Chapter 4

Object Oriented Design

The object oriented approach to software development promises better management of system complexity and a likely improvement in project outcomes such as quality and the project cycle time [1]. This requires creating a good design. The term ‘good’ refers to a design which is clear, easy to implement and easy to maintain. Design complexity has been conjectured to play a strong role in the quality of the resulting software system in object oriented development environment [1]. This necessitates the early assessment and evaluation of the object oriented design.

Fortunately, the object-oriented approach naturally lends itself to an early assessment and evaluation [2]. To accomplish this we need a proper set of metrics. Design metrics play an important role in helping developers understand design aspect of software and, hence, improve software quality and developer productivity [3]. Although, many object-oriented metrics has been proposed, but there is as yet no consensus on which are best, and most have not been well-validated [4]. Also, many of the metrics and quality models currently available for object oriented software analyses can be applied only after a product is complete or nearly complete [2]. They rely upon information extracted from the implementation of the product. This provides information late to help in improving internal product characteristics prior to the completion of the product. Thus, there is a need for metrics and models that can be applied in the early stages of development (requirements and design) to ensure that the analysis and design have favorable internal properties that will lead to the development of a quality end product.

As per the observation and also reported by [2] there are no known comprehensive and complete models or frame works that evaluate the overall quality of design developed using an object oriented approach based on its internal design properties. Internal design properties are used to predict the external design attributes, so it becomes necessary to
structure a model based on that. But before that it becomes essential to examine the existing quality models, metrics and frameworks of object oriented design

4.1 Previous Quality Models

Quality measures of object oriented code or design artifacts usually involve analyzing the structure of these artifacts with respect to the interdependencies of classes and components as well as their internal elements. The underlying assumption is that such measures can be used as objective measure to predict various external quality aspects of the code or design artifacts, e.g., maintainability and reliability. Quality is viewed from one’s perspective and hence the set of metrics that evolves comes from context independent view of quality.

One of the earliest software quality models was suggested by McCall [7] and his colleagues. McCall’s quality model defines software product qualities as a hierarchy of factors, criteria and metrics and was the first of the several models of the same form. The quality model defined in ISO/EIC 9126-1 “Software engineering product quality” standard classifies quality attributes as external, visible on system and internal, properties of subsystem and components. All these models vary in their hierarchical definition of quality, but they share a common difficulty. The models are vague in their definition of lower levels of details and metrics need to attain a quantitative assessment of product quality [2]. Another difficulty with the earlier models was the inability to account for dependency among quality attributes.

Dromey [8] [9] has addressed some of the problems of earlier models such as McCall’s and ISO 9126. The framework is a methodology for the development of quality models in a bottom-up fashion, providing an approach that will ensure that the lower level details are well specified and computable. The QMOOD [2] extended the Dromey’s generic quality model methodology. This model has the lower-level design metrics well defined in terms of design characteristics, and quality is assessed as an aggregation of the model’s individual high-level quality attributes. The high level quality attributes are assessed using a set of empirically identified and weighted object oriented design properties, which are derived from object oriented metrics which measure the lowest level structural, functional, and relational details of a design. The initial set of design quality attributes in QMOOD is “functionality”, “effectiveness”, “understandability”, “extendibility”, “reusability” and “flexibility”. Figure 4.1. Shows the QMOOD model.

![Figure 4.1 Level and Links in QMOOD](L1 L2 L3 L4 L12 L23 L34)

63
Although this model gives the indication that model of this type can be effectively used in monitoring the quality of software product it has not considered quality attributes such as reliability, maintainability and testability. There is evidence that design metrics is related to a variety of quality characteristics of software product such as reliability, testability and maintainability [10]. A set object oriented design metrics for these attributes are not considered by QMOOD. Also as shown in table 1 across each design property a single metric is used which is not self sufficient in itself to justify the design properties.

As discussed earlier a quality model takes the form of the viewpoint we take of quality which is still a vague and multifaceted concept, the earlier quality models provides a framework from which to proceed.

4.2 Overview Of The Existing Objects Oriented Metrics And Frameworks:

Software measures are a tool to measure the quality of software. The area of software measurement is also known as software metrics. A metric here is not considered in the sense of a metric space, it is considered as: measurement is a mapping of empirical objects to numerical objects by homomorphism. A homomorphism is a mapping, which preserves all relations and structures. Put in words: Software quality should be linearly related to software measure.

Measuring the relatedness in software started changing with the changing approach towards software development i.e., from traditional to object-oriented. History of software measurement by Horst Zuse gives complete overview of the milestones in the development of software measures. According to him more than 5000 paper about software measurements were published till 1996 and more till date.

As this study is related in identifying the candidate metrics for object oriented design it restrict the survey to the existing object oriented metrics and frameworks. In the next section a insight on some of the most widely cited object oriented metrics is brought forth.

4.2.1. Existing Object Oriented Metrics:

A. Moreau and Dominick

They proposed following metrics [11]:

a. Message Vocabulary Size (MVS)
b. Inheritance Complexity (IC)
c. Message domain Size (MDS)

The three defined metrics need classification such as what exactly is meant by “sending message”, and how the metrics are to be computed. However we can draw some parallels...
between these metrics and the three object-oriented software quality abstractions of coupling, inheritance complexity and cohesion.

**B. Chidamber and Kemerer**

The Chidamber and Kemerer metric suite [12] is the most cited set of metrics and also most criticized. There are six metrics in the suite, all of them being design metrics:

a. Weighted method per class (WMC)
b. Depth of inheritance tree (DIT)
c. Number of children (NOC)
d. Coupling between object classes (CBO)
e. Response for class (RFC)
f. Lack of cohesion in methods (LCOM)

Churcher and Shepperd [14] point out that definition of some of the basic direct counts are imprecise, which could have impact on the defined metrics. The main concern lies with the number of methods in a class count, used directly in computation of WMC and indirectly in LCOM. Due to the various possibilities in counting the methods, the results could vary dramatically, leading to confusion. Hitz and Montazeri [15] argue that CBO is not sensitive enough measure of coupling, since it considers all couples to be of equal strength. Henderson – Seller [16] shows LCOM measure is not sensitive enough for cases of high cohesion.

Li and Henry [17] conducted their own empirical experiments, and showed that by using a combination of five of six CK metrics (Omitting CBO), along with some newly defined metrics (see section 3.1.3), it is possible to predict maintenance effort required for a software system. Basili [18] shows that five of the six CK metrics were useful in predicting class fault-proneness during the high and low level phases of life cycle.

**C. Li and Henry**

They [17] present ten metrics in their system; they include five of the six metrics defined by CK, namely DIT, NOC, RFC, LCOM and WMC. In addition they define five more metrics of their own. These are:

a. Message Passing Coupling (MPC)
b. Data Abstraction Coupling (DAC)
c. Number of Methods (NOM)
d. Number of Semicolons (SIZE1)
e. Number of Properties (SIZE2)

Experimenting with these metrics the authors concluded that there is a strong relationship between metrics and maintenance effort in object oriented systems. Also maintenance effort can be predicted from combinations of metrics collected from source code.

However, in SIZE1, the authors use number of semicolons in a class, which is language-dependent and also not derivable until the source code is available. The DIT metric is
used as a measure of complexity, where the larger the value of DIT, the more complex
the system is supposed to be. But trying to minimize DIT leads to the guideline “do not
use inheritance at all”, while inheritance is one of the major advantages of the object
oriented paradigm.

D. Martin’s package metrics

Martin [19] identifies criteria for the proper distribution of classes into packages. These
criteria are essentially based on the notion of dependency. The goal is to reduce
dependency, especially dependencies on concrete class.

Unfortunately, Martin does not define what a dependency exactly is? He only says that
dependencies are caused by class relationships like inheritance, aggregation and uses. As
an educated guess the depends–on–relation, which includes the examples given by
Martin, is used for formal definitions.

Martin does not consider nested packages, even though dependencies of classes in
packages nested inside a package to classes within that package can be considered to
have a special status, as they are more “local” than dependencies from classes in outside
packages. The metrics proposed by Martin is as follows:

a. Relational cohesion (H)
b. Afferent coupling (Ca)
c. Efferent coupling (Ce)
d. Abstractness (A)
e. Instability (I)
f. Distance from main sequence (D)

Martin’s metrics focus on high level, architectural design issues, so they can be
formalized easily. There are some vague points in the original definitions, but these could
be overcome in the formalization by educated guesses.

E. Brito e Abreu

Brito e Abreu [20][21] derived a set of six metrics known as the MOOD (Metrics for
Object Oriented Design) metrics. It includes:

a. Method Hiding Factor (MIF)
b. Attribute Hiding Factor (AHF)
c. Method Inheritance Factor (MIF)
d. Attribute Inheritance Factor (AIF)
e. Polymorphism factor (POF)
f. Coupling Factor (COF)

These metrics refer object-oriented paradigm in following ways:
1. Encapsulation (MHF and AHF)
2. Inheritance (MIF and AIF)
3. Polymorphism (POF)
4. Message Passing (COF)

And are expressed as quotients

\[ \text{Metric} = \frac{X}{\text{Total}} \]

The numerator represents the actual use of those mechanisms for a design. The denominator acting as a normalizer represents the hypothetical maximum achievable use for the same mechanism on the same design. The value for each metric will therefore be in the range 0-1 i.e., between 0-100%.

Definition for MIF and AIF are inconsistent with the 0-1 scale as shown in [22]. Also the AIF is meaningless in the sense that the concept of inheritance concerns the behavior defined in a method, an attribute does not have behavior, and thus cannot be overridden or inherited.

The MOOD metrics have been subjected to much empirical evaluation, with claims made regarding the usefulness of the metrics to assess external attributes such as quality. The theoretical evaluation of MOOD metrics by [23] show that any empirical validation is premature due to the majority of the MOOD metrics being fundamentally flawed. The metrics either fails to meet the MOOD team’s own criteria or is founded on an imprecise, and in certain cases inaccurate, view of Object oriented paradigm.

**F. Lorenz and Kidd.**

Lorenz and Kidd [24] defined many object oriented design metrics, but did not validate nor thoroughly test them [13]. The metrics are listed below, along with the level at which they are taken:

**Table 4.1 Lorenz and Kidd Metrics**

<table>
<thead>
<tr>
<th>Property</th>
<th>Associated Metric</th>
</tr>
</thead>
<tbody>
<tr>
<td>Method Size</td>
<td>Number of messages send, number of statements, lines of code, average method size.</td>
</tr>
<tr>
<td>Method internals</td>
<td>Method complexity, strings of message send.</td>
</tr>
<tr>
<td>Class Size</td>
<td>Number of public instance methods per class, number of instance method per class,</td>
</tr>
<tr>
<td></td>
<td>average number of instance method per class, number of instance variables per class,</td>
</tr>
<tr>
<td></td>
<td>number of class variables per class.</td>
</tr>
<tr>
<td>Method Inheritance</td>
<td>Number of methods over ridden by a sub class (NOV), number of methods inherited</td>
</tr>
<tr>
<td></td>
<td>by a subclass, number of methods added in sub class, specialization index.</td>
</tr>
<tr>
<td>Class</td>
<td>Class cohesion, global usage, average number of parameters per</td>
</tr>
</tbody>
</table>
Chapter 4: Object Oriented Design

4.2.2 Existing Framework for Object Oriented Metrics:

Most aspects of software development process and respective products are too complex to be adequately captured by one single metric. However, the choice of a set of metrics exposes the well-known pitfalls of measuring:

- Too much, thereby getting overwhelmed by a big amount of unmanageable numeric data;
- Too little, thereby not gaining sufficient insight to be able take corrective actions,
- The wrong attributes, thereby deriving delusive conclusions.

To avoid these traps a framework for the implementation of metrics initiatives has to be adopted. Several researchers have proposed different frameworks for o o metrics along different dimensions in an attempt to organize the metric collection. Table 4.2. Shows the framework by Henderson –Seller [25].

**Table 4.2 Henderson –Seller Framework**

<table>
<thead>
<tr>
<th>Perspective</th>
<th>Measures</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inside a class</td>
<td>Size and Complexity</td>
<td>WMC, NOM, NO Attribute Count</td>
</tr>
<tr>
<td>External at the class level</td>
<td>Concerns interface of classes.</td>
<td>Metrics here can be viewed as measuring the services offered by a class.</td>
</tr>
<tr>
<td>System level</td>
<td>Measures from the above two perspectives</td>
<td></td>
</tr>
<tr>
<td>System level relationships</td>
<td>Coupling</td>
<td></td>
</tr>
<tr>
<td>Inheritance coupling</td>
<td>Inheritance hierarchy and coupling</td>
<td></td>
</tr>
</tbody>
</table>

Sheetz et.al. [4] Defines four levels along which metrics can be classified. All the metrics measures the complexity of the software. Table 4.3 gives the details of Sheetz framework.

**Table 4.3 Sheetz Framework.**

<table>
<thead>
<tr>
<th>Level</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>Variable level</td>
<td>Variable fan-in, Variable fan-out,…</td>
</tr>
<tr>
<td>Method level</td>
<td>Method input parameters, method parameters returned, object variables accessed, method</td>
</tr>
</tbody>
</table>
Yet another approach to classifying metrics comes from Bellin [4] as shown in table 4.4.

**Table 4.4 Bellin Framework**

<table>
<thead>
<tr>
<th>Group</th>
<th>Objective</th>
<th>Metrics</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Capturing statistical aspects of OO design</td>
<td>Number of classes, Number of methods, number of messages, number of receiving classes, Number of sender classes, Number of levels in hierarchy,…</td>
</tr>
<tr>
<td>B</td>
<td>Dealing with code reuse</td>
<td>Number of classes reused, percent of reused classes modified.</td>
</tr>
<tr>
<td>C</td>
<td>Deals with the quality of on abstraction of OO system.</td>
<td>Coupling, cohesion …</td>
</tr>
</tbody>
</table>

Abreu [20] has pointed out several frameworks, summarized in tables 4.5 to 4.7, below based on different perspectives such as target, structure and obtainment criterion. He also pointed out that the above taxonomies, although relevant, do not cover the semantics of metrics usage. Neither have they covered the level of abstraction within the paradigm concept. To overcome these problems a new framework was proposed i.e., TAPROOT (Taxonomy Precise for Object Oriented Metrics) framework. [5].

**Table 4.5 Target Taxonomy for Metrics**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Product Metrics</td>
<td>Quantification of attributes of the software development deliverables</td>
<td>Length in words of the user manual, lines of source code, number of relations in database.</td>
</tr>
<tr>
<td>Process Metric</td>
<td>Quantification of attributes of the software development process</td>
<td>Design duration, coding effort, maintenance cost. Average effort for the application of 1 test</td>
</tr>
<tr>
<td>Hybrid Metrics</td>
<td>Mixture of product and process metrics</td>
<td>Cost per function point, time to deliver n LOC, average monthly failure rate per I/O interface.</td>
</tr>
</tbody>
</table>

**Table 4.6 Structure taxonomy for Metrics**

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Elementary Metrics</td>
<td>Quantification of a single attribute of the software development process or deliverables.</td>
<td>Requirement specification dimension in words, LOC, time to complete the design phase,</td>
</tr>
</tbody>
</table>
Table 4.7 Obtainment Criterion Taxonomy for Metrics

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objective Metrics</td>
<td>Precisely defined and equally obtainable on a repeatable fashion, irrespective of the collector or time.</td>
<td>Number of uncommented lines of source of code, average number of yearly produced versions, the number of input screens.</td>
</tr>
<tr>
<td>Subjective Metrics</td>
<td>Depends upon the collector's judgment; may lead to incoherent and non repeatable measures</td>
<td>Programmer’s experience, average learning time, ease of utilization of a certain application</td>
</tr>
</tbody>
</table>

Table 4.8 TAPROOT Classification Framework

<table>
<thead>
<tr>
<th>Method</th>
<th>Class</th>
<th>System</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design</td>
<td>MD</td>
<td>CD</td>
</tr>
<tr>
<td>Size</td>
<td>MS</td>
<td>CS</td>
</tr>
<tr>
<td>Complexity</td>
<td>MC</td>
<td>CC</td>
</tr>
<tr>
<td>Reuse</td>
<td>MR</td>
<td>CR</td>
</tr>
<tr>
<td>Productivity</td>
<td>MP</td>
<td>CP</td>
</tr>
<tr>
<td>Quality</td>
<td>MQ</td>
<td>CQ</td>
</tr>
</tbody>
</table>

The metrics are classified along two “independent Vectors”, category and granularity. The authors [5] reveal that the categories were derived after a sample of 128 references was reviewed in order to find a common denominator in the extensive metric literature. The categories are design, size, complexity, reuse, productivity and quality. The second dimension granularity further refines the categories by considering metrics in each category at the method, class and system level.

However TAPROOT cannot be considered as a final proposal. Looking in depth at each metric abstraction it is observed that values across these metrics cannot be obtained till date in the implementation phase. So it cannot be considered as a framework to be used in the early design phase. Also at the granularity level packages has not been given any consideration. Therefore a new framework is needed which can be used in the early design phase for assessing the design quality and predicting the reliability of the object oriented software.

4.3 OO Analysis and Design Methods

Numerous reviews [41],[42],[43],[44] exist comparing the various object-oriented approaches. Fichman [43] presents an executive summary comparing various analysis and design approaches, focusing primarily on the structured paradigm, and concluding with a description of Wasserman OO Structured Design, Booch OO Design, and Wirfs-
Chapter 4: Object Oriented Design

Brock Responsibility-Driven Design. This summary was designed to transition the structured programmer into object-oriented programming. A more exhaustive OO summary was performed by Cribbs et. al [27]. They present comparison of the notations, terminologies, and models used in methods by Booch, Coad and Yourdon, Edwards and Odell and Martin, Graham, Rumbaugh, Shlaer and Mellor, Wasserman and Pircher, and Wirfs-Brock. This work also served as a transitional piece from the structure to the object oriented approach. Monarchi and Puhr [29] compared twenty three object-oriented analysis and design methodologies to identify common themes, and strengths and weaknesses.

A. Coad and Yourdon


This method clearly differentiates between analysis and design. The five major layers of analysis are the class-object layer, the structure layer, the subject layer, the attribute layer, the service layer. These layers are also applied to four components during design. These are problem domain component, human interaction component, task management component, data management component. The method clearly differentiates between analysis and design and the notations used in the two are same.

The method does not prescribe a development life cycle. However, this method can provide guidelines on the type of staff required, the interplay between these staff and the stages of development when they are required. There is little discussion of general project management issues.

B. Booch


This method addresses OO design. It does discuss OO analysis but spends little time guiding the reader through the painful; process of identifying objects from the problem domain. Instead it sites the techniques used by others for analyzing the problem. In fact the overall method is approached from the viewpoint of a set of techniques with out a well defined process (in common with many object oriented methods).

One interesting point is that whereas the Coad and Yourdon method differentiates between associative relationships and message passing between two objects, in the Booch method association between two objects implies a message passing relationship between the related objects. This makes sense that objects can only interact through message passing.

The Booch method picks individual techniques from many other methods, including the use of CRC cards to develop object relationship
**C. Jacobson**


OOSE is an “Use Case driven approach to Software Engineering”. Ivar Jacobson's Object-Oriented Software Engineering (OOSE) is one of the precursors to the more modern Unified Modeling Language (UML). OOSE includes requirements, an analysis, a design, an implementation, and a testing model.

**Requirement Model:**
The Jacobson requirements model includes a problem domain object diagram and use case diagrams. This model defines the limits and functionality of a system.

**Problem Domain Object Diagram:**
The problem domain object diagram provides a logical view of the system, which is used to specify the use cases for use case diagrams.

**Use Case Diagrams**
Use case diagrams illustrate how the outside world interacts with elements of the application system.

**Analysis Model**
Analysis model defines three types of objects in a system: interface objects, entity objects, and control objects.

**Design Model**
Design model shows how the system behaves. There are two types of diagrams under this model: interaction diagrams and state transition diagrams. Interaction diagrams are similar to UML's sequence diagrams. State transition diagrams are like UML statechart diagrams.

**D. Wirfs-Brock**

Method: Designing Object-Oriented Software (DOOS) [31] 1990

DOOS is divided into two parts:

1. An initial exploratory phase (Analysis in the software lifecycle): The main topic of the initial exploratory phase is determining the objects, responsibilities and collaborations of objects that play a role in the real world.
2. A detailed analysis phase (First part of Design in the software lifecycle): In the detailed analysis phase the results from the first part are refined and streamlined. At this point a full specification can be made.

The techniques that are used in the DOOS method are:

1. **Class-Responsibility-Collaboration card** (CRC card) on which a Class with its Super/Subclasses, Responsibilities and Collaborations is denoted.
2. **Subsystem cards** on which Subsystems, Contracts and Delegations are denoted.
3. Class Hierarchy Graphs to show the classes and their Inheritance hierarchies.
4. Venn Diagrams to examine the chosen Inheritance hierarchies for classes.

Collaborations graphs to show Classes, Subsystems and the Client-Server Collaborations between them. The clients and servers for contracts are denoted

E. Martin

Method: Object-Oriented Analysis and Design (OOAD) 1993

Although Object-Oriented Analysis is often divided into structural (static) and behavioral (dynamic) aspects, the OOAD method tries to integrate these two in their process part. However, they state that Object-Oriented Analysis should be founded on a behavioral foundation, so the main focus of the method is the behavioral aspects; structural aspects are identified and described as a derivation of the behavioral aspects.

Although the product part of the OOAD method doesn’t describe it as a part of the method, it is emphasized several times that a complex realm can be broken down in manageable chunks by first using Object Flow diagrams, for which no process part is available.

The OOAD method starts with identifying the goal of the realm, and this function as a starting point. In a cyclic way the Events and Objects are defined, and for each new identified Event the cycle has to be repeated, until all Events are defined. At this point it can be necessary to define Events on a more detailed level, if so, the cyclic approach has to be repeated for the Sub Events of the higher level Event.

The way the system is analyzed can be done in the described top-down way; however, a bottom-up approach can also be followed, in which lower level Events are described first.

A real process part is only available for the Analysis part, it is only briefly described how the analysis models can be transformed into a design and an implementation.

F. Rumbaugh


OMT along with Booch is considered to be one of the best Object-Oriented methodologies. It is used extensively by many companies, has a wealth of documentation available and a large number of case tools support it. The text describing OMT is excellent with a section on Analysis, which is worth reading regardless of the design methodology to be used. The main difficulty with OMT is
not what is produced, but the diagrams used to represent it. The diagrams in this methodology are angular and to the uninitiated (even with a notation guide) are difficult to follow; being ambiguous until a textual description is read. Where OMT fails miserably is when it comes to design, as it lacks the step-by-step approach of the analysis phase

**G. Shlaer and Mellor.**

Method: Object Lifecycles (OL) [33][34] 1992

OL divides System Development into OO Analysis and OO Design. OO Analysis is described in three steps:

1. Information modeling
2. In this step, the focus is on abstracting the conceptual entities in the problem domain in terms of objects and attributes. The associations that exist between the entities are formalized as relationships that are based on the policies, rules, and physical laws that prevail in the real world.
3. State modeling
4. This step is concerned with behavior of objects and relationships over time. State models are used to formalize the lifecycles of both objects and relationships. The state models, which consist of state transition diagrams and tables, communicate with each other by means of events. State models are defined by multi-layers of state transition diagrams to make the model of communication orderly and understandable.
5. Process modeling
6. The actions of the state models, which contain all of the required processing, are dissected into fundamental and reusable processes and are expressed by an enhanced form of the traditional data flow diagram - the Action DFD. The processes that are so derived can then be converted directly into operations of object-oriented design.

**OO Design**

Design is expressed in terms of a single program. Each program is made up of:

- a main program, responsible for intertask communicating, the invocation of operations, and the initialization of application classes
- four architectural classes that provide mechanisms required to initialize and traverse state machines and implement the Timer object of the analysis phase
- some number of application classes that are derived from the objects and state models as yielded during analysis.

Shlaer and Mellor use OODLE: a language-independent textual and graphical notation for object-oriented design.
**H. Coleman**


The Fusion method is a combination of different sections of different methods. It was discounted almost immediately due to its failure to describe an organized methodology for developing applications. A large amount of documentation is produced during the Fusion methodology, however the processes by which that documentation is produced, the manner in which that documentation links - or its overall cohesiveness, and the actual worth of the documentation produced is sadly lacking.

**I. de Champeaux**

Method: Object-Oriented System Development (OOSD) 1994

OOSD consists of the phases analysis, design and implementation. Analysis and design are extensively discussed, while implementation is only illustrated by some considerations and remarks.

A core vocabulary is used to represent the system that is modeled. The vocabulary is based on four components. A static and dynamic component form one axis that is used to distinguish the models that are produced in analysis. The other axis consists of a component for modeling states or behavior inside a single object, while the fourth component addresses interobject connections. In OOSD this approach is also known as the 'four component view'

**J. Henderson-Sellers and Edwards**

Method: The MOSES 1994

The most important aspects are the phases in the process lifecycle. The first (Business Planning) and last (Delivery) stages of the product lifecycle are less important, because they are independent to object-oriented technologies (OT) and there are many other texts covering these areas. The Growth period is the one that over the years has occupied the mind of the developers more and more. The Maturity period is the costly one. This period consists of several successive Enhancement periods. The goal is to improve the overall quality by each (Enhancement) period. Nevertheless we can decompose both the Growth and Maturity period in three stages. The main interest goes out to the build stage. It is this stage on which methods (both traditional and OO) have tended to focus. It is important to understand that it is possible to use this framework under different circumstances: MOSES can be applied within an iterative development process, the waterfall concept or the fountain concept.
All the five phases are based upon a set of activities. Each phase calls a different set of activities in a different order

**K. Jean-Marc Nerson and Kim Waldén**

Method: Business Object Notation (BON) [1] 1992

BON, the Business Object Notation, is a method and graphical notation for high-level object-oriented analysis and design. BON is based on concepts similar to those of Eiffel but can be used independently of Eiffel, for example by people using another O-O language for implementation. A particular attraction of BON is the availability of the supporting tool: EiffelCase

The three main concepts of the BON method are seamlessness, reversibility and software contracting:

- Seamlessness is the principle of using a consistent set of concepts and notations throughout the lifecycle, avoiding the impedance mismatches of traditional approaches.
- Reversibility guarantees that changes made at any step in the process, even as late as detailed implementation or maintenance can be reflected all the way back to the earlier steps, including analysis, so as to guarantee the consistency of the entire project baseline.
- Software contracting, an idea also fundamental in the Eiffel language, views the construction of a software system as a succession of precise contracts between its modules, to guarantee reliability and consistency.

**L. Wirfs-Brock and Beck**

Method: Responsibility Driven Design (RDD) including the CRC method 1989 and 1990. The RDD process follows these basic steps:

- First you identify the scenarios that you need to go through
- Then you come out with small components that could participate in those scenarios. You assign them responsibilities - "An object’s responsibilities are high-level statements about both the knowledge it maintains and the operations it supports."
- Components depend on other components or interact with them. So the final step is to depict those collaborations.

An intuitive way to represent the above concepts are CRC cards.

**M. Reenskaug and Skaar**

Method: Object oriented Role Analysis, synthesis and structuring (OORASS) 1991
OORASS is an analysis and design method, which emphasizes the role played by objects in the system. The goals of OORASS are:

1. **Total lifecycle support** - the method should cover the analysis, design, implementation, testing and maintenance of object-oriented programs in a seamless fashion
2. **Integrated description** - the method should support an integrated representation of all kinds of informal documentation as well as source code and formal descriptions of the design
3. **Separation of concerns** - the method should allow different concerns to be treated independently, to allow the whole system to be understood in parts
4. **Reuse** - the method should support the notion of reusable ideas, specifications, designs, code and compiled programs
5. **Tool support** - the method will be supported by an integrated support environment for the complete system lifecycle

One of the advantages of OORASS is the ability to construct simple role models representing object interaction, and then combine them in a uniform way to create larger models. This encourages reuse on a larger scale than is advocated in other object oriented methods.

**N. Wasserman**

Method: Object Oriented Structured Design (OOSD) [36] (1990)

OOSD is not so much of a design method as an object-oriented notation. As described by Wasserman et al., the "goal of OOSD is to provide a single architectural design notation that can support every software design." The main entities in OOSD are:

- **A system**, which consists of a set of units, which are either classes, modules or monitors.
- **A class**, which can inherit from, and use other classes. A class contains public and private units, and can raise exceptions.
- **A module**, which is either a data-object or an operation.
- **A monitor**, which is used to define and use asynchronous processes.

OOSD strengths derive from:

- Keeping modularity as the main component of the design method.
- Building on a familiar notation.
- Attempting to target a wide variety of applications and development languages, while supporting development from a wide range of analysis methods.
Without any method behind the notation, OOSD is useless to those trying to learn the principles of object orientation. It may, however, be of use to those already familiar with a method, who find its notation limiting.

O. Ada community

Method: Hierarchical Object Oriented Design (HOOD) 1989

Hierarchical Object-Oriented Design (HOOD) has been developed for the European Space Agency as a design/notation method for Ada.

The main process in HOOD is called the Basic Design Step. "A basic design step process is further split into four phases, thus defining a micro life-cycle for a design step." The phases can be summarized as follows:

1. Problem definition.

The context of the object to be designed is stated, with the goal of organizing and structuring the data from the requirement analysis phase. This is an opportunity to provide a completeness check on requirements and traceability to design.

1.1 Statement of the problem - the designer states the problem in correct sentences which provides:

- a clear and precise definition of the problem;
- the context of the system to design.

1.2 Analysis and structuring of requirement data - the designer gathers and analyses all the information relevant to the problem, including the environment of the system to be designed.

2. Development of solution strategy.

The outline solution of the problem stated above is described in terms of objects at a high level of abstraction.

3. Formalization of the strategy.

The objects and their associated operations are defined. A HOOD diagram of the proposed design solution is produced, allowing easy visualization of the concepts and further formalization. There are five sub phases in the formalization of the strategy:

3.1 Object identification.
3.2 Operation identification.
3.3 Grouping objects and operations (object operation table).
3.4 Graphical description.
3.5 Justification of design decisions.

4. Formalization of the solution.

The solution is formalized through:

- Formal definition of provided object interfaces
- Formal description of object and operation control structures

**P. Colbert**

Method: Object Oriented Software Development Method [37] 1989

OOSD develops a single consistent abstract model of the elements of a problem. During requirements analysis OOSD builds this model from the required objects, classes, functions, behavior, and properties of the problem. During design, the model is refined into architecture for software components, with a smooth transition to code. The constant refinement facilitates the tracing of requirements throughout the process. Careful study of object interactions and behavior makes this method sensitive to real–time issues.

OOSD allows particularly rigorous behavior analysis. This detects errors early. However, the price is a higher front–end work load, since time and effort must be invested in detailed behavior descriptions (unless descriptions from previous projects can be re–used), and a delay in the production of code (even though code production will be more efficient when it occurs). When there is no human–safety or other substantial reliability concern, a less formal use of the method may be adequate.

OOSD’s uniform support for localization and information hiding leads to well–defined objects. Each object is individually testable and provides sufficient information to develop tests. The model developed in requirements analysis provides sufficient information to begin designing integration or acceptance tests for the system, which are then refined as the model is refined. The effects of change can be predicted with assurance, both in the diagrams, forms, and tables of the model, and in the final system.

Formally defining the complete structure, interactions, behavior, and properties of all objects in the system, including the system object, allows validation and verification that the system is correctly implemented. Generating classes from objects in the system yields reliable, re–useable components. Carrying out the same essential activities during requirements analysis, preliminary design, and detailed design, results in unity of the model. The product is easy to understand and maintain, and closely follows the “real world”.
Q. Lewi, J., Steegmans, E., and Van Baelen, S

Method: Entity-Relationship Object-Oriented Specifications (EROOS) [38] 1990

EROOS consists of following phases, these are:
1. Analysis phase called EROOS-A, consists of making a history document from which the analysis plan is derived.
2. Design phase called EROOS-D, which is based on analysis plan.
3. Implementation phase, derived from design.

A major development principle is that the analysis model forces the underlying model for the design and implementation phases. A characteristic of EROOS technique is that object structure (decoration and refinement) and kernel functionality are strongly connected with each other. Adding a piece of object structure introduces additional kernel functionality. In the same way removing a piece of object structure introduces a reduction of kernel functionality.

R. Object Management Group


The Unified Modeling Language™ - UML - is OMG's most-used specification, and the way the world models not only application structure, behavior, and architecture, but also business process and data structure.

UML, along with the Meta Object Facility (MOF™), also provides a key foundation for OMG's Model-Driven Architecture, which unifies every step of development and integration from business modeling, through architectural and application modeling, to development, deployment, maintenance, and evolution.

The common mechanisms in UML that apply consistently throughout the language are specification, adornments, common divisions and extensibility mechanism. The diagrams supported by UML are: class diagram, object diagram, uses case diagram, sequence diagram, collaboration diagram, statechart diagram, activity diagram, component diagram and deployment diagram.

The UML is largely process- independent, meaning that it is not tied to any particular software development life cycle. However to get the most benefit from the UML, one should consider a process that is use case driven, architecture-centric, iterative and incremental.

Inspite of various OOA and OOD methodologies available, the most widely accepted and adopted software modeling notation, UML is evolving towards notation extensions that allow easy and accurate modeling of performance and reliability aspects of modern software and hardware systems [6]. The well-assessed extension mechanism of UML smooth the process of integration, as proven by numerous profiles that have been proposed [13] and [26] accepted since the creation of UML metamodel. For these reasons
and the standardization of UML this research work has targeted object oriented design model using UML for a consistent approach.

After taking the overview of the existing models, metrics and frameworks, a need is felt of having a model with a good framework and a set of metrics, which can be used in the early design phase for predicting the reliability of the object-oriented software.

Reference:


31. Rebecca Wirfs-Brock, Brian Wilkerson, Lauren Wiener, "Designing Object-Oriented Software" Released: 18 June, 1990