TRACKING AND CONTROLLING OBJECT ORIENTED SOFTWARE PROGRESS

Shalini Chawla[1], Arun bakshi[2]
[1] *Shalini Chawla, Northern India Engineering College, Delhi, India.*
[2] *Arun Bakshi,Gitarattan International Business School, Delhi, India.*

(er.shalinichawla[1], akshayabakshi[2])@gmail.com

Abstract
There are many ways to look at a problem to be solved using a software-based solution. One widely used approach to problem solving takes an object-oriented viewpoint. The definition of objects encompasses a description of attributes, behaviors, operations, and messages. An object encapsulates both data and the processing that is applied to the data. This important characteristic enables classes of objects to be built and inherently leads to libraries of reusable classes and objects. Because reuse is a critically important attribute of modern software engineering, the object-oriented paradigm is attractive to many software development organizations.

Index Terms **– Object oriented, Software development productivity, Classes.**

I. INTRODUCTION

Object oriented Software Engineering (OOSE) is an object modeling language and methodology. The approach of using object – oriented techniques for designing a system is referred to as object – oriented design. Object – oriented development approaches are best suited to projects that will implement systems using emerging object technologies to construct, manage, and assemble those objects into useful computer applications. Object oriented design is the continuation of object- oriented analysis, continuing to center the development focus on object modeling techniques. Object-oriented software engineering follows the same steps as conventional approaches. Analysis identifies objects and classes that are relevant to the problem domain; design provides the architecture, interface, and component- level detail; implementation (using an object-oriented language) transforms design into code; and testing exercises the object-oriented architecture, interfaces and components[2].

Object technologies lead to reuse, and reuse (of program components) leads to faster software development and higher-quality programs. Object-oriented software is easier to maintain because its structure is inherently decoupled. This leads to fewer side effects when changes have to be made and less frustration for the software engineer and the customer. In addition, object-oriented systems are easier to adapt and easier to scale (i.e., large systems can be created by assembling reusable subsystems).

II. MANAGEMENT OF OBJECT-ORIENTED SOFTWARE

Software project management can be subdivided into the following activities:

1. Establishing a common process framework for a project.
2. Using the framework and historical metrics to develop effort and time estimates.
3. Establishing deliverables and milestones that will enable progress to be measured.
4. Defining checkpoints for risk management, quality assurance, and control.
5. Managing the changes that invariably occur as the project progresses.
6. Tracking, monitoring, and controlling progress[3,6].

A. The Common Process Framework for OO

A common process framework defines an organization's approach to software engineering. It identifies the paradigm that is applied to build and maintain software and the tasks, milestones, and deliverables that will be required. It establishes the degree of rigor with which different kinds of projects will be approached. The CPF is always adaptable so it can meet the individual needs of a project team. This is its single most important characteristic. As we noted earlier in this chapter, object-oriented software engineering applies a process model that encourages iterative development. That is, OO software evolves through a number of cycles. The common process framework that is used to manage an OO project must be evolutionary in nature[1].

In essence the recursive/parallel model works in the following way:

- Do enough analysis to isolate major problem classes and connections.
- Do a little design to determine whether the classes and connections can be implemented in a practical way.
- Extract reusable objects from a library to build a rough prototype.
- Conduct some tests to uncover errors in the prototype.
- Get customer feedback on the prototype.
- Modify the analysis model based on what you've learned from the prototype, from doing design, and from customer feedback.

• Refine the design to accommodate your changes.
• Code special objects (that are not available from the library).
• Assemble a new prototype using objects from the library and the new objects you've created.
• Conduct some tests to uncover errors in the prototype.
• Get customer feedback on the prototype.
This approach continues until the prototype evolves into a production application.

B. OO Project Metrics and Estimation

Conventional software project estimation techniques require estimates of lines-of code (LOC) or function points (FP) as the primary driver for estimation. Because an overriding goal for OO projects should be reuse, LOC estimates make little sense. FP estimates can be used effectively because the information domain counts that are required are readily obtainable from the problem statement. FP analysis may provide value for estimating OO projects, but the FP measure does not provide enough granularity for the schedule and effort adjustments that are required as we iterate through the recursive/parallel paradigm[1,3].
The set of project metrics are as follows:
Number of scenario scripts: A *scenario script* (analogous to use-cases) is a detailed sequence of steps that describe the interaction between the user and the application. Each script is organized into triplets of the form
{**initiator,** *action*, **participant**}
where **initiator** is the object that requests some service (that initiates a message); *action* is the result of the request; and **participant** is the server object that satisfies the request. The number of scenario scripts is directly correlated to the size of the application and to the number of test cases that must be developed to exercise the system once it is constructed.
Number of key classes: *Key classes* are the "highly independent components" that are defined early in OOA. Because key classes are central to the problem domain, the number of such classes is an indication of the amount of effort required to develop the software and also an indication of
the potential amount of reuse to be applied during system development.
Number of support classes: Support classes are required to implement the system but are not immediately related to the problem domain. Examples might be GUI classes, database access and manipulation classes, and computation classes. In addition, support classes can be developed for each of the key classes. Support classes are defined iteratively throughout the recursive/ parallel process. The number of support classes is an indication of the amount of effort required to develop the software and also an indication of the potential amount of reuse to be applied during system development.
Average number of support classes per key class: In general, key classes are known early in the project. Support classes are defined throughout. If the average number of support classes per key class were known for a given problem domain, estimating (based on total number of classes) would be much simplified. Applications with a GUI have between two and three times the number of support classes as key classes. Non-GUI applications have between one and two times the number of support classes as key classes.
Number of subsystems: A *subsystem* is an aggregation of classes that support a function that is visible to the end-user of a system. Once subsystems are identified, it is easier to lay out a reasonable schedule in which work on subsystems is partitioned among project staff[5,4].

C. An OO Estimating and Scheduling Approach

The approach is as follows:
1. Develop estimates using effort decomposition, FP analysis, and any other method that is applicable for conventional applications.
2. Using OOA, develop scenario scripts (use-cases) and determine a count. Recognize that the number of scenario scripts may change as the project progresses.
3. Using OOA, determine the number of key classes.
4. Categorize the type of interface for the application and develop a multiplier for support classes:
Interface type Multiplier
Multiply the number of key classes (step 3) by the multiplier to obtain an estimate for the number of support classes.
5. Multiply the total number of classes (key + support) by the average number of work-units per class. There are 15 to 20 person-days per class.
6. Cross check the class-based estimate by multiplying the average number of work-units per scenario script.
Scheduling for object-oriented projects is complicated by the iterative nature of the process framework. A set of metrics that may assist during project scheduling are:
Number of major iterations: Thinking back to the spiral model, a major iteration would correspond to one 360º traversal of the spiral. The recursive/parallel process model would spawn a number of mini-spirals (localized iterations) that occur as the major iteration progresses. Lorenz and Kidd suggest that iterations of between 2.5 and 4 months in length are easiest to track and manage.
Number of completed contracts: A contract is "a group of related public responsibilities that are provided by subsystems and classes to their clients". A contract is an excellent milestone and at least one contract should be associated with each project iteration. A project manager can use completed contracts as a good indicator of progress on an OO project.

D. Tracking Progress for an OO Project

Although the recursive/parallel process model is the best framework for an OO project, task parallelism makes project tracking difficult[2]. The project manager can have difficulty establishing meaningful milestones for an OO project because a number of different things are happening at once. In general, the following major milestones can be considered "completed" when the criteria noted have been met.

Technical milestone: OO analysis completed
• All classes and the class hierarchy have been defined and reviewed.

• Class attributes and operations associated with a class have been defined and reviewed.
• Class relationships have been established and reviewed.
• A behavioral model has been created and reviewed.
• Reusable classes have been noted.
Technical milestone: OO design completed
• The set of subsystems has been defined and reviewed.
• Classes are allocated to subsystems and reviewed.
• Task allocation has been established and reviewed.
• Responsibilities and collaborations have been identified.
• Attributes and operations have been designed and reviewed.
• The messaging model has been created and reviewed.
Technical milestone: OO programming completed
• Each new class has been implemented in code from the design model.
• Extracted classes (from a reuse library) have been implemented.
• Prototype or increment has been built.
Technical milestone: OO testing
• The correctness and completeness of OO analysis and design models has been reviewed.
• A class-responsibility-collaboration network has been developed and reviewed.
• Test cases are designed and class-level tests have been conducted for each class.
• Test cases are designed and cluster testing is completed and the classes are integrated.
• System level tests have been completed.

III. DOMAIN ANALYSIS

This activity, called domain analysis, is performed when an organization wants to create a library of reusable classes (components) that will be broadly applicable to an entire category of applications.
With the reusability and domain analysis it is highly likely that
1. The project will be finished much earlier.
2. The cost of product will be significantly lower
3. The product produced will have fewer delivered defects.

A. The Domain Analysis Process

Software domain analysis is the identification, analysis, and specification of common requirements from a specific application domain, typically for reuse on multiple projects within that application domain.
The goal of domain analysis is straightforward: to find or create those classes that are broadly applicable, so that they may be reused.
Define the domain to be investigated:
To accomplish this, the analyst must first isolate the business area, system type, or product category of interest.
Next, both OO and non-OO "items" must be extracted. OO items includespecifications, designs, and code for existing OO application classes; support
classes (e.g., GUI classes or database access classes); commercial off-the-shelf (COTS) component libraries that are relevant to the domain; and test cases. Non-OO items encompass policies, procedures, plans, standards, and guidelines; parts of existing non-OO applications (including specification, design, and test information); metrics; and COTS non-OO software.
Categorize the items extracted from the domain: The items are organized into categories and the general defining characteristics of the category are defined. A classification scheme for the categories is proposed and naming conventions for each item are defined. When appropriate, classification hierarchies are established.
Collect a representative sample of applications in the domain: To accomplish this activity, the analyst must ensure that the application in question has items that fit into the categories that have already been defined.
During the early stages of use of object-technologies, a software organization will have few if any OO applications. Therefore, the domain analyst must "identify the conceptual (as opposed to physical) objects in each application."
Analyze each application in the sample: The following steps are followed by the analyst:
• Identify candidate reusable objects.
• Indicate the reasons that the object has been identified for reuse
.• Define adaptations to the object that may also be reusable.
• Estimate the percentage of applications in the domain that might make reuse of the object.
• Identify the objects by name and use configuration management techniques to control them. In addition, once the objects have been defined, the analyst should estimate what percentage of a typical application could be constructed using the reusable objects.

IV. DESIGN FOR OBJECT ORIENTED SYSTEM

The four layers of the OO design pyramid are:

- The subsystem layer
- The class and object layer
- The message layer
- The responsibilities layer

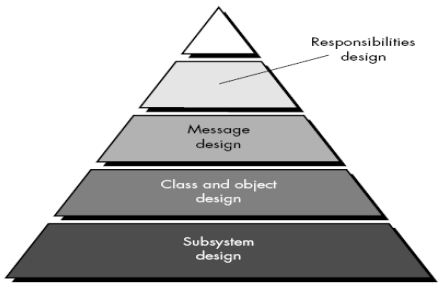

Figure1: Object Oriented System Design

The subsystem layer contains a representation of each of the subsystems that enable the software to achieve its customer-defined requirements and to implement the technical infrastructure that supports customer requirements.
The class and object layer contains the class hierarchies that enable the system to be created using generalizations and increasingly more targeted specializations. This layer also contains representations of each object.

The message layer contains the design details that enable each object to communicate with its collaborators. This layer establishes the external and internal interfaces for the system.
The responsibilities layer contains the data structure and algorithmic design for all attributes and operations for each object.

CONCLUSIONS AND FUTURE DIRECTIONS

Finally we conclude that very little theory exists for swarm-based systems and no robust systems should be deployed before we understand fundamental properties of stigmergic systems.
This section also deals with potential future research that might facilitate the need for implementation of stigmergic principles for various complex engineering problems.
Apart from the highlighted methodological problems, the domain of SI-based routing lacks of contributions falling in the two opposite areas of mathematical modeling and real-world implementations. From the one hand, simulation-based studies should be complemented with mathematical models, that allow to study very large systems and general algorithm properties, and can favor fair comparisons among the algorithms. From the other hand, simulation should be just the first step towards hardware implementations. Experiments with real test beds force the experimenter to face a wide set of problems and challenges that can hardly be replicated in simulation.

REFERENCES

1. The Unified Modeling Language User Guide by Grady Booch, James Rumbaugh, Ivar Jacobson.
2. Applying Use Cases – A Practical Guide by Geri Schneider and Jason P. Winters
3. Object Solutions – Managing the Object – Oriented Project by Grady Booch.
4. Surviving Object – Oriented Projects – A Manager Guide by Alistair Cockburn
5. Object Oriented Software Engineering – A Use Case Driven Approach by Ivar Jacobson, Magnus Christerson, Patrik Jonsson, and Gunnar Overgaard.
6. Designing Object – Oriented User Interfaces by Dave Collins.