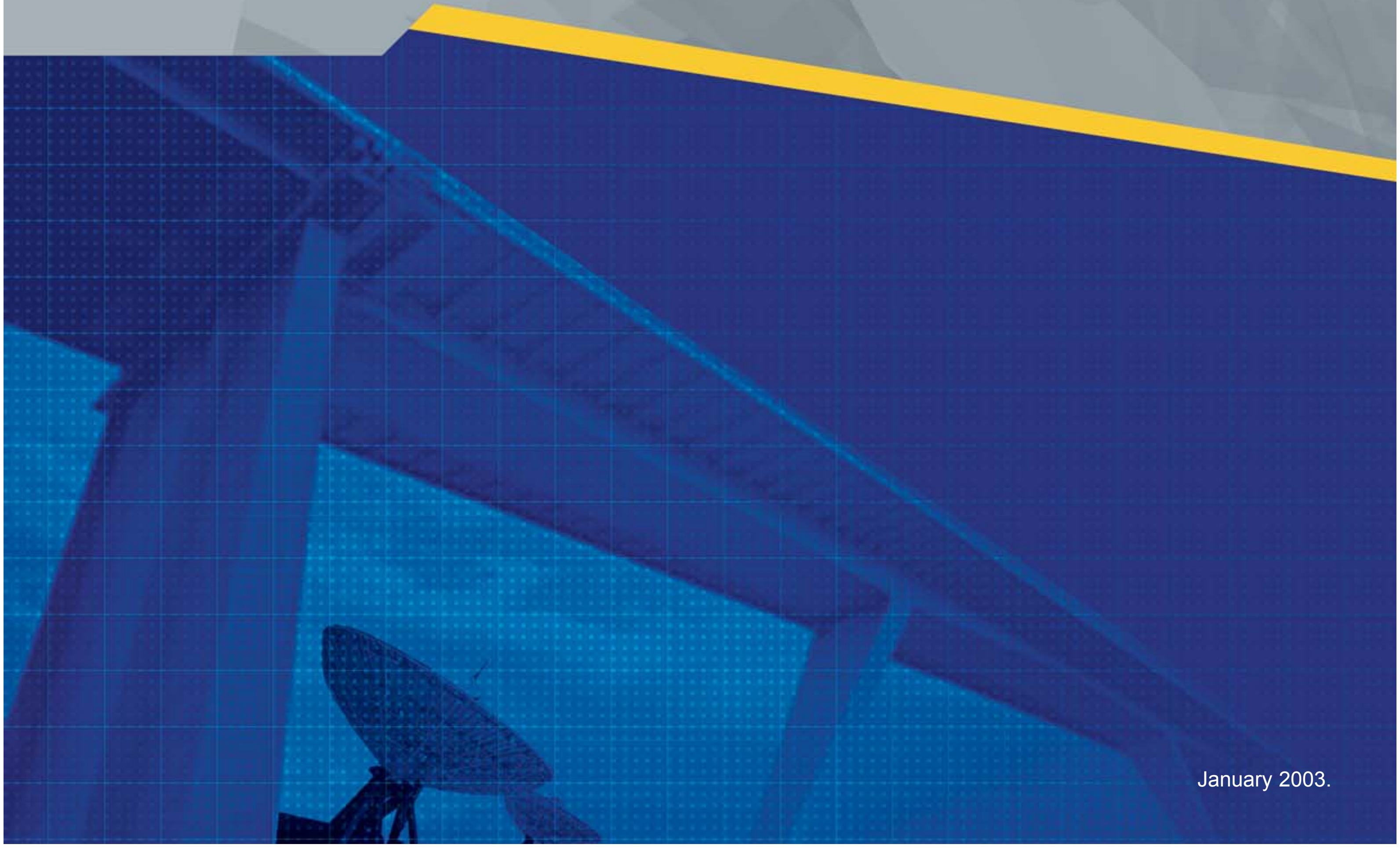
RETURN ON INVESTMENT MODELS

FOR STATIC ANALYSIS TOOLS

KHALED EL EMAM



January 2003.

Table of Contents

1 Introduction

A recent study [33] published by the U.S. Department of Commerce estimates that the economic consequences of inadequate software quality management practices, namely testing, amount to more than $59 billion per year. This is an astounding number at a national level and represents a nontrivial fraction of the GDP of the United States. For a single software organization, estimates of rework (fixing defects and bugs in software) can be as high as 80% of total development project costs [35]. Any, even modest, improvements to the quality of software can have significant financial impacts on individual organizations and on the overall economy.

This report discusses in depth the economic impacts of static analysis as one potential set of techniques to address quality deficiencies. Static analysis[1] is a set of tools and techniques for analyzing source code and software designs. The outcome of a static analysis could be:

- Metrics that quantitatively characterize the structure of the software
- Visual representations of the structure of the software (e.g., a graph showing which modules communicate with each other)
- A list of potential defects in the software

The information that is gained from such a static analysis can be very valuable for project managers and quality managers. The outputs of static analysis can be used to guide actions that result in:

- A reduction in software project costs (i.e., an increase in productivity)
- The delivery of higher quality software
- Reducing time to market (i.e., shorter delivery schedules)

The report will describe how static analysis can be used to achieve these results, and how to calculate the Return-On-Investment (ROI) from using static analysis tools and techniques. Examples are used to illustrate the use of the models. The ROI models can be customized to suite the particular life cycle of a project and their payback horizons. Our analysis and examples illustrate that a combination of static analysis technologies can save projects as much as 35-40% of their costs under rather modest assumptions by eliminating rework.

1.1 Concepts

As a starting point, we will define more precisely what is meant by *static analysis tools and techniques*. A static analysis involves the automated evaluation of a system's source code or design without actually executing it[2]. It is applicable to both functional software as well as object-

[1] There are other types of analyses that can be performed on a software system. One of these being a dynamic analysis. Whereas a static analysis does not require the software to be executed to produce results, a dynamic anlaysis requires the software to be executed and then information is extracted from the execution trace. We are only focused on static analysis.

[2] A design an be executed through a simulation, for example, the simulation of a state transition diagram or a state chart.

oriented software. The specific types of **evaluations** that are of interest to us are the following:

- Metrics are collected from the code or design. These metrics quantitatively characterize the size of the system, the coupling among the system's components, inheritance relationships among classes in an object-oriented system, and cohesion within functions and classes.
- Potential defects in the code are identified. For example, the use of uninitialized variables or NULL pointers, functions returning references to local objects, array bound violations, and incorrect memory deallocation are all likely to lead to a failure in the software. A static analysis tool builds abstract models of the software and its behavior, and uses that to identify potential defects.
- Static analysis tools can provide powerful visualization capabilities. For large systems, the visualization can give a concise picture of which parts of the system communicate with eachother, where the bottlenecks are, which parts are more complex than others. In addition, visualization tools allow programmers and designers to see which parts of a system are likely to be affected by a change in the code, and therefore can be helpful for impact analysis.

Figure 1 shows the sequence of mechanisms that would lead to concrete benefits from the use of static analysis.

Static analysis tools provide information to project managers, quality managers, architects, and programmers. It is up to them to take **actions** based on the information provided. Below are the types of actions that can be taken:

- **Automatic detection of defects.** Once a static analysis tool has identified all of the potential defects in a system, programmers and architects then fix these defects. The major advantage of the static analysis tools is that they save the programmers and architects time by finding the defects for them: a considerable amount of effort is typically spent during maintenance and testing just tracing from symptoms to the actual defects. Compared to other defect detection techniques, automatic detection is a large saving.
- **Risk management.** The metrics collected during static analysis can be used to identify the highest risk modules or components in a system. Most defects in software are found in a small percentage of the system's modules. If these high-risk modules are identified early then preventative actions can be taken by the project. An example of a preventative action is to inspect those high risk modules. A more detailed exposition of a risk management approach is presented in the appendix.
- **Efficient changes.** Static analysis results can also help reduce code change costs when fixing defects. A considerable amount of effort is spent looking for the defects to fix when a failure is discovered. This is called *isolation* effort. A software failure can occur during testing or operation. A visualization tool can help programmers see which modules are connected to each other,

and this can assist in navigating the software to where the defect is. In addition to reducing isolation effort, a visualization tool can help a programmer avoid bad fixes. These are code fixes that introduce new defects. Bad fixes are typically due to the programmer not realizing the module that has been changed is being used by another module. When one module is changed the other module breaks because now their behaviors are not compatible. Visualization tools can be very powerful for doing an impact analysis to identify the impact of a change.

- **Discovery of structured code.** The structure of long-lived systems usually deteriorates over time. This deterioration is due to all of the piecemeal changes that are made to the system throughout its lifetime. These changes may be to fix defects or to add new functionality. Badly structured systems are very expensive to maintain (each change takes a long time), and also have a higher risk of defects. At some point the organization may wish to rewrite the system to improve its structure. However, rather than rewriting the whole system it would be prudent to salvage parts of the existing system that are well structured. A visualization tool can assist the architects in identifying parts of the legacy system that can be reused in the new system Such reuse can reduce the overall cost of the new development.

  There are also cases where an existing system needs to be customized for multiple new clients. A visualization tool can identify the parts of that system that can be reused easily for each of these customizations. To the extent that the customizations can maximize reuse, their development costs can be a fraction of what it would have cost to develop the functionality anew.

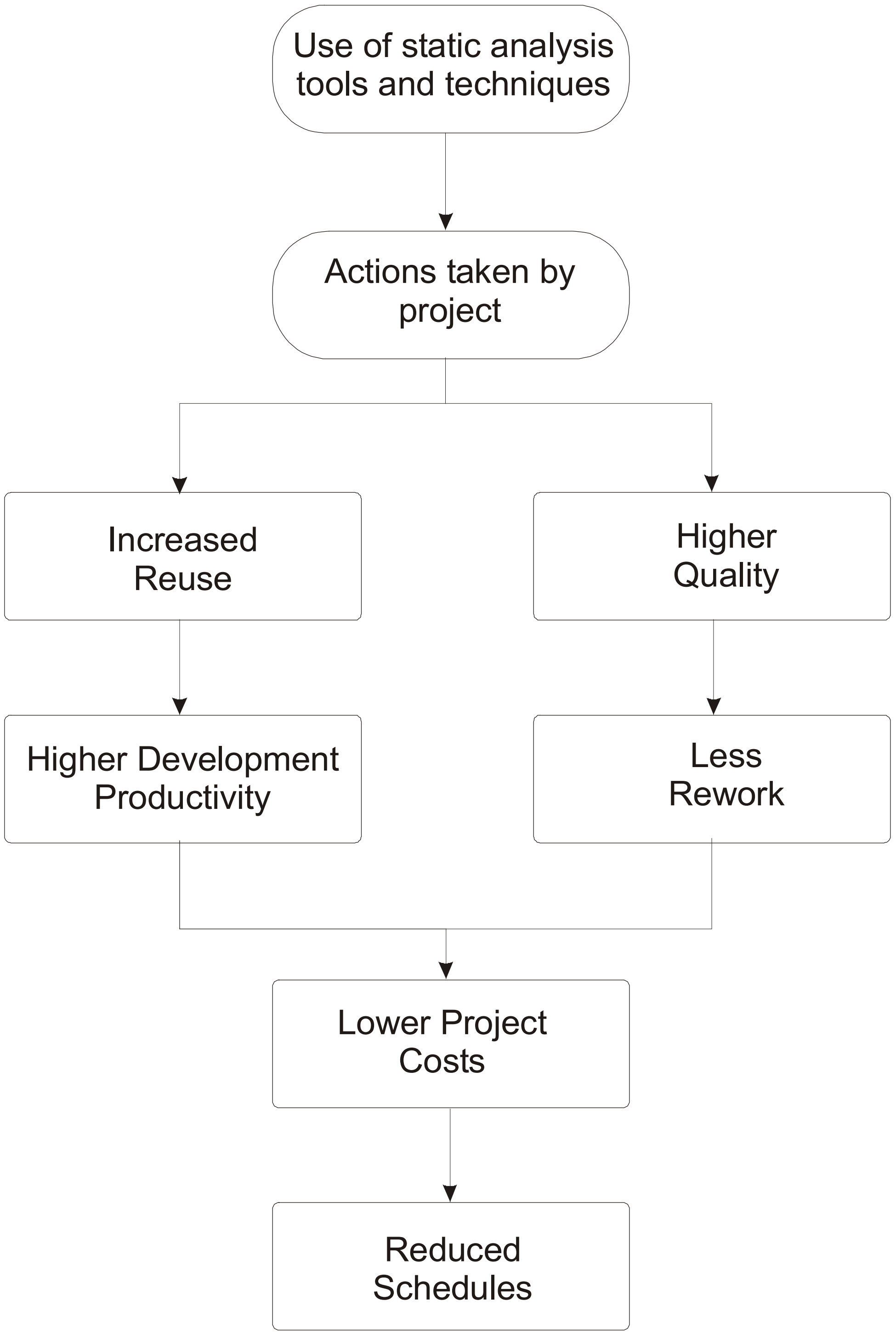


Figure 1: Illustration of the sequence of mechanisms that will lead to reductions in cost and schedule from a static analysis.

Depending on the actions that the project will take based on the static analysis results, the consequence would be either that there is increased reuse during development, the delivery of high quality software, or both. Increased reuse will lead to higher productivity. Higher quality will lead to lower rework.

It is not untypical that 50% or more of a project's cost can be rework. Rework means fixing defects. If the quality of the software is higher then less effort will be spent on rework since fewer defects need to be fixed.

Higher development productivity and lower rework result in reduced overall software project costs. And, lower total costs mean less effort by the project staff, which translates into a reduction in overall project schedule.

The above paragraphs have outlined the chain of causal events that would lead to reduced cost and schedule for a software project that uses static analysis. Of course, the specific benefits will depend on which actions are taken by the project. For instance, if the organization does not do nor does it plan on doing inspections, then there will be limited, if any, benefits from the use metrics and risk assessment techniques.

1.2 Project Costs

To quantify the benefits of static analysis, it is informative to get an understanding of software project costs.

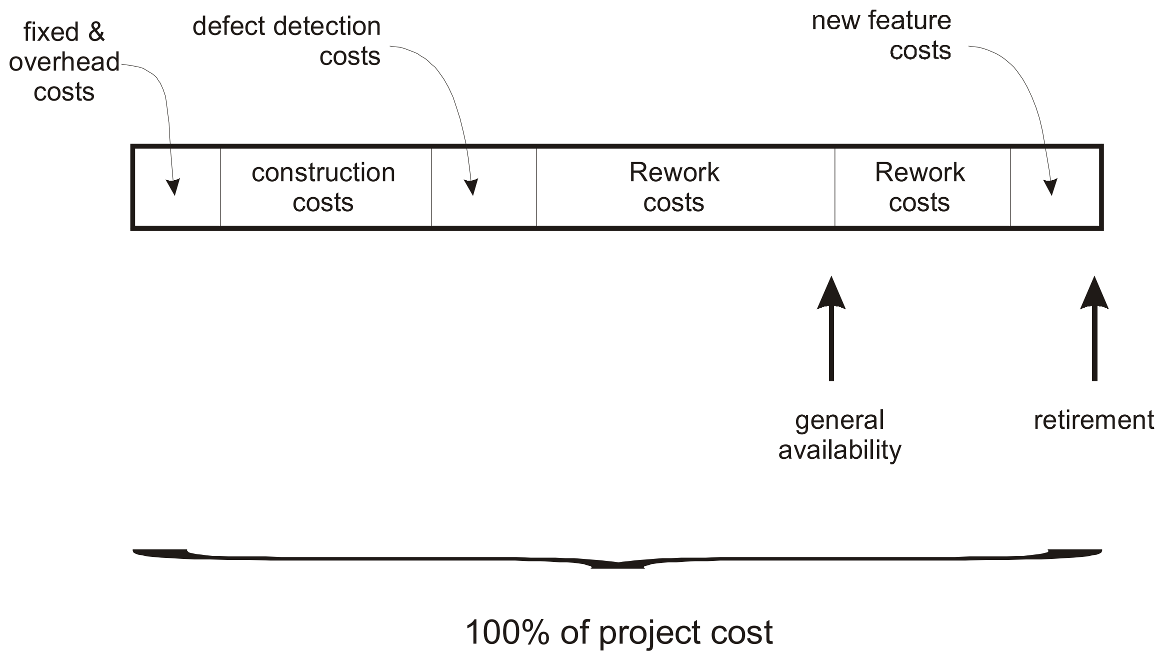


Figure 2: A breakdown of software project costs.

Figure 2 shows a typical breakdown of software projects costs. Every project has a fixed and overhead costs. These include things like rent, furniture, and electricity bills. Construction costs consist of the effort associated with the actual software development activities, such as requirements analysis, design and coding. Defect detection costs are the effort to look for defects introduced during construction. Defect detection includes activities such as inspections (peer reviews), testing, and root cause analysis. Rework costs are all costs to fix defects. There are rework costs pre-release (before general availability), and rework costs post-release. Pre-release rework is due to fixing defects found during inspections and testing, as well as other defect detection activities before the product is released. Post-release rework is due to fixing defects that were detected largely by customers. Although, a minority of defects will be found through internal testing even after a product is released. After a product is released, further effort is spent adding new features and porting the application. This breakdown covers the total life cycle of a software product.

Rework costs can be further itemized as follows:

- The effort to recreate the problem. This is relevant mostly for problems reported by users, where it can take some effort to first find out what the user did and the user's configuration, and then additional effort to set up a similar configuration to recreate the reported problem and confirm its existence.
- Trace from observed failure to the defect. This is relevant for problems reported by customers and failures observed during testing. It could take some time to trace from the symptoms that are observed to the actual defects in the code that need to be fixed. This tracing may be manual or can be aided by debuggers.
- Implement a fix. The fix for a particular failure may involve making changes to multiple modules or components. It may involve a redesign of part of the system which is no tlocalized.
- After the fix is made test cases have to be written to test for that particular failure. The test case is to ensure that the fix is correct, and the test cases go into the regression test suite. The regression test suite ensures that the problem does not occur again inadvertently due to future changes.
- The regression test suite is then rerun. Retesting may involve the setup of special hardware or databases, and can be quite expensive.
- Once the fix has passed regression testing, the change usually has to be documented.
- The fix, alone or perhaps along with other fixes, are packaged into an emergency release, a service release or a full release and shipped to the appropriate customers. If the product has not been released yet, then special packaging costs for the fix will not be incurred.

The first two bullets constitute what is known as *isolation effort* when making a change.

Static analysis can lower project costs, as depicted in Figure 1, in one of two ways:

- Reductions in the construction costs and new feature costs. This can be achieved through increased reuse instigated by static analysis.
- Reductions in rework costs. Rework costs can be reduced in a number of ways. Given the scope of this issue, it will be covered in detail below (see Section 1.4).

The following table provides a mapping between the actions described above and the types of cost reduction that would be expected from that action.

	More Reuse	Less Rework
Automatic detection of defects		X
Risk management		X
Efficient changes		X
Discovery of structured code	X	

1.3 The Payoff from Cost Reductions

In this subsection we will present, at a conceptual level, how the payoff from cost reductions comes about. First, let us consider the payoff from reuse.

In Figure 3 we can see the breakdown of project costs without reuse and the breakdown after reuse. It is expected that construction costs will decrease. There will be an investment in the reuse effort itself. This involves the effort to identify the code that should be reused, the licensing costs of tools that are required to do so, and effort that may be required to wrap and document interfaces. However, given that code is being reused from scratch, the overall savings in construction result in an overall reduction of project cost (i.e., overall project cost savings).

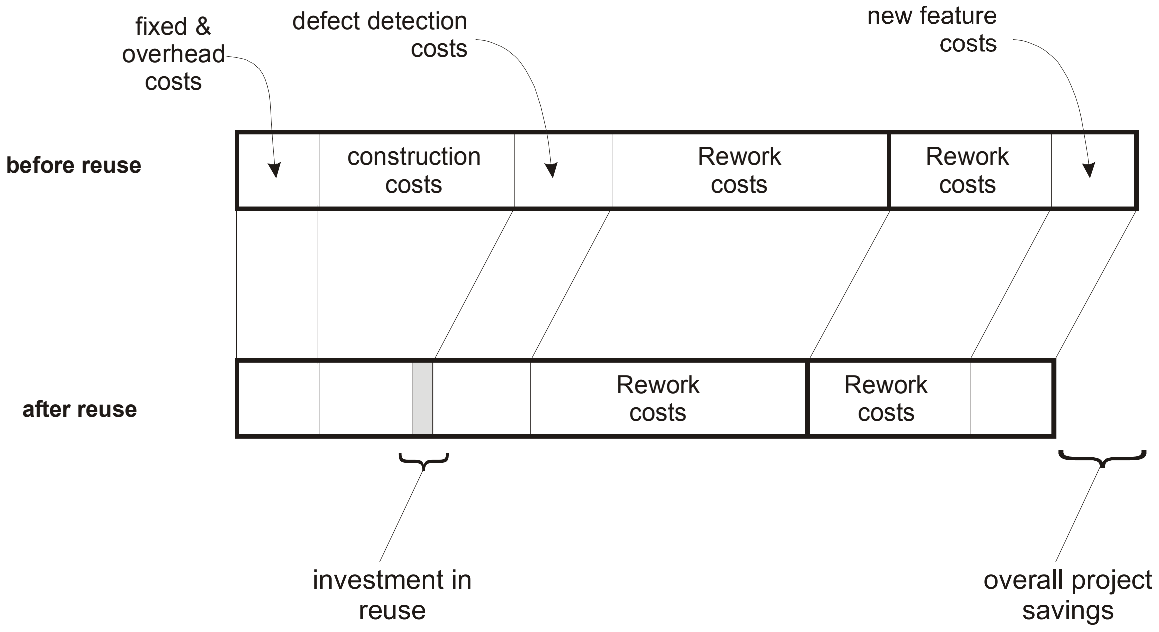


Figure 3: Illustration of the payoff from increased reuse.

Figure 4 illustrates the payoff from techniques that reduce rework. Here there is an initial investment in defect detection. Therefore defect detection costs go up. Rework costs before and after release go down considerably. Hence the pre- and post-release project cost would be smaller than for the initial project.

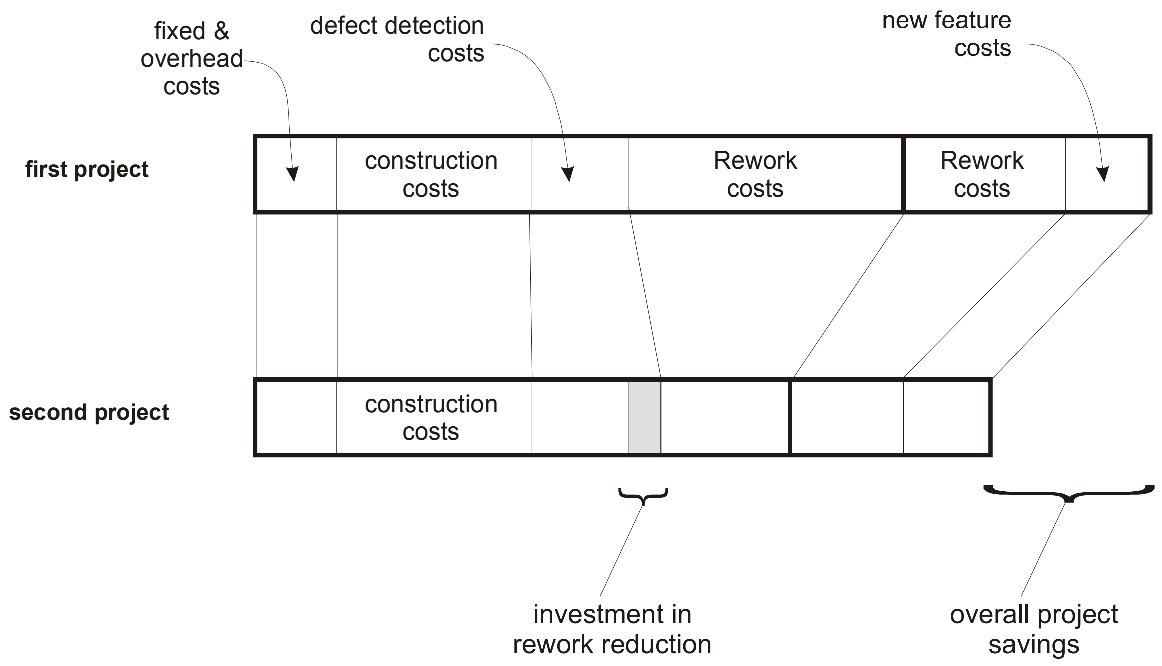


Figure 4: Illustration of the payoff from better defect detection techniques.

When we evaluate ROI we essentially look at the tradeoff between the amount that is invested versus the project savings.

1.4 Reducing Rework Costs

Rework, as the name suggests, involves doing work again. It describes fixing defects in software projects. The cost of rework rises as one moves into later phases of a project. Figure 5 illustrates this rise in the cost of fixing defects. The figure assumes that the software had a design defect. The cost of fixing the design defect during design is relatively low. If the defect escapes into coding, then the costs escalate. If the defect slips further into testing and later into release, and is found by the customer, the correction costs can be considerable. Therefore, it is much cheaper to find and correct defects as early as possible when their costs are relatively low.

There are many examples in the literature of the increasing costs of finding defects later in the life cycle. Khoshgoftaar [24] cites a case in a telecommunications company where the cost of a post-release defect is 200 times larger than finding and fixing the defect pre-release. Another telecommunications system project costs a post-release fix as high as 880 times more expensive than when done earlier in a project [3]. Further data from other domains show cost increases per defect greater than 100 times from before to after release [35] for severe defects.

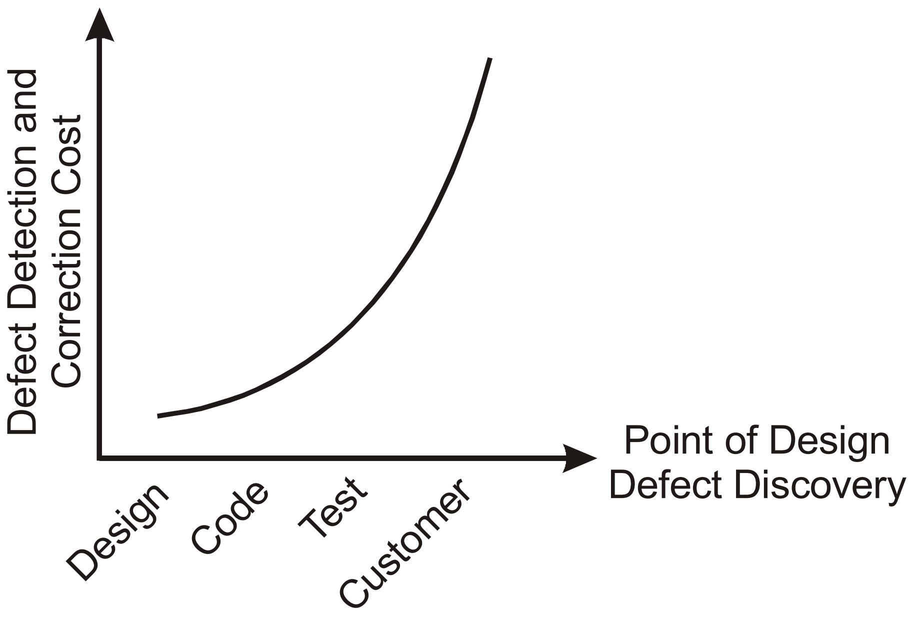


Figure 5: The increasing costs of fixing a design defect.

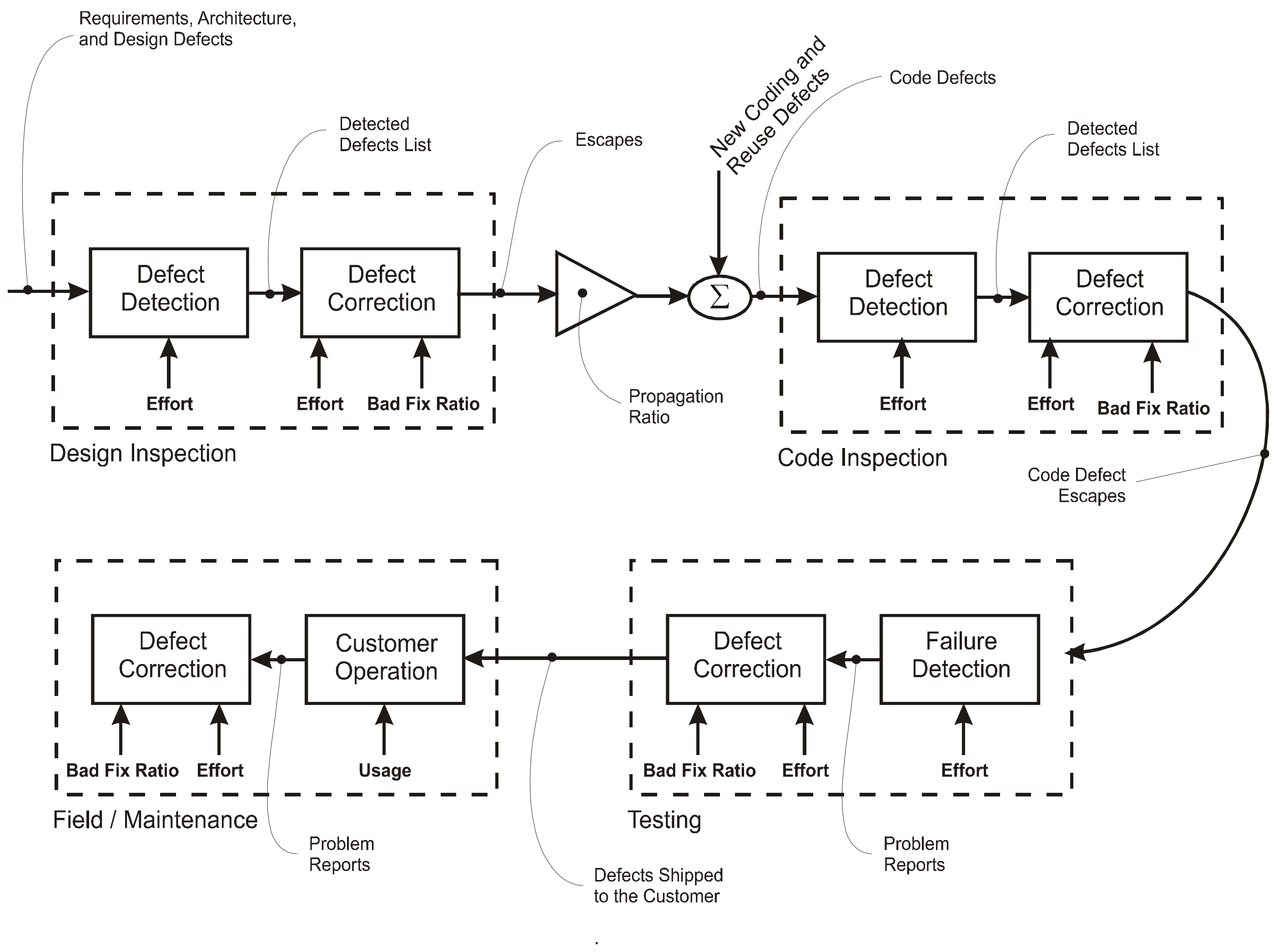


Figure 6: The defect detection life cycle. Although there is only one testing phase shown, in practice there are many types of testing and testing is performed over multiple phases.

To find defect as early as possible in the project, various defect detection activities are employed. Figure 6 shows an example of a defect detection life cycle. Here, both design and code defects are used to detect defects. Detecting defects during inspections is much cheaper than detecting them in testing or for them to be detected by the customer.

Defects in the requirements, architecture and design documents enter the design inspection process. Effort is spent during the design inspection to find and fix these defects. Some of the fixes will be bad and will introduce more defects. The **Bad Fix Ratio** tells us how many of the fixes that are made will introduce another defect. For example, if this is 50% then half of the fixes will not be done properly and will introduce another defect.

A design inspection will not capture all defects, so there will be escapes. The number of requirements and design defects detected during the inspection will depend on how well the inspection process is optimized. Escapes occur because there are certain types of defects that a design inspection is not really capable of detecting, and even for those defects that are targeted by the inspection, the inspection process rarely achieves 100% perfection. An escaped design defect can propagate into more than one coding defect. This is captured by the **Propagation Ratio**. Usually this is due to an incorrect assumption that is made at the design stage that leads to many instances of defects in the implementation, for example, assumptions about the capabilities of an external component.

Further defects will be added to the software during coding. Also, reused code either from third parties as a component or from an earlier release or project, will also have defects in it.

Code inspections behave, at the abstract, in a manner that is similar to design inspections. After code inspections the escapes go through testing. Note that there is no propagation ratio for code defects and no new defects are introduced after testing except due to bad fixes.

Testing identifies failures that are documented as PRs (Problem Reports). Programmers then trace from the PRs to defects and fix these. Testing may introduce defects due to bad fixes. Defects that escape testing go into the field where customers discover them. Customers report failures that are documented as PRs, and these then make their way to the development organization for fixing.

One of major drivers for pre-release defect detection effectiveness (i.e., how well these activities can find defects) is effort spent on defect detection. The more effort spent on defect detection the more defects will be found. The relationship is illustrated in Figure 7. The rate at which defects are found tends to plateau because most easy defects are found early and the harder defects remain. Harder defects require more effort to find. In the case of inspections, inspector fatigue results in defect detection slowing down after a certain period of time (around 2 hours of time per inspector).

The implication is that there is a trade-off between spending large amounts of effort detecting defects and having defects escape to the customer. For some applications there is actually no trade-off, for instance, for safety critical systems whatever effort is required to eliminate detected field defects will be spent. But for most projects this is a business decision. Similarly, if any of the defect detection activities is skipped, then the escapes will increase and the number of defects making it to the field will also increase.

Escapes from testing go to the customer. Post-release defects originate primarily from PRs from external customers. External customer PRs are driven largely by usage. But also, there are sometimes internal customers who generate PRs, and testing of changes and fixes also identify defects.

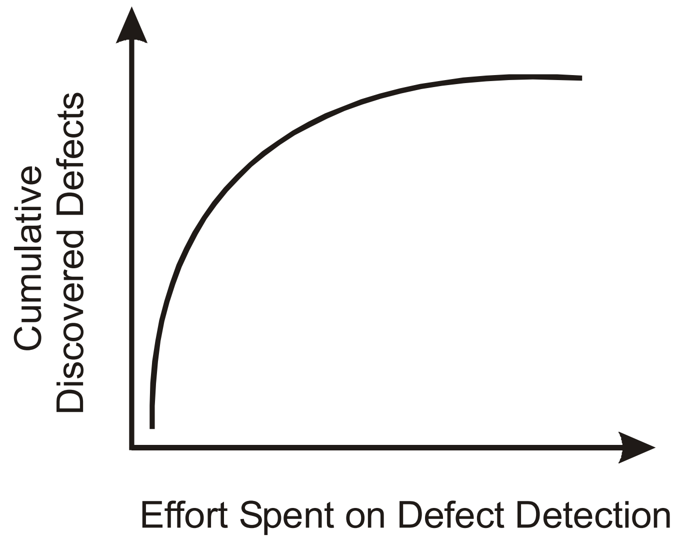


Figure 7: Relationship between effort spent on pre-release defect detection activities and defects found.

Now let us consider how the various actions we mentioned above will influence the rework costs.

1.4.1 Automatic Detection of Defects

When defects are automatically detected through static analysis, then this plays the same role as a design and code inspection. The inspections also are intended to statically detect defects. However, running a tool to detect this is less labor intensive and less costly. Once a defect is detected, then it is fixed.

Compared to the inspections in Figure 6, automatic defect detection essentially eliminates the defect detection costs. Therefore, the benefits of automatic defect detection can be looked at in two ways:

- Benefits compared to not having any inspections.
- Benefits compared to having design and code inspections.

1.4.2 Risk Management

Instead of inspecting all of the code and design models, risk assessment techniques would allow the project to inspect only the high risk modules or components. In practice, it is very difficult to inspect all of the code. Imagine if you were developing a 200,000 LOC system and had run inspections with each inspection covering 200 LOC. Then we are talking about almost 1000 inspections during a project. This number of inspections would take a considerable amount of time and effort to complete, and with today's release cycles would likely be impossible to accomplish. Most contemporary inspection implementations have a meeting, and typically a meeting with three or five busy engineers can only be scheduled a few weeks later. Imagine hundreds of meetings with each incurring a delay of a few weeks before they can be held. Votta provides a good discussion of this issue [37].

The benefit of risk management is then that it allows focused or targeted design and code inspections. This benefit can be evaluated in comparison with:

- The benefits of inspecting everything rather than focusing on the high risk modules.
- Not performing any inspections.

1.4.3 Efficient Changes

Static analysis can reduce the isolation effort when making changes to fix defects. As noted above, isolation effort can be nontrivial when fixing testing and post-release defects. Another benefit of static analysis is that it can reduce the bad fix ratio when making changes. The bad fix ratio ensures that even if all known defects are fixed, we will not end up with a defect-free product because the fixes introduced more defects. If bad fixes are reduced then the total number of defects that have to be fixed goes down. This has a direct impact on rework costs.

1.4.4 Epilogue

One important consideration when talking about rework costs is whether we are considering pre-release rework costs or both pre- and post-release rework costs. The benefits of practices such as inspections and static analysis really become evident when post-release costs are taken into account. The reason why post-release costs should be taken into account is that these costs have a direct impact on **time-to-profit**.

Figure 8 illustrates the difference between time-to-market and time-to-profit. The costs after a product is released determine the point at which a product makes a profit. Therefore, good practices, such as inspections and static analysis, during development can reduce post-release costs and shorten the time-to-profit.

This point will be illustrated further through some of our case studies and examples.

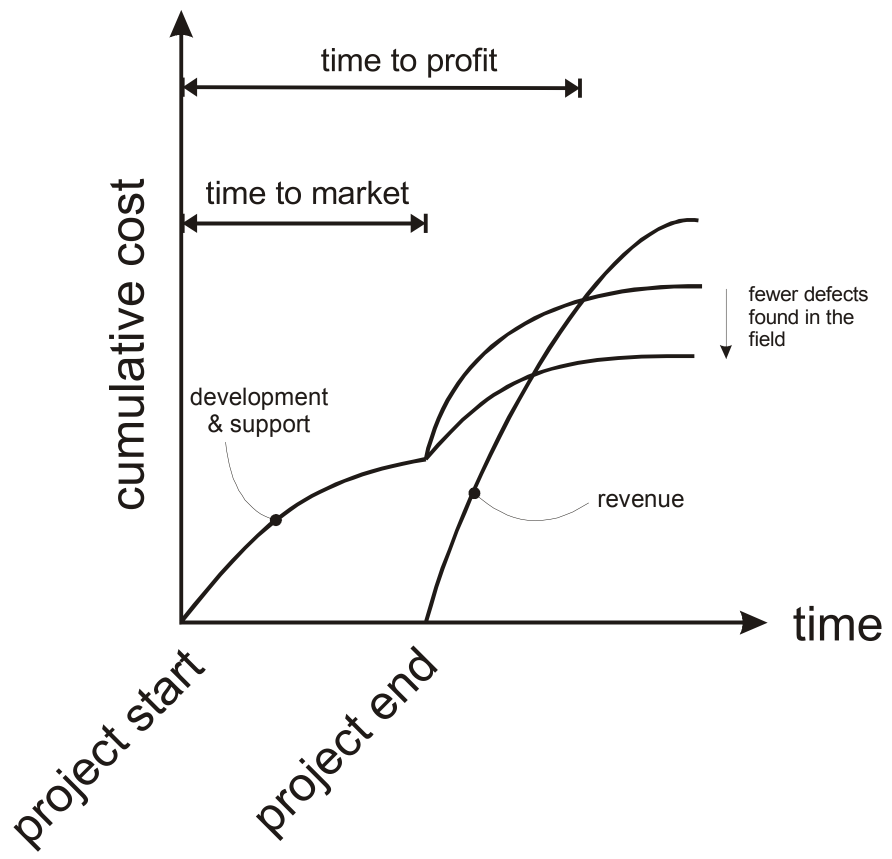


Figure 8: Example illustrating the difference between time-to-market and time-to-profit.

2 Calculating Return on Investment

This section presents a number of examples of calculating ROI based on the models that are articulated in the appendix. For each ROI model we have to make some assumptions about the defect detection life cycle. These assumptions will be stated for each of the models below.

2.1 Example 1: Automated Defect Detection

We assume a project that performs only testing before release. Therefore some defects are detected during testing and the remaining defects are found by customers. The ROI model was formulated as shown below (this model takes advantage of default values):

$$ROI = \hat{p}'_{Auto} \times \left(0.5 - \frac{0.55\hat{\varepsilon}_{Auto}}{\left(6\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(1-\left(0.9\times\hat{p}_{Test}\right)\right)\right)} \right)$$

The following are the input variables required and the values for our example:

Notation	Meaning	Value
$\hat{p}'_{Auto}$	The effectiveness of automatic detection of defects, taking into account bad fixes.	0.05
$\hat{\varepsilon}_{Auto}$	The effort to find and correct a defect during automatic detection	1 hr
$\hat{\varepsilon}_{Customer}$	The effort to find and correct a defect during post-release	50 hrs
$\hat{p}_{Test}$	The effectiveness of testing.	0.5
C	Total project cost	$1 million

The above data assume that automatic detection will find only 5% of the defects that are in the modules that are analyzed (a rather modest assumption). The result is as follows:

Notation	Meaning	Value
ROI	ROI expressed as project savings	2.4%
Savings	The effort to find and correct a defect during post-release	$24,000

The values would be different if we did not use the defaults for the effectiveness and costs of testing.

Let us look at a number of scenarios. If the effectiveness of automatic detection increases, the savings (percentage) would be as shown in Figure 9. The figure shows that at the maximum potential benefit of 20% effectiveness, the highest savings that one could expect under default values is around 9.5% cost savings from the project ($95,000). This is quite a large saving. Whether automatic detection can achieve as high as a 20% effectiveness is an open question, however, but it does illustrate potential.

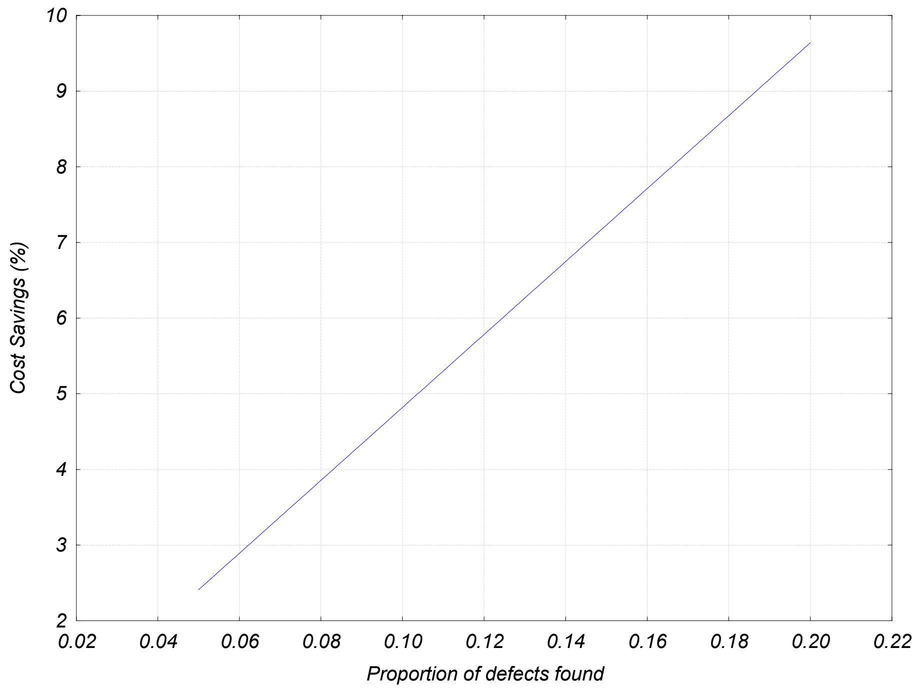


Figure 9: Graph showing the cost savings as a percentage as the effectiveness of automatic defect detection increases from a minimum of 0.05 to 0.2. This means that the static analysis finds from 5% to 20% of all the defects in the code.

An interesting observation can be made from Figure 10: at the values used in the table above, it is very unlikely that the cost savings will exceed 2.5% if the effectiveness of automatic defect detection is at 5% (using the default values) as post-release costs rise. This means that the benefit from automatic detection is applicable for projects with smaller post-release costs and that projects with high post-release costs should consider additional strategies to maximize savings. The reason is that automatic defect detection has a low effectiveness. Therefore its benefits will always be limited because as the cost of post-release defects increase, savings from the limited effectiveness will always be a small proportion of overall post-release costs.

We could not identify reliable data on the effectiveness of automatic detection tools. This is partially because the effectiveness will depend on the types of rules that are used to identify defects, and hence one could expect wide variation. The value that is used in this example should therefore be considered cautiously.

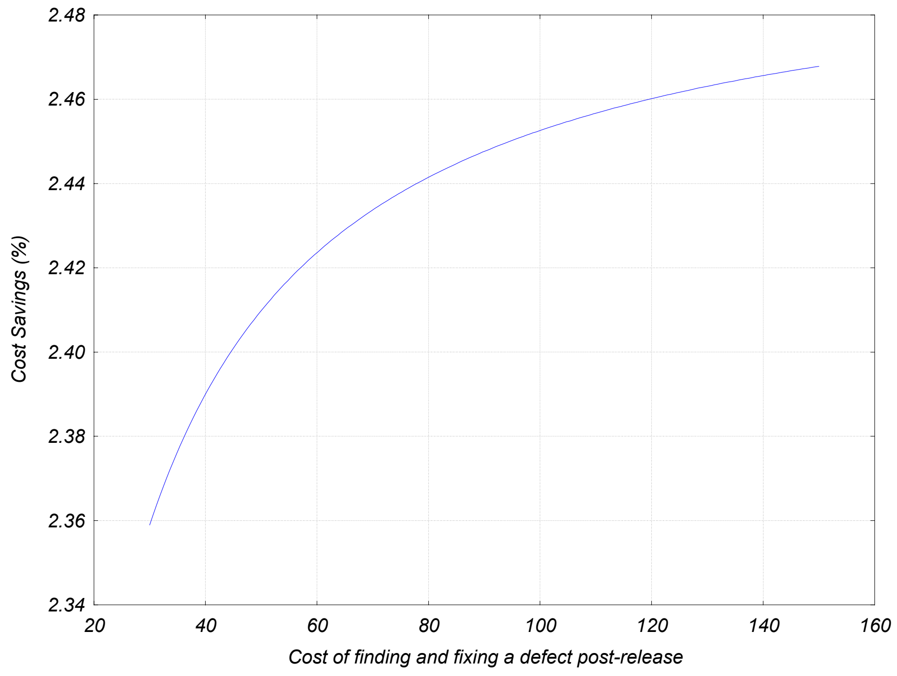


Figure 10: Graph showing the cost savings as a percentage as the cost of a post-release defect increases from 30 hours to 150 hours.

2.2 Example 2: Improving Maintenance Efficiency

We assume a project that performs only testing before release. Therefore some defects are detected during testing and the remaining defects are found by customers. The ROI model for maintenance efficiency was formulated as shown below (this model takes advantage of default values):

$$ROI_2 = \left(1 - \frac{\left(6 \times \hat{p}_{Test}\right) + \left(\hat{\varepsilon}'_{Customer} \times \left(1 - \left(\left(\hat{p}_{Test} \times 0.9\right) \times g\right)\right)\right)}{\left(6 \times \hat{p}_{Test}\right) + \left(\hat{\varepsilon}'_{Customer} \times \left(1 - \left(\hat{p}_{Test} \times 0.9\right)\right)\right)}\right) \times 0.5$$

The following are the input variables required and the values for our example:

Notation	Meaning	Value
g	The increase in good fixes during maintenance.	1.1
$\hat{\varepsilon}_{Customer}$	The effort to find and correct a defect during post-release	50 hrs
$\hat{p}_{Test}$	The effectiveness of testing.	0.5
C	Total project cost	$1 million

The above data assume that there will be a ten percent reduction in bad fixes during maintenance with the help of a visualization and navigation tool. The result is as follows:

Notation	Meaning	Value
ROI	ROI expressed as project savings	3.69%
Savings	The effort to find and correct a defect during post-release	$36,900

The values would be different if we did not use the defaults for the effectiveness and costs of testing. The plot in Figure 11 illustrates the potential savings even as the post-release costs per defect reach as high as 500 hours per defect. In general, we see that the savings tend to plateau at around 4 percent. Of course, this model only uses the default values for the remaining parameters, but it does clearly illustrate a plateau effect.

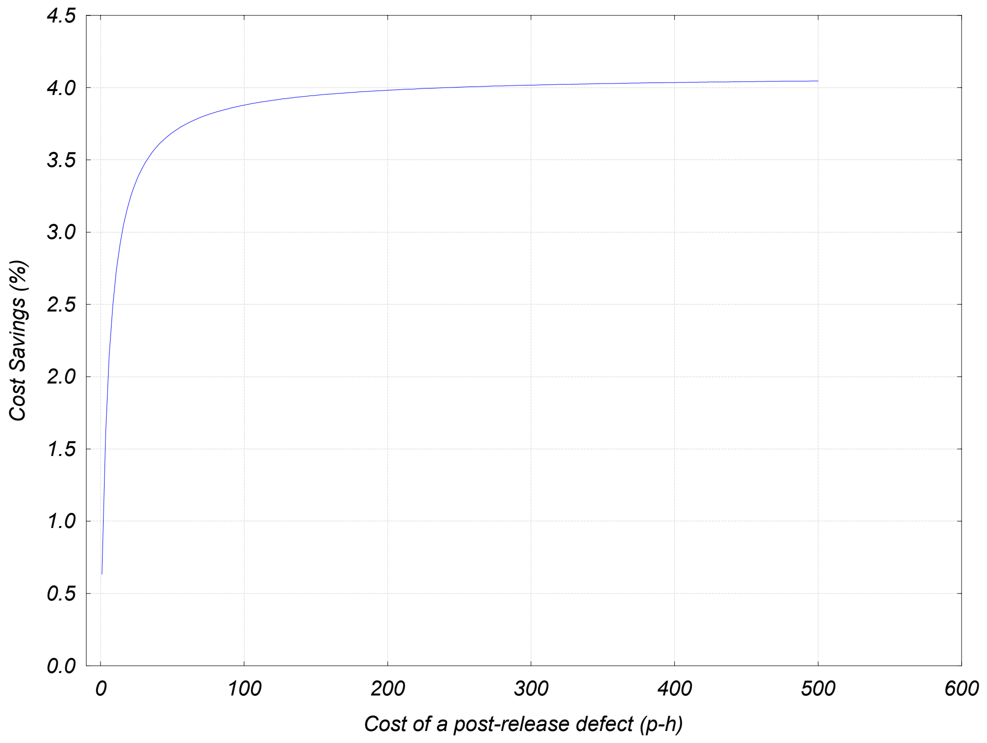


Figure 11: Graph showing the percentage cost savings from improvements due to reductions in bad fixes during testing and maintenance.

2.3 Example 3: Risk Assessment

We assume a project that performs only testing before release. Therefore some defects are detected during testing and the remaining defects are found by customers. Using the risk assessment methodology described earlier, code inspections are introduced. However, rather than inspecting all of the code, only the highest risk modules of the system are inspected. The ROI model for risk assessment was formulated as shown below (this model takes advantage of default values):

$$ROI = \frac{\left|\lambda_{Code\ Inspection}\right|}{\left|\alpha_{Test}\right|} \times \left(\frac{4.86\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(0.9 - 0.81\hat{p}_{Test}\right) - 1.5}{2 \times \left(6\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(1 - 0.9\hat{p}_{Test}\right)\right)} \right)$$

The following are the input variables required and the values for our example:

Notation	Meaning	Value
$\frac{\left\|\lambda_{Code\ Inspection}\right\|}{\left\|\alpha_{Test}\right\|}$	The proportion of defects in the modules that were chosen for inspection.	0.6
$\hat{\varepsilon}_{Customer}$	The effort to find and correct a defect during post-release	50 hrs
$\hat{p}_{Test}$	The effectiveness of testing.	0.5
C	Total project cost	$1 million

The above data assume that the risk assessment approach has identified the high-risk modules with 60% of the total defects, and that only these would go for inspection. There is considerable evidence showing that a large percentage of defects reside in a small proportion of the modules across a wide variety of systems [12, 22, 29, 31], however the exact percentages do vary. In the telecommunications sector, for example, it has been noted that only 10% of modules changed from one release to another contributed to post-release defects on one system; that 80% of the defects came from 20% of the modules based on data from Nortel switches, and that 20% of the modules contain about 40% to 80% of the defects at Alcatel [35]. During the development of the Rational Rose tool, it was found that 77% of source code defects were in subsystems that account for only 21% of the code [38]. During the development of the DOS operating system at IBM it was found that 21% of the modules that had more than one defect accounted for 78% of the total defects [19]. In another IBM operating system, it was noted that 47% of the post-release defects were associated with only 4% of the modules [30]. The 60% default errs on the conservative side of the published examples. The result is as follows:

Notation	Meaning	Value
ROI	ROI expressed as project savings	25.26%
Savings	The effort to find and correct a defect during post-release	$252,590

Figure 12 attempts to provide a more general overview of the benefits. The cost savings can be quite dramatic even under most risk assessment results.

An interesting observation can be made from Figure 13 in that up to 50% of the total project costs can be saved if the post-release costs go higher under the assumption that risk assessment techniques recommend the highest-risk modules for inspection.

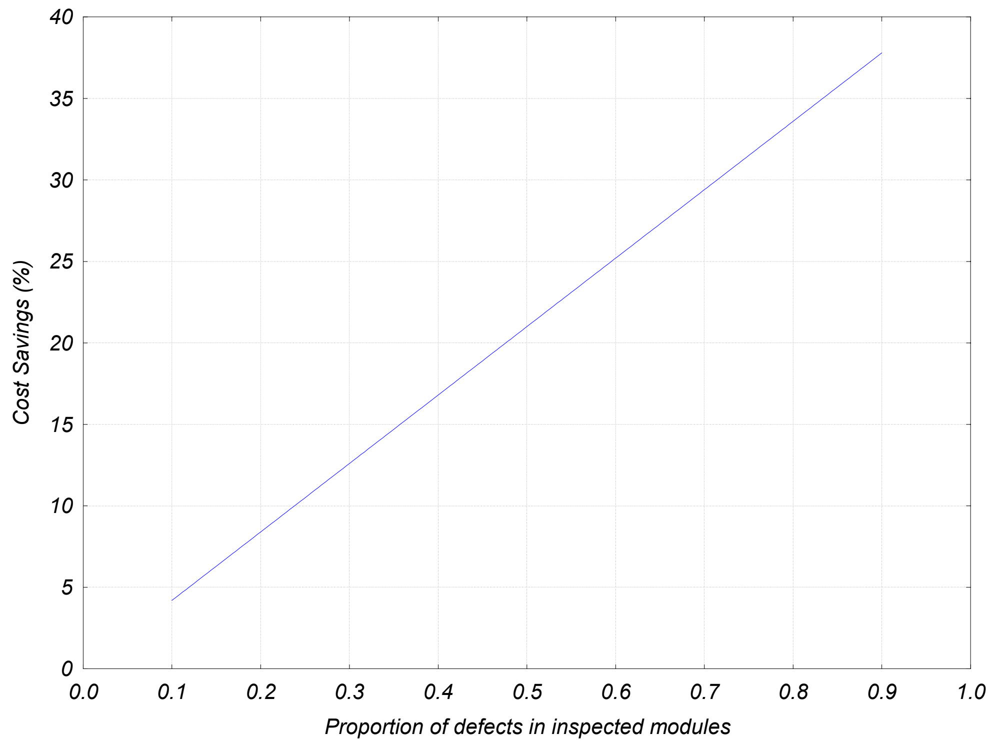


Figure 12: Plot showing the cost savings as the proportion of defects in the inspected modules increases. For example, if say 50% of the defects are in the inspected modules then more than 20% of the project costs would be saved.

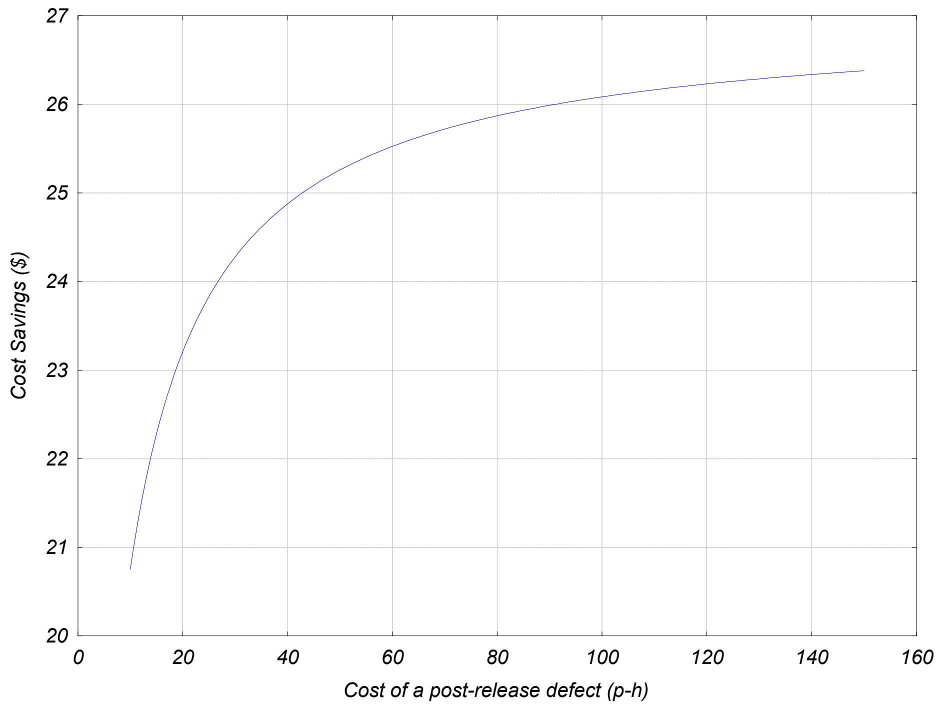


Figure 13: The cost savings vs. the post-release costs per defect increase. Here we assume that risk assessment techniques will result in modules containing 60% of the defects being inspected.

It should be noted that if the risk assessment results are used to drive design inspections, the savings can be even larger.

2.4 Example 4: Higher Reuse

We assume that a system is being re-architected, and that a visualization tool is being used to identify well structured parts of the system with coherent functionality that can be reused. The ROI model for reuse was formulated as shown below (this model takes advantage of default values):

$$ROI = \left(\left(\frac{RLOC}{TLOC} \times 0.8\right) - \frac{\kappa}{C}\right)$$

The following are the input variables required and the values for our example:

Notation	Meaning	Value
$\frac{RLOC}{TLOC}$	The proportion of the system that is reused.	0.1
$\frac{\kappa}{C}$	The proportion of total development budget spent on discovering code that should be reused.	0.05
C	Total project cost	$1 million

The above data assume that only 10% of the system (by code) will be reused, and to identify this 10%, 5% of the total budget is spent. The result is as follows:

Notation	Meaning	Value
ROI	ROI expressed as project savings	3%
Savings	The effort to find and correct a defect during post-release	$30,000

The benefits of reuse technology will depend strongly on the amount of reuse that can be achieved. For example, if a new system is a customization of en existing one, then a majority of the previous system can be reused. One can also conceive of situations where visualization tools can help identify modules that should *not* be reused and developed from scratch.

2.5 Payback Period

When calculating ROI, it is important to understand how long the payback period is going to be. In this section we will explain the relationship between payback period and the ROI models that we have presented.

Conceptually, the relationships among payback, time, and a project's lifetime are shown in Figure 14. Payback is defined in terms of realizing the savings that are calculated according to our ROI models. Some of these savings would be realized before a product is released, and some would be realized only post-release. All the savings will be realized at the end of the payback period, which can be at product release or any period afterwards.

For static analysis techniques that improve reuse, the payback period, as defined in the models here, ends at product release. This is illustrated in Figure 15.

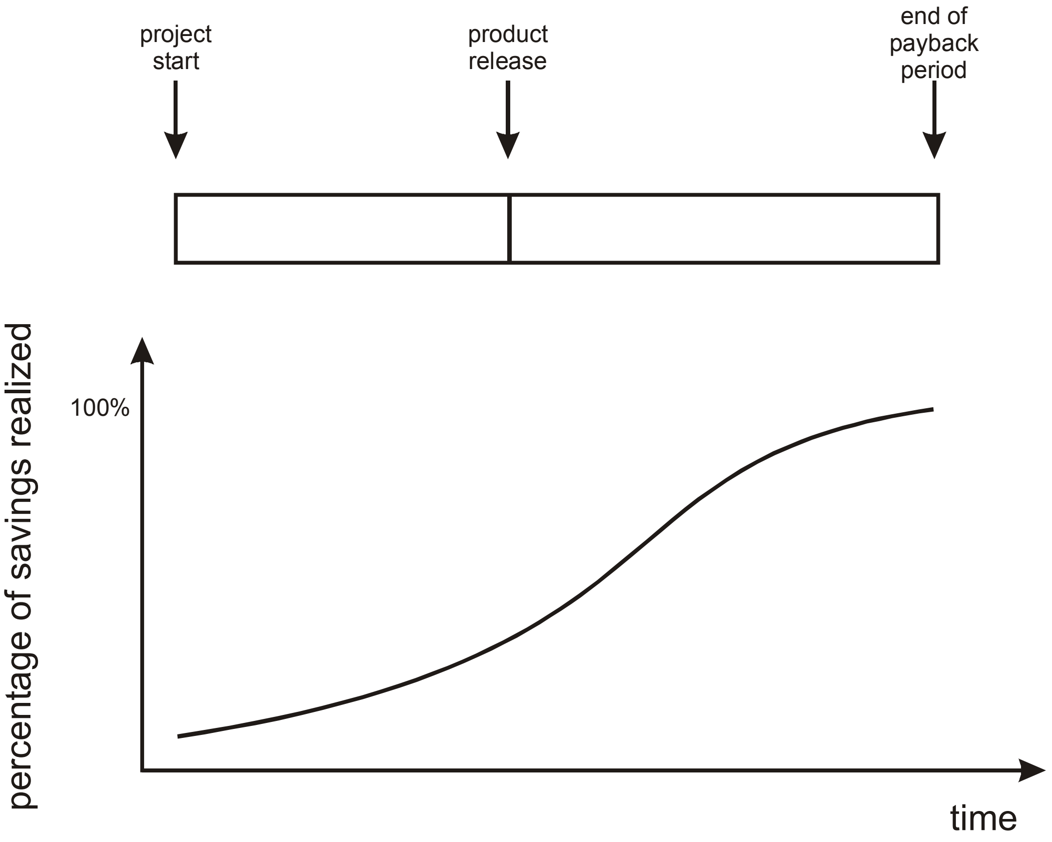


Figure 14: The relationship between payback in terms of savings and the passing of time as this relates to the lifetime of a software project. This pisture applies to static analysis techniques that **reduce rework**.

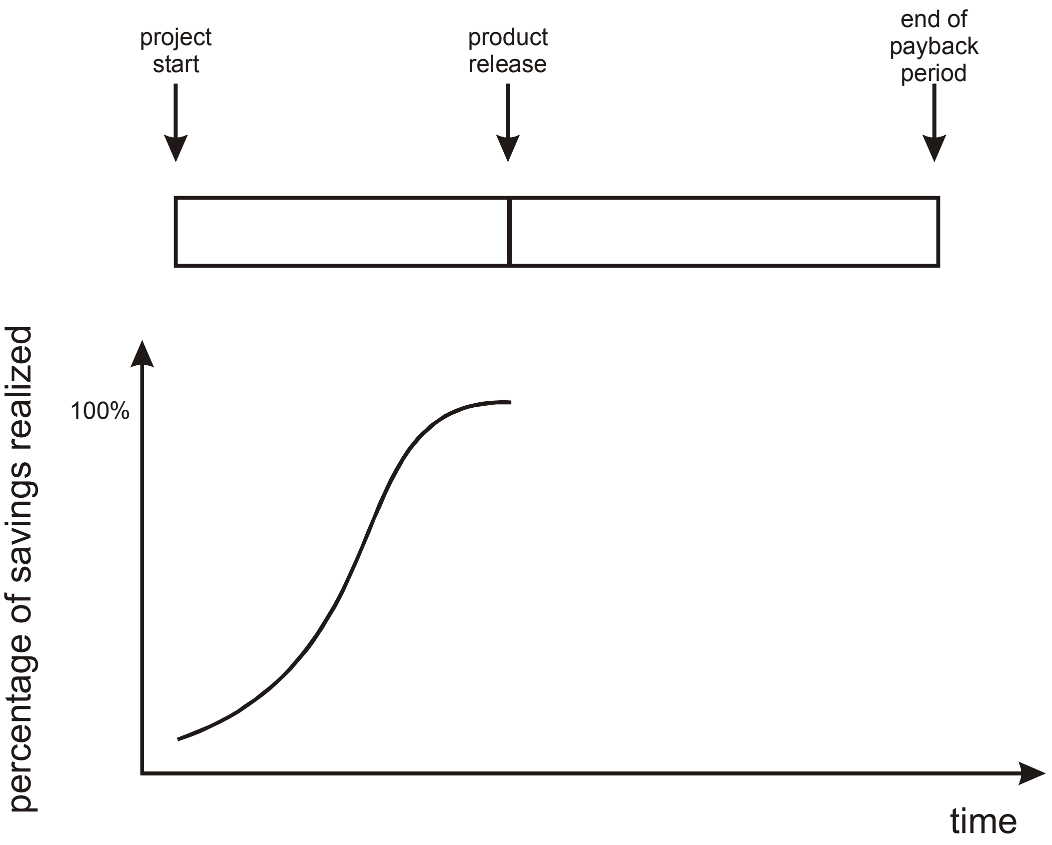


Figure 15: The relationship between payback in terms of savings and the passing of time as this relates to the lifetime of a software project. This pisture applies to static analysis techniques that **increase reuse**.

The end of the payback period is defined implicitly in the ROI models. There are three parameters that determine the payback period:

- The **total cost of the project**. If we define a project with say a 6 month post-release period, then this will have a lower overall cost than a project with a 12 month post-release period. Therefore, the total project cost that is used to convert the percentage savings to a dollar value reflects the project duration.
- The assumption is made implicitly in these models that the **post-release activity will find all of the defects that escape**. Therefore, the effectiveness of the post-release defect detection activity is taken to be one. This would be the case if the post-release period continued indefinitely. However, in practice, the post-release period will be limited, and would vary depending on how long the payback period is defined.

  The relationship between usage of a software product and defect discovery is shown in Figure 16. The effectiveness of the post-release defect detection activity will depend on how long a product is in the field. If the payback period only includes say the first month after release, then the effectiveness of post-release defect detection will be small comparted to a an accounting of six months of post-release time.

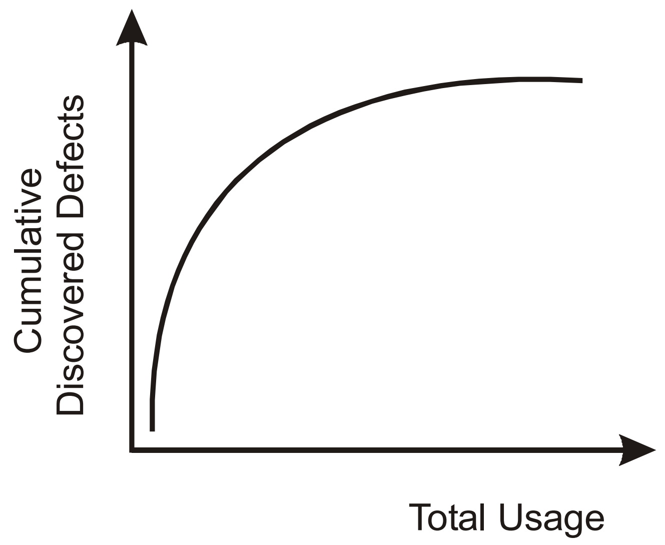


Figure 16: The relationship between defect discovery and the amount of usage that a product receives. Usage is determined by time (how long a product is in the field), and the number of users and installations.

The following new calculations explicitly consider the effectiveness of post-release activities. This is defined as: $\hat{p}'_{Customer}$. First is the **automated defect detection model**:

$$ROI_2 = \hat{p}'_{Auto} \times \left(1 - \frac{\hat{\varepsilon}_{Auto}}{(1-\beta)\left(\hat{\varepsilon}_{Test}\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\hat{p}'_{Customer}\left(1-\hat{p}'_{Test}\right)\right)} \right) \times \frac{C_3 + C_4 + C_5}{C}$$

Second is a new version of the **maintenance efficiency model**:

$$ROI_2 = \left(1 - \frac{\left(\hat{\varepsilon}_{Test} \times \hat{p}_{Test}\right) + \left(\hat{\varepsilon}'_{Customer}\hat{p}'_{Customer} \times g \times \left(1 - \left(\hat{p}'_{Test} \times g\right)\right)\right)}{\left(\hat{\varepsilon}_{Test} \times \hat{p}_{Test}\right) + \left(\hat{\varepsilon}'_{Customer}\hat{p}'_{Customer} \times \left(1 - \hat{p}'_{Test}\right)\right)} \right) \times \frac{C_3 + C_4 + C_5}{C}$$

The final model is for **risk assessment**:

$$ROI_2 = \frac{\left|\lambda_{Code\ Inspection}\right|}{\left|\alpha_{Test}\right|} \times \left(\frac{\hat{\varepsilon}_{Test}\hat{p}'_{Test}\left(1-\beta\right) + \hat{\varepsilon}'_{Customer}\hat{p}'_{Customer}\left(1-\hat{p}'_{Test}\right)\left(1-\beta\right) - \hat{\varepsilon}_{Code\ Inspection}}{\hat{\varepsilon}_{Test}\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\hat{p}'_{Customer}\left(1-\hat{p}'_{Test}\right)} \right) \times \frac{C_3 + C_4 + C_5}{C}$$

The reuse model is not affected by this because its scope is limited to development.

- The final factor that defines the payback period is **the cost of finding and fixing a defect post-release**. However, it is reasonable to assume that this will not change dramatically over time after a product is released, and thus can be considered to be invariant.[3]

The logic for determining the effectiveness of post-release activities at various points in time (i.e., for different payback periods) can be illustrated through an example. Let's say that, based on historical data, the lifetime of a product is five years, and that 80% of the post-release defects are found in the first year after release. If we want the payback period to be product development plus one year post-release, then the effectiveness of post-release activities is 0.8 rather than 1. We would then use the new equations above to compute ROI with the value of $\hat{p}'_{Customer}$ included.

3 Conclusions

This report formulated in detail a number of Return-on-Investment models for different static analysis techniques. The models were instantiated and illustrated for each of the techniques. Default values based on an extensive literature review were used to demonstrate how the models can be used to make the business case for static analysis.

A series of basic conclusions become evident from this exposition:

- It is possible to formulate strongly justifiable ROI models for static analysis techniques.
- The ROI from static analysis techniques that support risk assessment can have a dramatic impact on project costs, with savings as high as 25% of project costs under modest assumptions.
- Combining multiple static analysis approaches on a single project can reduce overall project costs by up to 40% under rather modest assumptions.
- The ROI from techniques that automatically detect defects and that reduce bad fixes are modest. However, for large projects a few percentage points savings can still be a large amount of money, and therefore these should not be dismissed.
- The ROI from static analysis techniques that can help identify reusable code are also modest, but may be worthwhile under some circumstances (e.g., the code to be reused is modular but very complex).

In summary, the biggest benefit can be gained from focused risk assessment techniques. These use metrics to find the highest risk modules in a system and inspection effort is targeted at these modules. For most large systems, inspecting all of the code is not a realistic option. Risk assessment provides a pragmatic solution to this with significant rewards.

Automatic defect detection and reductions in the cost of changes can also shave additional percentage points off software projects, and should be seriously considered for large projects.

[3] After release, costs are expected to be high due to rediscoveries. This results in many duplicate problem reports being opened by customers. However, rediscoveries, if they can be detected before they make it to the development organization, will not influence find and fix costs.

The benefits of static analysis for reuse will depend strongly on how much of a system can be reused. But, there is certainly the potential for large savings there as well.

4 Appendix A: Definitions

The following are definitions of terms that and explanations of concepts that are used throughout this report.

A **failure** is an incorrect result produced by the software. It may be incorrect according to a requirement specification, a design, or because it does not meet customer expectations. A failure may be functional (i.e., the software does not implement a feature properly) or a failure in performance (e.g., the software is too slow). Failures can only occur in executable software.[4]

A failure is caused by one or more **faults** or **defects**. We use these two terms interchangeably throughout.

Of course, a failure may be caused by multiple faults, not just one. However, a fault does not guarantee that a failure will occur. For instance, some faults are never discovered. Some faults are minor and never result in a visible failure (e.g., efficiency or problems with the tenth most significant digit). Some faults never manifest themselves because that part of the code is never executed (e.g., code that deals with a very rare condition).

Some failure occur but are not observed. For example, on some aircraft there are multiple computers running in parallel with each one having independently produced software. A failure may occur in one of the computers, but this does not result in an overall system failure because the other computers produce the correct result.

A fault is an incorrect step, process, or data in the software. When say a programmer fixes a piece of software, s/he is fixing defects. We cannot really count the true number of defects in the software because some defects may never be discovered. We can only count discovered defects. Therefore, to count defects we usually count the number of fixes.

An **error** is a human action that causes a defect to be inserted in the software. A single error may lead to many defects being inserted in the software. Understanding errors is important because if the frequency of a certain type of error can be reduced then the number of defects prevented can be substantial.

Once software is released, either to customers internal or external to the organization, then **Problem Reports** (PR) may be opened. A PR is a description of a failure. Different organizations will use alternative terminology, but almost all software organizations will have a problem reporting process. A problem report reflects a single instance of a perceived failure. This means that a problem report may not necessarily be a true failure. If there are many customers, then there may be many problem reports describing the same failure.

A **rediscovery** is when a someone identifies a problem report, failure, or defect that has already been discovered before. For instance, assume

[4] For some types of systems, a simulation environment may exist to simulate various parts of the design. In such a case you may have a failure in the design.

that a customer report results in creating a problem report. If another customer finds the same problem then this is a rediscovery. Similarly, if a tester detects a failure that has already been detected by another tester, then this is a rediscovery. Rediscoveries are expensive because effort is spent in making the rediscovery and in deciding that the incident is a rediscovery. For example, for problem reports a nontrivial amount of the support organization's effort is spent matching problem reports to determine if a new problem report is similar to a previous one.

When a defect is fixed, the fix itself may introduce one or more new defects. This is called a **bad fix**. Depending on the schedule pressure, the code structure, and the quality of the existing test suite, bad fixes may actually be a major problem by themselves.

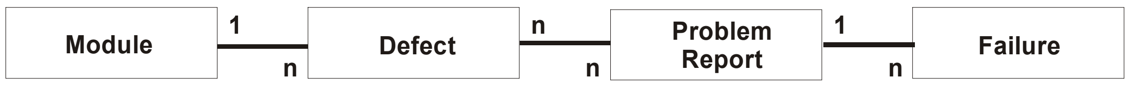


Figure 17: Basic Entity-Relationship model showing the relationship between failures, defects, and modules.

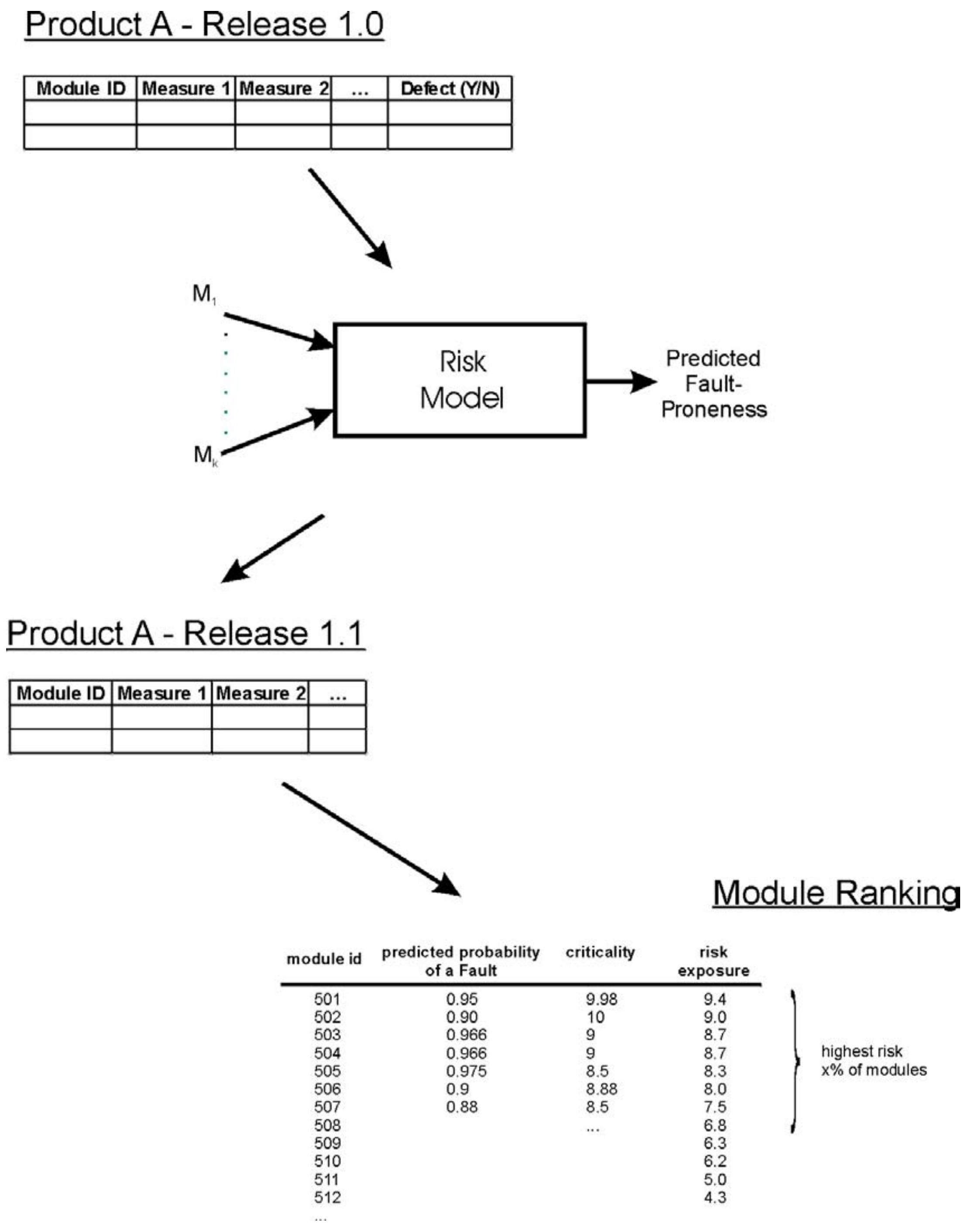
In many organizations whenever a failure is observed (through testing or from a customer) a Problem Report (PR) is opened. It is useful to distinguish between a failure and a failure instance. Consider the relationships in Figure 17. This shows that each failure instance may be associated with only a single PR, but that a PR can be associated with multiple failure instances. Multiple failure instances may be the same failure, but detected by multiple users, and hence they would all be matched to a single PR. For example, if the program cannot open files with long file names. This is a problem that may be experienced by multiple users and so there will be multiple failure instances. A single problem report would be opened for all of these failure instances explaining that the program cannot save files with long names. This single PR may have many rediscoveries associated with it.

A defect occurs in a single module, and each module may have multiple defects. A PR can be associated with multiple defects, possibly in the same module or across multiple modules.

5 Appendix B: A Methodology for Risk Management Using Metrics

Static metrics collected during a software project can be used to identify high-risk modules or components. This section describes a methodology that employs metrics to identify such high-risk modules.

An illustration of this methodology is given in Figure 18. The first step is to collect the static metrics from an earlier release. These metrics are then used as the basis for risk assessment in current and future releases.



Module ID	Measure 1	Measure 2	...	Defect (Y/N)

Module ID	Measure 1	Measure 2	...

module id	predicted probability of a Fault	criticality	risk exposure
501	0.95	9.98	9.4
502	0.90	10	9.0
503	0.966	9	8.7
504	0.966	9	8.7
505	0.975	8.5	8.3
506	0.9	8.88	8.0
507	0.88	8.5	7.5
508		...	6.8
509			6.3
510			6.2
511			5.0
512			4.3
...			

Figure 18: Overview of the risk assessment methodology.

To perform risk assessment using metrics it is necessary to build a *risk model*. Figure 18 illustrates where the risk model fits into the risk assessment process. A risk model is a statistical model that can be used to predict the probability that a module will have a post-release defect. As shown in the figure, one typically uses historical data to build such a model. The historical data may come from a previous release of the same product or even from another similar product within the same organization. The size, previous defects (say testing defects), and coupling metrics are collected from that earlier release, as well as data on the incidence of post-release defects (say within the first 6-12 months after release). Once this data is collected a risk model can be built.

A risk model is typically statistical model relating the metrics to the probability of a defect. Although many different statistical modeling techniques can be used, we have always obtained good results with a technique called logistic regression, which is well suited to this kind of modeling problem.

The risk model can then be used to predict the probability of a post-release defect for the current and subsequent releases. In the example in Figure 18 we assume that the risk model is built from release n and we want to identify the high-risk modules for release n+1. Once the risk model is built from the release n data, the same metrics are collected from release n+1. The release n+1 data excludes post-release defects because we do not know that (if we did we would not need a risk model). The new data is entered into the risk model which then comes out with the predicted probability.

The probability is a number between 0 and 1. The modules are then ranked by their predicted probability of a post-release defect, as shown in Figure 18. The top x% are considered the highest risk modules.

The risk assessment approach so far does not take into account the business importance of the modules. Some modules perform critical functions for the software. If these modules have a defect then the whole system will not function as expected. Even if the probability of a defect for these critical modules is relatively small, it is still important to pay attention to them. Similarly, some modules may have a high probability of a post-release defect but they perform very auxiliary functions. So if we have to make a choice, the low criticality modules would not be given much attention.

Criticality is a business decision. In our practice we usually use a 1 to 10 scale to assign criticality to the modules. For large systems this is not plausible. So use cases or requirements scenarios are assigned criticality values. Use cases are frequently used to describe "features" or a coherent set of functionality that the software is expected to perform. Some of these use cases have a very high business value. The project team would assign criticality values to the use cases on the 1 to 10 scale. Modules that implement highly critical scenarios are then considered to be very critical. So we essentially trace from the business requirements to the modules, see which modules implement the use case, and assign the module the same criticality as the use case. Since a single module may implement many use cases, some summary measure such as the average criticality may be used for each module. This process is much easier if a design tool is used to document use cases.

Multiplying the probability of a post-release defect with the criticality for each module gives the *risk exposure*. This is a summary measure that can be used to rank modules. The big advantage of risk exposure is that it captures the quality of the module as well as its business value. Ranking by risk exposure is illustrated in Figure 18.

The top x% of the modules are then the high-risk modules. Depending on the scenario you are in, these modules are then inspected, or tested first.

One example of where this was applied is a project that was being developed in Java and used UML design models. The project manager wanted to start design inspections but he did not have the resources to inspect everything. So a risk model was developed from a previous release and was used to identify the high-risk classes for the current release. The warranty cost savings for this strategy were calculated as shown in Figure 19. Here we see that if the highest-risk 20% of the classes were inspected, the warranty cost savings would be around 42%. This means that the warranty costs would be reduced by 42% of what they would have been had no inspections been conducted.

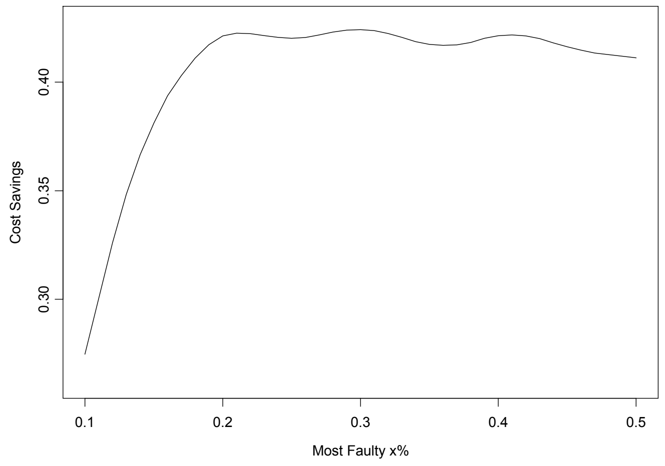


Figure 19: Warranty cost savings as the number of classes inspected rises.

The above methodology works whether one is using object-oriented techniques or structured techniques. It also works well irrespective of the size of the project.

6 Appendix C: ROI Models for Static Analysis

This appendix describes the details of the ROI models that we use for calculating ROI, as well as justifications for the choices made. We consider ROI for a whole project, taking into account pre- and post-release costs.

6.1 Definitions

6.1.1 Total Project Cost

Following our earlier breakdown of project costs, we define the individual costs for a project as follows:

Cost Item	Notation
Fixed & Overhead Costs	C_1
Construction Costs	C_2
Defect Detection Costs	C_3
Pre-release Rework Costs	C_4
Post-release Rework Costs	C_5
New Feature Costs	C_6
Total Life Cycle Costs	$C = C_1 + C_2 + C_3 + C_4 + C_5 + C_6$

All costs, except the fixed and overhead costs, are really made up of effort (i.e., labor costs).

6.1.2 Defect Detection Costs

To formulate ROI properly, it is also necessary that we breakdown the costs of defect detection activities. The following is the notation that we will use to define the individual effort items for each instance of a defect detection activity. For example, if we are talking about inspections, then this is the effort for a single inspection.

A project will have multiple instances of a defect detection activity. For instance, a project will have many code inspections. We use I to denote a specific instance of the inspection.

Effort Definition	Notation
Effort to find a defect	$E_{1,i}$
Effort to isolate a defect	$E_{2,i}$
Effort to fix a defect	$E_{3,i}$
Effort to finalize a defect (retesting, documentation, and packaging)	$E_{4,i}$
Effort to find and fix a defect	$E_{f,i} = E_{1,i} + E_{2,i} + E_{3,i} + E_{4,i}$

For specific defect detection activities, some of the above effort values may be zero or very small. For instance, during a code inspection the isolation effort would be close to zero since the inspection itself identifies the exact location of the defect.

It is also necessary to consider the isolation effort separately when considering defect detection effort. Let us say that out of the total correction effort, $E_{f,i}$, a certain proportion is isolation effort. We will denote this proportion as θ. Therefore, we can say that isolation effort is given by:

$$E_{2,i} = \theta \times E_{f,i}$$

The effort that Is not spent on isolation is given by $(1-\theta)\times E_{f,i}$. This will become handy later on.

6.1.3 Defect Counts

An important definition when evaluating reduction in rework is of the number of defects that exist in a document (e.g., a design or source code) prior to any defect detection activity:

$$\alpha_{f,i} = \left\{ x \middle| \text{x is a defect that exists in the document prior to defect detection} \right\}$$

This applies to a specific instance of a defect detection activity: instance i. The total number of defects is therefore given by the size of this set: $\left|\alpha_{f,i}\right|$.

The actual defect detection activity will also find a certain number of defects. This is defined as:

$$\lambda_{f,i} = \left\{ x \middle| \text{x is a defect that was found in the document during defect detection} \right\}$$

This applies to a specific instance of a defect detection activity: instance i. The total number of defects found is therefore given by the size of this set: $\left|\lambda_{f,i}\right|$.

Two important measures that characterize the defect detection activity need to be defined. The first is the effectiveness of defect detection. For a single instance of a defect detection activity (instance i) this is given by:

$$p_{f,i} = \frac{\left|\lambda_{f,i}\right|}{\left|\alpha_{f,i}\right|}$$

This gives the effectiveness for a single instance. Effectiveness is really the fraction of defects in the document that were found. An effectiveness of 0.5 means that half of the defects in the document were found during defect detection. For example, it would give the effectiveness of a single individual code inspection.

For all of the defect detection activities on a project (across all instances), we take the average:

$$\hat{p}_f = \overline{\left(\frac{\left|\lambda_{f,i}\right|}{\left|\alpha_{f,i}\right|}\right)}$$

Therefore, where f = "code inspections", then $\hat{p}_{code\ inspections}$ is the average effectiveness of all code inspections on the project.

6.1.4 Defect Detection Effort

The second important measure is the effort it takes to find and fix a defect:

$$\varepsilon_{f,i} = \frac{E_{f,i}}{\left|\lambda_{f,i}\right|}$$

This gives the effort per defect for a single instance i of the defect detection activity. For example, if the value is 2 hours for a given code inspection, then this means that it took two hours, on average, to find and fix a single defect during the code inspection.

For all of the defect detection activities on a project, we take the average:

$$\hat{\varepsilon}_f = \overline{\left(\frac{E_{f,i}}{\left|\lambda_{f,i}\right|}\right)}$$

Therefore, where f = “code inspections”, then $\hat{\varepsilon}_{code\ inspections}$ is the average (find and fix) effort per defect of all code inspections on the project.

The above formulations do not take into account the possibility that some of the fixes are bad fixes. We therefore have to account for the bad fix ratio, β . Let’s say if β is 0.1, this means that 1 in 10 fixes are bad. The proportion of correct fixes is given by $1-\beta$. Now we have the following for effectiveness:

$$\hat{p}'_f = \overline{\left(\frac{\left|\lambda_{f,i}\right|}{\left|\alpha_{f,i}\right|}\right)}(1-\beta)$$

And for the cost per defect we have:

$$\hat{\varepsilon}'_f = \overline{\left(\frac{E_{f,i}}{\left|\lambda_{f,i}\right|}\right)}\left(\frac{1}{1-\beta}\right)$$

This is effort per defect corrected for bad fixes. The more bad fixes the higher the effort per defect, on average.

6.1.5 Return on Investment Definitions

There are a number of different models that can be used to evaluate ROI for static analysis. We will explore two of them. The first is the most common ROI model. We will show that this model is not appropriate because it does not accurately account for the benefits of investments in software projects. We subsequently present the second model which we aregue is much more appropriate. The models here are presented at a rather conceptual level. Later in this appendix we will formulate the chosen ROI model precisely.

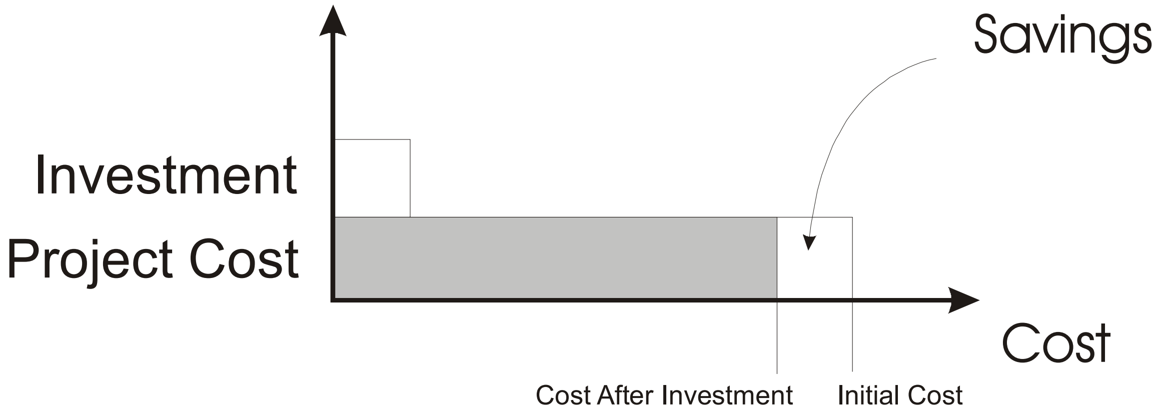


Figure 20: Definitions of ROI concepts.

Consider the diagram in Figure 20. The lower bar shows the cost of a software project. The top bar shows the cost of an investment in the project. The investment could be the use of static analysis techniques, for example.

After the investment in the project, the project cost goes down. So now the shaded area is the new project cost. The savings are marked in the figure. The question is whether that investment was worth the savings ?

The most common ROI model, and that has been used more often than not in software engineering, is shown below:

$$ROI_1 = \frac{Costs\,Saved - Cost\,Consumed}{Cost\,Consumed}$$

This ROI model gives how much the savings gained from the project were compared to the initial investment. Let us look at a couple of examples to show how this model works.

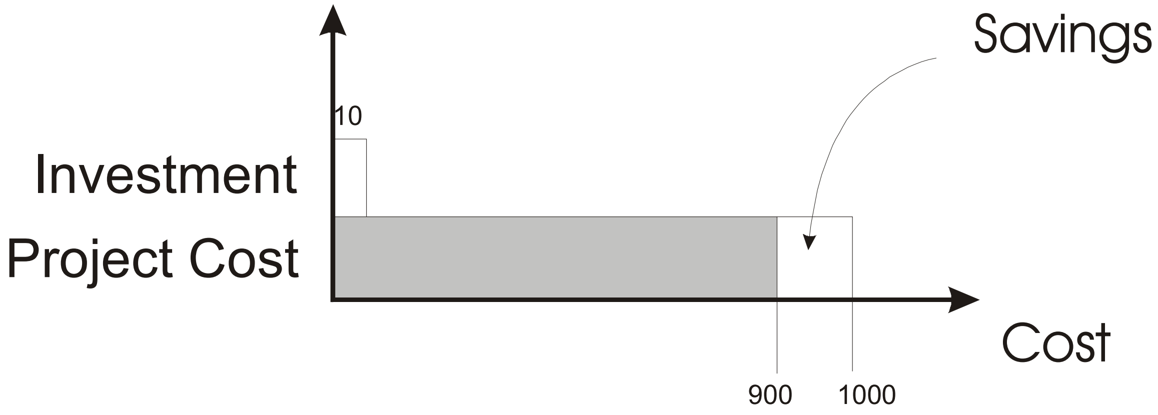


Figure 21: Example of investment A.

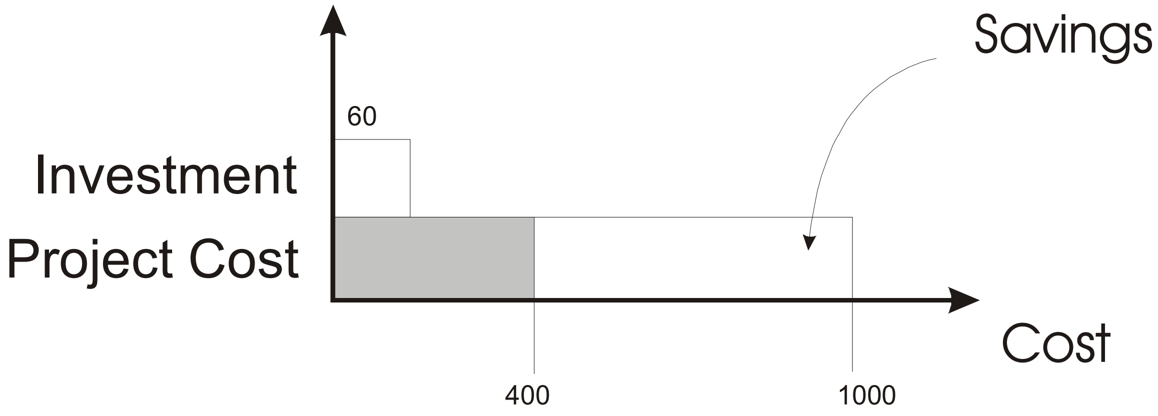


Figure 22: Example of investment B.

Let us assume that we have a software project that costs 1000 units. This is the total cost for the project, including 2 years of maintenance. In Figure 21 we have an example of a specific investment. The investment was of 10 units. The investment was in some techniques to improve the processes and quality of the product. The benefits gained were a reduction of the total project cost to 900 units. According to the traditional ROI model, the ROI for this investment is:

$$ROI_1 = \frac{100 - 10}{10} = 9$$

Now consider the second project in Figure 22. If you perform the same calculations for project B you will see that the ROI for project B is also 9. However, the savings for project B were dramatic: 4 times more savings than for project A. But the ROI is exactly the same. By most accounts, one would much prefer to have a project B and the ROI should reflect that.

Therefore, we do not use the traditional ROI calculations. Rather, we formulate an alternative model that is based on Kusumoto's work [26]. The model is given as follows:

$$ROI_2 = \frac{Costs\,Saved - Cost\,Consumed}{Original\,Cost}$$

This ROI model tracks the benefits of investments very well. It is interpreted as the overall project savings from the investment (i.e., how much money is saved during the project from the investment). For project A the ROI is then calculated as:

$$ROI_2 = \frac{100 - 10}{1000} = 0.09$$

This means that in project A, the investment only saved 9% of overall project cost. Now for project B we have:

$$ROI_2 = \frac{600-60}{1000} = 0.54$$

Therefore, project B had a savings of 54% from its original cost.

To make this concrete, if the original project cost was \$1 million. Then after investment B of \$60,000, the project would cost \$600,000, a \$400,000 reduction.

Kusumoto's basic model has been expanded and used to evaluate the benefits of various software engineering technologies, such as inspections and risk assessment [6, 7, 15]. In essence, this model has received acceptance in the software engineering sceintific community as a valid way to evaluate ROI.

6.2 The ROI Model

Below we start putting the whole ROI model together with all the appropriate parameterizations to account for the benefits of static analysis.

6.2.1 Overall Model

Reduced rework is due to defect detection activities. These can be inspections, testing, and static analysis. We assume that there are n consecutive defect detection activities, for example, design inspections, code inspections, testing, and post-release. These will be numbered sequentially from 1 to n.

Another factor that needs to be accounted for is reuse. We consider reuse in the same manner: as a saving.

The savings would be given by:

$$ROI_2 = \left(\frac{\left(\sum_{t=2}^{n} \theta_t \times \hat{p}'_t \times \gamma_t \right) - \left(\hat{\varepsilon}_S \times |\lambda_S| \right)}{\sum_{j=2}^{n} \varphi_j \times \chi_j} \times \frac{C_3 + C_4 + C_5}{C} \right) + \left(\left(\frac{RLOC}{TLOC} \times \left(1 - RCR\right) \right) - \frac{\kappa}{C} \right)$$

where:

$$\gamma_t = \begin{cases} |\lambda_1| \times (1-\beta) & ,t=2 \\ |\lambda_1| \times (1-\beta) \times \prod_{k=2}^{t-1} \left(1-\hat{p}'_k\right) & ,t>2 \end{cases}$$

and:

$$\chi_j = \begin{cases} \hat{p}_j \times |\alpha_1| & ,j=2 \\ \hat{p}'_j \times |\alpha_1| \times \prod_{k=2}^{j-1} \left(1-\hat{p}'_k\right) & ,j>2 \end{cases}$$

and:

$$\theta_t = \begin{cases} \hat{\varepsilon}_t & ,t \neq n \\ \hat{\varepsilon}_t' & ,t = n \end{cases}$$

Which accounts for the fact that the last phase, phase n, will have a higher cost because it finds all defects and must account for bad fixes within that (i.e., the bad fixes do not introduce defects that are passed on to the next phase). And similarly:

$$\varphi_j = \begin{cases} \hat{\varepsilon}_j & ,j \neq n \\ \hat{\varepsilon}_j' & ,j = n \end{cases}$$

The ROI model has two terms. The first concerns the ROI from reducing rework costs. The second term concerns productivity benefits from reuse.

To complete the model, the following definitions are also required:

Notation	**Meaning**
$RLOC$	Lines of code that will be reused
RCR	Relative Cost of Reuse
$TLOC$	Total size of the system

These definition cover the second term related to reuse. In practice, all that is needed is $\frac{RLOC}{TLOC}$ ratio, which is the proportion of the total system that would be reused.

The $\left(\hat{\varepsilon}_S' \times |\lambda_S|\right)$ term reflects the effort spent on the static analysis to reduce rework effort. For example, if the static analysis was the automated detection of defects, then this term is the effort to fix the defects that were found.

The RCR value captures the notion that reuse is cheaper than development from scratch. It quantifies the relative cost of reuse compared to development. RCR is typically used when evaluating the benefits of software reuse [32].

The κ value is the cost of doing the static analysis *for reuse*. It is expressed as $\frac{\kappa}{C}$, which means we can express it as a proportion of the total project budget.

If a particular investment does not result in reuse benefits, then the second term would be zero since $RLOC$ would be zero and κ would also be zero.

The outcome of this ROI model is a proportion of the original cost of the project (without static analysis) that would be saved if static analysis techniques were used. For example, if the ROI_2 value was 0.2, this means that 20% of the project cost is saved. If the project cost was estimated at \$1 million, then the savings would be \$200,000.

6.2.2 Defaults

Although the ROI model may seem overwhelming initially, it is actually straightforward to use. The model does not require information that would not normally be available to a project manager.

One initial set of simplifications are the defaults. Below we present a series of defaults that can be used.

Since ROI is comparing the project without static analysis to the project with static analysis, we will talk about the No Static Analysis project (NSA) and the Static Analysis (SA) project for short.

The first default is for the expression:

$$\frac{C_3 + C_4 + C_5}{C}$$

This actually is the proportion of total development effort that is rework. It is well documented that over half, and in some instances approaching 90%, of total project effort is spent on testing (for a wide variety of application domains) [4, 18, 21]. A large proportion of this testing effort is spent on fixing defects that have been discovered. It is safe to assume that in most NSA projects this is around 0.5 since rework related activities typically account for about half of a project's budget. One published report notes that 44% of total development project costs is rework [40]. Other data shows ranges of rework from 20% to as high as 80% [35].[5] If we add to this the post-release costs, then 50% of total costs being devoted to rework is a rather conservative value.

Another default concerns the effectiveness of inspections. A comprehensive literature review (this is presented in the appendix in Section 7) has found that the average effectiveness of design and code inspections is 0.57 (i.e., inspections tend to find, on average, 57% of the defects in the artifacts that are inspected). Thus, the value for:

$$\hat{p}_{design\ inspection} = \overline{\left(\frac{|\lambda_{f,i}|}{|\alpha_{f,i}|}\right)} = 0.57$$

and :

$$\hat{p}_{code\ inspection} = \overline{\left(\frac{|\lambda_{f,i}|}{|\alpha_{f,i}|}\right)} = 0.57$$

We use as the default for β the value 0.1 (i.e., about 1 in ten fixes introduce new defects). Fagan [11] notes that 1 in six fixes in an inspection introduces another defect. This number is likely larger for testing and post-release since only symptoms are observed during these

[5] The higher percentage being more typical in less mature (in the sense of teh CMM for Software) organizations.

stages of a project. We err on the conservative side by choosing a value of 1 in ten fixes.

Again, based on a comprehensive review of published industrial data (this is presented in the appendix in Section 7), we estimate that the average effort to find and correct a defect during inspections is:

Effort per Defect	**Value**
$\hat{\varepsilon}_{f=code\ inspection} = \overline{\left(\frac{E_{f,i}}{\left\|\lambda_{f,i}\right\|}\right)}$	1.5 hours
$\hat{\varepsilon}_{f=design\ inspection} = \overline{\left(\frac{E_{f,i}}{\left\|\lambda_{f,i}\right\|}\right)}$	1.5 hours
$\hat{\varepsilon}_{f=testing} = \overline{\left(\frac{E_{f,i}}{\left\|\lambda_{f,i}\right\|}\right)}$	6 hours

The RCR value indicates how much reuse costs as a proportion of new development cost. This value is estimated to be around 0.2. This means that reuse costs only 20% of new development costs, on average. This value is obtained from [32].

6.2.3 Illustration

To help provide some context for this ROI model, we will look at some examples.

Activity	**Number**
Testing	1
Warranty / Customer	2

Let us consider a project that has two phases where defects are detected: testing and post-release. We will denote these as phases 1 and 2. The phase numbers are used as indices in our model.

The cost of defect detection for this project is given by:

$$\overbrace{\left(\hat{\varepsilon}_{Test} \times \hat{p}_{Test} \times \left|\alpha_{Test}\right|\right)}^{actual\ cost\ of\ testing} + \underbrace{\left(\hat{\varepsilon}'_{Customer} \times \left|\alpha_{Test}\right| \times \left(1 - \hat{p}'_{Test}\right)\right)}_{actual\ warranty\ cost}$$

where $\left|\alpha_{Test}\right|$ is the total number of defects that enter into testing. We multiply this by the average cost of testing per defect and the effectiveness of testing. This gives us the total testing cost.

For warranty costs, we first determine how many defects will escape from testing. This is the $\left(1-\hat{p}'_{Test}\right)$ value. We multiply this by the average cost of defect detection and correction per defect post-release. We assume that after a product is released we will find the remaining detectable defects. Hence, the effectiveness of post-release is one (i.e., all detectable defects will be found eventually).

Now let us consider the situation whereby we add code inspections to this project. We wish to evaluate the savings that could potentially be gained from introducing this new technique. We can define the project as follows:

Activity	**Number**
Inspections	1
Testing	2
Warranty / Customer	3

The ROI for introducing inspections in a testing only life cycle is given by:

$$\frac{\overbrace{\left(\hat{\varepsilon}_{Test}\times\hat{p}'_{Test}\times\left|\lambda_{Inspection}\right|\times(1-\beta)\right)}^{\text{cost savings in testing from inspections}}+\overbrace{\left(\hat{\varepsilon}'_{Customer}\times\left|\lambda_{Inspection}\right|\times(1-\beta)\times\left(1-\hat{p}'_{Test}\right)\right)}^{\text{cost savings in warranty from inspections}}-\overbrace{\left(\hat{\varepsilon}_{Inspection}\times\left|\lambda_{Inspection}\right|\right)}^{\text{cost of inspections}}}{\underbrace{\left(\hat{\varepsilon}_{Test}\times\hat{p}_{Test}\times\left|\alpha_{Test}\right|\right)+\left(\hat{\varepsilon}'_{Customer}\times\left|\alpha_{Test}\right|\times\left(1-\hat{p}'_{Test}\right)\right)}_{\text{actual cost of testing and warranty without inspections}}}$$

$$\times\frac{\overbrace{C_3+C_4+C_5}^{\substack{\text{proportion of project}\\ \text{devoted to rework}}}}{C}$$

At the top we see the two elements of savings minus the cost of doing the actual inspections. The denominator of the first term is the actual cost of the project.

6.3 Example Instantiations

Depending on the type of static analysis performed and the nature of the development process, the ROI model can be instantiated differently. In this subsection we will look at specific instantiations.

6.3.1 Automated Defect Detection Model

The model presented below is useful when we want to evaluate the benefits of automated detection of defects using static analysis, for example, by identifying potential logic problems.

For the sake of this model, we will assume that the development process only has testing and post-release as the main defect detection activities. There are no code nor design inspections.

Automated defect detection will be implemented before testing. Defects that are detected that way are corrected.

The ROI model in this case is then given by:

$$ROI_2 = \hat{p}'_{Auto} \times \left(1 - \frac{\hat{\varepsilon}_{Auto}}{(1-\beta)\left(\hat{\varepsilon}_{Test}\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(1-\hat{p}'_{Test}\right)\right)}\right) \times \frac{C_3 + C_4 + C_5}{C}$$

where:

Notation	Meaning
$\hat{p}'_{Auto}$	The effectiveness of automatic detection of defects, taking into account bad fixes.
ε_{Auto}	The effort to find and correct a defect during automatic detection
ε_{Test}	The effort to find and correct a defect during testing
$\varepsilon_{Customer}$	The effort to find and correct a defect during post-release
$\hat{p}'_{Test}$	The effectiveness of testing, taking into account bad fixes. This can be given as (using default values): $\hat{p}_{Test} \times (1-\beta) = 0.9 \times \hat{p}_{Test}$
$\hat{p}_{Test}$	The effectiveness of testing, without taking into account bad fixes.

If we include some of our defaults, then we have a simplified model as follows:

$$ROI_2 = \hat{p}'_{Auto} \times \left(0.5 - \frac{0.55\hat{\varepsilon}_{Auto}}{\left(6\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(1-\left(0.9 \times \hat{p}_{Test}\right)\right)\right)}\right)$$

6.3.2 Improved Maintenance Efficiency Model

This model assumes that static analysis will be used to perform an impact analysis during changes to fix defects. The static analysis allows a programmer to visualize the relationships among modules in a system to see what would be affected by a change. This information can reduce the inadvertent introduction of defects when making a change.

We will assume that the project has testing and post-release as the two main activities for defect detection. Improvements in impact analysis will result in a reduction in the number of bad fixes. Therefore, changes in testing and post-release will have fewer bad fixes.

The logic of this model is that the cost of fixes does not change by this type of static analysis technique. However, the effectiveness of testing and post-release changes does increase. For instance, the effectiveness of testing changes from:

$$\hat{p}'_{Test}$$

to

$$\hat{p}'_{Test} \times g$$

where g is the increase in effectiveness due to a reduction in bad fixes. We define the good fix ratio as $(1-\beta)$, which is the proportion of fixes that do not introduce defects. The g indicates how much this increases using the impact analysis capabilities. For instance, if g is 1.1, this means that the proportion of good fixes increases by 10%. The proportion of good fixes after using static analysis is thus given by:

$$(1-\beta)g$$

where:

$$(1-\beta)g = \begin{cases} (1-\beta)g & ,(1-\beta)g \le 1 \\ 1 & ,(1-\beta)g > 1 \end{cases}$$

Based on our default value for value of β, this imposes a pragmatic maximum value or ceiling on g of 1.1.

The model itself is formulated as:

$$\frac{\text{cost of not using visualization - cost of using visualization}}{\text{cost of not using visualization}}$$

The final model then is expressed as:

$$ROI_2 = \left(1 - \frac{(\hat{\varepsilon}_{Test} \times \hat{p}_{Test}) + \left(\hat{\varepsilon}'_{Customer} \times \left(1 - (\hat{p}'_{Test} \times g)\right)\right)}{(\hat{\varepsilon}_{Test} \times \hat{p}_{Test}) + \left(\hat{\varepsilon}'_{Customer} \times (1 - \hat{p}'_{Test})\right)}\right) \times \frac{C_3 + C_4 + C_5}{C}$$

If we wish to instantiate this model with default values, we get:

$$ROI_2 = \left(1 - \frac{(6 \times \hat{p}_{Test}) + \left(\hat{\varepsilon}'_{Customer} \times \left(1 - ((\hat{p}_{Test} \times 0.9) \times g)\right)\right)}{(6 \times \hat{p}_{Test}) + \left(\hat{\varepsilon}'_{Customer} \times \left(1 - (\hat{p}_{Test} \times 0.9)\right)\right)}\right) \times 0.5$$

6.3.3 Risk Assessment Model

This model assumes that the organization will only inspect the highest risk modules. We assume that the initial process consists of a testing phase and post-release. Code inspections will be added for the highest risk modules. The ROI model is then:

$$ROI_2 = \frac{\left|\lambda_{Code\ Inspection}\right|}{\left|\alpha_{Test}\right|} \times \left(\frac{\hat{\varepsilon}_{Test}\hat{p}'_{Test}\left(1-\beta\right) + \hat{\varepsilon}'_{Customer}\left(1-\hat{p}'_{Test}\right)\left(1-\beta\right) - \hat{\varepsilon}_{Code\ Inspection}}{\hat{\varepsilon}_{Test}\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(1-\hat{p}'_{Test}\right)} \right) \times \frac{C_3 + C_4 + C_5}{C}$$

The first term is essentially the proportion of defects that are in modules that are inspected. These would be the high-risk modules. Determining the can be based on a logistic function, such as that shown in Figure 23.

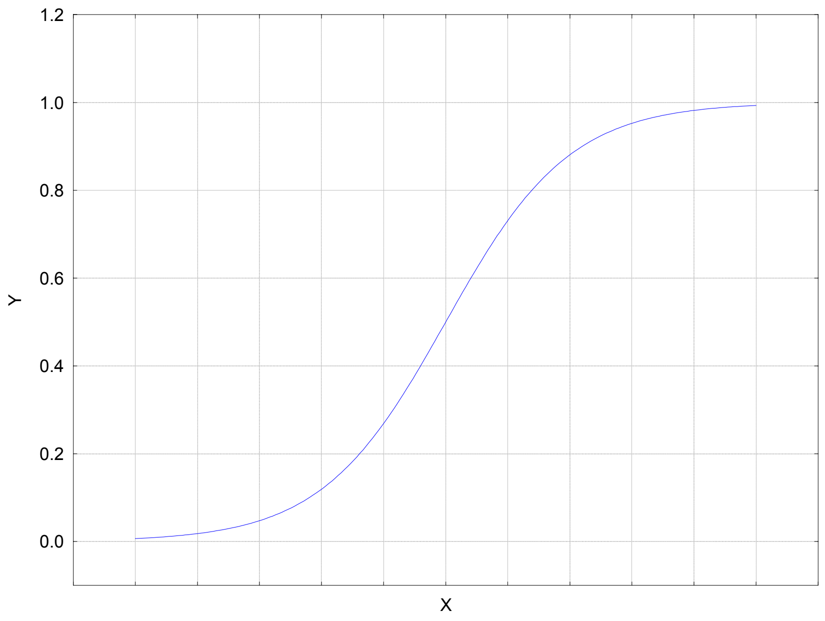


Figure 23: An example of a logistic function. The x-axis would be in standard deviation units in terms of some metrics, and the y-axis is the proportion of defects found. This s-shaped curve models the relationship between metrics and the occurrence of defects.

If we wish to instantiate this model with default values, we get:

$$ROI_2 = \frac{\left|\lambda_{Code\ Inspection}\right|}{\left|\alpha_{Test}\right|} \times \left(\frac{4.86\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(0.9 - 0.81\hat{p}_{Test}\right) - 1.5}{2 \times \left(6\hat{p}_{Test} + \hat{\varepsilon}'_{Customer}\left(1 - 0.9\hat{p}_{Test}\right)\right)} \right)$$

6.3.4 Higher Reuse Model

The higher reuse model is based directly on the ROI model as follows:

$$ROI_2 = \left(\left(\frac{RLOC}{TLOC} \times \left(1 - RCR\right) \right) - \frac{\kappa}{C} \right)$$

If we wish to instantiate this model with default values, we get:

$$ROI_2 = \left(\left(\frac{RLOC}{TLOC} \times 0.8 \right) - \frac{\kappa}{C} \right)$$

6.3.5 Multiple Models

It is possible to account for multiple types of static analyses. For the types of analyses that reduce rework effort, in principle the savings can just be added because all model instantiations that we derived have the same denominator. The savings from reuse can also be added. For example, if the savings from automated detection of defects and risk assessment were 5% and 10%, then the total cost savings to the project from combining these two would be 15%.

However, there is likely to be an overlap in savings between the automatic defect detection techniques and the risk assessment techniques. If both of these are applied on the same project, then they have to applied sequentially. For example, an automated defect detection tool is applied followed by focused inspections. Some of the defects that would have been discovered by inspections would have been discovered by the tool. Therefore, inspections would not necessarily have the same ROI as when they are used without automated detection. The simple sum of the ROI values from both will therefore be exaggerated. To overcome this deficiency one would have to know how many defects that are found through automated detection would have been found through an inspection. If the assumption is made that automated detection is just a more efficient way of finding the same defects as inspections, then a simple modification of the ROI model to put these two techniques in sequence would give the correct ROI.

6.4 Assumptions and Limitations

The above ROI models make a number of assumptions. While we endeavoured to ensure that the models are as accurate as possible, some assumptions were inevitable. These may be relaxed in future revisions of the model.

It should be noted that all of the assumptions that we have made are **conservative**. This means that they would result in a smaller estimate of ROI rather than inflating it. Therefore, if a business case is attractive under the current models, they would certainly be attractive when the assumptions are relaxed.

The first assumption concerns the rework impacts of reuse. There is evidence that reused code tends to be of higher quality than new code. This is due to the code being used in multiple projects, and hence receives more scrutiny (e.g., during inspections) and testing than new code. The impact on ROI of that is a reduction in rework costs from reuse. We did not account for a reduction in rework costs from reuse. The assumption we make is that reuse would only increase the productivity of the *construction* part of project costs. Should rework costs be accounted for then the benefits of technologies that improve reuse would be higher.

Poulin [32] also considers another benefit of reuse: Service Cost Avoidance. This is the potential saving from not having to maintain reused code. It is common that there is a different organization, group, or project that is maintaining a reuse library. To the extent that a project that reuses some of this code does not get involved in maintenance, then these maintenance costs are saved. In our context we are looking at reuse during a re-architecting phase. Therefore, the project that reuses parts of the system would still have to maintain the reused code. Consequently, we cannot really account for service cost reductions in our context.

6.5 Interpreting the ROI Values

In this section we will explain how to interpret and use the ROI values that are calculated.

First, it must be recognized that the ROI calculations, cost savings, and project costs as presented in these spreadsheets are estimates. Inevitably, there is some uncertainty in these estimates. The uncertainty stems from the variables that are not accounted for in the models (there are many other factors that influence project costs, but it is not possible to account for all of these since the model would then be unusable). Another source of uncertainty are the input values themselves. These values are typically averages calculated from historical data; to the extent that the future differs from the past these values will have some error.

Another important point to note is that the calculated ROI values are for a single project. A software organization will have multiple on-going and new projects. The total benefit of implementing static techniques to the organization can be calculated by generalizing the results to the organization. For example, if the ROI for a single project is say a 15% saving. Assuming that the input values are the same for other projects in the organization, then we can generalize to the whole organization and estimate that if static analysis is implemented on all projects in the organization, the overall savings would be 15%. If the software budget for all the projects is say $20 million, then that would translate into an estimated saving of $3 million. Note that this is **not an annual saving**, but a saving in total project budgets that may span multiple years (i.e., for the duration of the projects).

This saving does not account for the cost of acquiring any tools and the cost of training nor consulting. These costs have to be deducted from the savings. If you are implementing static analysis on a single project, then these costs would have to be deducted from the single project savings. If you are implementing static analysis in the whole organization, then these costs will be spread across multiple projects. In such a case, these costs would be deducted from the organizational savings (the calculation of which is described above).

For example, let’s say that acquiring an enterprise license for a tool costs $10,000 and that consulting fees are $10,000 for implementing some static analysis technology across the whole organization. Taking our organizational example from above, the net savings would then be $2,980,000 rather than $3 million.

7 Appendix D: Literature Review

The following review of the literature is used to determine the default values for our various ROI models. The default values are the average from published articles.

The following criteria were used to deem an article appropriate for inclusion in this average:

- There should be a clear indication that the published data are based on actual projects rather than being mere opinions or a rehashing of someone else's numbers. There are many opinion articles that were excluded.
- It should be clear what is being measured. For instance, if we are looking at inspections, then it should be clear that inspections actually took place and the data pertain to the inspections.
- The unit of observation should be clear. For example, for inspection data it should be stated whether the data pertain to all inspections performed on a project or to a single inspection.

The exceptions to the above criteria were review articles that already summarized evidence. In such a case quality judgement on the review methodology was performed to decide whether the data from the review should be included.

7.1 Effectiveness of Defect Detection Activities

In the following, the articles that give data on effectiveness are discussed.

Fagan [10] presents data from a development project at Aetna Life and Casualty, Hartford, Connecticut, USA. An application program of eight modules (4439 noncommentary source statements) was written in Cobol by two programmers. Design and code inspections were introduced into the development process, the number of inspection participants ranged between three and five. After 6 months of actual usage, 46 defects had been detected during development and usage of the program. Fagan reports that 38 defects had been detected by design and code inspections together, yielding a defect detection effectiveness for inspections of 82%. The remaining 8 defects had been found during unit test and preparation for acceptance test.

In another article, Fagan [11] publishes data from a project at IBM Respond, United Kingdom. A program of 6271 LOC in PL/1 was developed by 7 programmers. Over the life cycle of the product, 93% of all defects were detected by inspections. He also mentions two projects of the Standard Bank of South Africa (143 KLOC) and American Express (13 KLOC of system code), each with a defect detection effectiveness for inspections of over 50% without using trained inspection moderators.

Weller [39] presents data from a project at Bull HN Information Systems which replaced inefficient C code for a control microprocessor with Forth. After system test had been completed, code inspection effectiveness was around 70%.

Grady and van Slack [16] report on experiences from achieving widespread inspection use at HP. In one of the company's divisions inspections (focusing on code) typically found 60 to 70% of the defects.

Shirey [34] states that defect detection effectiveness of inspections is typically reported to range from 60 to 70%.

Barnard and Price [2] cite several references and report a defect detection effectiveness for code inspections varying from 30 to 75%. In their environment at AT&T Bell Laboratories, the authors achieved a defect detection effectiveness for code inspections of more than 70%.

McGibbon [27] presents data from Cardiac Pacemakers Inc. where inspections are used to improve the quality of life critical software. They observed that inspections removed 70 to 90% of all faults detected during development.

Collofello and Woodfield [8] evaluated reliability-assurance techniques in a case study - a large real-time software project that consisted of about 700,000 lines of code developed by over 400 developers. The project was performed over several years recording quality-assurance data for design, coding, and testing. The respective defect detection effectiveness are reported to be 54% for design inspections, 64% for code inspections, and 38% for testing. Although the authors state that testing efforts are normally identifiable as unit testing, integration testing, and acceptance testing, they do not provide more detail on the testing procedures in the project under examination.

Kitchenham et al. [25] report on experience at ICL, where 57.7% of defects were found by software inspections. The total proportion of development effort devoted to inspections was only 6%.

Gilb and Graham [14] include experience data from various sources in their discussion of the benefits and costs of inspections. IBM Rochester Labs publish values of 60% for source code inspections, 80% for inspections of pseudocode, and 88% for inspections of module and interface specifications.

Grady [17] performs a cost/benefit analysis for different techniques, among them design and code inspections. He states that the average percentage of defects found for design inspections is 55%, and 60% for code inspections.

Jones [20] discusses defect-removal effectiveness in the context of evaluating current practices in US industry. He gives approximate ranges and averages of defect detection effectiveness for various activities.

Franz and Shih [13] present data from code inspection of a sales and inventory tracking systems project at HP. This was a batch system written in COBOL. Their data indicate that inspections had 19% effectiveness for defects that could also be found during testing.

Meyer [28] performed an experiment to compare program testing to code walkthroughs and inspections. The subjects were 59 highly experienced data processing professionals testing and inspecting a PL/I program. Myers reports an average effectiveness value of 0.38 for inspections.

To summarize this data, we assume that effectiveness follows a triangular distribution. A triangular distribution has a maximum value, a minimum value, and a most likely value [9]. When little is known about the actual distribution of a variable, it is common to adopt a triangular distribution [36].

The maximum value for the effectiveness of design inspections is reported in [14] from IBM Rochester Labs. This was 0.84 (average for two types of design document). The minimal value was reported by Jones as 0.25 for informal design reviews [20]. The most likely value is the mid-point of the data from [8] and the industry mean reported in [20], which is 0.57.

For the minimum value of code inspection effectiveness, only the data from [13] is used (0.19). The maximum value of 0.7 was obtained in [39]. For the most likely value, the data from [8, 14, 16, 20, 28] was used to produce an average, which was 0.57.

7.2 Average Effort per Defect

In this section, the data on the average effort per defect for various defect detection techniques (design inspections, code inspections, testing) is summarized.

Ackerman et al. [1] present data on different projects as a sample of values from the literature and from private reports. As the inspection process is described in the article as a six-step process including rework and follow-up, the data should mirror the cost of finding and fixing defects.

The development group for a small warehouse-inventory system used inspections on detailed design and code. For detailed design, they reported 3.6 hours of individual preparation per thousand lines, 3.6 hours of meeting time per thousand lines, 1.0 hours per defect found, and 4.8 hours per major defect found (major defects are those that will affect execution). For code, the results were 7.9 hours of preparation per thousand lines, 4.4 hours of meetings per thousand lines, and 1.2 hours per defect found.

A major government-systems developer reported the following results from inspection of more than 562,000 lines of detailed design and 249,000 lines of code: For detailed design, 5.76 hours of individual preparation per thousand lines, 4.54 hours of meetings per thousand lines, and 0.58 hours per defect found. For code, 4.91 hours of individual preparation per thousand lines, 3.32 hours of meetings per thousand lines, and 0.67 hours per defect found.

Two quality engineers from a major government-systems contractor reported 3 to 5 staff-hours per major defect detected by inspections showing a surprising consistency over different applications and programming languages.

A banking computer-services firm found that it took 4.5 hours to eliminate a defect by unit testing compared to 2.2 hours by inspection (probably code inspections).

An operating-system development organization for a large mainframe manufacturer reported that the average effort involved in finding a design defect by inspections is 1.4 staff-hours compared to 8.5 staff-hours of effort to find a defect by testing.

Weller [39] reports data from a project that performed a conversion of 1200 lines of C code to Fortran for several timing-critical routines. While testing the rewritten code, it took 6 hours per failure. It was known from a pilot project in the organization that they had been finding defects in inspections at a cost of 1.43 hours per defect. Thus, the team stopped testing and inspected the rewritten code detecting defects at a cost of less than 1 hour per defect.

McGibbon [27] discussed software inspections and their return on investment as one of four categories of software process improvements. For modeling the effects of inspections, he uses a sample project of an estimated size of 39.967 LOC. It is assumed that if the cost to fix a defect during design is 1X, then fixing design defects during test is 10X and in post-release is 100X. Thus, the rework effort per defect for different phases is assumed to be 2.5 staff hours per defect for design inspections, 2.5 staff hours for code inspections, 25 staff hours for testing, and 250 staff hours for maintenance (customer-detected defects).

Collofello and Woodfield [8] discuss a model for evaluating the efficiency of defect detection. In order to conduct a quantitative analysis, they needed to estimate some factors for which they had not enough data. They performed a survey among many of the 400 members of a large real-time software project who were asked to estimate the effort needed to detect and correct a defect for different techniques. The results were 7.5 hours for a design error, 6.3 hours for a code error, both detected by inspections, 11.6 hours for an error found during testing, and 13.5 hours for an error discovered in the field.

Kitchenham et al. [25] report on experience at ICL where the cost of finding a defect in design inspections was 1.58 hours.

Gilb and Graham [14] include experience data from various sources in their discussion of the benefits and costs of inspections. A senior software engineer describes how software inspections started at Applicon. In the first year, 9 code inspections and 39 document inspections (other documents than code) were conducted and an average effort of 0.8 hours was spent to find and fix a major problem. After the second year, a total of 63 code inspections and 100 document inspections had been conducted and the average effort to find and fix a major problem was 0.9 hours.

Bourgeois [5] reports experience from a large maintenance program within Lockheed Martin Western Development Labs (LMWDL) where software inspections replaced structured walk-throughs in a number of projects. The analyzed program is staffed by more than 75 engineers who maintain and enhance over 2 million lines of code. The average effort for 23 conducted software inspections (6 participants) was 1.3 staff-hours per defect found and 2.7 staff-hours per defect found and fixed. Bourgeois also presents data from Jet Propulsion Laboratory which is used as an industry standard. There, the average effort for 171 software inspections (5 participants) was 1.1 staff-hours per defect found and 1.4 to 1.8 staff-hours per defect found and fixed.

Franz and Shih's data [13] indicate that the average effort per defect for code inspections was 1 hour and for testing was 6 hours.

In presenting the results of analyzing inspections data at JPL, Kelly et al. [23] report that it takes up to 17 hours to fix defects during formal testing, based on a project at JPL. They also report approximately 1.75 hours to find and fix defects during design inspections, and approximately 1.46 hours during code inspections.

Following the same logic as for effectiveness, we compute the average values of effort per defect for the various defect detection activities.

Considering the average effort per defect for design inspections, Ackerman et al. [1] provide the maximum value of 2.9 hours on average per defect for different design documents on a project. The same article also provides the minimum value obtained from another project. The most likely value was the average of another project in [1], and [23, 25]. This was 1.58 hours.

Considering the average effort per defect for code inspections, the maximum value for code inspections of 2.7 hours per defect was reported in [5]. The minimal value was reported in []. The most likely value was the mean of values reported in [1, 13, 23, 39], which was 1.46 hours.

Finally, for the average effort per defect for testing, the maximum value of 17 was obtained from Kelly et al. based on a project at JPL [23]. The minimum of 4.5 was obtained from Ackerman et al. [1]. The most likely value was the mean for projects reported in [13, 39], and was computed as 6 hours.

8 Author Biography

Khaled El Emam holds a number of positions in research and industry. He is VP of Technology for TrialStat Corporation, which develops IT solutions for clinical trials, and a Senior Research Officer at the National Research Council of Canada where he is the technical lead of the Software Quality Laboratory. In another capacity, Khaled supports software organizations with their quality improvement initiatives through training and consulting services. He is also a senior consultant with the Cutter Consortium.

Khaled is co-editor of ISO's project to develop an international standard defining the software measurement process (ISO/IEC 15939), and is leading the software engineering process area in the IEEE's project to define the Software Engineering Body of Knowledge. He has also co-edited two books on the software process, both published by the IEEE CS Press; and he is an adjunct professor at both the School of Computer Science at McGill University and the Department of Computer Science at the University of Quebec at Montreal. Currently, Khaled is a resident affiliate at the Software Engineering Institute in Pittsburgh. In addition, he is the current vice-president of the Ottawa Software Quality Association, a professional NFP organization that promotes quality practices in software organizations through seminars and certification. He is on the editorial boards of IEEE Transactions on Software Engineering and the Empirical Software Engineering Journal.

One of his recent assignments was with the project developing the on-board flight software for the Space Shuttle, where he is working with the QA team to develop new techniques to optimize the outcomes of software inspections. This work has results in a substantial improvement in the project's ability to target their defect detection efforts on the modules most likely to contain faults. He has also developed and applied a risk management methodology that helps managers focus on the high risk areas of their systems and allocate resources efficiently. The ROI from the application of the methodology varies from a 20% to over 40% saving in warranty costs for delivered applications. Another aspect of the methodology allows cost estimation and budgeting very early in a project when information is minimal. This has been demonstrated to have an accuracy as high as +/- 9% deviation from actual costs. It also allows project managers to identify the elements in their projects that have the biggest influence on costs, and estimate the probability of meeting fixed budgets.

Previously, Khaled was the International Trials Coordinator for the SPICE Trials, where he was leading the empirical evaluation of the emerging process assessment International Standard, ISO/IEC 15504, world-wide. This was a large effort to collect data and evaluate process assessments. He was the head of the Quantitative Methods Group at the Fraunhofer Institute for Experimental Software Engineering in Germany; a research scientist at the Centre de recherche informatique de Montreal (CRIM) in Canada; a researcher in the software engineering laboratory at McGill University; and worked in a number of research and development projects for organizations such as Toshiba International Company and Honeywell Control Systems in the UK, and Yokogawa Electric in Japan. Over the last ten years he has managed over half a dozen software development projects.

He has received a number of awards from ISO, IEEE, the DoD, and in 2002 has been ranked second world-wide in a evaluation of Systems and Software Scholars performed annually by the Journal of Systems and Software (the evaluationm identifies the top 15 scholars in the world).

Khaled El Emam obtained his Ph.D. from the Department of Electrical and Electronics Engineering, King's College, the University of London (UK) in 1994.

9 References

[1] A. F. Ackerman, L. S. Buchwald, and F. H. Lewski, "Software Inspections: An Effective Verification Process". In *IEEE Software, vol. 6, No. 3*, pp. 31-36, 1989.

[2] J. Barnard and A. Price, "Managing Code Inspection Information". In *IEEE Software*, vol. 11, pp. 59-69, 1994.

[3] W. Baziuk, "BNR/Nortel Path to Improve Product Quality, Reliability, and Customer Satisfaction". In *Sixth International Symposium on Software Reliability Engineering (ISSRE95)*, 1995.

[4] B. Beizer, *Software Testing Techniques*: International Thomson Computer Press, 1990.

[5] K. Bourgeois, "Process Insights from a Large-Scale Software Inspection Data Analysis". In *Crosstalk: The Journal of Defense Software Engineering*, vol. 9, no. 10, pp. 17-23, 1996.

[6] L. Briand, K. El-Emam, O. Laitenberger, and T. Fussbroich, "Using Simulation to Build Inspection Efficiency Benchmarks for Development Projects". In *Proceedings of the 20th International Conference on Software Engineering*, pp. 340-349, 1998.

[7] L. C. Briand, B. Freimut, and F. Vollei, "Assessing the Cost-Effectiveness of Inspections by Combining Project Data and Expert Opinion". In *Technical Report-International Software Engineering Research Network ISERN-99-14*, 1999.

[8] J. Collofello and S. Woodfield, "Evaluating the Effectiveness of Reliability-Assurance Techniques". In *Journal of Systems and Software*, pp. 191-195, 1989.

[9] M. Evans, N. hastings, and B. Peacock, *Statistical Distributions*: John Wiley & Sons, 1993.

[10] M. Fagan, "Design and Code Inspections to Reduce Errors in Program Development". In *IBM Systems*, vol. 15, no. 3, pp. 182-211, 1976.

[11] M. Fagan, "Advances in Software Inspections". In *IEEE Transactions on Software Engineering*, vol. 12, no. 7, pp. 744-751, 1986.

[12] N. Fenton and N. Ohlsson, "Quantitative Analysis of Faults and Failures in a Complex Software System". In *IEEE Transactions on Software Engineering*, vol. 26, no. 8, pp. 797 -814, 2000.

[13] L. Franz and J. Shih, "Estimating the Value of Inspections and Early Testing for Software Projects". In *Hewlett-Packard Journal*, pp. 60-67, December 1994.

[14] T. Gilb and D. Graham, *Software Inspection*: Addison-Wesley Publishing Company, 1993.

[15] D. Glasberg, K. El Emam, W. Melo, and N. Madhavji, "Validating Object-oriented Design Metrics on a Commercial Java Application," Technical Report, National research Council of Canada, NRC/ERB-1080 2000.

[16] R. Grady and T. V. Slack, "Key Lessons in Achieveing Widespread Inspection Use". In *IEEE Software*, vol. 11, no. 4, pp. 46-57, 1994.

[17] R. B. Grady, *Practical Software Metrics for Project Management and Process Improvement*: Englewood Cliffs, NJ, Prentice-Hall, 1992.

[18] B. Hailpern and P. Santhanam, "Software Debugiing, Testing, and Verification". In *IBM Systems Journal*, vol. 41, no. 1, 2002.

[19] W. Humphrey, *Managing the Software Process*: Addison Wesley, 1989.

[20] C. Jones, "Software Defect Removal Efficiency". In *IEEE Computer*, vol. 29, no. 4, pp. 94-95, 1991.

[21] C. Jones, *Software Assessments, Benchmarks, and Best Practices*: Addison-Wesley, 2000.

[22] M. Kaaniche and K. Kanoun, "Reliability of a Commercial Telecommunications System". In *Proceedings of the International Symposium on Software Reliability Engineering*, pp. 207-212, 1996.

[23] J. C. Kelly, , J. S. Sheriff, and J. Hops, "An Analysis of Defect Densities Found During Software Inspections". In *Journal of Systems Software*, vol. 17, pp. 111-117, 1992.

[24] T. Khoshgoftaar, E. Allen, W. Jones, and J. Hudepohl, "Return on Investment of Software Quality Predictions". In *Proceedings of the IEEE Workshop on Application-Specific Software Engineering Technology*, pp. 145 -150, 1998.

[25] B. Kitchenham, A. Kitchenham, and J. Fellows, "The Effects of Inspections on Software Quality and Productivity". In *ICL Technical Journal*, vol. 5, no. 1, pp. 112-122, 1986.

[26] S. Kusumoto, "Quantitative Evaluation of Software Reviews and Testing Processes," PhD Thesis Thesis, Osaka University 1993.

[27] T. McGibbon, "A Business Case for Software Process Improvement," Technical Report, A DACS State-of-the-Art Report (www.dacs.com/techs/roi.soar/soar.html) 1996.

[28] G. Meyer, "A Controlled Experiment in Program Testing and Code Walkthroughs/Inspections". In *Communications of the ACM*, vol. 21, no. 9, pp. 760-768, 1978.

[29] K.-H. Moller and D. Paulish, "An Empirical Investigation of Software Fault Distribution". In *Proceedings of the First International Software Metrics Symposium*, pp. 82-90, 1993.

[30] G. Myers, *The Art of Software Testing*: John Wiley & Sons, 1979.

[31] N. Ohlsson and H. Alberg, "Predicting Fault-Prone Software Modules in Telephone Switches". In *IEEE Transactions on Software Engineering*, vol. 22, no. 12, pp. 886-894, 1996.

[32] J. Poulin, *Measuring Software Reuse: Principles, Practices, and Economic Models*: Addison-Wesley, 1997.

[33] RTI, "The Economic Impacts of Inadequate Infrastructure for Software Testing," Technical Report, National Institute of Standards and Technology 2002.

[34] G. Shirey, "How Inspections Fail". In *Proceedings of the Ninth International Conference on Testing Computer Software*, pp. 151-159, 1992.

[35] F. Shull, V. Basili, B. Boehm, A. W. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, and M. Zelkowitz, "What We Have Learned About Fighting Defects". In *Proceedings of the Eighth IEEE Symposium on SOftware Metrics*, 2002.

[36] D. Vose, *Quantitative Risk Analysis: A Guide to Monte Carlo Simulation Modeling*: John Wiley & Sons, 1996.

[37] L. Votta, "Does Every Inspection Need a Meeting ?". In *ACM Software Engineering Notes*, vol. 18, no. 5, pp. 107-114, 1993.

[38] J. Walsh, "Preliminary Defect Data from the Iterative Development of a Large C++ Program". In *Proceedings of OOPSLA*, pp. 178-183, 1992.

[39] E. F. Weller, "Lessons from Three Years of Inspection Data". In *IEEE Software*, vol. 10, no. 5, pp. 38-45, 1993.

[40] D. Wheeler, B. Brykczynski, and R. Meeson, "An Introduction to Software Inspections," in *Software Inspection: An Industry Best Practice*, D. Wheeler, B. Brykczynski, and R. Meeson (eds.): IEEE Computer Society Press, 1996.