Software Reuse with Frameworks

Greg Hartman
What makes a framework

It is difficult to find a precise definition of frameworks. However, most of the literature discusses the following concerns:

- Reuse of high-level program structure (architecture)
- Reuse of implementation
- Separation of application-specific code and generic code
- Guidance for incremental development
- Instantiation of framework elements to extend the interface between application-specific code and the framework

Different frameworks have these attributes to different degrees. In addition, other systems share many of these features.

From my reading of papers in the area, frameworks appear to come from the GUI framework implemented in SmallTalk-80 [39-42]. Many of the same papers state that MacApp, released in 1986, was the first commercially successful framework[13].

Judging from the surveys of reuse techniques summarized below, frameworks came to the attention of the reuse community at some point between 1984 and 1990. This is consistent with the statements above, which would place the widespread awareness of frameworks at 1986.

<table>
<thead>
<tr>
<th>Citation</th>
<th>Year</th>
<th>Framework</th>
<th>Patterns</th>
<th>Code libraries</th>
<th>Specification libraries</th>
<th>Code generation</th>
<th>VHL languages</th>
<th>COTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>[30]</td>
<td>1984</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>[31]</td>
<td>1984</td>
<td>y</td>
<td>y</td>
<td></td>
<td>y</td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>[32]</td>
<td>1984</td>
<td>y</td>
<td>y</td>
<td></td>
<td></td>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td>[33]</td>
<td>1990</td>
<td>y</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>[34]</td>
<td>1992</td>
<td>y</td>
<td></td>
<td>y</td>
<td></td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
<tr>
<td>[35]</td>
<td>1995</td>
<td>y</td>
<td>y</td>
<td>y</td>
<td></td>
<td>y</td>
<td>y</td>
<td>y</td>
</tr>
</tbody>
</table>

Differences from other reuse approaches

The same surveys mention several related reuse strategies. Johnson mentions that the designers of the MVC based framework at Xerox consciously blended two existing reuse techniques: architectural patterns and code libraries [39]. The architectural patterns reduce the amount of time that the programmers must spend in design and ensure that all programs developed with the framework will share a common structure. This similarity can help developers to navigate in unfamiliar code[50]. When combined with an underlying code library, the shared architecture reduces the amount of glue code that programmers have to write, freeing them to concentrate on the unique features of their application [5].

Unlike frameworks, code generators[23], specification compilers like YACC [1], and very high-level languages[31] require considerable up-front investment from
programmers who have highly specialized skills. In addition, these techniques create multiple copies of the same code. While this meets the goals of creating the initial application in less time, it does create maintenance problems if a bug is discovered in the code generator.

Companies that invest in the development of program families [48] create software that has a separation of application-specific and reused code that looks similar to frameworks. However, the development process for the family starts with a domain analysis and attempts to construct an architecture that minimizes the member-specific code that must be added to create a new family member. If companies choose to create a domain specific application framework with a top-down analysis, these approaches are equivalent. However, some frameworks, like CORBA and Swing, focus on reusing design and code that recurs between very different domains. These frameworks are typically developed apart from any description of the domain, increasing the chances that the resulting framework is widely applicable, but also running the risk of causing the duplication of domain-specific code.

Domain specific software architectures (DSSAs) appear to be similar to program families and therefore have the same relationship to frameworks. The development of DSSAs involves a systematic, top-down domain analysis[26].

Plug-in architectures and operating system device drivers tend to use small, fixed size interfaces between the plug-in and the larger application. Like frameworks, these interfaces provide guidance to programmers who are attempting the use the framework.

However, the application-specific code in frameworks typically modifies the interface with the framework at runtime. This typically happens with the code instantiates classes in the framework. For example, in a GUI framework a programmer may only be required to implement simple interfaces for the model, view, and controller to create a working application. However, the application-specific code can instantiate a new button and associate a listener with the button. This scenario effectively extends the interface between the application-specific code in a way that is not typical of classical plug-in architectures and operating systems. This approach has the advantage of supporting incremental development, but also makes conformance to rules about the interface, such as every button has a listener, much more difficult to check at compile time.

**Relationship to Redwine & Riddle technology maturation**

The close relationship between frameworks and the other reuse techniques may explain their somewhat unusual traversal of the technology maturation timeline outlined in [2]. In theory, new technologies go through several stages: basic research, concept formulation, development and extension, internal exploration, external exploration, and finally popularization. This problem is further complicated by the lack of detailed information about the development of the MVC pattern in the SmallTalk-80 system mentioned in [6].
Basic Research: From what I have been able to discover, the development of frameworks began when developers of the GUI of SmallTalk-76 noticed that a pervasive architectural pattern of models, views, and controllers (MVC) in their code, but paradoxically noticed a large amount of duplicated code around the pattern. They decided to attempt to create reuse libraries in SmallTalk-80 to eliminate the duplicated code. To this end, they turned to standard OO reuse techniques, creating a library of shared classes and interfaces. In doing this, they were actually developing the first framework by leveraging prior research in reuse techniques.

Since the focus of the SmallTalk research centered OO languages and GUI development, the concept formation phase was delayed for several years. In addition, the lack of detailed records of the development of the MVC pattern and GUI of SmallTalk-80 make it impossible to identify a separate development and extension phase for frameworks. Eventually, the technology moved to other research and development groups, including Apple computer.

The transition of frameworks to internal exploration is clearly marked by the development of a second framework called the Toolkit for the Lisa in the 1979-1983 timeframe[7]. While the Lisa was a generally available product, the high price of the hardware and the complexity of the framework meant that few outside developers created software for the system. Part of the problem was the complexity of the Toolkit. In hindsight, the developers of the toolkit mention several unrelated causes. First, there were few models of similar frameworks to follow; in hindsight some of the design decisions in the Toolkit increased its complexity. Second, applications inherited both implementation and interface from the Toolkit. This technique, called a white-box framework, forcing application developers to know the internals of the framework. Finally, there was no underlying API that developers could turn to if the framework did not support their requirements.

In 1984, Apple released the Macintosh. The Macintosh did not use a framework, partially in response to the problems encountered with the Toolkit. However, developers quickly discovered that they needed to duplicate large amounts of non application-specific code from Apple’s documentation to make their systems work. In 1986, Apple released a new black-box framework. In this black-box framework, application developers inherited interfaces and not implementation from the framework, allowing them to create a working application by implementing four interfaces. This framework, called MacApp, quickly became commercially successful. From the perspective of the adoption of GUI frameworks, the technology entered the popularization phase within a period of years.

In 1988 Johnson and Foote proposed generalizing the techniques used to create GUI frameworks to create a development method to support reuse of both the architecture and the implementation of applications[39]. This paper marks the concept formation of frameworks as a reuse strategy. The prior application of these ideas to implement popular GUI frameworks allowed the concept to skip directly to external exploration, as can be seen by the inclusion of the technique in reuse survey papers within a few years of its introduction and experience reports that begin to appear in the mid 90’s.[44].
Research directions

Attempts to extend frameworks to domains beyond GUIs have encountered limitations including: the difficulty of estimating the cost and benefit of developing new frameworks, problems in composing multiple frameworks, and questions about how frameworks should be structured. These problems are outlined in more detail below:

The cost of developing a new framework is quite high, difficult to estimate, and the benefits of using the framework are difficult to quantify.[36] Without good estimates of these factors it is difficult to justify the creation of new frameworks in a commercial setting. GUI frameworks did not encounter the same problems, partially because the frameworks would apply to large classes of applications and, in the case of MacApp, prior experience with the SmallTalk-80 and Lisa systems demonstrated that the framework would greatly reduce the effort of developing new applications.

The costs of frameworks are also inflated by the amount of time that developers must devote to learning about the framework. In the case of MacApp, the promotional documents[8] indicated that one to two months were required to master the framework. For other frameworks, estimates range as high as one year[14]. While developers can (and typically do) develop applications during this time, the applications may require considerable rework. This investment in learning makes frameworks less attractive in business settings.

The success of GUI frameworks means that developers using the new framework are likely to have to integrate two or more frameworks into their applications. When developers attempt to do this, they are likely to encounter several problems, including: inversion of control from multiple frameworks, difficulty integrating with legacy systems because the frameworks rely on sub-classing, filling in functionality not handled by any of the frameworks, conflicting architectural styles, and reconciling different classes provided by the frameworks to represent the same domain entities.[15] Solving these problems increases the cost of reusing the framework, possibly to the point of making reuse of the framework impractical.

Some frameworks attempt to avoid these problems by using framelets (small groups of classes) to implement the functionality of the framework. Unlike a monolithic framework, developers should be able to combine the framelets to avoid pulling in framework support that does not help their application.[18] The framelets are usually designed so that they can be combined without adding additional glue code to the application, eliminating one of the disadvantages of traditional code reuse libraries. However, developers still need to locate the appropriate framelets for their application. Prior experience with reuse libraries indicates that this can be daunting task.

Additional research about the use of frameworks is needed to assess techniques for resolving these problems. Since some studies have indicated that expert programmers respond to situations differently than novice programmers[50], validation with experienced programmers in industrial settings seems to be essential.
Validation techniques

While formal experiments often allow tests of causality in addition to correlation, it is difficult to design experiments to directly answer questions posed by frameworks. Therefore, a case study approach seems to be preferable. It may be possible to design formal experiments in light of observations from these case studies to gain greater insight (causal relationships, predictive models, etc) into outcomes observed in the case studies.

There do appear to be some case studies that involve software reuse and frameworks:

<table>
<thead>
<tr>
<th>Citation</th>
<th>Subject</th>
<th>Task</th>
<th>Observations</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>[52]</td>
<td>Students</td>
<td>8 C++ programs</td>
<td>level of reuse, defect density, LOC reworked, time</td>
<td>Case study</td>
</tr>
<tr>
<td>[57]</td>
<td>1 industrial programmer</td>
<td>Completion of 6 projects, some with framework</td>
<td>LOC/hour, learning speed, defect rate</td>
<td>Case study with repetition</td>
</tr>
<tr>
<td>[53]</td>
<td>Teams of industrial programmers</td>
<td>Develop a framework, then a second application</td>
<td>LOC/hour, defect rate, completion time</td>
<td>Experience report</td>
</tr>
</tbody>
</table>

Several other studies are notable because they provide insights into the process of gathering data in industrial settings:

Salo[60] describes a technique where pseudo-experiments can be run by observing developers in one development project to get a control, intervening in some way, and observing the developers during a second project. He makes the point that data being gathered should be checked and analyzed as soon as possible. This both encourages the developers to gather the data and fixes any discrepancies before they can threaten the study.

Kitchenham points to a number of defects in other empirical studies in the field.[56]

Atkins describes a technique to automatically gather data through standard development tools.[55]

Soloway describes a way to test a deep hypothesis about how programmers reason able code in a controlled setting.[50]
Annotated Bibliography

The bibliography is grouped by section and ordered by year then author. This helps with the reconstruction of the evolution of the technology.

The sections are:

- Miscellaneous
- Experience reports [3-12, 14-18]
- Other approaches [19-22, 24, 25, 27, 28]
- Smalltalk history [29]
- Surveys [30-38]
- Framework techniques [39, 41-49]
- Validation examples [50-61]


   Could also be validation. Describes an experiment where the authors collect causal information in a development environment (case study). The first step is identifying the questions of interest (avoids missing data), decide on the analysis. For example, looked at why the changes were made by having developers fill in a form. They then tested the forms. Finally, they collected data from three different development projects. Forms were validated as they were collected (up to 50% error rate on the forms). Programmers often listed more than one change per form. Interface errors were very low. It turns out that the developers were reusing design (reason for listing as other technique).


   This marks an early attempt in the universities to deal with frameworks. Actually, wanted meyrowitz_86


   Motivates the need for a new approach to Mac application development by showing how little application-specific code is present in many of the examples given in the Inside Macintosh. Proposes OO and frameworks (expandable applications). Cites simula and smalltalk as prior art in a box on page 12.

Compares Smalltalk and MacOS at a high level. Page 296 goes into detail about MVC, noting that it isn't well documented.

7. Schmucker, K.J.: Object-oriented programming for the Macintosh. (1986) 315-

Lisa framework suffered from complexity because there was no underlying toolkit that programmers could use for the weird cases.


2-3 weeks to learn object Pascal and then an additional one to two months to learn about MacApp. Based on 18 months experience shipping Macintosh applications. OO technology only recently available (Smalltalk 80, Objective C in 1983 and Clascal in 1984). Talks about adding multi-level undo to the framework.


Also seemed to popularize frameworks. Good cite list for MacApp. Argues for frameworks from the deficiencies of API's and skeleton programs: duplication of code that should be in the application, programmer must figure out the structure of the application, can't ensure UI consistency above widget level. Skeletons also make the application code more complex and less manageable.


Cited as saying that 50% of code is in the user interface of GUI applications.


Type of paper: Position Paper. What is the hypothesis: IS research needs new methods to move beyond applied research to technology transfer into industry. Existing techniques (theorem proof, mathematical modeling, subjective/argumentative, phenomenological/hermeneutic research, survey, case study, forecasting, and futures research) aren’t applicable to technology transfer. Instead, try game/role playing, simulation, action research, and (laboratory or field) experimentation. Validation: Persuasion and quotes from authorities. Do you believe the result, and why? I’m confused. How is a case study different from a field experiment? It’s also confusing from the point of view of software engineering: it seems like we’re still debating whether or not SE research is pure or applied. This seems to imply that it has to go even further than applied research.


Cost per line increases with the number of lines of code. GUI’s increased the complexity of software, increasing the program size (see myers_rosson_92).


Cited for a 6-12 month learning curve for GUI frameworks.

Discusses problems that developers encounter with frameworks and some potential solutions. Problems include: inversion of control from multiple frameworks (coordinating callbacks), integration with legacy systems (due to reliance on sub-classing), the framework gap (functionality not handled by any of the frameworks), architectural mismatches (conflicting architectural styles), overlapping components (different classes for the same domain entities), and integrating functionality from multiple frameworks (passing messages between the frameworks). Discusses strategies like wrapping frameworks and using multiple threads to resolve these conflicts.


Argues that managers of small projects won't invest the time to scale existing practices.


Something about problems scaling complex practices to small projects.


TODO: Points to some interesting articles on the problems with frameworks.


Proxy for a book by same author similar title. Tracked down because it talks about reusing design.


Cited as early source for reusing design


Cited as considered reusability over the years. Mostly seems to be 4GL advertisement. Talks about the need to use iteration to get around specification mistakes.


Outlines different uses for sub-classing, then talks about traits. Traits sound a lot like aspects, but appear to be abstract classes. It also covers what happens when multiple inheritance creates duplicates of traits (the diamond problem). They seem to assume that the trait will be duplicated (no virtual inheritance). The paper then discusses algorithms to determine the memory layout. In the end, most of the system could have been implemented with single inheritance, but there were some exceptions like the table classes.


Big contribution relative to frameworks: start with entire applications. Gives example of lint reusing the first passes of the C compiler. Must have a high-level language to pull this off. Runs into the problem of spurious changes.


Shows use of a technique called logic structures. Programs categorized into one of seven categories. Dictates the structure of the resulting program. Achieves 60% reuse.


Mostly discusses patterns and how to methodically provide components fit for composition.


Unpublished work that seems to look a lot at design patterns, and argues for components that implement the patterns.


Frameworks evolved from SmallTalk


Lists several general and specific goals: creating reusable building blocks (make libraries of code, design, and specification), make general architectures, make patterns, make code generators, make very high level languages, and generate programs from specifications.


Lists cost drivers: now making more complex systems, more demand for programmers, and software methodologies don't improve as fast as hardware cost drops. Proposes using shrink-wrap software (possibly with modification). Thinks that new computer systems may help (UNIX, etc). Improved languages like Ada and Pascal. Rapid prototyping. Reuse of code. Reuse of designs. Application generators. Existing reuse mechanisms: subroutine libraries, compilers (think yacc), incorporation of elements into programming languages, parameterized systems (choose from a list of options, then compile a working system). Spectrum is now widening: reusable code (need mechanisms to identify components, specify components, cataloging components, very high level program producing systems, transformation of a specification into a program, application generators, and prototyping. Packages in Ada help, as do generic procedures. Shrink-wrap applications like Visicalc also seem to help, since a secondary industry of templates has grown up.


Argues for reuse of data, architecture, code, designs (by field), entire programs and subsystems. Notes the rise of spreadsheet templates.

Discusses efforts to standardize OO terms and interaction patterns. Details several different research methodologies (responsibility driven design, object oriented software engineering, role modeling, frameworks).


TODO: Constructs a hierarchy of problems for reuse.


Portrays reuse as a way to cope with software failure: industry wants higher quality than what we can now produce. Discusses classification schemes (what is being reused vs. how reuse is happening). Talks about source code, schemas (patterns), transformational systems, application generators, VHLLs. Discusses effect on the development process, techniques for measuring reuse. Also discusses problems in reuse, especially in OO systems. Discusses component retrieval from a library.


Thinking about economics. Software reuse is still not routine. Unusual from the perspective of engineering, where reuse is routine. Differs from software design, since design tries to produce finished systems, reuse attempts to make building blocks. Must weigh costs when considering reuse. Need good theories to support practice. Theoretical research is necessary, but has scaling problems. Research should proceed with simple models.


The paper is 21 pages of quotes (about three paragraphs each) from various books discussing software reuse. There are some photographs from a building materials recycler interspersed. My guess is that there are fewer than three pages of original text. The selection of passages seems to have been thought through, but the paper doesn't present a unified story.


This paper summarizes empirical studies of class libraries, generics (or templates), and inheritance and argues that these techniques don't eliminate redundancies. Aspects also get some treatment. The paper is plugging a particular solution (XVCL).


Provides one of the more accepted definitions of frameworks and compares them to reuse libraries. Makes a distinction between frameworks where programmers provide additional components that conform to well understood protocols (black-box) and frameworks where the programmer sub-classes the framework to do the implementation (white-box). Inversion of control is often, but not always a characteristic of frameworks. Also discusses ways to develop a framework.


Reference for presence of MVC pattern in SmallTalk-76 and the desire to facilitate code reuse in SmallTalk-80

42. Linton, M.A., Vlissides, J.M., Calder, P.R.: Composing user interfaces with InterViews. Computer 22 (1989) 8-22

Cited as a paper that popularized frameworks


Mentioned as one of the first papers to discuss problems of evaluating IT investments.


Claimed to mark a shift to small frameworks


TODO: Seems to survey why programmers make use of inheritance.


An update on the state of the art in frameworks. Discusses the relationship between frameworks and components and notes that domain-specific frameworks tend to be proprietary while technology frameworks are widely available. Also makes the distinction between design and implementation reuse. Design reuse is more desirable, since it can be widely applied. Unlike patterns, frameworks carry implementation. Frameworks make heavy use of patterns internally.


Creates a framework for aspects.


Enhanced method: An enhanced method is described for the design of reusable software based on analyzing the common features of the family with step-wise refinement and information hiding. Correlation to real-world setting Reuse along the lines of the classical model was happening. The examples chosen were plausible programs. Hypothesis: stepwise refinement is always profitable. Module specifications can be profitable if family members will vary widely. Proof: feasibility demonstrated by examples. Profitability argued from the definition of specifications: they are so clear that the programmer doesn't need to look at most of the code. Do I believe it? Creating and maintaining specifications like this is extremely expensive, so stepwise refinement can be unprofitable.

Unvalidated enhanced method (a.k.a. whitepaper) An enhanced method is proposed for reducing the cost of extending product lines based on a combination of development tools, standard components, frameworks, and models. Correlation to real-world setting Reuse is still a problem, but it has been solved in other industries. This is easier to see if we focus on economies of scope (mass design), not scale (mass production). Hypothesis: the upfront costs of this method can be contained with improved abstraction techniques and CASE tools. More specialized factories will be more valuable. Proof: persuasion only Do I believe it? I'm least convinced by the argument about costs. Most of the other points here look convincing.


Hypothesizes that expert programmers have shared plans on what should happen in programs. Writes programs that don't conform to the hypothesized plans, then compares experts and non-expert comprehension. Expects to see a big difference in expert performance, less for non-expert. Also checks if experts can fill in a single missing line in programs. Both held in studies with short programs with students. Type of paper: Empirical Study Correlation to the real-world setting: There isn't much of one, although the hypothesized rules and test programs seem vaguely realistic. What is the hypothesis: Experienced programmers differ from novices. First, they follow stereotypical plans when constructing software. Second, they follow rules of programming discourse. Validation: Construct programs that follow and don't follow the plans and discourse rules. Show these programs with one line missing to experts and novices. Ask the programmers to fill in the gap. See if the gap between the experts and the novices is greater for the programs that follow the rules. Also show full programs to experts and novices and ask them to recall the programs verbatim. Result: All of the hypotheses were verified. One of the programs (sqrt) seems to cause some minor problems. Do you believe the result, and why? I do believe the results. The principle objection, that these results may not scale to larger programs, doesn't seem plausible. It also lines up with my subjective industry experience.


Type of paper: Empirical survey Correlation to the real-world setting: Papers were chosen from well known journals in computer science and software engineering. What is the hypothesis: Computer science and software engineering papers leave out experimental validation in cases where it is used in natural sciences. Validation: The authors classified a random sample of papers from CS and journals in the natural sciences. They classified the papers to identify which ones should have experimental validation. Finally, they looked for experimental validation in the papers. Result: 40% of CS publications that should have experimental validation don't. In SE, the ratio is closer to 50% of CS publications that should have experimental validation don't. Other fields that are younger than CS seem to have a better validation rate. Do you believe the result, and why? I do. I also think that there are a lot of experimental studies with serious validity problems.


Type of paper: Experience Report (Case Study). Hypothesis: Validation: Studied 8 systems written in C++ by students. Quantified the level of reuse, defect density, amount of rework, and time spent on the program. The reuse should be negatively correlated to the other variables. Result: Design of the study precludes concluding causation. Statistically significant correlations exist for defect density, rework not normalized to faults, and productivity. Reuse does not affect the amount of rework to fix an identified fault. Do I believe it? My only reservation is in the area of external validity: students may not be good substitutes for experienced programmers.

Type of paper: This paper is an experience report. Correlation to the real-world setting: The paper reaches its conclusions by studying 10 real-world projects that developed application frameworks and then proceeded to develop a second application on the framework. What is the hypothesis: Frameworks will not increase the amount of code that programmers produce in a given unit of time, but they will decrease the amount of work that needs to be done to deliver the system. Validation: Use two different techniques to compare the productivity of the programmers as they implement the framework and the initial system to their productivity implementing a second system on the framework. One of the measures only looks at the size of the delivered system. The other measure subtracts the reused code from the size of the system. Result: The first metric indicates a productivity improvement of approximately 30%. When the second measure is used (the one that excludes reused code), the productivity increase drops to 10%. Do you believe the result, and why? There are several problems here. The two metrics are radically different, and the author admits that one of them is not well suited to OO systems. It would be helpful to have more samples, but I realize that this would be difficult. I would like to see more investigation of why there still seems to be a 10% productivity improvement on the second system.


Type of paper: The presentation is as an experience report, but there is also an enhanced model. Correlation to the real-world setting: Looking at 15 years of telephone switch development. What is the hypothesis: code decay exists. Validation: Define decay carefully as “change more difficult than it used to be.” Construct a series of reasonable diagnostic tests to find code decay. Run them over the 15 years of development and see if the tests predict changes the time spent working on change requests. Result: It's there. Small changes tend to be harder. Software may increase in value from the changes that cause decay. Code can decay even if it doesn't change (creeps in from other regions). Explains reasons, symptoms, and risk factors for code decay. Code decay seems to exist. The number of developers touching the code doesn't correlate to the rate of decay. Do you believe the result, and why? There are some big inferences that the authors point out. One that I'm worried about is that programmers may just forget. By their definition, this is code decay, but it seems odd. I've also found that 1 LOC changes cause problems.


Type of paper: Empirical Study. Correlation to the real-world setting: The authors are observing a large, real-world software project over several years. The only problem that I see is that the approach to managing multiple versions of the software with specialized preprocessor directives is a bit unusual; most projects would have used branching. What is the hypothesis: An editor that automatically handles the preprocessor directives will reduce the effort needed to make these changes and also lead to shorter development intervals. Validation: Modify the editor to place invisible tags on the directives. Gather monthly status reports where developers mention which bugs they were working on. Gather data from the modification request database to tie the source changes to specific bugs. Use regression analysis to calculate effort for individual bugs. Then use these effort numbers and evidence from the source control system to fit an effort equation to predict the effort to fix a bug. The authors paired similar developers (in terms of experience and workload), one who doesn’t use the tool and another one who does and see if the effort predictions vary with the use of the tool. Result: Developers are 40% more productive with the tool and releases seem to come out faster. Do you believe the result, and why? Not really: Most of the developers studied only used the tool ½ the time. They tended to use it more toward the end of the study, so some of this is learning curve. However, I’m concerned that there may be other confounds. I don’t have great confidence in the effort calculations, since they’re coming from fairly sparse data (the monthly reports). The study failed to gather data that may have helped: developer interviews about why they did or didn’t use the tool, the frequency of corrupted
#version directives, the curve of productivity for new developers, the number of questions that
senior developers answered (the need for this wouldn’t have been obvious before the study), and
possible the time spent in the editor. To be fair, the authors acknowledge most of these limitations;
I’m listing them for future reference.

K.E., Rosenberg, J.: Preliminary guidelines for empirical research in software engineering. IEEE

Type of paper: Enhanced Method. Hypothesis/correlation to the real-world setting: The standard
of empirical validation of papers in software engineering is poor. This paper provides guidelines
which will improve the quality of experiments in the field. Validation: The authors cautiously cite
examples of some well-known papers in the field. The methods are supported by appeals to
clinical research in medicine. Summary of the guidelines: C1: 2.1 Provide information about the
industrial context. 2.2.1 Identify potential confounds 2.2.2 Avoid over-simplification in formal
experiments C2: State a hypothesis, preferably one that is deep (not just technology A is better,
but technology A is better because…) This allows people to understand where your results may
apply. The authors mention that there are contradictory studies in SE. C3: If the research isn’t
exploratory, define a single question in advance. Otherwise, you may just keep trying hypotheses
until you hit on one that happens to be statistically significant. C4: Cite and discuss the related
work. D1: Identify the population. D2. Define the candidate selection process P4: Present raw
data. The authors advance an argument of why students may be a legitimate population to study.
Do you believe the result, and why? I think that these guidelines are valuable for formal
experiments. However, I’m not convinced that we should make formal experiments and deep
hypotheses the gold standard for software engineering validation. Although the authors don’t state
this explicitly, I get the impression that this is their agenda from the space given to formal
experiments and the extensive list of citations from clinical research.

57. Morisio, M., Romano, D., Stamelos, I.: Quality, productivity, and learning in framework-based
876-888

Frameworks need management commitment, a stable domain, and highly skilled developers. We
lack knowledge to support managers in decisions about frameworks: when will they pay off? Type
of paper: Experience Report (Case Study) Hypothesis: The use of frameworks improves
productivity and quality. Net productivity = writes code faster (LOC/hour, not counting reuse), not
just completes application faster. The developer will learn faster with the framework. Learning
reduces the defect rate faster with frameworks. Validation: Hire an industrial programmer and
have him complete six projects, some using a framework. Monitor the developer’s productivity
and look for learning effects. Result: Net productivity is higher, learning is faster in terms of
productivity. Learning’s influence on quality wasn’t significant in frameworks, but was without
them. Framework code tends to be higher quality, but the gap narrows with learning. Do I believe
it? Yes, but I’m a bit surprised that net productivity was detectable. It would be helpful to have a
deeper hypothesis of why this is the case.

58. Dingsøyr, T., Røyrvik, E.: An empirical study of an informal knowledge repository in a medium-
sized software consulting company. In: ICSE ’03: Proceedings of the 25th International

Type of paper: experience report with multiple data sources. Data sources employed included
Interviews, Observation (logbooks, screenshots and pictures). Conflicting views of company
structure Observation of a flat structure, interviews indicated a hierarchy Conclusion (somewhat
tangential for our purposes) Use of tool seemed to depend on job function No consensus on use of
hierarchy to structure data in the tool

Adopting XP at two companies cut the pre-release and post-release defect rate (1.0->0.6) while improving programmer productivity (1.0->1.7) KLoC and in one case improving morale. More studies are under way.


Assumes an iterative development environment. Inject ideas gleaned from last development cycle. Look for changes in the metrics. Paper type: improved method (for studying development). The authors correctly point out that the results still don't generalize. The paper contains many practical suggestions for running these studies.


TODO: fix cite when actually published.