### 

|  | Jacobson's OOSE | Booch Methodology | Rumbaugh's OMT |
| --- | --- | --- | --- |
| Focus | Use Cases and system behavior. | Design and architecture, focusing on the system's structure and components. | Modeling, with a strong emphasis on data and functionality. |
| Core Components | 1. Use Cases2. Object Model3. Dynamic Model4. Functional Model | 1. Object Model2. Dynamic Model3. Functional Model | 1. Object Model2. Dynamic Model3. Functional Model |
| Primary Methodology | Use-Case Driven Development (Use cases as the starting point) | Architecture and component design, focusing on classes and their interactions. | Modeling through diagrams to capture data, behavior, and interactions. |
| Modeling Diagrams | 1. Use Case Diagrams2. Class Diagrams3. Sequence Diagrams4. Collaboration Diagrams | 1. Class Diagrams2. Interaction Diagrams3. State Diagrams4. Activity Diagrams | 1. Class Diagrams2. Object Diagrams3. State Diagrams4. Sequence Diagrams |
| Notations | Unified Modeling Language (UML) became the standard. | Original Booch notation (influenced UML). | Object Modeling Technique (also influenced UML). |
| Primary Users | Focused on software engineers and analysts working on large-scale systems. | Primarily targeted at system architects and designers. | Suitable for both analysts and designers. |
| Strengths | 1. Emphasizes functional requirements.2. Strong alignment with iterative development.3. Helps in identifying key system interactions. | 1. Strong at class-based design.2. Good for system-level design.3. Emphasizes modular and component-based design. | 1. Strong focus on object-oriented design.2. Effective for complex systems with clear object models.3. Useful for both analysis and design. |
| Weaknesses | 1. Less prescriptive in terms of detailed design.2. May not be as suitable for smaller or less complex systems. | 1. Somewhat heavy-handed for small projects.2. The learning curve can be steep for beginners. | 1. Can be seen as too focused on static modeling.2. Difficult to transition to dynamic models. |
| Development Process | Emphasizes iterative and incremental development, focusing on evolving use cases. | Iterative development with an emphasis on structure and components. | Structured, but with a heavy focus on object interactions. |
| Relationship with UML | Jacobson's OOSD heavily influenced UML, especially the use case approach. | Booch’s methodology also influenced UML, especially in object-oriented modeling. | Rumbaugh’s OMT also contributed significantly to UML, particularly in static modeling. |
| Typical Applications | Large-scale software systems with a focus on functional requirements. | Large and complex systems that require detailed architectural design. | Systems that require modeling of both objects and their interactions in detail. |

### Summary:

### Jacobson's OOSD focuses on *use cases*, making it ideal for understanding user interactions with the system.

### Booch emphasizes *system architecture* and class structures, often useful in the design phase of large systems.

### Rumbaugh's OMT is balanced and focuses on *object modeling*, making it well-suited for both analysis and design.

### **Types of Models in SDLC**

* **Waterfall Model**: Linear, sequential approach where each phase must be completed before moving to the next.
* **RAD (Rapid Application Development)**: Emphasizes rapid prototyping and iterative feedback.
* **Agile Process Model**: Flexible, iterative, and incremental approach, delivering small parts of software quickly.
* **Spiral Model**: Combines iterative and waterfall; ideal for risk assessment with cycles through planning, risk analysis, development, and evaluation.
* **Open Source Model**: Development where source code is open to the public for collaboration and contribution.
* **DSDM (Dynamic Systems Development Method)**: An Agile framework focused on rapid delivery and quality.
* **Scrum**: Agile framework using sprints and collaborative teams to deliver projects incrementally.
* **Crystal**: Agile framework with a focus on team size, criticality, and project priorities.
* **FDD (Feature-Driven Development)**: Agile, feature-centric approach to development, emphasizing quick, iterative cycles.

### **CMMI (Capability Maturity Model Integration)**

* **Levels**:
  1. Initial: Unpredictable, reactive processes.
  2. Managed: Basic project management, repeatable processes.
  3. Defined: Standardized processes across the organization.
  4. Quantitatively Managed: Data-driven process improvement.
  5. Optimizing: Continuous process improvement.

### **FP, LOC, and COCOMO**

* **FP (Function Points)**: Measures functionality based on requirements, focusing on user interactions and outputs.
* **LOC (Lines of Code)**: Measures the size of software by counting lines of source code.
* **COCOMO I & II (COnstructive COst MOdel)**: Estimation models predicting cost, effort, and schedule.
  + **COCOMO I**: Basic model, mainly for traditional software.
  + **COCOMO II**: Improved model accounting for modern software practices, reuse, and maintenance.

### **EVA, ECP, BVA, PERT & CPM**

* **EVA (Earned Value Analysis)**: Measures project performance and progress.
* **ECP (Expected Change Processing)**: Involves handling changes to project scope and requirements.
* **BVA (Boundary Value Analysis)**: Testing method focusing on boundary values of input data.
* **PERT (Program Evaluation and Review Technique) & CPM (Critical Path Method)**: Project scheduling methods for planning and control.

### **Rumbaugh OMT, Booch Methodology, Jacobson OOSE**

* **Rumbaugh OMT**: Object Modeling Technique for creating object-oriented designs.
* **Booch Methodology**: Focuses on OO software design, using various diagrams.
* **Jacobson OOSE (Object-Oriented Software Engineering)**: Introduced use cases to software design.

### **Requirement Elicitation**

* Process of gathering requirements from stakeholders to understand project needs.

### **Non-Functional and Functional Requirements**

* **Functional**: Defines what the software should do.
* **Non-Functional**: Defines quality attributes, like performance and usability.

### **Design Quality & Guidelines**

* Ensures the software design meets functionality, efficiency, and usability goals.

### **Design Activities**

* Activities include architecture design, interface design, and data structure design.

### **Design Principles**

* **Abstraction**: Hiding complexity to make systems easier to understand.
* **Refinement**: Iteratively developing details from general ideas.
* **Modularity**: Dividing software into distinct, interchangeable modules.
* **Refactoring**: Improving code without changing functionality.
* **Cohesion**: The degree to which elements within a module relate (e.g., functional, sequential).
* **Coupling**: The level of interdependency between modules (e.g., data, control).

### **Architectural Styles**

* Defines the structure of a system (e.g., layered, client-server).

### **CBSE (Component-Based Software Engineering)**

* Focuses on building software with reusable components.

### **White and Black Box Testing**

* **White Box Testing**: Testing internal structures or workings of a program.
* **Black Box Testing**: Testing software functionality without internal knowledge.

### **Alpha and Beta Testing**

* **Alpha Testing**: Internal testing by developers.
* **Beta Testing**: External testing by end-users.

### **Version Control**

* Manages code changes across versions (e.g., Git).

### **Deployment and Component Diagram**

* **Deployment Diagram**: Shows physical deployment of software components.
* **Component Diagram**: Shows the organization and dependencies of components.

### **Types of Transformations**

* Changes applied to software artifacts for better design and functionality.

### **Packages, Interface, Mapping Models to Code**

* **Packages**: Group related classes or modules.
* **Interface**: Defines a contract for classes without implementation.
* **Mapping Models to Code**: Translating design models into executable code.

### **Risk Management (Sources and Approaches)**

* Managing risks arising from technical, financial, or schedule issues.
  + **Boehm’s Techniques**: Risk identification, assessment, prioritization, and mitigation.

### **Principles of Testing**

* Establishes the goals, methods, and best practices in software testing.

### **Requirements of Testing**

* Ensures the software meets defined requirements and quality standards.

### **Strategic Issues in Testing**

* Consideration of cost, timing, and resource allocation in testing.

### **SQA Activities**

* **Software Quality Assurance**: Includes reviews, audits, and testing to ensure quality.

### **FTR (Formal Technical Review) & Review Meeting**

* Formal peer review process to identify defects in development phases.

### **Levels of Testing**

* **Unit Testing**: Testing individual components.
* **Integration Testing**: Testing interactions between components.
* **Regression Testing**: Ensuring new changes don’t introduce defects.
* **Acceptance Testing**: Verifying the software meets user needs.

### **Cyclomatic Complexity**

* Measures the complexity of a program’s control flow.

### **Graph-Based Testing**

* Tests using graphs to represent program states and transitions.

### **OOA and OOD Models**

* **Object-Oriented Analysis (OOA)**: Analyzing requirements with an OO perspective.
* **Object-Oriented Design (OOD)**: Designing software based on OOA, creating OO architectures.

### **Object-Oriented Testing Strategies**

* Methods to test object-oriented software, focusing on classes, inheritance, and polymorphism.

### **Software Rejuvenation & Software Re-engineering Process Model**

* **Software Rejuvenation**: Improving aging software to extend its life.
* **Software Re-engineering**: Updating existing software to improve performance or functionality.

### **Software Maintenance**

* The ongoing process of updating and improving software after deployment.

### **Verification and Validation (V&V)**

* **Verification**: Ensures the product is built correctly by checking if it meets specified requirements at each stage of development. This process answers the question, "Are we building the product right?" Verification involves activities like reviews, inspections, and testing.
* **Validation**: Ensures the final product meets the user’s needs and expectations. It answers, "Are we building the right product?" Validation is typically performed toward the end of the development cycle, involving user acceptance testing and beta testing.

### **Types of Software Maintenance**

1. **Corrective Maintenance**: Fixes bugs or errors discovered in the software after it is deployed. This maintenance type addresses issues like system crashes, incorrect outputs, and functionality failures.
2. **Adaptive Maintenance**: Modifies the software to keep it compatible with new environments, such as hardware, operating systems, or policies, without changing its functionality.
3. **Perfective Maintenance**: Enhances or improves software by adding new features, modifying functionality, or improving performance based on user feedback or emerging requirements.
4. **Preventive Maintenance**: Involves making changes to the software to increase its maintainability, reduce future risks, and prevent potential issues before they arise, helping to extend the software’s life.

### **Why Projects Are Late**

* **Poor Planning**: Inaccurate estimates and scheduling.
* **Scope Creep**: Uncontrolled addition of features.
* **Unclear Requirements**: Leads to rework.
* **Resource Shortages**: Insufficient team or tools.
* **Communication Issues**: Misunderstandings and delays.
* **Technical Challenges**: Unforeseen complexities.
* **Inadequate Testing**: Bugs found late in development.

### **Testing vs. Debugging**

* **Testing**: Process of finding defects by executing code under various conditions.
* **Debugging**: The activity of locating, analyzing, and fixing defects found during testing.