Software Development Methodologies

Lecturer: **Raman Ramsin**

Lecture 4

Seminal Object-Oriented Methodologies:

A Feature-Focused Review **(Part 2)**

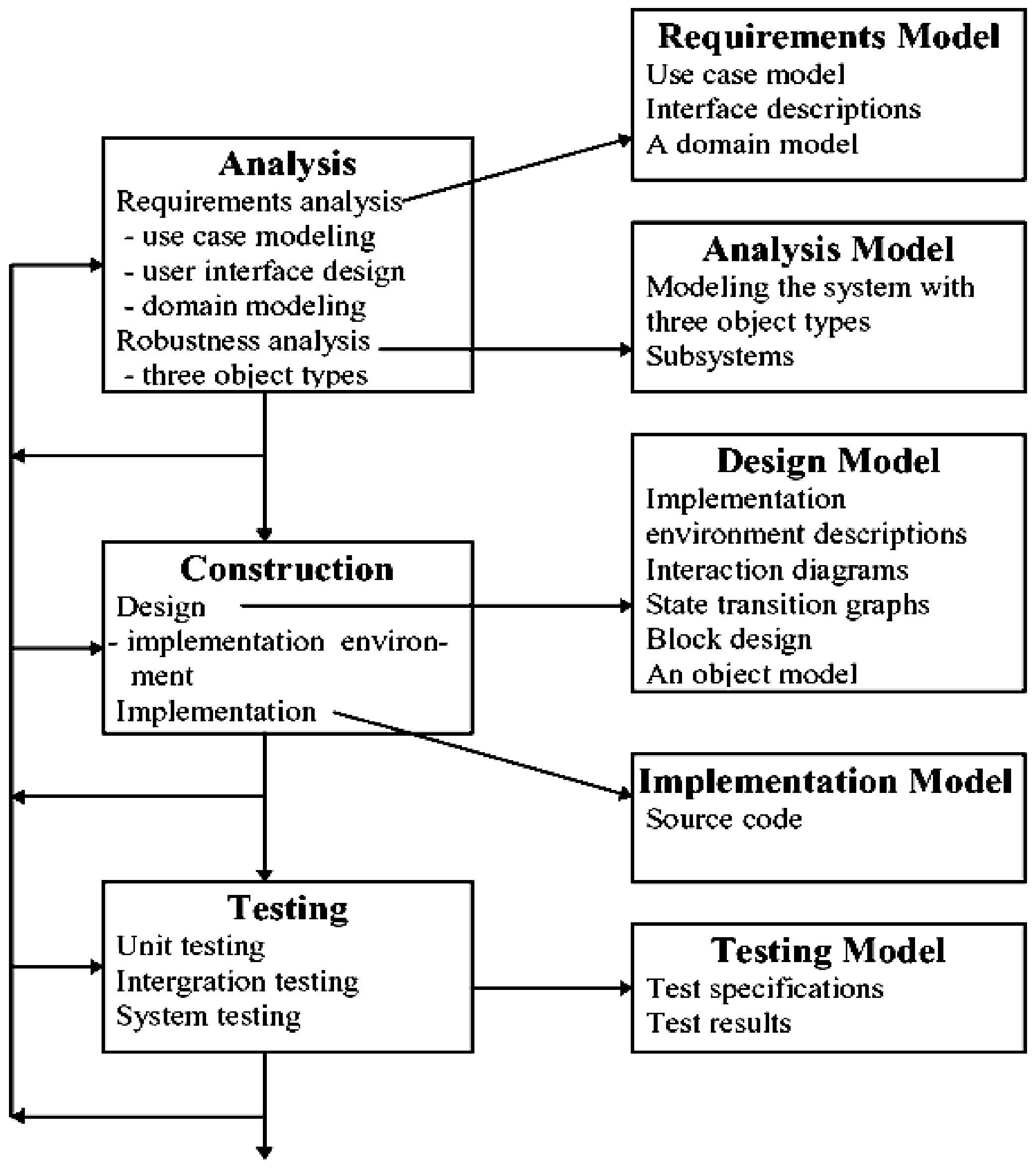
Object-Oriented Software Engineering (OOSE)

- First introduced by Jacobson et al. in 1992
- A simplified version of Jacobson's *Objectory* methodology, first introduced in 1987 and later the property of Rational Corporation (now acquired by IBM)
- Covers the full generic lifecycle

OOSE: Process

- *Analysis:* focusing on understanding the system and creating a conceptual model. Consists of two non-sequential, iterative subphases:
  - *Requirements Analysis*, aiming at eliciting and modeling the requirements of the system. A *Requirements Model* is produced.
  - *Robustness Analysis,* aiming at modeling the structure of the system. An *Analysis Model* is produced.
- *Construction:* focusing on creating a blueprint of the software and producing the code. Consists of two subphases:
  - *Design*, aiming at modeling the run-time structure of the system, and also the inter-object and intra-object behaviour. A *Design Model* is produced.
  - *Implementation*, aiming at building the software. An *Implementation Model* (including the code) is produced.
- *Testing:* focusing on verifying and validating the implemented system. A *Test Model* is produced.

OOSE: Process



[Jacobson et al. 1992]

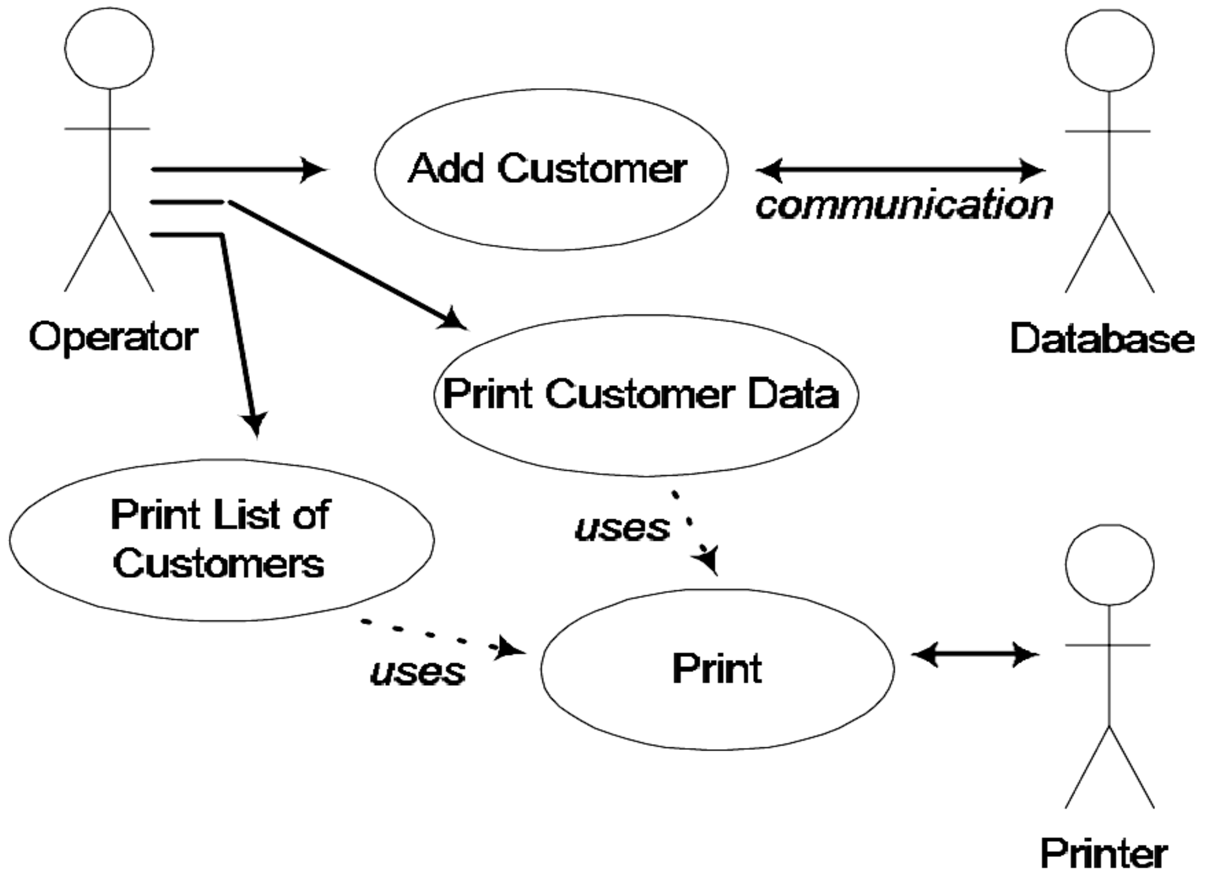
OOSE: Analysis – Requirements Analysis

- Aim: Specify and model the functionality required of the system, typical means and forms of interacting with the system, and the structure of the problem domain.

- The model to be developed is the *Requirements Model*, further divided into three submodels:
  - A *Use Case Model:* which delimits the system and describes the functional requirements from the user's perspective.
  - A *Domain Object Model:* consists of objects representing entities derived from the problem domain, and their *inheritance, aggregation* and *association* relationships.
  - *Interface Descriptions:* provide detailed logical specifications of the user interface and interfaces with other systems.

OOSE: Analysis - Requirements Model

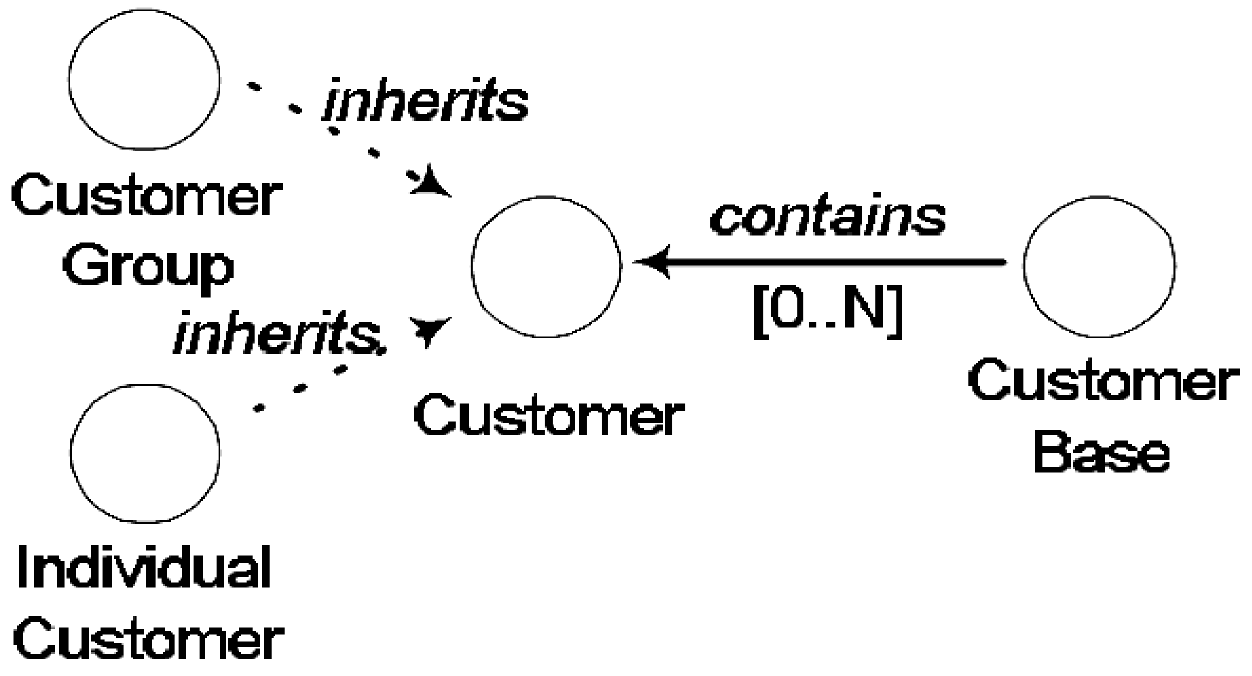
Use Case Model



[Jacobson et al. 1992]

OOSE: Analysis - Requirements Model

Domain Object Model



[Jacobson et al. 1992]

OOSE: Analysis – Robustness Analysis

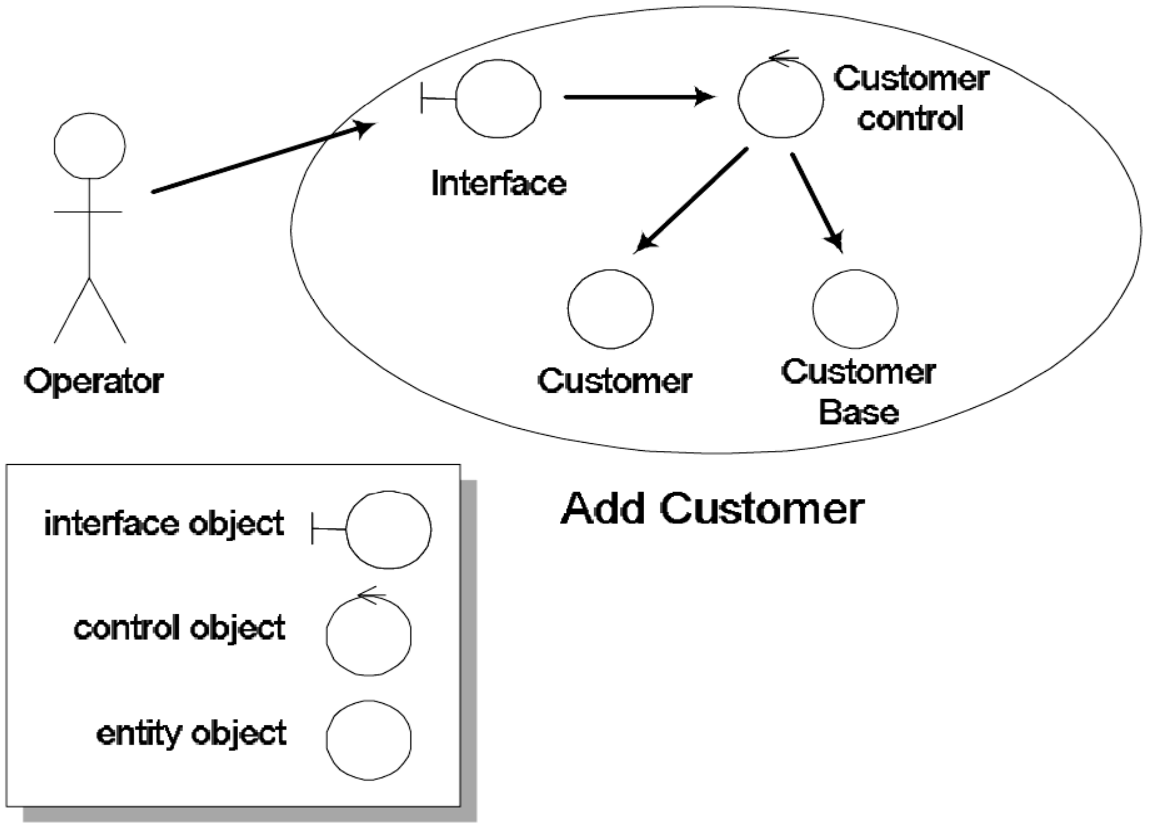
- Aim: Map the *Requirements Model* to a logical configuration of the system that is robust and adaptable to change.

- The model to be developed is the *Analysis Model:*
  - Shows how the functionality of each and every use case is realized by collaboration among typed objects (called *Analysis Objects*).
  - Shows the subsystems of the system.

- *Analysis Objects* can be of three types:
  - *Entity*: Represent entities with persistent state, typically outliving the use cases they help realize. They are usually derived from the domain object model.
  - *Interface*: Represent entities that manage transactions between the system and the actors in the outside world.
  - *Control*: Represent functionality not inherently belonging to other types of objects. They typically act as controllers or coordinators of the processing going on in the use cases.

OOSE: Analysis - Analysis Model

Analysis Model



[Jacobson et al. 1992]

OOSE: Construction - Design

- Aim: Refine the *Analysis Model* by taking into account implementation features.

- The model to be developed is the *Design Model:*
  - describes the features of the implementation environment
  - describes the details of the design classes (referred to as *blocks*) necessary to implement the system
  - describes the way run-time objects should behave and interact in order to realize the use cases

OOSE: Construction - Design

- Three Subphases:

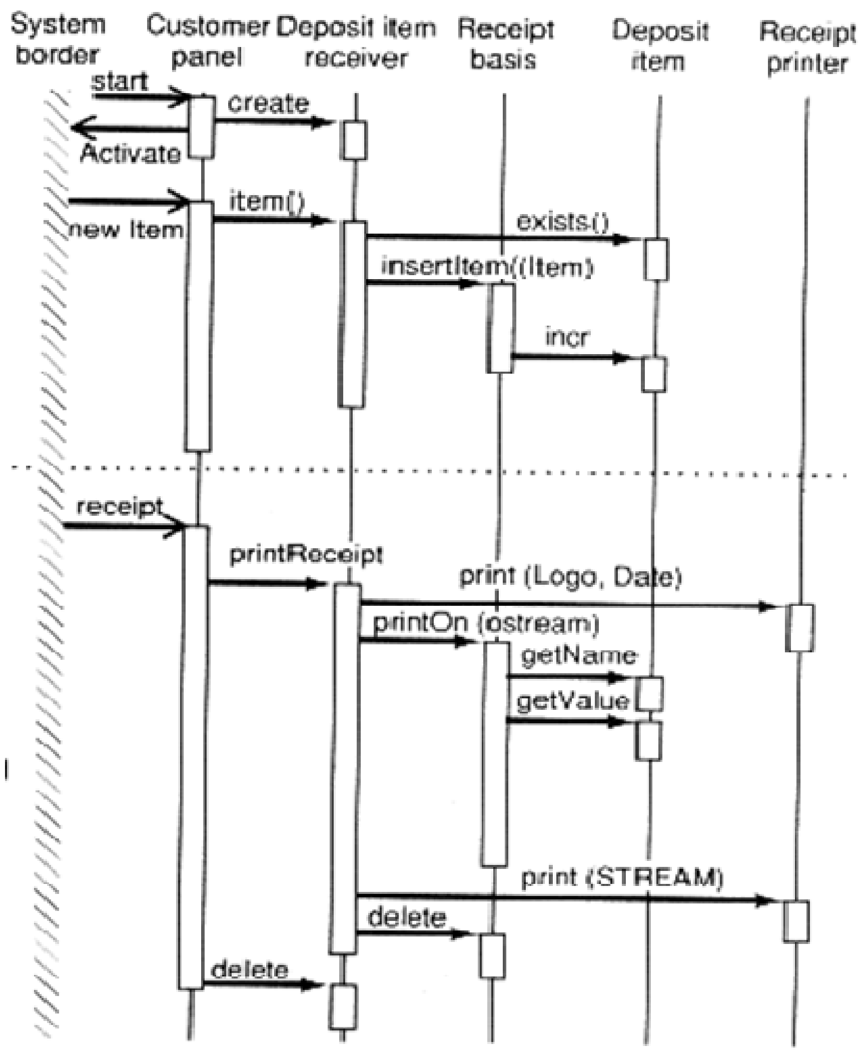
  1. Determination of the features of the implementation environment (DBMS, programming language features, distribution considerations,...)

  2. Definition of *blocks* (design classes) and their structure:
     1. Each object in the Analysis Model is mapped to a *block*.
     2. Implementation-specific blocks are added and the collection is revised.
     3. Interfaces and semantics of operations are defined.

  3. Specification of the sequences of interactions among objects and the dynamic behaviour of each block:
     1. An *Interaction Diagram* is drawn for each of the use cases.
     2. A *State Transition Graph* is used for describing the behaviour of each block.

OOSE: Construction - Design Model

Interaction Diagram



[Jacobson et al. 1992]

OOSE: Construction - Implementation

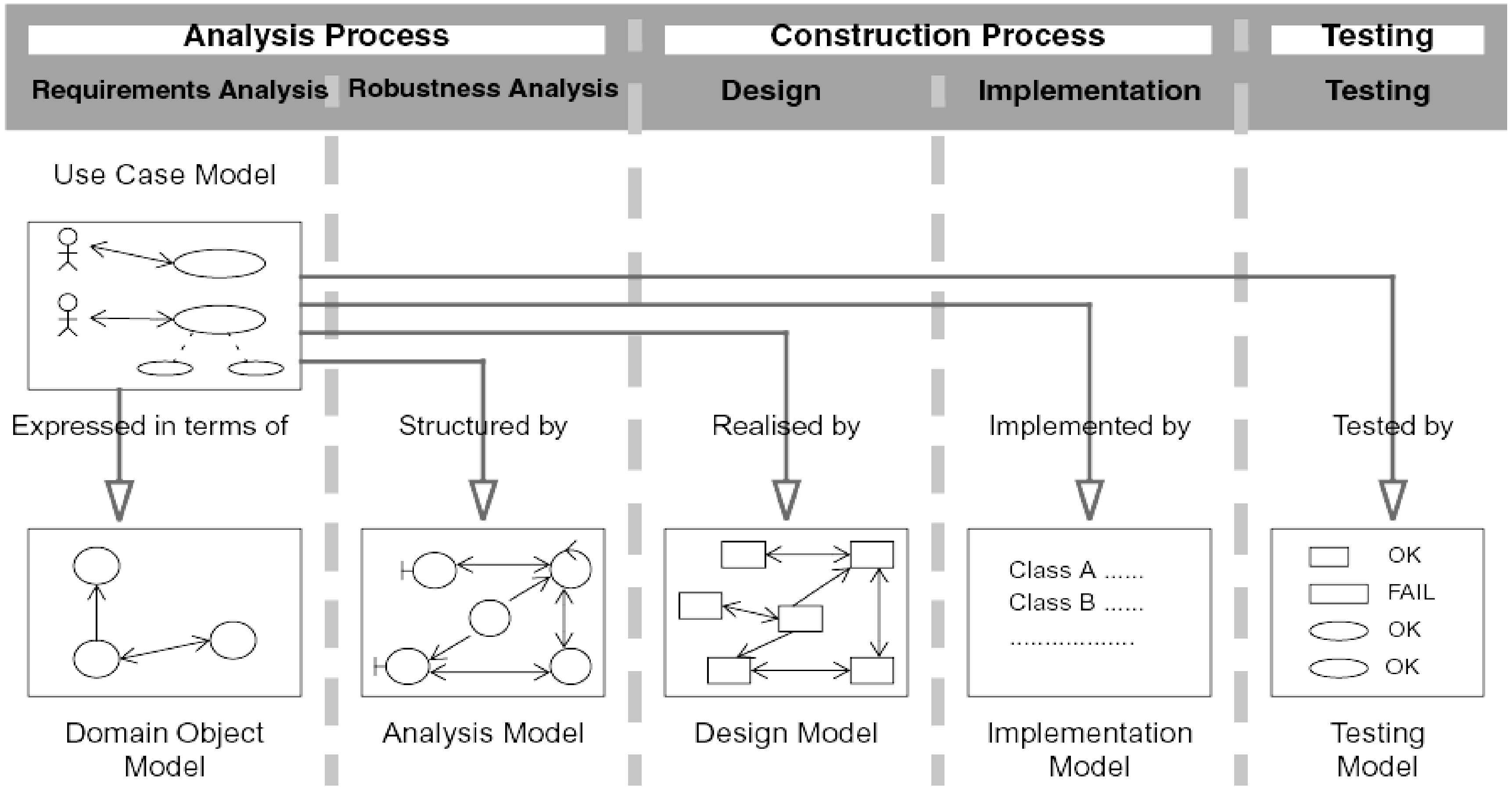
- Aim: Produce the code from the specifications of the packages and blocks defined in the design model.

- The model to be developed is the *Implementation Model*, which consists of the actual source code and accompanying documentation.

OOSE: Testing

- Aim: Verify and validate the implementation model
- The model to be developed is the *Testing Model*, which mainly consists of:
  - Test plan
  - Test specifications
  - Test results.
- Testing is done at three levels, starting from the lowest level:
  - blocks are tested first
  - Use cases are tested next
  - Finally, tests are performed on the whole system

OOSE: Pivotal role of the Use Case Model



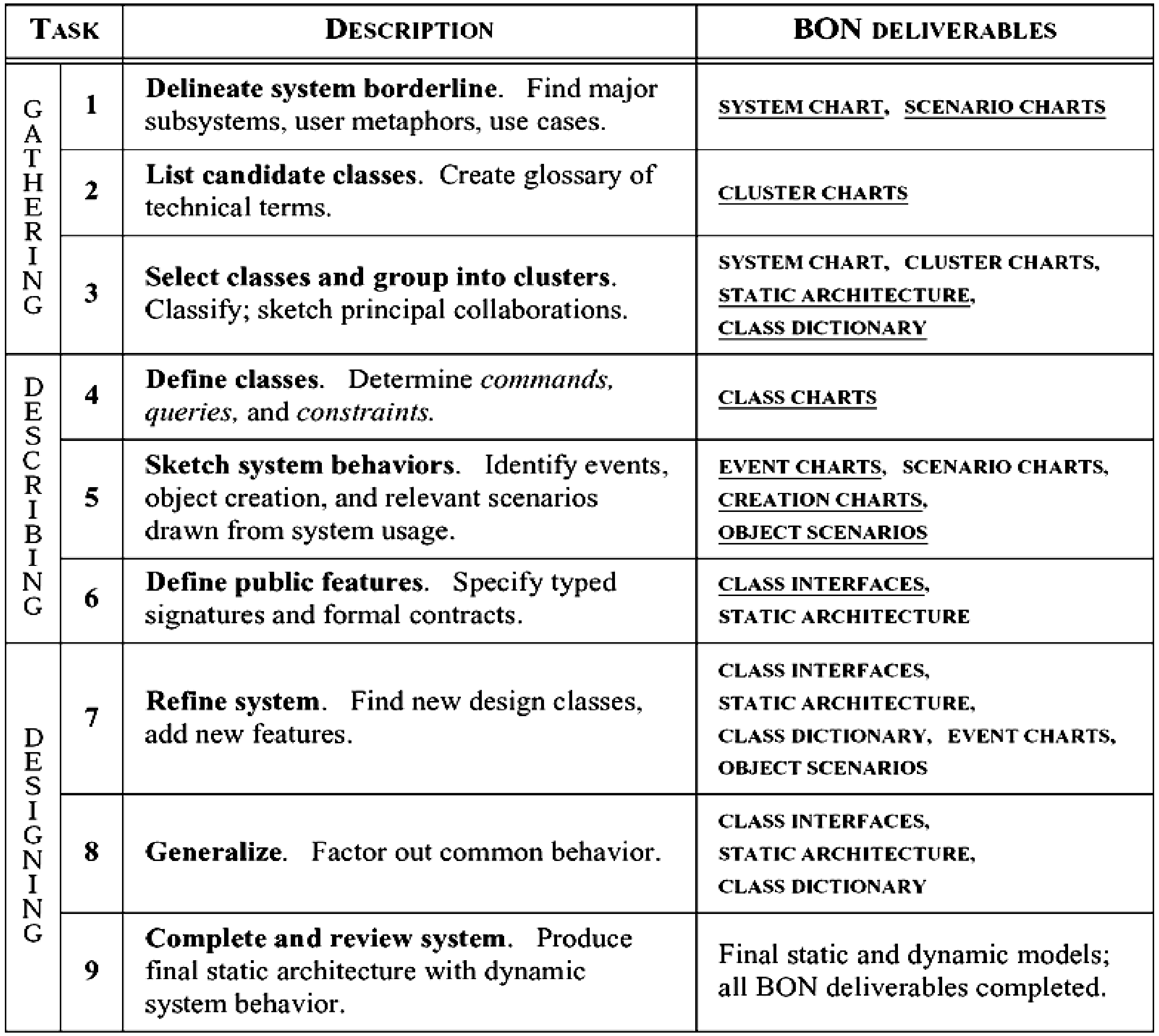
[Jacobson et al. 1992]

Business Object Notation (BON)

- First introduced by Nerson in a 1992 article, with the acronym standing for "*Better* Object Notation"
- A revised and detailed version of the methodology was put forward in 1995; this time the acronym stood for "*Business* Object Notation".
- The process spans the analysis and design phases of the generic software development lifecycle.
- Deeply influenced by Eiffel's assertion mechanisms and the notion of *Design by Contract*

BON: Process

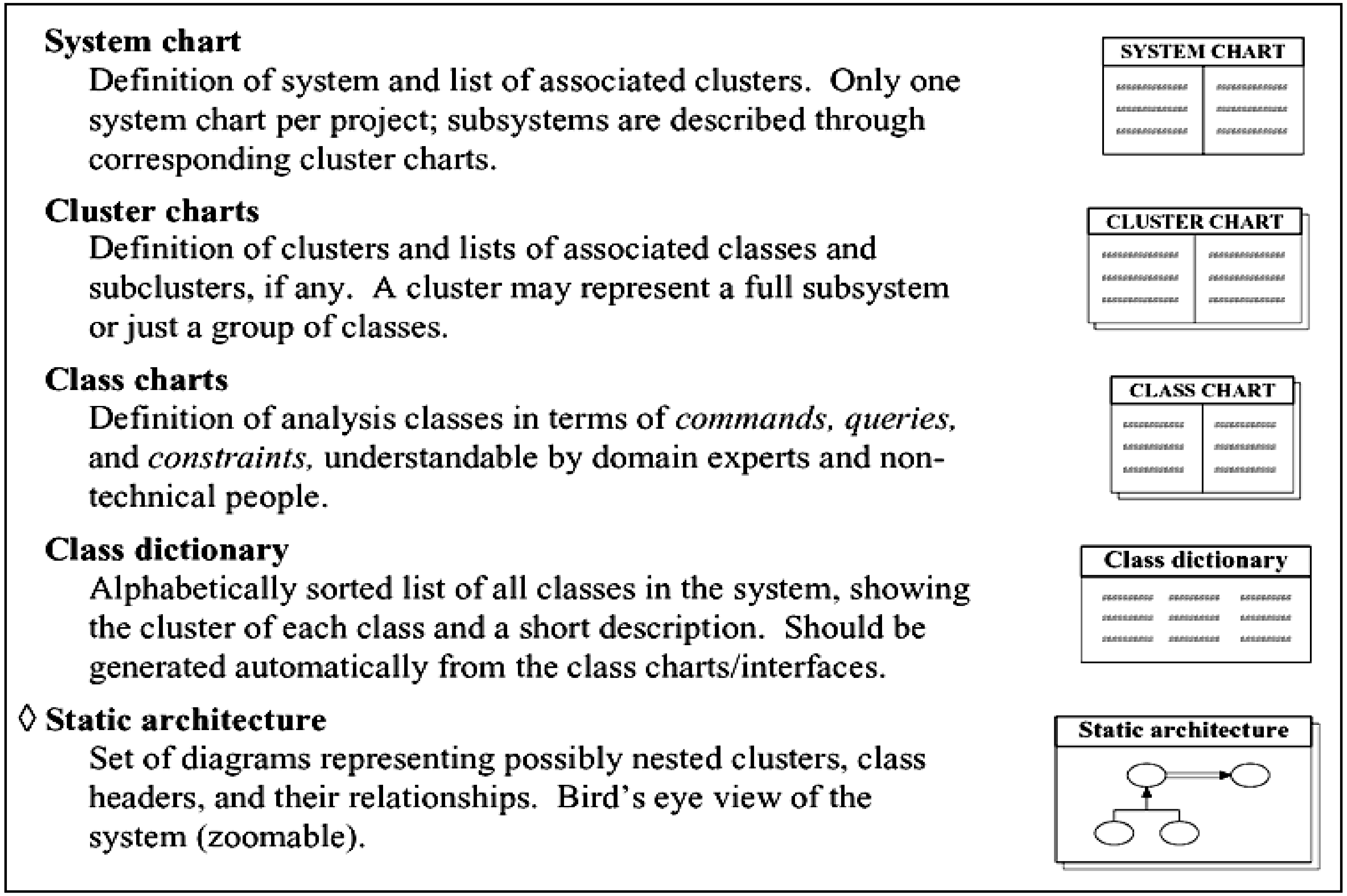
- Consists of nine steps, or tasks.
- Tasks 1-6 focus on analysis and tasks 7-9 deal with design.



Task		Description	BON deliverables
GATHERING	1	**Delineate system borderline.** Find major subsystems, user metaphors, use cases.	SYSTEM CHART, SCENARIO CHARTS
	2	**List candidate classes.** Create glossary of technical terms.	CLUSTER CHARTS
	3	**Select classes and group into clusters.** Classify; sketch principal collaborations.	SYSTEM CHART, CLUSTER CHARTS, STATIC ARCHITECTURE, CLASS DICTIONARY
DESCRIBING	4	**Define classes.** Determine *commands, queries,* and *constraints.*	CLASS CHARTS
	5	**Sketch system behaviors.** Identify events, object creation, and relevant scenarios drawn from system usage.	EVENT CHARTS, SCENARIO CHARTS, CREATION CHARTS, OBJECT SCENARIOS
	6	**Define public features.** Specify typed signatures and formal contracts.	CLASS INTERFACES, STATIC ARCHITECTURE
DESIGNING	7	**Refine system.** Find new design classes, add new features.	CLASS INTERFACES, STATIC ARCHITECTURE, CLASS DICTIONARY, EVENT CHARTS, OBJECT SCENARIOS
	8	**Generalize.** Factor out common behavior.	CLASS INTERFACES, STATIC ARCHITECTURE, CLASS DICTIONARY
	9	**Complete and review system.** Produce final static architecture with dynamic system behavior.	Final static and dynamic models; all BON deliverables completed.

[Walden and Nerson 1995]

BON: Products



System chart

Definition of system and list of associated clusters. Only one system chart per project; subsystems are described through corresponding cluster charts.

Cluster charts

Definition of clusters and lists of associated classes and subclusters, if any. A cluster may represent a full subsystem or just a group of classes.

Class charts

Definition of analysis classes in terms of *commands*, *queries*, and *constraints*, understandable by domain experts and non-technical people.

Class dictionary

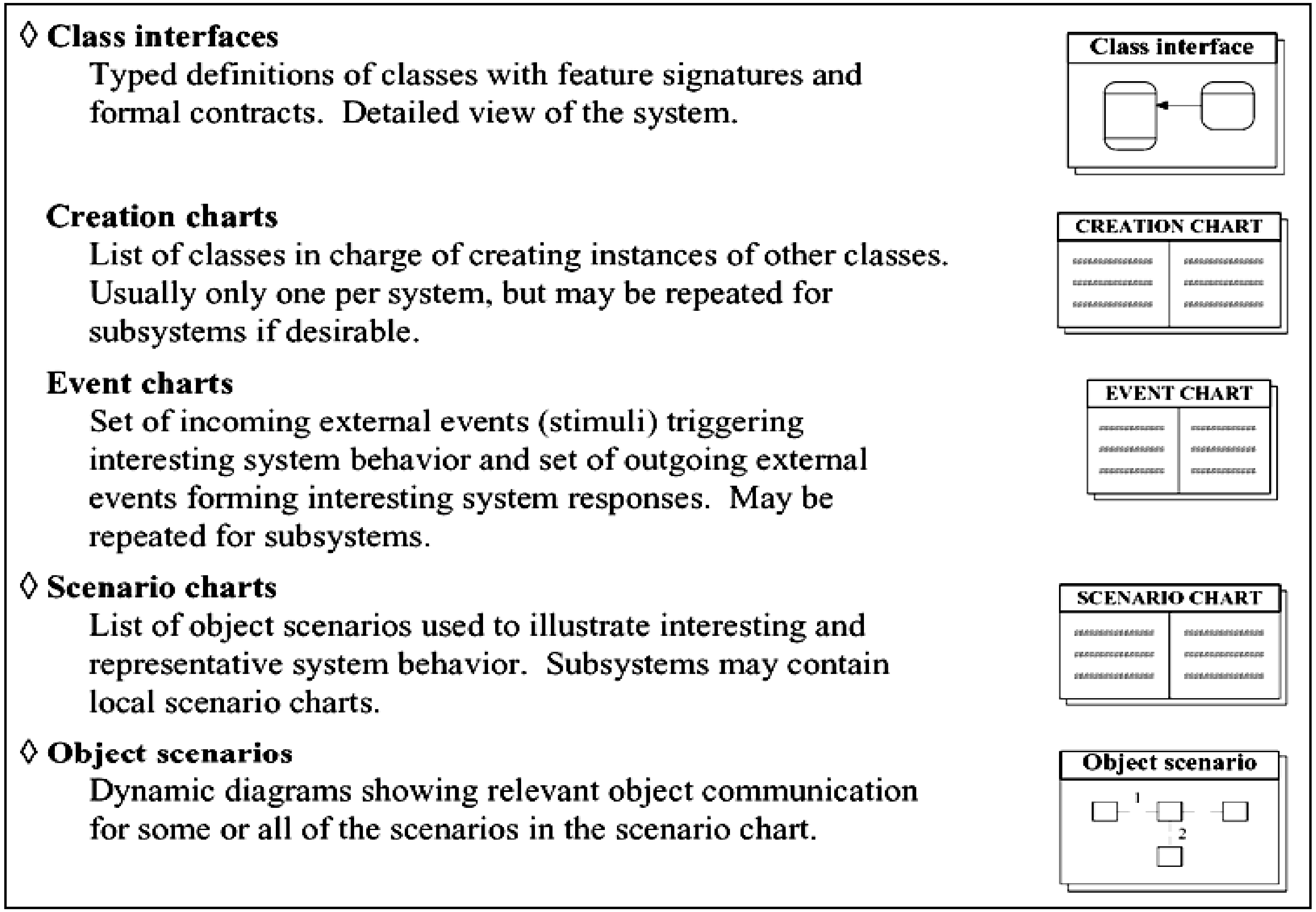
Alphabetically sorted list of all classes in the system, showing the cluster of each class and a short description. Should be generated automatically from the class charts/interfaces.

◊ **Static architecture**

Set of diagrams representing possibly nested clusters, class headers, and their relationships. Bird's eye view of the system (zoomable).

[Walden and Nerson 1995]

BON: Products



◊ **Class interfaces**
Typed definitions of classes with feature signatures and formal contracts. Detailed view of the system.

Creation charts
List of classes in charge of creating instances of other classes. Usually only one per system, but may be repeated for subsystems if desirable.

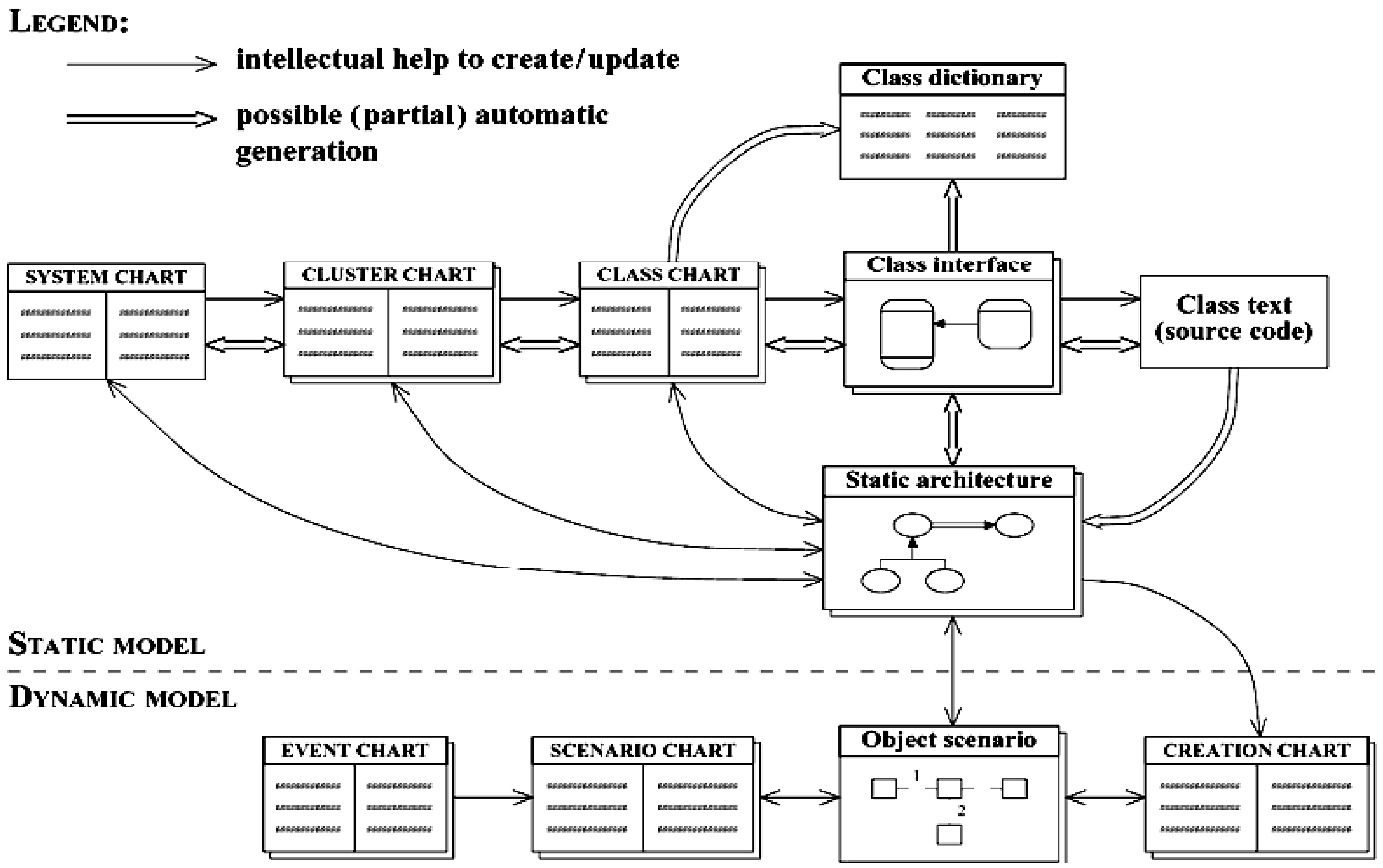
Event charts
Set of incoming external events (stimuli) triggering interesting system behavior and set of outgoing external events forming interesting system responses. May be repeated for subsystems.

◊ **Scenario charts**
List of object scenarios used to illustrate interesting and representative system behavior. Subsystems may contain local scenario charts.

◊ **Object scenarios**
Dynamic diagrams showing relevant object communication for some or all of the scenarios in the scenario chart.

[Walden and Nerson 1995]

BON: Products



[Walden and Nerson 1995]

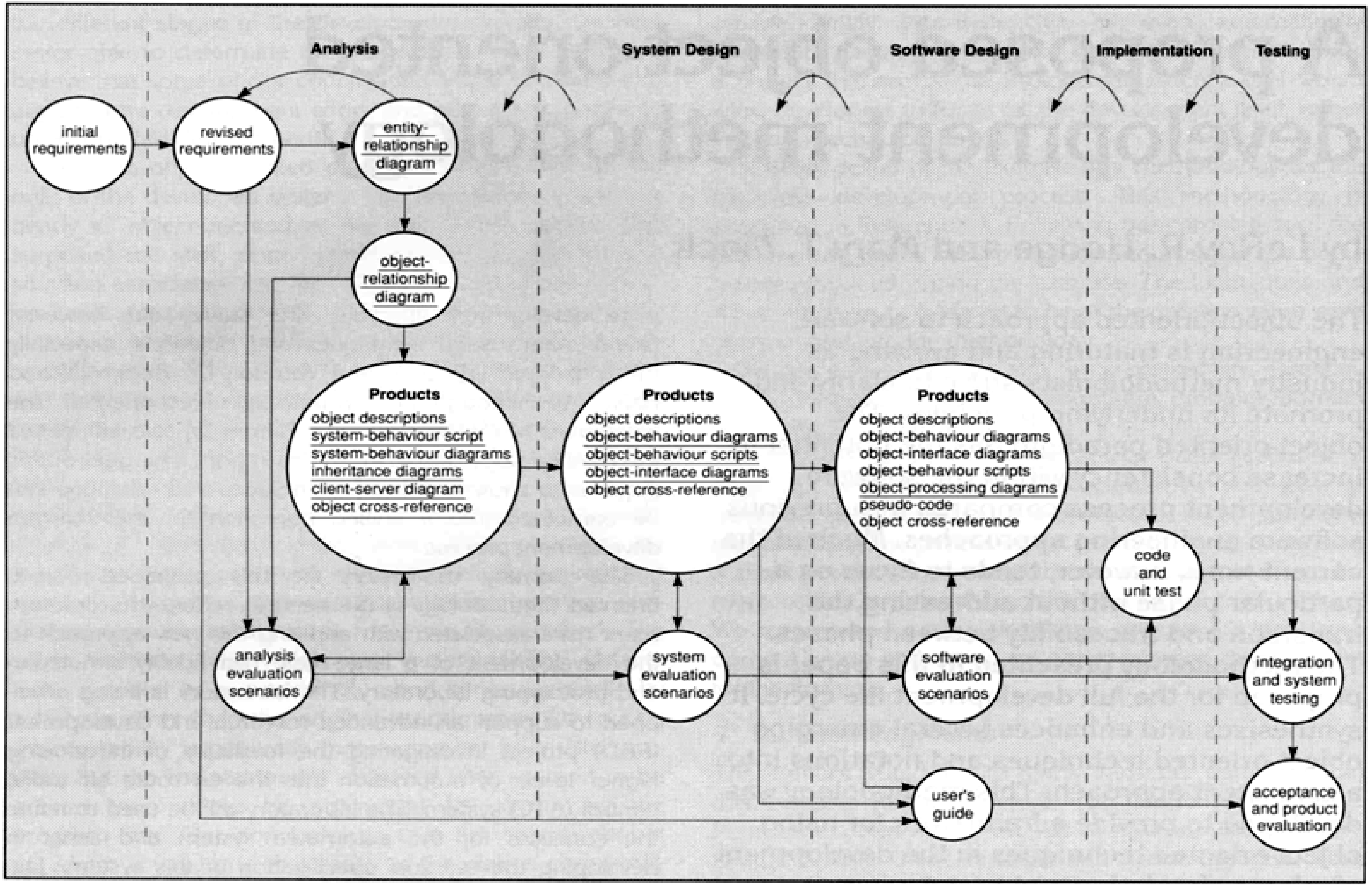
Hodge-Mock

- First introduced in a 1992 article
- Result of research to find an OO methodology for use in a simulation and prototyping laboratory, striving to introduce higher levels of automation into Air Traffic Control (ATC) systems
- Extremely rich as to the types of diagrams and tables produced during the development process
- Due to strong mapping relationships among them, versions of most diagrams and tables are directly derivable from those initially produced.
- Spanning the full generic lifecycle

Hodge-Mock: Process

- *Analysis:* focus on defining the requirements and identifying the scope, structure and behaviour of the system. Consists of four subphases:
  - *Requirements Analysis:* with the focus on eliciting the requirements
  - *Information Analysis:* focus on determining the classes in the problem domain, interrelationships, and collaborations among their instances
  - *Event Analysis:* focus on identifying the behaviour of the system through viewing the system as a stimulus-response machine
  - *Transition to System Design:* focus on providing a more detailed view of the collaborations among objects
- *System Design:* with the focus on adding design classes to the class structure of the system and refining the external behaviour of each class
- *Software Design:* with the focus on adding implementation-specific classes and details to the class structure of the system, and specifying the internal structure and behaviour of each class
- *Implementation:* with the focus on coding and unit testing
- *Testing:* focusing on system-level verification and validation

Hodge-Mock: Process



[Hodge and Mock 1992]

References

- Jacobson, I., Christerson, M., Jonsson, P., Övergaard, G., *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.
- Walden, K., Nerson, J., *Seamless Object-Oriented Software Architecture*. Prentice-Hall, 1995.
- Hodge, L. R., Mock, M. T., “A proposed object-oriented development methodology”. *Software Engineering Journal*, March 1992, pp. 119-129.