SOFTWARE RELEASE METHODOLOGY: A CASE STUDY

Annie Ibrahim Rana, Muhammad Waseem Arfi
National University of Computer and Emerging Sciences, Lahore
mscs089@nu.edu.pk, mspm0334@nu.edu.pk

Abstract

Software Release Management is an important key technology for distributing the project/product to the customer. The key success factor of any Software Product lies in how delicately the product is released to the customer. The traditional SCM system does not guarantee to handle Release Management issues of a complex system. Complex systems involve complex database, N-tiers just to name a few. Each kind of application involves special technical consideration from a release perspective. In this paper, we analyze different quality parameters related to the release of a product. These parameters should be handled through Software Release Model. The chosen model that supports those parameters is discussed. A controlled environment is tested for those parameters through the use of this model.

Keywords: *software release management, software configuration management, version control, stability, consistency, maintenance time, bugs, delta version model*

1 INTRODUCTION

"Release management is the process of planning, building, testing and deploying hardware and software, the version control and storage of software."[1] Release Management process becomes vital in Product Development & Deployment, it basically manages the frequency of Product releases and their levels, i.e. full release or patches. Release management is not just what goes into the product development environment but also how something goes into the product development environment.

Implementing a true release model results in two businesses benefits, reduction in overall cost and improved customer satisfaction. Intech Process Automation (www.intechww.com) is in the process of Product Development. It is required to manage multiple releases of a product. At the moment, internal releases are continually produced. Software Configuration Management practices are formally followed for Development and producing internal releases. No special consideration is being given to handle multiple releases. The following are the issues encountered while producing multiple releases and need to be taken care of:

- There are multiple clients using multiple releases of the product. Each client may encounter issues. ***Those issues may also exist in other releases, so issues should be resolved in all victim releases*** and the client should be served with the appropriate patch. Organization cannot pressurize client to go for the latest release. It involves legal as well as moral issues. Moreover, Maintenance should not have to fix the same issue in all of the releases again and again. The release model should be able to handle issues fixed in all of the releases of the Product where that issue exists. For example our product has two releases versioned 1.1.0 and 1.2.0 running on client sides C1 and C2 respectively. Client "C1" has reported an issue in his release i.e., 1.1.0, this issue should not only be resolved in release 1.1.0 but also in release 1.2.0 if the issue exists in that release too.
- This mechanism will greatly reduce the **maintenance time** in the long run. Moreover client C2 who has not yet reported the bug will be more **satisfied and feel secured** by having a more **stabilized** release.
- The solution model should be flexible to enable us to mark the release as ready-to-market. After marking any release as ready-to-market, the particular release will not be introduced to any new functionality enhancement.

Rather only bug fixes will be introduced to ready-to-market release. Such a process will always be followed before shipping the release to the customer. For example a release R1 is going to be shipped to customer C1 on March, 2005; we will mark the release as ready-to-market in Dec, 2004. After that no more functionality will

be added to release R1 but just the bugs in release R1 will be fixed. Such a mechanism will ensure that release R1 is made stabilized before shipping to the customer C1. This will also make sure that our release is performing consistently over a period of time.

A model needs to be adopted that fulfills certain parameters identified in a release. The parameters are described above. One of the solution models is Delta Versioning Model that guarantees fulfilling the criteria mentioned above. A brief introduction of the model is given below.

2 THE DELTA VERSIONING MODEL

According to this model, a delta is generated for each version. It means that each change in this model is treated as a delta. It can be seen in the following figure 2.1.

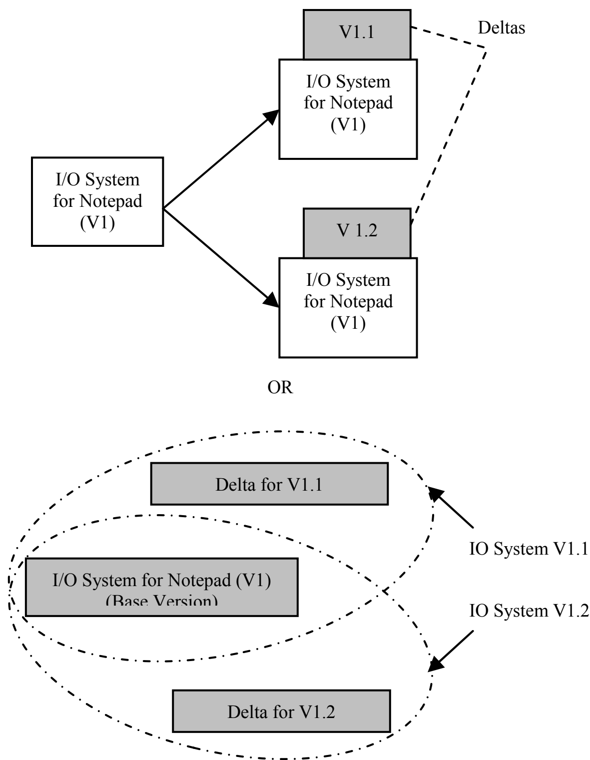


Figure No. 2.1: Delta Versioning Mechanism

Let's suppose there is an IO system for notepad, and due to memory size variation, we need to have two different versions for each memory size. Now the common module will be kept separately as single copy for both versions and we will have two variant modules as deltas. Now if there is a common change required in all versions then we will make changes only in the single copy of common modules and if change is related to a specific version then changes will be made separately in corresponding delta. This model is supported by the ClearCase configuration management tool.

2.1 Description of Delta Version Model

"A *delta* is the difference between two versions and serves to identify the changes and to save space in the repository [2]."

Versions differ with respect to specific properties (e.g., represented by versioned attributes).

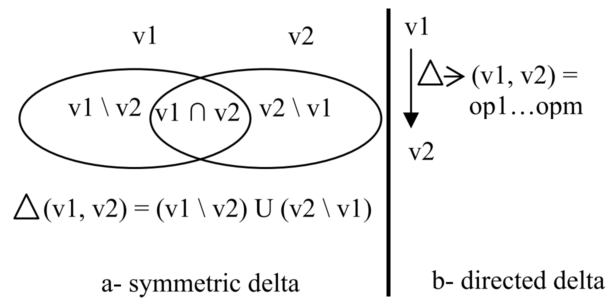


Figure No. 2.2: Symmetric & Directed Delta

The difference between two versions is called a *delta.* This term suggests that differences should be small compared to invariants. Delta can be defined in two ways (figure 2.2): a symmetric delta between two versions v1 and v2 consists of properties specific to both v1 and v2 (v1 \ v2 and*v*2 \ *v*1, respectively, where \ denotes set difference); or a *directed delta,* also called a *change,* is a sequence of (elementary) change operations *op*1...*opm* which, when applied to one version *v*1, yields another version *v*2 (note the correspondence to transaction logs in databases). In practice, deltas are not necessarily small. In the worst case, the common part of *v*1 and *v*2 may even be empty.

In fact, items may undergo major changes, and the common properties may become smaller and smaller the more versions are created. For example, it is usually unrealistic to assume that all versions of module bodies realize the same interface. On the other hand, common properties do have to be asserted because otherwise it does not make sense to group versions at all.

A way out of this dilemma is *multilevel versioning;* that is, a version may have versions themselves [3].

Symmetric delta shown in figure 2.2 is well suited for systems which have common modules.

2.2 Delta Model for Research Problem

There are three issues concerning a release. First of all those issues are defined in the following:

- **Stability** is defined as “The condition of being stable or resistant to change” and “the quality of being free from change or variation”. Customer needs such a release that is stable enough so that it does not affect the processes running on client-side. Our product is meant for Oil & Gas industry and will be installed on plants. The prime concern of the customer is that the release should not disrupt his/her plant operations in any way. It should be made sure through the stability of the product. Stability will be calculated through the number of bugs reported and bugs fixed in a release. This data will show that either the release is going towards stability or not. A trend of bugs reported in the release vs. bugs removed from the release will show the stability curve of any release. If the bug reporting trend is declining, it will show that the release is going towards stability. This should be made sure before shipping the release to the customer. And moreover, severity of bugs is also counted, severe and crucial bugs fixation will be at high priority, so one parameter to check the stability of a release can be measured in terms of number of all reported severe bugs (reported bugs relate to perceived quality).
- **Consistency** is defined as “Agreement or logical coherence among releases about common bug or bugs” and “uniformity of successive results or events”. A common bug can be defined as a bug shared by different releases. For any release to become consistent with other releases, the common bugs should be fixed in all victim releases, the data could be taken from Bug tracking system. Some of the releases happen to produce the same bug as produced by previous releases and it may be already fixed in any other release. This tendency shows the inconsistency among releases regarding that common bug. This inconsistency needs to be measured and minimized.
- **Maintenance** is defined as “The work of keeping something in proper condition”. We need to maintain all of our releases in proper condition. Maintenance time can be measured by looking at how many times, a bug need to be fixed by the maintenance team. In fact if a common bug already has been fixed in a release during maintenance then same maintenance time should be utilized to fix that common bug in other releases too. Again this data will be made available through bug tracking system. In bug fixing not only the Maintenance team is involved but testing team and Issue Management Committee is also involved for this process. This time should be lowered down through use of appropriate model.

It is required to have consistency in all related version (horizontal (revisions), vertical (variants)), stability in each version and reduced maintenance time. How this under discussion model will achieve these all concerns or case study objectives, let’s have a light discussion on the delta versioning model.

At Intech Process Automation, functional modules are used to be made for a desired system. Each module is given a module id and all modules baselines are kept in repository. Different versions of releases are tracked through the traceability matrix. As we have gotten the data related to releases but common bug data is found only for two versions, for example if we have releases’ versions as v1.1, v1.2, v1.3, v1.4…and so on but the common bug data is found for version v1.1 and v1.2, similarly for v1.3 and v1.4 and so on, so the common modules for two related versions (previous and superseding versions) will be placed under the base version lets call it vb1.0, and differing modules will be placed under the deltas for each respective version, lets call them Vd1, Vd2. At the moment, only 10 releases are studied for the case study (this is limitation of this case study) and common bugs are tracked for only five sets of releases and each set has two releases as mentioned earlier. Now let’s view the case problem at root level. When ever a bug is reported in a specific version, the bug is checked and reported. Then bug is used to be tracked in all related versions. If a bug traces are found in other versions the bug status is changed to common bug (in our case there will be only two

releases common bugs), else it will remain specific bug for that claiming or reporting version. When a common bug is traced in other versions, then impacted modules are also traced and tracked. If the target modules (most likely to be changed to fix that change) are in base version set of modules then bug fixation will be done at single place through the patch and implicitly fixing the bug for all those versions that are sharing the base version set of modules. But if the impacted modules exist in any of the version delta then as modules are not common or not shared in any other version, then patch will be made separately for each impacted module.

This does not implies that bug does not exist in any other version, it may exist, but here we are only concerned with the common modules shared by related versions, which has a common bug. So in later case of bug fixation, patch will be made for that respective version module. And if more than two versions have reported the same kind of bug, and the impacted modules are in respective deltas then individual patches will be made for each version, as impacted modules are in deltas rather than in base version, so this model will only do its best when a bug fixation is need to be made for the modules in the base version.

Now coming back to the case study concerns, first we will talk about the consistency among different versions. We have defined the consistency for the versions as, "The bug reported in a version should be fixed in that particular version as well as all other impacted versions which have the same bug in them", and now this is the time to rephrase the definition in more technical way: "The bug reported in a version should be fixed in that particular version as well as in all those *versions which share the same base version, and bug exists in base version***".** And the delta model will assure this thing that a bug will be fixed in all versions sharing the same base version modules or bug will not be fixed in any version sharing the same base version, so in a way making the versions consistent for the common bug among them.

Our second concern of the case study is stability of the versions, and now we will discuss that how this model stabilize versions. We have defined the stability as, "A version will be considered stable if there is not a single bug exist in that version", but this is idealistic situation and realistically impossible, but if it is possible even then it is very hard to attain such situation. Stability is a subjective measure but our aim is to define the stability of a version objectively. We consider a version stable when there is no bug reported (***perceived quality***), and stability will be minimized as reported bugs increase. Now suppose there is a version v.1.1 and version v1.2, version 1.2 supersedes the version v1.1. Now if reported bug is in version 1.1 and that bug also exists in version 1.2 then it may possible that common bug is in a common module. If we use delta versioning model then it means the common modules will be in the base version. Now fixing the problem for version 1.1 will also fix the problem for version 1.2 implicitly, so even the bug has not reported yet from version 1.2 but this model helps in reducing the risk of bug (improves the ***actual quality***) so stabilizing the versions.

Our next concern of case study is reduced maintenance time for versions, and now we will discuss that how delta versioning model achieves above stated goal (reduced maintenance time). We have defined maintenance time as, "The time taken for a bug or bugs fixation separately in all impacted versions that have the bug or bugs". And with going again into the architecture of the delta version model, the bug fixation time will be reduced through this model for all impacted versions which have bug or bugs in the base version. Because early we were fixing a common bug in common module of all impacted versions as each version has its own redundant copy of the module so it was multiplying the time for bug fixation in each version. But now this will be done in lesser time with help of the delta versioning model.

And one thing must be clear about time, that we are measuring time in number of transitions (a transition is physical bug fixation attempt before testing (unit, functional etc.).

The data has been made available by Intech Process Automation as described in Scope section. Moreover, we have implemented the delta model in the Test Environment and investigated the performance of the model against the parameters defined earlier. The research methodology used for this case study has been described as follows:

3 THE TEST ENVIRONMENT

The model cannot be implemented directly to the existing system. A test environment needs to be created to test the model that how the system will perform using Delta model that is selected for our research. In order to analyze the model effectiveness, older release(s) are selected to pass through this model and examine the effect

against the parameters identified for improvement in the existing system. Certain steps have been taken in order to implement the test environment according to the model.

3.1 Identifying relationship among Modules

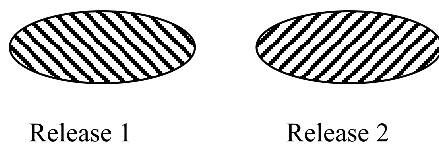
In the current practice, different releases do have common modules but they are treated in isolation.

Different releases are shipped to different customers. If both customers complain about same issue, the issue needs to be resolved in both of the releases. It is clear from the figure 3.1 that each release is treated in its individuality despite the fact that each release shares some commonality. The delta model emphasizes to identify the common modules in releases and share them.

Each release has some commonality with the other release. Moreover, each release has some features not available in other releases.

It is shown in figure 3.2 that the common modules in releases are identified and shared so that changes in one releases are also incorporated in the other release where those features are shared.

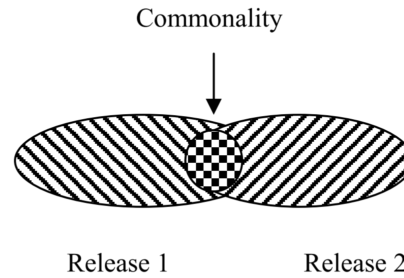
For test purposes, three releases are chosen as candidate. These releases are tested for the delta model. The focus is to test the releases for common bugs. When the modules are shared among releases, a common bug fixed in release 1 will be automatically fixed in release 2 as well. The releases are tested to eliminate the common bugs using the model and it does not just work for two releases, it can work for many releases sharing some common modules.



Release 1 Release 2

Figure 3.1: Releases relationship (before model implementation)

Consistency is automatically improved as common issues are fixed in multiple releases. In the existing system, a change made to release 1 need to be incorporated in release 2 as well manually. A lot of in-consistencies are introduced as of the manual process. These inconsistencies are removed in the test environment because a change made in a shared module will be automatically incorporated in the all of the concerned releases.



Release 1 Release 2

Figure 3.2: Releases relationship (after model implementation)

3.2 Common Bugs in Common Modules

Common bugs in common modules are easy to fix. Reason being the shared portion between releases is taken out. Whenever, a change is made in shared potion in one release this change is automatically incorporated in the other release. The shared portion among release is highlighted in Figure 3.2.

The test has to be conducted on some old releases so that it could be seen that what effect has been made by using the delta model. Three releases are selected being the candidate release. Three releases are selected so that we can make two set of releases to compare with the old data for common bugs. The selected releases are 103, 104 and 105.

Each release set has been taken out with old data in which the common bugs are not fixed. Using the delta model, the bugs are fixed using the fixing code taken from old releases. In this way, using the model, the bugs are fixed to see the effect of the model. Using the model we are able to fix a common bug in more than one releases with effort applied once.

3.3 Common Bugs in Isolated Modules

A common bug can be present in two releases in isolated modules. This means that the module containing bug is not shared. Now whenever a change will be made in one release, it should be made in the other release as well. This change need to be incorporated by the maintenance team. The isolated modules are shown in Figure 3.1. This is the portion in a release that has no commonality.

3.4 Bugs Verification

Bug verification is done in the release being built using the delta model. Common bugs are tested for fixation. A release is built using the changed code of the release that has fixed the bugs. The release is tested against the test cases devised to find that either the bug is fixed or not. The data is again collected for the three (3) candidate releases and then an analysis is performed.

4 HYPOTHESIS

We have proposed three research hypotheses (different from statistical hypothesis) based on the collected data analysis to check the three quality parameters (Stability, Consistency and Maintenance Time) for the delta versioning model. The common bugs have a significant impact on stability, consistency and maintenance time apparently and delta versioning model will hit the common bugs in common files or modules among different releases.

4.1 Hypothesis 1

The Delta versioning model will ensure the stability in each release as if all reported common bugs are fixed in each release, this is again idealistic myth. We want to check model performance for stability, and stability depends on the number of common bugs. So stability is dependent variable on number of common bugs (ignoring individual bugs), we have urged to get the impact of model on stability after implementation:

Null Hypothesis (Ho): "There will be no significant change in stability of a release through fixation of common reported bugs with the help of Delta Versioning Model"

Alternate Hypothesis (H1): "There will be a significant change in stability of a release through fixation of common reported bugs with the help of Delta Versioning Model"

Here significant change indicates increase in stability by decreasing the number of common bugs (increment in bugs is not a significant change).

We have formulated a formula based on the threshold values for number of bugs to check significant impact of model on stability.

Let's denote Stability with S, number of fatal bugs with B_{fat} , number of major bugs with B_{maj} and number of minor bugs with B_{min}. The formula is:

$$S = \frac{[(C_{fat}-B_{fat})(C_{maj}-B_{maj})(C_{min}-B_{min})]}{3}$$

Where

C_{fat} = Constant for threshold value for fatal bugs where $C_{fat} = 1$

C_{maj} = Constant for threshold value for major bugs where $C_{maj} = 10$

C_{min} = Constant for threshold value for minor bugs where $C_{min} = 30$

B_{fat} = Number of fatal bugs where B_{fat} = 0, 1 and if >1 then equivalent to 1

B_{maj} = Number of major bugs where B_{maj} = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 and if >10 then equivalent to 10

B_{min} = Number of minor bugs where B_{min} = 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30 and if >30 then equivalent to 30

The above formula shows that bugs have linear relation with stability and have direct impact on the value of stability. It is really a hard decision to get a linear relationship of stability and reported bugs as measuring stability is a complex task, here in formula we have ignored many factors e.g. customer's view for significance of a bug.

If there is one bug or more than one fatal bugs in a release then stability value will be zero, If there are ten or more than ten major bugs then stability of release will be zero and if there are twenty or more than twenty minor bugs then stability will be zero. The number three (3) dividing the formula is the constant to rationale the formula to 100%.

We used simple data analysis approach for hypothesis validity. We have not used any statistical test because of parameter values variation as shown in above formula e.g. the threshold value for major bugs is 10 in any release but more than 10 bugs will also be considered as 10.

4.2 Hypothesis 2

The model will ensure the consistency in all releases as if all common bugs reported are fixed in all impacted releases. Consistency is a trivial case as compared to stability. If there is a common bug in two releases' common file or common module, and if one has gotten a fixation for the bug and the other have not got the fixation then both releases will be inconsistent to each other (initial they were consistent as both

releases were containing the common bug in their common file or common module). The delta version model will increase the consistency as the fixation for a common bug in common file or module of two releases will be made on both sides (both releases).

Null Hypothesis (Ho): “There will no significant change in consistency of releases through fixation of common reported bugs with the help of Delta Versioning Model”

Alternate Hypothesis (H1): “There will be a significant change in consistency of releases through fixation of common reported bugs with the help of Delta Versioning Model”.

We have used simple data analysis approach for the verifying the hypothesis.

Release	Before Model Implementation				After Model Implementation			
	Reported Bugs			Stable	Reported Bugs			Stable
	Fatal	Major	Minor	%age	Fatal	Major	Minor	%age
105	0	8	16	4%	0	1	16	42%
104	10	17	13	0%	5	7	11	0%

Table 6.1: Stability hypothesis validation

4.3 Hypothesis 3

The model will reduce the maintenance time to half in terms of transactions as if transactions required to fix common bugs in releases is reduced as compared to transaction to fix common bugs in each release separately.

Null Hypothesis (Ho): “There will be no significant change in maintenance time to fix common reported bugs with the help of Delta Versioning Model”.

Alternate Hypothesis (H1): “There will be a significant change in maintenance time to fix common reported bugs with the help of Delta Versioning Model”.

Here significant change means decrease in number of transitions which indirectly indicates the decrease in maintenance time. We have devised a formula which can predict expected number of transition for fixation of common bugs in common files or modules of different releases, this formula is devised on the idea that model will perform only one time effort (transitions) to fix a common bug in common files or modules of different releases.

Let’s denote the number of transitions with T, sum of all number of transitions to fix common bugs in common files or modules of different releases with ΣT_r where T_r is number of transitions to fix a common bug in each release, and total number of releases having common bugs in common files and modules with N_r.

$$\mathbf{T} = \frac{\Sigma T_r}{N_r}$$

Where

T = Number of transitions to fix common bugs in common files or modules of different related releases after the implementation of the model.

T_r = Number of transitions to fix a common bug in each release

N_r = Number of releases having common bugs in common files and modules

We have used simple data analysis approach to check the validity of hypothesis.

5 HYPOTHESES METRICS

This is an extra analysis effort to check the precision of the deduced results based on the data collected after the implementation of the model. We have tested hypotheses for delta versioning model against three metrics as given below:

$\Sigma C_{CV} / \Sigma B_{CV}$ (Total common bugs fixed in each release / Total common bugs reported in each release)

Above metric is used to check the validity of “hypothesis 1”. If value tends to be zero then “hypothesis 1” will be false and if value tends to be 1 then hypothesis will be true.

$\Sigma C_{CTV} / \Sigma B_{CTV}$ (Total common bugs fixed in all releases having common bugs / Total common bugs reported in all releases having common bugs).

Above metric is used to check the validity of “hypothesis 2”. If value tends to be zero then “hypothesis 2” will be false and if value tends to be 1 then hypothesis will be true.

$\Sigma TC_{CTV} / \Sigma TSC_{CV}$ (Total time of common bugs fixed in all releases / Total time of common bugs fixed separately in each version)

Above metric is formulated to check the validity of “hypothesis 3”. If value of time metric resulted to be 1 the “hypothesis 3” will be false for time, and vise versa.

Release	Before Model Implementation				After Model Implementation			
	Common Bugs			Consist	Common Bugs			Consist
	Fatal	Major	Minor	YES/NO	Fatal	Major	Minor	YES/NO
105	0	7	0	Yes	0	0	0	Yes
104	5	10	2	Yes	0	0	0	Yes
103	3	4	1	Yes	3	4	1	Yes

Table 6.2: Maintenance time hypothesis validation

Model has been implemented and monitored under test environment; metrics data has been collected and analyzed.

6 DISCUSSION

6.1 Hypotheses Validation

6.1.1 Stability Hypothesis:

Delta versioning model could be tested only for three releases 103, 104, and 105. Now we will check the stability of each version before implementation of the model.

Release	Before Model Implementation	After Model Implementation
	Number of Transition	Number of Transition
105	98	49
104	238	119
103	112	112

Table 6.3: Consistency hypothesis validation

The tabulated data in table 6.1 shows that the values of stability of releases 104 and 105 are 0%, 0% and 4% respectively before implementing the model and after implementation of the model values are 0%, 0% and 42% respectively. So there is an improvement in stability in two releases 104, and 105 although the stability of 104 is 0% and this is due to its own fatal bugs (if we just look at the common bugs fixation ratio then ratio is 100%). By looking at the data we can reject the null hypothesis, but with some precincts due to limited tested data.

6.1.2 Consistency Hypothesis

Now we will check the consistency of each version with other version before and after implementation of the model.

We could just test consistency between 104 and 105, and 104 and 103. We could not test for 103 and 102 so common bugs remained untouched.

Above tabulated data shows that there is no significant change in consistency, though common bugs have been removed completely, so model has not caused any inconsistency among different versions. So looking at the above data, we reject the alternative consistency hypothesis which claims for a consistency improvement.

One aspect should be noted here that we have not caught a single case with any inconsistent releases having common bugs before implementation of the model.

6.1.3 Maintenance Time Hypothesis

Now we will check the maintenance time in terms of number of transition to fix the bugs before and after implementation of the model.

From above tabulated data, we can see that the number of transitions to fix common bugs in common files or modules of two different releases has become half of the number of transitions before implementing the model, which as accordance with the formula. So we can reject the null hypothesis. We have not tested common bugs' fixation of 103 and 102.

6.2 Metrics Verdict

ΣC_{CV} / ΣB_{CV} (Total common bugs fixed each release / Total common bugs reported in each release)

No	Releases	ΣCCV / ΣBCV
1	105	7/7 = 1
2	104	17/17 = 1
3	103	0/8 = 0

Table 6.4: Hypothesis 1 Metric Validity

Here we have taken the sum of all fatal, major and minor bugs. Result has proved the validity of "hypothesis 1".

ΣC_{CTV} / ΣB_{CTV} (Total common bugs fixed in all releases having common bugs / Total common bugs reported in all releases having common bugs)

No	Releases	ΣCCTV / ΣBCTV
1	105	14/14 = 1
2	104	34/34 = 1
3	103	0/16 = 0

Table 6.5: Hypothesis 2 Metric Validity

Here we have taken the sum of all fatal, major and minor common bugs in set of two releases (105 and 104, 104 and 103). Result has proved

the validity of “hypothesis 2” as model keeps the consistency.

ΣTC_{CTV} / ΣTSC_{CV} (Total time of common bugs fixed in all releases / Total time of common bugs fixed separately in each version)

No	Releases	ΣTCCTV / ΣTSCCV
1	105	49/98 = 0.5
2	104	119/238 = 0.5
3	103	112/112 = 1

Table 6.6: Hypothesis 3 Metric Validity

Result has proved the validity of “hypothesis 3” as transitions have been reduced to half.

7 LIMITATIONS OF STUDY

There are many limitation of the study; limitations are related to model, prior study, test environment and many other factors. The limitations are discussed as follows:

Customer point of view for the major and minor bugs in stability is missing in stability formula, we have chosen a specific threshold value for bugs, and more work can be done to use prioritized bugs.

There were thirty external releases and hundreds of internal releases we could just get data for only ten releases.

Test Environment is very small, even model works perfectly for common bugs and supports our hypotheses positively but tested data is only for three releases.

We have just focused on the common bugs in common files or modules of different as studied model has only access to only common bugs.

For stability, individual bugs are as important as common bugs but as we said earlier that due to model limitations we could only fix common bugs for stability.

8 CONCLUSION

The focus of the case study was to devise a model to handle bug fixes for multiple clients using the same product but different versions. The data analysis elaborates that different releases should share common modules among themselves so that whenever some bug is fixed in one release; it gets automatically fixed in the other release. Doing so will help minimize maintenance time / cost. Reason being a bug will need to be fixed only once. If this behavior is seen from a product’s perspective, it’s a huge gain towards maintenance.

For a particular situation, we have investigated the impact of number of bugs on the stability, consistency and maintenance time of the product by analyzing the collected data. In addition delta model has been proposed to be utilized for taking the shared modules among releases.

The product under consideration consists of around 30-40 releases belonging to Industrial Automation industry. The scope of the study was to take the data of 10 releases and out of the 10 releases, 3 releases were selected as candidate to be inducted into a test environment to study the proposed model effects.

To test our hypothesis, we have used simple statistical technique to analyze the stability of the release after taking it through the delta model. The results do not represent just a fixed trend rather it shows that common bugs can be taken care of easily. The time required for maintenance is reduced, more stable and more consistent release is produced using the proposed model.

The essence of making use of common modules cannot only be applied to multiple releases of a product rather it could also be applied to multiple products. Suppose, there is a BASE product and on the basis of the BASE product, a FLAVOR product is being developed. The FLAVOR product has extra features on top of the BASE product. In order to achieve that common modules approach can be employed in FLAVOR product so that whenever there is a change in BASE product, it will be automatically reflected in the FLAVOR product with no extra cost. The same Delta model can be used for this scenario. We have not talked about it in our case study. But BASE-FLAVOR relation needs to be researched more.

9 FUTURE WORK

Future study can be conducted to try to use the same methodology on projects that belong to different domains. The projects that usually make use of a lot of reusable components can be tested against this model to examine its effectiveness.

Moreover, delta model effect must be examined while using different products. Multiple products with multiple releases handling should be checked. Moreover, a lot of other parameters can be tested along with stability, consistency and maintenance time. The customer definition of bug’s severity can be given more importance. Maintenance time can be

calculated in actually man hours to see the model effect for any improvements.

10 REFERENCES

[1] "*Release Management*", by British Educational Communications and Technology Agency, http://www.becta.org.uk

[2] "A Case Study of Configuration Management with Clear Case in Industrial Environment" by Ulf Asklund & Boris Magnussun, Proceedings of SCM7, International Workshop on Software Configuration Management, R. Conaradi (Ed.), Boston, May 1997.

[3] "Towards Virtual Software Configuration Management – A Case Study" by Tua Rahikkala, Technical Research Center of Finland, 2000

[4] "Transaction oriented Configuration Management" by Peter Fieler & Grace Downy 1990, SEI Carnegie Mellon University, 1990

[5] "IEEE standards for Software Project Management Plans" 1998

[6] "The Past, Present, and Future of Configuration Management" by Susan A. Dart, SEI Carnegie Mellon University, 1992.

[7] "Configuration Management (CM) Plans: The beginning to your CM solutions" by Nadine M. Bounds & Susan A. Dart, SEI Carnegie Mellon University, 1993.

[8] "Configuration Models in Commercial Environments" by Peter H. Feiler, SEI Carnegie Mellon University, 1991.

[9] "Product-line development requires sophisticated software configuration management" by Dijon, FRANCE W. Schafer, IEEE Computer Society http://csdl.computer.org, 1996

[10] "Configuration management with logical structures" by Yi-Jing Lin, IEEE Computer Society http://csdl.computer.org, 1996

[11] "A Software Configuration Management Model for Supporting Component-Based Software Development" by Hond Mei, Lu Zhand, Fuqing Yang, ACM SigSoft 2001.

[12] "Software Release Methodology" by Michael E. Bays, Prentice Hall PTR, 2003

[13] "An Integrative Model for Configuration Management and Version Control" by Lars Bendix, 1996

[14] "Managing Software Process" by Watts S. Humphrey, Pearson Education Inc, 2002