Defect Analytics: OLAP for Delivery Excellence

Suhas Mallya
MindTree Limited

Vidya Rao
MindTree Limited

17/4C, Rupena Agrahara, Hosur Road
Bangalore 560068
India
{suhas_mallya, vidya_rao}@mindtree.com

Abstract

Most defect-tracking systems provide canned reports and the ability to create custom reports. These reports provide a snapshot of the defects in the system and allow different types of filters to be applied. However, all these reports provide limited insights into the defects of a product and often, the reports are operational or tactical in nature. Defect Analytics uses OLAP on a defects repository to produce actionable insights that cannot be obtained from the standard reports generated through a defect tracking system.

In this paper, we develop the idea of Defect Analytics and present examples to illustrate its significance along with some initial work we have done to translate these concepts into a tool.

1. Introduction and Significance

A defect-tracking system is a commonly used but important CASE tool in any software development environment. As a product goes through multiple releases and multiple versions over a period of time, the defect repository becomes a rich record of the "lifecycle journey" of the product.

Most defect tracking systems provide reporting capabilities – canned reports as well as the ability to generate custom reports. However, these are characterized by two shortcomings: one, they are restricted to a given set of release(s) and two, they are based on snapshots of the defect repository (i.e., the state of the defects at the time the report was generated). In other words, these reports do not provide release-by-release comparison and they do not profile the lifecycle of the defects. Release-by-release comparison of defects helps identify patterns and trends that may prompt a deeper causal analysis than may have been identified by standard reports. For example, hypothetically speaking, a standard reports may reveal that a particular component is the single largest source of bugs in a particular release. While this is useful information, if it is established that the component has consistently been the single largest source of bugs for the last 5 releases, it puts a different light on the situation. If it is additionally determined that in the last 8 releases, this component has been troublesome from the 4th release onwards, this merits some investigation into what happened in the 4th release. Perhaps the product was ported to Linux from a Windows platform or was extended to a Linux platform? Perhaps a particular library does not work as well on Linux? This could prompt a rethink of the design. While standard reports will help the project manager focus on the number and clustering of bugs around the component, it will be in the context of meeting the delivery deadline. Scratching the surface will help identify longer-term solutions.

Similarly, analysis of the lifecycle of the defect could offer other interesting insights. As an example, consider that the turnaround time for closure of bugs in a particular release is 10 days, where the standard workflow/lifecycle of a bug is OPEN → ASSIGNED → FIXED → CLOSED. An analysis of the state transition throughout the lifecycle of a bug could reveal for instance, that within these 10 days, 5 days were spent for the bug to go from 'OPEN' to 'ASSIGNED'. In this case, the simple action of delegating to more than one person, the responsibility of assigning bugs, could potentially yield a significant improvement in the turnaround time of closure.

Thus, the significance of Defect Analytics lies in the ability to provide release-on-release comparison and lifecycle state-transition analysis. The specific parameters to be compared or profiled can be appropriately tailored on a case-by-case basis within the broad framework that we present in this paper.

15th International Conference on Management of Data
COMAD 2009, Mysore, India, December 9–12, 2009

2. Application and Usage of Defect Analytics

Defect Analytics can be used in the following areas:

1. Delivery Excellence – through identification of trends & patterns that could potentially derail smooth delivery and execution, and through identification of operational & process inefficiencies that would easily translate into hidden costs

2. Sustenance Engineering – by providing concrete data and a scientific approach for prioritizing areas that require attention and for determining team composition.

A key thing to note about the outputs of defect analytics is that unlike other metrics-driven system, most of the outputs of defect analytics are consultative in nature, rather than prescriptive. They must be understood in the larger context and not all data about the larger context may be available within the defect repository. For instance, a drop in developer productivity expressed in LOC/hour is an absolute measure that makes a statement on the productivity. Similarly, a file-to-fix ratio is an absolute measure that speaks about the overall relevance of bugs logged (or attention paid to bugs logged). However, in the context of defect analytics, a spike in the number of bugs in a particular component over the previous release could be attributed to several factors like a design change, enhanced test coverage or addition of new features. Such factors are not captured in the defects repository but must be kept in mind when interpreting the report that shows a spike of bugs for that particular component.

3. Sample Outputs

This section presents a small subset of the outputs that are possible through defect analytics. As stated earlier, the specific reports generated can be tailored to meet specific requirements and will be based on the nature, quality and volume of source data available.

3.1 Frequency Distribution

This report shows the weekly frequency of filing and closing bugs. It also plots the mean for # of defects filed per week and # of defects closed per week. This helps assess whether QA (filing+closing) has been progressing at a steady pace or whether there are spikes – triggered perhaps, by some external event like a review or an approaching deadline. Needless to say, clustering around the mean is a healthy sign; too many spikes above or below the mean are a cause for concern – or at least, for investigation.

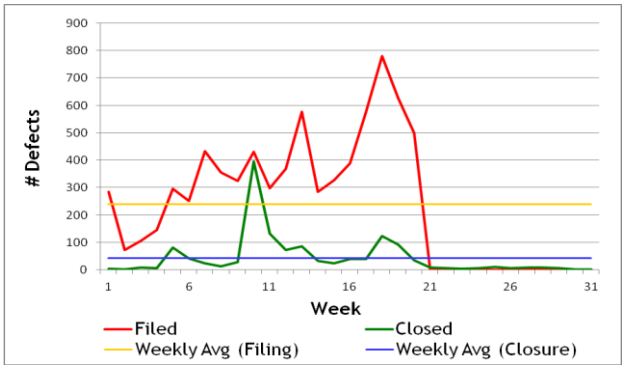

Fig. 1. Average bug filing and closure frequency

In the illustration shown here, there are significant spikes in weeks 9, 12 and 18 and there is consistently a large gap between bugs filed and bugs closed. If a similar graph for a subsequent release is superimposed on this, it may reveal interesting patterns – especially if the spikes are in the same relative period of the total test cycle.

3.2 Filed-to-Rejected ratio

This report compares the number of bugs filed, with the number of bugs rejected, across multiple releases. A high filed-to-rejected ratio is unhealthy because it suggests that a large number of non-bugs were flagged as bugs. Comparison across releases helps determine the trend.

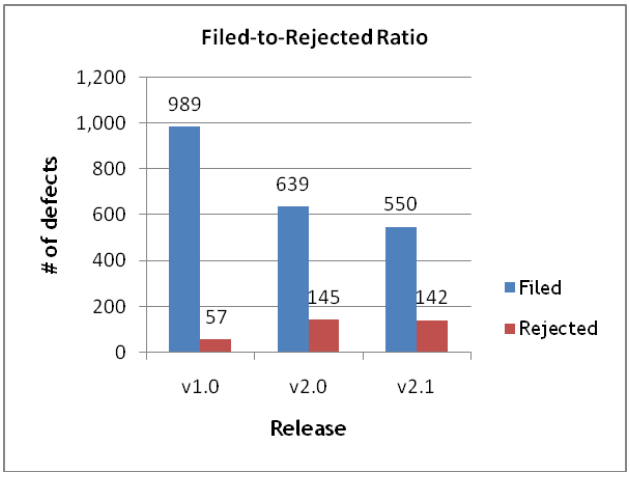

Fig. 2. Filed-to-Rejected Ratio across releases

In the illustration above, we are seeing an unhealthy trend: while the total number of bugs is decreasing across releases, the total number of rejects is going up and definitely going up as a percentage of total bugs.

3.3 Average closure by reason

This report analyses "Closed" bugs to profile how bugs were actually "closed". In most defect tracking systems, "CLOSED" is the final state, but each closure may be for

a variety of reasons. A bug that is closed because it has been fixed and the fix has been accepted by the test engineer is a "true closure" – anything else is a candidate for investigation, especially if they are in large numbers.

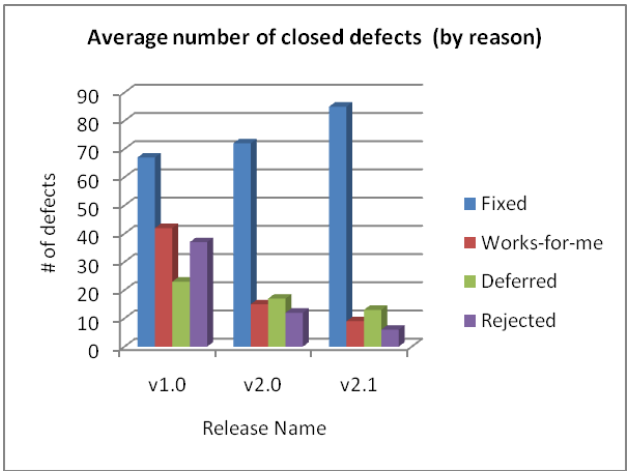

Fig. 3. Average defect closure by reason

In the illustration above, "FIXED" is the single largest reason for all closures, which is a good sign. However, v1.0 reflects a high number of "Works-for-me", which suggests a non-reproducible bug. These are always a source of concern because they can always spring nasty surprises. The other good indicator though in this illustration is that the number of "Works-for-me" has significantly reduced in subsequent releases.

3.4 Distribution of Defects across components

This report profiles the total defects filed, across the various components of the product for a given release, and for each component, shows the break-up by priority. This helps focus attention on the right area and prioritize accordingly.

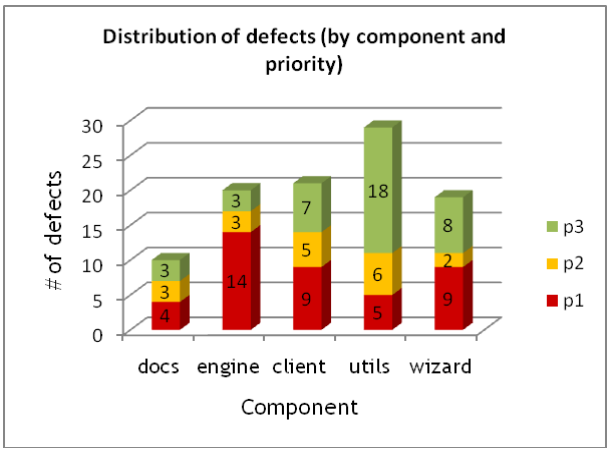

Fig. 4. Distribution across defects, by priority

In the illustration above, even though "utils" has the highest number of defects, the real focus has to be on the "engine" component. This is because over 60% of the defects in "utils" are P3 priority, while 70% of the defects in engine are P1 priority. The absolute count too, of P1s in "engine" is much higher.

4. High-level architecture

The Defect Analytics solution being developed is viewed and designed as an extensible solution that comprises a set of generic analytical reports (like the ones illustrated in Sec. 3), the data model to support it and a set of scripts to load the data model from different sources (i.e., different defect repositories).

The extensibility of the tool lies in its ability to support additional defect repositories with little incremental effort and no change to the existing system. To support an additional defect repository, the system assumes that a dump from the repository will be made available in a delimited text file. A new loader would have to be developed specific to that data repository, which loads the extracted data into the data model for the defect analytics tool.

In its current state, the tool supports Bugzilla and Microsoft TFS and outputs are generated in MS Excel by directly accessing the data model through an ODBC connection. The tool can easily be extending to support different defect repositories as well as by using other tools such as Jaspersoft for generating end-user reports.

The following is a high-level architecture of the current system. As is explained in subsequent sections, the data model for the staging and reporting areas are specifically designed to provide a certain degree of flexibility.

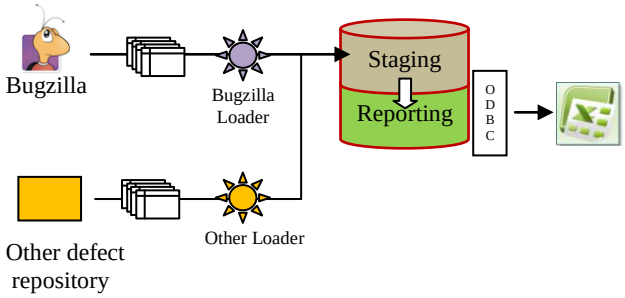

Fig. 5. High-level Architecture

The current system uses Oracle 10g Express for the staging and reporting databases. This was chosen based on in-built support for analytic functions which offer ease of use and better performance compared to conventional SQL programming. However, this will be ported to

MySQL or Postgres so as to have a solution based entirely on an open-source stack. The ETL processes use a combination of Perl, Oracle SQL*Loader and PL/SQL. The front-end reports can be generated using any ODBC-enabled tool, including Excel (all outputs illustrated in this paper have been generated through Excel).

5. Methodology

The typical attributes of a defect are the source of origin, the component to which it belongs, the tester who filed it, the priority and severity, the developer who works on the defect and the states in which it passes through from creation to closure. Of the above attributes, some of them such as the priority of a defect or the developer who worked on it may be subject to change.

The typical states of a defect considered here, through which it passes are: OPEN, WORKING (or ASSIGNED), FIXED, TESTING, and CLOSED. The typical state transition (a.k.a. lifecycle or defect workflow) that has been considered is illustrated below:

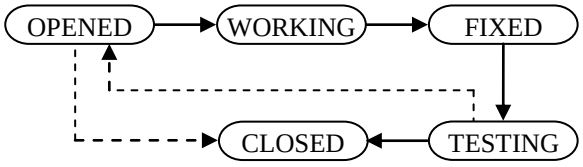

Fig. 6. Lifecycle considered for initial work

It is important to note that the specific states listed here as well as the state transition shown can differ from organization to organization, from project to project and could vary over time. This assumes significance because the data model to support defect analytics must cater to such vagaries.

For the release-wise defect trend analytics, we consider the final value of attributes as at the end of the release. However, we track the changes made to the defect attributes and the states to enrich our defect analytic metrics. The data model has been designed to provide for flexibility for definition of the lifecycle state machine, product components and product releases, such that organization-specific or project-specific definitions for the following can be accommodated in the analysis.

6. Data Warehouse Design

The data model is built using dimensional modeling technique and uses a star schema to support the overall objective of providing release-on-release comparison and lifecycle transition analysis.

We consider the following 9 shared dimensions: *Time, Release, Component, Source, Priority, Severity, State, Resolution and Team Member* and the following 2 fact tables: *Defect Fact* and *Release Fact*.

Since we capture all the required attributes of a defect in the Defect Fact table itself, we decide to keep the 'Defect Id' as a degenerate dimension (DD)

The grain of the Defect Fact table is '*facts over time by defect by component by source by priority by severity by state by resolution by tester by developer*' and that of the Release Fact table is '*facts over time by release by component by source by priority by severity by state by resolution by tester by developer*'. Such a design allows us to roll-up to a higher level by release and drill-down at a finer granular level by the various defect attributes. For example, we can view the number of defects by release or we can view the number of defects by release by source by priority by tester.

6.1. Schema

The Defect Fact table combines the properties of the transaction fact (a defect is recorded only if it is filed) and the accumulating snapshot fact (represents the entire life of the defect) to capture the final value of the attributes as at the end of a release and the defect workflow. It also aggregates facts at the defect level. Note that accumulating snapshots represent a process that evolves over time, whereas in our fact table, we record it at one shot at the end of the product release. Nevertheless, modeling it in this way, allows us to efficiently track the workflow history.The Release Fact table is modeled as a periodic snapshot fact and stores the aggregation of defects per release drawn from the Defect Fact table.

The facts are set to a default value of NULL so that aggregate functions such as SUM, COUNT etc. can handle them properly. The pre-computed values and aggregates that the Defect Fact table stores are: the number of times a defect was reassigned, the number of times a defect changed its priority, the date on which it was opened, fixed, tested and closed and so on. The pre-computed values and aggregates that the Release Fact table stores are: number of defects, number of defects reassigned, number of defects that changed priority, the lag (in days) between the day the defect was opened to the day it was closed and so on.

A release trend analysis would particularly need all data from all the dimensions in the query. This requires an outer join between the fact and all the corresponding dimensions of interest. The result should be dense on the Release dimension and within that on all the dimensions referenced in the user query. We achieve this using the 'partitioned outer join' feature of Oracle 10g.

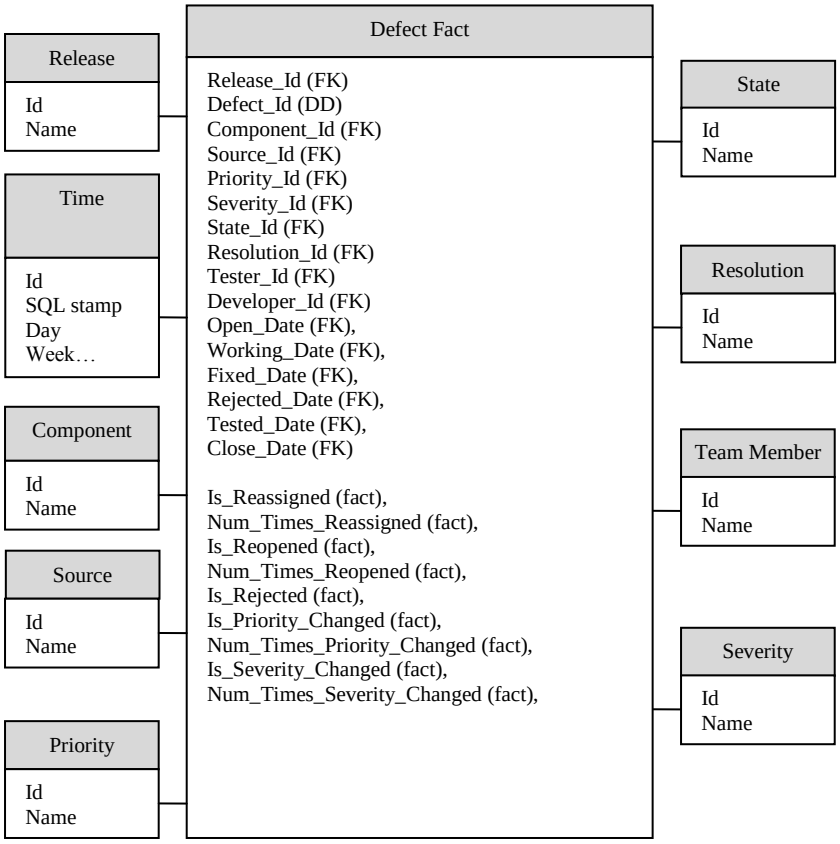

Fig. 7 – Data model for the "Defect" fact

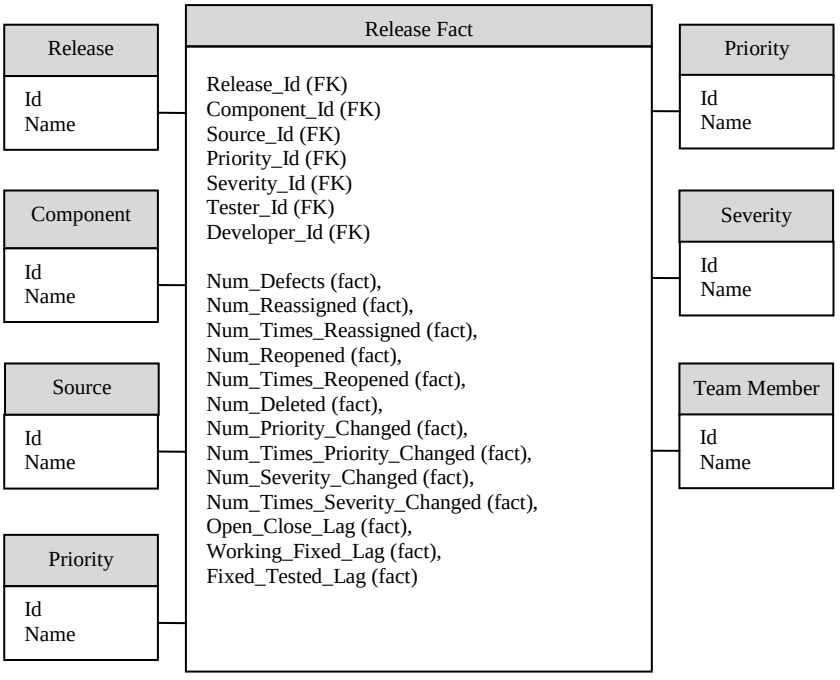

Fig. 8 – Data model for the "Release" fact

6.2. ETL

As shown in Sec. 4, data is extracted from the defect repositories (source) and loaded into a staging area from which it is finally loaded into the reporting database. The current solution includes the defect repository-specific loader but not the extraction of source data.

We track changes in the attributes of a defect (such as change of priority, change of assignee etc.) in the staging phase using the 'tuple versioning' mechanism. So, we capture two values for change: start date and end date. The tuple that has the final change has a valid value for a start date and a NULL value for the end date. A tuple with the old value has a valid start date and the end date equal to the start date of its successor. The structure of one of such prone-to-change defect attributes, Defect Priority, would then look like:

- Release_Id (FK),
- Defect_Id (DD),
- Priority_Id (FK),
- Start_Date (FK),
- End_Date (FK)

6.3. Reports

The data model for the reporting database contains the data model shown in Sec. 5.1 as well as a set of views that make it easy for end-users to extract the kind of data required to generate their reports.

With the above schema, we can define the defect analytics queries easily and store them as view objects in the repository.

Using an ODBC bridge, any ODBC-able front-end can be used to generate reports. In the current version, we are using only MS Excel.

7. Roadmap and Future Work

1. Define a set of "magic reports" that will serve as *de facto* indicators of the condition of the quality process and overall product health and implement them in the defect analytics solution.
2. Model and cater to different business definitions of lifecycle state machine, components, release definitions, etc.
3. Capture additional external data that will allow for richer interpretation of the results (e.g., for the Release dimension, capture data about proposed release date, actual release date and profile of changes made in the release, as a delta over previous releases).
4. Migrate the solution architecture to an open-source stack, including porting the staging and reporting database to MySQL or Postgres.
5. Enhance the ETL mechanism by bringing in a greater degree of automation – e.g., periodic loading of defects from the defect repository.
6. Create a front-end "Control Panel" application for setup, configuration and administration of the defect analytics tool

8. Conclusion

"Defect Analytics" is an innovative application of an established technology (Data Warehousing, OLAP) in an even-more established process (defect logging and tracking).

Our initial work of developing this concept and translating the key elements into a software tool only scratch the surface of the full potential of defect analytics. Experiences from initial implementations of this solution in real-life SDLC scenarios must be ploughed back to enrich the solution and eventually take up some of the roadmap items listed in Sec. 6 of this paper.