The Methodology of Methodologies; or, Evaluating Current Methodologies: Why and How

Tim Menzies
Artificial Intelligence Laboratory, School of Computer Science and Engineering,
University of New South Wales
PO Box 1, Kensington, NSW, Australia, 2033
timm@cse.unsw.edu.au

Philip Haynes
Group Treasury,
State Bank of NSW,
Level 40 Grosvenor Place, 225 George Street, Sydney, NSW 2000
philip@dev.state.com.au

ABSTRACT: A good methodology should be an accurate *description* of sound software engineering (SE) practice. Without empirically-supported method-ologies, we run the risk of using potentially inaccurate *prescriptions* of the software engineering process. We reject arguments that SE is unmeasurable. Related fields, such as knowledge acquisition, routinely perform repeatable experiments on specification development and maintenance. A sample of these results are presented here. For a devotee of the OO approach such as ourselves, these empirical results are very counter-intuitive. They motivate our call for a thorough empirical investigations of all the truisms of object-oriented (OO) SE such as: (i) OO is better than functional decomposition; (ii) OO promotes re-use; (iii) OO programs are easier to maintain and have fewer errors than alternative approaches; and (iv) OO is currently our best technique for SE. As a starting point for these investigations, we include designs for several experiments. Many of these experiments could be performed by commercial practitioners.

1. Introduction

On what basis do we decide on the myriad issues that preoccupy us such as (i) multiple vs single inheritance; (ii) the relative merits of templates or deriving from root; and (iii) how to decide if a method is private /protected/ public? More fundamentally, do the issues we debate *really matter*? Are we currently focused on issues that are truly relevant or are we bogged down on minor details? What are the real factors that crucially determine good design, fewer errors, faster delivery, and easier maintenance?

We reject answers to these questions based on 'intuition", "the voice of experience", "everybody knows" or "it is widely accepted". Well-argued beliefs widely-held in a community may not survive attempts at experimental verification. For example:

- Galen's descriptions of human physiology, written in 200B.C., were studied as virtual gospel for millennia. However, no one thought to check Galen's descriptions until one upstart surgeon had the gall[1] to pick up a scalpel and perform dissections for himself. Versalius's *De Humani Corporis Fabrica*, published in 1543, showed that many of Galen's descriptions were inaccurate.
- Henri Fayol told us in 1916 that managers plan, organise, co-ordinate and control. This model of management lasted nearly six decades until proved inadequate by the empirical observations of Henry Mintzberg [17]. Fayol's terminology reflects the vague objectives of a manager but are inadequate for characterising the day-to-day activity of managers[2].
- It is usually accepted that structured programming are a "good thing". However, surveys of published empirical results relating to this issue are inconclusive. Further, the experimental methods used to collect those results is questionable [6].

Galen and Fayol offered *prescriptive* views of what should be. Versalius and Mintzberg offered us *descriptive* views of what is, based on empirical observations.

Debates regarding appropriate OO SE techniques occupy much of our time. We fear that many of

[1] Pun intended.

[2] For example, a study of 56 U.S. foreman found that they averaged 583 activities in an eight-hour shift (one every 48 seconds). Another study of 160 British middle and top managers found that they worked for half an hour or more without interruption only once every two days. These empirical results of actual managerial behaviour does not fit Fayol's model of managers as systematic planners [17].

these discussions are prescriptive. The empirical evidence supporting our current generation of OO SE methodologies is very weak [6]. Lacking experimental results, we cannot authoritatively state that any one methodology is better than any other. Further, certain empirical results in the related field of knowledge acquisition (KA) that run counter to the intuitions and truisms of OO. We therefore call for a thorough empirical investigation of all these truisms, e.g.: (i) OO is better than functional decomposition; (ii) OO promotes re-use; (iii) OO programs are easier to maintain and have fewer errors that alternative approaches; and (iv) OO is currently our best technique for SE. As a starting point for these investigations, we include designs for several experiments. Many of these experiments could be performed by commercial practitioners.

Note that while article is aimed at an OO audience, its general argument applies to much of the software enterprise. The current experimental basis for belief in *any* methodology is dubious (for a very short list of the current evidence, see [6]). Consequently, proponents of alternatives to OO (e.g. functional decomposition) should not read this article as an endorsement of their proposals.

Section two reviews the KA results. Section three describes several repeatable SE experiments. Section four makes some general points regarding experimentation. Our appendix includes sample problems suitable for reproducible software engineering experiments at a university.

2. Interesting Empirical Results

This section discusses certain empirical results from the knowledge acquisition (KA) community. For a devotee of the OO approach, these results will be somewhat counter-intuitive. We present them as motivation for our general call for empirical investigations of all aspects of the OO SE process.

The goal of KA is some high-level representation of an expert's reasoning processes. To convert KA into conventional SE, perform the following global substitutions: (i) "upper-CASE tool" for "expert system shell"[3]; (ii) "analysis" for "knowledge acquisition"; (iii) "specification" for "knowledge base"; (iv) "maintenance" for "knowledge maintenance". Also, add a decent search mechanism; i.e. a fast OO-SQL language which can search over all instances of a class or its subclass, accessing code or data as required. This search engine should be support a variety of intelligent search techniques [19].

3 That is, expert systems shells focus on the explicit representation of the high-level logic of the system, not the low-level implementation details.

2.1. Re-use Results

As far as we are aware, the best published demonstration of increased productivity using re-usable components is DEC's SPARK/ BURN/ FIREFIGHTER (SBF)[13]. SBF was an experiment in providing automated support for the entire software life-cycle, from specification to maintenance. SPARK was a meta-CASE tool that automatically configured a problem-specific CASE tool using diagrams supplied by a business user. BURN executed the problem-specific CASE tool which assisted business users to fill our the details of their domain. BURN generated an executable version of the program which was run and debugged using the FIREFIGHTER tool. Using the SBF toolkit, the time required to build a range of expert systems was dramatically reduced (e.g. in one case, 200 days to 7). For more details on SBF (and its limitations) see [14].

Unfortunately for proponents of the OO approach, SBF was developed via a functional decomposition paradigm. A reverse engineering of numerous expert systems suggested that a small number (less than two dozen) of high-level functions were re-used in many applications. Examples of these functions included *classify*, *specify*, *compare*, and *match*. SBF was built on top of such a library.

We know of no other publications in the OO literature demonstrating a higher level of productivity and re-use. Quite the reverse, in fact. At Tools '92, [15] claimed that examples of reusable OO libraries are the rare exception rather than the rule. This claim was not challenged by the conference referees or attendees. Since writing that article, we have found only two publications in an international refereed conference or journal that empirically studies the reuse issue: Lewis *et. al.* [10] and Stark [22]. Lewis *et. al.* took a group of university senior-level SE students and had them build applications. The students were divided into four groups according to two dimensions:

- With and without a class library of components relevant to the problem at hand
- Using C++ or Pascal; i.e. with and without the object paradigm

One of Lewis *et al*'s conclusions that if programmers do not reuse code, then the OO paradigm does not promote higher productivity that the functional decomposition paradigm (i.e. Pascal). Combining this result with the experience of [15] (i.e. reusable libraries are the rare exception), we see that claims that OO is more productive than the functional decomposition paradigm require further empirical analysis.

The Stark study is an example of such an empirical study. Unfortunately, Fenton *et. al.* characterise the Stark study as interesting, but with a poor experimental design [6]. The study discusses the impacts of OO technology over seven years and eleven software projects at NASA. Verbatim code re-use of 90% of ADA source code from prior projects is reported. Stark gives little detail on the nature of the projects, so this claim is unconvincing. In other results, described in better detail, Stark reports increasing code reuse from a 20-30% base line (before OOT) to 75-80%. This increased level of reuse arose from a reorganisation of some FORTRAN code:

> *(The developers) developed separate interface routines and file formats for each kind d sensor. Only a mission-specific front-end telemetry processor had to be developed for new missions. [22]*

Stark notes that this OO-style reorganisation occurred not at the analysis or design phase, but during the final coding phase. Stark reads this result as an endorsement of OO in general. A more critical observer could read it as evidence that the real benefit in OO is a general organisation principle that is language/methodology independent. Perhaps some subset of our current OO technology is the real source of its power. If so, then we should dispense with possibly-superfluous and complicating details (e.g. classes, inheritance, automatic garbage collection) while still applying "OO" to existing techniques (e.g. FORTRAN). We make no comment here regarding the "true" conclusion from the Stark study, except to note that more experimentation is required.

2.2. Maintenance Results

There are few empirical studies of software development with even fewer studies of maintenance. One result from KA community is the variability of a specification. There seems to be no "correct" model of any domain. Rather, specifications seem to be a construct that varies according to who says, and when. [21] took a group of geology experts and had them construct knowledge bases for the same problem. The experts then reviewed each other's knowledge base and, after 12 weeks, their own. Table 1 shows that experts may disagree significantly about what constitutes a "correct" specification. For example, experts only agreed with each other, at best one-third of the time.

Expert pairs	*Understands (max = 100)*	*Agrees (max = 100)*
E_1,E_2	62.5	33.3
E_2,E_1	61.1	26.7
E_1,E_3	31.2	8.3
E_3,E_1	42.9	33.3
E_2,E_3	44.4	20.0
E_3,E_2	71.4	33.3

Table 1: E_x , E_y *denotes a review by expert* E_x *of expert* E_y*'s specification.*

Table 2 shows the expert's assessment of their own knowledge base, 12 weeks after they wrote it. When old knowledge is reviewed, it may be found wanting. For example, expert 1 could only understand 62.5% of what he'd written 12 weeks before. All experts disagreed (to some extent) with their own ideas from the past (as shown in the *Agrees* column of Table 2).

Expert	*Understands (max = 100)*	*Agrees (max = 100)*
E_1	62.5	81.2
E_2	77.8	94.4
E_3	85.7	78.6

Table 2 *Self-review of a specification, 12 weeks after it was written.*

Tables 1 and 2 suggest that maintenance is a non-trivial issue. Experts may change their mind. What was "correct" before may now be considered "wrong". This is a major problem for program maintenance (the moving target syndrome).

Techniques for taming the maintenance problem may not be a simple add-on to existing methodologies, but may require a fundamental restructuring of the software process. For example, PIERS is an expert system for interpreting biochemistry results in routine daily use at St. Vincent's Hospital Sydney [20]. PIERS's expertise covers 20% of the biochemical tests performed at the hospital. The system processes 500 cases per day at 99% accuracy. PEIRS contains 1380 rules and is one of the largest expert systems in routine use in the world today. The system was built and is maintained using a structured patching methodology called RDR [2]. Whenever a case results in an inappropriate conclusion, the patch knowledge is entered in as an *unless* test beneath the rule that resulted in error. As the specification develops, it grows into a binary tree with knowledge patches stored at every node (see figure 1). At runtime, the final conclusion is the conclusion of the last satisfied node.

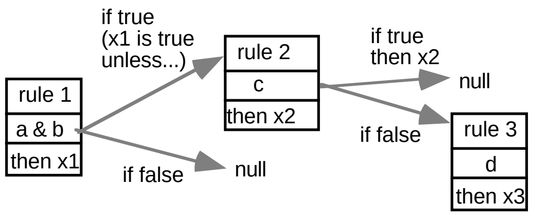

Figure 1: *An RDR-tree. At runtime, the output conclusion is the conclusion of the last satisfied node.*

The advantage of this approach is that knowledge that works is never changed. New knowledge is always an addition to the specification, never a re-write. This is an application of the heuristic: "if it ain't broke, don't fix it".

Large expert systems are notoriously hard to build and maintain [23]. Neither of these problems were found to be true with structured patching of PIERS. Development was simple. Minimal preliminary analysis was performed. After the database connections were made (using standard software engineering techniques), experts just considered the cases present on a particular day and told PIERS what to say for each such case. Maintenance time is constant (2-6 new rules per day) and very simple (a total of a few minutes each day).

RDR can be characterised as a "minimal-analysis" tool. After some initial analysis, the system goes live and is patched in the context of its errors. Global re-organisations are forbidden. Our pre-experimental intuition was that such global reorganisations are an essential precondition for good software. If so, then OO analysis should be able to deliver a better PIERS than using RDR.

The following thought experiment compares OO analysis with RDR. Assuming the use of OO SE methodologies, generate time estimates for building and maintaining a system that models 20% of human biochemistry sufficiently to make diagnoses that are 99% accurate. Factor into your time estimates:

- *The changing nature of the field.* PIERS was originally built for one "domain" (thyroid tests). It now covers a dozen domains.
- *The hypothetical nature of the problem.* Many areas of human biochemistry are open research issues.
- *Knowledge of prior unsuccessful attempts.* Prior experiments in the same domain used extensive preliminary analysis but never made it out of the prototype stage [18].

Compare your time estimates to the development and maintenance effort of PIERS[4] using RDR. If your OO estimates for development and maintenance are greater than RDR then start doubting the value of extensive analysis.

The PIERS conclusion is that, given a structured patching environment, extensive preliminary analysis is a *optional* part of the software development life-cycle. This conclusion is counter-intuitive and somewhat controversial. If this result generalises, then entire communities of software analysts face unemployment as users build their own systems using structured patching. However, opponents of the PIERS conclusion have little empirical evidence to support their opposition. Experiments with OO maintenance are described below.

[4] PIERS's interface, inference engine, and database connections were built by one part-time PhD student between June 1990 and May 1991. PIER's rules were built by a doctor in fulltime practice, during his spare moments. The doctor and the PhD student interacted occasionally. After the system went live (May 1991), the doctor spent a few minutes each day (< 30) extending the rules. The specification of the diagnosis knowledge has since grown from 200 rules to 1380 rules (690%).

3. Reproducible SE Experiments

This previous section described empirical results that challenge the supremacy of the OO paradigm. This section describes experiments for replying to that challenge. Note that all but the last one could be performed in a commercial context.

3.1. SE Experiments are Hard (?)

A common reaction when we discuss empirical experimentation in SE is "SE experiments are too hard". For example, the [7] study was hard work. However, no matter how hard are SE experiments, not performing them is even harder. If we don't understand what we have done right or wrong in the past, then we run the risk of repeating past mistakes and ignoring past successes.

Experimentation can be simplified by up-front experimental design. Most of the difficulty of the [7] study were from having to collect statistics in a post-hoc manner. We describe techniques below for data collection in parallel with software development.

Prior to that discussion, we note that post-hoc analysis is not always impossibly difficult. For example, the following study took 2 months elapsed time, but less than seven days actual time. One of us (Haynes) performed all the experimental work as a background task while working as a fulltime software engineer.

[8] reports an analysis of five Smalltalk systems that found a simple linear relationship between number of objects referenced by a class and the source lines of code in that object (OSLOC). Smalltalk's untyped variables makes determining "objects referenced" problematic. The *butterfly heuristic* makes a reasonably accurate guess as follows:

- Given a set of objects $O_1,\ldots O_i$ defining features $F_1\ldots F_j$, the set $UNIQUE \subset F$ represents the features defined in only one object.
- Determining what object is referenced by a call to a method $Met \in UNIQUE$ is trivial. A table *Refs* of the number of references an object makes to another object is computed for the *UNIQUE* method calls. From *Refs*, for each object O_i we can compute $O_{i.common}$: the object referenced most commonly by O_i.
- The objects referenced by $Met \notin UNIQUE$ is indeterminate; i.e. we have to guess. The butterfly heuristic says that object references act like migrating butterflies: where they land commonly is where they will land always. The

butterfly heuristic assigns indeterminate object references in O_i to $O_{\mathrm{i}.common}$.

- This heuristic was verified by a statistical analysis of the differences between heuristically generated object reference tables and tables built by hand. No significant difference could be detected [8].

Once the heuristic was confirmed, several Smalltalk applications were compared[5]. The following relationship was empirically determined:

$$OSLOC = references/m - b$$
$$0.024 \leq m \leq 0.036$$
$$3.8 \leq b \leq 7.2$$

Such an equation could be used for software size estimation. Combined with knowledge of OSLOC/hour for the programmers assigned to a project, this equation could then be used for detailed time estimation.

[8] argues that further data collection and better experimental technique could enhance the precision of this expression. The significance of this result to our discussion here is that it was generated in a small amount of time using existing resources. It is a reproducible experiment. The conclusions of the experiment are clear. The design of an experiment to attempt to falsify this result is obvious (i.e. measure other programs).

Summary: SE experiments are not necessarily hard. Empirical data can be found *if you care to look for it* and this may not be a long search.

3.2. Size Estimators

We have offered above a linear relationship between number of objects reference and OSLOC. This relationship could be explored in other applications. In environments like Smalltalk that support automatic tools for accessing method references, then it should be a simple matter to apply the butterfly heuristic and confirm/ refine/ refute this formula.

If versions of the design documents are available, then it would also be interesting to compute Δ, the difference in objects referenced between the initial design and the final code. If a definite pattern can be detected in Δ, then size estimation could be performed on initial design documents.

3.3. Time Estimators

If information is available on OSLOC/hour for different programmers, then we could convert size estimates into time estimates based on the available programmer population. Such statistics could be simply collected:

1) If the source code for the editor of the development environment was accessible, the each key stroke (including backspace and accounting for cuts and pastes) could be calculated automatically. Whenever a new method was saved, its size delta could be logged. Logs could be refined for method creation, changing, deletion.
2) If the source code is stored in text files, then the size of the output of a UNIX *diff* could be collected each day for each programmer. Note that this method #2 is second-best to method #1 since it is not as fine-grained as method #1.

In keeping with the general theme of this paper, we argue that such measurements should be made with a view to assessing some active hypothesis. If developers kept paper time sheets that recorded their work times, then time estimates made using this technique could be compared with actual time. This technique would be rejected if the actual vs estimated time were statistically different.

3.4. Error Logs

If developers can customise the exception handlers of their environment, then a log could be appended to whenever an exception is raised. See the appendix of [16] for sample code that customises the Smalltalk exception handler.

The error log could have at least the following entries:

errors(time, date, object, method)

where *object* and *method* are the object and method that raised the exception. Ideally, some database keeps a log of error message texts and current users. The error log could then be usefully extended to:

errors(time, date, object, method, user_id, error_id)

Recording *user_id* implies some method of signing users on/off to the system.

Once such error logs are available, and if we can track which programmers developed which classes, then OSLOC/hour for a programmer could be more realistically assessed vs Errors/hour in the code written by particular programmers.

3.4. Surrogate Error Measures

Note that errors detected in the above manner are a subset of all possible errors. For example, an incorrect result resulting from an incorrect design may be silently written to a database without calling the exception handler.

The total set of errors can only be determined by a visual inspection of a program. Such a process is slow. It must be done by the top experts in the code/business problem. Hence, it is very expensive.

An interesting result would be the development of a surrogate error measure. That is, some automatically

5 One locally generated application, and four others (some taken from the Manchester FTP goodies library st.cs.uiuc.edu).

measurable quantity (e.g. calls to the exception handler) that is related to the number of total errors.

To assess the utility of the above error log as a surrogate measure, we could compare the above error logs to total error lists generated from a sample set of applications.

3.5. Environment Statistics

Consider a development environment that logs errors and OSLOC/hour as described above. Add to this environment a facility to record the number of runs of the program (e.g. every time a method was called that was created or changed during a development). Such an environment could automatically log most of the [10] statistics during development, without requiring additional effort on the part of the programmer. However, instead of measuring Lewis *et al* 's "Time to Fix Run Time Errors", we would propose an alternative "error half-life" measure calculated as follows:

- Recall that each error has a unique id.
- Divide the development period up into N time periods.
- For each error id, create a 2-D plot of time vs errors per time period.
- Fit a curve. The parameters of that curve are the error half-life (called a half-life since our pre-experimental intuition is that the curve will be a decaying exponential).

Different methodologies could now be assessed according to the different error half-lives they generate.

3.6. Use-Case Library Experiments

Jacobson defines a "use case" as a typical use of a program (e.g. creating a new account or looking for a bad loan risk). Such use cases are typically at two levels: (i) the business user level which describe some business event; and (ii) programmer level which describe some internal processing (e.g. accessing a record via the database interface). A use-case description should be detailed enough for an outsider to be able to look at the program's operation and categorically state that the program can/can not run that use case. Use cases are a "large-grain" function point. According to Jacobson, 20-30 use cases represent about 3 months development [9].

Use cases can be used for several purposes. Analysis can proceed, in part, by the development of business level use cases. Designs can be reviewed with respect to the use case library; e.g. what percent of the use case library can be handled by the current design? Mapping changes to the use case library vs time can highlight design creep. Metrics for managing evolutionary development can also be generated from the use case library: (i) a continually growing use case library means runaway design creep; (ii) developers can have short-term goals such as "implement these 30 use cases in the next 3 months"; (iii) end-of-project can be estimated by charting the percentage coverage of the use-case library vs time; and (iv) business-users can gain a perspective into the development by assessing the current handling of the business-level use cases.

Use cases can also be used for software engineering experiments. If each use case is assigned a unique id and marked as "done" when satisfactorily completed, then we can chart "Number of completed use cases per fortnight". Different methodologies could then be assessed according to how quickly they can implement use cases. Further, using our knowledge of OSLOC/hour and error half-life for the programmers involved, we can adjust the results to account for programmers of different speeds and reliability.

3.7. Comparing "Lab-Rats"

The attendees of Tools '94 have two populations of "Lab-Rats" at their disposal: university students and commercial programmers. Lewis *et al.* base their results on university students, arguing that the students behaviour was not too different to commercial programmers. If this was true, then repeatable SE experiments under controlled conditions could be performed on university classes and generalised to commercial programmers. However, Lewis *et al*'s experimental evidence for this is brief and could be criticised. One important difference between the projects conducted in university and industry is *scale*. Computer science has been sarcastically described as the study of programs less than 3000 lines long[6]. Commercial projects can be much bigger. Factors that grow non-linearly with size become crucial in large systems, but can be missed in small systems.

If software was developed in environments that automatically computed error half-lives, OSLOC/hour/programmer, and use-case coverage, then we could compare our two populations of "lab rats". Not only would such a study be insightful regarding teaching methods, it would also serve to confirm/refute the Lewis *et al.* claim. The optimum result for such a study would be a set of *mapping parameters* that let us map the measurable behaviour in university lab rats to commercial lab rats (e.g. university students OSLOC =*x* times commercial OSLOC).

3.8. Development Experiments

Organisations could develop the same project with different programmer populations using different "tools" (i.e. methodologies, languages), collect the above statistics, and report on the impact of different approaches on program errors and delivery rates of use-cases.

6 About the largest code chunk that can be built and rigorously analysed in the space of one PhD.

If the Lewis *et al* assumption (see last section) proves to be correct, then multiple experiments could be quickly conducted amongst university students. If the mapping parameters (discussed above) are known, then results from these university experiments could be applied to commercial problems.

3.9. Maintenance Experiments

Experiments with software maintenance could also be conducted, given the above environment. Methodologies that give rise to fewer bugs could be favoured for maintenance. Bug reports and change requests could be mapped into new use cases. The ability of different approaches to handle extensions to the use case library could be charted and compared.

3.10. Functional vs OO

While collecting data to assess the Lewis *et al* assumption, university researchers could perform numerous SE experiments in the SE classes. One pressing experiment is a comparative analysis of OO and functional decomposition approaches:

- Students must first demonstrate proficiency in functional decomposition and OO methods (e.g. passing a prior subject on OO and functional design *and* coding[7]).
- Students must then maintain someone else's code written using either a functional decomposition or object-oriented paradigm (denoted P_1).
- Students must then build a new system using a paradigm P_2 ($P_1 <> P_2$).

Note that students would still be trained in multiple approaches (i.e. their education would not suffer as a result of their participation in the experiment). Further, their experience in maintaining a system would motivate their use of a methodology for development.

We list here some pragmatic issues:

1) The OO and functional decomposition development should take place in an environment that collects the parameters discussed above. Lewis *et al* used logs written by hand by each student. We feel that automatic error logging is better, but hand collection would suffice if this was not possible.
2) Given the time involved, this may be a full year course.
3) Given the need for lab-rats to demonstrate proficiency in two paradigms prior to their participation in this study, this would be an experiment for third-years or post-graduate students.
4) Given the last point, the pre-requisite subjects must be co-ordinated with this SE subject; i.e. to teach the object and functional decomposition language and (ideally) do so in the session before the SE subject is conducted.
5) Given the need to maintain programs in the first half of the study, the programs to be maintained must be available. Developing such programs, or acquiring them, must be done prior to subject commencement.
6) Given the need to study Δ (see above discussion on *Size Estimators*), students doing the coding must supply an initial design prior to writing a line of code.
7) The functional decomposition and OO languages must be chosen with care least we confound paradigm issues with other issues. For example, Lewis *et al.* compared Pascal development with C++ development since certain OO language (e.g. Smalltalk) have a development environment far superior to the simple text editors typically used for C++ source. Also, both Pascal and C++ are strongly-typed.
8) In order for meaningful comparisons to be made for the coding problem, the same problem should be implemented by different groups. Suggested coding problems are discussed in our appendix.

We note that an OO extension to Lisp, CLOS, also runs in the same interactive environment and uses many of the same debugging tools. Students using the functional decomposition approach should be forbidden to use `defclass`, `:include` in a `defstruct`, `funcall`, `apply`, and `eval`. Students using the OO approach would be required to create a minimum number of classes C with at least a certain number of methods per class M. Actual figures for C and M would have to be dependant on the application.

We advocate Lisp/CLOS since these systems have customisable exception handlers, whereas standard Pascal implementations do not. Further, we know of numerous Lisp/ CLOS Internet FTP sites where exemplar programs can be accessed. Lastly, sophisticated and free Lisp/CLOS environments are available for many platforms. Hence we suggest Lisp and CLOS for this study.

4. Warnings

4.1. Avoid Shotgun Experiments

We strongly caution against performing SE experiments without an active hypothesis. Courtney and Gurstafson describe such "shotgun experiments" as follows:

> *…a hypothesis is not stated. The researchers are experimenting with many different aspects of software projects …researchers are loading*

[7] We stress the need for our lab rats to demonstrate adequate proficiency in *both* analysis *and* coding skills.

numerous variables and taking a shot to see if they hit anything. …The standard shotgun approach is to correlate as many measures as possible against one or more dependent variables…

Courtney and Gurstafson caution "the likelihood of finding accidental relationships is high" and demonstrate this via an analysis of 15 random variable[8]. Numerous spurious correlations were "discovered" between the random variables [3].

For example, the experimental design of [5] has certain shotgun-ish features. They offer a set of automatically generated metrics for class coherence. However, they make only a short and untested comment (in their section 3.4.3) on how to use their metrics to generate value judgements regarding good/bad design. For our purposes (i.e. the detection of the essential features that can generate good software), their work is therefore incomplete.

Theory informs data collection and no theory results in very poor data collection. We have been careful to describe the goals of our experiments: (i) software size and cost estimation; (ii) methodology evaluation via use-case delivery time and error half-lives. In our framework, hypothesis can be stated, then confirmed or rejected.

4.2. Time for Experimentation

It will take *at least* three to four years of trying before we can generate definitive empirical results. We base this estimate on experiments in the KA community in reproducible expert systems development [11, 12]. Initially proposed in 1990, the Sisyphus[9] project took 1 year and 2 attempts to generate a problem that motivated a wide community of researchers. In all, 14 different approaches were proposed for two problems by different groups. The implementation and analysis of various approaches for the Sisyphus-1 problem took another year. These approaches were all presented at the same conference. A general consensus at the end of that first project was that Sisyphus-1 (room allocation under constraints) was too small to be representative of real-world expert systems.

Sisyphus-2 was much larger (elevator configuration) and was aided by the "Jost document": an impressive and precise description of a prior system that performed this task. Resources like the Jost document are scarce, which limits problem selection.

In our view, the data collection for the Sisyphus-2 project was inadequate. Data was collected in a non-uniform manner and so comparative assessment studies are hard. Further, the development teams were focused on validating their own technique, rather than trying a range of techniques (including their own) to test if their technique was superior/ inferior to techniques from other paradigms. This deficiency was not recognised till after Sisyphus-2 was complete (1994). Hopefully, data collection will be more rigorous for future Sisyphus projects.

Despite these data collection problems, the KA community has benefited from the Sisyphus experiences. They now have a greater understanding of each other's techniques. This understanding has lead to an invaluable cross-fertilisation of ideas and a significant maturation of KA development approaches. Further, the KA community is now developing empirical evaluation methodologies.

Generalising this experience, we can see that (i) such reproducible experiments conducted by groups of researchers are a valuable exercise; (ii) developing a sufficing empirical experimental regime takes several years and several attempts (i.e. we should not give up if we fail first time); (iii) such efforts are greatly enhanced by scarce resources such as the Jost document.

5. Conclusion

i) Contrary to popular OO folklore, the supremacy of the OO paradigm is an open research issue. Empirical evidence for statements such as "OO code is re-usable", "OO programming is more productive that functional decomposition", and "OO generates more robust code" is lacking. Further, OO has yet to match results from other communities which use a variety of non-OO techniques (e.g. the SBF results).

ii) Contrary to popular OO SE folklore, repeatable SE experiments are possible (see the list in the section 3).

iii) Given (i), and the feasibility of (ii), the OO SE community should halt the generation of new methodologies while it actively and empirically explores the strengths and weaknesses of current approaches. Based on the Sisyphus experience, we predict that such an empirical analysis will take at least three to four years and will involve at least two false starts.

iv) Given (iii), no new OO methodology should be proposed till 1998.

6. References

1. Bell, D., I. Morrey, and J. Pugh, **Software Engineering: A Programming Approach**. 2 ed. 1992, Prentice Hall. 338.
2. Compton, P.J. and R. Jansen, *A philosophical basis for knowledge acquisition.* **Knowledge Acquisition**, 1990. **2**: p. 241-257.

[8] E.G. var1 = random(x)*14+10; var12 = random(x)/random(y).

[9] Sisyphus was doomed for eternity to roll the same rock up the same hill again and again and again...

3. Courtney, R.E. and D.A. Gustafson, *Shotgun Correlations in Software Measures* **Software Engineering Journal**, 1983. (January): p. 5-11.
4. Date, C.J., **An Introduction to Database Systems**. 5 ed. 1990, Addison-Wesley.
5. Durnota, B. and C. Mingins, *Tree-Based Coherence Metrics in Object-Oriented Design*, in **Tools 12 & 9**, C. Mingins, *et al.*, Editor. 1993, Prentice Hall. p. 489-504.
6. Fenton, N., S.L. Pfleeger, and R.L. Glass, *Science and Substance: A Challenge to Software Engineers* **IEEE Software**, 1994. (July): p. 86-95.
7. Haynes, P. and T.J. Menzies. *"C++ is Better than Smalltalk"?* in **Tools Pacific 1993**. 1993. Melbourne, Australia: Prentice Hall.
8. Haynes, P. and T.J. Menzies. *The Effects of Class Coupling on Class Size in Smalltalk Systems* in **Tools '94**. 1994. Melbourne:
9. Jacobson, I., M. Christerson, P. Jonsson, and G. Overgaard, **Object-Oriented Software Engineering: A UseCase Driven Approach**. 1992, Addison-Wesley. 520.
10. Lewis, J.A., S.M. Henry, D.G. Kafura, and R.S. Schulman. *An Empirical Study of the Object-Oriented Paradigm and Software Reuse* in **OOPSLA '91**. 1991.
11. Linster, M., *A review of Sisyphus 91 & 92: Models of Problem-Solving Knowledge*, in **Knowledge Acquisition for Knowledge-Based Systems**, N. Aussenac, *et al.*, Editor. 1992, Springer-Verlag: p. 159-182.
12. Linster, M., *Sisyphus '92: Models of problem solving*. 1992, Institut fur Angewandte Informationstechnil der Gesellchaft fur Mathematik und Dataverarbeitung:
13. Marques, D., G. Dallemagne, G. Kliner, J. McDermott, and D. Tung, *Easy Programming: Empowering People to Build Their Own Applications* **IEEE Expert**, 1992. (June): p. 16-29.
14. Menzies, T.J. and P. Compton. *Knowledge Acquisition for Performance Systems; or: When can "tests" replace "tasks"?* in **Proceedings of the 8th AAAI-Sponsored Banff Knowledge Acquisition for Knowledge-Based Systems Workshop**. 1994. Banff, Canada:
15. Menzies, T.J., J. Edwards, and K. Ng. *The Mysterious Case of the Missing Re-usable Class Libraries* in **Tools Pacific 1992**. 1992. Sydney, Australia: Prentice Hall.
16. Menzies, T.J. and R. Spurret. *How to edit "it"; or a Black-Box Constraint Based Framework for User Interaction with arbitrary Structures* in **Tools Pacific 12**. 1993. Melbourne: Prentice Hall.
17. Mintzberg, H., *The Manager's Job: Folklore and Fact* **Harvard Business Review**, 1975. (July-August): p. 29-61.
18. Patil, R.S., P. Szolovitis, and W.B. Schwartz. *Causal Understanding of Patient Illness in Medical Diagnosis* in **IJCAI '81**. 1981.
19. Pearl, J. and R.E. Korf, *Search Techniques* **Ann. Rev. Comput. Sci. 1987**, 1987. **2**: p. 451-67.
20. Preston, P., G. Edwards, and P. Compton. *A 1600 rule expert system without knowledge engineers.* in **Second World Congress on Expert Systems**. 1993. Lisbon: Pergamon.
21. Shaw, M.L.G. *Validation in a Knowledge Acquisition System with Multiple Experts* in **Proceedings of the International Conference on Fifth Generation Computer Systems**. 1988.
22. Stark, M., *Impacts of Object-Oriented Technologies: Seven Years of Software Engineering* **J. Systems Software**, 1993. **23**: p. 163-169.
23. Van de Brug, A., J. Bachant, and J. McDermott, *The Taming of R1* **IEEE Expert**, 1986. (Fall): p. 33-39.
24. Wirth, N., **Algorithms + Data Structures = Programs**. 1976, Prentice-Hall.

7. Appendix

This appendix describes projects we feel are suitable for reproducible university software engineering experiments (e.g. the comparison between the functional decomposition and OO paradigm).

Sisyphus-1 and Sisyphus-2 are clearly documented and existing published solutions exist in a international publications. Both are pure inference tasks (as apposed to e.g. the interactive interface tasks below). Both have an objective criteria of success. There exists only one legal allocation of staff to rooms in the Sisyphus-1 problem. There exist numerous solutions which satisfy the constraints of the Sisyphus-2 elevator configuration problem, but the best solution is the cheapest elevator that satisfies the constraints.

Another well-specified software engineering problem would be the construction of an SQL system that could handle all the SQL examples of chapters 4,5,6,7, and 8 of Date's database text [4]. Initially students could be asked to assume RAM storage. Students maintaining the code could be asked to make certain changes such as (i) disc-based storage using B*-trees (see section 4.5.1. of [24]); (ii) speeding up the code by at least a factor of 10; (iii) adding integrity rules (see chapter 11 of Date), or (iv) a QBE front-end to the SQL (see examples in section 14.6).

Other possibilities have less-well-defined goals, but far more succinct descriptions. We believe that inferring a complete specification from a partial description of the problem is a valuable training exercise for an apprentice software engineer. Certain visual problems have a very succinct partial description. For example, students could be asked to build a system that inputs Figure 2 and outputs an graph to an ascii file.

```
label Percent of Students Passing Comp321
range 1983 50 1993 100
bottom ticks 1983 1985 1987 1989 1991 1993
left ticks 50 60 70 80 90 100
1984 61
1985 72
1986 55
1987 65
1988 71
1989 73
1990 70
1991 79
1992 66
1993 73
```

Figure 2: *Input to a simple Grapher application.*

Grapher is a small introductory problem. *Grapher* could be made more intricate by telling students that the parser of the input must be separate to the graph generator which is also separate to the output routines. This could be motivated by saying that, in the future: (i) the system will later be used to write to some X-terminals which can address a two-dimensional space; and (ii) different import routines will be used (e.g. from a spreadsheet rather than text files like Figure 2).

Other, more intricate problems are *Breakout* and *Smart-Ask*. These systems can be succinctly specified with one screen and a few lines of explanation.

- *Breakout* is a simple video game specified by [1] that can be implemented on ascii terminals. Both an OO and a functional decomposition specification are supplied (in sections 6.2 and 9.1).
- *Smart-Ask* is an intelligent table for life insurance clerks that contains pre- and post-processing rules for controlling which table to use and what to do with the generated table. For more details, see section 2.1 of [15].

In terms of complexity, *Breakout*, *Grapher*, and Sisyphus-1, are much simpler that the other three. Sisyphus-2 is possibly the hardest since the Jost document takes a while to understand. The SQL task is probably longer than Sisyphus-2, but the conceptual problems are easier. Tacit in *Smart-Ask* are interactive authoring tools for building the tables/rules and a database for sharing tables/ student results.