Software Modelling Languages

Marek Rychlý
rychly@fit.vutbr.cz

Brno University of Technology
Faculty of Information Technology
Department of Information Systems

Information Systems Analysis and Design (AIS)
3 October 2019

Outline

1. Modern Software (Development) Process Models
   - Prototyping
   - Iterative/Incremental and Agile Development
   - Model Driven Development and Component Based Development
2. Software Design and Software Modelling Languages
   - Function-Oriented Design
   - Object-Oriented Design and Unified Modelling Language
   - Kruchten's 4+1 View Model of Architecture

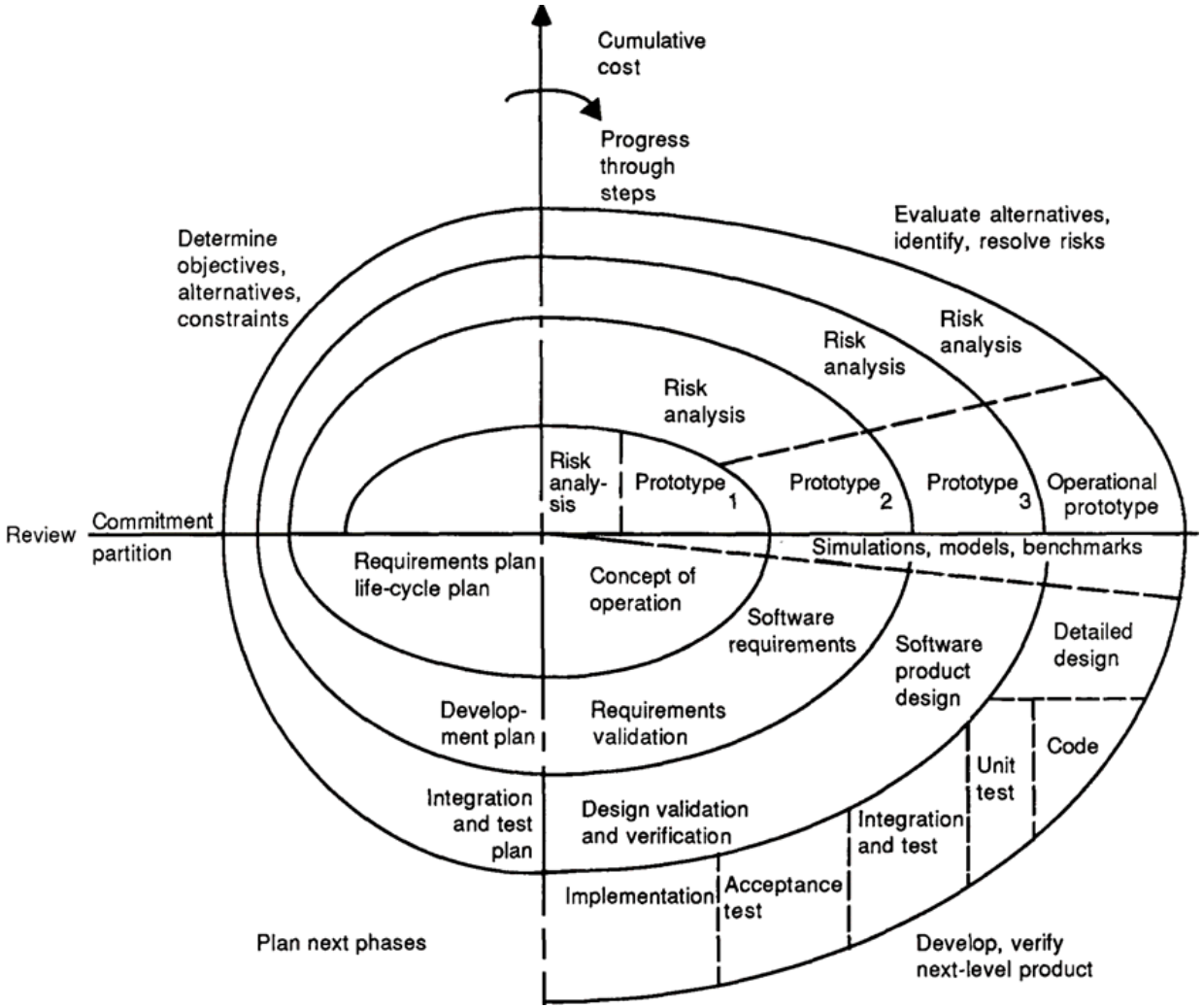
Software Prototyping

Definition (Prototyping by ISO/IEC/IEEE 24765:2010)

A hardware and software development technique in which a preliminary version of part or all of the hardware or software is developed to permit user feedback, determine feasibility, or investigate timing or other issues in support of the development process.

- Already introduced in the spiral SW process model.
  (by Barry W. Boehm in 1988, see slide 5)
- Several types of the prototyping with different goals.
  (throw-away, evolutionary, low/high fidelity prototyping, etc.)
- Utilised in nowadays development processes.
  (rapid application development, daily builds by continuous integration tools, etc.)

Software Prototyping in the Spiral Model



(adopted from “A spiral model of software development and enhancement” by B. W. Boehm)

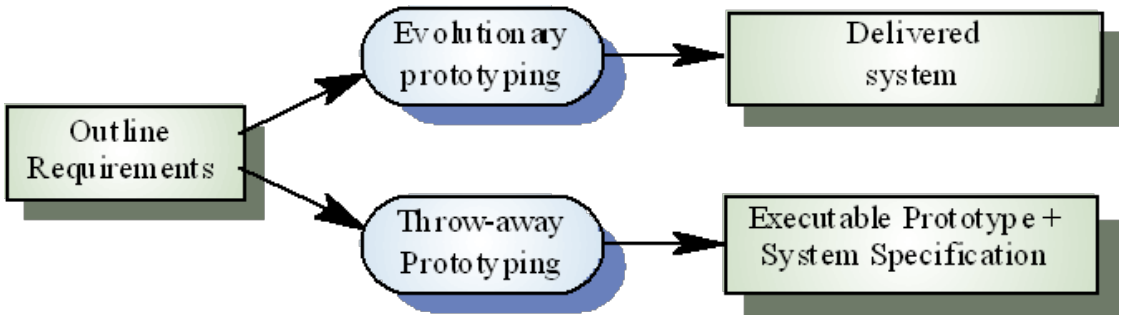
Types of Software Prototypes

throw-away developed from initial requirements but not used in the final product (however, its components can be reused)
(just a presentational prototype for a limited purposed, to clarify the requirements and reduce a project risk; deprecated nowadays)

evolutionary to build a robust prototype and constantly improve it
(a quicker delivery of initial versions of a software; the modern way)

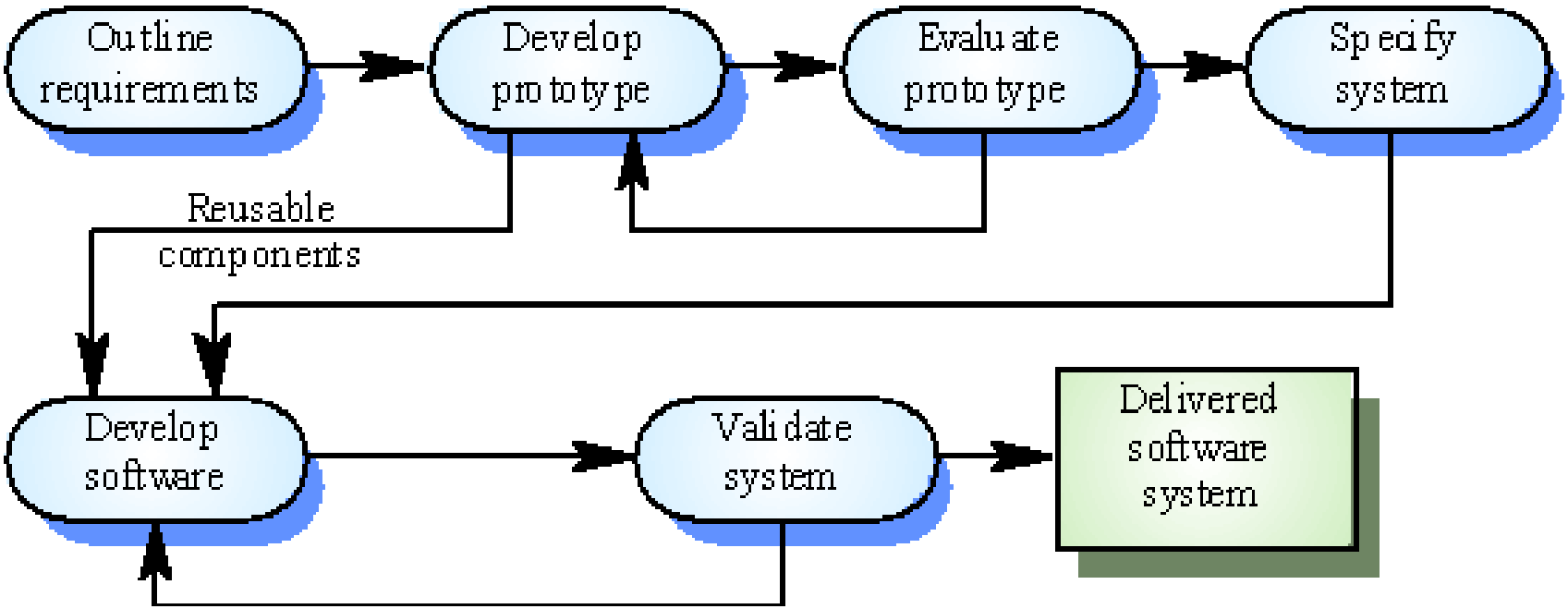
low fidelity to demonstrate general look&feel with a limited function
(to educate, communicate, and inform, not to show core functionality)

high fidelity to demonstrate core functionality in a user interface
(a fully interactive working system, however, not optimised)



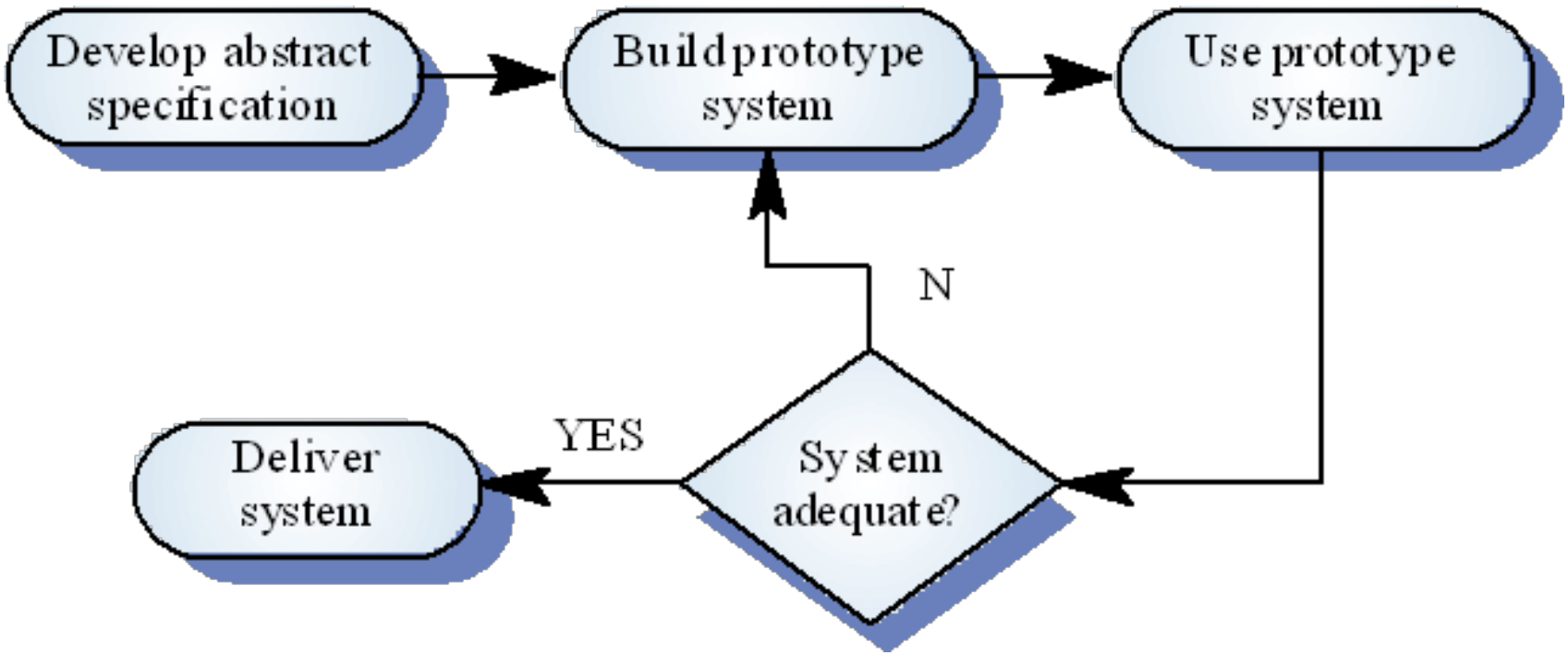
(adopted from “Software Engineering” by Ian Sommerville)

Throw-away Prototyping



(adopted from “Software Engineering” by Ian Sommerville)

Evolutionary Prototyping

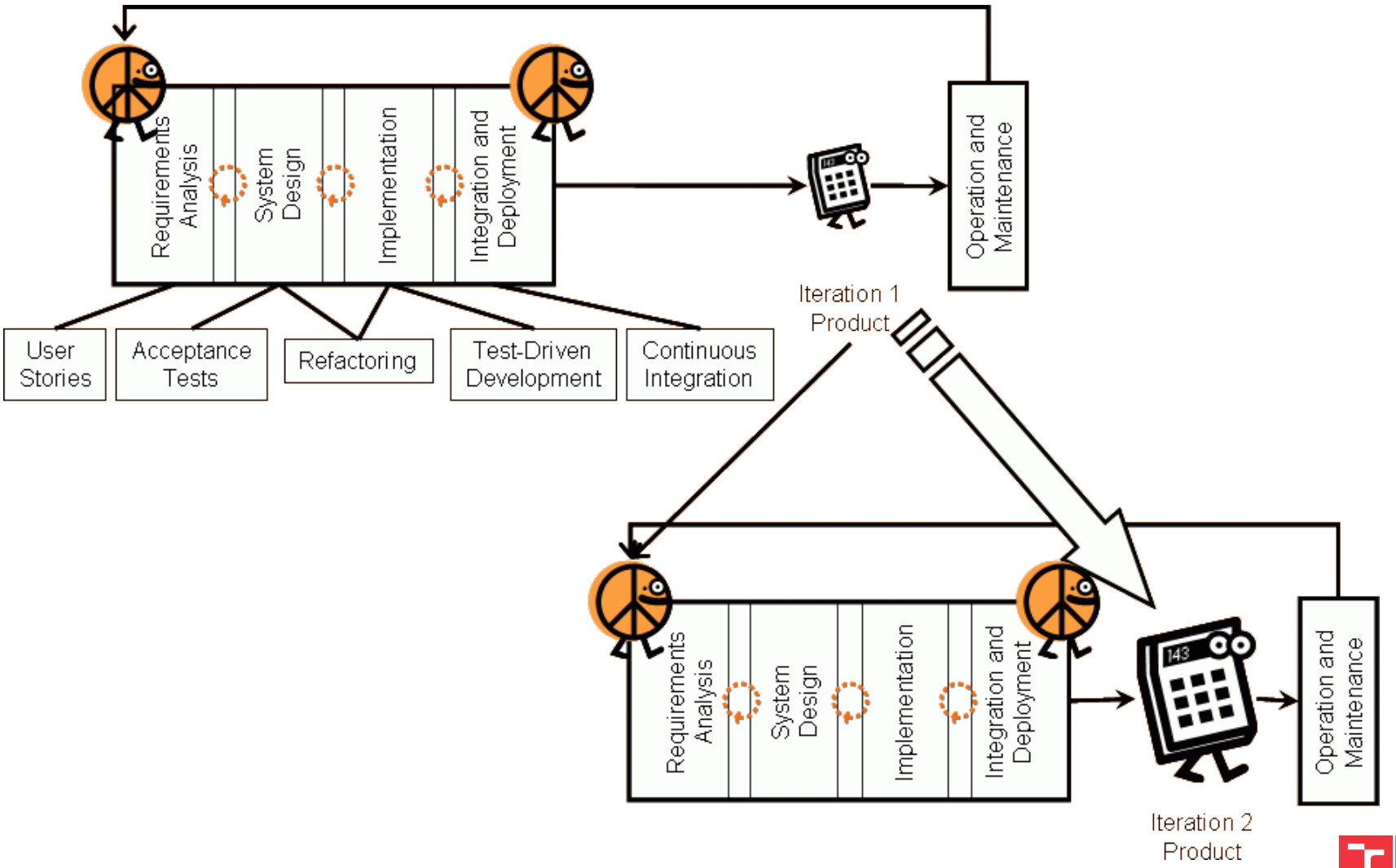


(adopted from “Software Engineering” by Ian Sommerville)

Iterative and Incremental Development Processes

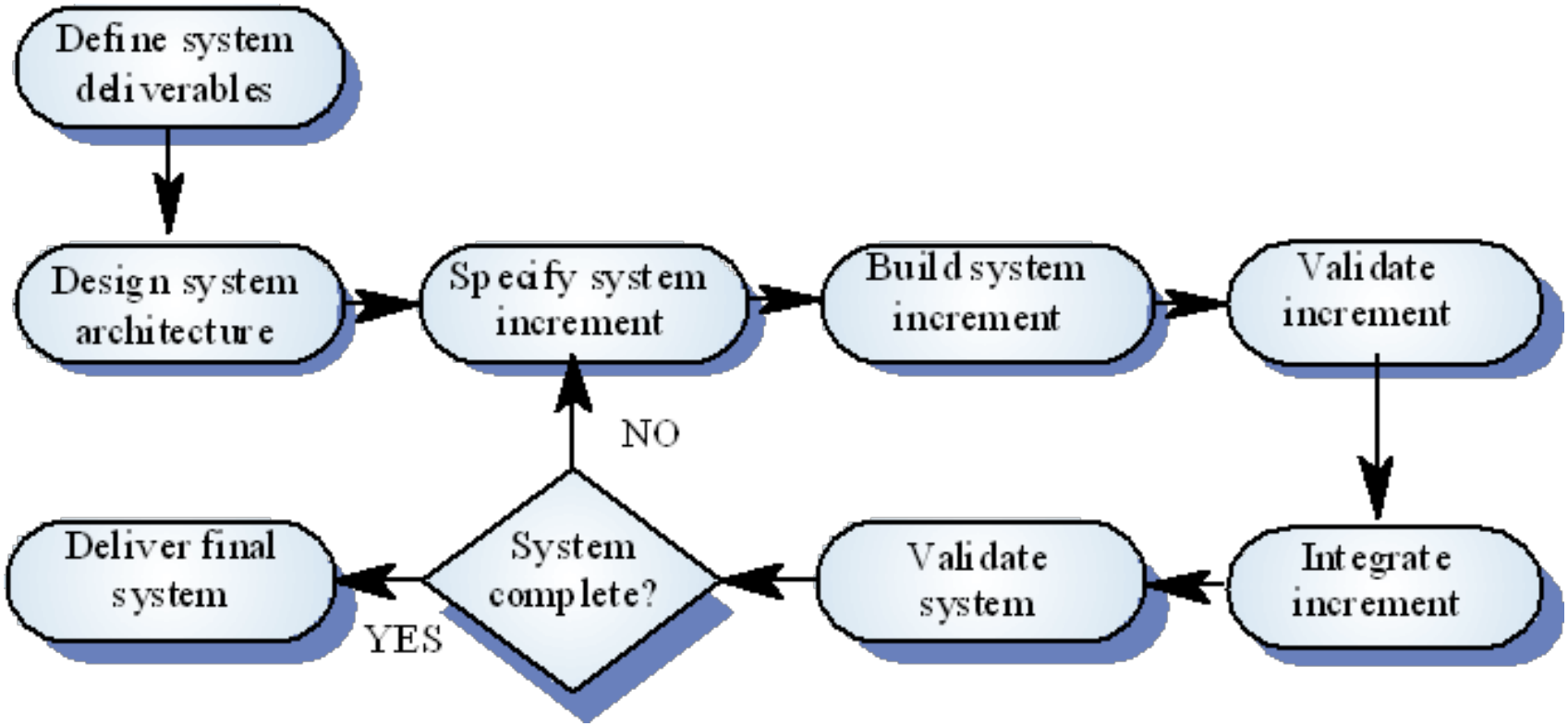
- Each iteration of in a development process improves the product.
  (to add new functionality, to improve its quality, etc.)
- The iterations have typically a fixed or time-boxed length.
  (from one to several months in RUP to two weeks in agile per iteration)
- Incremental development in the iterations.
  (each increment add a new functionality or extends an existing)
- The result of each iteration is a build.
  (it can be produced automatically by continuous integration tools)

Iterative Development Process (with Agile Techniques)



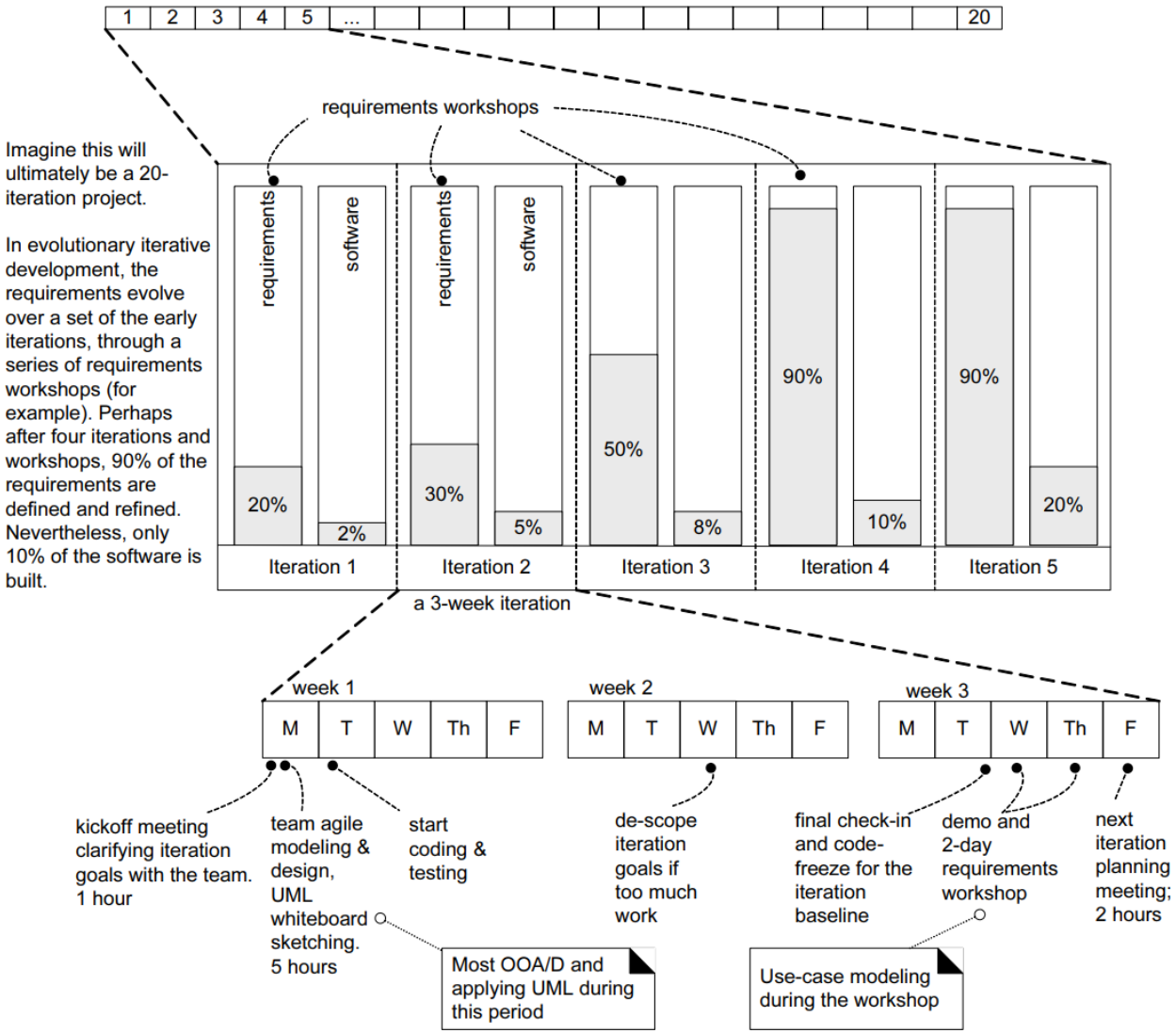
(adopted from “Practical Software Engineering” by Maciaszek&Liong)

Incremental Development Process



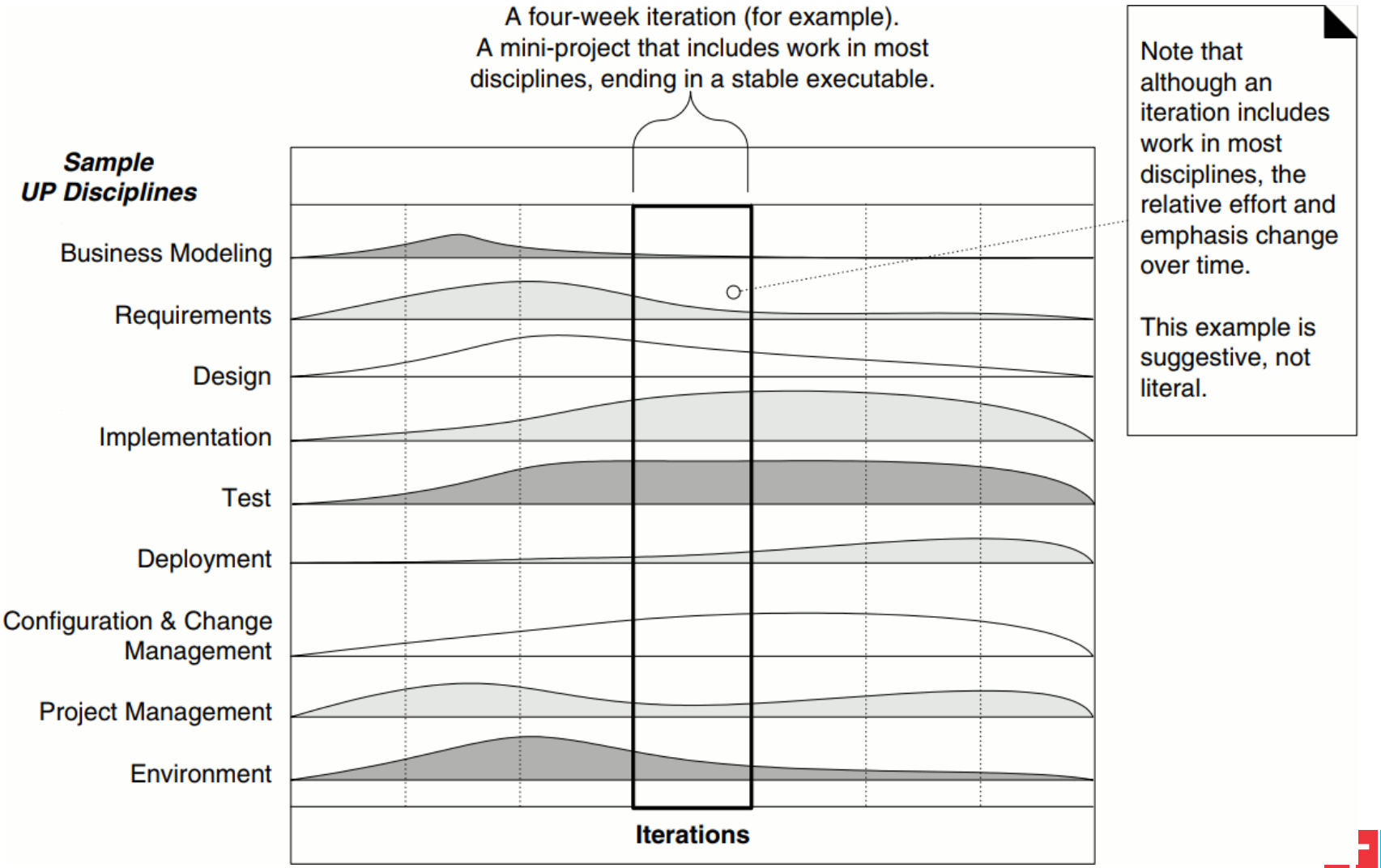
(adopted from “Software Engineering” by Ian Sommerville)

Requirements/Software Progress through Iterations



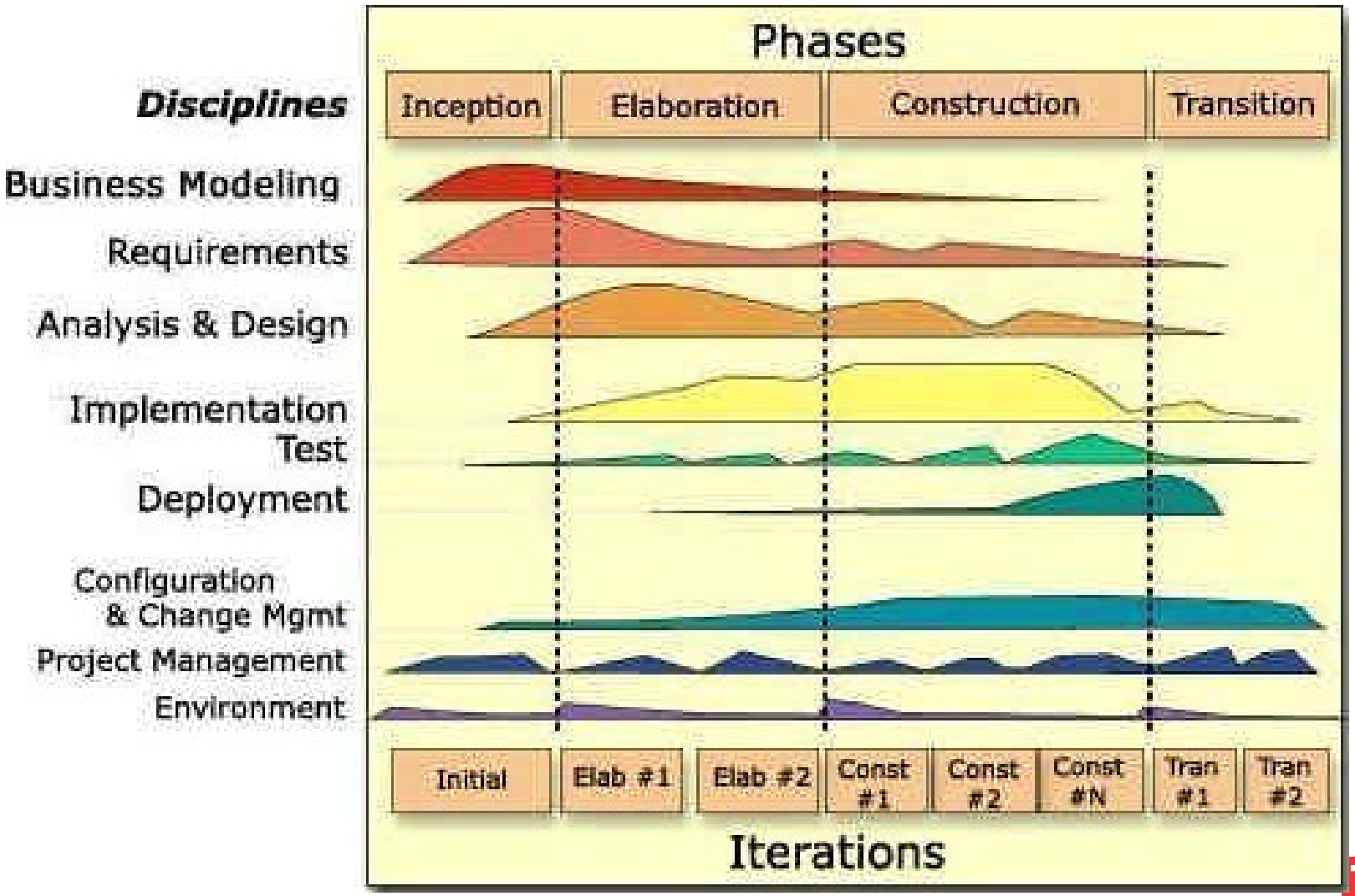
(adopted from “Applying UML and Patterns” by Craig Larman)

Unified Process Disciples through Iterations



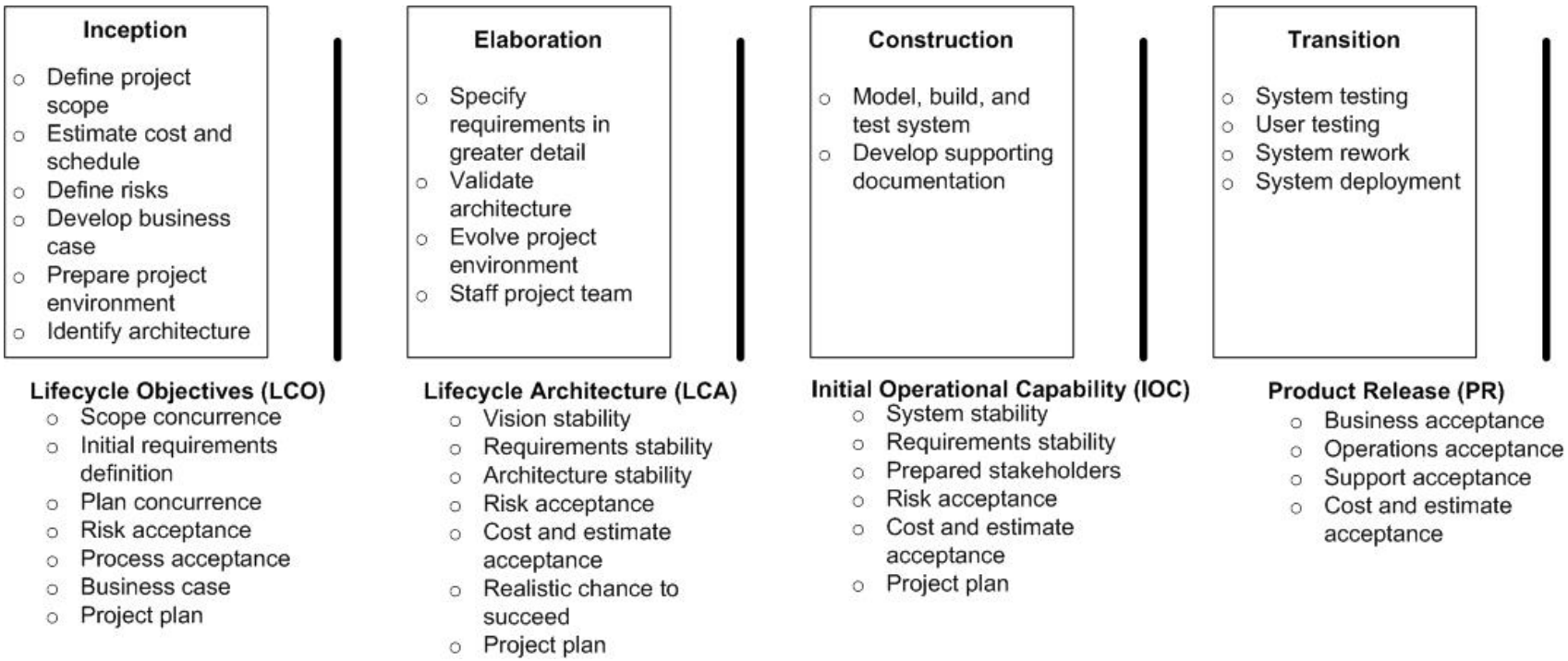
(adopted from “Applying UML and Patterns” by Craig Larman)

(Rational) Unified Process Phases



(adopted from “A Manager’s Introduction to The Rational Unified Process” by Scott W. Ambler)

(Rational) Unified Process Milestones



(adopted from “A Manager’s Introduction to The Rational Unified Process” by Scott W. Ambler)

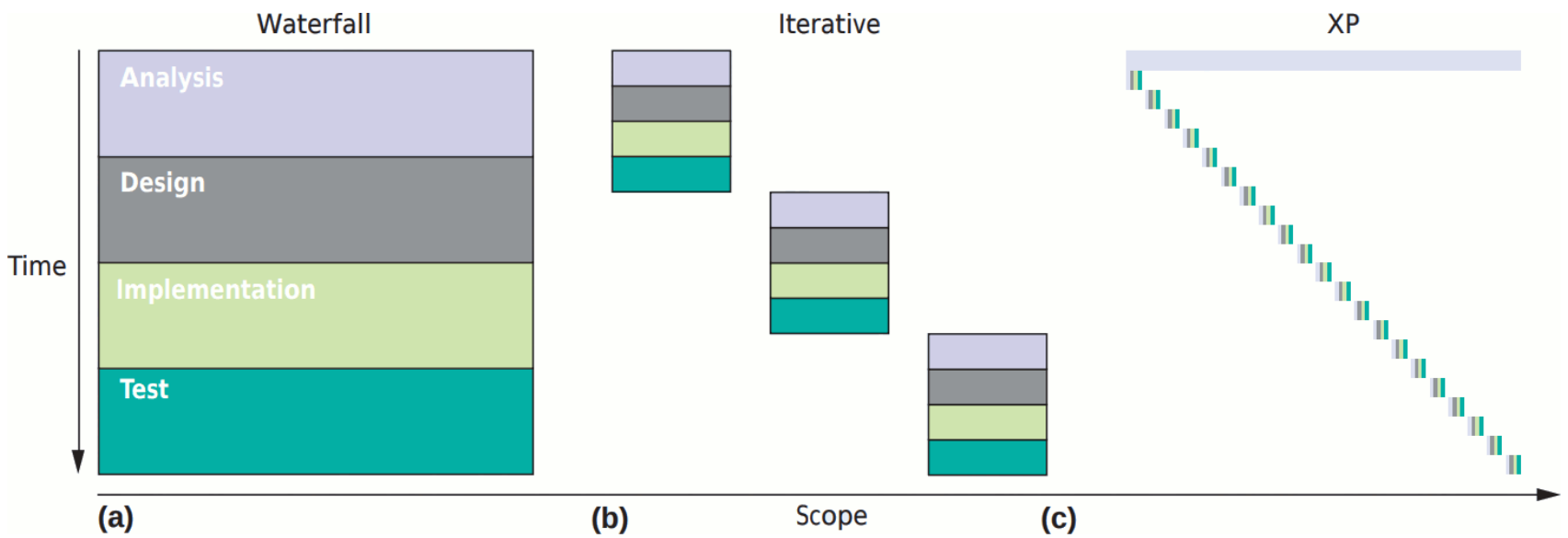
Agile Software Development

- Introduced in Manifesto for Agile Software Development (2001).
  (by Agile Alliance: Alistair Cockburn, Ken Schwaber, Jeff Sutherland et al.)
- The manifesto proclaimed that they value:
  - **Individuals and interactions** over processes and tools
  - **Working software** over comprehensive documentation
  - **Customer collaboration** over contract negotiation
  - **Responding to change** over following a plan
- Recommended developers to follow 12 principles.
  (see slide 17)
- The agile development supported by many practices.
  (agile modelling, Scrum, test-driven development, extreme programming, etc.)

Principles behind the Agile Manifesto

1. Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.
2. Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.
3. Deliver working software frequently, from a couple of weeks to a couple of months, with a preference to the shorter timescale.
4. Business people and developers must work together daily throughout the project.
5. Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.
6. The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.
7. Working software is the primary measure of progress.
8. Agile processes promote sustainable development. The sponsors, developers, and users should be able to maintain a constant pace indefinitely.
9. Continuous attention to technical excellence and good design enhances agility.
10. Simplicity–the art of maximizing the amount of work not done–is essential.
11. The best architectures, requirements, and designs emerge from self-organizing teams.
12. At regular intervals, the team reflects on how to become more effective, then tunes and adjusts its behaviour accordingly.

Waterfall vs. Iterative vs. Agile/XP Process

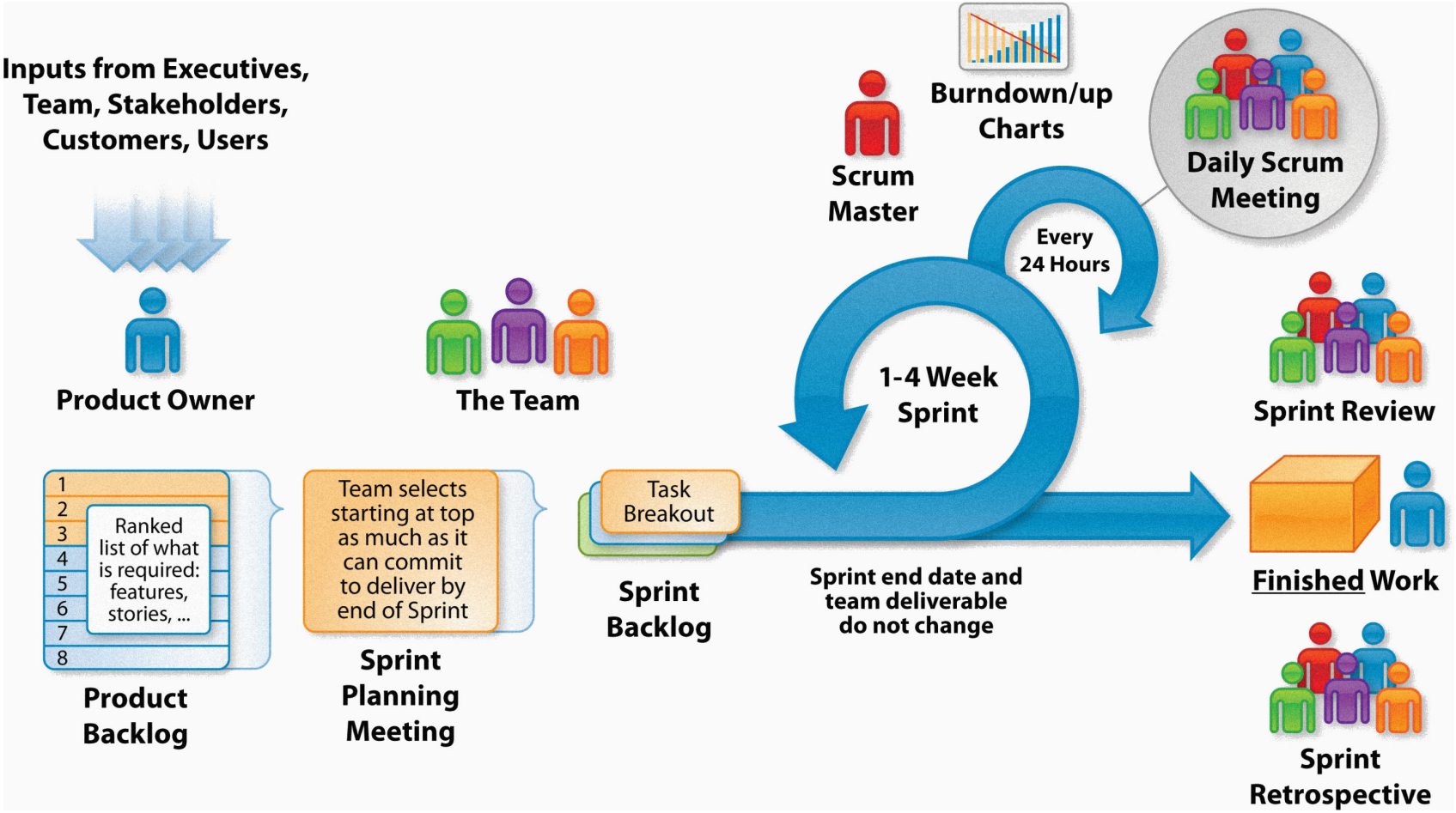


(adopted from “Embracing change with extreme programming” by K. Beck)

Scrum

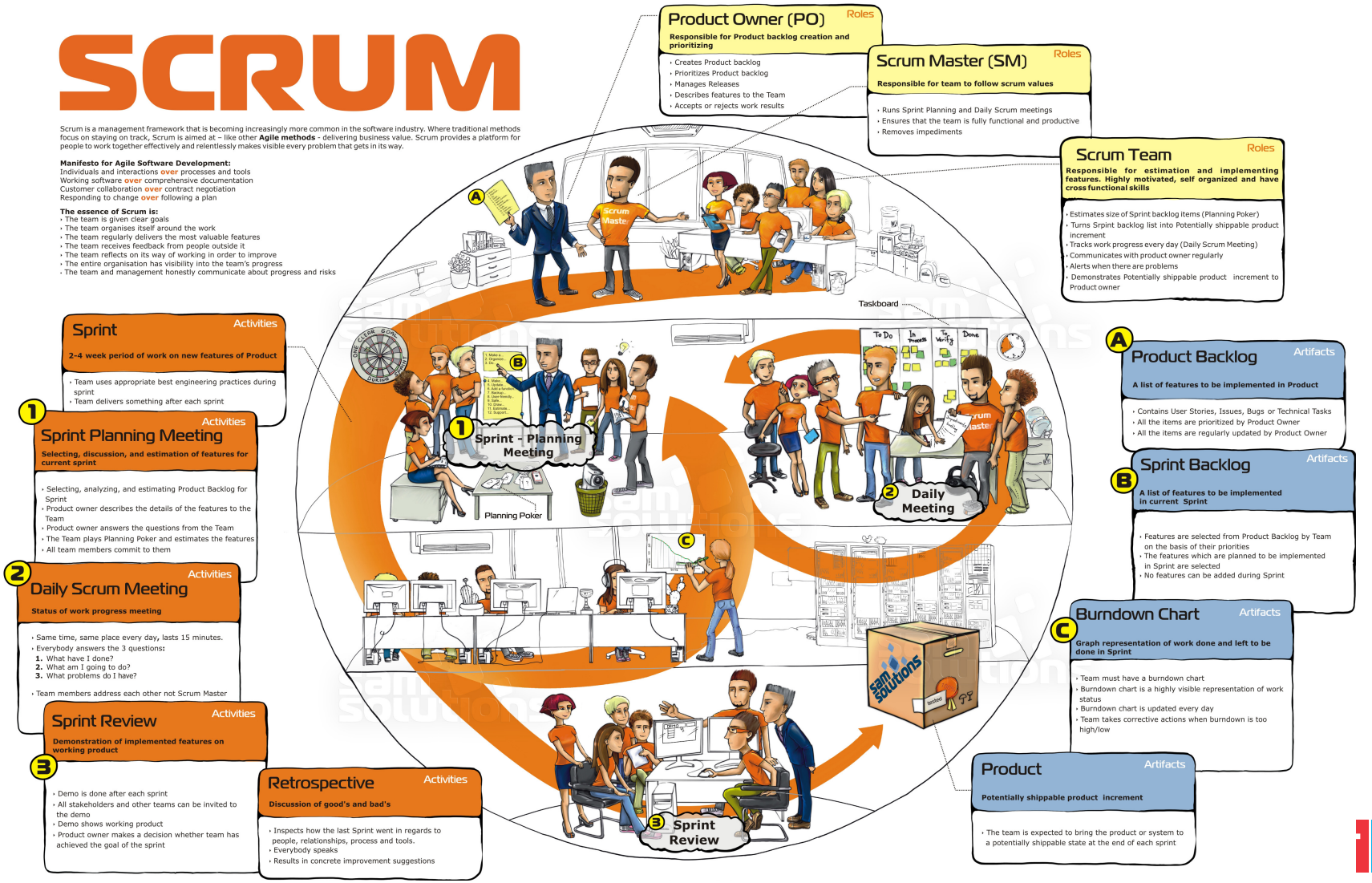
- A framework for (agile) software development management.
  (iterative and incremental, agile because of “requirements volatility” in “backlog”)
- Development in 3 to 8 iterations, each takes 2 to 4 weeks.
  (so called “sprints”; track progress/re-plan in daily 15-minutes stand-up meetings)
- Small self-organising teams of 3 to 9 developers.
  (motivated/helped by “Scrum master” and assisted by “product owner”)
- The object-oriented approach to development responsibilities.
  (a developer is responsible for a particular set of classes/interfaces)
- Risk analysis in sprint review and retrospective meetings.
  (inspecting a result and team/work conditions in the last sprint, respectively)

Scrum Development Process



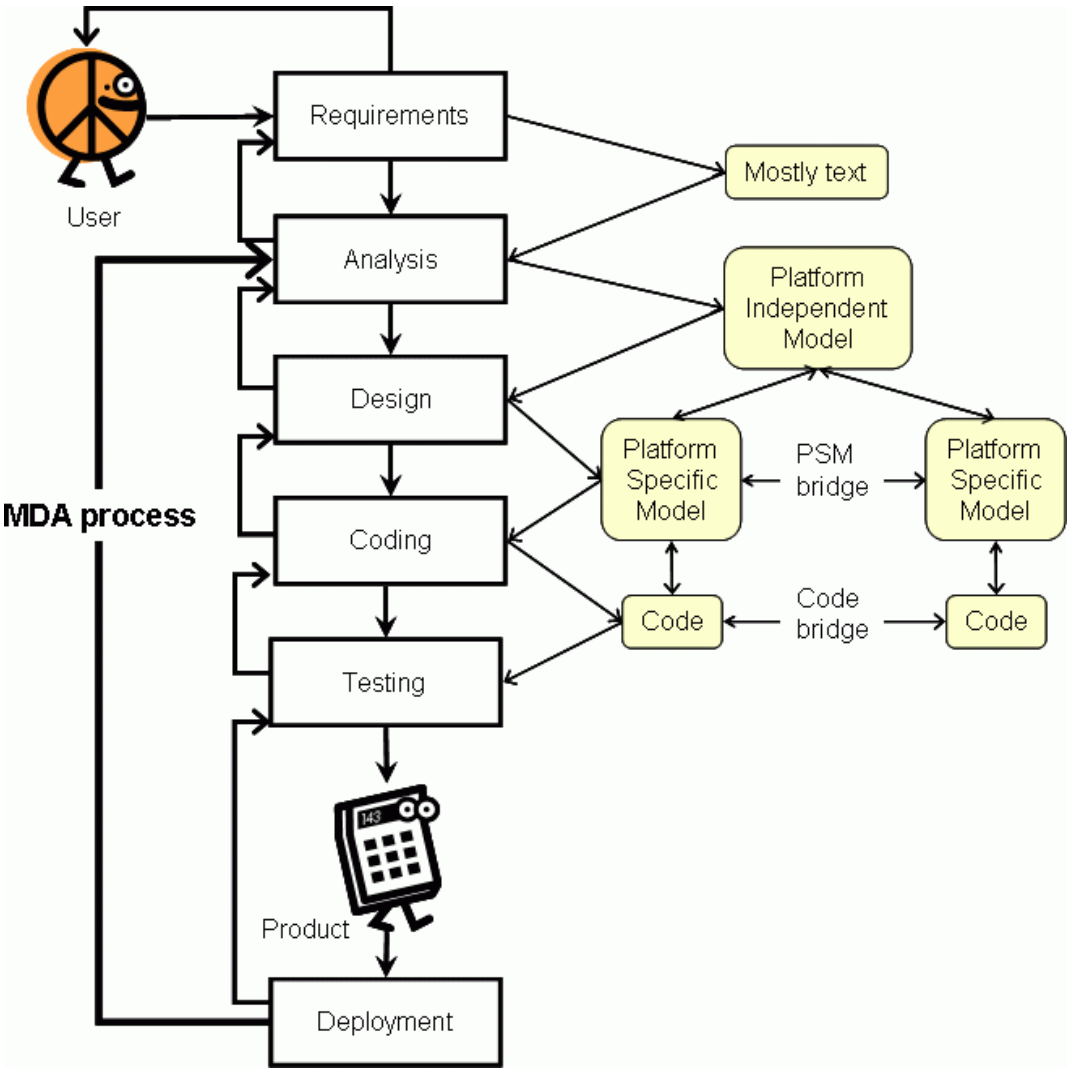
(adopted from “The Agile Scrum Framework at a Glance” by Agile For All)

Scrum Dictionary



(adopted from “Agile Approach” by SaM Solutions)

Model Driven Development/Architecture



(adopted from “Practical Software Engineering” by Maciaszek&Liong)

Computation/Platform Independent/Specific Models

CIM Computation Independent Model
(domain models useful to a variety of stakeholders, e.g., to set up a vocabulary)

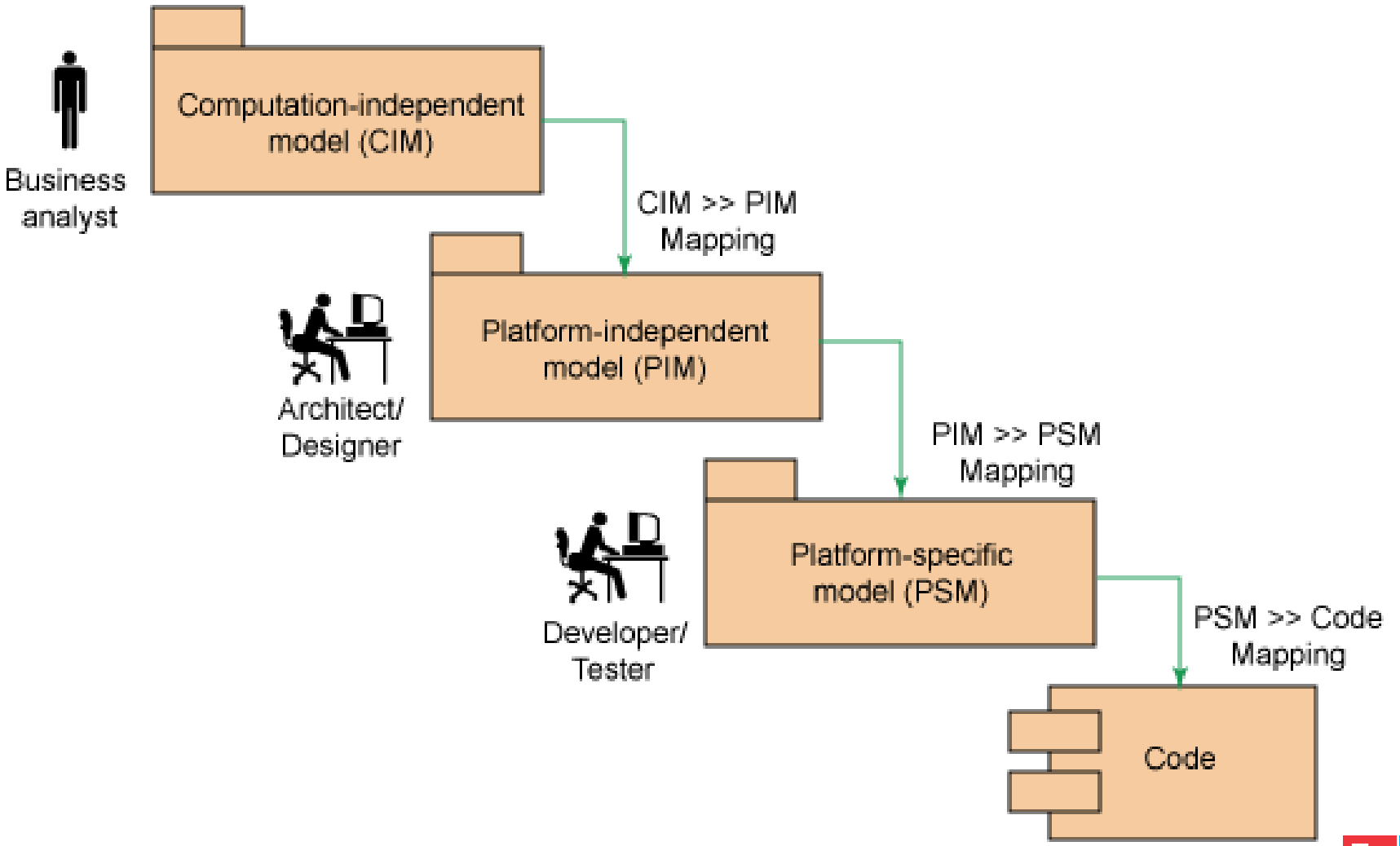
PIM Platform Independent Model
(specification/design of a system without technical details for particular platforms)

PSM Platform (or Framework) Specific Model
(specification/design of the system in terms of a particular platform)

code a particular implementation in a (platform native) source code

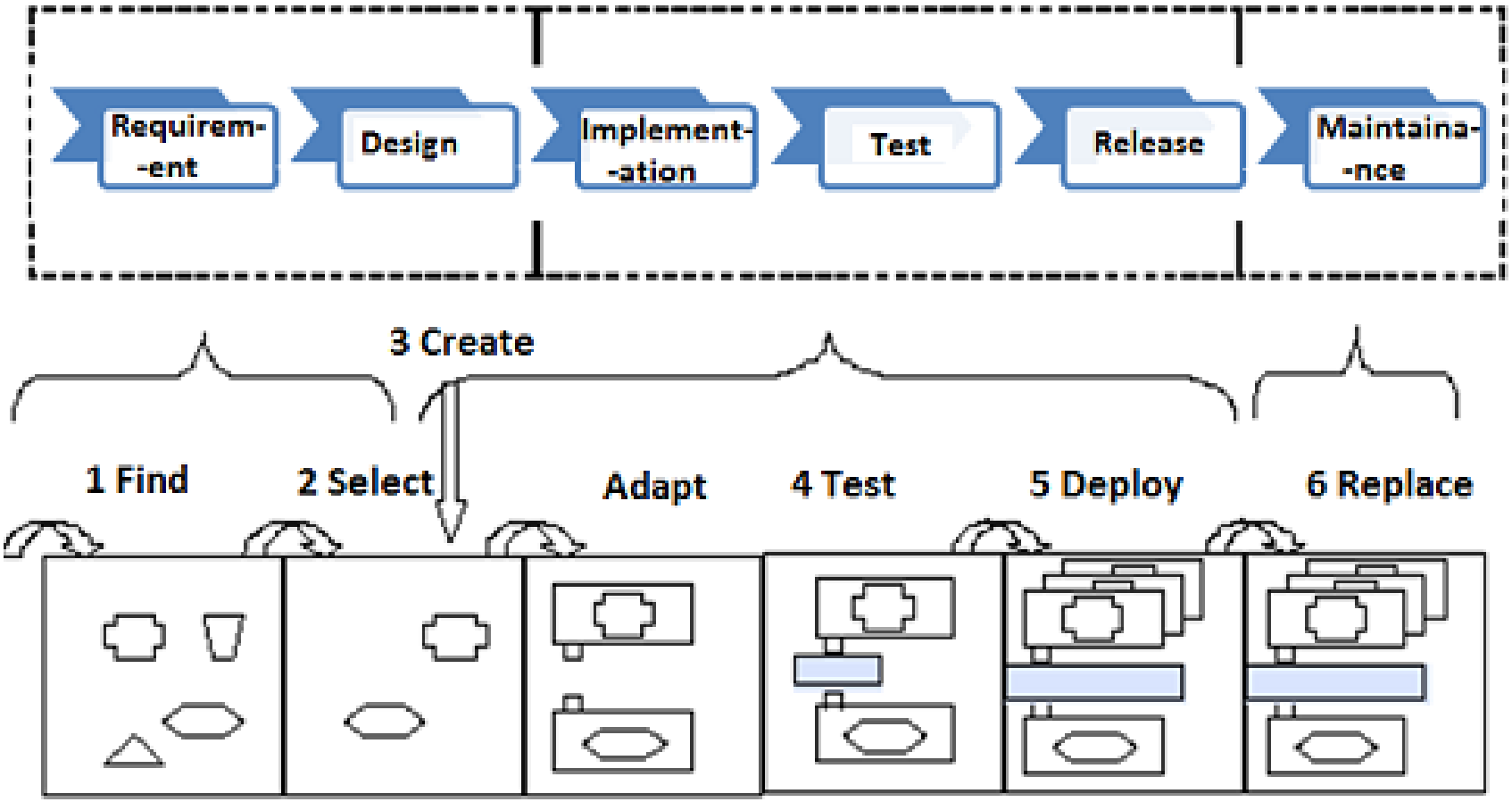
Model-to-model (M2M) and model-to-text (M2T) transformations.
(to transform models or for code-generation, respectively; automatic and bidirectional)

Roles in MDA Models



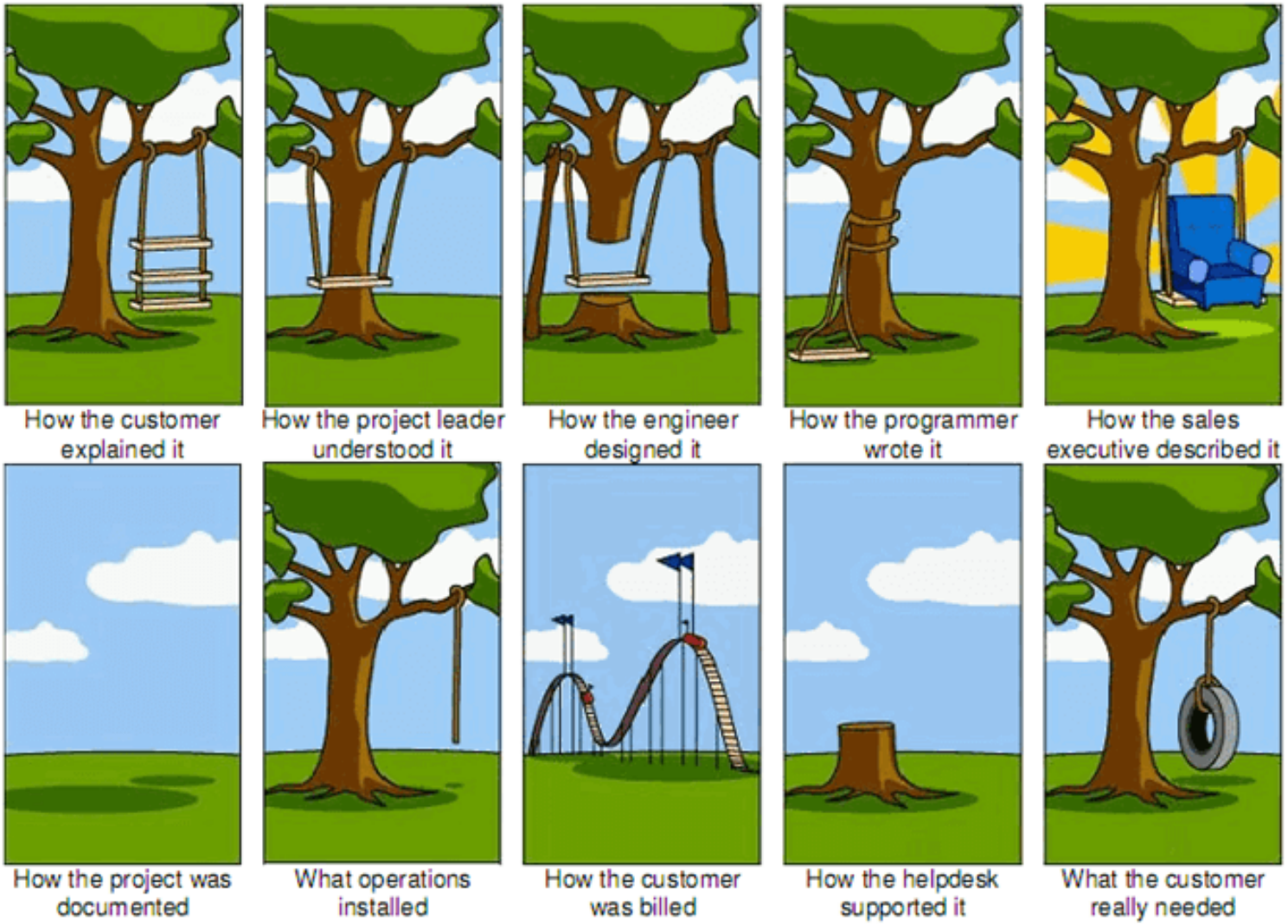
(adopted from “Model transformations” by Paolo Ciancarini)

Component Based Development Process



(adopted from “Reuse Based Software Engineering” by H. Mili et al.)

The Project Management Tree Swing Cartoon



(adopted from “The Project Management Tree Swing Cartoon, Past and Present” by Jewel Ward)

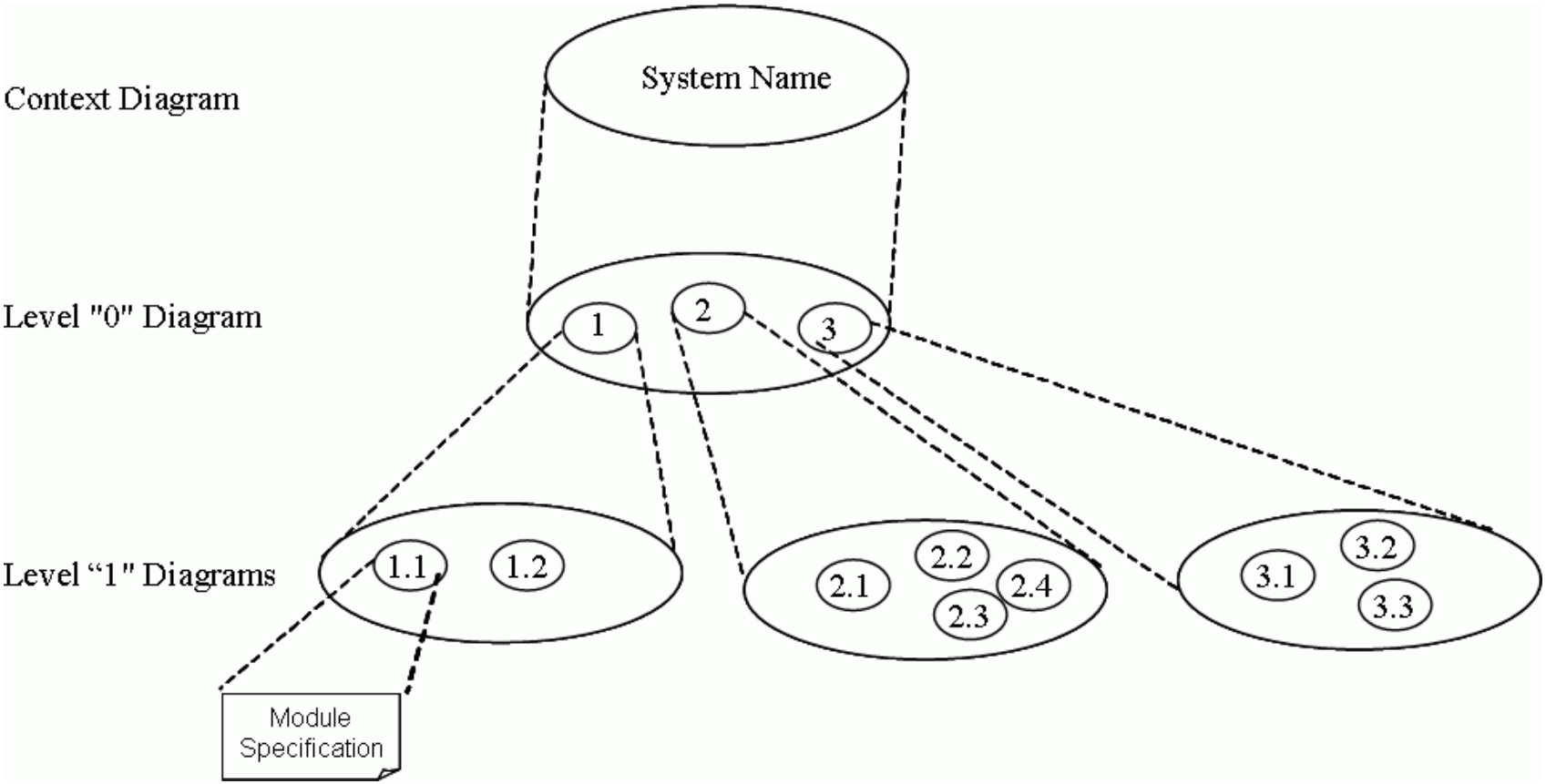
Software Design

> *The fundamental problem is that designers are obliged to use current information to predict a future state that will not come about unless their predictions are correct.*
>
> — *John Chris Jones: Design Methods (1970)*

Structured methods – software broken into functions and subroutines.
(only a single entry point and a single exit point for any function or routine)

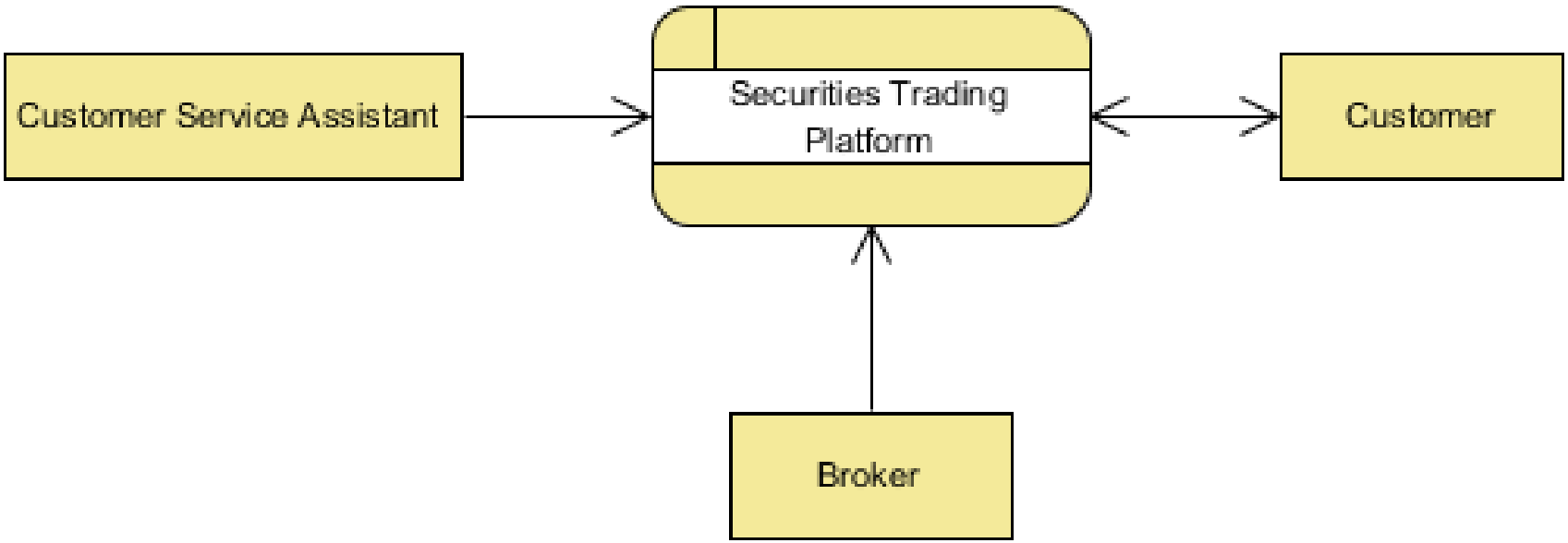
- a data-flow model (DFD)
- an entity-relationship model (ERD)
- a structural model
- an object-oriented model

Structured Modelling



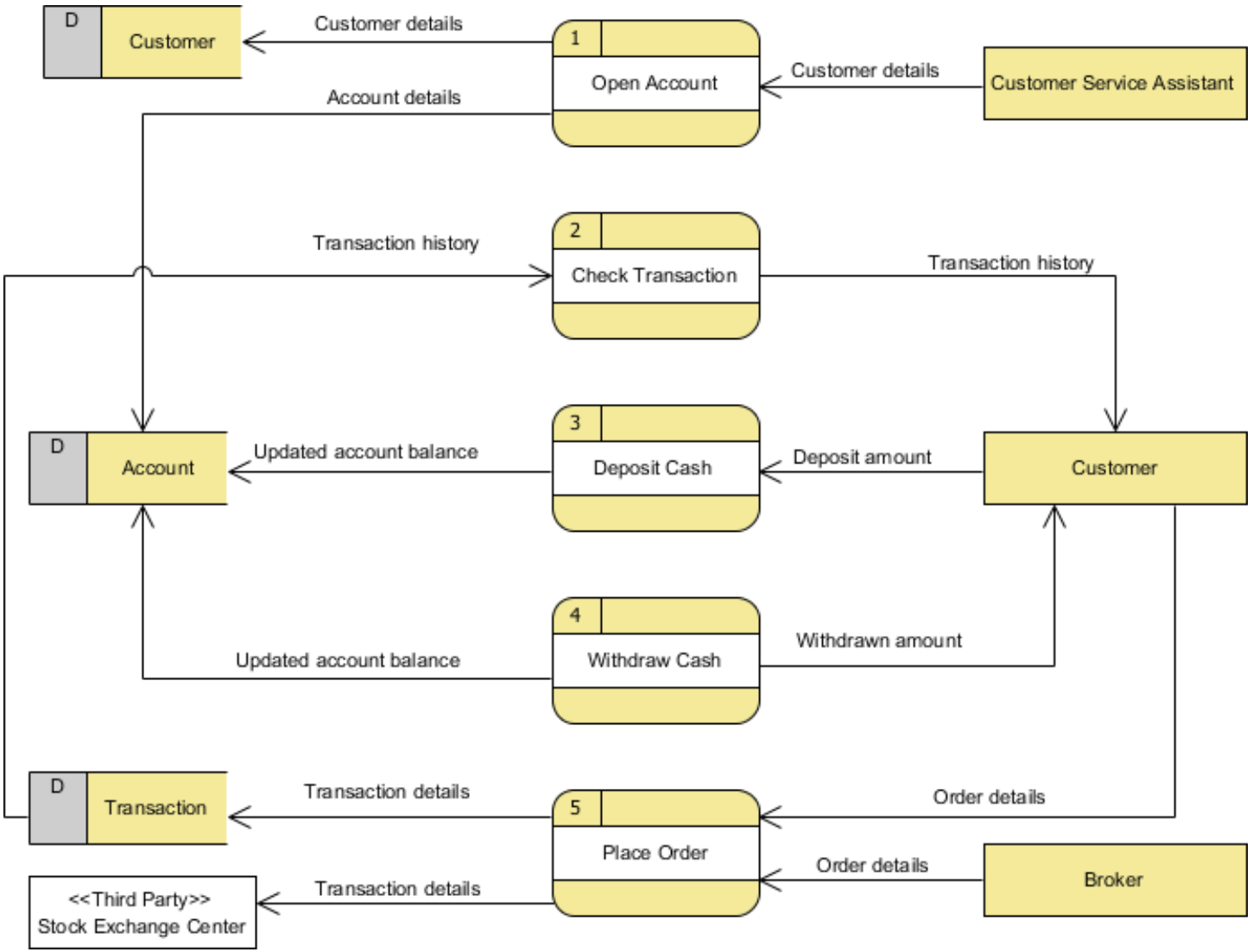
(adopted from "Practical Software Engineering" by Maciaszek&Liong)

Example: Securities Trading Platform – Context DFD



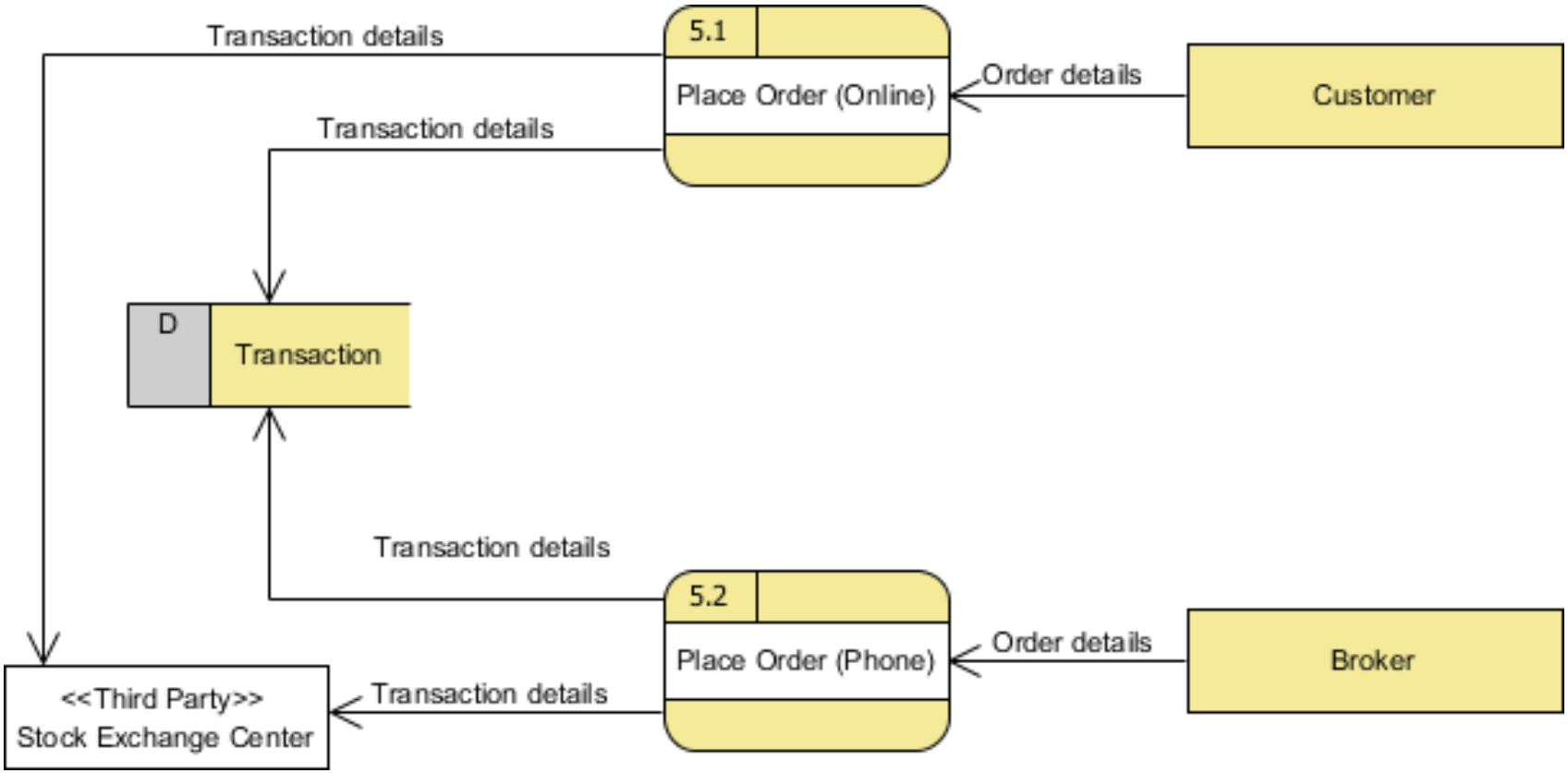
(adopted from "DFD with Examples - Securities Trading Platform" by Visual Paradigm)

Example: Securities Trading Platform – Level 1 DFD



(adopted from "DFD with Examples - Securities Trading Platform" by Visual Paradigm)

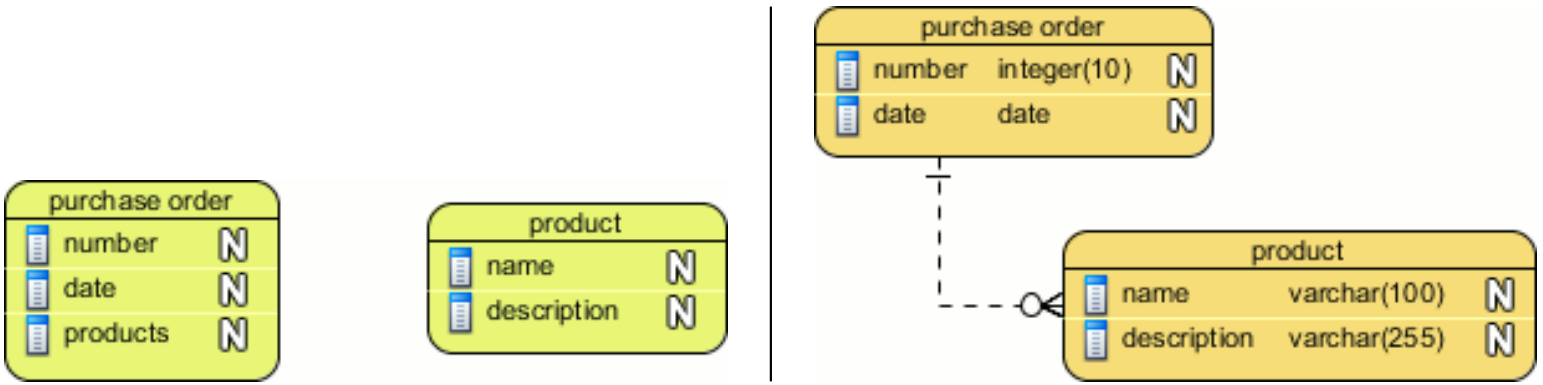
Example: Securities Trading Platform – Level 2 DFD

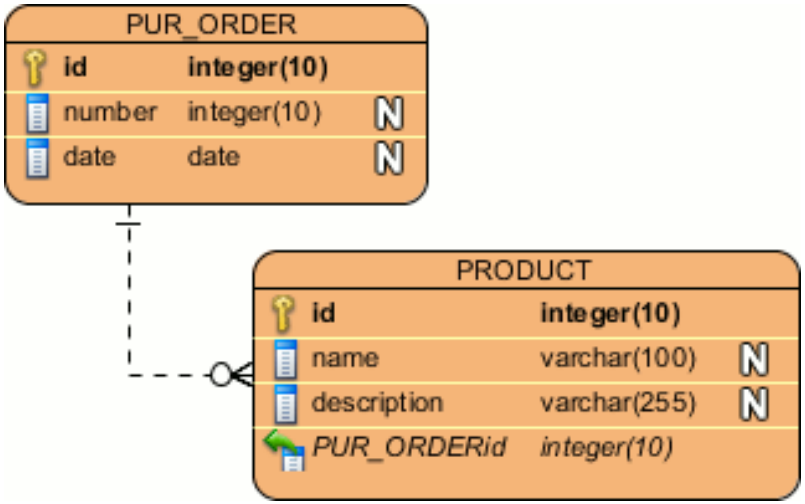


(adopted from "DFD with Examples - Securities Trading Platform" by Visual Paradigm)

Example: Order Processing System – ERD

conceptual, logical, and physical





(adopted from "Entity Relationship Diagram" by Visual Paradigm)

Object-Oriented Design

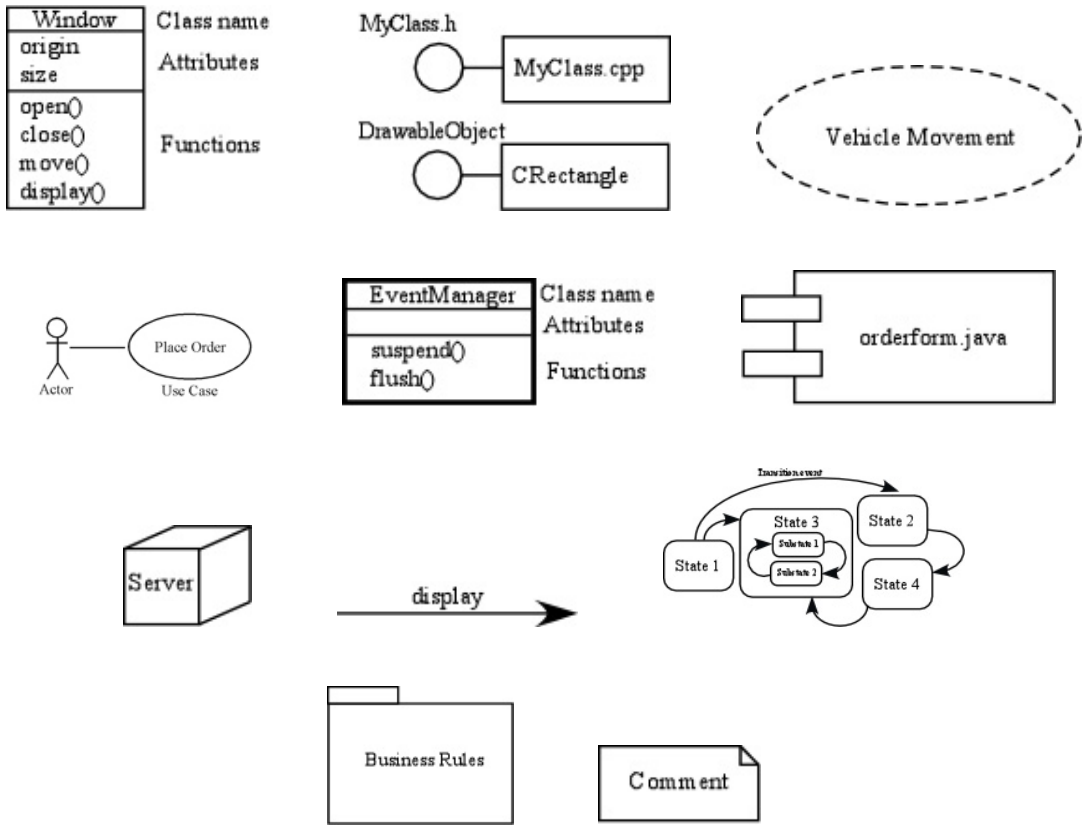
- It is based on object-oriented approach.
  (the idea of information hiding, i.e., encapsulation)
- A system is viewed as a set of interacting objects.
  (each object has its own private state and relationships to other objects)
- The objects communicate by calling on methods/services offered by other objects on their well-defined interfaces.
  (rather than by sharing variables, e.g., attribute values)
- The message passing model allows objects
  - to have well-defined interfaces and dependencies,
  - to be implemented as concurrent processes,
  - to reduce the overall system coupling.
- It is a dominant design strategy for new software systems.

Unified Modelling Language (UML)

- A modelling language for specification, visualisation, development, and documentation of software systems artefacts.
  (however, it is a general-purpose language, suitable also for modelling of enterprise processes, hardware systems, network infrastructure, etc.)
- UML version 2.5 (2015) extending UML 2.0 (2005) and later.
  (UML 1.5 from 2003 is deprecated)
- The conceptual model of UML defines three major elements:
  - UML building blocks (things, relationships, diagrams);
  - rules to connect the building blocks;
  - common mechanisms of UML.

UML Things

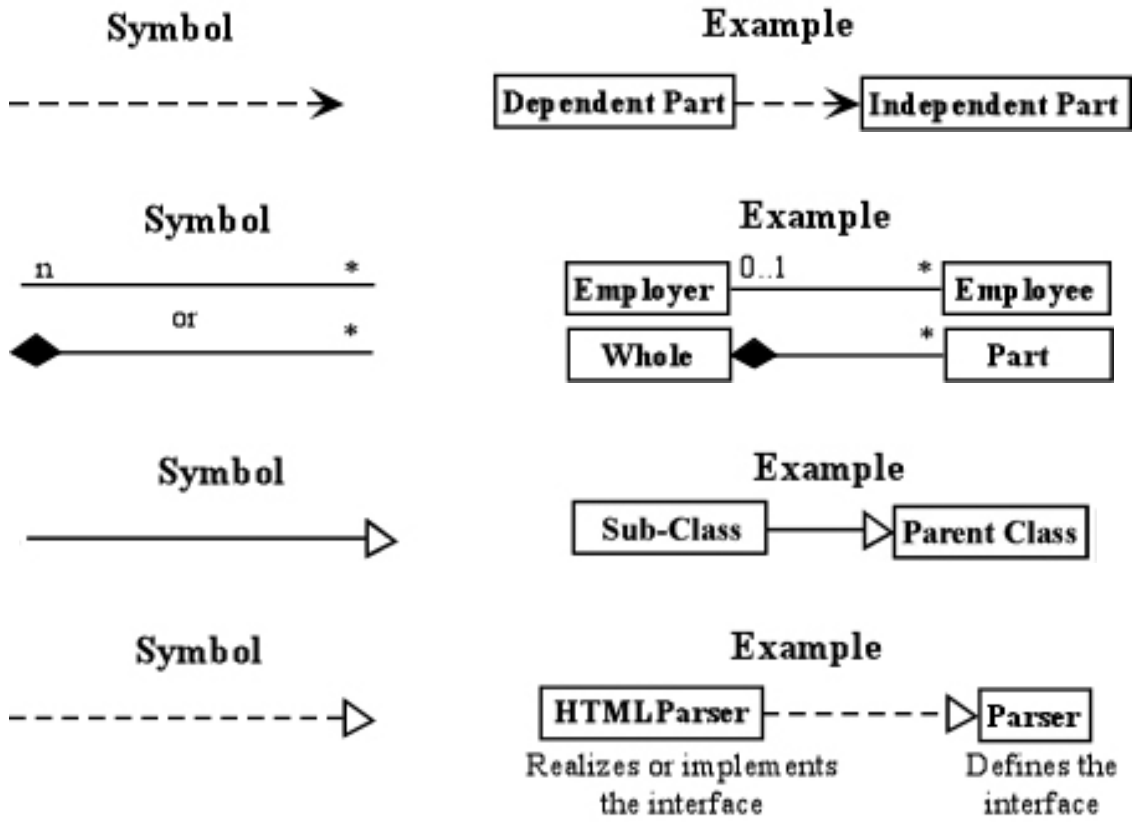
- Structural Things
  - class
  - interface
  - collaboration
  - use case
  - active class
  - component
  - node
- Behavioural Things
  - interaction
  - state machine
- Grouping Things
  - package
- Annotation Things
  - note



(adopted from "Unified Modeling Language" by Rick Coleman)

UML Relationships

- Dependency
- Association
- Generalisation
- Realisation



(adopted from "Unified Modeling Language" by Rick Coleman)

UML Diagrams

- Diagrams for modelling static structure:
  - Class diagram
  - Component diagram
  - Composite structure diagram
  - Deployment diagram
  - Object diagram
  - Package diagram
  - Profile diagram
- Diagrams for modelling dynamic behaviour:
  - Activity diagram
  - Interaction diagrams
    - Sequence diagram
    - Communication diagram
    - Timing diagram
    - Interaction overview diagram
  - State diagram
  - Use case diagram

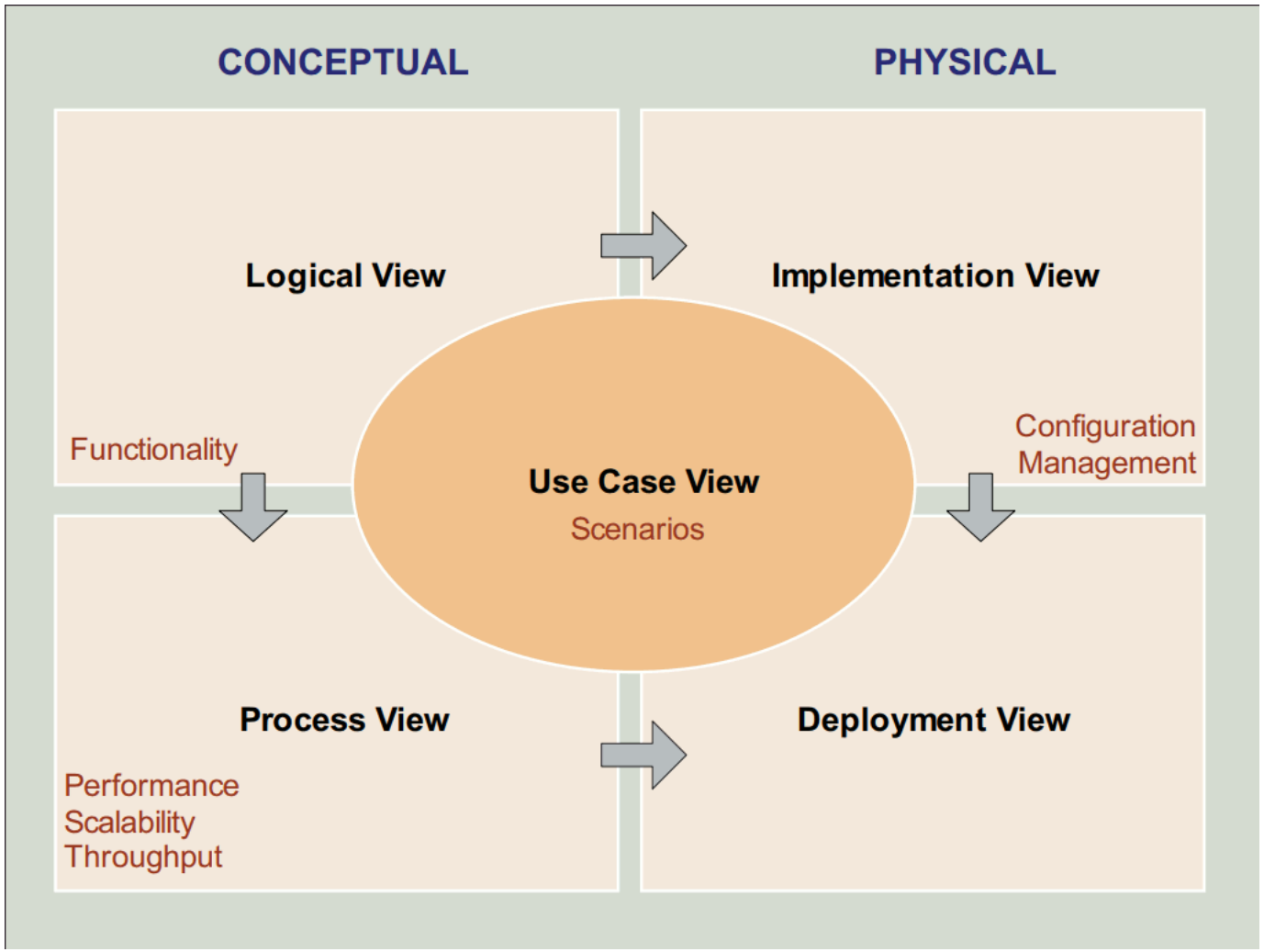
UML Common Mechanisms

- Specifications
  (every part of the graphical notation has a specification for a textual statement)
- Adornments
  (for rendering constraints attached to an element in a note)
- Common divisions
  - class vs. object
    (a class is an abstraction, an object is its one concrete manifestation)
  - interface vs. implementation
    (it describes provided services only, or their internal implementations)
- Extensibility mechanisms
  - stereotypes
    (to specify particular semantics of UML elements)
  - tagged values
    (to extend the properties of UML building blocks by additional information)
  - constraints (OCL)
    (to add new rules or to modify existing ones for particular UML elements)

UML and System Architecture

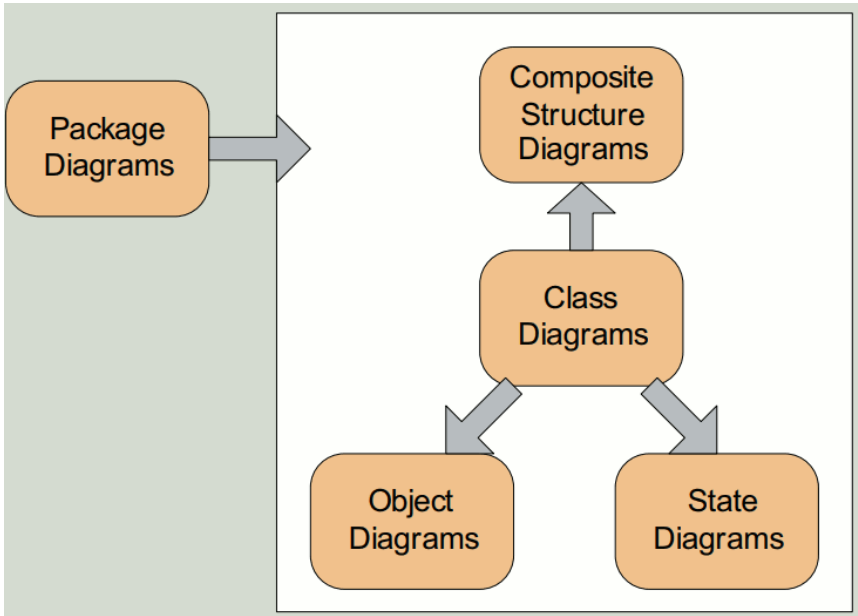
- An architecture is decomposition and organization of components.
  (including interconnections, interactions, mechanisms, and design principles)
- UML compatible with Kruchten's 4+1 View Model of Architecture.
  - Logical view
    (concerned with the functionality that a system provides to end-user)
  - Process view
    (deals with the dynamic aspects/run-time behavior of the system)
  - Physical/Deployment view
    (the topology&connections of SW components on the physical layer)
  - Development/Implementation view
    (illustrates a system from a programmer's perspective; SW management)
  - Scenarios/Use Case view
    (sequences of interactions between objects and between processes)

The 4+1 View Model



(adopted from "Applying 4+1 View Architecture with UML 2" by FCGSS)

Logical View with UML 2



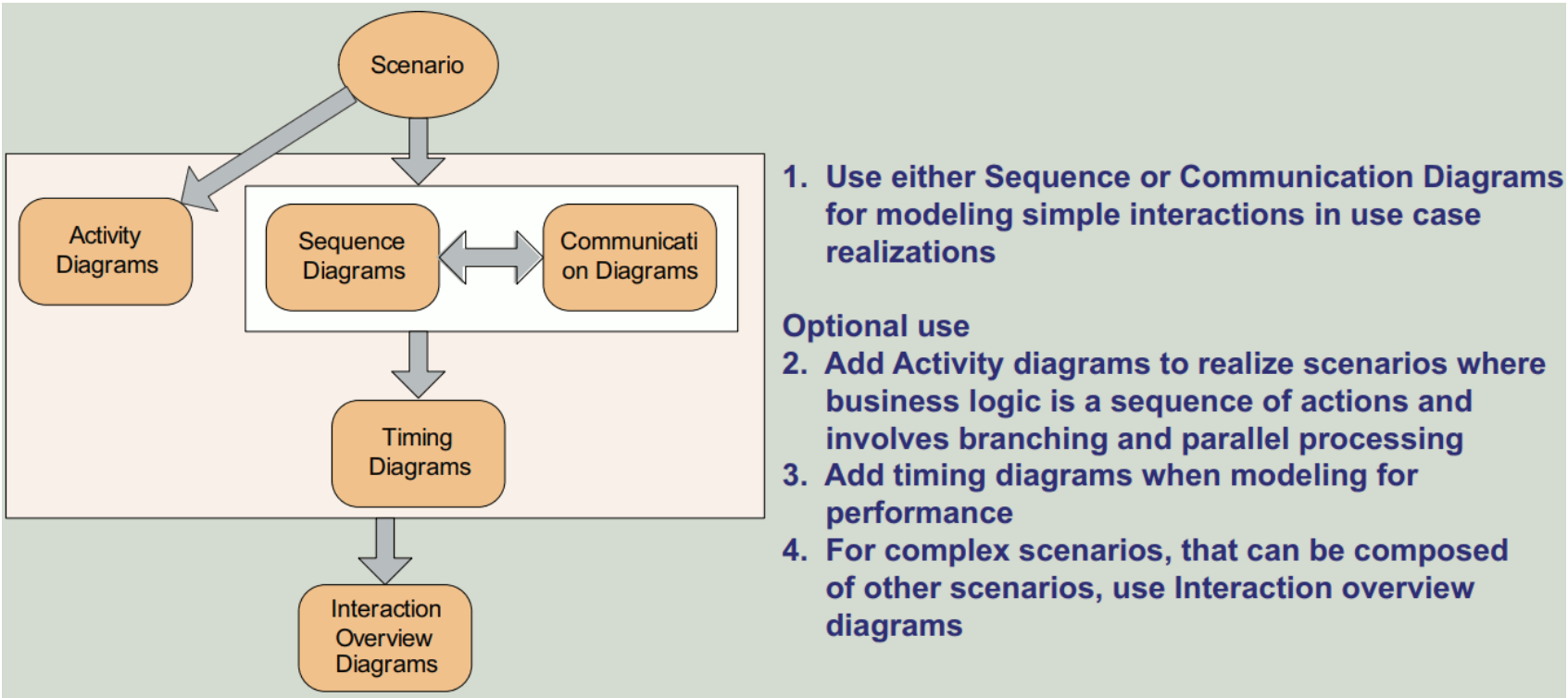
1. **Start with class diagrams to model the system**
2. **Use package diagrams to logically group diagrams**

Optional use

3. **Object diagrams when relationships between classes need to be explained through instances**
4. **State Charts when internal states of a specific class are to be explained**
5. **Composite Structures when parts of a class and relationships between parts are to be modeled**

(adopted from "Applying 4+1 View Architecture with UML 2" by FCGSS)

Process View with UML 2



(adopted from "Applying 4+1 View Architecture with UML 2" by FCGSS)

Implementation and Deployment Views with UML 2

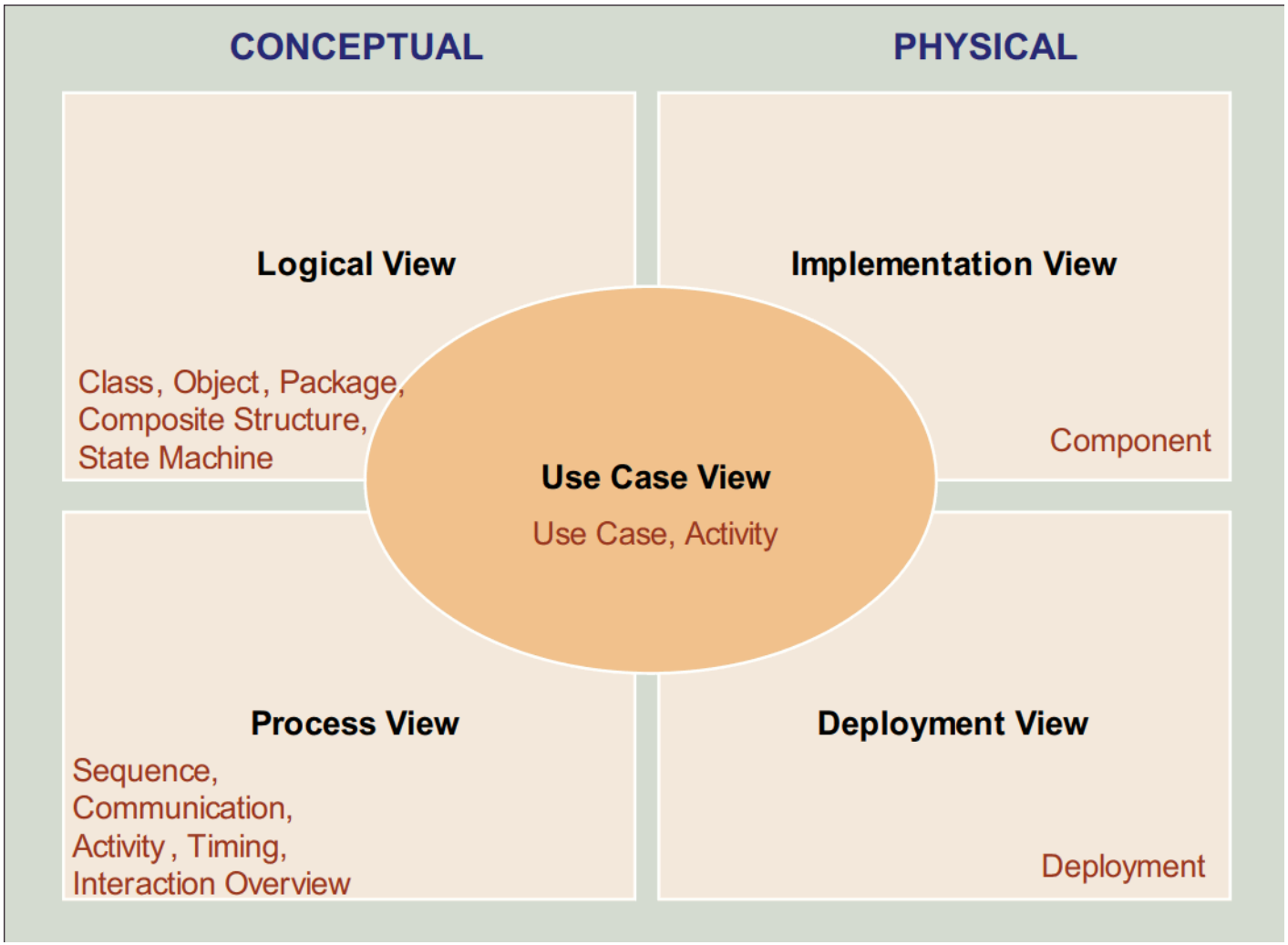
Implementation View = Subsystem Decomposition

- a view of a system's architecture
  (it encompasses the components used to assemble and release a physical system)
- focuses on configuration management and actual SW module organization
- the diagrams represent the physical-level artifacts that are built

Deployment View = Mapping Software to Hardware

- a view of the system's hardware topology on which the system executes
  (it encompasses a topology of hardware nodes and software deployment)
- focuses on distribution, communication and provisioning
- the diagrams show the physical disposition of the artifacts in real-world

UML Models in 4+1 View Architecture



(adopted from "Applying 4+1 View Architecture with UML 2" by FCGSS)

Example of OOA/D: Dice Game

(adopted from "Applying UML and Patterns" by Craig Larman)

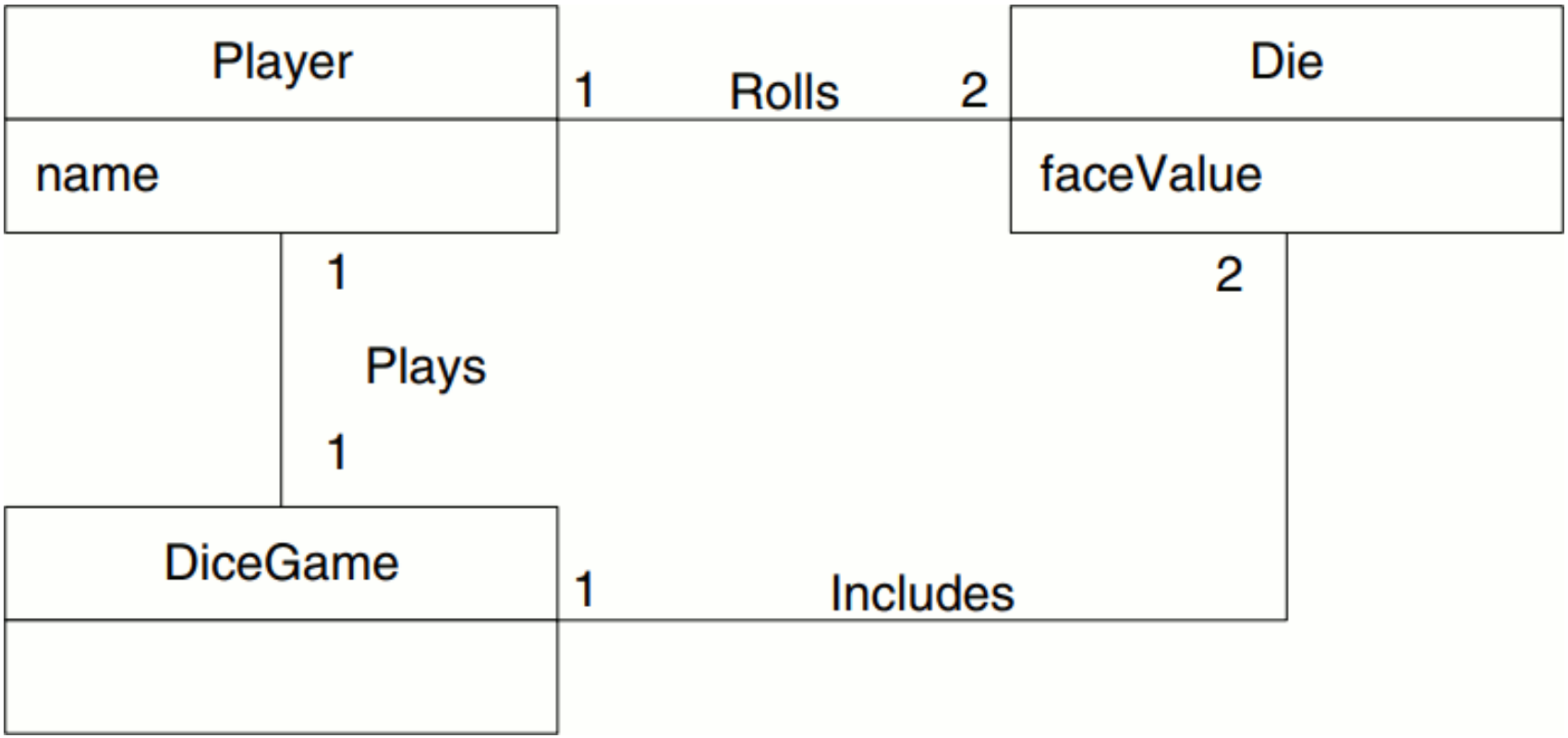
- A "dice game" in which software simulates a player rolling two dice. If the total is seven, they win; otherwise, they lose.
- Object-oriented analysis and design process (OOA/D):
  1. Define use cases
  2. Define domain model
  3. Define interaction diagrams
  4. Define design class diagrams

1st Step: Define Use Cases

- Use cases are simply written stories (not an OO artifact).
- "Play a Dice Game" use case:

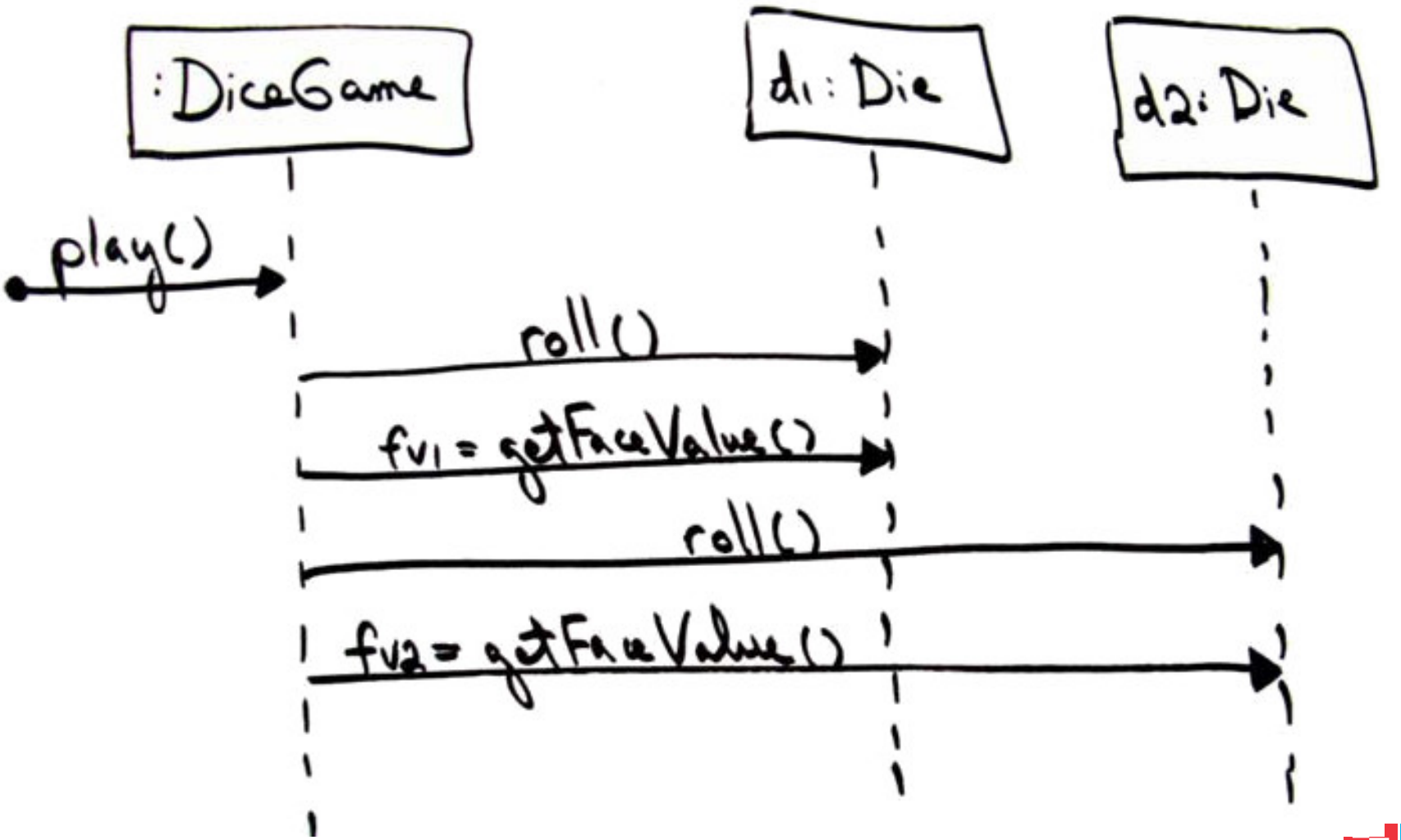
  Player requests to roll the dice. System presents results: If the dice face value totals seven, player wins; otherwise, player loses.

Dice Game – 2nd Step: Define Domain Model



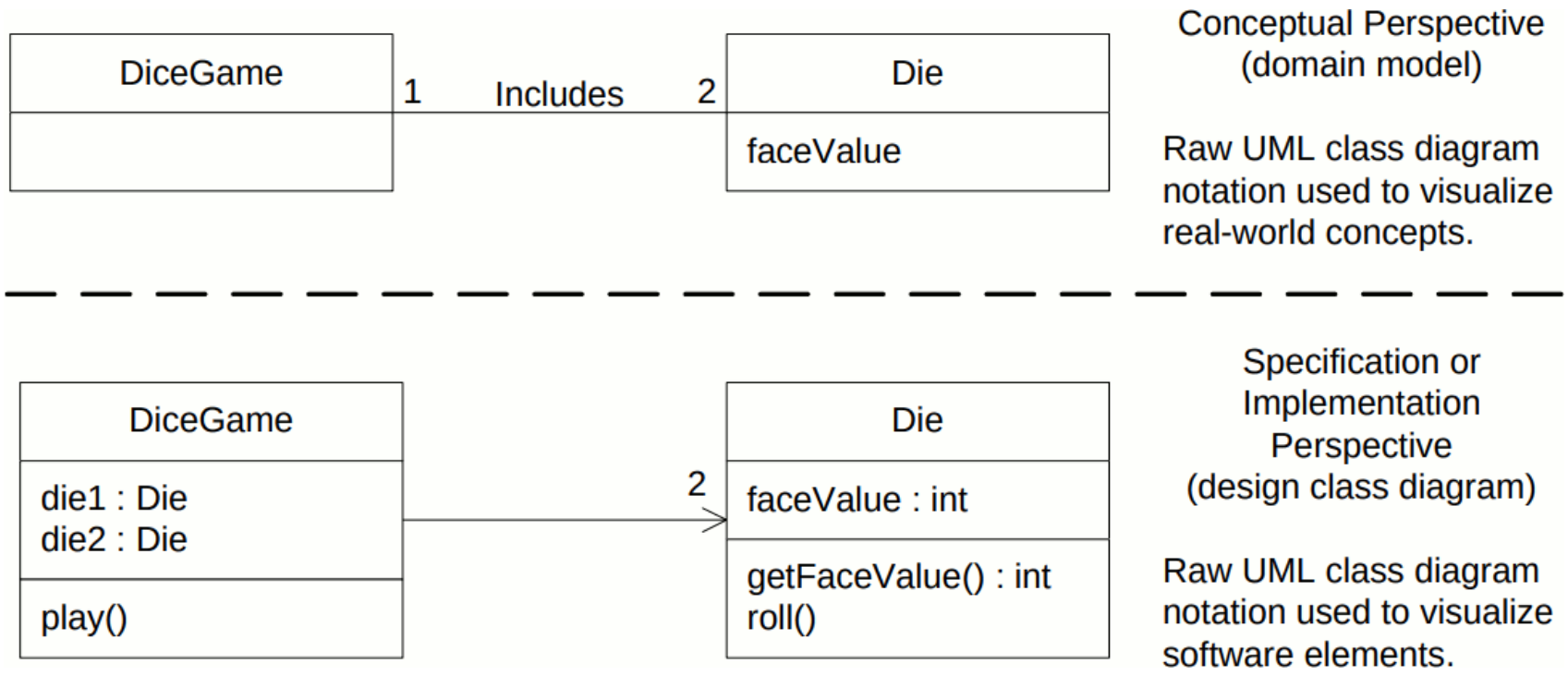
(adopted from "Applying UML and Patterns" by Craig Larman)

Dice Game – 3rd Step: Define Interaction Diagrams



(adopted from "Applying UML and Patterns" by Craig Larman)

Dice Game – 4th Step: Define Design Class Diagrams



(adopted from "Applying UML and Patterns" by Craig Larman)

Why to implement the application logic in "DiceGame" and not in "Player"?

Thank you for your attention!

Marek Rychlý
<rychly@fit.vutbr.cz>