

Object-Oriented vs Waterfall Software Development

Thomas J. Cheatham
Middle Tennessee State University

John H. Crenshaw
Western Kentucky University

Introduction.

The classical waterfall methodology for software development is a phased approach consisting of analysis, design, implementation, testing and maintenance. It has been successful in developing software with structured programming principles. *Object-oriented programming* (OOP) has been around since the development of SmallTalk in the early '70s. In 1990 there is still no widely accepted "life-cycle" that uses OOP. Yet grandiose claims have been made about the benefits of OOP. First, object-oriented programming is embedded in a software development methodology modeled after the classical waterfall approach. Then the relative merits of the object-oriented and structured programming paradigms are studied in a multi-project student experiment.

An Object-Oriented Software Development (OOSD) Methodology.

Object-oriented programming was pioneered in the SmallTalk language [5]. The main features are abstraction, encapsulation, inheritance and reuse. An entity in the problem domain is abstracted as an *object*. The object encapsulates related data (in the form of instance variables) and its operations (called *methods*). An object can be *inherited* by another object to provide reuse. The reuse is "flexible" in that the "child" object can add new operations (and data) and/or modify existing ones. In addition to SmallTalk, several other languages

provide varying levels of support for OOP: Ada, C++ , Objective-C, Object Pascal, Turbo Pascal 5.5, Clue, Eiffel, Actor, LOOPS, and FLAVORS to name a few.

An early *OOSD* approach that is losing favor with software developers begins with classical (functional) analysis and design followed by OO implementation. This method requires essentially two decompositions of the system-- first into functions and again into objects. It does not take full advantage of the object abstraction. To do so requires the use of objects earlier in the software life-cycle. Booch [3] describes a method of *object-oriented design* (OOD) and Bailin [1] suggests an approach to *object-oriented analysis* (OOA). Neither is easy to apply or widely accepted. When objects are being developed for reuse, the software life-cycle is often applied to an object (class). That is, analysis, design, implementation, and testing are performed on an individual object (*class*). In the OOSD methodology discussed in this paper, the goal was not to develop reusable components, though reuse of existing software was encouraged in both paradigms. An OO software life-cycle was defined that mimics the *waterfall* (WF) methodology except it is object centered instead of function centered. The proposed *OOSD* focuses on objects during analysis, design, implementation and testing. The deliverables at the end of each phase parallel those of the WF methodology. Tables 1-4 outline this OOSD methodology and contrast the deliverables for the two methods. The WF deliverables followed Pressman[6].

Table 1 - Analysis Document

WF Analysis	OO Analysis
* Problem description	Problem description
* Functional and non-functional requirements	Functional and non-functional requirements
* Diagrams of the system (high level data-flow, etc)	Entity-Relationship Diagrams
* Data dictionary	Entity dictionary
* Prologue describing each major function	Prologue describing each entity
* Black box system validation test cases	Black box entity validation test cases

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

Table 2 - Design Document

WF Design	OO Design
* Decompose system into modules	Develop object diagram
* Determine data structures	Determine data structures
* Develop algorithms	Develop specifications for each object
* Develop pseudo-code and/or flow-charts	Develop pseudo-code for each operation
* Develop cross-reference showing how reqs met by design	Develop cross-reference mapping functions to objects
* Develop test strategies/classes for integration	Develop test strategies for class integration

Table 3 - Implementation Deliverables

Structured Programming	OO Programming
* Develop source code in VAX Pascal Turbo Pascal C-language or other	Develop source code in Turbo Pascal 5.5 Zortech C++ AT&T C++ or other
* Debug source code including any reused (library) code	Debug source code including any reused objects
* Clean compile of source	Clean compile of source

Table 4 - Testing Document

WF Testing	OO Testing
* Unit test each function	Unit test each class
* Integration test	Integration test
* Validation test	Validation test
* Black-box system test	Black-box system test
* Regression test as needed	Regression test as needed
* Log all tests	Log all tests

In the WF methodology *unit testing* proceeds as follows. For each function, calculate the *McCabe Cyclomatic Complexity*, determine a set of basis paths, a set of test

cases/data, and expected as well as actual results. As a rule functions with a complexity greater than 10 should be decomposed.

In the OOSD methodology, unit testing focuses on a class of objects. For each class, test its member functions as above. Test all constructors/destructors and overloaded operators. Calculate the *class cyclomatic complexity* as the sum of the complexity of its methods excluding inherited methods. As a rule, if a class has a complexity greater than 100, it should be decomposed using inheritance. Integration test all methods within the class. Be sure the class specifications are met.

The *object diagram* of Seidewitz and Stark[7] provides a guide to object integration in the OOSD methodology as described in Cheatham and Mellinger[4].

In both methodologies, a "test log" entry should specify who, what, when, inputs, outputs both expected and actual, and conclusions.

Claimed Advantages of OOSD.

Every new idea in computing generates more "media hype" than the last, often promising more than it can deliver. Associated with the "object-oriented" buzz words are familiar claims such as

- OOSD offers increased productivity
- OOSD provides reduced complexity
- OOSD requires fewer lines of code
- OOSD produces reusable software
- OOSD is faster
- OOS is easier to debug
- OOS is easier to maintain

An experiment involving seniors and graduate students, described in the next section, was conducted to test some of these claims. The experiment is modeled, in part, after the multi-project experiment of Boehm, et al [2].

The Experiment.

Six teams were selected from a senior/graduate level projects class of 18 students. The teams were selected by a program that matched pairs of teams based on the *least team difference* in

- (1) Credit hours of Computer Science (CS) course work,
- (2) Cumulative GPA (undergraduate or graduate) in CS,
- (3) Number of months since first learned to program,

- (4) Number of months since first learned Pascal,
- (5) Number of months since first learned Turbo Pascal,
- (6) Number of months since first used a micro-computer.

That is, a pair of 3-member teams was chosen from the remaining pool of N students such that

$$(\bar{X}_1 - \bar{Y}_1)^2 + (\bar{X}_2 - \bar{Y}_2)^2 + (\bar{X}_3 - \bar{Y}_3)^2 + (\bar{X}_4 - \bar{Y}_4)^2 + (\bar{X}_5 - \bar{Y}_5)^2 + (\bar{X}_6 - \bar{Y}_6)^2$$

is minimal over all pairs of teams (X, Y). Here \bar{X}_i and \bar{Y}_i represent the average of (1) to (6) for team X and Y respectively.

Thus, three pairs of 3-member teams were created: Teams 1 and 2, Teams 3 and 4, and Teams 5 and 6. Even though no attempt was made to guarantee that there was no significant difference among all six teams, the differences in the six teams turned out to be statistically insignificant (with ANOVA at the 10% level). Instead, the intention was to guarantee that, within each of the three pairs of teams, both teams were essentially equivalent.

Table 5 below presents the scores for the six teams on the selection criteria.

Table 5 - Team Selection

Team/Paradigm	Selection Criteria					
	1	2	3	4	5	6
1. WF1	27	3.4	64	43	29	78
2. OO1	30	3.4	59	39	23	73
3. WF2	33	2.9	93	36	28	73
4. OO2	31	3.1	93	38	27	73
5. WF3	32	3.1	90	53	29	76
6. OO3	40	3.5	100	51	31	67
Average of WF	31	3.1	83	44	29	76
Average of OO	33	3.3	84	43	27	71

The two teams in a pair of teams were assigned the *same problem*. Based as much as possible on the preference of the team members, one team was asked to use the traditional WF methodology while the other used the OOSD methodology described in the previous section. During the first six weeks of the course all students were exposed to both methodologies. This was the first time many of the students had studied OOSD. Very few students had written a program using objects. A student who knows the C-language can certainly make a contribution to implementation in C++ even if he has no experience with C++ per se. A similar statement is true for Turbo Pascal and objects in Turbo Pascal 5.5.

In a *projects class*, choosing problems of the correct size and level of difficulty is not easy. It was complicated by working in two paradigms. Our philosophy was to err on the "too difficult" side and if necessary, reduce or relax the requirements. We felt the experiment would be more meaningful if we had three pairs of teams working on three different problems:

- Problem (1)** which favors a WF solution,
- Problem (2)** which is middle-of-the-road, and
- Problem (3)** which favors an OO solution.

This is not an easy assignment and at best is subjective. Even after solving a problem using both methods, it may be debatable which is better, easier, etc. (for that problem). It is generally agreed that the "user interface" is a natural place for objects. So a system with a heavy emphasis on the user interface may favor the OO approach. Of course the user interface is a significant portion of most interactive systems.

The following three problems were chosen to match 1 - 3 above.

- Problem (1):** a system to track graduate student progress.
- Problem (2):** a simulation of a 24-hour automatic teller.
- Problem (3):** a full-screen editor modeled after VAX/EDT.

Teams 1 and 2 (*WF1* and *OO1*) were assigned Problem 1. Teams 3 and 4 (*WF2* and *OO2*) were assigned Problem 2 and Teams 5 and 6 (*WF3* and *OO3*) were assigned Problem 3.

The easy way out for the professors would be to select the problems, assign the teams and leave the rest to them. This would not be fair to either methodology. During the first six weeks, each phase of the life cycle was discussed, a brief lab was assigned, and the project deliverable for the phase was defined. The students *did not know* which paradigm they would be assigned so they were motivated to understand both. Extra time was spent on the OOSD

methodology since it was not as well defined or understood.

The teams had nine weeks to complete their project with two weeks each allocated to *analysis*, *design*, *implementation* and *testing*. This left one week for (external) documentation. Table 6 shows the schedule of deliverables.

Table 6 - Schedule of Deliverables

Date	Activity Completed	Deliverable
Mon 3/12	Analysis	Analysis Document
Wed 3/14	Our Reactions	
Mon 3/26	Design	Design Document
Wed 3/28	Our Reactions	
Mon 4/9	Implementation	Executable & Source
Wed 4/11	Our Reactions	
Mon 4/23	Tested System	Test Document & System
Mon 4/23	Acceptance Test	Classroom Demo
Wed 4/25	Acceptance Test	Classroom Demo

Due to time constraints, documents were not updated by the teams to reflect our suggestions. However, it was expected that our suggestions would be reflected in the next deliverable. Every Monday at 4:00 pm, each group was required to submit a single time sheet signed by all three team members.

Both professors read and commented on all deliverables. The amount of reading required between a Monday delivery deadline and our reactions on the following Wednesday was horrendous. And, often, the flood of red ink was unappreciated.

Results.

There is general agreement that a solution to a problem may be easier to implement in one language than another. For instance, a problem solution involving set manipulation is easier to implement using Pascal than it is in COBOL because Pascal has a built-in set data type and corresponding operations. COBOL has neither. More research needs to be done to determine if some problems are easier to solve in one paradigm or another. Specifically, which problem types are easier to solve in the OO paradigm? This study reveals some insight as seen in the comparison of total development man-hours for the three problems in Table 7.

Table 7 - Man-Hours

Problem	Conjecture	Total Man-Hours	
		WF	OO
1. Grad. Tracking	Favors WF	199	597
2. ATM Simulation	Middle	260	397
3. Editor	Favors OO	451	242
Average		303	412

The larger average for the OO teams does not support the claim that OOSD is "*faster*." The three-to-one ratio of man-hours in Problem 1, can be partially explained. This team (OO1) wanted to debate every issue in a group setting, so took four times (323 hours verses 89) longer in analysis and design than their counterpart (WF1). It is noteworthy that the OO solution to this problem in a typical WF domain required fewer lines of code (1850 verses 2325).

The lack of experience with OOSD is another factor that must be considered. It takes experience to become efficient at solving problems with objects. Our students lacked this experience. On the other hand, implementation of the solution took the same amount of time in OOSD and WFSD (334 man-hours verses 321, on the average) in spite of the lack of experience with object-oriented programming. Table 8 gives the man-hours by phase. Using *analysis of variance* (ANOVA, 10% level) the differences are not statistically significant.

Table 8 - Man-Hours by Phase

Phase	Team							
	WF1	WF2	WF3	WFavg	OO1	OO2	OO3	OOavg
Analysis	32	60	69	54	131	86	45	87
Design	57	46	59	53	192	38	43	91
Implement	29	25	267	107	59	205	70	111
Debug	32	77	8	39	95	19	23	46
Test	32	33	39	36	88	37	54	60
Document	17	19	9	15	32	11	7	17
Total	199	260	451	303	597	396	242	412

The OO solutions were, on the average, 10% smaller in delivered source instructions (DSI) -- 15% if (unmodified) "reused" code is not counted. That is, the OO solution required 15% less original (or modified) code. But the OO teams showed lower productivity regardless of how it is measured. See Table 9 below. None of the differences were statistically significant.

Table 9 - Productivity

Measure	Team							
	WF1	WF2	WF3	WFavg	OO1	OO2	OO3	OOavg
DSI	2325	1063	2706	2031	1850	1345	2353	1849
DSI/Total Hrs	12	4	6	7	3	3	10	5
DSI/Impl. Hrs	80	43	10	40	31	7	34	24
(*)DSI - Reuse	1961	1063	2706	1910	1850	1345	1691	1629
(*)/Total Hrs	10	4	6	7	3	3	7	4
(*)/Impl. Hrs	67	43	10	40	31	7	24	21
Documen Pgs	94	167	151	137	210	143	94	149
Pg/(Hr-Imp-Db)	.7	1.1	.9	.9	.5	.8	.6	.6

Documentation pages include analysis, design, and test documents plus user's guide and internal comment lines. The only "quality of product" measurement we have is the average team grade on all the deliverables. The average for the three WF teams was 83 and 81 for the OO teams. No team did a satisfactory job of testing. All six systems were delivered with problems.

Conclusions.

More experimentation should be done! From this experiment, it appears that some problems are more amenable to OO solution than others and that one's gut feeling may be enough to tell the difference. In fact, *it may be easier to classify a problem by paradigm than by size*. We did a fair job of the former but a poor job of the latter with our three problems. The OO solutions required less code but took longer on the average. The low usage of reused code and the lack of experience with OOSD may account for the lower productivity. Statistically, none of the differences are significant.

REFERENCES

1. Bailin, S., An Object-Oriented Requirements Specification Method, Communications ACM, Vol 32, 1989, pp. 608-623.
2. Boehm, B. W., T. E. Gray, and T. Seewaldt, Prototyping Verses Specifying: A Multiproject Experiment, IEEE Trans. Soft. Eng., Vol SE-10, No 3, 1984, pp. 290-302.
3. Booch, G., Object-Oriented Development, IEEE Trans. Soft. Eng., Vol SE-12, 1986, pp. 211-221.
4. Cheatham, T., and L. Mellinger, Testing Object-Oriented Software Systems, Proceedings ACM CSC'90, 1990, pp. 161-165.
5. Goldberg, A., Smalltalk-80, The Interactive Programming Environment, Addison-Wesley, Reading, Mass., 1984.
6. Pressman, R. S., Software Engineering: A Beginner's Guide, McGraw-Hill, New York, NY, 1988.
7. Seidewitz, E., and M. Stark, Toward a General Object-Oriented Software Development Methodology, Collected Software Engineering Papers, Vol 4, No 14, 1986, pp. 25-38.