




# Agile Record

The Magazine for Agile Developers and Agile Testers





# Distributed Agile – The Most Common Bad Smells

by Raja Bavani

In the programming world, the term 'bad smell' refers to negative characteristics of code that could adversely impact design and code quality. Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behavior. Its heart is a series of small behavior preserving transformations. Refactoring improves the quality of design and code. In a broader context, software development and testing life cycles do signal bad smells or negative characteristics from time to time, and from project to project. Recognizing such bad smells and responding to them at the right time is essential to keeping projects on track. In our experience, Distributed Agile Software Development projects involve many nuances that could result in tricky situations that impact the satisfaction levels of stakeholders. Refactoring of life-cycle processes is necessary to tune the delivery engine towards delivering quality products. This is not a one-time activity. It needs to happen continuously at regular intervals, and the way it is done can differ from project to project.

## Risks and Bad Smells

Risks are uncertainties that could affect project performance adversely. For example, a risk could impact project costs (because of slipping schedules or effort variance) or affect the quality of deliverables and reduce customer satisfaction. In most cases risks are identified before they occur. On the other hand, bad smells are felt or experienced in real-time. They are an indication of project risks or greater probability of producing mediocre results. Mediocre results attributed to average quality gradually become an unexpected bottleneck during the product life cycle. For instance, mediocre results could impact a business critical situation related to product release or migration. This can be avoided if we recognize and fine-tune the corresponding processes and also apply corrective actions. Any causative process that relates to a bad smell is a candidate for refactoring. Recognizing and responding to bad smells facilitates the timely refactoring of processes.

Projects need to take calculated risks. Also, projects need to respond to bad smells in a timely manner. Good examples to illustrate these facts are: a) Scheduling a training program on a project-specific tool to enhance the skills of team members for better productivity, and b) Improving the query resolution process when there are pending queries or too many communication steps to resolve a query. The former is an approach towards risk mitigation, and the latter is a response to a bad smell that needs immediate action. This makes it evident that the ability of the project teams to recognize and respond to bad smells gives definite added value in Agile projects. It helps not only to avoid certain risks, but also to apply continuous improvements to nullify the probability of mediocrity and hence to provide predictable deliverables of high quality.

Presented below is a set of ten bad smells that are most commonly experienced in Distributed Agile Software Development.

## 1. Integration Nightmare

This occurs when product integration becomes messy, which results in a schedule slippage. As a result, the level of predictability becomes very low.

Integration issues consume significant effort, especially when the code base involves product modules undergoing maintenance as well as newly developed interdependent modules. Timely planning and corrective actions are crucial to mitigate delays in resolving integration issues. When an integration strategy is not effective and efficient, the project quality suffers because of unexpected delays in product integration. Continuous integration is not a destination but a journey. The strategy to accomplish continuous integration cannot be the same for all types of project. A wiser approach during the first few cycles is to have a dedicated team of engineers that focuses on integration. The responsibility of this team should be to decide when to stop everything else to fix integration issues as a priority, and to ensure that integration efforts during subsequent iterations are optimized. Also, it is

essential to budget for the integration activity depending on the complexity of the product, and to resolve integration issues when team members across sites are available for collaboration and issue resolution.

## 2. The Vicious Cycle

Software projects encounter this bad smell when the number of new defects is on the rise from iteration to iteration. Our experience says that too much aggression in catching up with delivery requirements results in quality issues. If the number of defects in subsequent deliveries is on the rise, it is time to recognize it as a bad smell and respond to it. Buggy deliveries make the team stretch in implementing new functionalities as well as fix defects during subsequent iterations. Distributed development is prone to this syndrome. Disciplined personal practices and continuous focus on enhancing product knowledge are critical in eliminating this syndrome in Distributed Agile environments.

Understanding the quality of deliveries in quantitative terms is the key to recognizing this bad smell. Periodic quantitative status checks help in knowing the trend of defects injected in each delivery. When there is a trend of growing defects, the team needs to be involved in analyzing the nature of defects, finding the root causes and implementing corrective and preventive measures. This tends to lead Agile practitioners towards process orientation on an as-needed basis and provides valuable inputs.

## 3. Uncertain Assumptions vs. Convenience

In distributed projects, uncertain assumptions tend to linger without any action on validation or clarification. Assumptions made during the initial stages of projects go unnoticed until the end-users raise issues after final delivery. There has to be a balance between 'Uncertain Assumptions' and 'Convenience' when we go Agile. We need to make certain uncertain assumptions to make progress. However, at regular intervals we need to clarify or validate these assumptions and make timely corrections. The impact of unresolved 'Uncertain Assumptions' on testing and Product Quality could be fatal. Eventually, customers' perception of product quality would remain negative due to their initial experience during User Acceptance Testing. Besides, depending on the magnitude of such assumptions the overall product testing activities may have to be repeated in part or full in order to ensure a successful release. Finally, the product release may not happen as planned. In distributed environments the chances of executing projects with 'Uncertain Assumptions' are higher, and hence an additional level of status check is required to have assumptions validated or clarified at regular intervals. In order to avoid this from happening, prepare and review the list of assumptions at regular intervals. Also, clarify assumptions and involve all relevant stakeholders in this activity.

## 4. Regression Tests – Tip of the Iceberg?

This symptom is felt when the efforts spent on regression testing grow larger than expected over a period of time and actually become an area of concern.

Agile practitioners do recommend independent QA/Testing, as it adds value to product quality. The incremental growth in the size of regression testing is one of the characteristics of Agile projects. However, in case of large projects involving product development of multiple product modules, regression testing grows rapidly and consumes significant effort compared to projects involving development or maintenance of stand-alone applications. In one of our projects we could compress the time required for regression testing by 50% using homegrown automation tools. This experience gave us an insight into the need to increase the level of automation during subsequent iterations. It also helped us in reducing manual testing effort during release cycles.

Test strategy, test planning and test automation are the key ingredients to manage regression testing effectively and efficiently. Build regression test suites and plan for regression testing from the initial stages of a project. Leverage test automation tools to optimize the efforts expended on regression testing.

## 5. Stretched Query Resolution

This happens when individual interactions stretch over multiple transactions with long pending queries.

Timely query resolutions provide clarity for Agile teams. When it comes to Distributed Agile projects, timely query resolutions become very crucial due to the geographical spread of the team and the absence of customers on-site for face-to-face interaction and query resolution. In this environment, there are times when team members start managing query resolution through emails, chats and telephone conversations instead of using a centralized query-tracking tool. First of all, it is valuable to use a centralized query-tracking tool. Next, it is important to watch out for pending queries and resolve them in time. Else, the team is forced to work with ambiguity. This is sure to impact product quality.

Perform status checks in addition to query tracking through a centralized query-tracking tool. Watch out for 1-1 interactions that show insignificant results. Facilitate the resolution of stretched queries and streamline project progress.

## 6. Ever-increasing 'Not a Bug' and 'Non-Reproducible' Defects

This can be found when every delivery is characterized by an increasing number of 'Not a Bug' (NAB) or 'Non-Reproducible' (NR) defects.

Identification of NAB or NR defects during defect classification is a natural occurrence. However, if the trend shows a growth in the percentage of NAB or NR defects, it is a bad smell, as it would involve communication among team members in discussing and confirming the classification of such defects. Generally, NAB defects indicate the need to improve the level of product knowledge among team members, whilst NR defects indicate the need to improve the thoroughness and perfection in the testing process.

Reducing the number of NAB or NR defects can be accomplished through positive reinforcement. Setting up identical testing environments and configuration management processes across

sites is necessary to control the number of NR defects. Knowledge sharing sessions are essential to accomplish the reduction of NAB defects. Periodic visits of Subject Matter Experts (SME) to share business requirements, product knowledge, product architecture and complex test conditions are essential in a distributed environment. In all our projects we budget time for knowledge sharing sessions and team meetings to discuss product functionality and implementation aspects. We encourage team members to ask questions and get them resolved on time.

In order to recognize and respond to this smell, it is required to monitor the number of defects that get classified as (NAB) or (NR) and find the root causes. Knowledge transfer sessions and team meetings to understand the product requirements and design help in reducing the number of NAB and NR defects.

### 7. Trivial Code Quality Issues

You smell this when code reviewers report trivial code quality issues.

There are two primary dimensions of software quality, namely internal quality and external quality. External quality is an attribute that relates to the end-user experience. External quality can be assessed and improved through defect prevention as well as black box testing. Issues related to internal quality could pose serious consequences in the form of unexpected naive defects, technical issues and maintenance nightmares. Poor internal quality encompasses the root causes for issues related to external quality. Thus, in order to improve software quality, internal quality must be improved.

Trivial code quality issues occur due to various reasons, such as a) introduction of new developers who do not understand the coding standards (implicit or explicit) followed by the team, or b) aggressive timelines that force team members to do quick fixes and dirty enhancements. In Agile environments we build and empower individuals to deliver quality results. A well-performing Agile team produces consistent results. Whenever there is a change, such as the introduction of new team members, there is a good chance of encountering code quality issues. Aligning new team members towards writing good quality code is very critical. This is true for pair programming, too.

In environments where pair programming is not practical, we have seen alternative techniques such as defect prevention and static analysis yield good results in improving code quality.

### 8. Inefficient QA Build

It is not a good sign when a successful QA build happens after multiple fixes and attempts. Multiple attempts to make a successful QA build reduce the time available for testing. This is a high-level impact. Besides, delays in providing a stable QA build impacts the overall mindset of the team, and such delays pose questions on the predictability of successful builds. In some of our projects, we recognized this smell during initial deliveries. We found this bad smell whenever a new product or module got integrated with the product suite. Thus this bad smell was occurring

after every 6 or 8 deliveries and would then disappear again after 2 or 3 cycles. We responded to this by collecting process improvement ideas from our leads and implementing them.

Setting up development and QA environments that are similar in all technical aspects is a must to improve the predictability of successful QA builds. Subsequent attention on product specific configuration parameters and seed data is a must to avoid unexpected crashes or product behavior in QA environments. In a distributed environment there is an additional responsibility to ensure that builds are made successfully in different environments at every site. Failure to recognize and respond to this will result in the recurrence of build issues. This means a trend in compressed QA cycles and hence a job not well done when it comes to assuring quality.

Automating the build process and ensuring that development and QA environments are similar is essential. Also, it is very important to set up development and QA environments with the right kind of seed data and configuration parameters.

### 9. No Issues or Feedback from Customer

It is definitely a bad smell when customers do not report issues or provide feedback during initial iterations. In such cases, it is highly possible that there will be a considerable number of issues or feedback that may only surface during subsequent iterations.

Providing early and frequent delivery of working software is at the heart of Agile projects, and so is obtaining early and frequent feedback from customers. Lack of attention on either of these would increase the risk of receiving disappointing results. For example, in a bimonthly delivery model with a product release cycle of 12 weeks, any slippage in the feedback process during the first few deliveries will result in multiple issues during the rest of the development process.

Early and frequent deliveries facilitate customers in understanding the product behavior in addition to ensuring the integrity of build and deployment. In our experience we got prompt feedback from our customers on the integrity of builds and deployment processes with respect to each delivery. However, obtaining feedback on product functionality was a challenging task for us in an aggressive product development environment. We collaborated with our customers in working towards obtaining timely feedback. Our customer organized product demos for some of the critical deliveries and provided us feedback. In addition to this, product owners invested time in exploring the product and provided us their feedback.

Collaboration is essential in order to respond to this smell. Absence of issues during initial deliveries is the symptom and customer collaboration to facilitate feedback right from early stages is the solution. It is paramount to collaborate with the customer in getting substantial feedback from the early stages of the development process for continuous improvement.

## 10. No Exploratory Testing or Investigation

Typically, project teams follow the traditional way of test-case-based testing and do not find time for exploration or investigation. In such cases this bad smell can be seen when tricky and hard-to-find defects are reported during product demos by customers.

Agile teams need to explore and investigate the product that they build or test. Focusing on user stories or customer requirements during the initial deliveries will be good enough to ensure early and frequent deliveries. As the team continues to accomplish development and maintenance of multiple products or product modules over several months, exploration and investigation are required to manage the product better in terms of maintenance, new development as well as QA/testing. To do this, a shift from an 'iteration-based' focus to a 'release-based' focus on development and testing is necessary. We collaborate with our customers in obtaining a broader view that provides visibility of multiple releases over several months. This makes our team understand the nature and timelines of impending releases and perform investigation and exploration on a broader perspective. With this awareness we leverage our efforts in exploring the product or investigating issues with a broader perspective.

Any approach to development, debugging or QA/testing will not yield results if it lacks exploration and investigation. Large projects that involve software product development will suffer if there is no stress on exploratory testing or investigation. It is essential to build a culture of exploration and investigation and let the team members understand the product from the end-user's point-of-view. Establishing a broader perspective of the development and release requirements to the team and shifting away from an 'iteration-based' approach to development or testing is a must to open up avenues for exploration.

### Conclusion

A methodology that embraces Agile practices for software development is not the panacea to ensure on-time and quality deliverables. A great deal of conscious monitoring is required to exploit the benefits of Agile practices, especially in distributed or virtual teams. Generalizing these bad smells and deriving best practices is not justifiable as many of them are project specific. However, some of the bad smells discussed in this article may provide insights on how to handle similar situations in software projects. In our experience, all of these bad smells provided us with a reassurance of the importance of disciplined personal practices, defect prevention, internal quality, knowledge sharing, status reviews, test automation, rigorous query resolution, customer feedback, exploratory testing and investigation. ■

### > About the author



#### Raja Bavani

heads delivery for MindTree's Software Product Engineering (SPE) group in Pune and also plays the role of Software Product Engineering (SPE) evangelist. He has more than 20 years of experience in the IT industry and has published papers

at international conferences on topics related to code quality, distributed agile, customer value management and software estimation. His SPE experience started during the early 90s, when he was involved in porting a leading ERP product across various UNIX platforms. Later he moved onto products that involved data mining and master data management. During early 2000, he worked with some of the niche independent software vendors in the hospitality and finance domains. At MindTree, he worked with project teams that executed SPE services for some of the top vendors of virtualization platforms, business service management solutions and health care products. His other areas of interests include global delivery model, requirement engineering, software architecture, software reuse, customer value management, knowledge management, and IT outsourcing. He regularly interfaces with educational institutions to offer guest lectures and writes for technical conferences. His SPE blog is available at <http://www.mindtree.com/blogs/category/software-product-engineering>. He can be reached at [raja\\_bavani@mindtree.com](mailto:raja_bavani@mindtree.com).