Modeling Agile Development: When is it Effective?

Karim Chichakly
isee systems, inc.
31 Old Etna Road, Suite 9N
Lebanon, NH 03766
(603) 448-4990/FAX: (603) 448-4992
kchichakly@iseesys.com

Abstract

It is very difficult to deliver high-quality software (i.e., with very few bugs) in a reasonable time period. Indeed, it is not unusual on medium to large projects to spend as much time fixing bugs as delivering new features. One of the most challenging issues in software development is keeping pace with changing customer requirements.

Agile development was born from the idea that software development needs to be quick on its feet, responding to changing customer requirements without compromising delivery schedules or quality. It was founded on the principle of embracing change rather than fighting it. Some of the fundamental principles of Agile development include frequent customer interaction, frequent releases, writing tests before code, nightly builds with automated testing, and not implementing more than you know the customer needs.

Yet there is a surging debate about whether Agile works and when it works. This paper investigates when Agile development methods may work and the relative advantages of different parts of the methodology.

Keywords: Agile, Rework Cycle, Software Development, Project Management

Overview

This paper explores the implications of the following facets of Agile development:

- Short development cycles, or more specifically, frequent releases – these cycles are generated by the model
- Using a schedule/quality tradeoff that leads to both undiscovered and discovered rework moving from one release to the next (in particular, what are the implications of *starting* a project with a large amount of undiscovered and discovered rework?)
- Frequent customer interactions
- Frequent reviews between developers
- "Test first" (rather than code first) – *before* any code is written
- Frequent integrations, nightly builds with automated regression tests
- Availability to customers of frequent stable builds (vs. established and defined betas)
- Having a less predictable set of releases in the field (does not impact this model)
- Being adaptive to regularly changing requirements (hence the name "agile"), including adapting processes as required
- Main metric is working software vs. tasks completed
- Close face-to-face cooperation and communication
- Continuous attention to technical excellence and good design
- Always choose simple (spec or design) over complicated – it may change!

Software development is notorious for delivering too little too late, often with many bugs. It is very difficult to deliver high-quality software (i.e., with very few bugs) in a reasonable time period. Indeed, it is not unusual on medium to large projects to spend as much time fixing bugs as delivering new features. The reasons for this are many, but the rework cycle we have been studying gives a lot of insight into the problem.

One of the most challenging issues in software development has been changing customer requirements. Since software is fungible, most everyone has some idea of how to change it. Also, it is a documented phenomenon that people cannot adequately visualize how a piece of software is going to work by walking through examples. As soon as the real thing is there, they instantly find several things that need to be changed. Prototyping, RAD, and spiral development have all been things that have been tried to close this gap. Finally, the 90s saw a lot of companies shifting to a more customer-centric approach given heightened competition and greater expectations.

Agile was born from the idea that software development needs to be quick on its feet, while at the same time delivering quality software (in the land of software, change is usually *bad* because it means the introduction of lots of new bugs). Indeed, Agile was founded on the principle of embracing change (it is a fact of life) rather than fighting it.

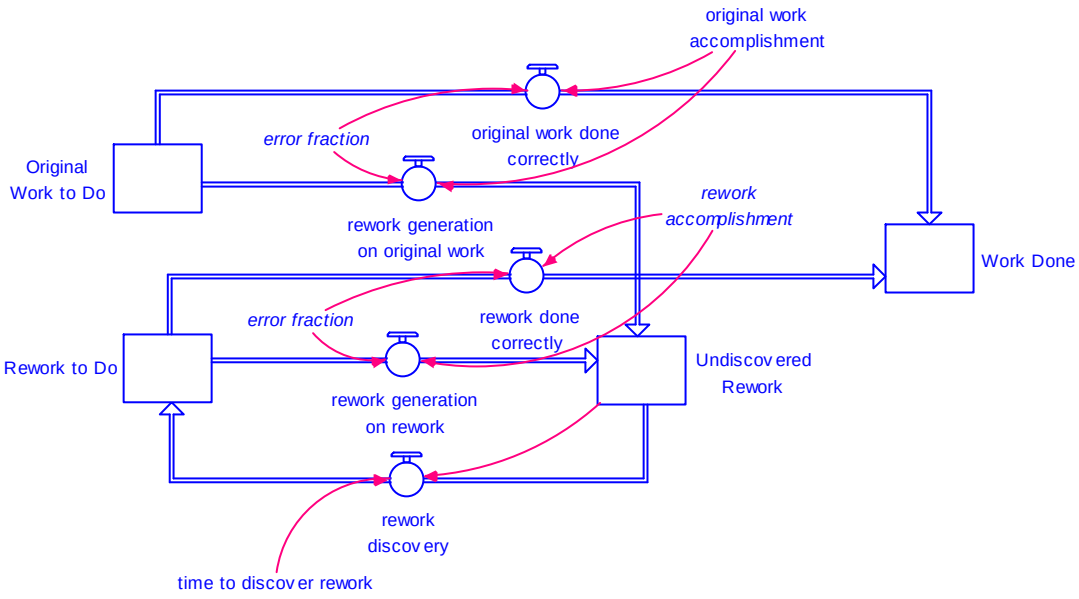
Yet there is a surging debate about whether Agile works (and when it does). A lot of people feel it is just a bunch of hooey that someone made up to make some money. Others think it only works for specific types of people (and that there aren't that many of them around). Then there

are those who are religious zealots about it. Finally, there are people like me who think there are some good ideas there, but the entire thing as a package is a bit much for most people to use (and rather unproven).

This paper investigates when Agile methods may work and the relative advantages of different parts of the methodology.

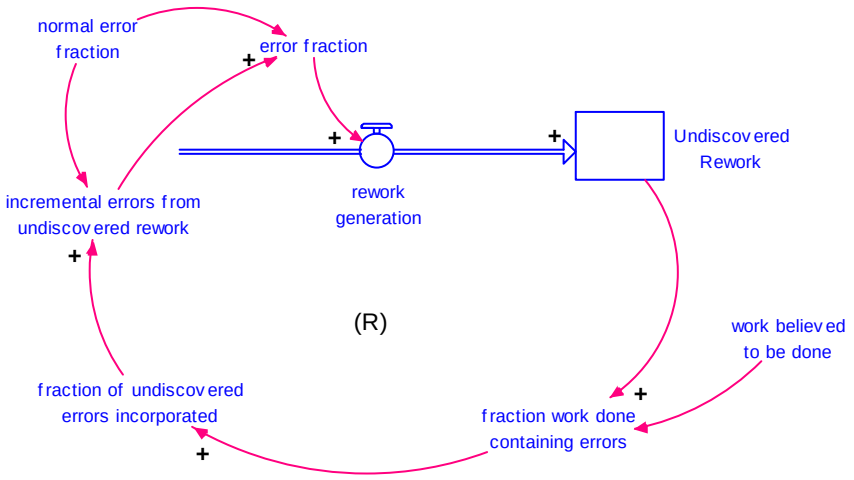
The Rework Cycle

The dynamics of a project revolves around the rework cycle. The structure below was given to the author by Jim Lyneis:



Each project begins with a stock of original work to do. As the work is completed, a fraction of the work is done incorrectly (determined by *error fraction*). This rework remains undiscovered for a time. As it is discovered, it is fixed, again generating a fraction of errors. Eventually, all of the work and rework is completed and the project is finished. Note that rework is not just restricted to specific bugs that have been introduced in the code, but also includes problems in the customer requirements and the project design.

Both the *error fraction* and the *time to discover rework* play critical roles. If the *error fraction* is high, much of the work will need to be redone. If the *time to discover rework* is very long, the rework may not be discovered until late in the project cycle, or even worse, after the product has shipped. Even worse, the *error fraction* necessarily increases when there are errors in the system, because the code that new code is being tested against is not producing the correct results. This is called the "errors on errors feedback" and it is a reinforcing loop (shown below). It is therefore advantageous to decrease the *time to discover rework* as much as possible.



In the waterfall paradigm, the entire project is placed in *Original Work to Do* and the project progresses until most of the required work is completed. In Agile, however, the project is broken down into separate phases (four in these simulations). When each phase is completed, based on a measure of quality of the completed work, *Original Work to Do* is injected with the work for the next phase.

Agile Background

The most fundamental precept of Agile is that the code base never stray very far from a completely working system.[1] This is guaranteed with a number of basic principles.

The first is an automated build and test system that runs every single night. This verifies the code correctly builds on all supported platforms and also regression tests it to make sure nothing was broken. Errors in the build or the testing are reported via e-mail or RSS feed so everyone knows right away that something is amiss. Also, when it is successful, an quasi-official build is available for manual testing (for this is the same system that builds official releases; this is a critical point: release builds are just nightly builds that meet a certain criteria, so testing is always being performed on whatever eventually gets shipped).

The second is the concept of "Test first". This means programmers are responsible for designing and writing automated tests *before* they write the actual code that needs to be tested. This serves many purposes, the most transparent one being it guarantees a body of regression tests (so-called "unit tests"). More importantly, it forces the programmer to think about the number of ways his code might break before he writes the code, thus ensuring he writes more robust code. Finally, as he writes the code, he now has something to test it against (as he goes) rather than just assuming the thing works.

[1] It is possible to have experimental branches of the code that don't really work, but they should never stray too far (or too long) from the trunk (main branch).

All regression tests must stay current. When a new bug is found that none of the regression tests uncovered, a new test is added for that specific problem. This ensures the bug will not reappear in later versions by incorporating what has been learned into the nightly automated tests.

Lastly, customer releases must be frequent, perhaps as often as once a month, to get quick feedback and to ensure effort is not being spent in an unproductive direction (e.g., an over specified feature). These short cycles make it very different from other approaches. A consequence of never straying far from a working system is also that most people applying these techniques offer intermediate alpha versions to customers at the end of every single week, shortening the feedback delay even more. Customers are also usually closely involved in deciding the features that need to be implemented (they get a vote anyway).

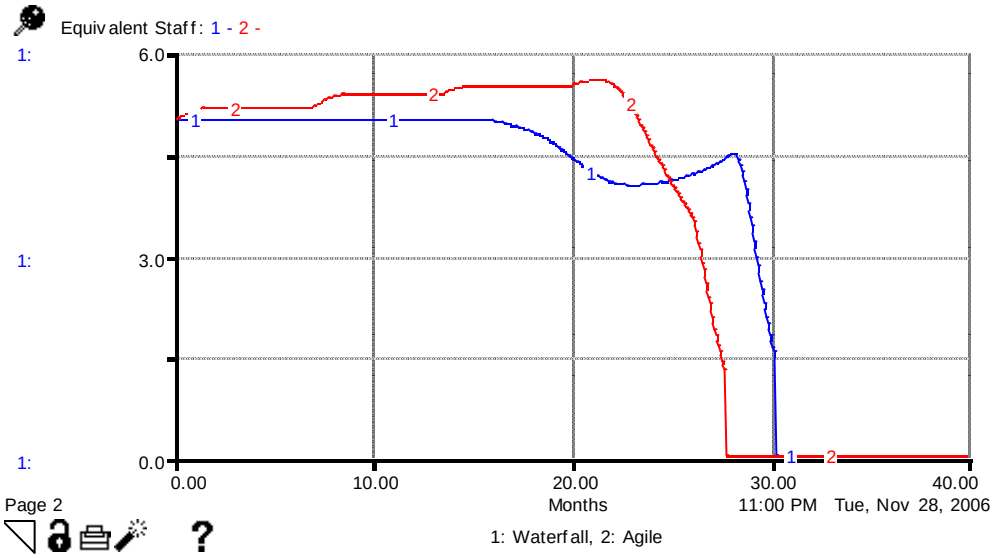
All of this leads to a system that is neither schedule-driven nor feature-driven, though both of these aspects still play important roles. The decision to release is based more on "Is it working?" by some standard that will be different for a weekly (intermediate) release than for an "official" release. "Is there something useful to someone?" is the second most important question. It doesn't really matter anymore how much is done or exactly what features are there because another release isn't far behind (and a less-stable weekly release is even closer behind if the customer is willing to take the risk).

To clarify the standards, the weekly releases don't have a very high threshold. Pretty much if it built and passed all the automated tests, it can be released. The monthly (or bimonthly) release needs to pass more stringent tests, but even these can be relaxed in the face of the weekly releases that can patch a problem (and the confidence one gets from automated regression tests). This means that both discovered rework and undiscovered rework can move from release to release. Indeed, the fact of trying to meet a monthly target – and that the fact of a periodic release is more important than what is in it – means that some of the *original* work will also move from one release to another.

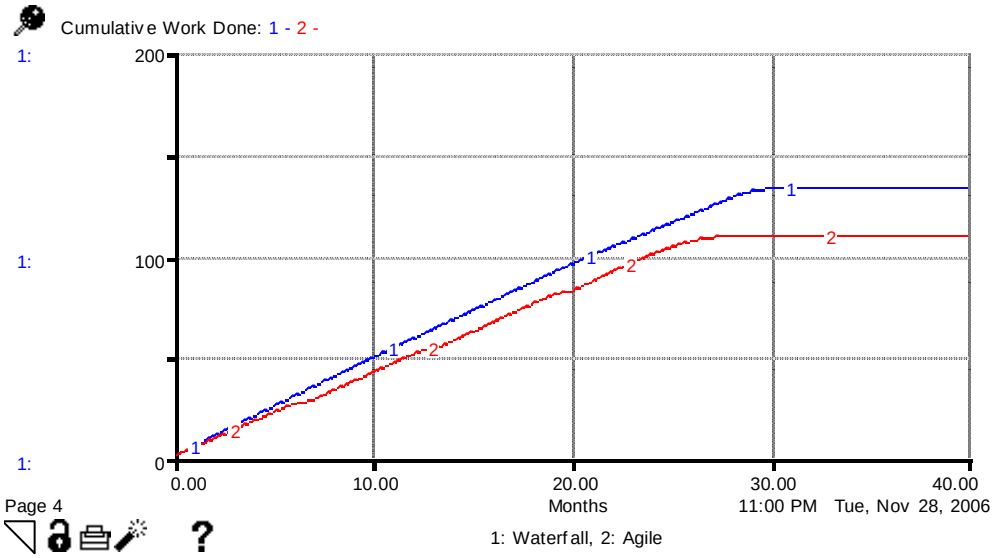
The combination of automated builds and tests, writing tests first, and frequent releases all lead to a reduction in the time to discover rework. This reduces the size of undiscovered rework, which reduces the gain of the errors-on-errors reinforcing loop, thus allowing an Agile project to complete in advance of an equivalent waterfall project.

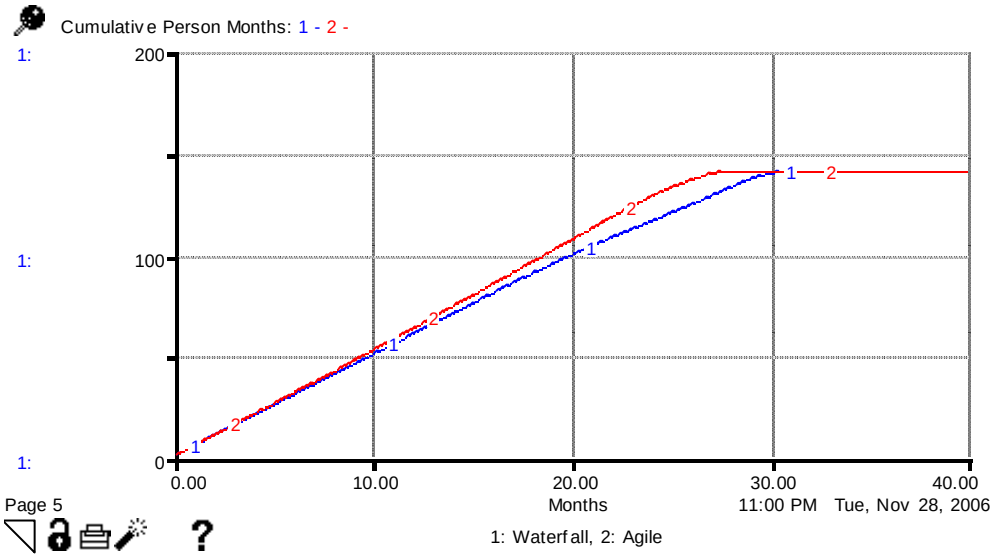
Base Run

The base run is a consistent schedule, waterfall vs. Agile. The consistent schedule chosen was 5 experienced staff members working on 100 tasks over 29 weeks with an estimated rework fraction of 0.3. It was expected that Agile would finish close to the waterfall model, perhaps a little earlier. As can be seen, it finishes a month early while the waterfall model finishes a month late.

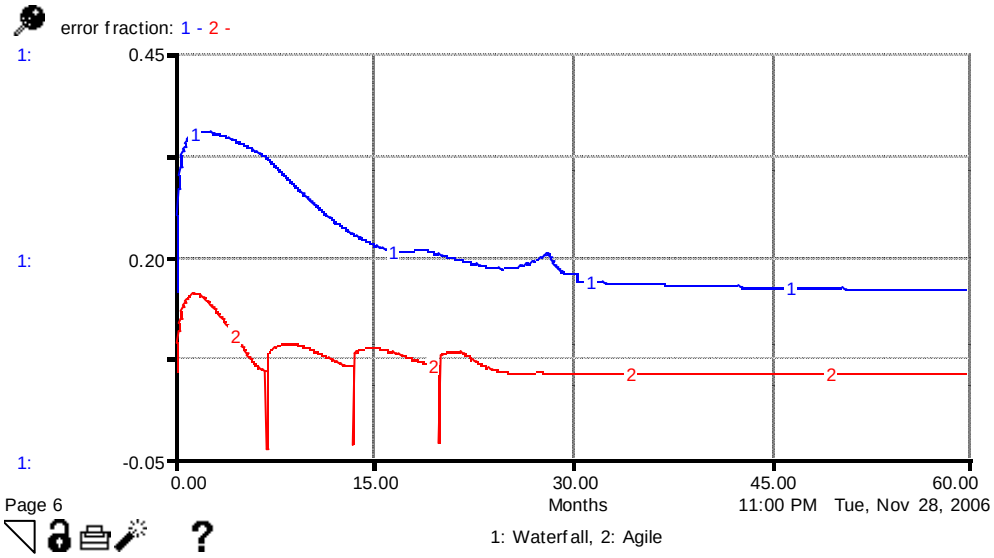


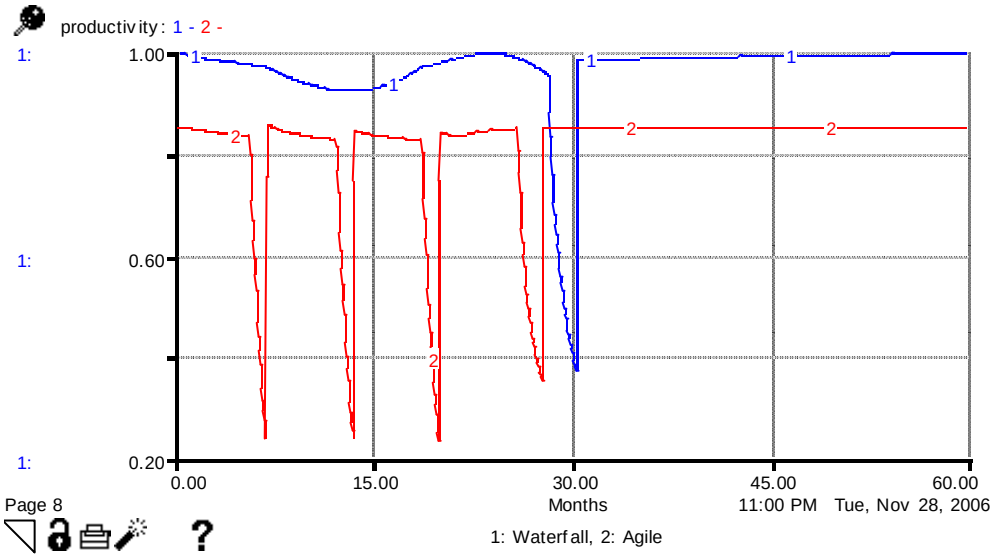
While the Agile project required less work to be done, it cost exactly the same as the Waterfall project:

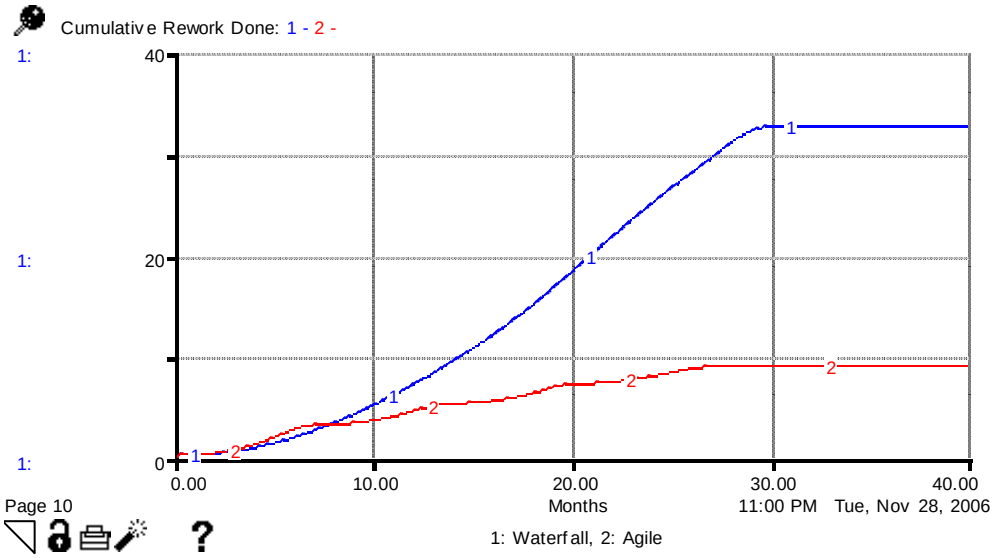




The error fractions and productivities are shown below (the error fraction spikes are just a DT-length artifact of switching phases). Note they are both lower in the Agile case (as expected – they are both lower because of the features built into the process to keep quality high). It is also interesting to note the total rework done in each case (shown last, and again lower in the Agile case).



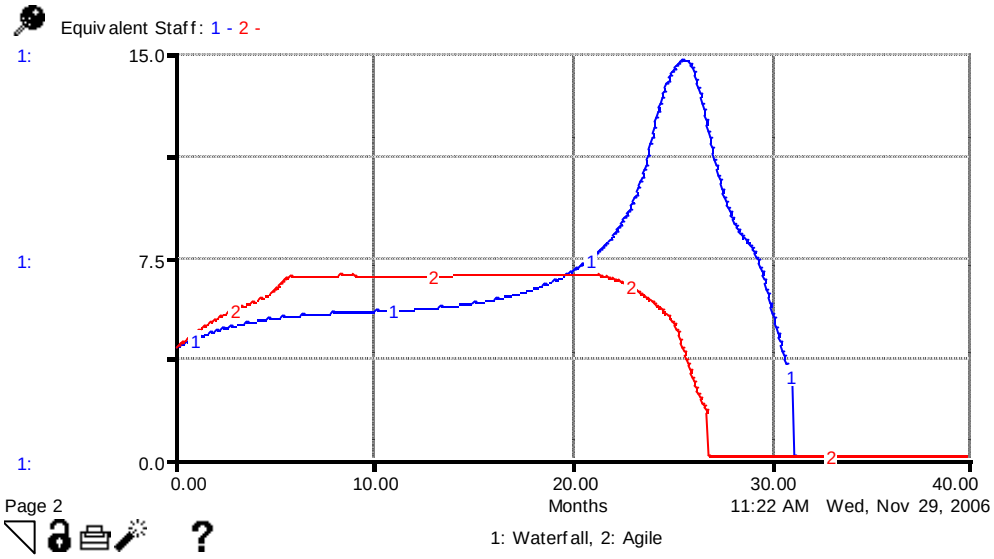

productivity: 1 - 2 -
1:
1.00
0.60
0.20
0.00
15.00
30.00
45.00
60.00
Months
Page 8
11:00 PM Tue, Nov 28, 2006
1: Waterfall, 2: Agile


Cumulative Rework Done: 1 - 2 -
1:
40
20
0
0.00
10.00
20.00
30.00
40.00
Months
Page 10
11:00 PM Tue, Nov 28, 2006
1: Waterfall, 2: Agile

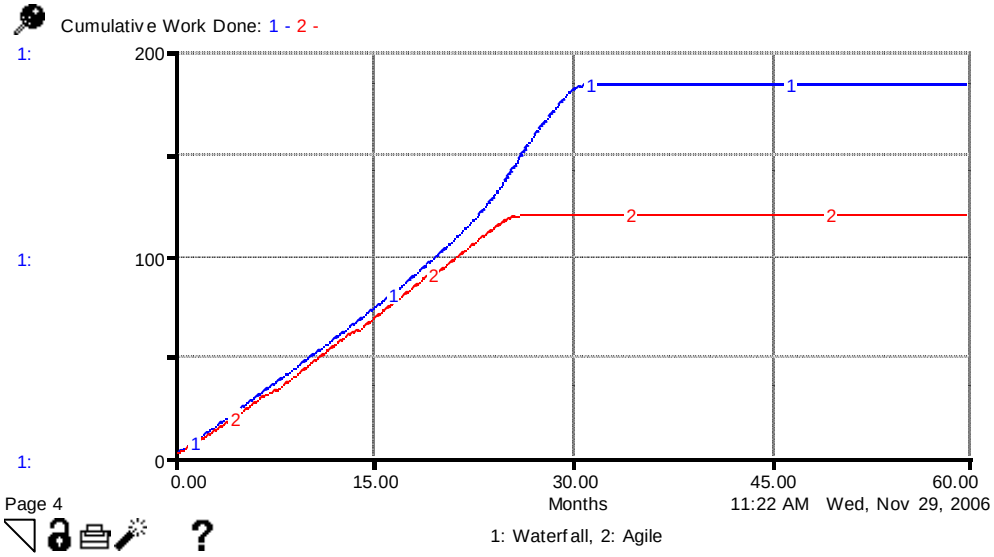
Analyses

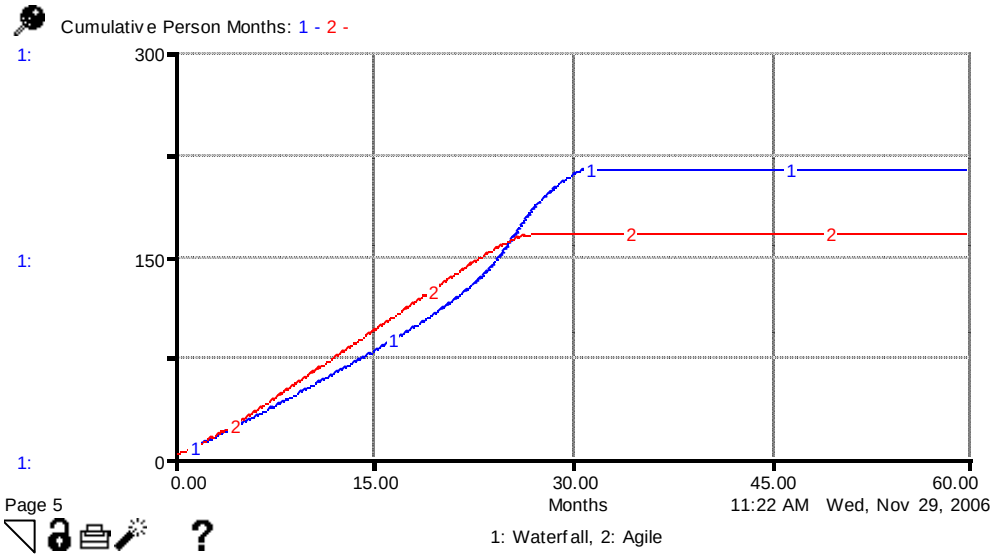
Inconsistent Project

The obvious first question is how much better (if any) is Agile with an inconsistent case? The inconsistent case chosen is starting with 4 experienced staff members to finish 100 tasks in 25 weeks. In this situation, Agile finishes in month 27 (2 months late) vs. waterfall's month 31. Note the more consistent staffing with the shorter cycles. Also note that required staffing grows quicker in the Agile case. This is due to the pressure of the shorter phases; we know sooner that we need more staff members. This earlier feedback alone gives Agile an edge over waterfall.

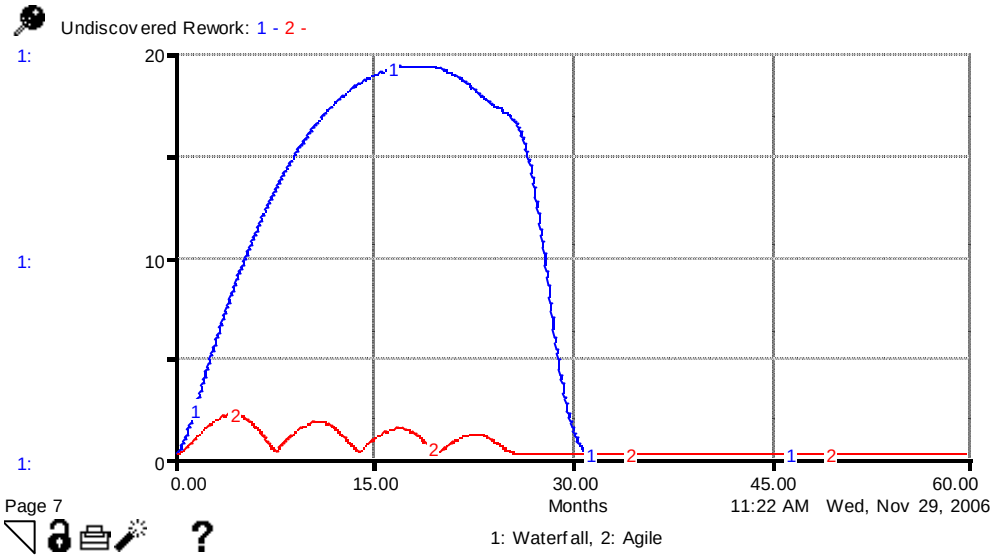


The lower error fractions in Agile lead to less work being done. This combined with the significantly earlier finish leads to lower costs.

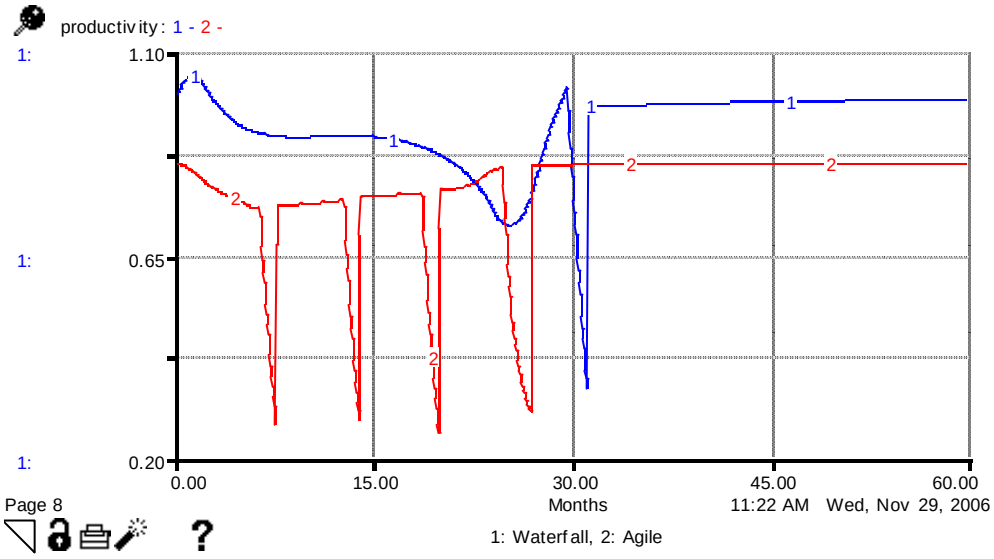


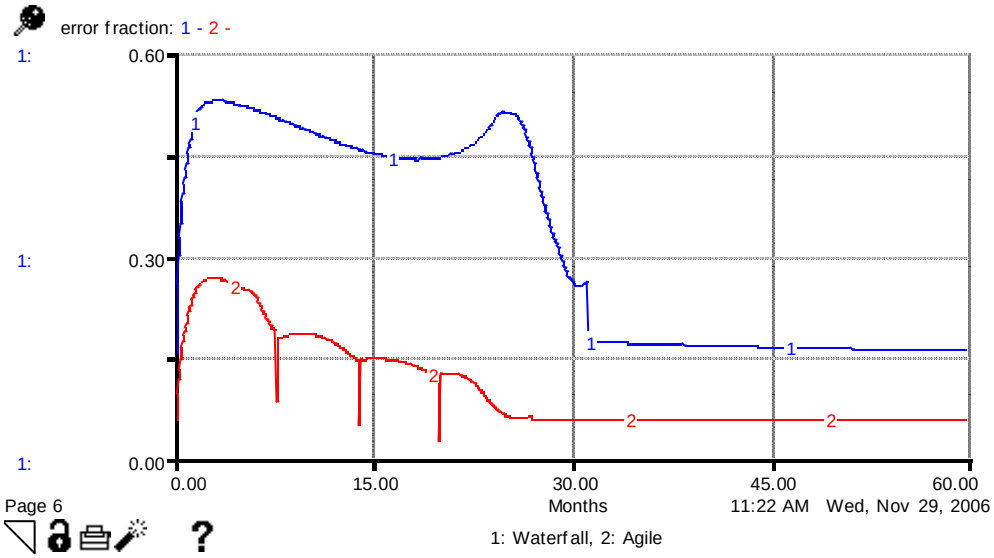


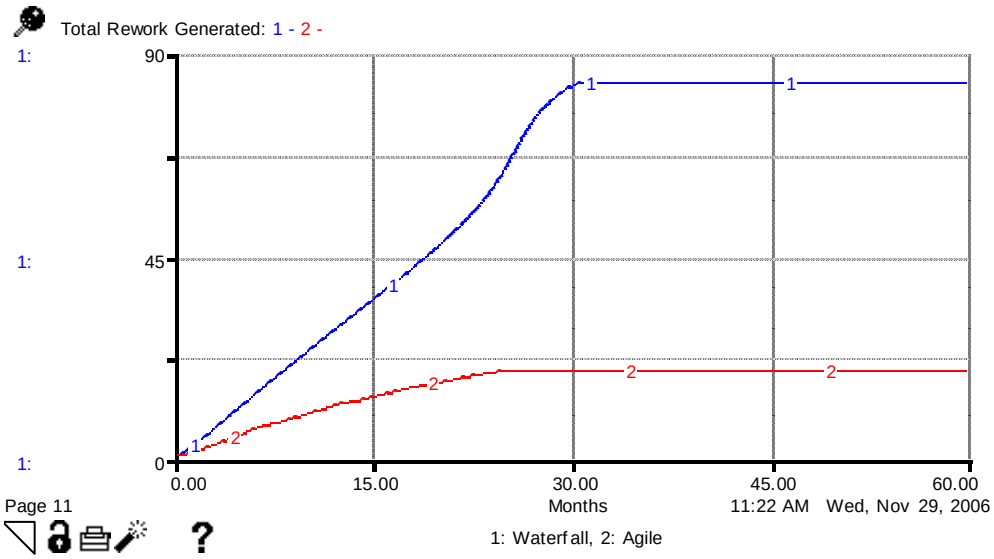
Note the fact that we develop the project in phases with minimum quality requirements means that *Undiscovered Rework* never grows out of control, so we do not get the large error-on-error effects we see in the waterfall case.



Note these benefits are accrued despite having lower productivity due. This is because the focus on high quality (which causes much of this productivity loss) keeps the error fraction and the rework generated low.

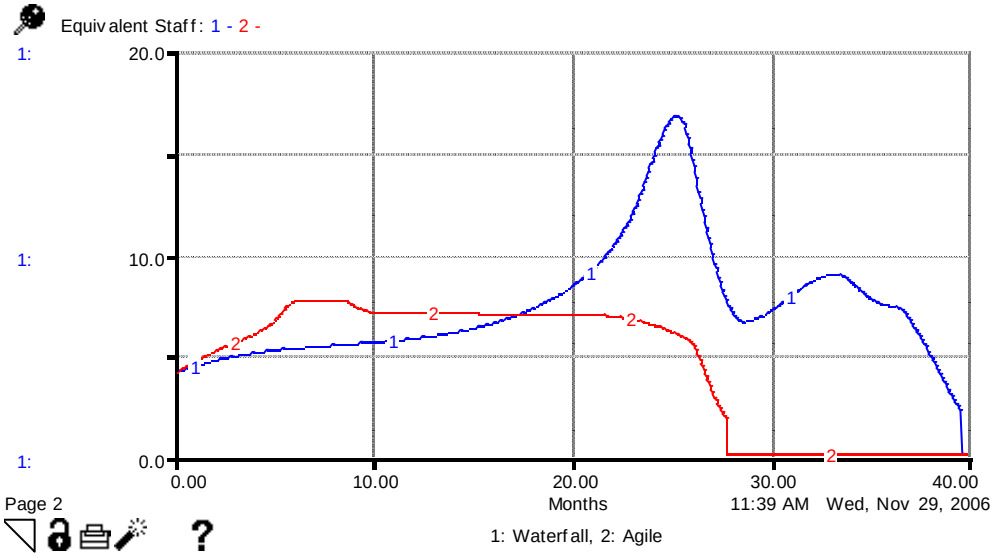

productivity: 1 - 2 -
1:
1.10
0.65
0.20
0.00
15.00
30.00
45.00
60.00
Months
Page 8
11:22 AM Wed, Nov 29, 2006
1: Waterfall, 2: Agile

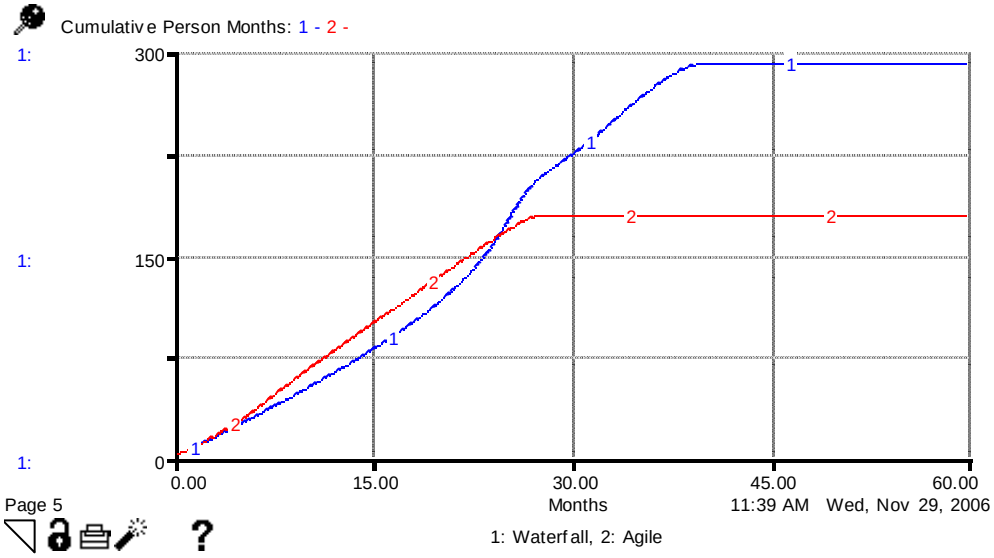

error fraction: 1 - 2 -
1:
0.60
0.30
0.00
0.00
15.00
30.00
45.00
60.00
Months
Page 6
11:22 AM Wed, Nov 29, 2006
1: Waterfall, 2: Agile


Total Rework Generated: 1 - 2 -
1:
90
45
0
0.00
15.00
30.00
45.00
60.00
Months
Page 11
11:22 AM Wed, Nov 29, 2006
1: Waterfall, 2: Agile

Inconsistent Project with Uncertain Customer Requirements

In the face of uncertain customer requirements, which Agile was designed to address, Agile performs even better. Agile still finishes around month 27 (at month 27.8 - 2.8 months late) while waterfall runs almost out to month 40 (at month 39.8 - 14.8 months late). The cost for waterfall is almost 50% higher than Agile, while the overall Agile cost has changed very little from the case without uncertain customer requirements.





Summary of Inconsistent Missions

The following tables summarize the two inconsistent mission cases, first without and then with uncertain customer requirements. Note that Agile development provides a definite advantage to projects that have an inconsistent mission, finishing 14% earlier at about 75% of the cost.

Project Type	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
Waterfall:	31.25		212.84		183.04	
Agile:	26.9375	-13.80	163.82	-23.03	118.68	-35.16

With uncertain customer requirements, Agile really shines, cutting 30% off the project length (only two months late vs. 15 months late) at around 60% of the cost. Clearly, Agile lives up to its purpose of resiliency in the face of changing customer requirements.

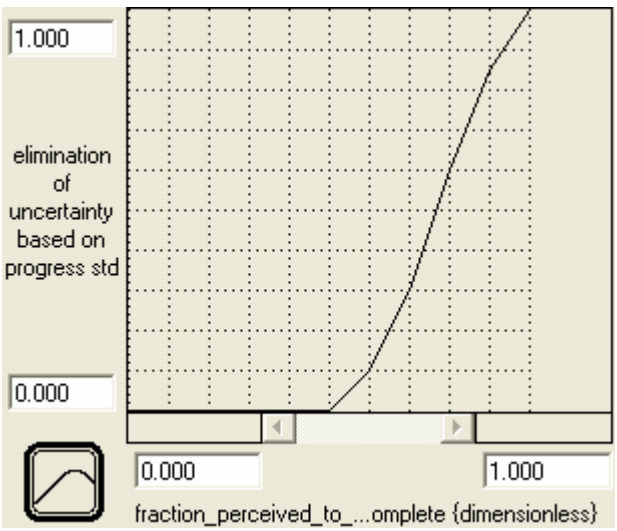
Project Type	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
Waterfall:	39.75		290.98		258.11	
Agile:	27.875	-29.87	178.58	-38.63	127.93	-50.44

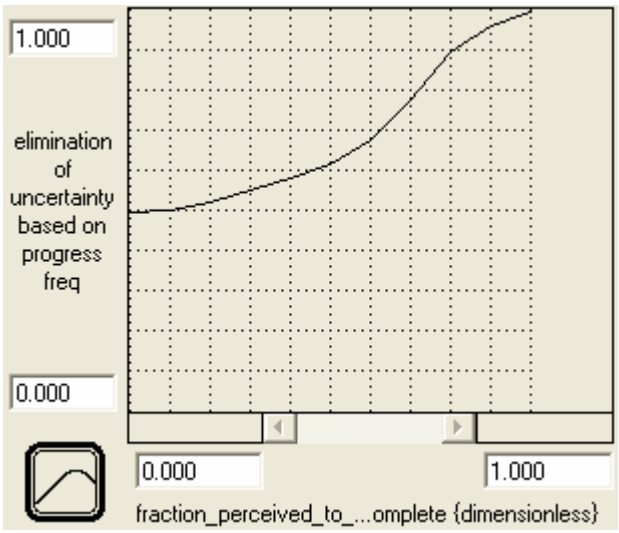
Analysis of Benefits of Individual Aspects of Agile

Benefits of Frequent Releases and Customer Interactions

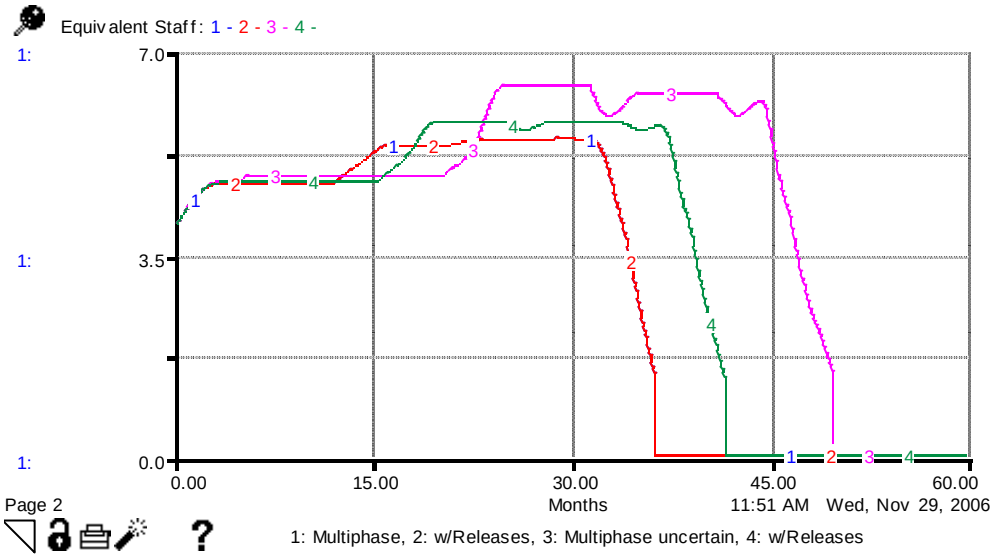
For this test, the base case has all Agile effects turned off. This will then be compared to the same case with only the frequent release effect turned on, both with and without uncertain customer requirements. The inconsistent mission used previously will be retained as a baseline.

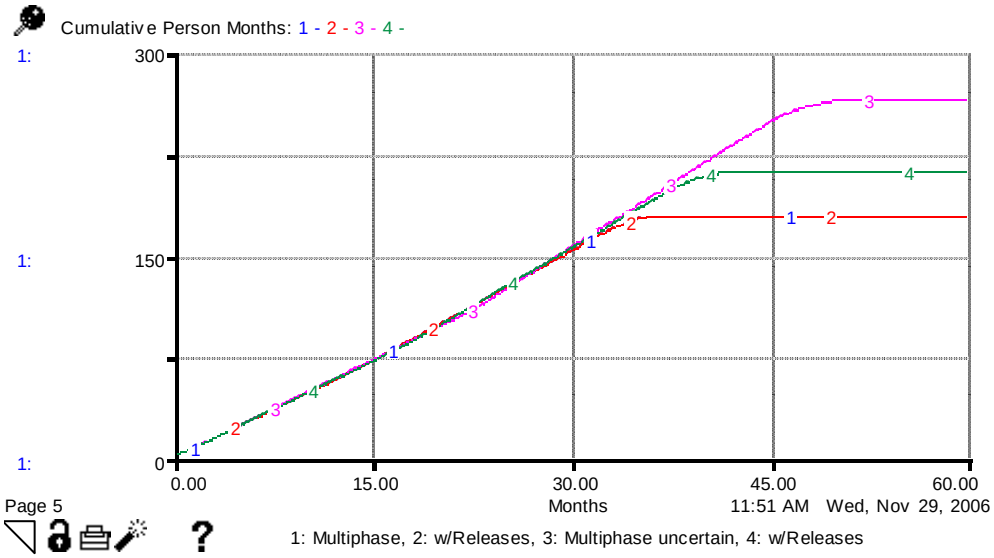
Frequent releases serve to reduce customer uncertainty, as do frequent customer interactions. This cuts the uncertainty in half, as well as changing the shape of the curve to be more consistent. The traditional shape for waterfall is shown on the left below while the revised shape for Agile on the right.





As shown in the following graph, there frequent releases and customer interactions have little to no effect if there are not changing customer requirements (graphs 1 and 2). However, in the case of uncertain customer requirements (graphs 3 and 4), there is a marked improvement, finishing after 41.75 months rather than 50) when there are frequent releases and customer interactions. Not surprisingly, the cost is also much lower.





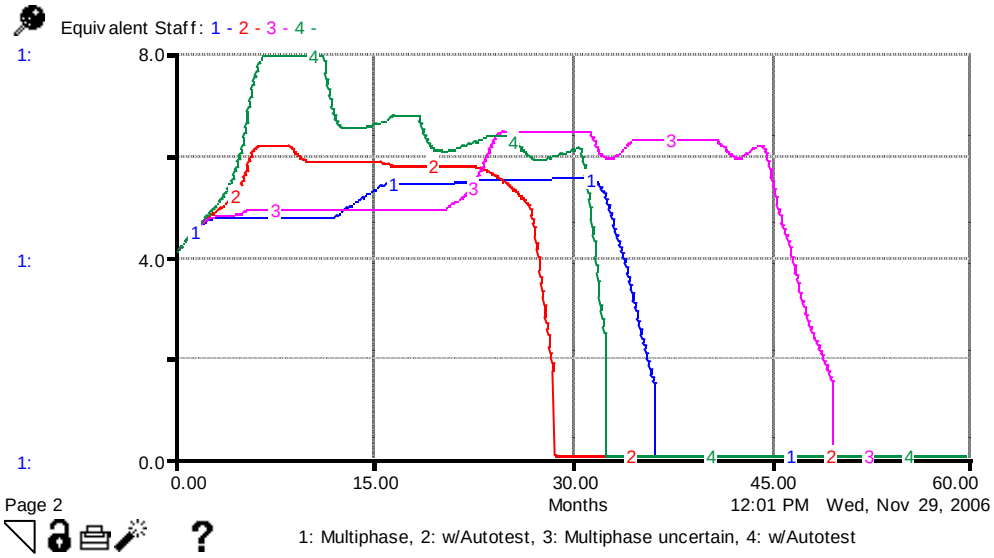
Benefits of Nightly Builds and Automated Testing

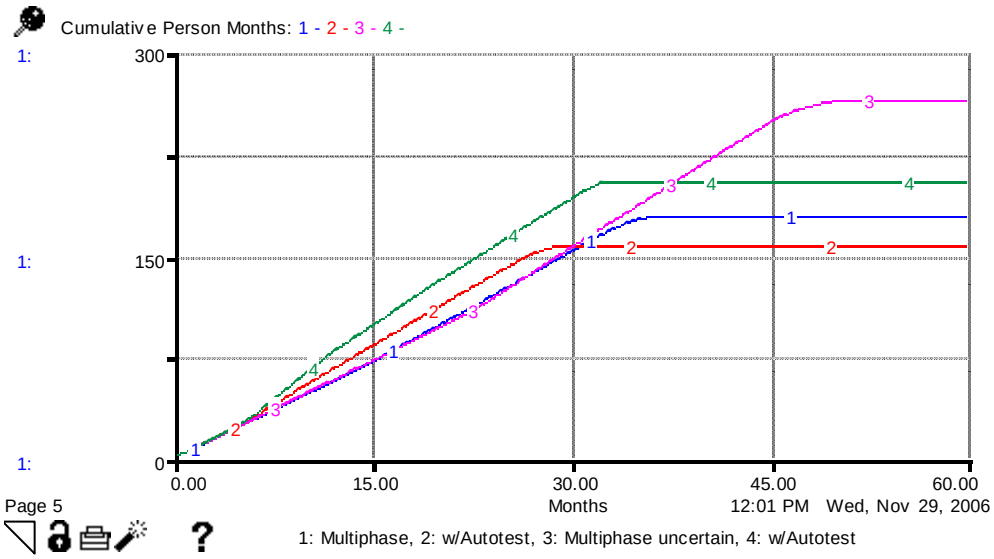
For this test, the base case has all Agile effects turned off. This will then be compared to the same case with only the automated testing effect turned on, both with and without uncertain customer requirements. The inconsistent mission used previously will be retained as a baseline.

Nightly build and automated tests lead directly to a shorter rework discovery delay (reduces the maximum from 12 months to 12/*phases* months, where *phases* is the number of phases in the project). The *maximum time to discovery rework* will also be varied to see if the model is sensitive to its value.

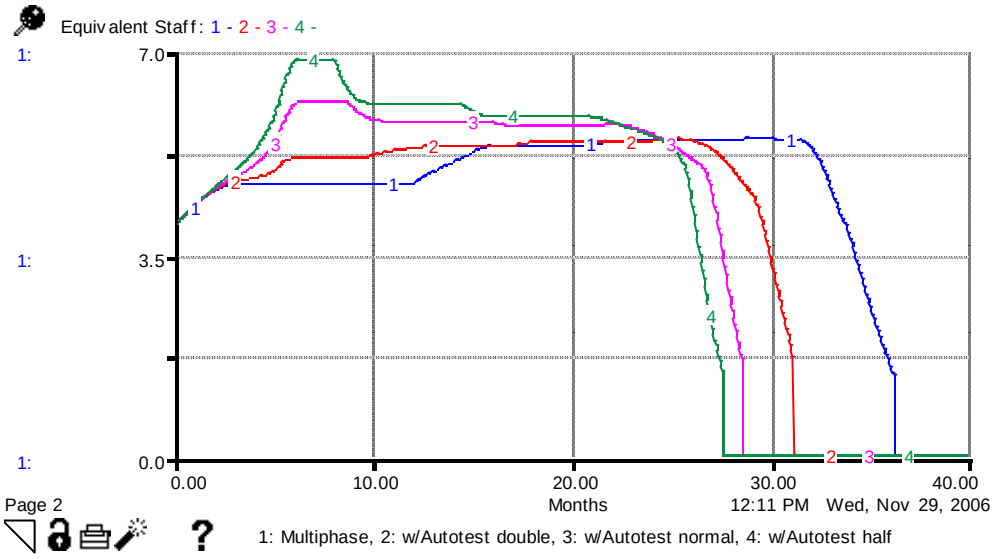
Note that automated testing has a significant effect both without (graphs 1 and 2) and with (graphs 3 and 4) uncertain customer requirements. In the former case, the project finishes 7.5 months earlier (month 28.7 vs. 36.25), while in the latter case the project finishes 17.5 months

earlier (month 32.5 vs. 49.8). Once again the respective costs are much lower, although the work completed in each case is virtually identical (rework generated is almost the same).

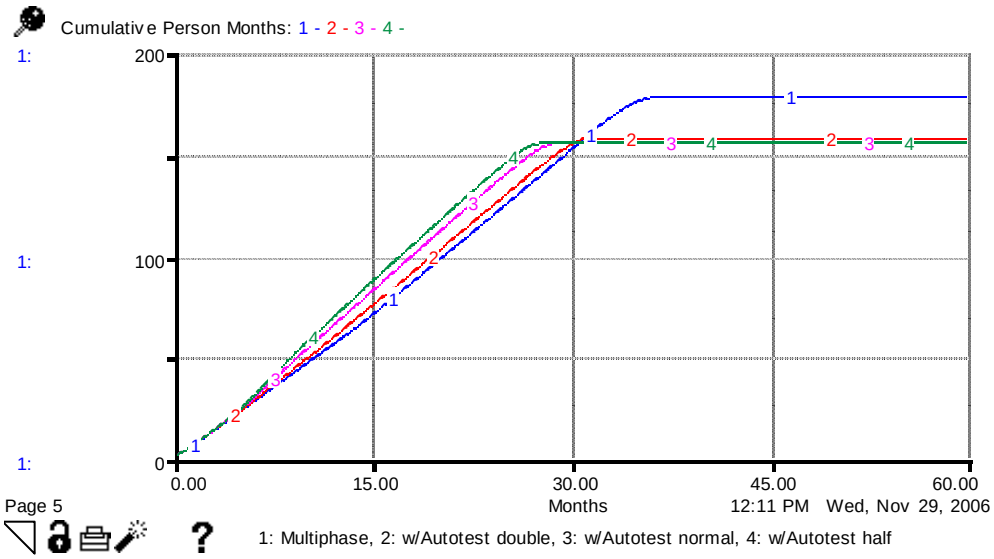




To test the sensitivity of the *maximum time to discover rework*, the base case (curve 1) was compared against the automated tests case with the original time (curve 3), double the original time (curve 2) and half the original time (curve 4). It is clear from the progression of the curves that the model is somewhat sensitive to this value. However, halving it made little difference (only a one month improvement), while doubling it still provided a significant effect (5 months better) over the base case. Halving it also led to an excessive ramp-up in the first phase, suggesting the value might be too small.



These changes had little impact on the project cost.



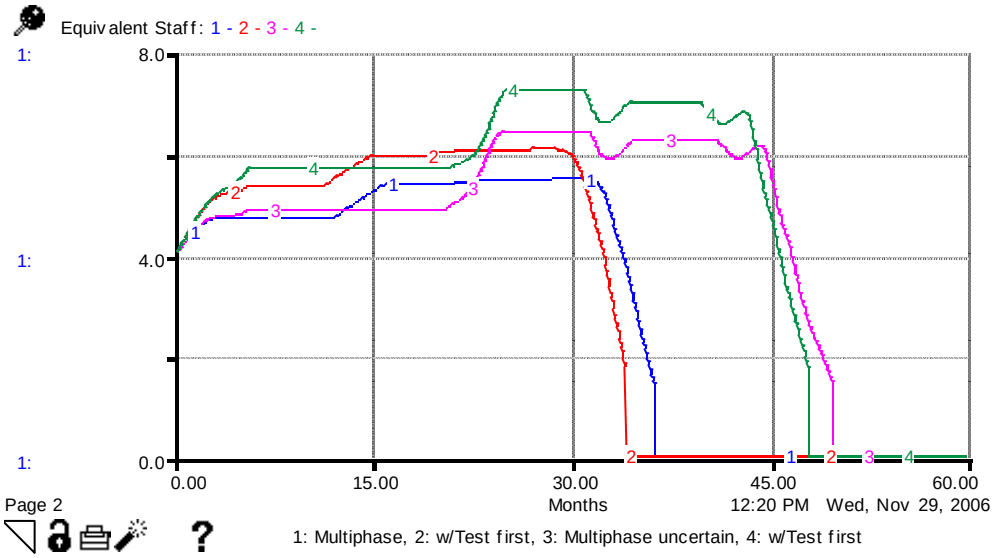
Benefits of "Test First"

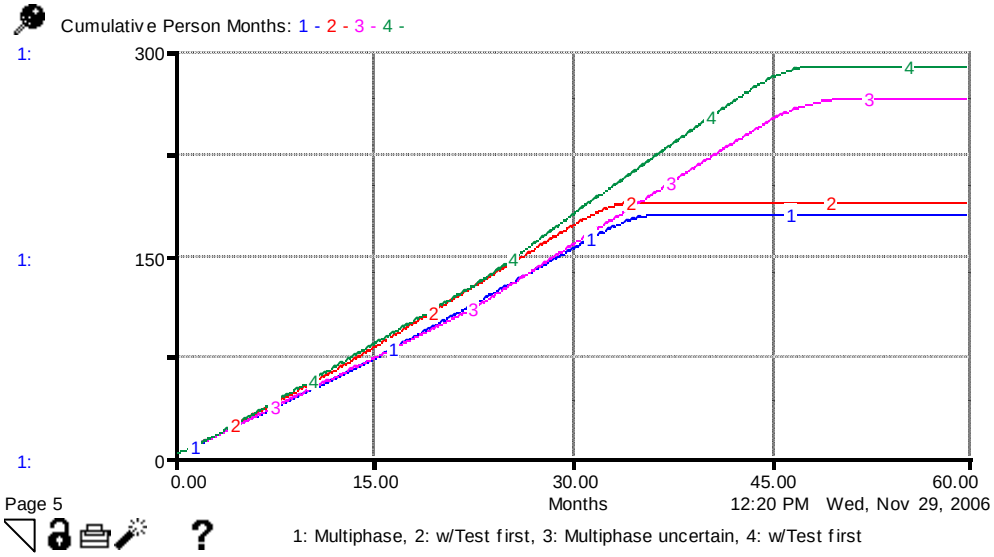
For this test, the base case has all Agile effects turned off. This will then be compared to the same case with only the "test first" effect turned on, both with and without uncertain customer requirements. The inconsistent mission used previously will be retained as a baseline.

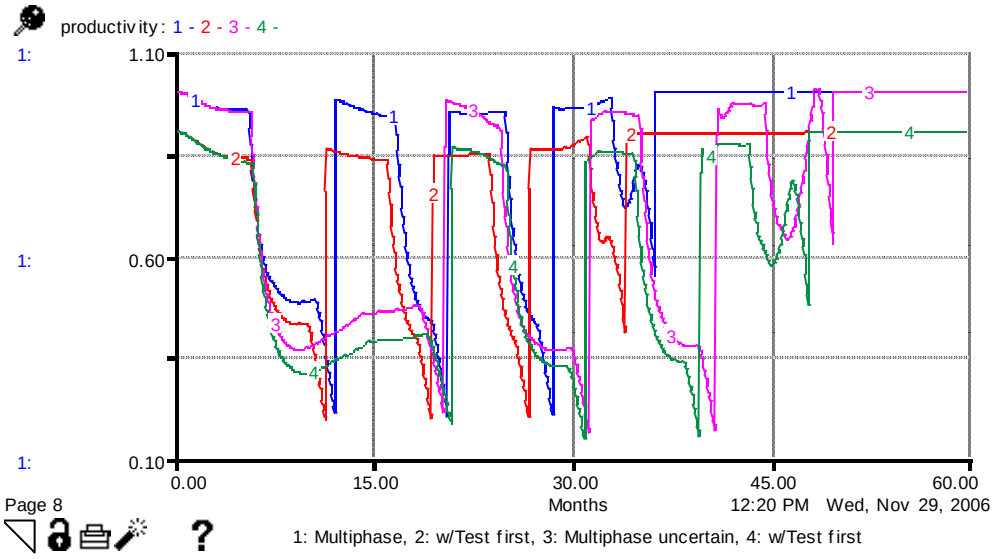
The "test first" effect lowers productivity (and slows initial progress) by 10% as tests are written instead of shipping code[2]. It also decreases the normal error fraction by 5%.

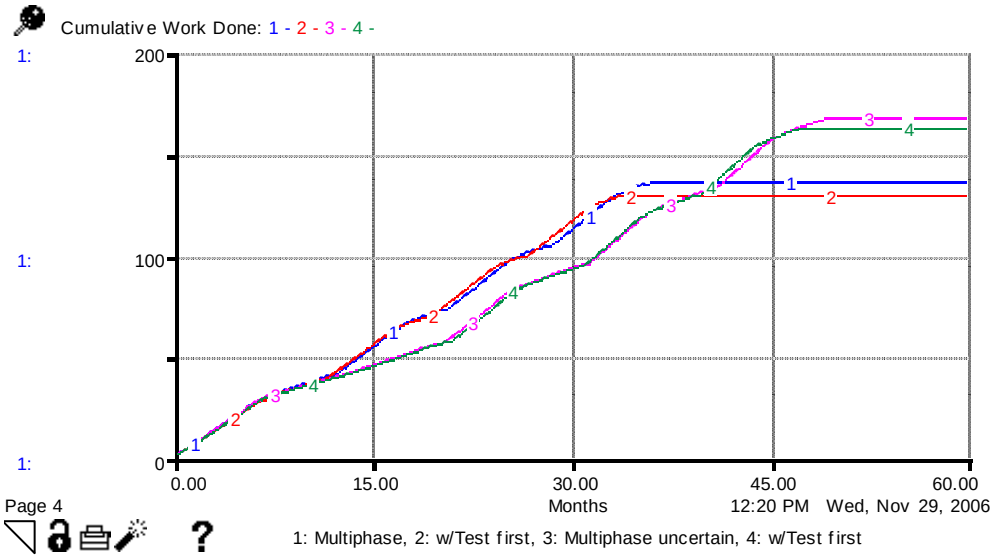
[2] Note this also could have been implemented by increasing the scope of the project. I felt decreasing the productivity more accurately represented what actually happens.

Note that “test first” gives about a two month improvement in the delivery date in both cases, though it is slightly less in the face of uncertain customer requirements (curves 3 and 4). The project cost, however, is higher, especially in the face of uncertain customer requirements – despite slightly lower total work done due to a lower error fraction. This is because productivity is lower in “test first” due to the time required to write the tests. The cost is that much higher when there are changing customer requirements because the project runs longer at the lower productivity.









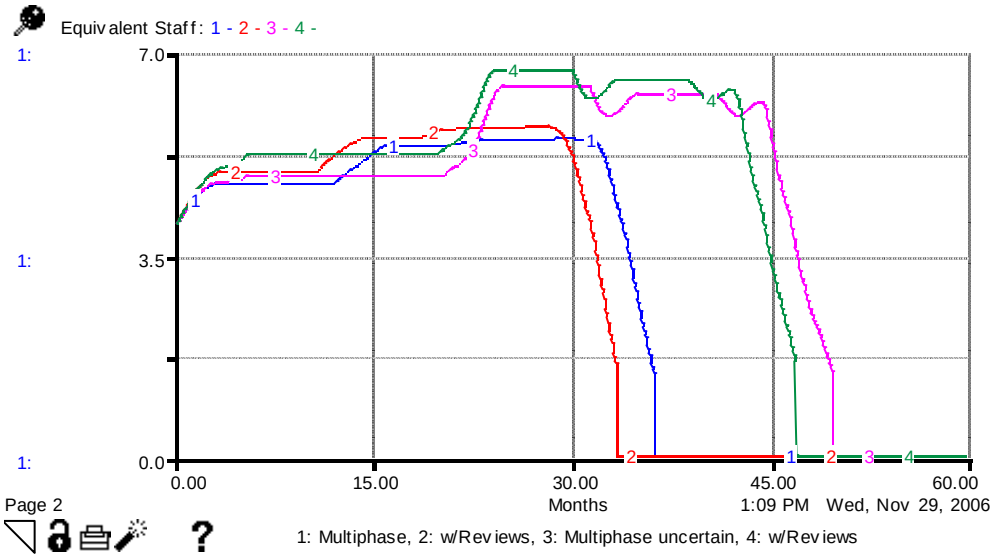
Benefits of Design and Code Reviews

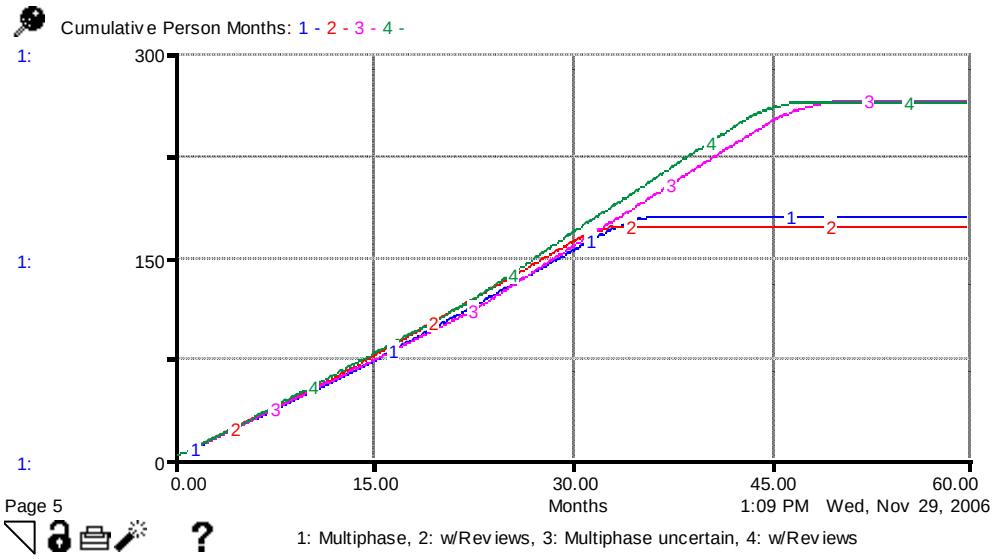
For this test, the base case has all Agile effects turned off. This will then be compared to the same case with only the reviews effect turned on, both with and without uncertain customer requirements. The inconsistent mission used previously will be retained as a baseline.

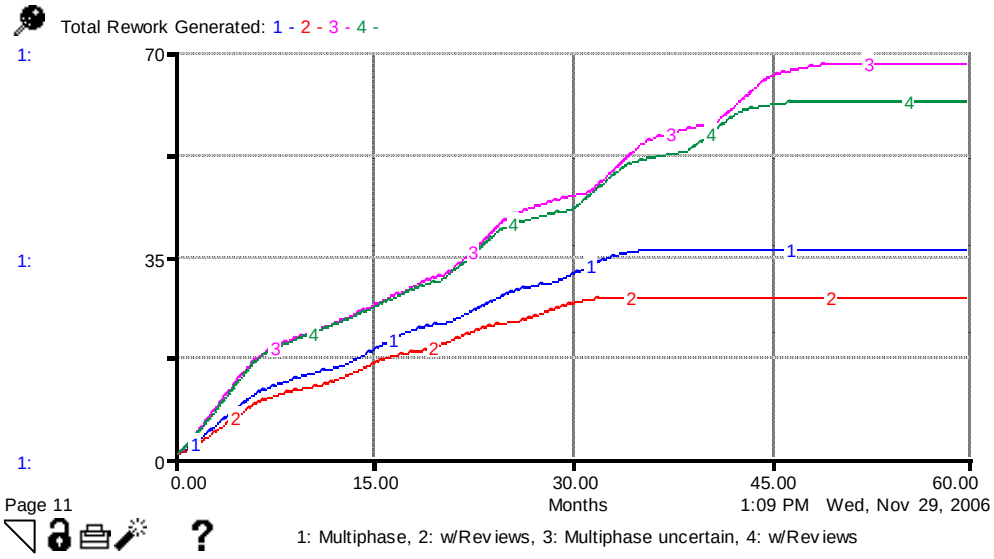
Frequent reviews between developers, including pair programming (which I think most people can only tolerate in very short doses) and face-to-face communication, leads to both a lower normal error fraction and a lower productivity (5% reduction on both). Continuous attention to technical excellence and design may also lower productivity somewhat, but definitely lowers error fraction. It has been assumed that the 5% reduction already given for reviews sufficiently covers this effect as well.

Note that having design and code reviews improves the project schedule by about three months in both cases at about the same cost. There is a modest cost savings (about 4%) without

customer changes (curves 1 and 2). These are tied to proportionally lower levels of rework generation (because of a lower error rate), and hence, total work done.





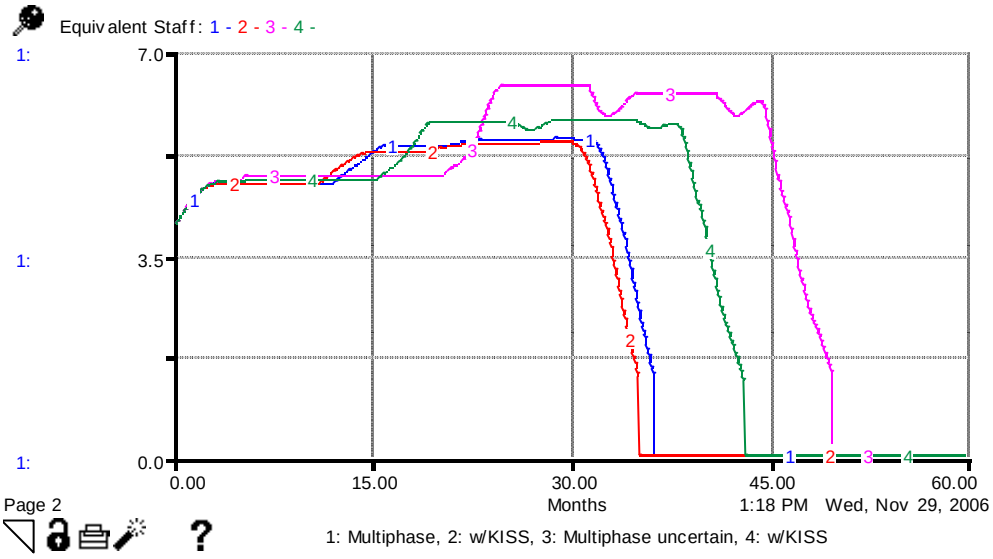


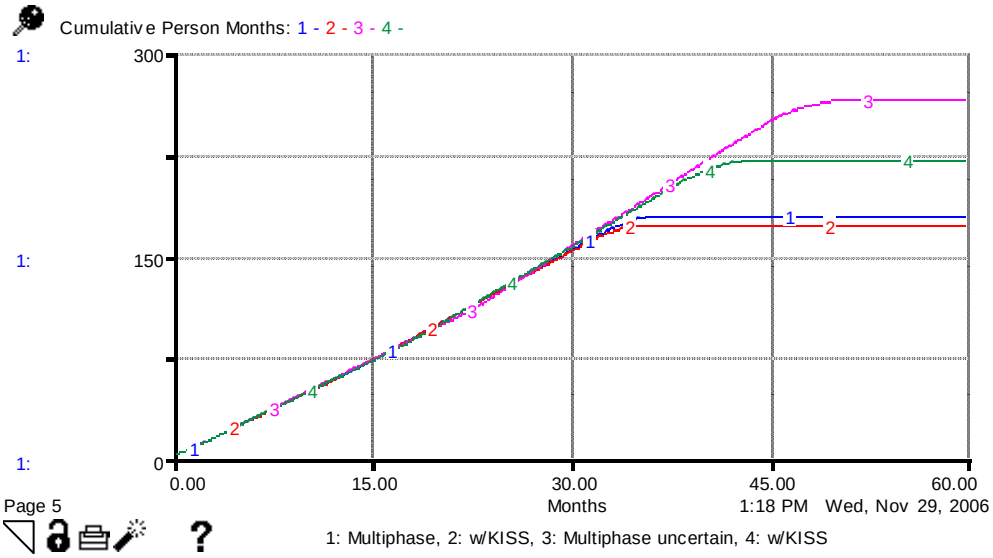
Benefits of Keeping Things Simple

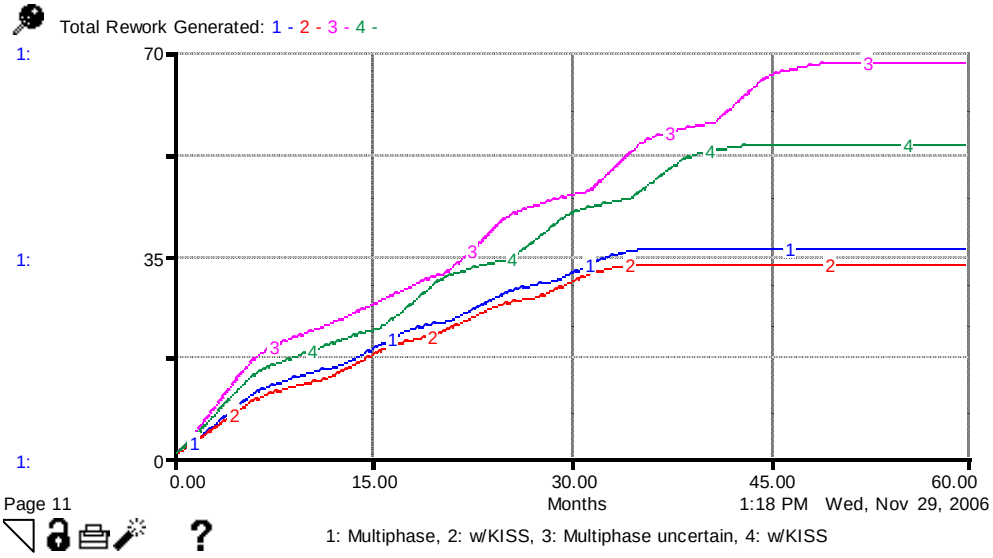
For this test, the base case has all Agile effects turned off. This will then be compared to the same case with only the KISS effect turned on, both with and without uncertain customer requirements. The inconsistent mission used previously will be retained as a baseline.

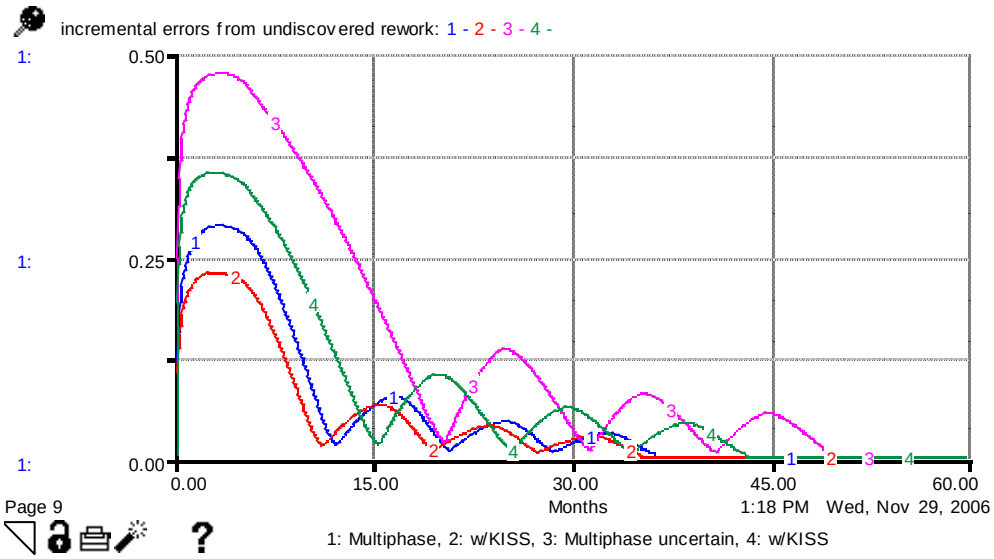
Choosing simple over complicated both reduces error fraction and the effect of rework for uncertain customer requirements (because you haven't yet developed everything that the customer may be changing). Rather than reducing the error fraction, I decided it made more sense to reduce the strength of the errors on errors effect because that is where complexity is likely to cause the worst trouble (reduce by 10%). The uncertain customer requirements effect was also reduced another 5%.

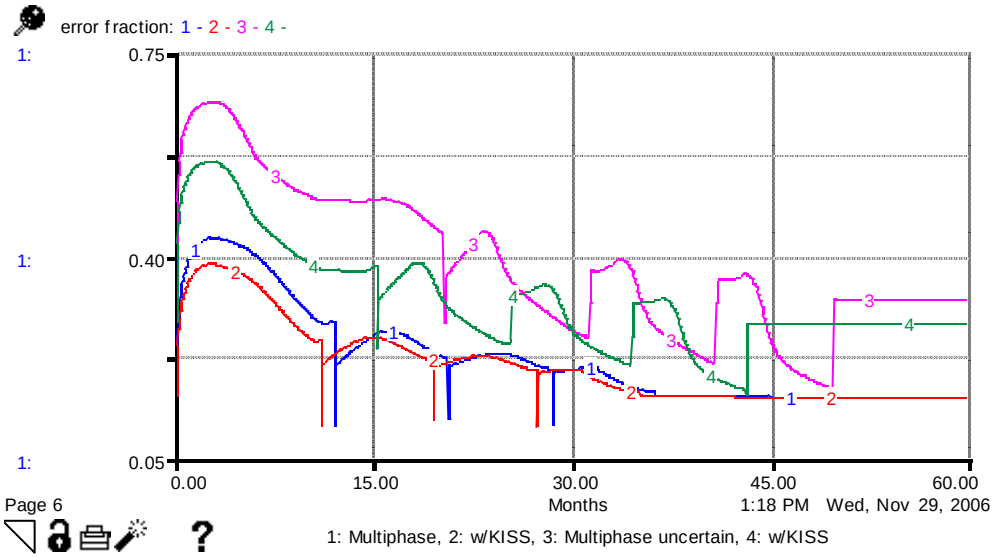
Note that choosing simpler solutions has a minor impact (1 month - curves 1 and 2) on both the schedule and the cost when customer requirements are not changing and a dramatic impact (6.5 months - curves 3 and 4) when they are. These changes are directly tied to the amount of rework generated. The improvements can be clearly seen in the graphs of *incremental errors from undiscovered rework* (which includes discovered rework as well) and *error fraction*.


Equivalent Staff: 1 - 2 - 3 - 4 -
1:
7.0
1:
3.5
1:
0.0
0.00
15.00
30.00
45.00
60.00
Months
Page 2
1:18 PM Wed, Nov 29, 2006
1: Multiphase, 2: w/KISS, 3: Multiphase uncertain, 4: w/KISS


Cumulative Person Months: 1 - 2 - 3 - 4 -
1:
300
1:
150
1:
0
0.00
15.00
30.00
45.00
60.00
Months
Page 5
1:18 PM Wed, Nov 29, 2006
1: Multiphase, 2: w/KISS, 3: Multiphase uncertain, 4: w/KISS


Total Rework Generated: 1 - 2 - 3 - 4 -
1:
70
1:
35
1:
0
0.00
15.00
30.00
45.00
60.00
Months
Page 11
1:18 PM Wed, Nov 29, 2006
1: Multiphase, 2: w/KISS, 3: Multiphase uncertain, 4: w/KISS





Summary of Individual Effects

The above results are summarized in the following two tables. The first shows the results without uncertain customer requirements, while the second shows the results with uncertain customer requirements. As can be seen from the table below, frequent releases have no impact when there are not changing customer requirements. On the other hand, nightly builds and automated tests give a tremendous advantage in terms of both time and cost. "Test first" gives a modest gain in schedule at a modest cost. The remaining two effects show modest advantages in both time and cost.

Agile feature (known reqs)	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
Base (none):	36.375		178.33		135.35	
Frequent release:	36.375	0.00	178.33	0.00	135.35	0.00
Nightly builds/test:	28.6875	-21.13	155.13	-13.01	133.64	-1.26
"Test First":	34.125	-6.19	186.42	4.54	128.70	-4.91
Design/Code reviews:	33.4375	-8.08	170.68	-4.29	127.01	-6.16
KISS:	35.1875	-3.26	171.12	-4.04	132.51	-2.10

The story is a little different when there are changing customer requirements. Frequent releases, nightly builds, and avoiding complexity have strong schedule and cost gains while reviews have modest schedule gains with no difference in cost. "Test First" has a smaller impact than in the previous case at a higher cost.

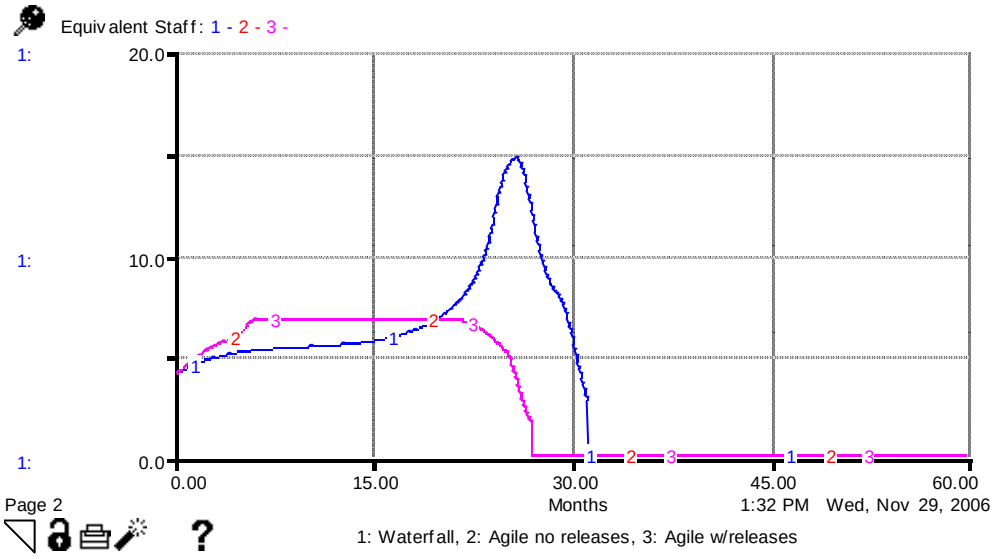
Does this then mean that the doctrine of "test first" should be abandoned? Unfortunately, the automated tests that run with nightly builds (yielding a very strong benefit in both time and cost) depend on the automated tests being written. Whether they are written first or last does not directly affect the productivity, though writing them last tends to increase the error fraction, worsening the results shown in the table. In other words, to reap the benefits of nightly automated tests, the tests *must* be written and it is better to write them first rather than last. This is not a potential practice to drop.

Agile feature (uncertain reqs)	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
Base (none):	49.875		263.66		167.57	
Frequent release:	41.625	-16.54	210.61	-20.12	148.83	-11.18
Nightly builds/test:	32.5625	-34.71	203.54	-22.80	167.83	0.16
"Test First":	48	-3.76	288.53	9.43	162.74	-2.88
Design/Code reviews:	47.0625	-5.64	263.10	-0.21	160.86	-4.00
KISS:	43.25	-13.28	219.12	-16.89	153.16	-8.60

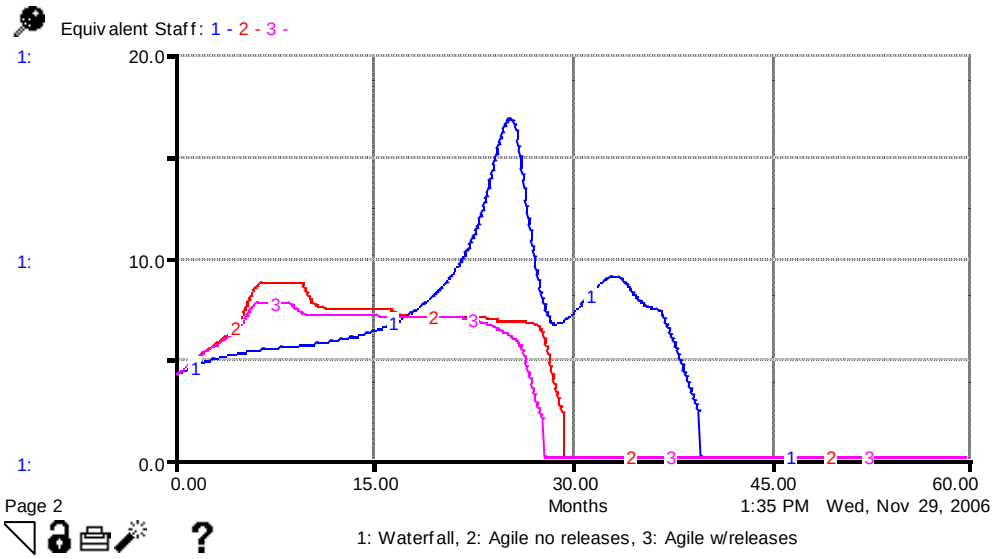
Agile Without Frequent Releases

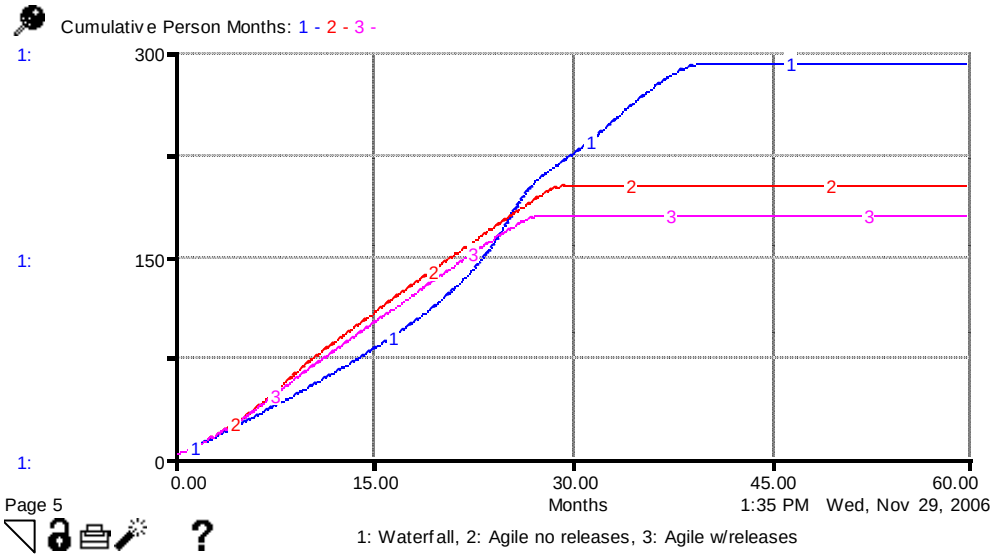
One of the burning questions is whether these process improvements still work without frequent official releases. Since Agile assumes that you are always working in small releasable chunks, even if you don't actually release the product, should this aspect be retained in the comparison? This is hard to answer equivocally because there will be differences between projects in the amount of customer interaction and releases to the customer. These tests have been run assuming these remain the same as for the waterfall case.

Without uncertain customer requirements, there is no difference between frequent releases (curve 3) and not having frequent releases (curve 2). [The waterfall case is included for comparison in curve 1.]



However, when there are uncertain customer requirements, there is an improvement in the schedule (1.5 months) by having frequent releases and customer interaction. In addition, there is a cost savings. This isn't a big surprise because frequent releases showed an improvement on their own.





These results are summarized in the tables below. The first table shows the results without changing customer requirements. Not surprisingly (since we already saw frequent releases have no impact in this case), failing to have frequent releases and customer interactions has no impact.

Project Type	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
Waterfall:	31.25		212.84		183.04	
Agile:	26.9375	-13.80	163.82	-23.03	118.68	-35.16
Agile no rels:	26.9375	-13.80	163.82	-23.03	118.68	-35.16

However, with changing customer requirements, there is a noticeable difference. Failure to have frequent releases increases the project length 5% (1.5 months) and the cost 12%. Note there is also more work to accomplish (11%). This is from additional rework that must be done due to increased *Undiscovered Rework* from changing customer requirements. This also leads to a higher error fraction due to the errors-on-errors feedback, causing even more rework.

Project Type	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
Waterfall:	39.75		290.98		258.11	
Agile:	27.875	-29.87	178.58	-38.63	127.93	-50.44
Agile no rels:	29.3125	-26.26	199.77	-31.35	141.46	-45.19
Cost of no rels:		5.16		11.87		10.58

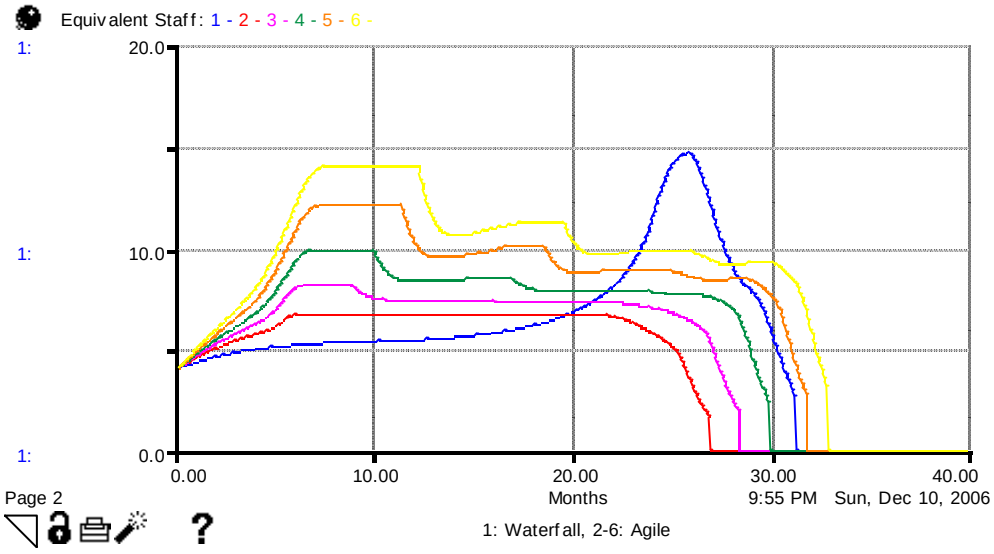
Sensitivity of Assumptions

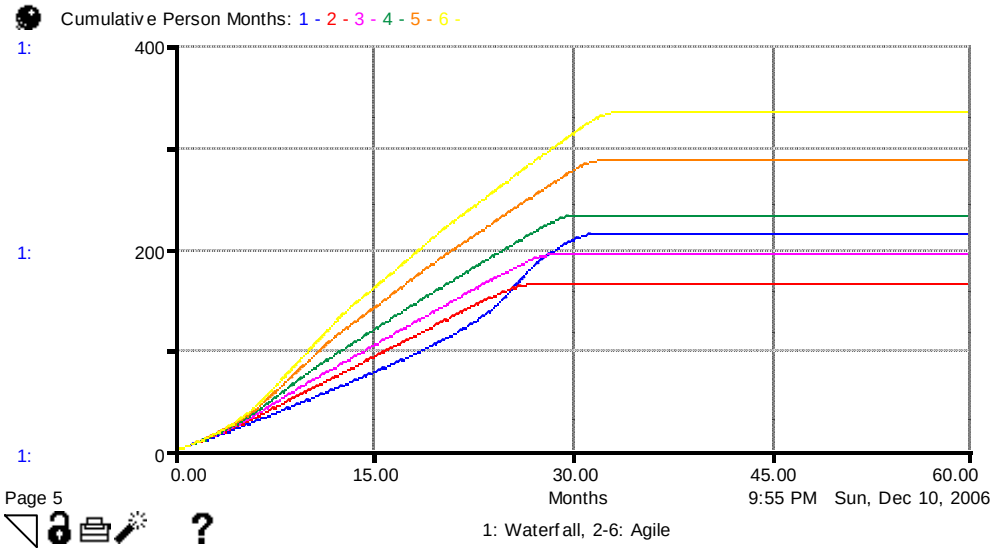
A number of assumptions have been made about how well Agile performs in terms of error fraction and productivity. What if the error fraction improvements in Agile aren't as high as suggested (or the waterfall error fraction is lower)? What if the price of Agile is even higher in terms of productivity? This will very likely be true in the early stages of adoption. Additionally, what if these changes also have a relative impact on new staff members (or maybe we were too optimistic originally)? Using the inconsistent mission, the following series of tests look at a progressive worsening of these parameters, as shown in the table below. Note the relative impact of new staff members in Agile is never worse than the waterfall case, which seems reasonable.

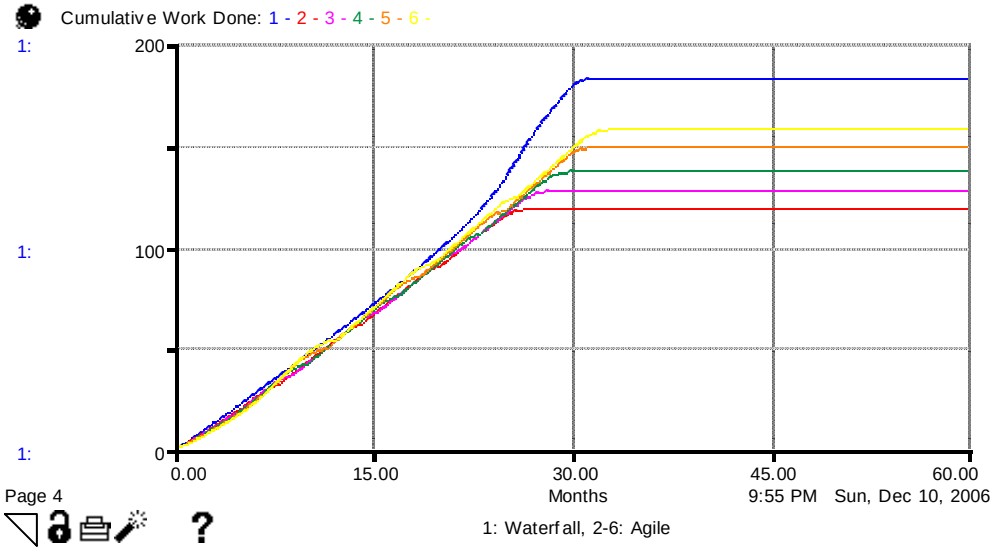
Case number	normal error fraction	Normal productivity	incr. errors: new staff	productivity: new staff
Waterfall:				
1	0.15	1.00	0.50	0.50
Agile:				
2	0.05	0.85	0.35	0.65
3	0.08	0.80	0.40	0.60
4	0.11	0.75	0.45	0.55
5	0.15	0.70	0.50	0.50
6	0.18	0.65	0.50	0.50

The following curves show the staffing behavior, cost, and work done for the six cases. Note that while most of the Agile cases finish before, or close to, the waterfall case, all but two of the Agile cases are more expensive than the waterfall case. This is despite the fact that all of the Agile cases accomplish less work (fewer tasks) over the course of the project.

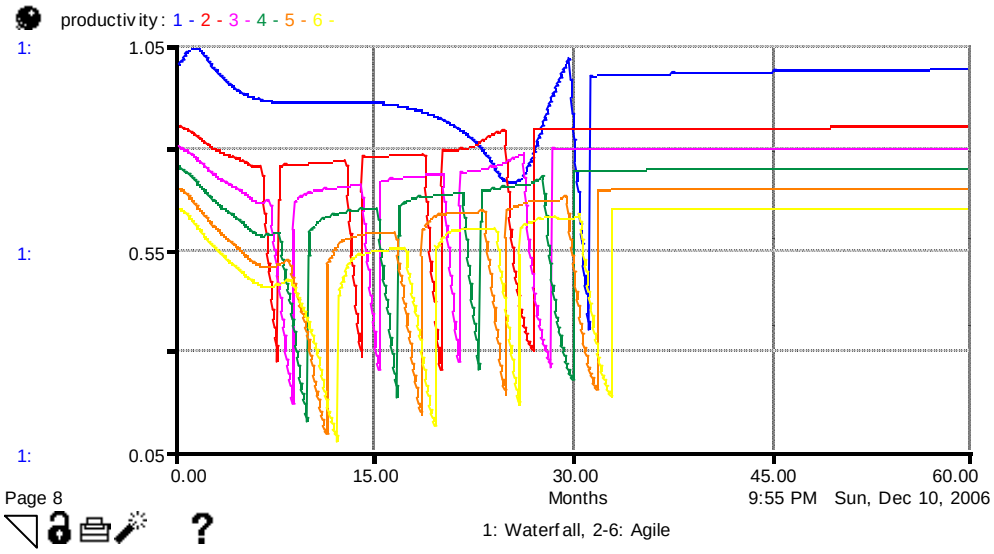
Note also how the first phase of most of the Agile cases over-staffs in an attempt to finish on time. Some of this staff is subsequently let go in the second phase.



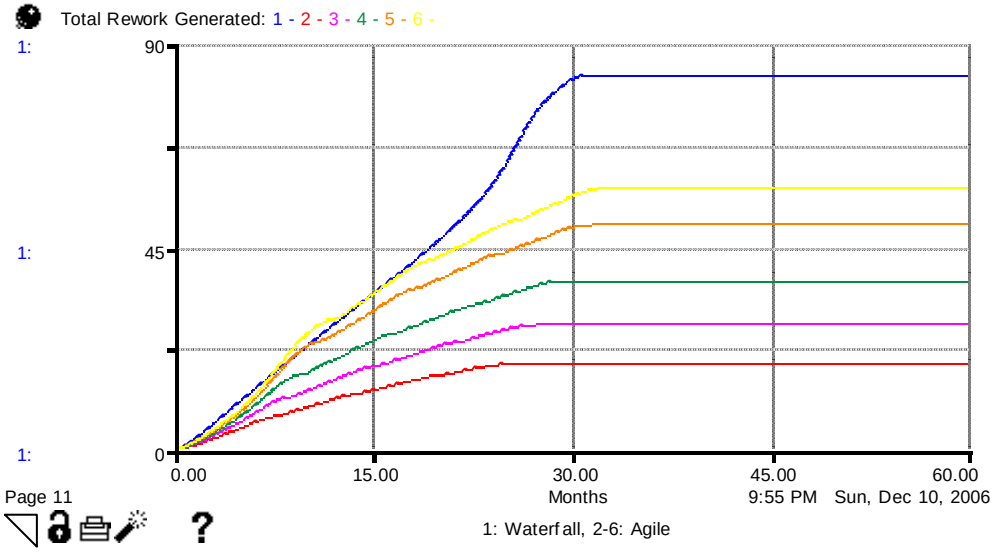




The higher Agile costs are easily explained by the productivity curves. Productivity in Agile is considerably below the waterfall case across the entire project.

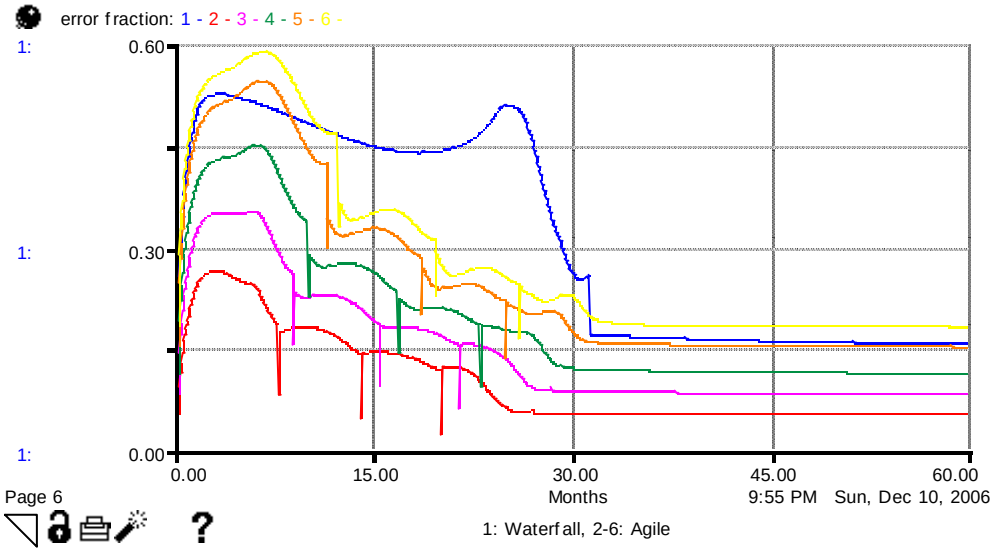


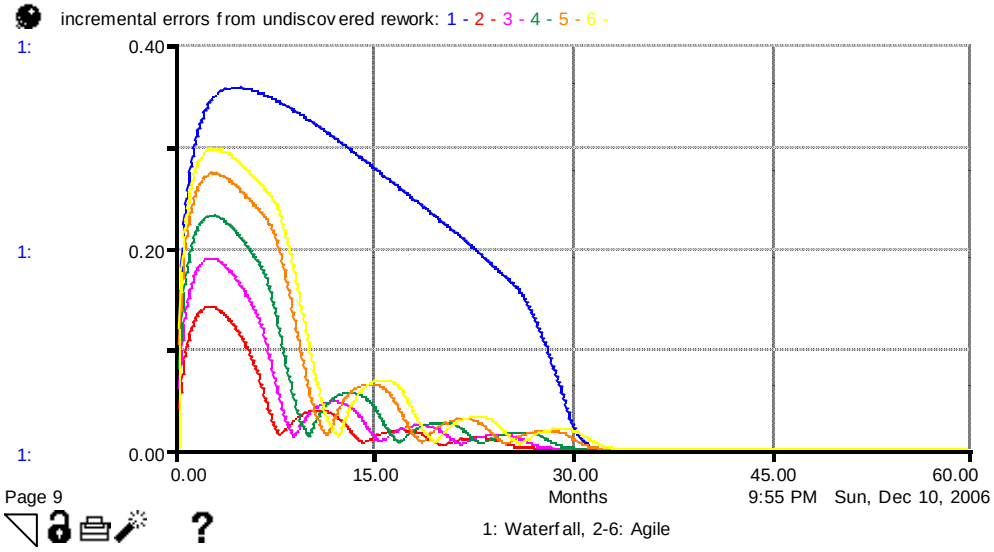
Finally, the total work accomplished is lower in all Agile cases because the rework generated is lower than in the waterfall case.

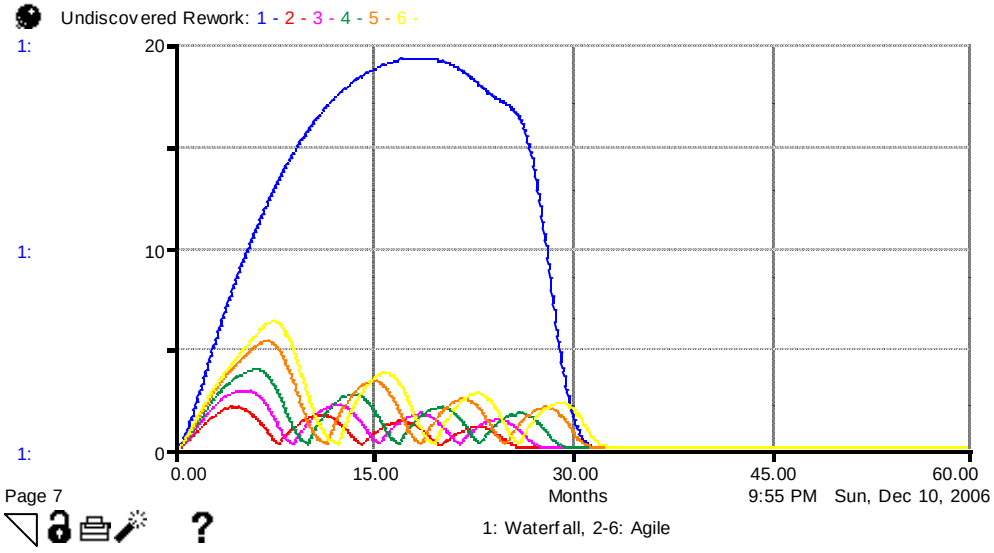


This is, of course, directly tied to lower error fractions in the Agile case across the entire project (shown below). Lower error fractions also mean a lower errors-on-errors effect. The bump in the first phase of each Agile case is caused by ramping up the staff at the beginning.

There is something of a paradox here, though. How can the error fraction remain lower than the waterfall case when the last two cases (five and six) set the Agile error fraction equal to and then greater than the waterfall case? This is because *incremental errors from undiscovered rework* stays smaller due to a slightly smaller effect in the Agile case, a shorter rework discovery delay, and the clearing out of *Undiscovered Rework* at the end of each phase.


error fraction: 1 - 2 - 3 - 4 - 5 - 6 -
1:
0.60
1:
0.30
1:
0.00
0.00
15.00
30.00
45.00
60.00
Months
Page 6
9:55 PM Sun, Dec 10, 2006
1: Waterfall, 2-6: Agile


incremental errors from undiscovered rework: 1 - 2 - 3 - 4 - 5 - 6 -
1:
0.40
1:
0.20
1:
0.00
0.00
15.00
30.00
45.00
60.00
Months
Page 9
9:55 PM Sun, Dec 10, 2006
1: Waterfall, 2-6: Agile


Undiscovered Rework: 1 - 2 - 3 - 4 - 5 - 6 -
1:
20
1:
10
1:
0
0.00
15.00
30.00
45.00
60.00
Months
Page 7
9:55 PM Sun, Dec 10, 2006
1: Waterfall, 2-6: Agile

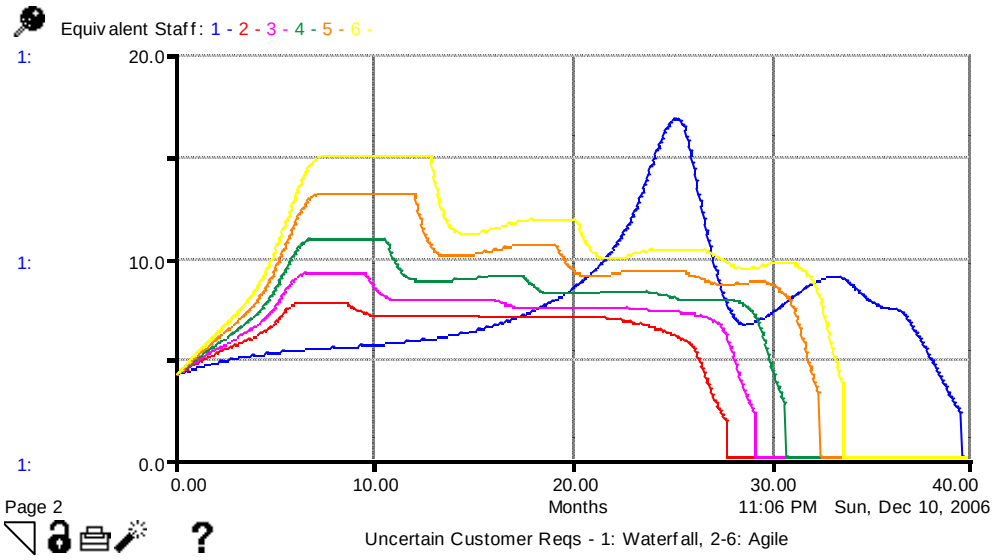
These results are summarized in the table below. Note that all of these tests were done with uncertain customer requirements turned off. It is clear from the table that Agile roughly breaks even with waterfall (4% earlier at 9% higher cost) when Agile has a slightly better error fraction (and much worse productivity). When the error fraction reaches parity with waterfall, the project takes longer and costs quite a bit more. Further degradation leads to much higher costs. As stated earlier, these last two cases are more likely when the methodology is first adopted, giving the classic worse-before-better behavior.

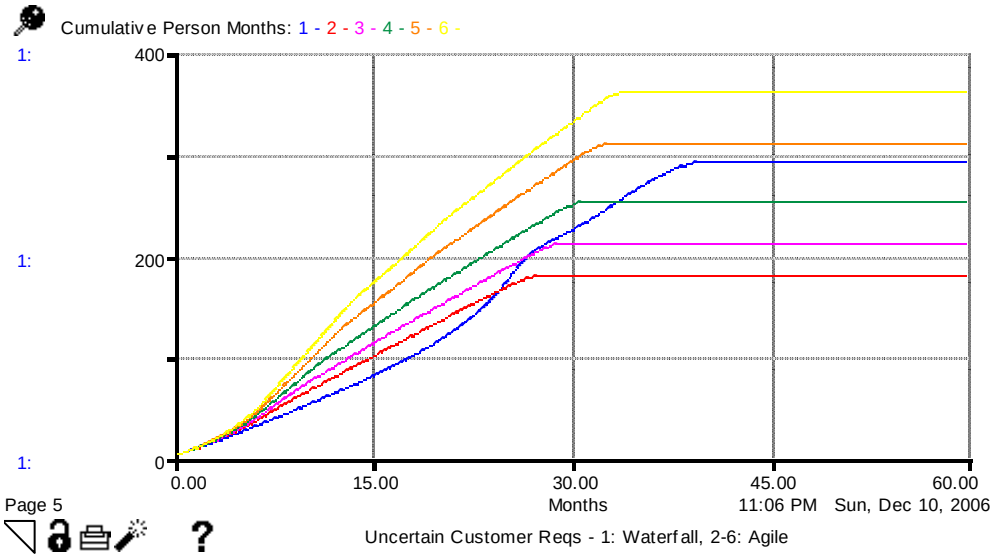
There is no question that Agile will normally give *some* improvement in error fraction over waterfall and it is hard to believe that productivity will be much worse than the 75% given in that middle case (4). Therefore, conservatively, under normal circumstances with an inconsistent mission, we expect Agile to be no later than waterfall, with about a 10% increase in cost. If we are able to improve productivity somewhat while lowering error fraction even more (case 3), Agile can beat the waterfall schedule with no additional cost. This is, of course, assuming the maximum rework discovery delay has been reduced. Although changes to this parameter have a relatively small impact on the Agile case, the small difference is enough to tip the balance back to waterfall.

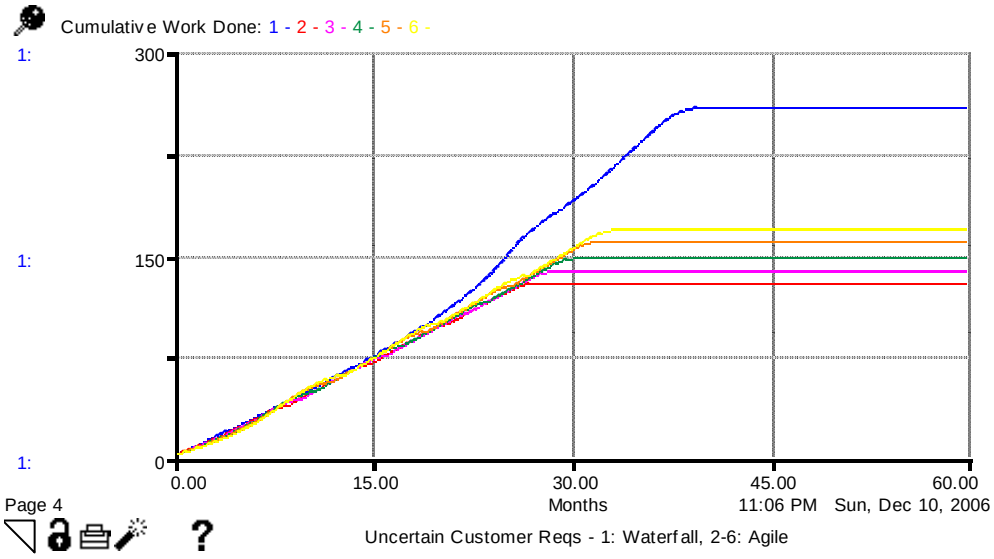
Case number	normal error fraction	normal productivity	incr. errors: new staff	productivity: new staff	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
	Waterfall:									
1	0.15	1.00	0.50	0.50	31.25		212.84		183.04	
	Agile:									
2	0.05	0.85	0.35	0.65	26.9375	-13.80	163.82	-23.03	118.68	-35.16
3	0.08	0.80	0.40	0.60	28.375	-9.20	193.49	-9.09	127.40	-30.40
4	0.11	0.75	0.45	0.55	29.9375	-4.20	231.85	8.93	137.10	-25.10
5	0.15	0.70	0.50	0.50	31.8125	1.80	286.12	34.43	149.68	-18.23
6	0.18	0.65	0.50	0.50	32.875	5.20	333.64	56.76	157.95	-13.71

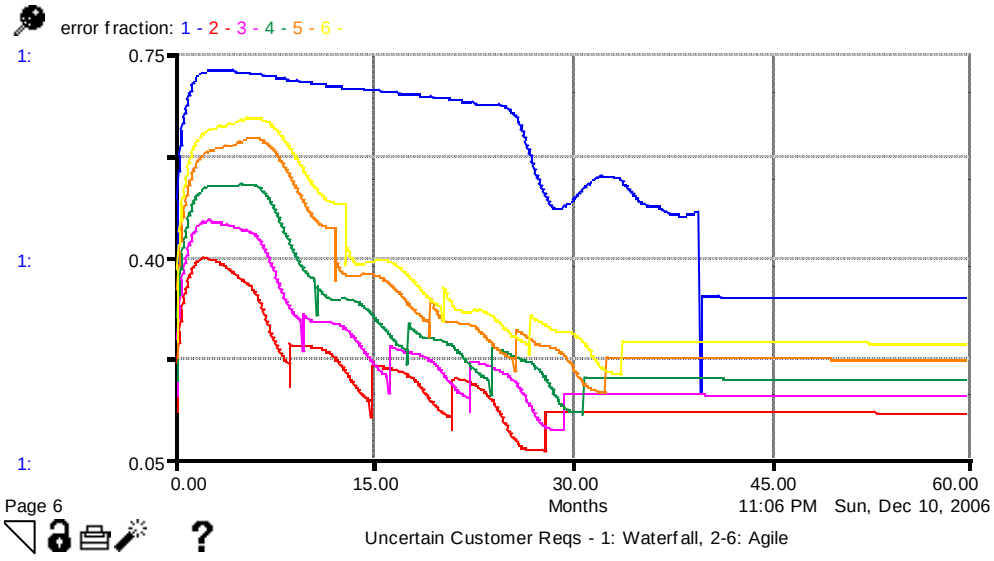
With uncertain customer requirements, Agile wins out in everything except cost in the last two cases (shown below – graphs follow). This is, of course, the case that Agile was born to handle. Note that even in case 5, when the error fraction is the *same* as in the waterfall case (but the productivity is *much* lower), Agile finishes almost 20% ahead of the waterfall case with only a 6% increase in project cost. Also take a moment to compare the project lengths with those above. Observe that uncertain customer requirements pushed every Agile case back by less than a month. Compare this to the 8.5 month difference in the waterfall case. Clearly, this is where Agile shines.

Case number	normal error fraction	normal productivity	incr. errors: new staff	productivity: new staff	Project Length (mo)	Delta from Base (%)	Project Cost (person-mo)	Delta from Base (%)	Work Completed (tasks)	Delta from Base (%)
	Waterfall:									
1	0.15	1.00	0.50	0.50	39.75		290.98		258.11	
	Agile:									
2	0.05	0.85	0.35	0.65	27.875	-29.87	178.58	-38.63	127.93	-50.44
3	0.08	0.80	0.40	0.60	29.3125	-26.26	210.22	-27.75	136.79	-47.00
4	0.11	0.75	0.45	0.55	30.875	-22.33	251.28	-13.64	146.55	-43.22
5	0.15	0.70	0.50	0.50	32.5625	-18.08	308.14	5.90	159.10	-38.36
6	0.18	0.65	0.50	0.50	33.75	-15.09	359.85	23.67	167.53	-35.09

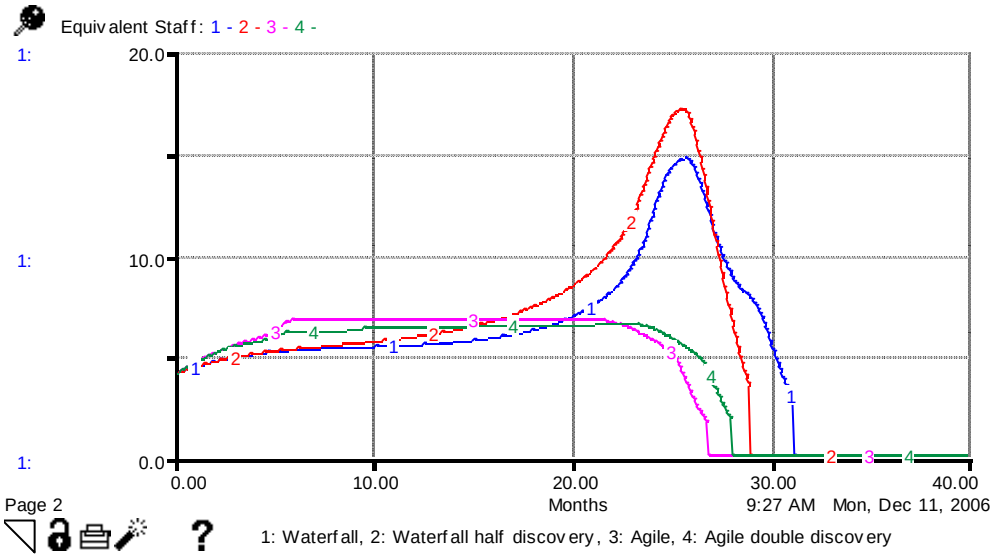




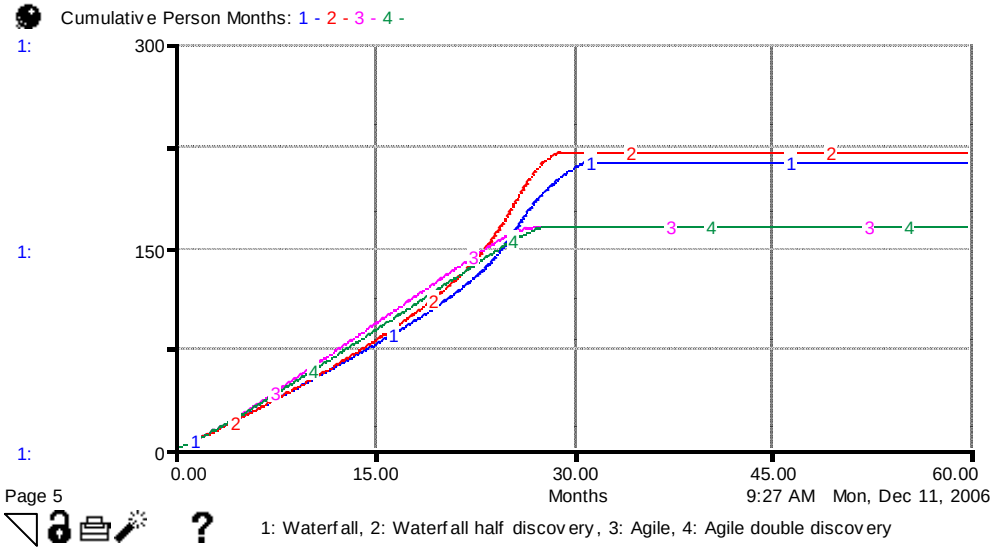




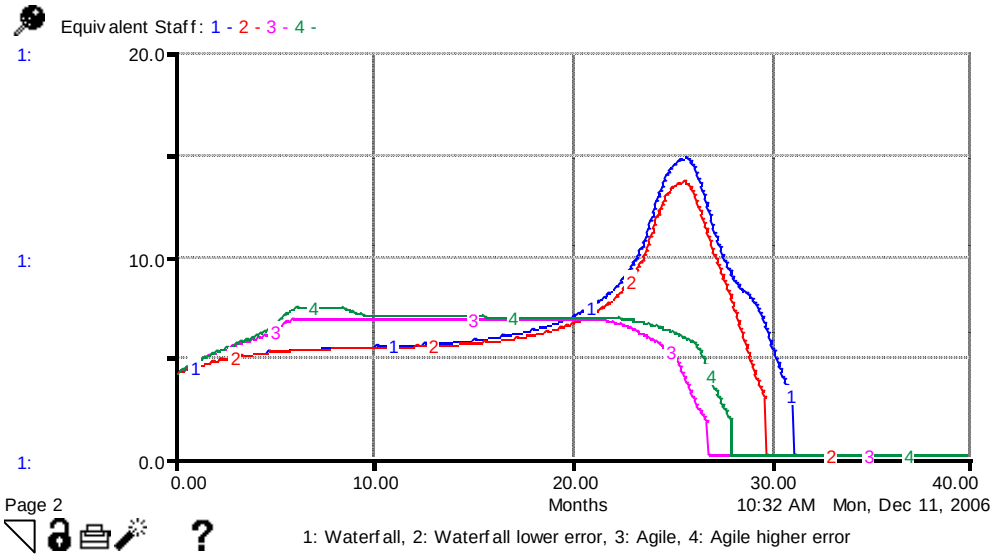
These results lead to an interesting question when there is an inconsistent schedule, but there are not uncertain customer requirements: If we can cut the rework discovery delay in half in the waterfall case, and also double the rework discovery delay in Agile (so they are actually the same), will the waterfall model give similar results? Doing so, leads to the following behavior:

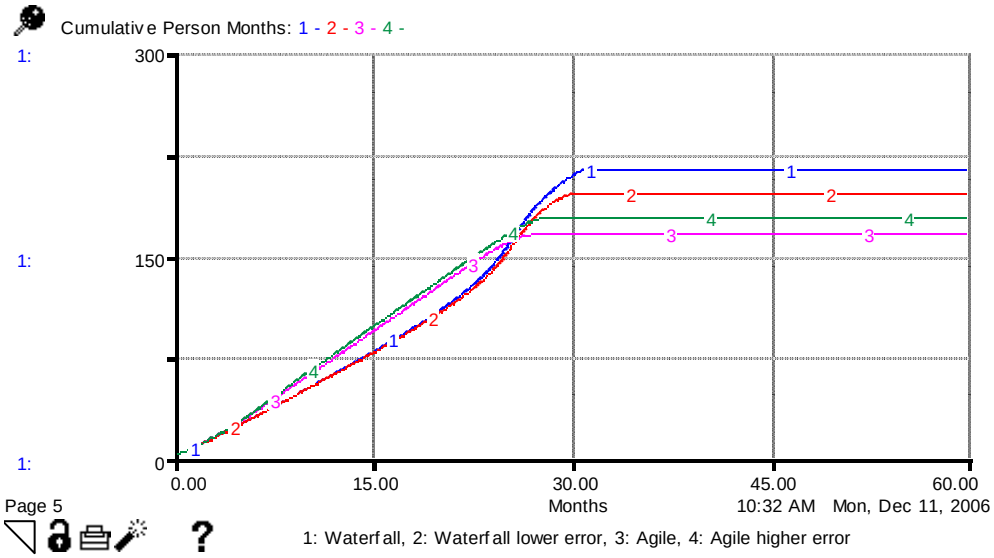


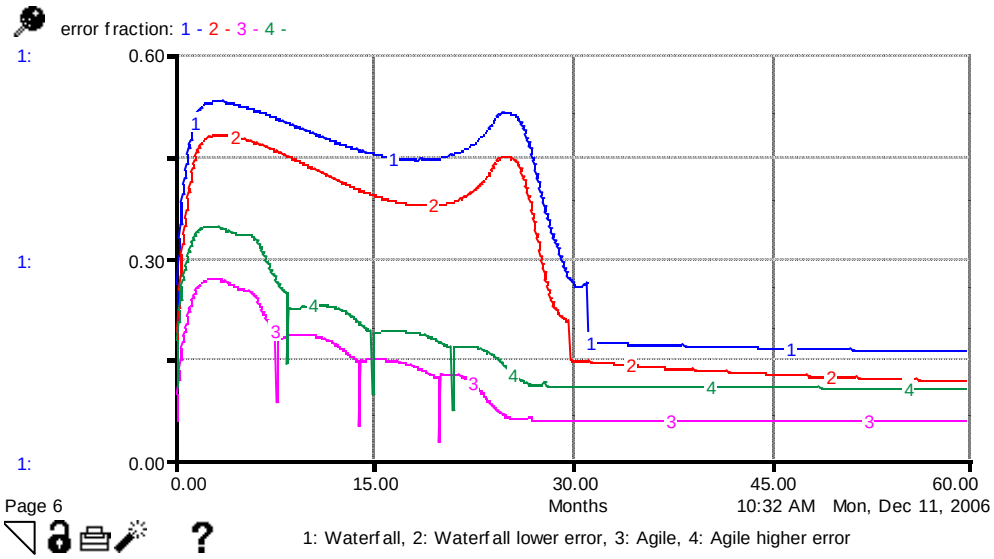
The original cases are shown in curves 1 and 3, while the curves with the same rework discovery delays are shown on curves 2 and 4 (red and green). Note there is much less of an advantage to Agile (about a month) which could be lost to other factors, such as a slightly higher error fraction or a slightly lower productivity. Agile's costs, though, remain lower (25% lower) because there is still less rework due to lower error fractions and removing (almost) all rework before moving to the next phase.

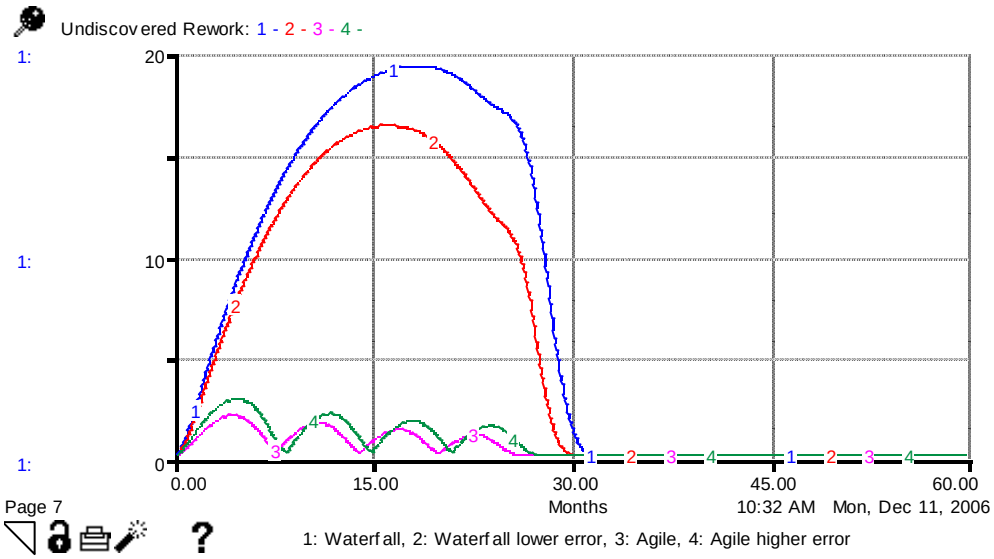


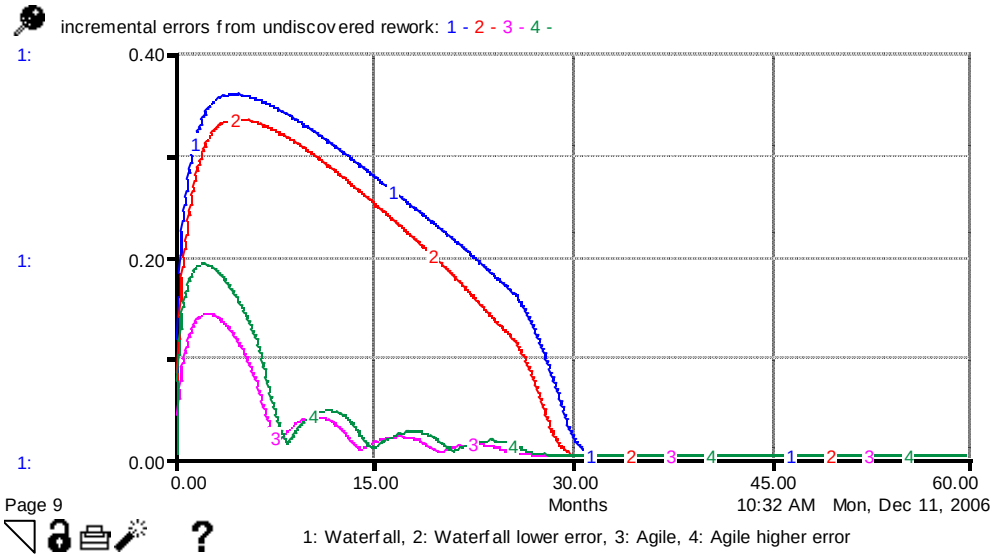
Finally, under these same circumstances, does Agile still have an advantage if the error fraction is increased while the waterfall error fraction is decreased, so they are equal (0.1)? The results clearly show that it still does have this advantage, finishing 2 months earlier with a 10% cost saving. This advantage is caused by a lower error fraction due to consistently lower levels of *Undiscovered Rework*. These consistently lower levels are, as mentioned earlier, due to both a short rework discovery delay and the fact that the *Undiscovered Rework* is not allowed to grow across the length of the project (it is cleared out each phase).


Equivalent Staff: 1 - 2 - 3 - 4 -
1:
20.0
10.0
0.0
0.00
10.00
20.00
30.00
40.00
Months
Page 2
10:32 AM Mon, Dec 11, 2006
1: Waterfall, 2: Waterfall lower error, 3: Agile, 4: Agile higher error


Cumulative Person Months: 1 - 2 - 3 - 4 -
1:
300
150
0
0.00
15.00
30.00
45.00
60.00
Months
Page 5
10:32 AM Mon, Dec 11, 2006
1: Waterfall, 2: Waterfall lower error, 3: Agile, 4: Agile higher error


error fraction: 1 - 2 - 3 - 4 -
1:
0.60
0.30
0.00
0.00
15.00
30.00
45.00
60.00
Months
Page 6
10:32 AM Mon, Dec 11, 2006
1: Waterfall, 2: Waterfall lower error, 3: Agile, 4: Agile higher error


Undiscovered Rework: 1 - 2 - 3 - 4 -
1:
20
1:
10
1:
0
0.00
15.00
30.00
45.00
60.00
Months
Page 7
10:32 AM Mon, Dec 11, 2006
1: Waterfall, 2: Waterfall lower error, 3: Agile, 4: Agile higher error

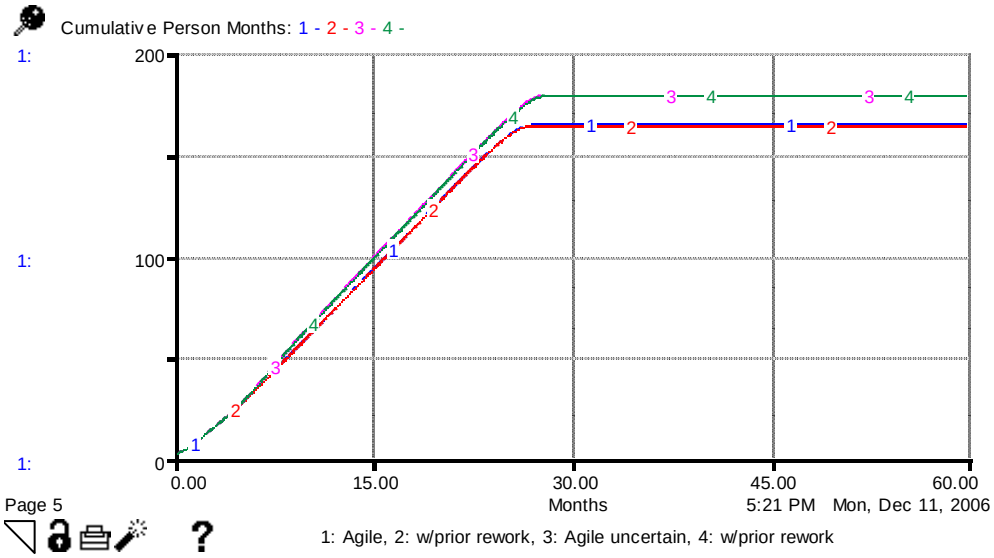

incremental errors from undiscovered rework: 1 - 2 - 3 - 4 -
1:
0.40
1:
0.20
1:
0.00
0.00
15.00
30.00
45.00
60.00
Months
Page 9
10:32 AM Mon, Dec 11, 2006
1: Waterfall, 2: Waterfall lower error, 3: Agile, 4: Agile higher error

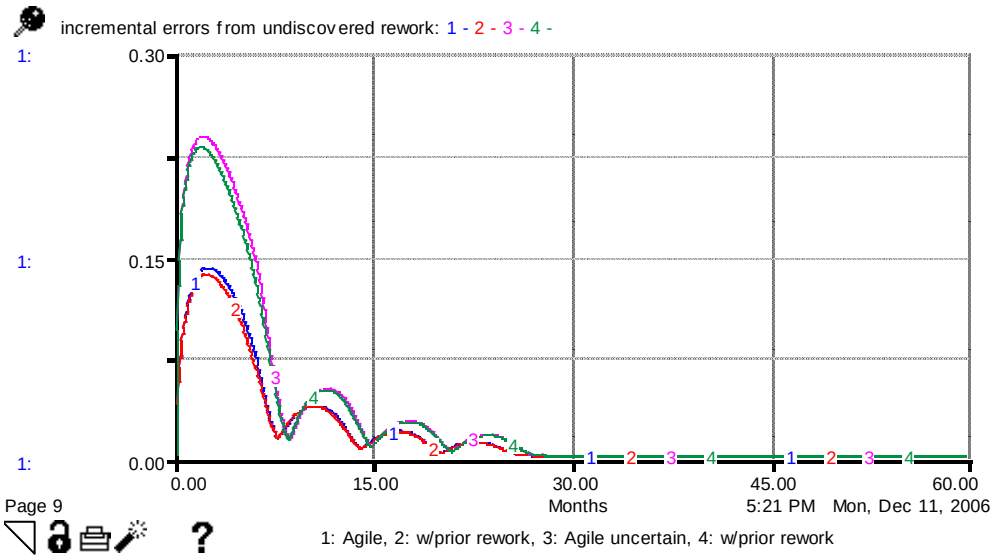
Additional Performance Issues

There remain two questions regarding the performance of Agile:

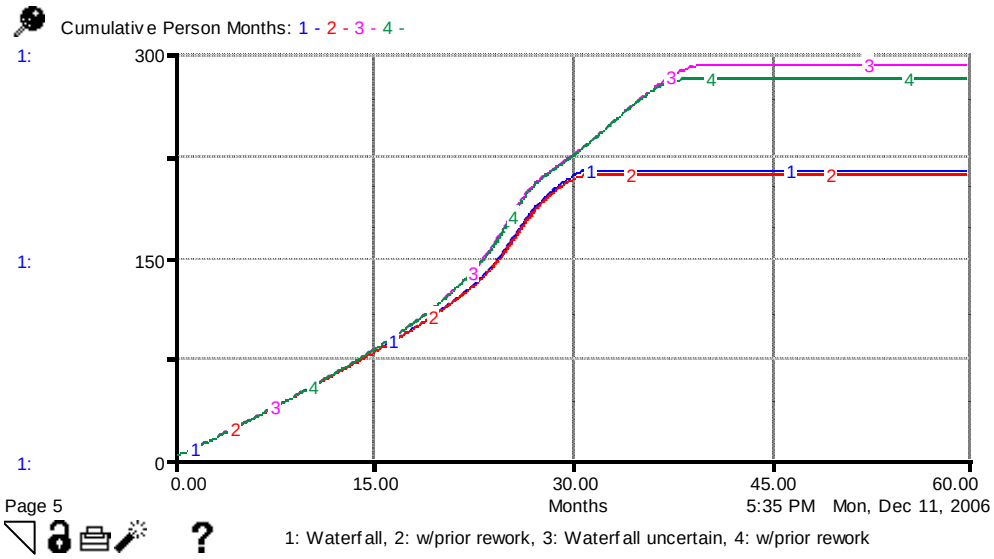
- Can the performance be improved by prioritizing rework over original work?
- Will performance get worse if one phase's rework isn't completed before moving onto the next phase (something that we saw was disastrous for the waterfall model)?

Surprisingly, prioritizing rework over original work makes little difference to the Agile case as the rework is already being done relatively quickly (very close to as you go). The four cases [certain customer requirements: base (1) and prioritized (2), uncertain customer requirements: base (3) and prioritized (3)] are shown below. Note that there is no noticeable difference in the project length or cost in any of the cases, though the incremental errors from rework are slightly higher in the non-prioritized cases.

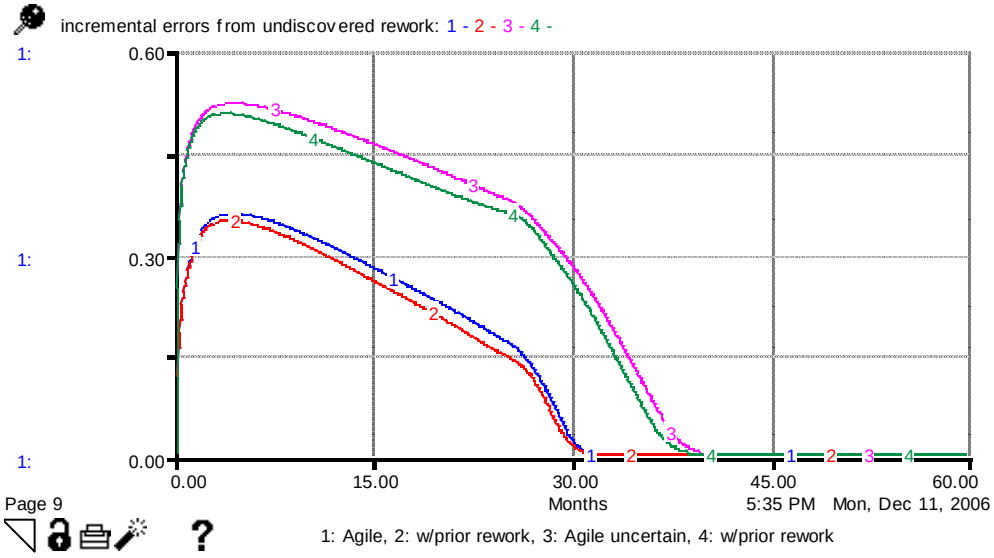




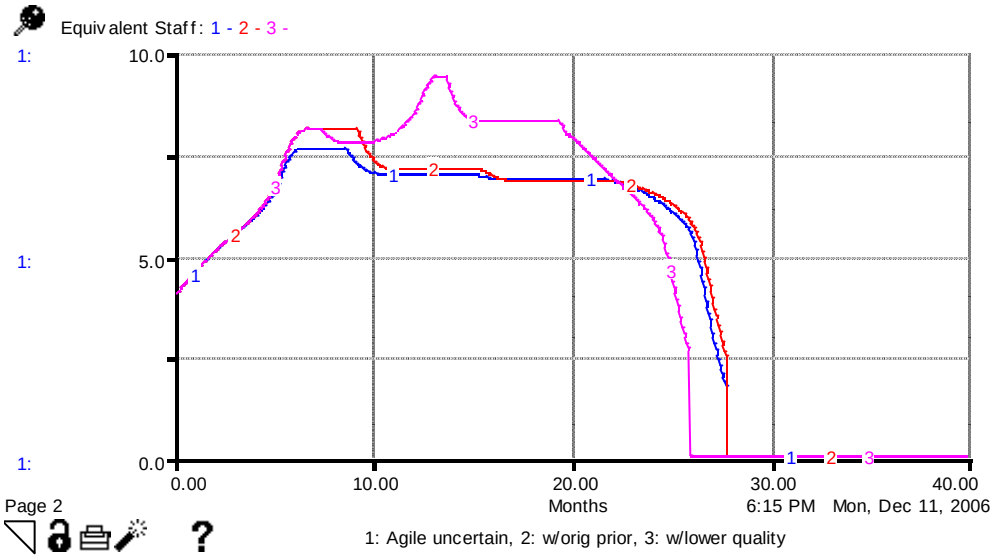
Note that there is a slight advantage to doing this in the waterfall case, bringing the project in one month sooner at a lower cost with uncertain customer requirements.



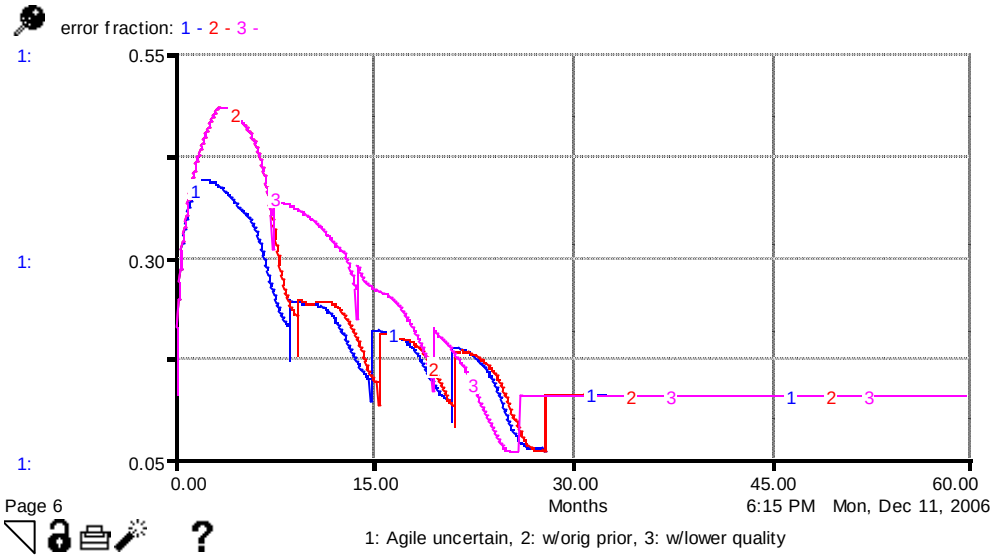
In this case, the incremental errors effect is also noticeably lower when fixing rework has top priority.

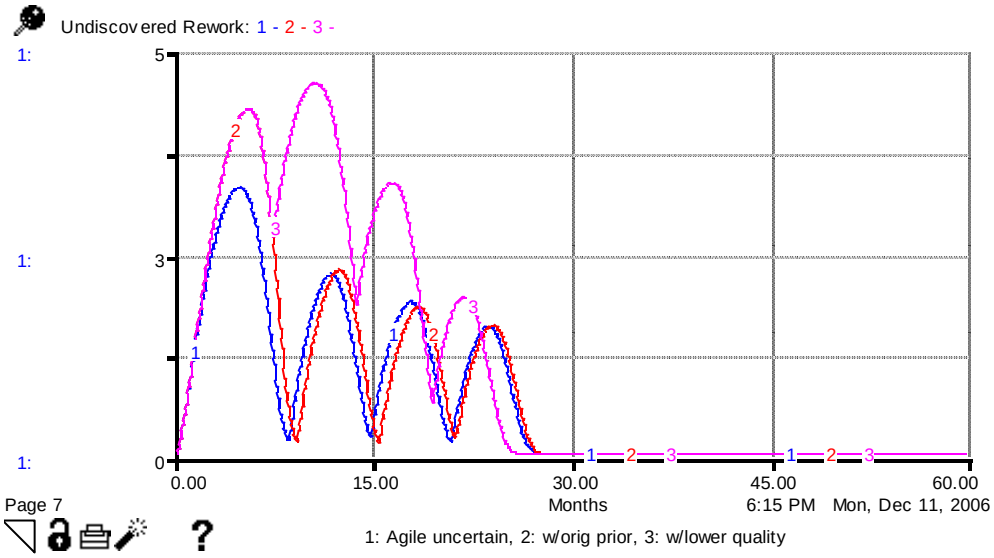


With regards to the second question, there is no change in performance if the release quality threshold is lowered because original work is constraining the end of the phase in the base case. However, if the model is changed to give priority to original work and the quality threshold is lowered from 0.99 to 0.9 (i.e., the end of phase threshold for *Rework To Do* is raised from 0.01 of original work to 0.1 of original work), there are some very surprising results. The project actually finishes almost two months sooner! In the following graphs, the first curve (blue) is the base case with uncertain customer requirements, the second curve (red) is the same case with priority given to original work, and the final curve (magenta) lowers the quality threshold.



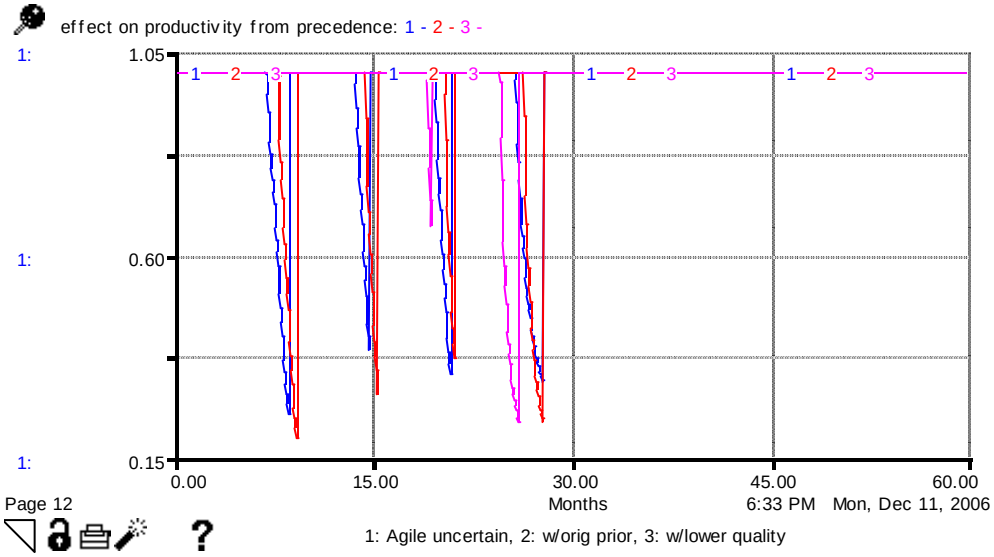
As expected, the error fraction is higher due to a additional backlog of *Undiscovered Rework* and *Rework to Do* (as well as that bump of new hires, which also brings productivity down some).

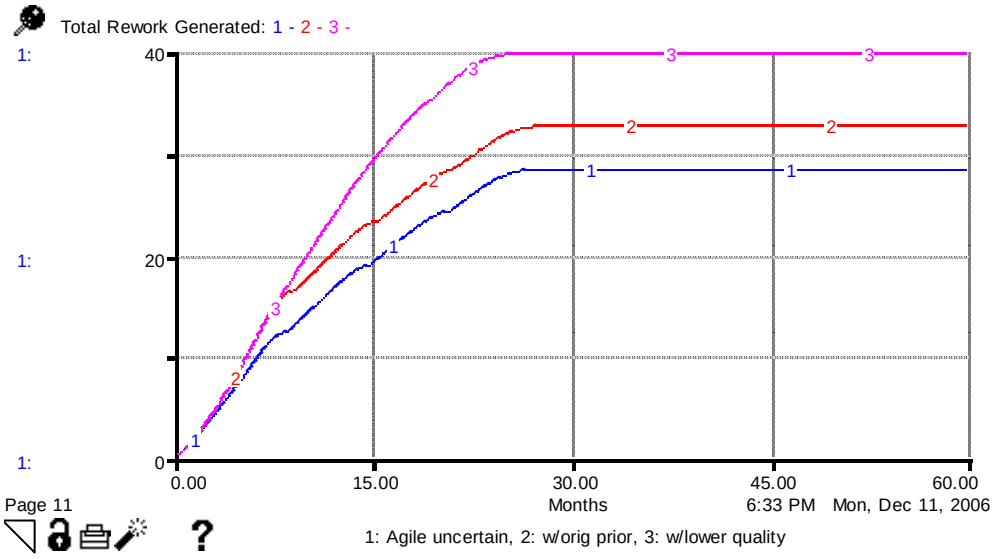


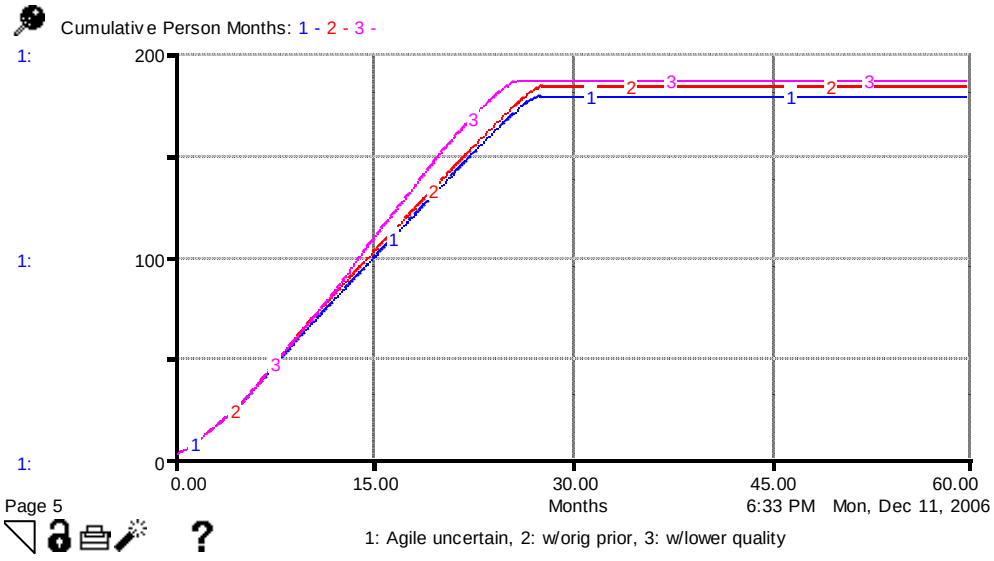


The explanation for the earlier finish comes from precedence effects on productivity. Because original work is being given priority and assigned at the highest possible level, there are no precedence effects on original work. Any precedence effects on rework are caused by idle time while waiting for more rework to be discovered. As shown in curve 2 below (original work priority), the initial high quality standard causes productivity delays at the end of each phase. However, in the third curve, the next phase is started before the rework in the current phase is completed (or even completely discovered). Therefore, the delays in discovering rework do not occur (and precedence constraints do not apply) until the end of the project when there is still the desire to deliver a final quality product. Note in this last case that the precedence effects on productivity stay at one until the very end of the project. The additional (and consistent) productivity is what causes the project to finish earlier than the base case.

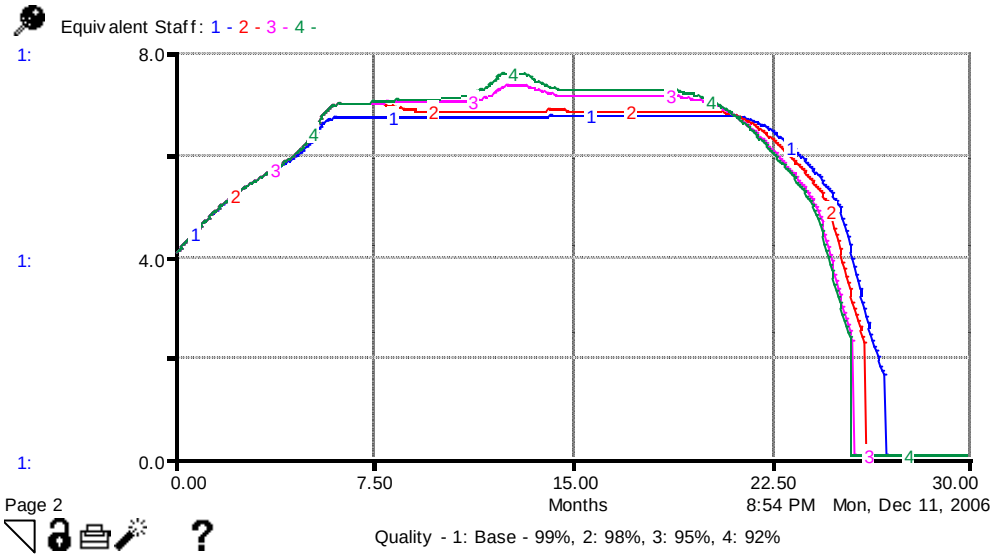
Note there is considerably more rework generated in this case, but the cost is only marginally higher (4% - due to the better productivity on average and the early finish).

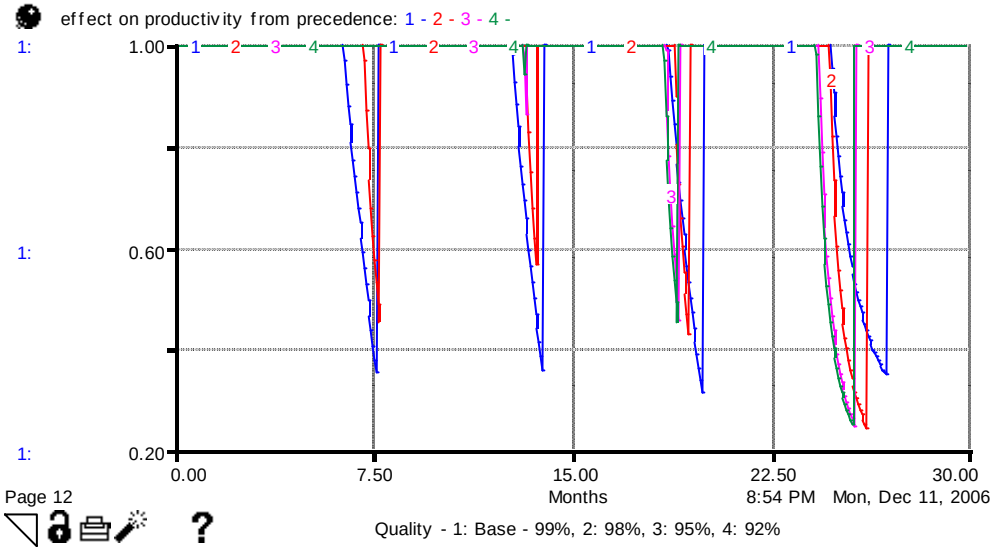




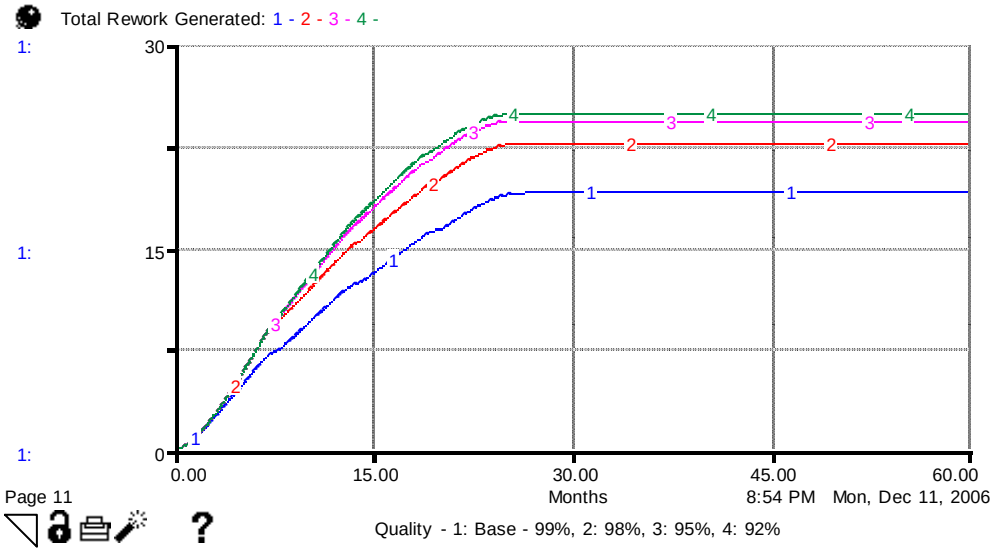


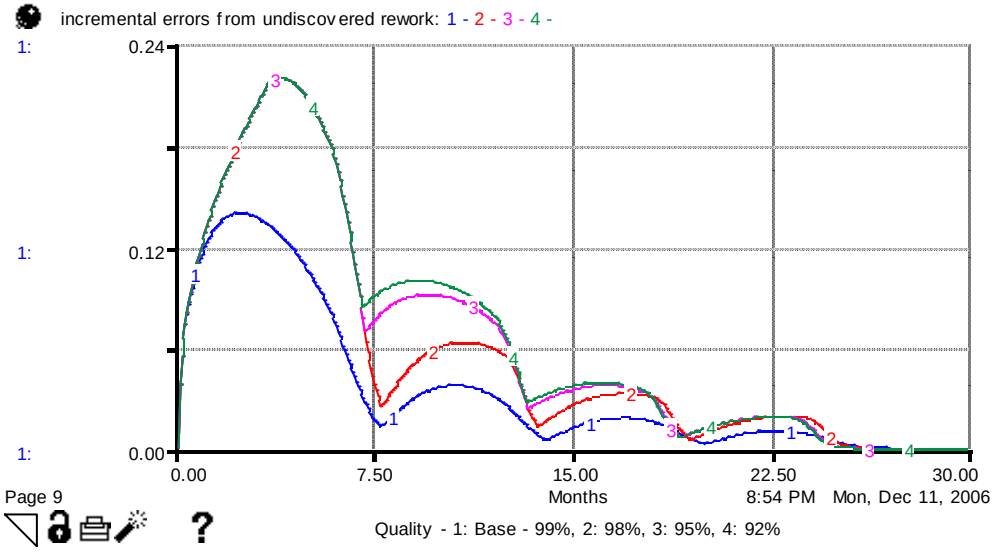
This model does not explore the corporate level issues of reducing quality this much. I do not believe the small time savings to be worth the risk at 90% quality. Sensitivity tests below, though, show the benefits saturate at the 92% level and a substantial benefit at the 98% level (about a month) with no change in cost. This small reduction in quality between phases is probably acceptable for many projects.



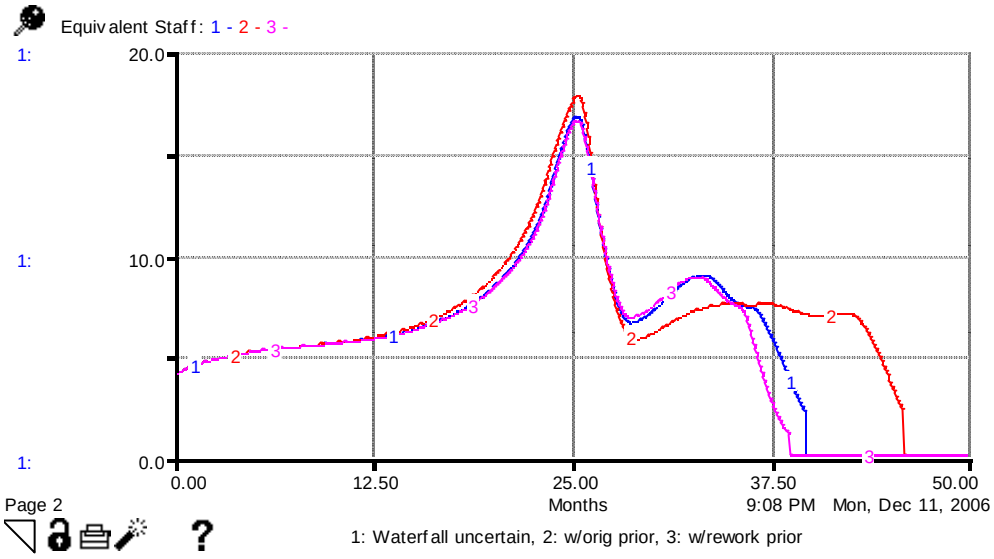


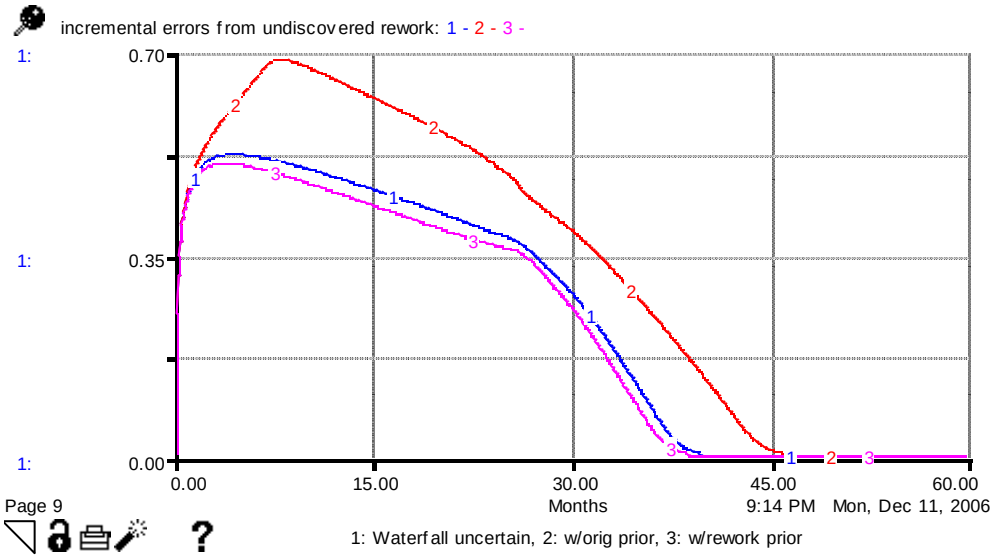
The graph of rework generated below gives an idea of the overall impact on errors-on-errors of the various quality levels. The incremental errors from rework are shown after that. Note that all levels of quality have the same errors-on-errors effect in the first phase. This is a consequence of the change to giving priority to original work. In subsequent phases, though, higher quality leads to lower incremental errors.





It is interesting to note that giving original work priority in the waterfall case is extremely deleterious to the project's completion (shown below – curve one is the waterfall base case with uncertain requirements, curve two gives priority to original work, and curve three gives priority to rework). Note also that giving priority to rework does not help as much as might have been anticipated (the no priority algorithm in place tends to already favor rework a bit).


Equivalent Staff: 1 - 2 - 3 -
1:
20.0
10.0
0.0
0.00
12.50
25.00
37.50
50.00
Months
Page 2
9:08 PM Mon, Dec 11, 2006
1: Waterfall uncertain, 2: w/orig prior, 3: w/rework prior


incremental errors from undiscovered rework: 1 - 2 - 3 -
1:
0.70
0.35
0.00
0.00
15.00
30.00
45.00
60.00
Months
Page 9
9:14 PM Mon, Dec 11, 2006
1: Waterfall uncertain, 2: w/orig prior, 3: w/rework prior

Summary

In summary, Agile methods do not appear to make a difference with a consistent mission. Both the finish time and the cost are approximately the same despite the lower normal productivity in Agile. This, of course, means that if there is a danger of not being able to control these parameters, or if there is reason to believe the Agile parameters are worse than used in this model, that managers should stick to the waterfall model for consistent missions. Additional support for this course of action can be found in the cost of switching (there isn't any benefit to switch and it will be worse before it is back to the same again, so why do it?).

When there is an inconsistent mission, Agile methods can win out over waterfall methods – again, assuming you have already adopted the methodology and fought your way through the worse-before-better startup transition. There are, however, cases where even this may be risky. As shown in the table in the middle of page 27, if your organization's error fraction and productivity under Agile fall just a little short of the values used in this model, Agile will begin to cost more. It also very quickly begins to cost a *lot* more.

Finally, when there is both an inconsistent mission and uncertain customer requirements, Agile is pretty much guaranteed to meet or beat the waterfall method, which was its main goal when it was developed. As shown in the table at the bottom of page 27, your organization's parameters under Agile have to be the same or worse than those for waterfall in order to turn out worse in either cost or project length. This is not likely except under the initial transition of switching to Agile.

In conclusion, the ideas behind Agile do indeed help projects come in earlier when there are changing customer requirements. The combination of frequent releases to, and interactions with, customers, nightly builds and automated tests, writing tests before code, and avoiding unnecessary complexity all work together to allow the project to adapt more easily to changing conditions.

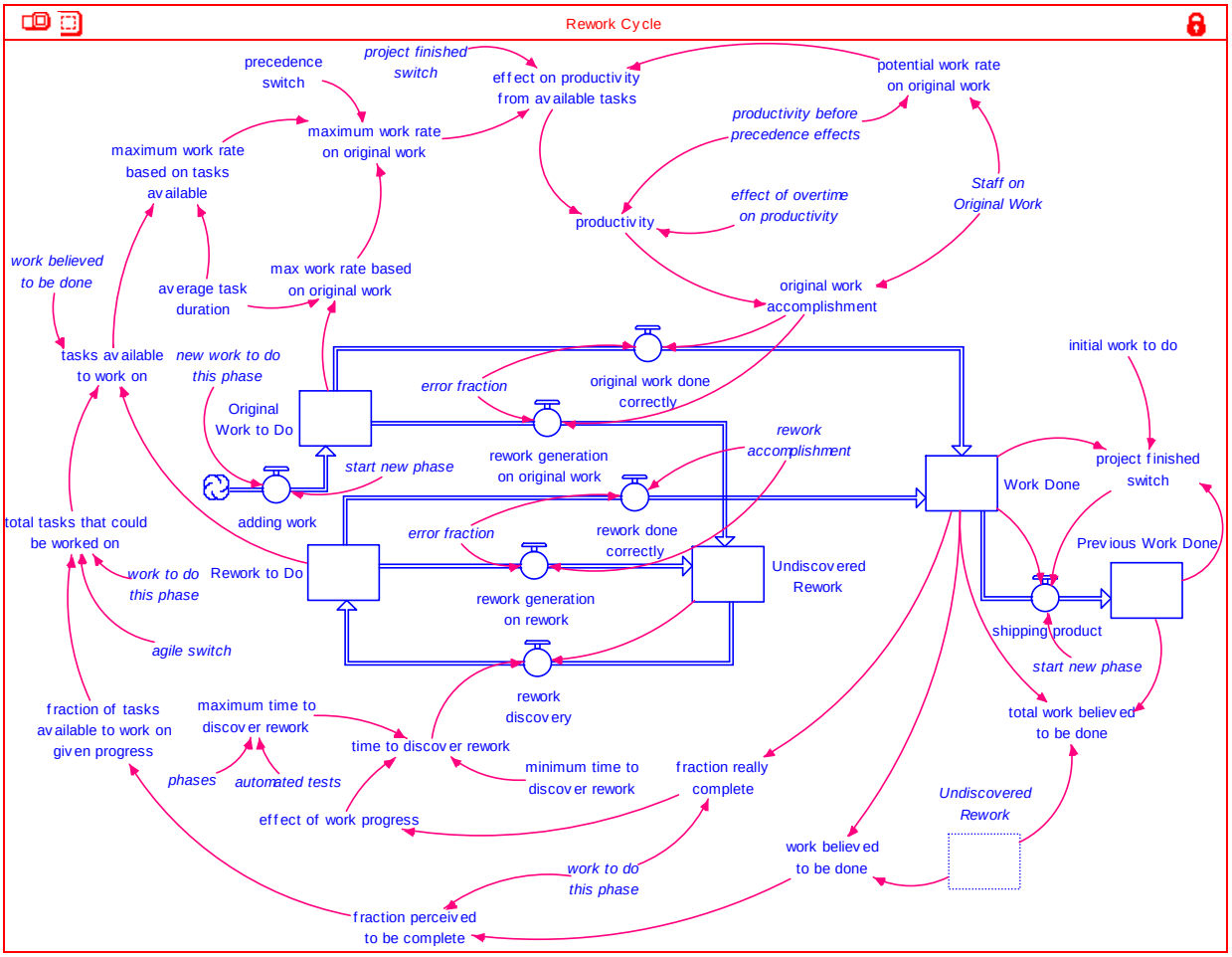
References

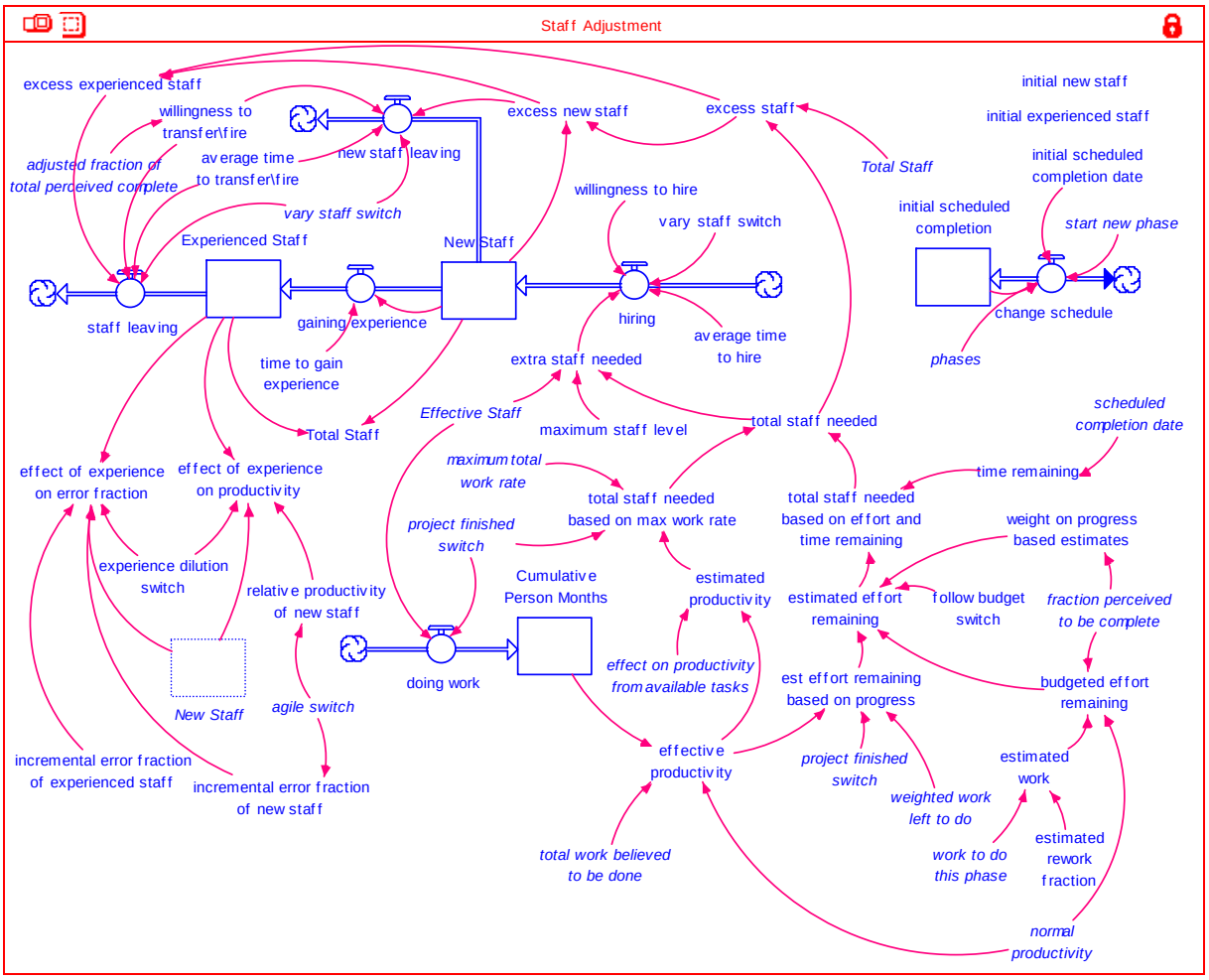
Beck, K, et. al. 2001. Manifesto for Agile Software Development, http://www.agilemanifesto.org

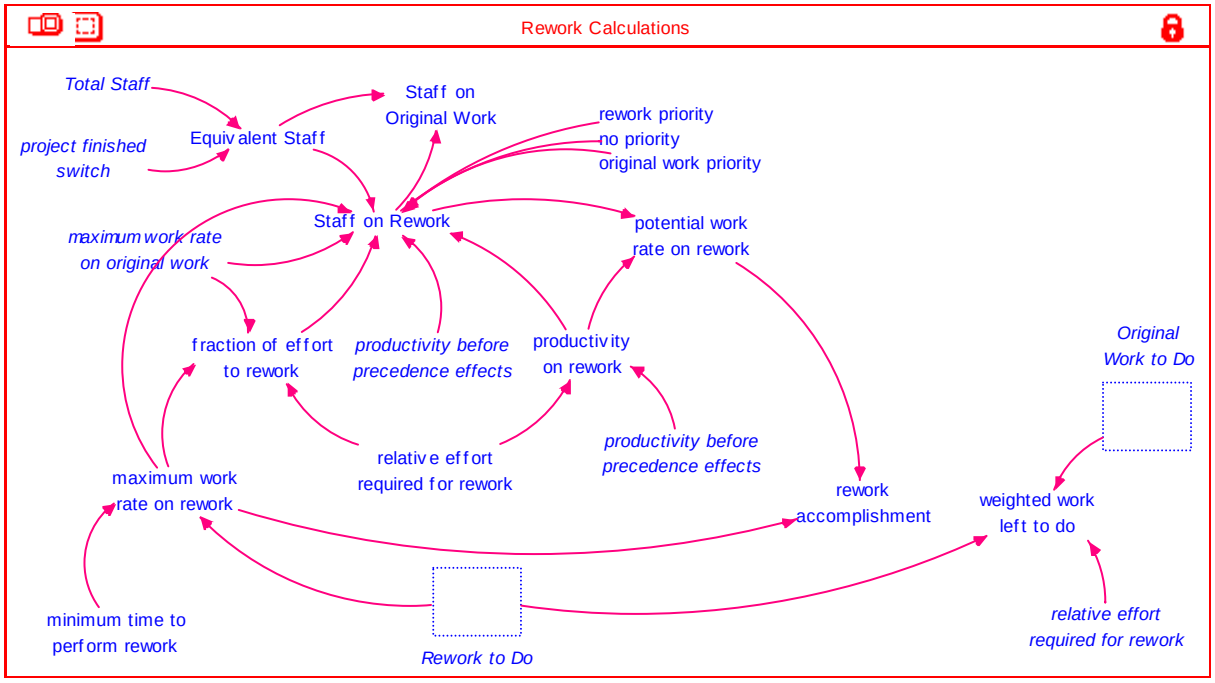
Cockburn, A. 2004. *Crystal Clear: A Human-Powered Methodology for Small Teams*. Addison-Wesley: Boston, MA.

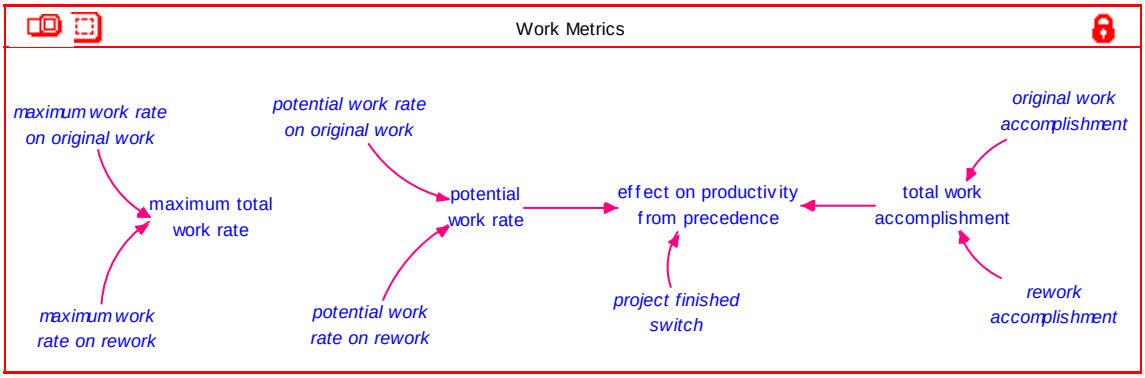
Lyneis JM. 2006. Project Dynamics. Lecture notes, Worcester Polytechnic Institute: Worcester, MA

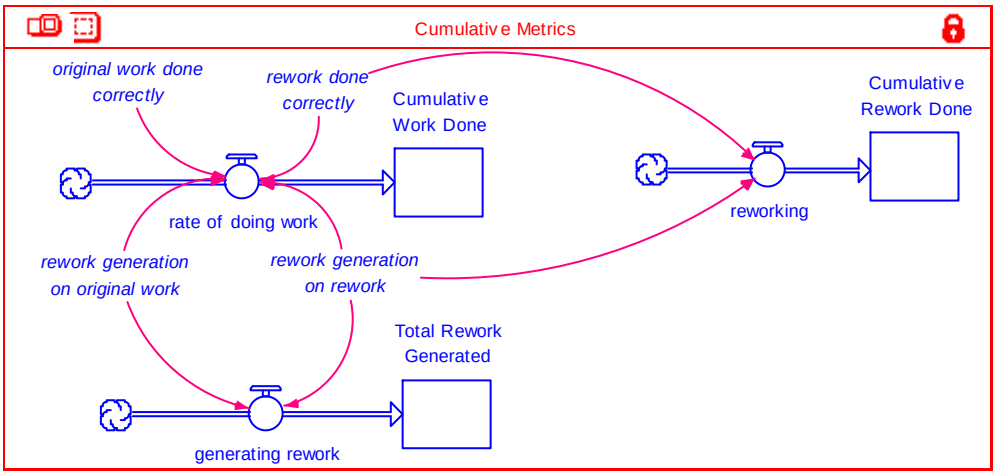
Appendix: Model Structure

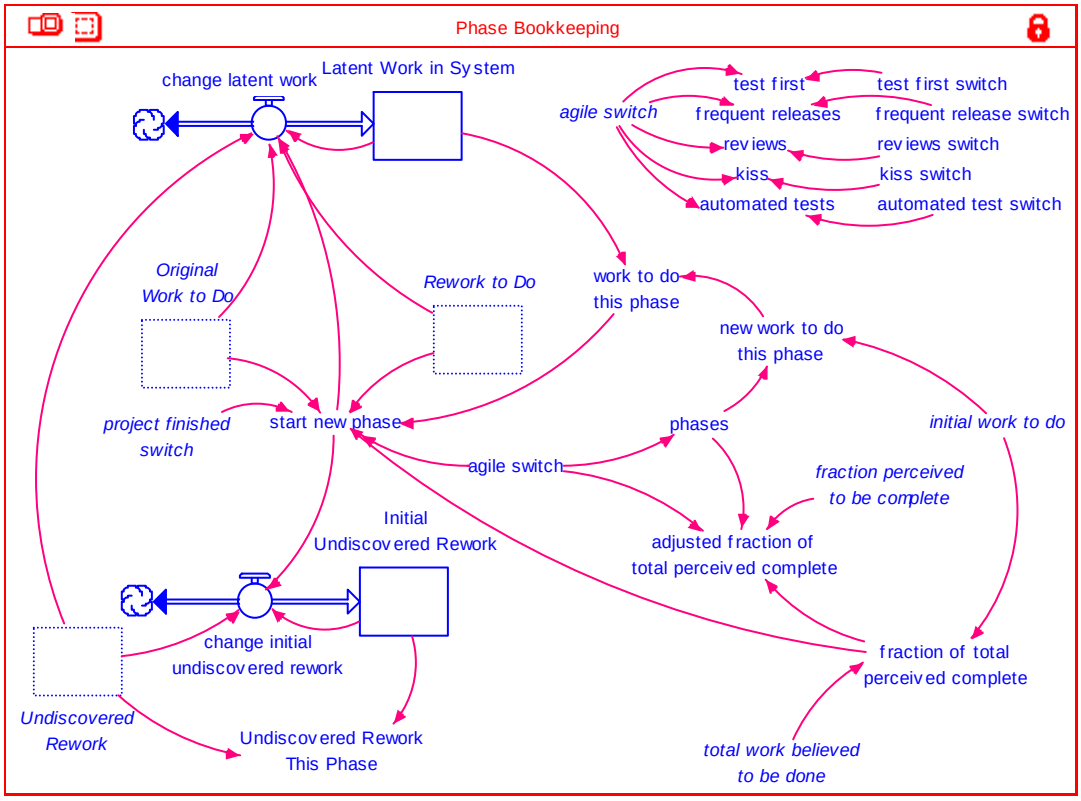


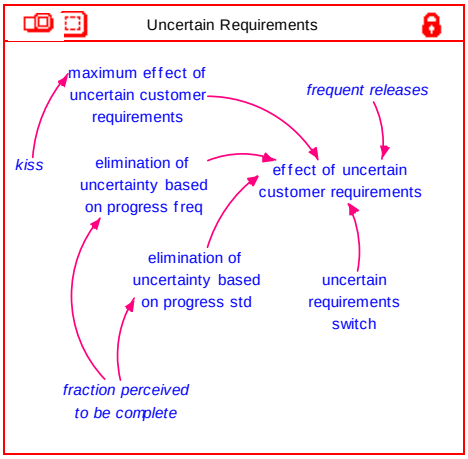

Staff Adjustment
excess experienced staff
willingness to transfer\fire
adjusted fraction of total perceived complete
average time to transfer\fire
new staff leaving
excess new staff
excess staff
Total Staff
initial new staff
initial experienced staff
initial scheduled completion date
initial scheduled completion
start new phase
change schedule
phases
vary staff switch
Experienced Staff
New Staff
willingness to hire
vary staff switch
staff leaving
gaining experience
hiring
average time to hire
time to gain experience
extra staff needed
Total Staff
Effective Staff
maximum staff level
total staff needed
scheduled completion date
time remaining
effect of experience on error fraction
effect of experience on productivity
maximum total work rate
total staff needed based on max work rate
total staff needed based on effort and time remaining
weight on progress based estimates
project finished switch
experience dilution switch
relative productivity of new staff
Cumulative Person Months
estimated productivity
estimated effort remaining
follow budget switch
fraction perceived to be complete
New Staff
agile switch
doing work
effect on productivity from available tasks
est effort remaining based on progress
budgeted effort remaining
incremental error fraction of experienced staff
incremental error fraction of new staff
effective productivity
project finished switch
estimated work
weighted work left to do
work to do this phase
estimated rework fraction
total work believed to be done
normal productivity

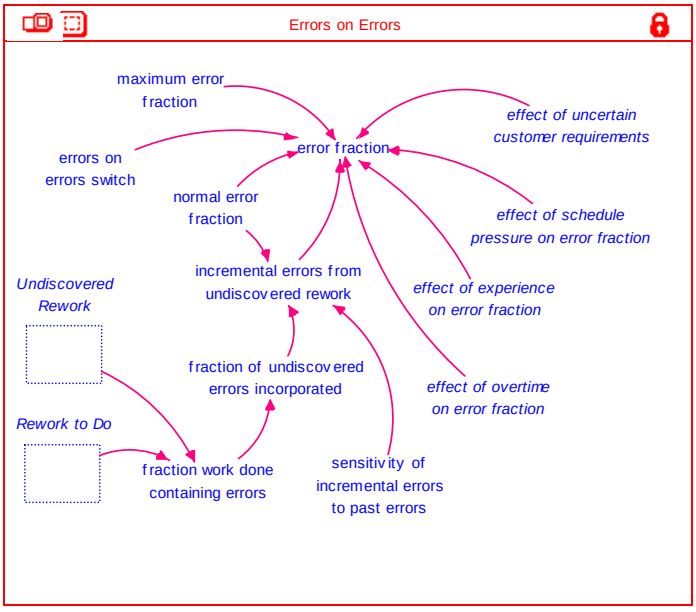


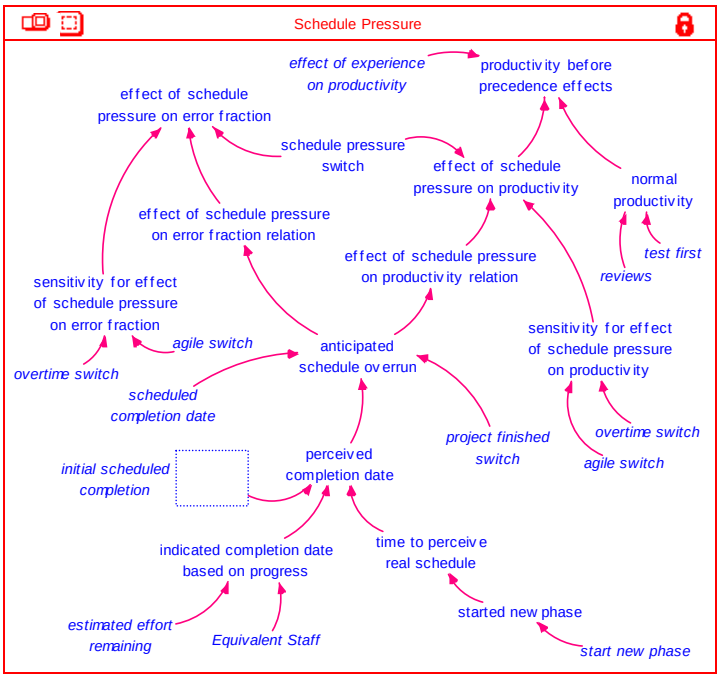


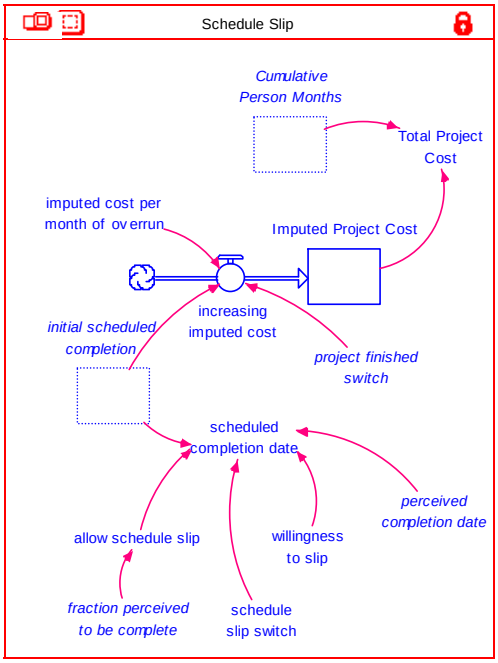


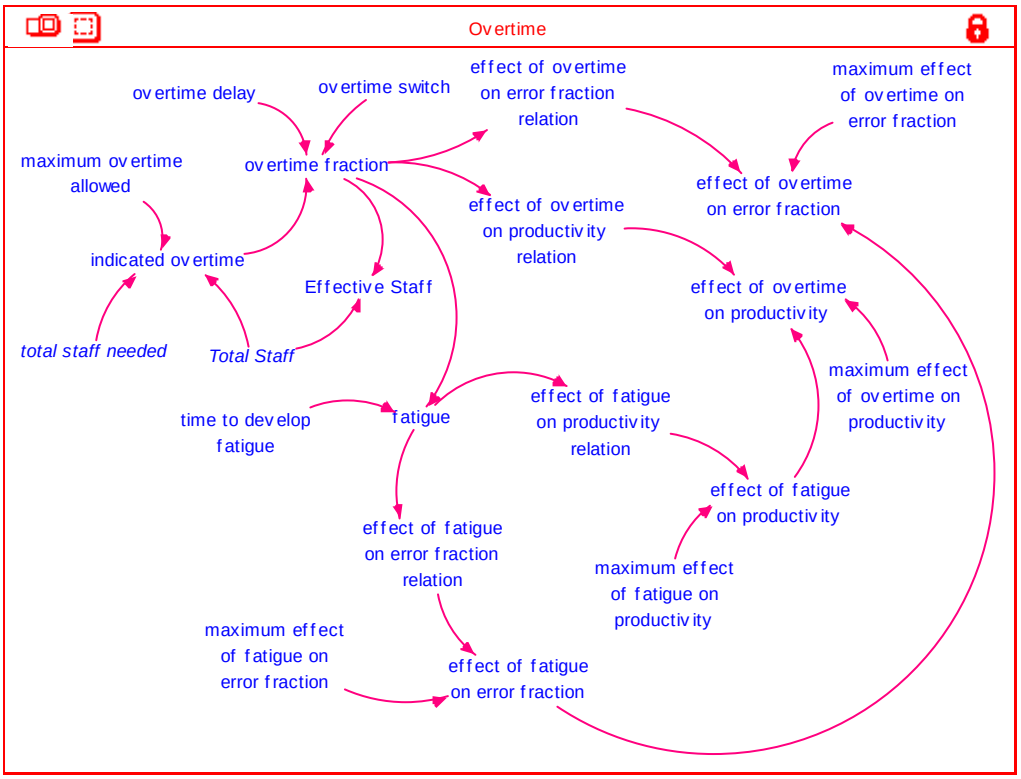

Phase Bookkeeping
change latent work
Latent Work in System
agile switch
test first
frequent releases
reviews
kiss
automated tests
test first switch
frequent release switch
reviews switch
kiss switch
automated test switch
Original Work to Do
Rework to Do
work to do this phase
new work to do this phase
project finished switch
start new phase
phases
initial work to do
agile switch
fraction perceived to be complete
Initial Undiscovered Rework
adjusted fraction of total perceived complete
change initial undiscovered rework
fraction of total perceived complete
Undiscovered Rework
Undiscovered Rework This Phase
total work believed to be done


Uncertain Requirements
maximum effect of uncertain customer requirements
frequent releases
kiss
elimination of uncertainty based on progress freq
effect of uncertain customer requirements
elimination of uncertainty based on progress std
uncertain requirements switch
fraction perceived to be complete


Errors on Errors
maximum error fraction
effect of uncertain customer requirements
error fraction
errors on errors switch
normal error fraction
effect of schedule pressure on error fraction
incremental errors from undiscovered rework
Undiscovered Rework
effect of experience on error fraction
fraction of undiscovered errors incorporated
effect of overtime on error fraction
Rework to Do
fraction work done containing errors
sensitivity of incremental errors to past errors







Cumulative Metrics
Cumulative_Rework_Done(t) = Cumulative_Rework_Done(t - dt) + (reworking) * dt
INIT Cumulative_Rework_Done = 0 {tasks}
INFLOWS:
reworking = rework_done_correctly + rework_generation_on_rework {tasks/mo}

Cumulative_Work_Done(t) = Cumulative_Work_Done(t - dt) + (rate_of_doing_work) * dt
INIT Cumulative_Work_Done = 0 {tasks}
INFLOWS:
rate_of_doing_work = original_work_done_correctly + rework_generation_on_original_work +
    rework_done_correctly + rework_generation_on_rework {tasks/mo}

Total_Rework_Generated(t) = Total_Rework_Generated(t - dt) + (generating_rework) * dt
INIT Total_Rework_Generated = 0 {tasks}
INFLOWS:
generating_rework = rework_generation_on_original_work + rework_generation_on_rework {tasks/mo}

Errors on Errors

error_fraction = maximum_error_fraction - ((maximum_error_fraction - normal_error_fraction)*(1 -
    incremental_errors_from_undiscovered_rework*errors_on_errors_switch)*(1 -
    effect_of_schedule_pressure_on_error_fraction)*(1 - effect_of_experience_on_error_fraction)*(1 -
    effect_of_uncertain_customer_requirements)*(1 - effect_of_overtime_on_error_fraction)) {dimensionless}
errors_on_errors_switch = 1{dimensionless}
    DOCUMENT: Set to 1 to enable the errors on errors feedback loop. Otherwise, set to zero.
fraction_work_done_containing_errors = IF (total_work_believed_to_be_done = 0 OR project_finished_switch)
    THEN 0
    ELSE (Rework_to_Do + Undiscovered_Rework)/total_work_believed_to_be_done {dimensionless}
incremental_errors_from_undiscovered_rework = (1 -
    normal_error_fraction)*fraction_of_undiscovered_errors_incorporated*
    sensitivity_of_incremental_errors_to_past_errors {dimensionless}
maximum_error_fraction = 1 {dimensionless}
normal_error_fraction = IF test_first AND reviews THEN 0.05
    ELSE IF test_first OR reviews THEN 0.10
    ELSE 0.15 {dimensionless}
sensitivity_of_incremental_errors_to_past_errors = IF kiss THEN 0.8 ELSE 0.9 {dimensionless}
    DOCUMENT: Used to scale graphical function "fraction of undiscovered errors incorporated". Normally set to one for no scaling (range is zero to one).

    Decrease sensitivity when including discovered rework in calculation so it is comparable to numbers when didn't separate discovered rework out.
fraction_of_undiscovered_errors_incorporated = GRAPH(fraction_work_done_containing_errors {dimensionless})
    (0.00, 0.00), (0.1, 0.1), (0.2, 0.2), (0.3, 0.3), (0.4, 0.4), (0.5, 0.5), (0.6, 0.6), (0.7, 0.7), (0.8, 0.8),
    (0.9, 0.9), (1, 1.00)

Overtime

effect_of_fatigue_on_error_fraction =
    maximum_effect_of_fatigue_on_error_fraction*effect_of_fatigue_on_error_fraction_relation {dimensionless}
effect_of_fatigue_on_productivity =
    maximum_effect_of_fatigue_on_productivity*effect_of_fatigue_on_productivity_relation {dimensionless}
effect_of_overtime_on_error_fraction =
    maximum_effect_of_overtime_on_error_fraction*effect_of_overtime_on_error_fraction_relation +
    effect_of_fatigue_on_error_fraction {dimensionless}
    DOCUMENT: The fatigue effect from overtime is added in here to create an aggregate effect of overtime.
effect_of_overtime_on_productivity = 1 +
    maximum_effect__of_overtime_on_productivity*effect_of_overtime_on_productivity_relation +
    effect_of_fatigue_on_productivity {dimensionless}
    DOCUMENT: The (negative) fatigue effect from overtime is added in here to create an aggregate effect from overtime. This technically should affect "productivity before effects", but this relies on a subordinate calculation from this (total staff needed).
Effective_Staff = (1 + overtime_fraction)*Total_Staff {people}
    DOCUMENT: Effective number of full-time people working on the project (FTEs).

fatigue = SMTH3(overtime_fraction, time_to_develop_fatigue) {dimensionless}
indicated_overtime = IF (Total_Staff <> 0)
    THEN MIN(MAX(total_staff_needed - Total_Staff, 0)/Total_Staff, maximum_overtime_allowed)
    ELSE 0 {dimensionless}
maximum_effect__of_overtime_on_productivity = 0.5 {dimensionless}
maximum_effect_of_fatigue_on_error_fraction = 0.5 {dimensionless}
maximum_effect_of_fatigue_on_productivity = 0.5 {dimensionless}
maximum_effect_of_overtime_on_error_fraction = 0.3 {dimensionless}
maximum_overtime_allowed = 0.5 {dimensionless}
    DOCUMENT: Largest fraction of overtime allowed by management.
overtime_delay = 1 {months}
    DOCUMENT: This is short because the reaction to overtime pressure is usually pretty quick.
overtime_fraction = overtime_switch*SMTH3(indicated_overtime, overtime_delay) {dimensionless}
overtime_switch = 0 {dimensionless}
    DOCUMENT: Set to one to enable overtime and zero to disable it.
time_to_develop_fatigue = 6 {months}
effect_of_fatigue_on_error_fraction_relation = GRAPH(fatigue {dimensionless})
    (0.00, 0.00), (0.1, 0.02), (0.2, 0.04), (0.3, 0.075), (0.4, 0.12), (0.5, 0.195), (0.6, 0.275), (0.7, 0.365), (0.8, 0.5),
    (0.9, 0.72), (1, 1.00)
effect_of_fatigue_on_productivity_relation = GRAPH(fatigue {dimensionless})
    (0.00, 0.00), (0.1, -0.01), (0.2, -0.035), (0.3, -0.06), (0.4, -0.105), (0.5, -0.155), (0.6, -0.22),
    (0.7, -0.31), (0.8, -0.415), (0.9, -0.61), (1, -1.00)
effect_of_overtime_on_error_fraction_relation = GRAPH(overtime_fraction {dimensionless})
    (0.00, 0.00), (0.1, 0.02), (0.2, 0.04), (0.3, 0.075), (0.4, 0.13), (0.5, 0.2), (0.6, 0.295), (0.7, 0.41),
    (0.8, 0.56), (0.9, 0.775), (1, 1.00)
effect_of_overtime_on_productivity_relation = GRAPH(overtime_fraction {dimensionless})
    (0.00, 0.00), (0.1, 0.1), (0.2, 0.2), (0.3, 0.3), (0.4, 0.4), (0.5, 0.5), (0.6, 0.6), (0.7, 0.7), (0.8, 0.8),
    (0.9, 0.9), (1, 1.00)
    DOCUMENT: This is assumed a linear effect. Of course, it won't be, but fatigue will handle this.

Phase Bookkeeping
Initial_Undiscovered_Rework(t) = Initial_Undiscovered_Rework(t - dt) + (change_initial_undiscovered_rework) *
    dt
INIT Initial_Undiscovered_Rework = 0 {tasks}
INFLOWS:
change_initial_undiscovered_rework = IF start_new_phase THEN (Undiscovered_Rework -
    Initial_Undiscovered_Rework)/DT ELSE 0 {tasks/mo}

Latent_Work_in_System(t) = Latent_Work_in_System(t - dt) + (change_latent_work) * dt
INIT Latent_Work_in_System = 0 {tasks}
INFLOWS:
change_latent_work = IF start_new_phase THEN (Original_Work_to_Do + Rework_to_Do +
    Undiscovered_Rework - Latent_Work_in_System)/DT ELSE 0 {tasks/mo}

adjusted_fraction_of_total_perceived_complete = IF (agile_switch AND fraction_of_total_perceived_complete +
    1/phases < 1)
    THEN (1 - fraction_perceived_to_be_complete) {release excess staff at start of each phase}
    ELSE fraction_of_total_perceived_complete
    DOCUMENT: Don't let fraction complete exceed 50% until in last phase during agile (otherwise, we let staff
    go between phases).
agile_switch = 1 {dimensionless}
    DOCUMENT: Set to 1 to enable Agile tests (zero for conventional waterfall).
automated_test_switch = 1 {dimensionless}
    DOCUMENT: Set to zero to disabled nightly builds and automated testing (Agile only).
automated_tests = agile_switch AND automated_test_switch {dimensionless}
fraction_of_total_perceived_complete = total_work_believed_to_be_done/initial_work_to_do {dimensionless}

frequent_release_switch = 1 {dimensionless}
DOCUMENT: Set to zero to turn off the effects of frequent releases and customer interactions (Agile and uncertain customer requirements only).
frequent_releases = agile_switch AND frequent_release_switch {dimensionless}
kiss = agile_switch AND kiss_switch {dimensionless}
kiss_switch = 1 {dimensionless}
DOCUMENT: Set to zero to disable the KISS (Keep It Simple Stupid) effects in agile.
new_work_to_do_this_phase = initial_work_to_do/phases {tasks}
phases = IF agile_switch THEN 4 ELSE 1 {dimensionless}
reviews = agile_switch AND reviews_switch {dimensionless}
reviews_switch = 1 {dimensionless}
DOCUMENT: Set to zero to disable the effects of design and code reviews, as well as pair programming and commitment to technical and design excellence (Agile only).
start_new_phase = agile_switch AND (NOT project_finished_switch) AND fraction_of_total_perceived_complete < 0.95 AND Original_Work_to_Do < 0.04*work_to_do_this_phase AND Rework_to_Do < .01*work_to_do_this_phase {dimensionless}
DOCUMENT: We start a new phase if we haven't finished (and aren't close to finishing - within 5%) and we've finished a significant amount of the original work (at least 96% - numbers down to 90% have little additional effect) and have met a given standard of quality (less than 1% errors).
test_first = agile_switch AND test_first_switch {dimensionless}
test_first_switch = 1 {dimensionless}
DOCUMENT: Set to zero in Agile to turn off "test first".
Undiscovered_Rework_This_Phase = MAX(0, Undiscovered_Rework - Initial_Undiscovered_Rework) {tasks}
work_to_do_this_phase = new_work_to_do_this_phase + Latent_Work_in_System {tasks}
DOCUMENT: All work that must be done this phase (including things we don't know about, such as undiscovered rework). Note that most of this is known (Undiscovered Rework is marginal), so we use this even in policy decisions that should only be based on things we know. If we run cases where Undiscovered Rework becomes larger (unlikely), we will need to separate out another variable for these policies (estimated_work and start_new_phase).

Rework Calculations

Equivalent_Staff = IF project_finished_switch THEN 0 ELSE Total_Staff {people}
fraction_of_effort_to_rework = MIN(1,
maximum_work_rate_on_rework*relative_effort_required_for_rework/(MAX(0.001,
maximum_work_rate_on_rework*relative_effort_required_for_rework +
maximum_work_rate_on_original_work))) {dimensionless}
maximum_work_rate_on_rework = Rework_to_Do/minimum_time_to_perform_rework {tasks/mo}
minimum_time_to_perform_rework = 0.25 {months}
no_priority = 1 {dimensionless}
DOCUMENT: Set to one to have no priority, i.e., do work as best we can as it comes in (other two priorities must be zero).
original_work_priority = 0 {dimensionless}
DOCUMENT: Set to one to give original work priority (other two priorities must be zero).
potential_work_rate_on_rework = productivity_on_rework*Staff_on_Rework {tasks/mo}
productivity_on_rework = productivity_before_precedence_effects/relative_effort_required_for_rework {tasks/mo/person}
relative_effort_required_for_rework = 1 {dimensionless}
DOCUMENT: This is the effort required to fix problems relative to original work. A value of 1 means the effort is the same. A value of 0.5 means it takes half the effort while a value of 2 means it takes twice the effort.
rework_accomplishment = MIN(potential_work_rate_on_rework, maximum_work_rate_on_rework) {tasks/mo}
rework_priority = 0 {dimensionless}
DOCUMENT: Set to one to give rework priority (other two priorities must be zero).
Staff_on_Original_Work = Equivalent_Staff - Staff_on_Rework {people}
Staff_on_Rework = no_priority*fraction_of_effort_to_rework*Equivalent_Staff +
rework_priority*MIN(maximum_work_rate_on_rework/productivity_on_rework, Equivalent_Staff) +

original_work_priority*MAX(Equivalent_Staff - maximum_work_rate_on_original_work/
productivity_before_precedence_effects, 0) {people}
weighted_work_left_to_do = Original_Work_to_Do + relative_effort_required_for_rework*Rework_to_Do {tasks}

Rework Cycle
Original_Work_to_Do(t) = Original_Work_to_Do(t - dt) + (adding_work - rework_generation_on_original_work -
original_work_done_correctly) * dt
INIT Original_Work_to_Do = new_work_to_do_this_phase {tasks}
INFLOWS:
adding_work = IF start_new_phase THEN (new_work_to_do_this_phase)/DT ELSE 0 {tasks/mo}
OUTFLOWS:
rework_generation_on_original_work = error_fraction*original_work_accomplishment {tasks/mo}
original_work_done_correctly = (1 - error_fraction)*original_work_accomplishment {tasks/mo}

Previous_Work_Done(t) = Previous_Work_Done(t - dt) + (shipping_product) * dt
INIT Previous_Work_Done = 0 {tasks}
INFLOWS:
shipping_product = IF (start_new_phase OR project_finished_switch) THEN Work_Done/DT ELSE 0 {tasks/mo}

Rework_to_Do(t) = Rework_to_Do(t - dt) + (rework_discovery - rework_generation_on_rework -
rework_done_correctly) * dt
INIT Rework_to_Do = 0 {tasks}
INFLOWS:
rework_discovery = Undiscovered_Rework/time_to_discover_rework {tasks/mo}
OUTFLOWS:
rework_generation_on_rework = error_fraction*rework_accomplishment {tasks/mo}
rework_done_correctly = (1 - error_fraction)*rework_accomplishment {tasks/mo}

Undiscovered_Rework(t) = Undiscovered_Rework(t - dt) + (rework_generation_on_original_work +
rework_generation_on_rework - rework_discovery) * dt
INIT Undiscovered_Rework = 0 {tasks}
INFLOWS:
rework_generation_on_original_work = error_fraction*original_work_accomplishment {tasks/mo}
rework_generation_on_rework = error_fraction*rework_accomplishment {tasks/mo}
OUTFLOWS:
rework_discovery = Undiscovered_Rework/time_to_discover_rework {tasks/mo}

Work_Done(t) = Work_Done(t - dt) + (original_work_done_correctly + rework_done_correctly - shipping_product)
* dt
INIT Work_Done = 0 {tasks}
INFLOWS:
original_work_done_correctly = (1 - error_fraction)*original_work_accomplishment {tasks/mo}
rework_done_correctly = (1 - error_fraction)*rework_accomplishment {tasks/mo}
OUTFLOWS:
shipping_product = IF (start_new_phase OR project_finished_switch) THEN Work_Done/DT ELSE 0 {tasks/mo}

average_task_duration = 1 {mo}
effect_on_productivity_from_available_tasks = IF (project_finished_switch OR
potential_work_rate_on_original_work = 0) THEN 1 ELSE MIN(1,
maximum_work_rate_on_original_work/potential_work_rate_on_original_work) {dimensionless}
fraction_perceived_to_be_complete = work_believed_to_be_done/work_to_do_this_phase {dimensionless}
fraction_really_complete = Work_Done/work_to_do_this_phase {dimensionless}
initial_work_to_do = 100 {tasks}
max_work_rate_based_on_original_work = Original_Work_to_Do/average_task_duration {tasks/mo}
maximum_time_to_discover_rework = IF automated_tests THEN 12/phases ELSE 12 {mo}

DOCUMENT: Although it is true there may be rework that shows up later than specified here, if we do not divide the rework discovery between the phases, we cannot reasonably compare projects completed in one phase to projects completed in a number of phases. It therefore behooves shorter projects to take steps to ensure that the rework discovery delay is shortened (which Agile does, so we are safe). Project that don't can be modeled by removing the division (the result is disaastrous when the rework delay is larger than the scheduled project).

NOTE: Added automated tests switch to control including division or not.

maximum_work_rate_based_on_tasks_available = tasks_available_to_work_on/average_task_duration {tasks/mo}

maximum_work_rate_on_original_work = IF precedence_switch
THEN maximum_work_rate_based_on_tasks_available
ELSE max_work_rate_based_on_original_work {tasks/mo}

minimum_time_to_discover_rework = 0.25 {mo}

original_work_accomplishment = productivity*Staff_on_Original_Work {tasks/mo}

potential_work_rate_on_original_work = productivity_before_precedence_effects*Staff_on_Original_Work {tasks/mo}

precedence_switch = 1 {dimensionless}

productivity = productivity_before_precedence_effects*effect_on_productivity_from_available_tasks* effect_of_overtime_on_productivity { tasks/mo/person }

project_finished_switch = IF Work_Done + Previous_Work_Done >= initial_work_to_do - 0.4 {epsilon} THEN 1 ELSE 0 {dimensionless}

tasks_available_to_work_on = MAX(total_tasks_that_could_be_worked_on - work_believed_to_be_done - Rework_to_Do, 0) {tasks}

time_to_discover_rework = effect_of_work_progress*maximum_time_to_discover_rework + (1 - effect_of_work_progress)*minimum_time_to_discover_rework {mo}

total_tasks_that_could_be_worked_on = MIN(1, fraction_of_tasks_available_to_work_on_given_progress + agile_switch*0.12)*work_to_do_this_phase {tasks}
DOCUMENT: We have to bump up task fraction by 0.12 in Agile or we would have no tasks to work on. This may seem arbitrary, but with a smaller scope there are probably fewer interdependencies (meaning it could probably be bumped up even more). However, care must be taken on an Agile project to reduce the number of dependencies as too many will severely limit the number of people on the project (relative to a traditional project).

total_work_believed_to_be_done = Previous_Work_Done + Work_Done + Undiscovered_Rework {tasks}

work_believed_to_be_done = Work_Done + Undiscovered_Rework {tasks}

effect_of_work_progress = GRAPH(fraction_really_complete {dimensionless})
(0.00, 1.00), (0.1, 1.00), (0.2, 0.95), (0.3, 0.85), (0.4, 0.75), (0.5, 0.6), (0.6, 0.4), (0.7, 0.25), (0.8, 0.15), (0.9, 0.05), (1, 0.00)

fraction_of_tasks_available_to_work_on_given_progress = GRAPH(fraction_perceived_to_be_complete {dimensionless})
(0.00, 0.1), (0.1, 0.2), (0.2, 0.3), (0.3, 0.4), (0.4, 0.5), (0.5, 0.6), (0.6, 0.7), (0.7, 0.8), (0.8, 0.9), (0.9, 1.00), (1, 1.00)

Schedule Pressure

anticipated_schedule_overrun = IF project_finished_switch THEN 0 ELSE (perceived_completion_date - scheduled_completion_date)/MAX(17.5, scheduled_completion_date) {dimensionless}
DOCUMENT: We never divide by less than 17.5 because for short projects (e.g., the first phase of a multi-phase project), schedule pressure is too severe because of the very early due date (differences as a fraction of the actual due date are very large).

effect_of_schedule_pressure_on_error_fraction = IF schedule_pressure_switch
THEN sensitivity_for_effect_of_schedule_pressure_on_error_fraction* effect_of_schedule_pressure_on_error_fraction_relation + (1 - sensitivity_for_effect_of_schedule_pressure_on_error_fraction) - 1
ELSE 0 {dimensionless}

effect_of_schedule_pressure_on_productivity = IF schedule_pressure_switch

THEN sensitivity_for_effect_of_schedule_pressure_on_productivity*
effect_of_schedule_pressure_on_productivity_relation + (1 -
sensitivity_for_effect_of_schedule_pressure_on_productivity)
ELSE 1 {dimensionless}
indicated_completion_date_based_on_progress = IF Equivalent_Staff <> 0 THEN TIME +
estimated_effort_remaining/Equivalent_Staff ELSE TIME {months}
normal_productivity = IF test_first AND reviews THEN 0.85
ELSE IF test_first THEN 0.9
ELSE IF reviews THEN 0.95
ELSE 1 {tasks/mo/person}
perceived_completion_date = SMTH1(indicated_completion_date_based_on_progress,
time_to_perceive_real_schedule, initial_scheduled_completion) {months}
productivity_before_precedence_effects = normal_productivity*effect_of_schedule_pressure_on_productivity*
effect_of_experience_on_productivity{tasks/mo/person}
schedule_pressure_switch = 1 {dimensionless}
DOCUMENT: Switch to enable schedule pressures on productivity to make up for schedule delays (set to one to enable and zero to disable).
sensitivity_for_effect_of_schedule_pressure_on_error_fraction = IF overtime_switch THEN 0.5 ELSE IF
agile_switch THEN 0.75 ELSE 1 {dimensionless}
DOCUMENT: Reduce schedule pressure effect if implementing overtime or Agile (give overtime precedence).
sensitivity_for_effect_of_schedule_pressure_on_productivity = IF overtime_switch THEN 0.5 ELSE IF
agile_switch THEN 0.75 ELSE 1 {dimensionless}
DOCUMENT: Reduce schedule pressure effect if implementing overtime or Agile (give overtime precedence).
started_new_phase = DELAY(start_new_phase, DT) {dimensionless}
DOCUMENT: Pulses in the first DT of the new phase (vs. start_new_phase which pulses in the last DT of the previous phase).
time_to_perceive_real_schedule = IF started_new_phase THEN DT ELSE 1 {months}
DOCUMENT: At the start of each phase, the delay is reset to DT to reinitialize the smooth for this phase. Note that this only works because it is a SMTH1 (a SMTH3 would require 3*DT to reset) and that it introduces an artifactual 1 DT delay in the response of schedule pressure (in that one DT, pressure will essentially be removed). DT is small enough that we do not have to worry about this.
effect_of_schedule_pressure_on_error_fraction_relation = GRAPH(anticipated_schedule_overrun {dimensionless})
(-0.2, 0.85), (-0.1, 0.97), (-2.78e-017, 1.00), (0.1, 1.03), (0.2, 1.08), (0.3, 1.17), (0.4, 1.25), (0.5, 1.34), (0.6, 1.39), (0.7, 1.40)
effect_of_schedule_pressure_on_productivity_relation = GRAPH(anticipated_schedule_overrun {dimensionless})
(-0.2, 0.85), (-0.1, 0.97), (-2.78e-017, 1.00), (0.1, 1.03), (0.2, 1.08), (0.3, 1.17), (0.4, 1.25), (0.5, 1.34), (0.6, 1.39), (0.7, 1.40)

Schedule Slip

Imputed_Project_Cost(t) = Imputed_Project_Cost(t - dt) + (increasing_imputed_cost) * dt
INIT Imputed_Project_Cost = 0 {person-mo}
INFLOWS:
increasing_imputed_cost = IF (TIME < initial_scheduled_completion) OR project_finished_switch
THEN 0
ELSE imputed_cost_per_month_of_overrun {person-months/month}

imputed_cost_per_month_of_overrun = 10 {person-months/month}
schedule_slip_switch = 0 {dimensionless}
DOCUMENT: Set to 1 to enable schedule slipping (zero to disable).
scheduled_completion_date = initial_scheduled_completion + (perceived_completion_date -
initial_scheduled_completion)*willingness_to_slip*allow_schedule_slip*schedule_slip_switch {months}
Total_Project_Cost = Cumulative_Person_Months + Imputed_Project_Cost {person-months}
willingness_to_slip = 1 {dimensionless}
DOCUMENT: Ranges from zero to one. One means slip completely, zero means slip not at all.
allow_schedule_slip = GRAPH(fraction_perceived_to_be_complete {dimensionless})

(0.00, 1.00), (0.1, 1.00), (0.2, 1.00), (0.3, 1.00), (0.4, 1.00), (0.5, 1.00), (0.6, 1.00), (0.7, 1.00), (0.8, 1.00), (0.9, 1.00), (1, 1.00)
DOCUMENT: Controls period over which it is okay to slip project (allows there to be schedule pressure in the beginning of the project).

Staff Adjustment

Cumulative_Person_Months(t) = Cumulative_Person_Months(t - dt) + (doing_work) * dt
INIT Cumulative_Person_Months = 0 {person-months}
INFLOWS:
doing_work = IF project_finished_switch THEN 0 ELSE Effective_Staff {person-months/month}

Experienced_Staff(t) = Experienced_Staff(t - dt) + (gaining_experience - staff_leaving) * dt
INIT Experienced_Staff = initial_experienced_staff {people}
INFLOWS:
gaining_experience = New_Staff/time_to_gain_experience
OUTFLOWS:
staff_leaving = IF vary_staff_switch THEN
willingness_to_transfer\fire*excess_experienced_staff/average_time_to_transfer\fire ELSE 0 {people/mo}
DOCUMENT: We remove inexperienced staff before experienced staff.

initial_scheduled_completion(t) = initial_scheduled_completion(t - dt) + (change_schedule) * dt
INIT initial_scheduled_completion = initial_scheduled_completion_date/phases {months}
INFLOWS:
change_schedule = IF start_new_phase THEN (TIME + initial_scheduled_completion_date/phases - initial_scheduled_completion)/DT ELSE 0 {months/mo}

New_Staff(t) = New_Staff(t - dt) + (hiring - gaining_experience - new_staff_leaving) * dt
INIT New_Staff = initial_new_staff {people}
INFLOWS:
hiring = IF vary_staff_switch
THEN willingness_to_hire*extra_staff_needed/average_time_to_hire
ELSE 0 {people}
DOCUMENT: Note we could also add staff by reallocation, but here assume hiring is the worst case.
OUTFLOWS:
gaining_experience = New_Staff/time_to_gain_experience
new_staff_leaving = IF vary_staff_switch THEN
willingness_to_transfer\fire*excess_new_staff/average_time_to_transfer\fire ELSE 0 {people/mo}

average_time_to_hire = 4
average_time_to_transfer\fire = 1 {month}
DOCUMENT: Reallocation is usually fairly quick.
budgeted_effort_remaining = (estimated_work/normal_productivity)*(1 - fraction_perceived_to_be_complete) {person-months}
effect_of_experience_on_error_fraction = IF experience_dilution_switch
THEN (New_Staff*incremental_error_fraction_of_new_staff +
Experienced_Staff*incremental_error_fraction_of_experienced_staff)/(New_Staff + Experienced_Staff)
ELSE 0 {dimensionless}
DOCUMENT: The incremental error fraction based on additional new staff (where fraction is between zero and one).

Note (New_Staff + Experienced_Staff) is just Total_Staff. This is done to avoid redundancy of data connections.
effect_of_experience_on_productivity = IF experience_dilution_switch
THEN (New_Staff*relative_productivity_of_new_staff + Experienced_Staff)/(New_Staff + Experienced_Staff)
ELSE 1 {dimensionless}

DOCUMENT: The new staff have lower productivity, so are treated as fractional experienced staff. The total experienced staff equivalents is divided by the total staff to find the fraction of full (experienced) productivity in effect.

Note (New_Staff + Experienced_Staff) is just Total_Staff. This is done to avoid redundancy of data connections.

effective_productivity = IF total_work_believed_to_be_done = 0 OR Cumulative_Person_Months = 0
THEN normal_productivity
ELSE total_work_believed_to_be_done/Cumulative_Person_Months {tasks/person/month}
DOCUMENT: This is the long-term average productivity.

est_effort_remaining_based_on_progress = IF project_finished_switch THEN 0 ELSE
weighted_work_left_to_do/effective_productivity { person-months }

estimated_effort_remaining = IF follow_budget_switch
THEN budgeted_effort_remaining*(1 - weight_on_progress_based_estimates) +
est_effort_remaining_based_on_progress*weight_on_progress_based_estimates
ELSE est_effort_remaining_based_on_progress {person-months}

estimated_productivity = effective_productivity/effect_on_productivity_from_available_tasks {tasks/mo/person}
DOCUMENT: Divide by effective productivity based on tasks remaining to reduce averse effects at end of project cycle.

estimated_rework_fraction = 0.4
DOCUMENT: The estimated total rework that will be done over the course of the project as a fraction of the original work.

estimated_work = (1 + estimated_rework_fraction)*work_to_do_this_phase {tasks}

excess_experienced_staff = MAX(0, excess_staff - excess_new_staff) {people}
DOCUMENT: The MAX function isn't strictly necessary here. It is here to guard against errors elsewhere in the model.

excess_new_staff = MIN(New_Staff, excess_staff) {people}

excess_staff = MAX(0, Total_Staff - total_staff_needed) {people}

experience_dilution_switch = 1 {dimensionless}
DOCUMENT: Set to one to enable experience dilution effects (zero to disable).

extra_staff_needed = MAX(0, MIN(total_staff_needed, maximum_staff_level) - Effective_Staff) {people}

follow_budget_switch = 1 {dimensionless}
DOCUMENT: Set to 1 to follow project project with regards to staffing. Set to zero to staff based on progress and schedule.

incremental_error_fraction_of_experienced_staff = 0 {dimensionless}
DOCUMENT: Fraction of errors generated by the experienced staff above the normal error rate. This will usually be zero because we would just change the normal error rate otherwise.

incremental_error_fraction_of_new_staff = IF agile_switch THEN 0.35 ELSE 0.5 {dimensionless}
DOCUMENT: Fraction more errors generated by new staff than by experienced staff (i.e., new staff error fraction/experienced staff error fraction - 1).

Improve by 15% for Agile due to short cycles, so easier to jump into a project.

initial_experienced_staff = 4 {people}

initial_new_staff = 0 {people}

initial_scheduled_completion_date = 25 {months}

maximum_staff_level = 25 {people}

relative_productivity_of_new_staff = IF agile_switch THEN 0.65 ELSE 0.5 {dimensionless}
DOCUMENT: New staff productivity as a fraction of experienced staff (i.e., new staff productivity/experienced staff productivity).

Improve by 15% for Agile due to short cycles, so easier to jump into a project.

time_remaining = MAX(1, scheduled_completion_date - TIME) {months}
DOCUMENT: Automatically extend the schedule one month if we are not yet done. It is fairly typical to change the date in these cases and we do not want a zero result here. Note this is the time left to when we hope to finish the project.

time_to_gain_experience = 24 {months}

DOCUMENT: Time to gain experience overall (should be shorter for just this project).
Total_Staff = New_Staff + Experienced_Staff {people}
total_staff_needed = MIN(total_staff_needed_based_on_effort_and_time_remaining, total_staff_needed_based_on_max_work_rate) {people}
total_staff_needed_based_on_effort_and_time_remaining = estimated_effort_remaining/time_remaining {people}
total_staff_needed_based_on_max_work_rate = IF project_finished_switch THEN 0 ELSE maximum_total_work_rate/estimated_productivity {people}
vary_staff_switch = 1
DOCUMENT: Switch to enable staff to vary to make up for schedule delays (set to one to enable and zero to disable).

This also controls the letting go of people at the end of a project.
willingness_to_hire = 1
DOCUMENT: Varied between zero and one. Zero means we are not willing to hire anyone no matter what happens to the schedule (setting it to zero has the same effect as setting vary_staff_switch to zero). One means hire as required to meet the schedule. A value in-between allows some hiring to take place. In this case, it is treated as the fraction of needed hires we are willing to hire at any point.
weight_on_progress_based_estimates = GRAPH(fraction_perceived_to_be_complete {dimensionless})
(0.00, 0.00), (0.1, 0.00), (0.2, 0.00), (0.3, 0.1), (0.4, 0.25), (0.5, 0.5), (0.6, 0.75), (0.7, 0.9), (0.8, 1.00), (0.9, 1.00), (1, 1.00)
willingness_to_transfer\fire = GRAPH(adjusted_fraction_of_total_perceived_complete {dimensionless})
(0.00, 0.00), (0.1, 0.00), (0.2, 0.00), (0.3, 0.00), (0.4, 0.00), (0.5, 0.00), (0.6, 0.00), (0.7, 0.1), (0.8, 0.5), (0.9, 0.9), (1, 1.00)

Uncertain Requirements
effect_of_uncertain_customer_requirements = IF frequent_releases
THEN uncertain_requirements_switch*maximum_effect_of_uncertain_customer_requirements*(1 - elimination_of_uncertainty_based_on_progress_freq)
ELSE uncertain_requirements_switch*maximum_effect_of_uncertain_customer_requirements*(1 - elimination_of_uncertainty_based_on_progress_std) {dimensionless}
maximum_effect_of_uncertain_customer_requirements = IF kiss THEN 0.15 ELSE 0.2 {dimensionless}
uncertain_requirements_switch = 0 {dimensionless}
DOCUMENT: Set to 1 to enable the effect of uncertain customer requirements on errors (and zero to disable).
elimination_of_uncertainty_based_on_progress_freq = GRAPH(fraction_perceived_to_be_complete {dimensionless})
(0.00, 0.495), (0.1, 0.5), (0.2, 0.52), (0.3, 0.545), (0.4, 0.58), (0.5, 0.615), (0.6, 0.675), (0.7, 0.775), (0.8, 0.895), (0.9, 0.96), (1, 0.995)
elimination_of_uncertainty_based_on_progress_std = GRAPH(fraction_perceived_to_be_complete {dimensionless})
(0.00, 0.00), (0.1, 0.00), (0.2, 0.00), (0.3, 0.00), (0.4, 0.00), (0.5, 0.00), (0.6, 0.1), (0.7, 0.3), (0.8, 0.6), (0.9, 0.85), (1, 1.00)

Work Metrics
effect_on_productivity_from_precedence = IF project_finished_switch THEN 1 ELSE total_work_accomplishment/potential_work_rate {dimensionless}
maximum_total_work_rate = maximum_work_rate_on_original_work + maximum_work_rate_on_rework {tasks/mo}
potential_work_rate = potential_work_rate_on_original_work + potential_work_rate_on_rework {tasks/mo}
total_work_accomplishment = original_work_accomplishment + rework_accomplishment {tasks/mo}