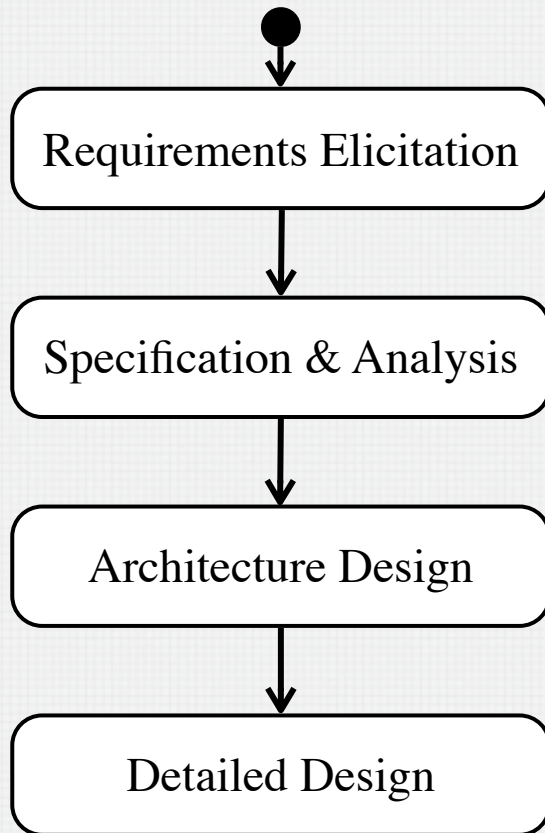


Package Diagram

```
archInstance {  
  componentInstance {  
    description = "Component 1"  
    interfaceInstance{...}  
  }  
  connectorInstance{  
    description = "Component 1"  
    interfaceInstance{...}  
  }  
  linkInstance{  
    description = "Conn1 to  
    Conn1 Link"  
    point {  
      (link) anchorOnInterface =  
        "#compl.IFACE_BOTTOM"  
    }  
  }  
}
```

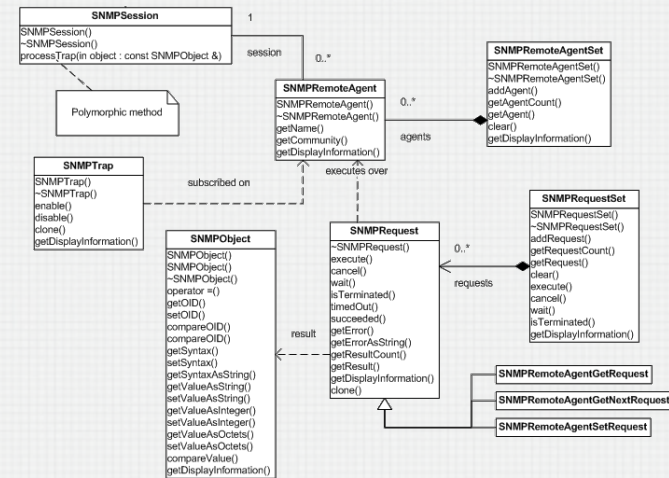
ADL Code

EXAMPLE MODELS

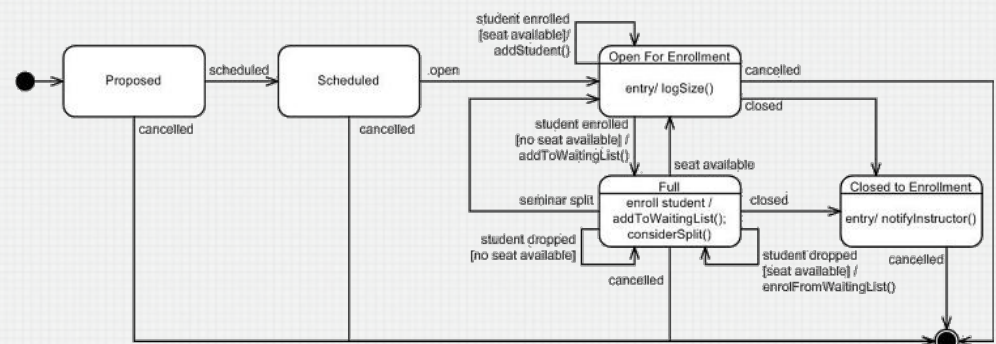


How exactly?

Generated Artifacts

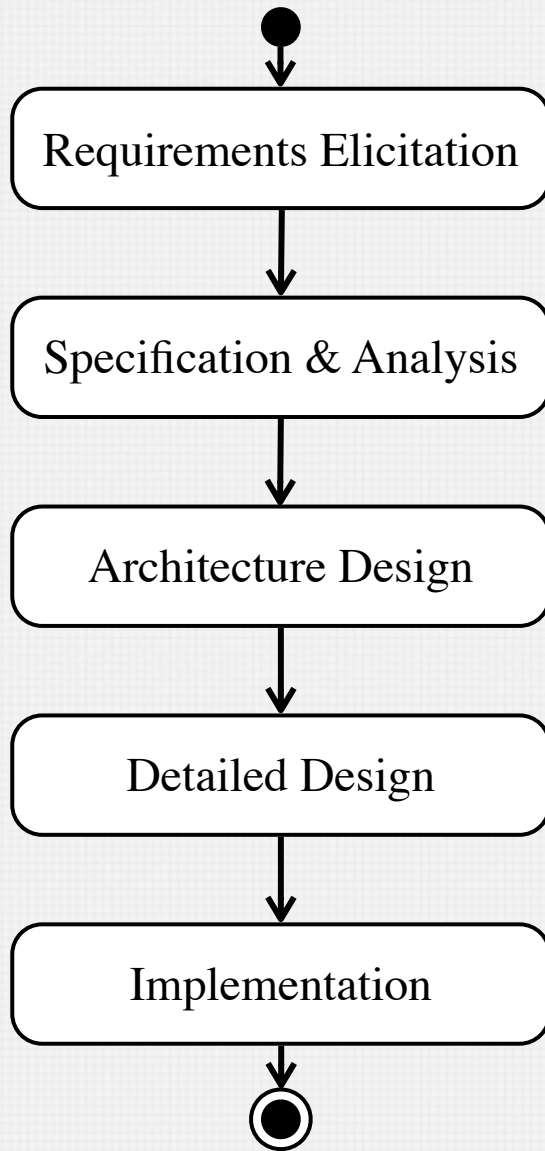


Class Diagram

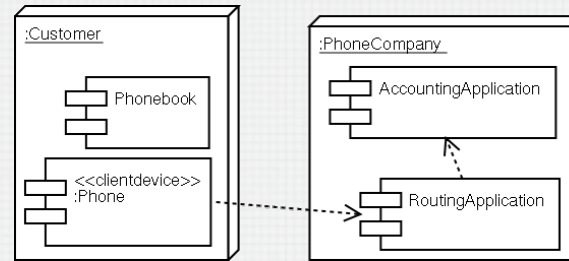


State Diagram

EXAMPLE MODELS



Generated Artifacts



Deployment Diagram

```
<?xml version="1.0"?>
<config version="1.0" serial="137"
timestamp="1145938502.12">
  <Lib>
    <Account>
      <LastUsed>1145938465</LastUsed>
      <LocalData>4215</LocalData>
      <Migration>63</Migration>
    </Account>
    <Call>
      <IncomingPolicy>everyone</IncomingPolicy>
      <MicVolume>96</MicVolume>
      <SkypeInPolicy>everyone</SkypeInPolicy>
    </Call>
    <UI>
      <Profile>
        <LastOnlineStatus>2</LastOnlineStatus>
      </Profile>
    </UI>
  </config>
```

Configuration File

```
public class Asteroid
extends Model {

  //position
  float xPos;
  float yPos;

  //dynamics
  float speed;

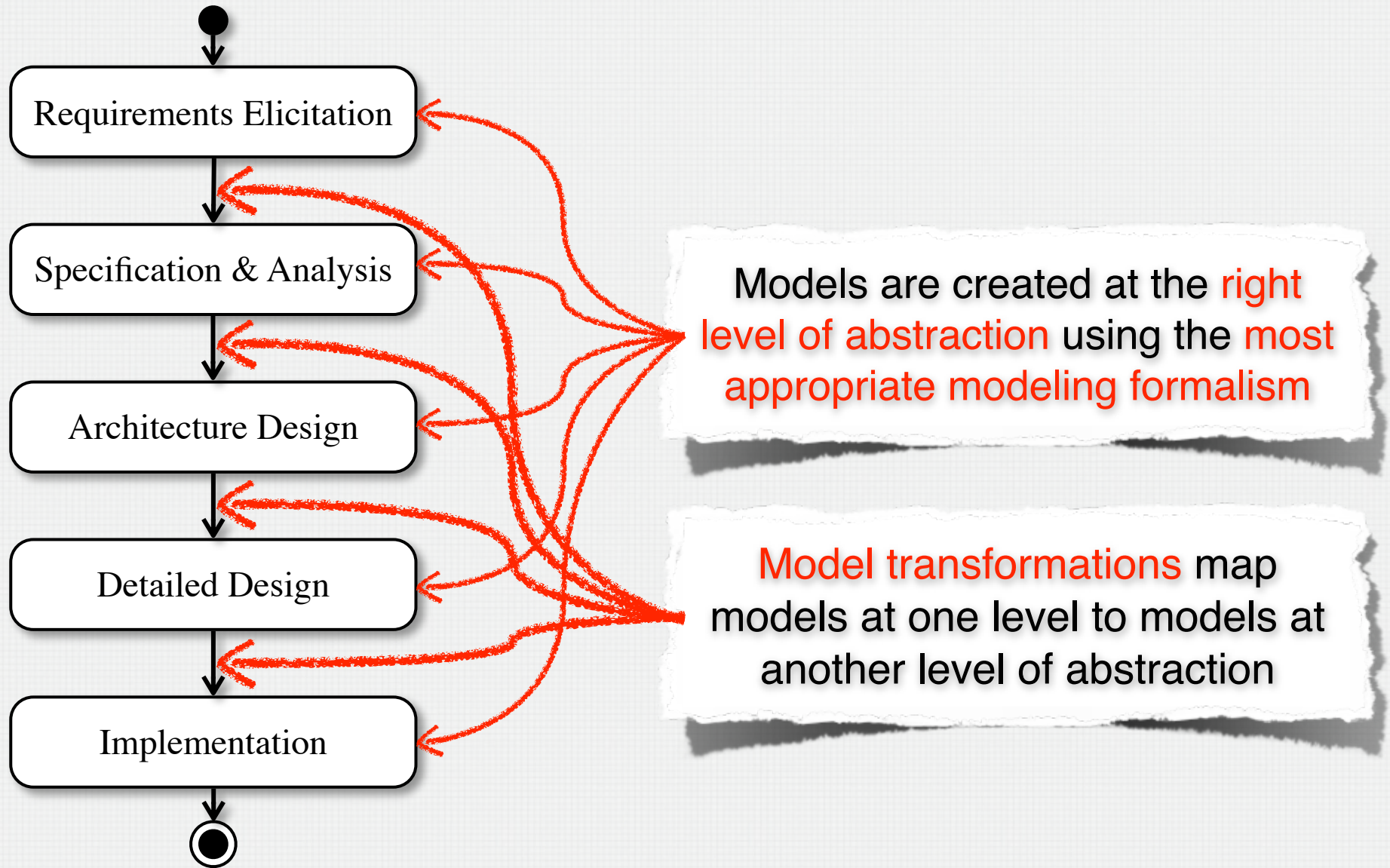
  public Asteroid() {
    xPos = ConstantWORLD_MAX_X;
    yPos = 0;
  }

  public void moveAsteroid() {
    xPos = xPos - speed;
  }

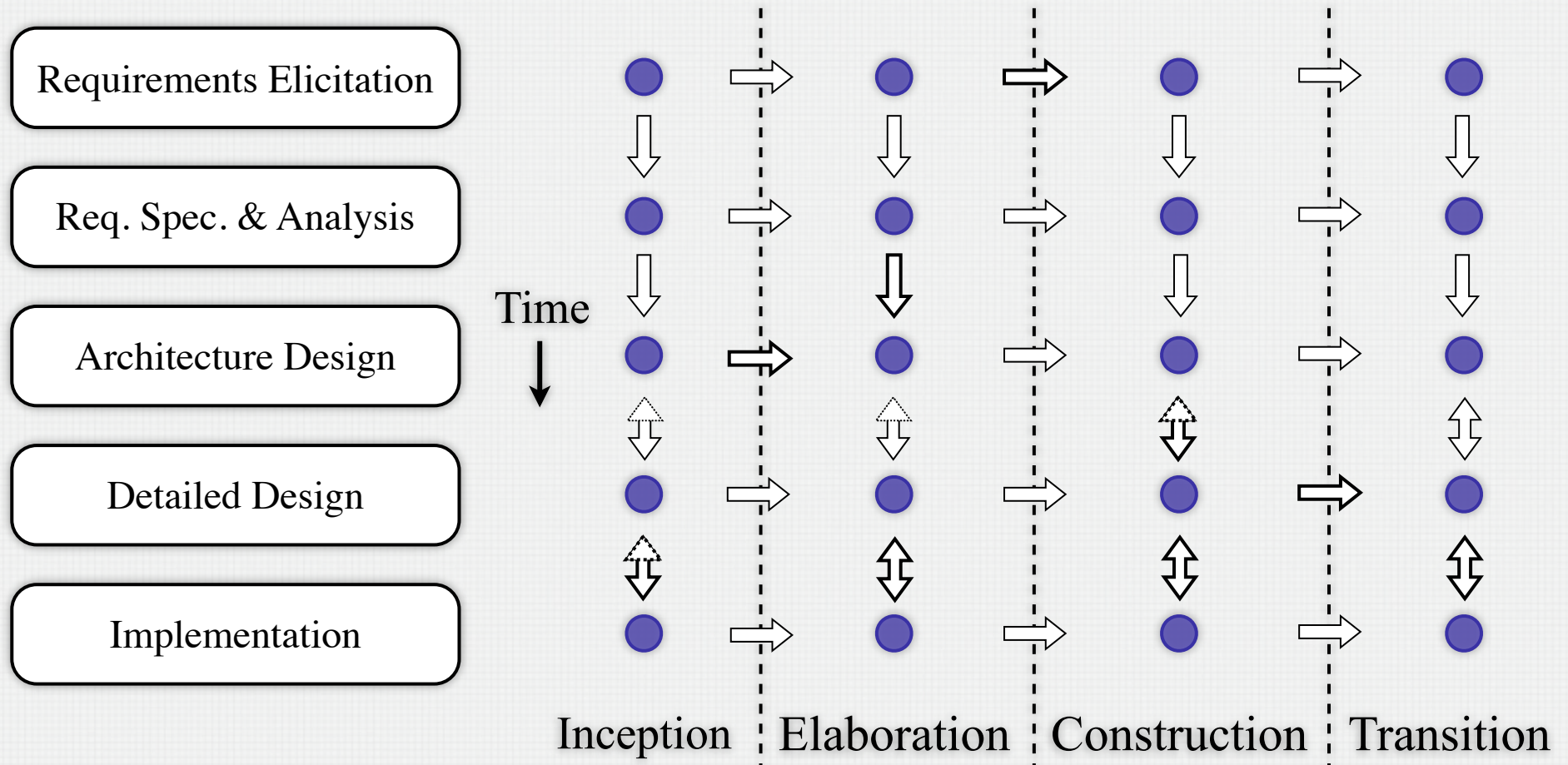
  public boolean outOfBounds() {
    return xPos < 0;
  }
}
```

Java Code

MODEL-DRIVEN ENGINEERING



MODERN MDE



ON MODELLING

- The **choice of models** and diagrams **has a profound influence upon how a problem is attacked** and how a corresponding solution is shaped
- **Abstraction is a key** to learning and communicating
- Every complex system is best approached through a **small set of nearly independent views of a model**; no single view is sufficient
- Every aspect of a system may be expressed at different levels of abstraction / fidelity

OBJECT-ORIENTATION

- Object-orientation is based on old principles
 - **Abstraction**
 - **Information hiding** and encapsulation
 - **Modularity**
 - **Classification**
- Object-orientation is based on a few concepts
 - **Object**
 - Groups together state and behaviour
 - **Class**
 - **Inheritance**
 - **Polymorphism**

OBJECT-ORIENTATION AND SE

- **Object-Orientation** stems from object-oriented programming, but **can be applied within the whole software development life cycle**
 - Requirements Elicitation and Specification
 - Design
 - Implementation
 - Testing
- **Object-Orientation is a way of thinking** about problems using models organized by real-world entities
- **Object-Orientation is an engineering method** used to create a representation of the problem domain and map it into a software solution

Experience has shown that OO alone is not enough!

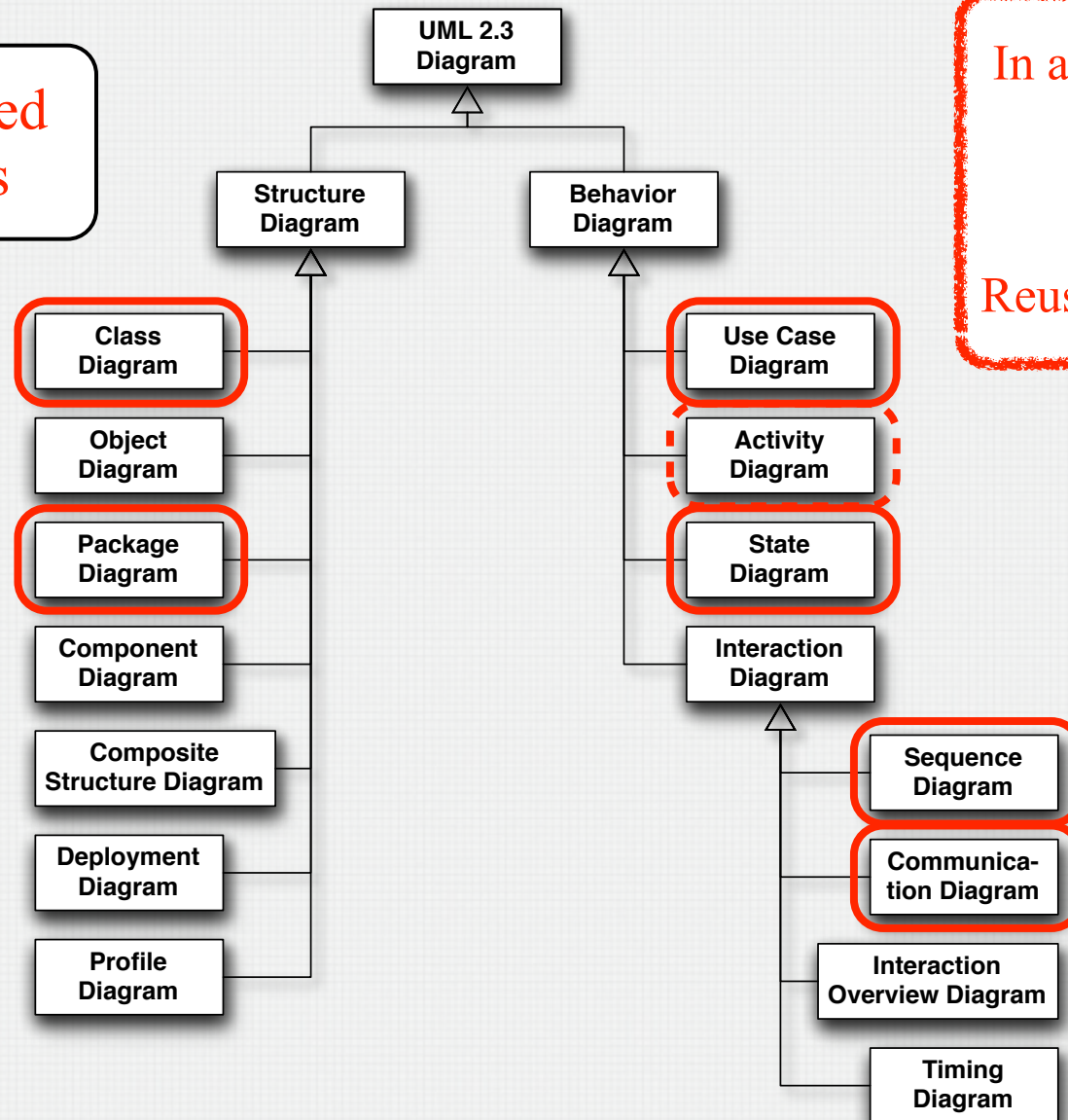
SCOPE OF UML

- The **Unified Modeling Language** (UML) is a language for specifying, constructing, visualizing, and documenting the artifacts of a software-intensive system
- UML fuses the concepts of the Booch, OMT, and OOSE methods
- UML is a single, common, and widely usable modelling language
- UML incorporates the object-oriented community's consensus on core modelling concepts
- UML allows deviations to be expressed in terms of its extension mechanisms
- Current version: UML 2.4.1



UML DIAGRAMS

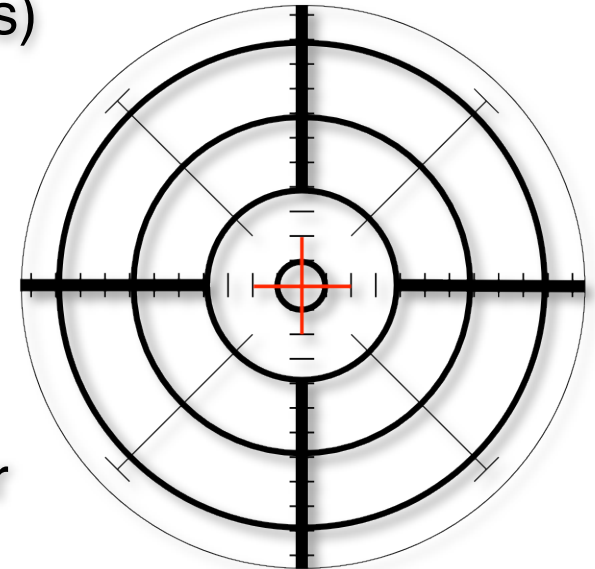
Notations Used
in this Class



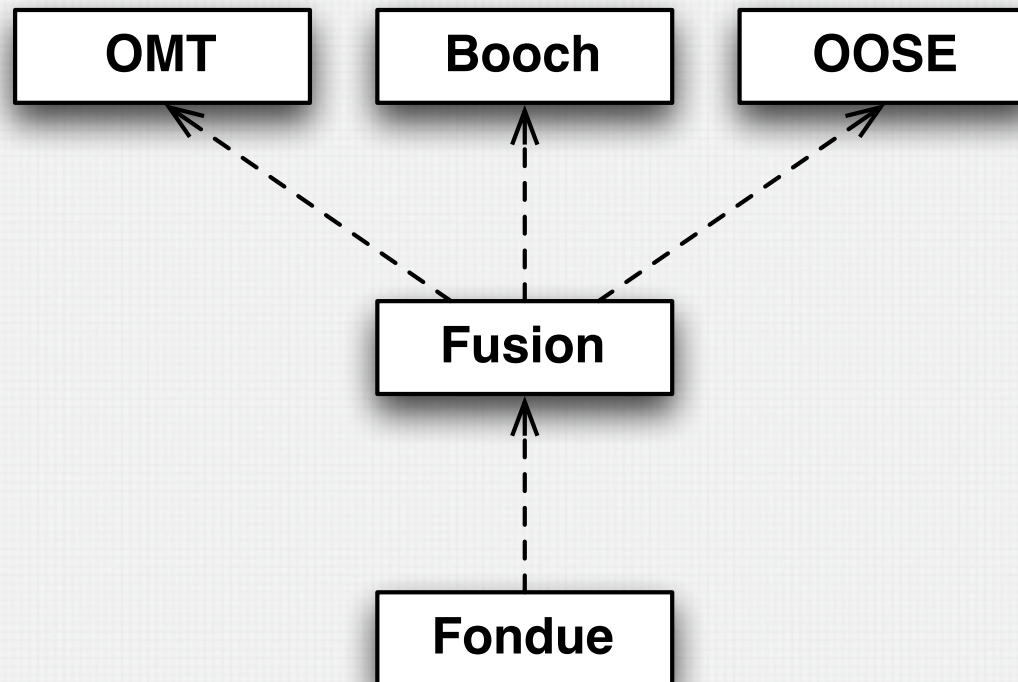
In addition we're going
to use:
Goal Models
Use Case Maps
Reusable Aspect Models

OUTSIDE THE SCOPE OF UML

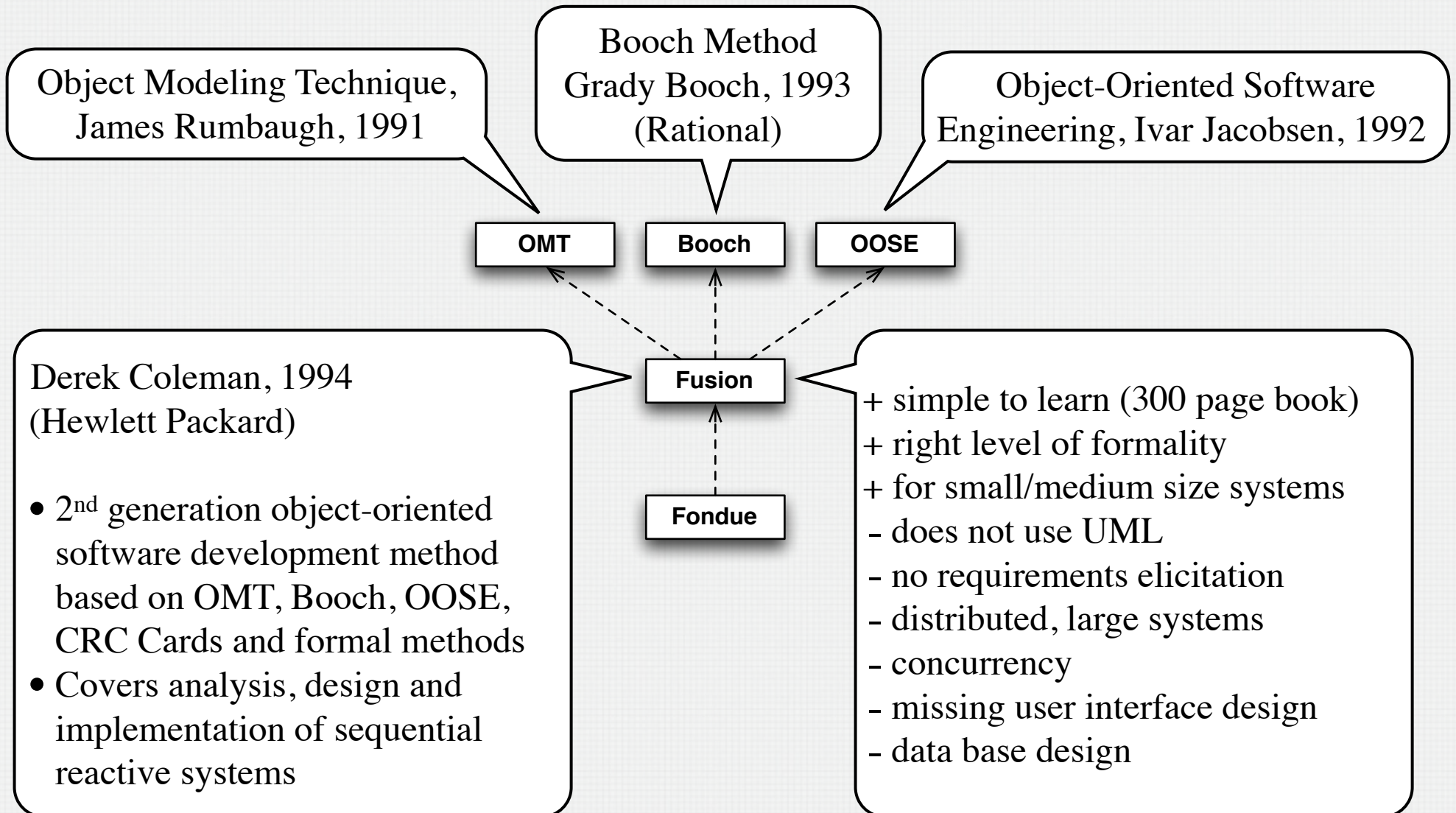
- Programming Languages
 - **UML is not** intended to be **a visual programming language**
- Tools
 - **UML does not define a tool interface**, storage, or run-time model (which could be used by CASE tool developers)
- Process
 - **UML is** intentionally **process independent**
 - Processes by their very nature must be tailored to the organization, culture, and problem domain at hand
 - What works in one context would be a disaster in another
 - The selection of a particular process will vary greatly, depending on such things like problem domain, implementation technology, and skills of the team



HISTORY OF FONDUE



HISTORY OF FONDUE

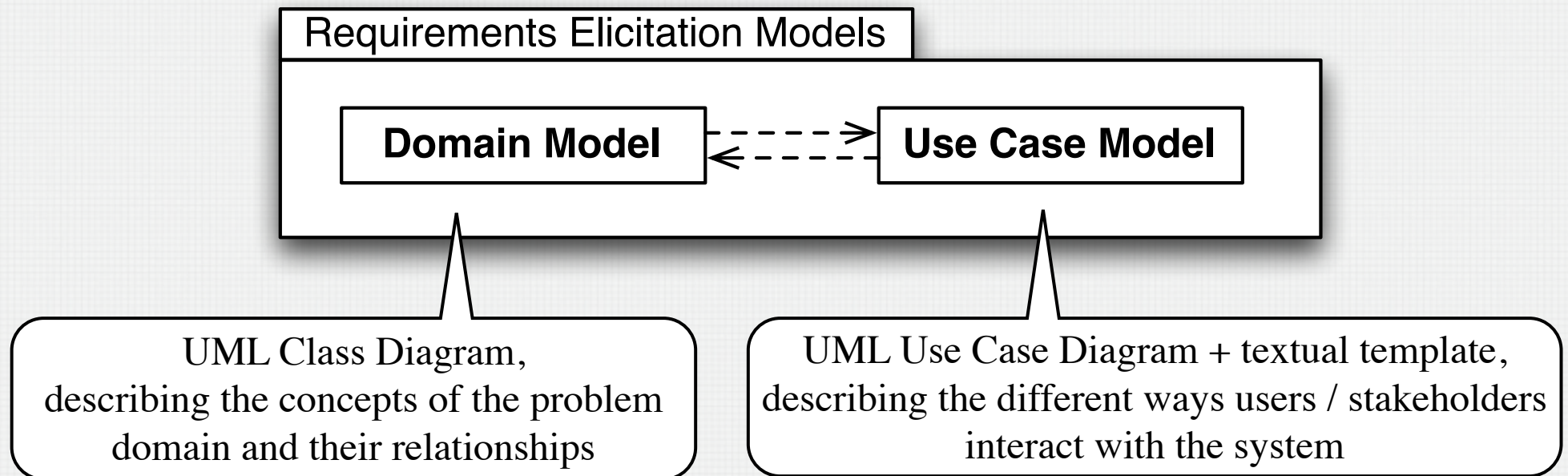


WHAT IS FONDUE?

- **Fondue** is a development method that **extends** the process and the models of the original **Fusion** method
- **Fondue uses** the **UML** notation
- **Use cases** are used during requirements elicitation
- **Operations are specified formally** in operation schemas by pre- and postconditions **using** the **OCL**
- Fondue follows the philosophy of **model-driven engineering**
 - For example: the domain model is refined into a concept model, then into a design class model and finally an implementation class model

FONDUE MODELS: REQUIREMENTS ELICITATION

In addition we're going to use Goal Models to capture intentions of stakeholders, soft goals and tradeoffs



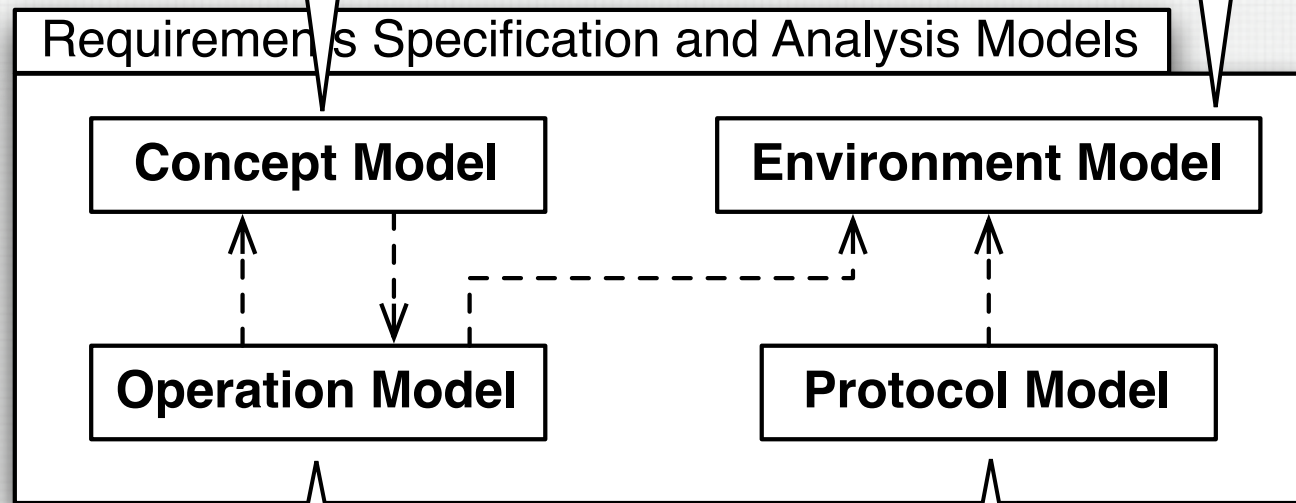
REQUIREMENTS SPECIFICATION / ANALYSIS

- The analyst defines the intended behaviour of the system
- Models are produced, which describe
 - The **concepts** that exist **in the system**.
 - The **relationships between concepts**.
 - The **boundaries of the system**.
 - The **operations** that can be performed on the system.
 - The **allowable sequences of those operations**.
- Fondue does not attach the operations to particular classes during analysis.

FONDUE MODELS: REQUIREMENTS SPEC.

UML Class Diagram,
describing the conceptual
state of the system

UML Communication Diagram,
describing the system interface (i.e. system
boundary and input / output messages)



OCL Pre- and Postconditions,
describing the desired effect of each
system operation on the conceptual state

We're going to use
Use Case Maps to describe the
system protocol more formally

DESIGN ACTIVITIES

- The designer chooses how the system operations are to be implemented by interacting objects at run-time.
- Different ways of breaking up a system operation into interactions can/should be tried.
- The operations are attached to classes.
- The design phase delivers models that show:
 - **How system operations are implemented** by interacting objects.
 - **How classes refer to one another** (in order to achieve interaction).
 - How classes are related by **inheritance**.
 - The **attributes** and **methods** of classes.

DESIGN: HOW?

In addition we're going to use:
Reusable Aspect Models, to illustrate
the advantages of aspect-orientation

UML Communication Diagram or Sequence
Diagram, describing how system operations are
implemented by communicating objects at run-time

Design Models

Interaction Model

UML Class Diagram,
showing inter-class
dependencies

Dependency Model

Design Class Model

Inheritance Model

UML Class Diagram, describing the
static structure of the design

UML Class Diagram, describing the inheritance
relationships among classes the design

STEPWISE REFINEMENT

- Initially, some classes end up with lots of responsibilities
 - Designers may need to investigate the substructures of some classes and their operations
- Hierarchical decomposition is used
 - The class is regarded as a subsystem
 - The analysis and design phases are applied to the subsystem
- Commonalities between classes are discovered
 - Inheritance is used to refactor common structure and behaviour



IMPLEMENTATION: BUILD IT

- The design is mapped to a particular programming language.
- Fondue provides guidance on how this is done:
 - Inheritance, attributes, and methods are implemented in classes (if the construct is provided by the programming language).
 - Object interactions are implemented as calls to methods belonging to classes.
 - The permitted sequences of operations are recognized by a finite state machine.
- Result: Implementation Class Model
(Class Diagram / Text / Code)

QUESTIONS

