Applications of Simulation and AI Search: Assessing the Relative Merits of Agile vs Traditional Software Development

Bryan Lemon, Aaron Riesbeck, Tim Menzies, Justin Price, Joseph D'Alessandro,
Rikard Carlsson, Tomi Prifiti, Fayola Peters, Hiuhua Lu, Dan Port*
Lane Department of Computer Science and Electrical Engineering, West Virginia University
*Information Technology Management, University of Hawaii
bryan@bryanlemon.com,tim@menzies.us,{ariesbeck|tprifti|hlu3}@mix.wvu.edu
{justin.n.price|jdalessa57|ricca742|fayolapeters}@gmail.com,dport@hawaii.edu

Abstract

We implemented Boehm-Turner's model of agile and plan-based software development. That tool is augmented with an AI search engine to find the key factors that predict for the success of agile or traditional plan-based software developments. According to our simulations and AI search engine: (1) in no case did agile methods perform worse than plan-based approaches; (2) in some cases, agile performed best. Hence, we recommend that the default development practice for organizations be an agile method.

The simplicity of this style of analysis begs the question: why is so much time wasted on evidence-less debates on software process when a simple combination of simulation plus automatic search can mature the dialogue much faster?

1. Introduction

There are too many examples in software engineering of positions being defended without empirical evidence. For example, in the 1990s, the third author published extensively on the supposed benefits of rule-based [1] and object-oriented programming [2] without hard evidence to back up those claims. Many other prominent writers were just as unconstrained. In the mid-1990s Booch [3], Rumbaugh [4] and Jacobson [5] engaged in an high-profile and extended debate about the merits of different styles of object-oriented modeling. Nowhere in that debate was any empirical evidence that one kind of model was easier to write/ read/ debug/ maintain than any other kind. That discussion is now closed- but not because any participant won the debate. Rather, a commercial company hired Booch and Rumbaugh and Jacobson to create one unified modeling language (UML [6]).

One explanation for this propensity to argue without evidence is the "data drought" problem reported by metrics-guru Norman Fenton. After years of advocating careful data collection [7], he now despairs of that approach. At a keynote address in 2007[1] Fenton shocked his audience by saying: *"....much of the current software metrics research is inherently irrelevant to the industrial mix ... any software metrics program that depends on some extensive metrics collection is doomed to failure"*. The COCOMO [8] experience supports Fenton's pessimism. After 26 years of trying, Boehm et al. have only collected less than 200 sample projects for the COCOMO database [9]. There are many reasons for this data drought including lack of data being collected or the business sensitivity associated with the data, as well as differences in how the metrics are defined, collected and archived. Also, recently, we have seen a decreasing willingness for organizations to share their operational data [10].

When data-based evidence is missing, model-based evidence can fill in the gaps. Model-based methods can be used to build an executable form of the intuitions and mental models of domain experts. If part of a model is not known with certainty, model variables can become ranges, rather than point values.

Sometimes, software process models are difficult to build. For example Raffo [11] spent two years carefully building and tuning a software process model for a North-Western software construction company. Also, some software process models are hard to understand- especially when the uncertainties associated with the model lead to numerous, very wide, ranges. These uncertainties can lead to voluminous output if the model is used within some Monte Carlo rig. Analysts may find the output of such models confusing, rather

1. http://promisedata.org/?cat=130

than clarifying.

Not all process models are problematic. With the right automated support, *software process modeling can be very easy*. Given a well-defined theory and some automatic AI search tools, it is possible to collect model-based evidence about the value of variants on software processes. For example, our case study concerns *requirements prioritization policies*. Such policies control the order in which a team implements the requirements. The spectrum of these policies can be characterized by two extremes:

- In traditional *plan-based* prioritization, the ordering is performed once, prior to starting the work.
- In *agile* projects, orderings may change often.

In highly dynamic environments, the standard argument is that agile is a better method for reacting to the arrival of new requirements and/or existing requirements changing value. In the literature, the agile versus plan-based debate is often cast as a "one or the other" proposition. This is a false dichotomy that ignores development practices that use interesting combinations of of plan-based and agile development. A more nuanced view is offered by Boehm and Turner [12]. They define five scales that can characterize the difference between plan-based and agile methods: project size, project criticality, requirements dynamism, personnel, and organizational culture.

Previously, at ASE'08 [13], Port & Olkov & Menzies (hereafter, POM) studied the effects of different prioritization policies while adjusting the rate at which new requirements arrive and/or change value. Grünbacher [14] criticized the POM model (version 1), noting that it explores only one of the five scales proposed by Boehm and Turner.

An advantage of model-based evidential reasoning is the ability to repeat a prior analysis. Once a model limitation is apparent, the model can be changed and re-simulated. For example, in the following study, the POM model is extended along the lines proposed by Grünbacher then explored using AI search engines.

Using the model and the AI search engine, we study plan-based and agile policies in different contexts. We found (1) no case where agile does worse than plan-based; (2) in some cases, agile performs much better. This leads to the following conclusion:

> *The default development methodology for an organization should be agile.*

Also, we show that (sometimes) model-based reasoning about software processes is easy:

- The revised model was implemented in eight weeks by a graduate AI class working $\approx 20\%$ of their time on this task. This team had no experience in processing modeling and little experience with the implementation language (LISP).
- The team ran the simulations on a CPU farm of 20 Linux machines. The results reported in this paper could be generated in one overnight run.

The simplicity of our analysis begs the question: why is so much time wasted on evidence-less debates when a simple combination of simulation plus automatic search can mature the dialogue much faster?

2. Simulator Specifications

2.1. POM2: Summary

POM2 shuffles $requirements$ through four stages:

1) They are *initially* in the "to do" heap;
2) *Next* the requirements are revealed to a team;
3) *Subsequently* a team moves the requirement to their implementation plan;
4) *Finally* the implemented requirement may move to a list of completed requirements.

Step 4 is optional: if a project suffers from early termination, some requirements may never reach completion. Within POM2:

- $Requirements$ have $costs$ and $values$.
- $Projects$ have trees of $requirements$;
- $Teams$ implement different sub-trees but may be reliant on $requirements$ from other sub-trees;
- $Teams$ $sizes$ (# of members) may differ;
- $Teams$ work in $iterations$ and each $iteration$ implements a small number of $requirements$.
- A $team$'s $plan$ is the set of $requirements$ to be implemented in the next $iteration$.
- The size of each $plan$ is determined by budgetary considerations.
- When the $plan$ is done, $teams$ fill up the $plan$ again by moving $requirements$ from their requirements sub-trees.
- The $project$ ends if the $requirements$ sub-trees is empty or management orders $early$ $termiantion$.
- $Teams$ order the $requirements$ in their $plans$ using a $requirements$ $prioritization$ $policy$.

POM2's inputs are shown at the top of Figure 1. These inputs include project $size$ (number of personnel in all teams), project $criticality$, requirements $dynamism$, $personnel$ types, and organizational $culture$. These scales effect the processing within POM2:

- In the case of high $criticality$ systems, $requirements$ $cost$ more to develop.
- In the case of high requirements $dynamism$, the $requiirements$' $cost$ and $values$ change frequently.

- Different organizational $cultures$ react differently to those changes. Some $cultures$ may frequently re-prioritize $requirements$ while other $cultures$ are more prone to ignore value changes, least that disturbs the development $plan$.

The following description of POM2 is somewhat lengthy. Much of that discussion is a justification of different decisions. The actual code of POM2, shown in Figure 1 is quite short.

2.2. Differences to POM1

POM2 differs from POM1 as follows:

- POM1 assumed one team implemented the requirements while, in POM2, there are multiple teams.
- POM1's requirements had no dependents. POM2's requirements have dependents and a parent requirement cannot start till all its child requirements are finished.
- The POM1 paper assumed small projects with about 10 developers per project and a maximum of 25 requirements. Within our POM2 simulator, we work with projects involving up to 300 developers, we therefore expanded the maximum number of requirements from 25 to 750 ($\frac{300}{10} * 25 = 750$)

2.3. Requirements, Trees, Heaps, and Projects

Within POM2, a project is divided into teams, each of which implements a set of requirements. Each requirement has a value and cost (assigned randomly based on the value and cost scales described by Port et al. [13]). These values may change over the lifetime of a requirement, using the mechanics discussed below.

These requirements are stored in acyclic trees representing the work breakdown structure. The tree demands that all child-requirements must be completed before a parent can be started. A requirement becomes "ready" when all of its children are "completed." Initially, all leaf requirements are "ready," since by definition, they have no child that can block their start.

Initially, all teams are assigned different trees (this tree becomes the team's heap, from which new requirements are pulled). A standard business situation is that sometimes teams must wait for other teams to finish some requirements. To model this situation, the following mechanism is used to add inter-team dependency between requirements. This mechanism is controlled by the "inter-dependency" variable within our simulator. The inter-team dependencies are generated as follows:

INPUTS:
criticality:
.82 <= criticality <= 1.26 (defined in $2.6.1)
criticality modifier:
2.0 <= criticality modifier <= 10.0 (defined in §2.6.1)
culture:
0 <= culture <= 100 (defined in §2.6.3)
initial known:
.4 <= initial known <= .7 (defined in §2.3)
inter-dependency:
0 <= inter-dependency <= 100 (defined in §2.3)
size:
3 <= size <= 300 (defined in §2.6.4)
team size:
1.0 <= team size <= 44.0 (defined in §2.6.4)
dynamism:
1.0 <= dynamism <= 50.0 (defined in §2.6.2)

OUTPUTS:
performance score:
0 <= score <= 1 (defined in §2.7)

```
while X < numberOfTrials do
  tasks := generateTasks(size)
  teams := generateTeams(tasks, teamSize)
  teams.applyDependancy(interDependancy)
  teams.applyCriticality(criticality, modifier)
  stoppingAt := iterationsToComplete(2 to 6)
  while iteration < stoppingAt do
    for all team in teams do
      team.budget+=(TotalCost/6)
      AvailableTasks := null
      for all task in team do
        if noDependencies&&no Children then
          if notCompleted then
            AvailableTasks.append(task)
          end if
        end if
      end for
      AvailableTasks.applySortingPolicy(type)
      for all task in AvailableTasks do
        if budget − task.cost) < 0 then
          break
        else
          budget := budget − task.cost
          task.completed := true
        end if
      end for
      if budget > 0 then
        if AvailableTasks is empty then
          budget := 0
        end if
      end if
      team.discoverNewTasks(λ)
      for all task ∈ team do
        change := (N(0, σ) ∗ culture)
        task.value += (maxTaskValue ∗ change)
      end for
    end for
  end while
end while
```

Figure 1: POM2 Pseudo Code.

- For all requirements at the same level in a tree, one dependency is added to and from another tree, at the same level. Approaching generation in this manner removes the possibility of cyclic dependencies (these cannot be completed).
- At the leaves of the tree, there are many requirements, so the addition of one to/from dependency has little impact on the completion rate of requirements. Higher in the tree, where there are fewer requirements, even one inter-team dependency may significantly slow down the requirement completion process.

This method best mimics a common structure seen in industry: teams often share large sub-systems (requirements that use many sub-requirements) rather than very small requirement assemblies.

Trees are generated as follows. Nodes are assigned children according to an exponential distribution: (50, 24, 12, 6, 3, 1.5)% of nodes have (0,1,2,3,4,5) nodes (respectively). Note that we have no theoretical justification for this distribution. Our literature searches did not find papers describing the average structure of a software project and the number of dependents for each requirements. We return to this issue below.

As a software project progresses it is not unusual for a development team to discover that they have new requirements to complete. We model this as follows:

- If the size of the project (number of personnel) is set to 200, the number of requirements for the project will be set to 500 (these ratios come from the literature review of the original POM paper).
- From these 500 example requirements, we determine, before the simulation begins, the number of new requirements that will be added at each project iteration. If we determine that we will add 6 requirements after the first iteration we will randomly generate these additional requirements before the simulation starts. These requirements are initially labeled as "hidden" and are marked "visible" when the simulator encounters the iteration for which they were added.

2.4. Iterations

Teams maintain a *plan*, which is a set of requirements that have been pulled from their heap. Initially some percentage of requirements are pulled from the heap and placed in the plan, these represent the known set of requirements at the beginning of the software project. This percentage is controlled by the "initial known" variable within our simulator. *Plans* are processed in iterations. Within our model, the total maximum number of iterations is 6, but it is entirely possible for a project to end early. Such early terminations are very common in iterative projects if (for example) management decides that the resources of an organization should be transferred from an existing project to a new one.

Early termination is modeled by assigning a probability of .9 that a project may continue after iteration i. This means that at 6 iterations, a project has a 53% probability of finishing. This early termination rule models the industry reality that many projects are canceled before they're originally planned end date.

The number of requirements that can be completed in each iteration is controlled by the budget. Following [13], the budget is defined as:

$$budget = \frac{(total\ initial\ cost)}{number\ of\ iterations} \quad (1)$$

Where "total initial cost" is computed from the cost of the team's plan, as it is known at the start of the project, after criticality is applied (discussed below).

2.5. Prioritization Policies

POM2 explores three prioritization policies.

1) Plan Based (PB) : The Plan-based policy is different from the other two methods. This policy models a non-agile development method where the order of requirements are sorted once at the start of development using the $\frac{value}{cost}$ numbers assumed at the start of the project. While the other policies sort at every iteration, the plan-based policy only adjusts requirement values at every iteration, "but does not resort them." [13]
2) Agile (AG) : Agile policies sort requirements (every iteration) only using value. According to Cao et al. [15] this is the standard method advocated by the agile community.
3) Agile2 (AG2) : The same as AG, but sorts on value/cost. This is another standard method used throughout the agile community.

POM1 explored a fourth *hybrid* policy that combined features of agile with plan-based methods. That policy assumed a linear structure of requirements and there are technical difficulties with implementing that structure over the trees of POM2. Hence, we leave exploration of hybrid methods for future work.

2.6. Handling Boehm and Turner

POM2 implements four of the five scales identified by Boehm and Turner [16] that distinguished agile

	1	2	3	4	5
Level	very low	low	nominal	high	very high
Impact	none	impact on discretionary funds	impact on essential funds	single life	many lives
Cost	0.82	0.92	1.00	1.10	1.26

Figure 2: Effort multipliers on cost due to criticality. Learned via regression from 161 projects. From COCOMO [8].

from traditional plan-based projects. These four scales are:

- project $criticality$;
- requirements $dynamism$;
- organizational $culture$;
- $size$ (total number of developers in all teams);

POM2 does not implement the fifth scale ($personnel$) due to theoretical reasons, see below.

2.6.1. Criticality. Boehm and Turner [16] comment that agile methods are untested on safety-critical products as they present potential difficulties with simple design and lack of documentation. Conversely, plan-based methods evolved to handle highly critical products which are hard to tailor down efficiently to low-criticality products.

With this in mind, Boehm and Turner measure criticality in terms of losses due to impact or defects and ranges from "none" (best for agile development) to "impact on discretionary funds" to "impact on essential funds" to "loss of single life" to "loss of many lives". Thus, plan-based methods are best suited to projects that must be carefully planned, lest defects cause loss of many lives.

According to the COCOMO research [17] the effect criticality has on cost is displayed in Figure 2. POM2 assumes that all requirements performed by one team are of equal criticality (this reflects the industrial standard that specialist teams work on particular portions of the code base). We assume that X% of the teams are affected by the criticality, where $X = 2 \ldots 10$. Within our simulator, we refer to X as the criticality modifier. We adjust the cost of each requirement in a selected teams tree as follows:

$$cost' = cost * X^{criticality} \tag{2}$$

2.6.2. Dynamism. Project dynamism measures how frequently new requirements are created and how often existing requirements change value. As dynamism *increases*, we discover *more* new requirements and the value of the requirements becomes $more$ variable.

Boehm and Turner measure dynamism in terms of the percent of requirements changed each month and has the range 50% (best for agile) to 1 (best for planned-based).

To implement this factor, the POM1 assumptions for dynamism [13] are adopted:

- Initially, we only mark

  $$30\% \leq 40 * N(0,1) \leq 70$$

  of the requirements in the project tree as "visible." At each iteration, for each team, we make visible

  $$new = Poisson(\lambda)$$

  more hidden requirements in the tree.
- Another parameter controlling dynamism is σ. At each iteration, every uncompleated requirement (in the plan or in the project tree) is visited, and and its value is alterd by:

  $$value' \text{ += } maxRequirementValue * N(0, \sigma) \tag{3}$$
- Note that POM2 links σ and λ are linked such that high σ values implies high λ values (and vice versa). Specifically, after setting σ, we use

  $$\lambda = \frac{\sigma}{10}$$

There are some special cases of the above process that requires further discussion. If the dynamism parameter tells a team to find a "new" number of requirements, but there are not that many "visible" or "ready" requirements, then a team's plan may cost less to complete than the budget for that iteration. In the case where there is some remaining budget, and there are no requirements that can be completed, the budget is burned. We call this "twiddling thumbs" where a team of employees ceases to work while they wait on the completion of work from another team.

In another case, a team may encounter a situation where their remaining budget is not large enough to complete the next available requirement. If this happens, the budget is kept and carried over to the next iteration. This is accomplished by maintaining two numbers: *totalAccumulatedBudget* and *totalSpentBudget*. Budget is burned by setting the *totalSpentBudget* to the *totalAccumulatedBudget*. At the start of each iteration, *totalAccumulatedBudget* is incremented by the budget for the iteration.

Before continuing, we digress to discuss another method (which we do not use) for handling leftover budgets. We considered allowing a team to work on requirements from other teams. However, Brook's Law [18] (adding programmers to a late requirement makes it later) convinced us of the folly of that approach. Teams are specialists in the quirks and capabilities of their own code base. An outsider coming in to work

temporarily on a small part of another team's code base can be quite unproductive (since they do not know the quirks of that code). Further, they can slow up the remaining team (while they teach the newcomer the tricks of that code).

2.6.3. Culture. For a fully agile project, changing requirement values means resorting the requirements in the plan to ensure that most cost-effective and valuable requirement is completed first. However, as discussed in this section, the corporate culture may inhibit that resorting process.

According to Boehm and Turner, culture is measured in terms of the percent of the staff thriving on chaos and has the range 90% (best for agile),70,50,30, 10 (best for plan-based). At $culture = 90\%$, the changes to a requirements value described above (see $Dynamism$) are used when resorting for the next iteration. However, at $culture = 10\%$, developers are loathed to change the initial project plan since this introduces a degree of disorder into their work life.

We therefore distinguish between the true value and the accepted of requirement, calculated as follows:

$$accepted = value + (value * N(0, \sigma) * culture) \quad (4)$$

As the percent thriving on chaos decreases, "culture" drops to 0 and the accepted value remains as the old value. Note that the accepted value is used to resort (exception: not for the plan-based policy, which never resorts) the requirements, but when performance statistics are gathered, we use the true value.

2.6.4. Size. $Size$ is measured in terms of the number of personnel and has the range 3 (typically, best for agile), 10, 30, 100, 300 (typically, best for plan-based). The size of a POM2 project is picked at random from this range and the number of requirements is then set to $size * 2.5$.

We planned to build teams using the results of [19] who report that the size of software development teams has $(min, mean, sd) = (1, 8, 20)$. However, this leads to a large number of single-person teams. We modified that result slightly in consultation with some of our NASA colleagues. POM2 repeatedly selects team size randomly from the distribution of Figure 3 until the total team size exceeds $size$.

2.6.5. Personnel. Having described the scales implemented in POM2, we now discuss the theoretical problems that lead to an unimplemented $personnel$ scale. Boehm and Turner describe project $personnel$ using the Cockburn mixtures model. In summary, the mixtures model divides programmers into three groups:

.

X = # of personnel	Y = frequency of X
1	3
3	9
6	18
9	28
12	22
15	18
18	15
21	12
24	8
27	3
30	2
33	1
36	1
39	1
42	1

Figure 3: Distribution of team sizes.

Alpha: The most productive, most flexible programmers. Alpha programmer are able to revise a method, breaking its rules to fit an unprecedented new situation.

Beta: With training, Beta programmers are able to perform discretionary method steps such as sizing stories to fit increments, composing patterns, compound refactoring, or complex COTS integration.

Gamma: May have technical skills, but gamma programmers are unable or unwilling to collaborate or follow shared methods

Applying the conventions of the Boehm and Turner *personnel scale*, lower personnel values indicate *more* alpha programmers on the team.

We planned to use the COCOMO effort multipliers [8] to derive a cost delta for requirements implementation (the *more* alphas on the team, the *less* the development effort). Strangely, however, when we applied those effort multipliers, the net effect was nearly zero. To see "zero effect", consider the COCOMO multipliers relating to programmer and analyst capability:

capability	very low	low	nominal	high	very high
acap= analyst	1.42	1.19	1.00	0.85	0.71
pcap= programmer	1.34	1.15	1.00	0.88	0.76

After averaging the rows, combining the lows and the highs, and expressing the results as ratios of the very high values, we arrive at the following "slowdown factors" representing the effects of using different kinds of programmers: $\{1.6, 1.22, 1\}$ for using {gamma, beta, alpha} programmers, respectively. With these factors, we had planned to shrink the budget for an iteration according to how many less-than-good programmers were assigned to the iteration.

$$budget = budget/(1.6 * \gamma + 1.22 * \beta + \alpha). \quad (5)$$

where γ and β and α are the ratios of different kinds of assigned programmers.

Boehm and Turner define personnel to range 1 to 5 where each step defines ratios of alpha, beta, and gamma programmers. For example, as shown in Figure 4, if $personnel = 5$ then 65% of the team comprises alpha programmers. The last row of that figure demonstrates "zero effect". This row shows the weighed sum of the ratio of programmer types as calculated by Equation 5. Note that the weighted sums are nearly all the same. That is, the net effect of these personnel types on productivity is almost constant. For that reason, POM2 ignores the productivity dimension and we will explore this issue in future work.

		Boehm & Turner's $size$ scale				
	programmer types	1	2	3	4	5
α	% alpha	45%	50%	55%	60%	65%
β	% beta	40%	30%	20%	10%	0%
γ	% gamma	15%	20%	25%	30%	35
SUM:	using Equation 5	118	119	120	121	121

Figure 4: Boehm & Turner's $personnel$ scale selects for different ratios of alpha, beta, and gamma programmers of a project.

2.7. Performance Score

As POM2 requirements are completed, they are marked "completed." At the end of an iteration, all completed requirements are moved to a "done" set, and requirements left in the plan are passed to the next iteration. As each new requirement is added to the "done" set, statistics are kept on the team's performance:

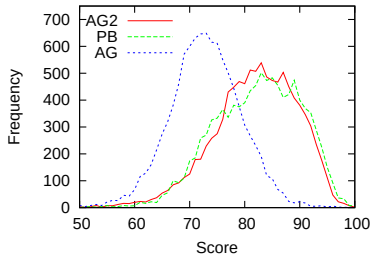
1) Sum of costs (of completed requirements)
2) Sum of values (of completed requirements)

By tracking these two statistics on a (X,Y) plane, for $X = 1 \ldots N$, we can visualize the time-varying performance of the team: When the project is finished, the plot can be compared to an *optimal frontier*, obtained by sorting all the "done" requirements using the final $\frac{value}{cost}$ for all those requirements. Note that the sort used to generate the optimal frontier relies on information not available halfway through the simulation (specifically, the $\frac{final\ cost}{value\ of\ each\ requirement}$). The Optimal Frontier works on the same set of requirements, except that those requirements $\frac{cost}{value}$ pairs are updated to their new value before the simulation starts. No method can do better than the optimal frontier.

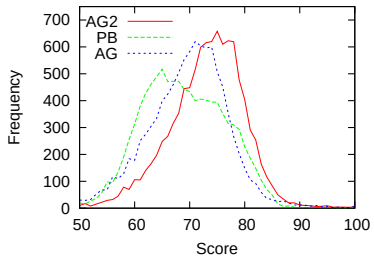
Scoring works as follows. If a project finishes after completing, say 100 requirements, then we would score the run as follows:

1 Take the final $\frac{cumulative\ value}{cumulative\ cost}$ generated by the project at requirement 100.

Very low dynamisms: $\sigma = 0, \lambda = 0$



Medium dynamism: $\sigma = 0.15, \lambda = 0.015$



High dynamism: $\sigma = 2, \lambda = 0.2$

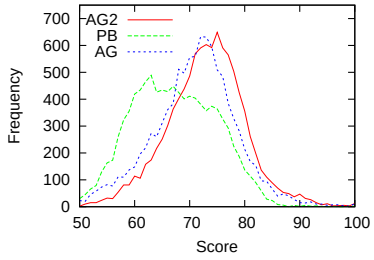


Figure 5: Distribution of Performance scores, controlling σ and λ, while choosing at random for other model inputs.

2 Divide that by the $\frac{cumulative\ value}{cumulative\ cost}$ figure from the optimal frontier at 100 requirements.

The performance scores generated in this way are shown in Figure 5. Note that, in those results, the agile projects that sort using just $value$ always perform worse than the agile2 projects that use $value/cost$ (the blue plots always score lower than the red plots). This is an interesting result since, according to Cao and Ramesh [15], sorting on just $value$ is a common agile practice. In a personal communication, Alistair Cockburn has stressed the need adjust priorities using value and cost.

Two other interesting features of Figure 5, are:

- Agile methods do not perform better than plan-based when dynamism is low (see top plot).
- The median performance of plan-based *decreases* as dynamism *increases* (the green curve moves further to the right).

That is, while there is no advantage of use agile for low dynamism situations, agile methods can adapt to increasing dynamism (while plan-based cannot). This leads to the following conclusion: organizations should adopt agile2 processes as their default methodology.

3. Searching

The Figure 5 results are interesting, but they are a two-dimensional summary of a ten-dimensional space:

- one output value;
- one choice of prioritization policy;
- eight input values.

It is possible that that the general finding that agile2 works as well, or better than plan-based does not hold for regions within that 10-D space. Before adopting the above conclusion, we need to find any special regions in the 10-D of POM2.

We define these regions as follows: Given a model with $inputs$ to a model that are a set of $feature_i = range_i$, find the smallest subsets that most change the model $output$. Each such subset is special region since it is here where the score changes most. Our preferred tool for this analysis is KEYS [20]. KEYS is a combination of a greedy search and a Bayesian ranking method called BORE (short for "best" or "rest"). The greedy search starts to explore the domain. Assume that the model has N accessible states which are the inputs to the model (in our case, our sates are assignments to the Boehm and Turner scales). These inputs are randomized from a distribution $\{X_i, i = 1 \dots N\}$. A collection of inputs to the model is called a treatment.

$$\{x_1, x_2, \dots, x_N\} = treatment \tag{6}$$

After M number of initial samples, a second phase starts and one of the inputs is fixed to a desired range $X_i = M_j$. This is performed for each of the input variables (in our case, the Boehm scales) until the stopping criteria is met. Keys stops if the improvement from the previous round is less than 5%. The range is selected using the BORE heuristic (described below). At the next run the $treatment = \{fixed\} \bigcup \{randomizede\}$. KEYS' pseudo-code is shown in Figure 6.

At its core, KEYS uses the BORE heuristic to rank ranges. BORE takes N inputs that have been scores with outputs. For this study, we used the performance scores generated by the method of §2.7. It assumes that any numeric values have been converted into a set of bins. For our work, we divide the Boehm Turner scales into ten equal width ranges.

while $Fixed\_Inputs \neq Total\_Inputs$ **do**
  for I = 1 to N **do**
    $Selected[1...(I-1)]$:= best decisions so far
    $Guessed$:= random settings to the remaining mitigations
    $Input := Selected \cup Guessed$
    $Scores := Score(Input)$
  end for
  $Top\_Range := Bore(Scores)$
  $Fixed\_Inputs$ += 1
  $Selected(Fixed\_Inputs) = Top\_Range$
end while
return $Selected$

Figure 6: KEYS pseudo-code

These N inputs are then split into two groups - the 10% best (with highest value) and the 90% rest [21]. The probability that a input range is found in $best$ is then determined by using Bayes theorem. The theorem uses evidence E and a prior probability P(H) for hypothesis $H \in \{best, rest\}$, to calculate a posteriori probability $P(H|E) = P(E|H)P(H)/P(E)$. When applying the theorem, likelihoods are computed from observed frequencies (hereafter, $freq$). These likelihoods (hereafter, $like$) are then normalized to create probabilities. This normalization cancels out $P(E)$ in Bayes theorem. For example, after N = 10, 000 runs are divided into 1,000 best solutions and 9,000 rest, the value for X_i is in a certain range might appear 10 times in the best solutions, but only 5 times in the rest. Therefore:

$$\begin{aligned}
P(best) &= 1000/10000 = 0.1 \\
P(rest) &= 9000/10000 = 0.9 \\
freq(E|best) &= 10/1000 = 0.01 \\
freq(E|rest) &= 5/9000 = 0.00056 \\
like(best|E) &= freq(E|best)P(best) = 0.001 \\
like(rest|E) &= freq(E|rest)P(rest) = 0.000504 \\
P(best|E) &= \frac{like(best|E)}{like(best|E)+like(rest|E)} = 0.66
\end{aligned}$$

With this setup the model is sensitive to small frequencies. For example, the computed value of 0.66 hides the fact that the range was only seen 15 times in 10,000 samples. To penalize such samples, we introduce a *support* term into the calculations. This support term should be large if a range is frequent; i.e. the $like$ value suffices as a support measure. Hence, when KEYS looks for ranges that most effect output, it uses:

$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \tag{7}$$

In other work (under review, submitted to ASE 2009) we show that BORE+KEYS is surprisingly powerful at finding these regions. Empirically, we have shown that KEYS can out-perform an interesting range of alternate search engines such as simulated annealing, A-STAR, and MaxWalkSat.

		Note: ranges run $bin_i <= value < bin_{i+1}$									
		bin 1	bin 2	bin 3	bin 4	bin 5	bin 6	bin 7	bin 8	bin 9	bin 10
	criticality	.82	.86	.91	.95	1.0	1.04	1.08	1.13	1.17	1.22
	criticality modifier	2.0	2.8	3.6	4.4	5.2	6.0	6.8	7.6	8.4	9.2
	culture	0	10	20	30	40	50	60	70	80	90
	initial known	.40	.43	.46	.49	.52	.55	.58	.61	.64	.67
	inter-dependency	0	10	20	30	40	50	60	70	80	90
	size (total personnel)	3.0	32.7	62.4	92.1	121.8	151.5	181.2	210.9	240.6	270.3
	team size (people per team)	1.0	5.1	9.2	13.3	17.4	21.5	25.6	29.7	33.8	37.9
policy / dynamism											
plan-based / very low	criticality										
$\sigma = \lambda = 0$	criticality modifier										
	culture										
	initial known									2	4
	inter-dependency							1	1	1	
	size	89	2								
	team size										1
plan-based / medium	criticality								5	21	52
$\sigma = 0.15, \lambda = 0.015$	criticality modifier										
	culture										
	initial known										
	inter-dependency						1	2	6	6	8
	size										
	team size										1
plan-based / very-high	criticality							1	6	22	58
$\sigma = 2, \lambda = 0.2$	criticality modifier										
	culture										
	initial known										
	interdependency						2	3	6	10	11
	size										
	team size										1
agile 2 /very low	criticality										
$\sigma = \lambda = 0$	criticality modifier										
	culture										
	initial known							1	4	12	27
	interdependency						1	4	6	5	3
	size	72	1								
	team size									1	4
agile 2/ medium	criticality	1						1	1	1	1
$\sigma = 0.15, \lambda = 0.015$	criticality modifier							1	1	1	1
	culture						3	10	19	22	29
	initial known					1		2	2	2	2
	interdependency				1	1	1	1		1	
	size	97									
	team size		17	20	11	6	1				
agile 2 / very high	criticality	1				1		1	1	1	1
$\sigma = 2, \lambda = 0.2$	criticality modifier	1	1	1	1	1	1	1		1	1
	culture					1	4	13	17	20	22
	initial known						1	1	1	1	1
	interdependency		1	1	1	1	1	1	1		1
	size	100									
	team size		15	18	11	5	2	1			

Figure 7: Results. The top table shows the division of each numeric inputs divided into ten bins. The rest of this figure show the percent frequencies with which a range appears in the treatments found by KEYS. Results taken from 1000 repeats for each pair of policy/dynamism. Darker colors denote higher frequencies.

4. Results

Figure 7 shows the ranges found in KEYS' treatments, while randomly selecting prioritization policies and the eight inputs on POM2. The results are split between two prioritization policies (AG2 and plan-based) and three levels of requirements dynamism (low, medium, high). Each combination of policy*dynamism was run 1000 times (resulting in 6*1,000=6,000 runs).

Notice that, in all cases, the treatments found by KEYS increased the performance scores reported in §2.7 by a statistically significant amount (Mann-Whitney, 95%).

The columns in Figure 7 refer to discrete ranges of the input space, called $bins$. Each $bins$ covers one-tenth of the maximum range on an input. For example, the *bin1* column refers to results for (say) $.82 \leq criticality < .86$.

Each cell in Figure 7 shows the percent frequency of that bin occurring in the treatments found by KEYS in the 1000 run. Empty cells were never found in treatments. Darker cells show more frequent values.

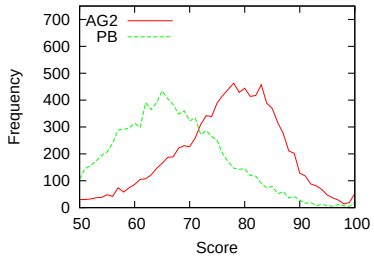
In the subsequent discussion, we ignore most effects that do not occur in the majority (over 50%) of the runs. There are several majority-case effects in Figure 7 that deserve our attention. Firstly, there are some strong negative results that hold across all dynamisms and prioritizations. One input ($criticality\ modifier$) was never selected by KEYS. Two inputs ($initial\ known$ and $inter-dependancy$) were never selected in the majority case ($> 50\%$ of the runs).

Secondly, with respect to agile2, there were frequent effects found for total $size$ of all personnel, the $team\ size$ of individual projects, and organizational $culture$:

- In 72 to 100% of our agile2 runs, KEYS selected for the smallest possible total team size variables ($size = bin1$; i.e. about 3 to 20 people).
- If we add up the percentages for $culture$ in $bin7, bin8, bin9, bin10$ then for (medium, high dynamism), cultural factors appeared in (80,72)% respectively of the treatments found by KEYS. This effect is predicted by theory: recalling Equation 4, the higher the $culture$ value, the more likely it is that teams will react to changing requirements $values$.
- Another somewhat smaller effect for agile2 can be found for the size of individual teams. KEYS found that for medium and very high dynamism, it is useful to have individual team sizes in $bin2, bin3, bin4, bin5$ (i.e. about 5 to 17 people).

Curiously, smallest size teams ($size = bin1$; i.e. below 5 people) was never selected by KEYS. We conclude that agile fails for very small teams working with a large number of other small teams when programmers spend so much time interfacing with other teams that they cannot get anything done themselves.

Medium dynamism: $\sigma = 0.15, \lambda = 0.015$



High dynamism: $\sigma = 2, \lambda = 0.2$

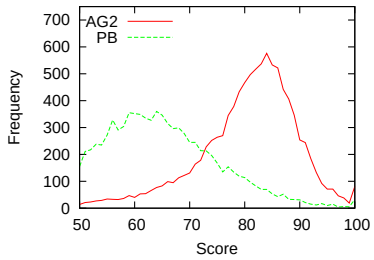


Figure 8: 1000 runs of KEYS controlling σ and λ while using $\frac{1}{10}$-th size ($3 \leq size < 32.17$) and top $\frac{1}{10} - th$ criticality (≥ 1.22) and choosing at random for the other inputs.

Thirdly, Figure 7 shows that many results are very similar; e.g. all the agile2 results offer nearly the same pattern. In fact, KEYS and Figure 7 show us that POM2 contains two major divisions of its 10-dimensional space:

- In division one, we find all all the agile2 results, as well as the low dynamism results for plan-base prioritization share the same distrubution. This division is characterized by best treatments mentioning very small total team size and high values on the culture scale.
- In division two, we find the medium and high dynamism in plan-based prioritization results deviate from the above distribution. This division is characterized by best treatments mentioning very high criticality ranges.

In terms of testing our general conclusion (about agile being as good or better than plan-based), the place to run tests is in the union of the two divisions: at very high criticality and very small overall team size. The results of those runs are shown in Figure 8. Clearly, in the union of those two regions, agile2 produces much larger median scores than plan-based. This figure lends support to the general conclusion of this paper. If management is given a choice between agile2 and

plan-based methods, they should adopt agile2.

5. Conclusion

Building software process models can be complicated by a drought of data associated with software projects. When data is scarce, model-based evidence can be used to make a reasoned case for reconfiguration of a project. If some uncertainty exists about a value in the model, that value can be represented as a range of possibilities.

Executing models containing ranges of possibilities can pose comprehension problems. If used carelessly, analysts can become confused by an information overdose. Consequently, it is useful to augment a simulation engine with a search engine that can find the most interesting regions within the model input/output space.

This paper was a case study with combination of simulator (POM2) and search engine (KEYS). The original version of POM dealt only with dynamism. This new POM2 version includes concepts such as organizational culture, project criticality and development by multiple large teams. Our AI search engine explored the state space of our model to find regions that significantly improved the output performance score of the model (value of completed tasks, as a function of how soon they were delivered). We found no case where plan-based out-perform agile2. In fact, sometimes agile2 performed much better than plan-based (Figure 8). We therefore recommend that:

> *The default development process for an organization be agile2.*

Notice that while Figure 8 explores *one* interesting region within POM2, it does not explore *all* possible combinations of subsets of ranges of that model. Such an analysis is beyond the scope of this short paper. However, if management ever finds themselves working in a particular part of the POM2 space, our models could be used to repeat an analysis like Figure 8 (i.e. to assess different policies for those particular projects in that particular space).

Note also that in two areas, POM2 found limits in software engineering theory. In order to mature the conclusions of this paper, two issues need to be addressed:

- What is the effect of mixtures of different types of programmers on a project? In Figure 4, we showed one calculation that concluded that mixtures had a zero effect on a project. This is an unexpected result and one that needs further investigation.
- What are the patterns of dependencies between requirements? Lacking any guidance from the literature, POM2 built its network of dependencies between tasks using the methods of §2.3. More work is required to uncover the expected patterns of dependencies in real-world projects.

Finally, we do not know why so much time is wasted on evidence-less debates in SE. Very simple simulations/search engines can automatically explore the nuances of a debate. Using such automated tools, it is possible to make interesting conclusions with minimal machinery and effort. Our model definitions took just days of work; the implementation required a few weeks of work; and the runs to gather our data took just one night (we used a lab of 20 Linux boxes for our simulations). Consequently, our conclusion is that evidence-less debates:

- should be strongly deprecated;
- could be replaced, if possible, with debates that use model-based evidence.

We used model based reasoning by applying the KEYS search engine to the POM2 model. We concluded that agile2 performs as good or much better than plan-based development.

References

[1] T. Menzies, J. Black, J. Fleming, and M. Dean, "An expert system for raising pigs," in *The first Conference on Practical Applications of Prolog*, 1992, available from http://menzies.us/pdf/ukapril92.pdf.

[2] P. Haynes and T. Menzies, "C++ is Better than Smalltalk?" in *Tools Pacific 1993*, 1993, pp. 75–82.

[3] G. Booch, *Object-Oriented Design with Applications (second edition)*. Benjamin/ Cummings, 1994.

[4] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorenson, *Object-Oriented Modeling and Design*. Prentice-Hall, 1991.

[5] I. Jacobson, M. Christerson, P. Jonsson, and G. Overgaard, *Object-Oriented Software Engineering: A Use Case Driven Approach*. Addison-Wesley, 1992.

[6] G. Booch, I. Jacobsen, and J. Rumbaugh, *Version 1.0 of the Unified Modeling Language*, Rational, 1997, http://www.rational.com/ot/uml/1.0/index.html.

[7] N. E. Fenton and S. Pfleeger, *Software Metrics: A Rigorous & Practical Approach (second edition)*. International Thompson Press, 1995.

[8] B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts, *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[9] T. Menzies, O. Elrawas, J. Hihn, M. F. anm B. Boehm, and R. Madachy, "The business case for automated software engineerng," in *ASE '07: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*. New York, NY, USA: ACM, 2007, pp. 303–312, available from http://menzies.us/pdf/07casease-v0.pdf.

[10] T. Menzies, S. Williams, O. Elrawas, D. Baker, B. Boehm, J. Hihn, K. Lum, and R. Madachy, "Accurate estimates without local data?" *Software Process Improvement and Practice (to appear)*, 2009.

[11] D. Raffo, "Personal communication," 2005.

[12] B. Boehm and R. Turner, "Using risk to balance agile and plan-driven methods," *Computer*, vol. 36, no. 6, pp. 57–66, June 2003.

[13] D. Port, A. Olkov, and T. Menzies, "Using simulation to investigate requirements prioritization strategies," in *Automated Software Engineering, 2008. ASE 2008. 23rd IEEE/ACM International Conference on*, Sept. 2008, pp. 268–277.

[14] P. Grunbacker, "Personal communication," 2008.

[15] L. Cao and B. Ramesh, "Requirements engineering practices: An empirical study," *IEEE Software*, vol. 25, no. 1, pp. 60–67, 2008.

[16] B. T. Boehm, "Balancing agility and discipline: Evaluating and integrating agile and plan-driven methods," in *proceedings 26th International Conference on Software Engineering (ICSE)*, 2004, pp. 718–719.

[17] S. Chulani, B. Boehm, and B. Steece, "Bayesian analysis of empirical software engineering cost models," *Software Engineering, IEEE Transactions on*, vol. 25, no. 4, pp. 573–583, Jul/Aug 1999.

[18] F. Brooks, *The Mythical Man-Month: Essays on Software Engineering*, 2nd ed. Reading, MA: Addison-Wesley, 1995.

[19] P. C. Pendharkar and J. A. Rodger, "An empirical study of the impact of team size on software development effort," *Inf. Technol. and Management*, vol. 8, no. 4, pp. 253–262, 2007.

[20] T. Menzies, O. Jalali, and M. Feather, "Optimizing requirements decisions with keys," *Proceedings PROMISE 08 ICSE*, 2008.

[21] R. Clark, "Faster treatment learning," Master's thesis, Portland State University, 2005.