Automated Defect Prevention for Embedded Systems Software Development

by Wiktor Grodowski
Parasoft SA
http://www.parasoft-embedded.com

Abstract:
Defects prevention in embedded software development is a primary issue. Techniques like code static analysis has been already widely used in the industry. But there are other defects prevention techniques present in IT software development, which, if tailored and applied to the embedded environment, can significantly improve productivity with even higher quality. This paper will explain an approach to software development and management that makes quality a continuous process throughout the software development life cycle. This strategy is enabled by an infrastructure that automates repetitive tasks; tracks project status, and provides instant access to the information needed for informed decision making and process improvement. Applying Automated Defect Prevention (ADP), organizations can evolve a sustainable quality process that improves productivity delivering predictable outcomes. Due to flexibility of the ADP methodology, it can be applied to Embedded Software Development by well known practices like Static Analysis, Unit Testing, Code Review into a continuous and automated process.

Chapter 1. Introduction
Automated Defect Prevention, or abbreviated: ADP, is a practical approach to software management through process improvement. This strategy is enabled by an infrastructure that automates repetitive tasks, tracks project status, and provides instant access to the information needed for informed decision making and process improvement. Applying ADP, one can evolve a sustainable quality process that delivers predictable outcomes. ADP stands out from the current software landscape as a result of two unique features:
- Its approach to quality as a continuous process.
- Its far-reaching emphasis on automation.
It can be applied to any team, regardless of its structure, projects, or development method. It can be used from the very starting point of a project, as well as it can be phased into existing projects and processes, and applied to large and complex code bases, without overwhelming the people involved.

The focus throughout is on evolution, not revolution. Surprisingly often one may find, that the tools required to implement ADP are already there, all that needs to be done is a
small adjustment in automation. ADP governing body does not expect anyone to abandon current processes and workflows in search of a silver bullet solution that will solve all current problems. There is no such perfect development environment or development process that will be a panacea for all software development groups and projects. Consequently, rather than offer step-by-step implementation details for a one-size-fits-all solution, ADP describes the key practices that have helped many different teams improve their processes.

According to Barry Boehm and Victor Basili, the cost of removing defects from the software grows dramatically as a function of time in the development. Defect prevention not only reduces the total number of faults, but it also shifts defect discovery to early phases of the development cycle because defects uncovered in one iteration of software development are prevented from recurring in subsequent iterations. Such an improvement can have a great impact on the efficiency of software development, because removing defects in late phases of the life cycle can be very costly [1]

Chapter 2. A look at the history

Automated Defect Prevention takes its roots from Deming's[2] and Juran's[3] statistical quality control concepts. These concepts have been in use in manufacturing industries for decades, implemented initially in the automotive world, spreading to other sectors later on.

"In 1947, Deming was involved in early planning for the 1951 Japanese Census. The Allied powers were occupying Japan, and he was asked by the U.S. Department of the Army to assist with the census. While Deming was there, his expertise in quality control techniques, combined with his involvement in Japanese society, led to his receiving an invitation from the Japanese Union of Scientists and Engineers. JUSE members had studied Shewhart's techniques, and as part of Japan's reconstruction efforts, they sought an expert to teach statistical control. During June–August 1950, Deming trained hundreds of engineers, managers, and scholars in statistical process control (SPC) and concepts of quality. He also conducted at least one session for top management. Deming's message to Japan's chief executives: improving quality will reduce expenses while increasing productivity and market share. Perhaps the best known of these management lectures was delivered at the Mt. Hakone Conference Center in August 1950. A number of Japanese manufacturers applied his techniques widely and experienced theretofore unheard of levels of quality and productivity. The improved quality combined with the lowered cost created new international demand for Japanese products."[5]

ADP is an attempt to introduce Deming's methodology to SDLC in a following basic steps:

1. Identify a defect.
2. Find the root cause of the defect.
3. Locate the point in the production line that caused the defect.
4. Implement preventative practices to ensure that similar defects do not reoccur.
5. Monitor the process to verify effectiveness of the preventive practices.

So it's a way to encourage software developers and management to start preventing errors instead of simply detecting them.

Chapter 3. Principles of ADP

Automated Defect Prevention is based on six principles. They are designed to give guidelines, flexible enough to easily fit almost any development environment without much need to make revolutionary changes to already existent procedures. The principles go as follows:
**Principle 1**: Establishment of Infrastructure

*“Build a strong foundation through integration of people and technology”*

The first principle stresses the need for the establishment of an infrastructure. People, technology, and the interactions of the people with this technology, form such an infrastructure required to support a project. Thus, while all principles are equally important to ADP’s successful implementation, without the first principle, the others could not exist.

**Principle 2**: Application of General Best Practices

*“Learn from others’ mistakes”*

The premise of the second principle is the necessity of applying well accepted industry best practices, such as those defined for requirements management, design, or coding, in the lifecycle. Without understanding and embracing these practices, a company could not operate efficiently.

**Principle 3**: Customization of Best Practices

*“Learn from your own mistakes”*

Next, customization of industry practices is necessary. Customized best practices are organization and project specific. They are created based on action plans after the root cause analysis of defects uncovered during development, to prevent those defects from recurring. Thus, defect prevention is not possible without applying this principle.

**Principle 4**: Measurement and Tracking of Project Status

*“Understand the past and present to make decisions about the future”*

The need for measurements and tracking of project status is crucial to understanding whether the practices are being properly applied, to identifying problem areas requiring corrections, and to making critical decisions regarding project status and release readiness of the product. These factors form the basis for the fourth principle.

**Principle 5**: Automation

*“Let the computer do it”*

The necessity of automation in modern software development hardly needs justification. Not only does automation facilitate implementation of the best practices and collection of project-related data, it also reduces the error-prone human influence on process implementation. All repetitive and mundane tasks should be automated whenever possible in any portion of the software lifecycle.

**Principle 6**: Incremental Implementation of ADP’s Practices and Policies

Finally, incremental implementation of ADP is critical to its success. The gradual approach of introducing it group-by-group and practice-by-practice is essential to achieve a desired organizational culture change, as change is unsettling and it always encounters some degree of resistance. Both the complex nature of software projects themselves and the novelty of ADP warrant this systematic approach.

While the establishment of the infrastructure is a starting point and thus a foundation of the methodology, both automation and incremental implementation are the overarching principles that should be applied throughout ADP’s lifecycle. [¹,²]
Chapter 4. What ADP can give to a software company?

Many people within the industry think, that defect prevention is useless in software business. They believe in defects detection instead. The rapid evolution in software means, that no stable production line can be assembled, which – in many people's minds – is an inevitable prerequisite to implement Deming's principles. In fact, producing software does not require a traditional rigid production line of machines making machines, but rather a sophisticated and flexible infrastructure, capable of both adapting to minor changes within each development cycle, and adjusting to the major overhauls that occur from one release of software to another, or to a new product. Thus, ADP defines such an infrastructure as one of its six fundamental principles, application of which results in defect prevention becoming an intrinsic part of software development.1

First principle of ADP states, that one needs to establish infrastructure to integrate people and the technology involved. This can be done by enhancing people's traditional roles of project managers, developers, testers and QAs to include defect prevention duties. Experience has taught, that best practices must be directly embedded into the software development groups that will use them. This means, that each team member in the development group needs to understand his/her role — be it developer, architect, or project manager — and that he/she needs to understand how to adhere to that role. Also their roles should include using software to help them performing their tasks and automate them. As a minimum, a development group must have a functioning source control system, automated build system, problem tracking system, and an automated reporting system. This technology can be expanded as defect prevention matures in an organization, as the needs of a project change, or as additional software tools and people are added. In a model situation, each developer, tester, and SQA team member has a local installation of supporting technology; the technology configuration is determined by the team architect and standardized across the team. Gradually, the infrastructure should be expanded to include more technology and support other groups. An automated requirements management system is a good start for a requirements group. A repository of project artifacts including a database of best practices and policies, for team members' easy access, should be created. An automated testing system would facilitate the work of both developers and testers.

Second and third principle support and complement each other. They teach, that general best practices should be implemented in a company, and – furthermore – should be expanded by specific best practices and static error detection, based on a specifics of the organization. The set of general best practices is the product of software industry experts examining the most common errors and then developing rules designed to prevent these common errors. They represent a wealth of knowledge that was created through years of experience of many organizations. By adopting and, where possible, automating these "out-of-the-box" general practices, an organization can instantly progress from following the few best practices that it introduced over the years to a comprehensive set of standards that have been developed, tested, and perfected by industry experts. When teams are working on any type of project, it is always prudent to follow the practices and standards that industry experts have designed for the relevant application, language, platform, and technologies. The customized best practices address the project-specific problems. Some practices are very valuable in one type of development context, but not applicable to others. Customized practices are necessary because many modern development projects have vastly different needs, depending on the nuances of the product. Thus, there is a need for a mechanism that provides for customization of the best practices in order to prevent these unique defects. This principle is based on Deming's error prevention concept. It recommends that each time a defect is discovered, a new customized practice is defined to prevent re-occurrence of the defect in the future. Once the new custom practice or rule is defined, it has to become an integral part of the methodology and its application should be, if possible, both automated and seamless. The adherence to the custom practices should be monitored, so consequently,
the development methodology would become increasingly defect-resistant.

To make informed decisions, management must be able to analyze measures reflecting project status information. These measures are quantitative representations of product or process attributes. Thus, a fourth principle of ADP tells about measurement and tracking of project status. Measures could be assigned specific, absolute values, such as the number of defects uncovered, or they could characterize the degree to which a system, or its component, possesses a given attribute (also called metrics). These measures could also denote more general statistical indicators, such as confidence factors, derived from many basic measures. Indicators, for example, could provide information such as whether the project progresses according to its schedule and whether the costs are within the planned budget. At a more detailed level, many other project-essential statistics should be available; for example, the number of implemented requirements features, the number of failed tests, coverage of tests, or the number of defects and their severity, etc. Project indicators help in prompt identification of problems, so they can be remedied in a timely manner. Additionally, when observed over an extended period, those indicators can be used to assess product quality and its deployment readiness.

The all-important principle of ADP is, of course, automation: an activity that can be performed completely without human intervention. With ever growing complexity of software systems and many different cost-driven constraints, automation becomes a necessity for the delivery of quality software systems. Thus, Walker Royce in his top 10 principles of modern software management[^6] lists automation as a critical feature that facilitates or even encourages perpetual change in iterative processes. Although many managers hesitate to invest in auxiliary technology due to the high cost of licenses and training, the long-term benefits of automation largely outweigh these drawbacks. Moreover, repetitive, monotonous, and tedious tasks tend to have a negative impact on employee morale. They cause frustration and boredom and as such lead to ineffectiveness of people. Boredom often causes carelessness, which in turn leads to mistakes. Automation of repetitive and mundane tasks helps to overcome this ineffectiveness by providing an environment where team members can focus on more challenging and creative tasks. Thus, the role of automation is not to function as a substitute for people, but to improve the working conditions for people. Additionally, since people are excellent at creating new processes, but not very good at maintaining them, automation helps in sustaining existing process and consequently in embracing change, by providing an environment, where people can focus on activities that people do best: being imaginative and innovative.

Last, nonetheless very important, principle behind ADP is, that – in order to minimize the difficulties induced by yet another change and consequently encounter a resistance – ADP has to be introduced gradually to an organization: group-by-group and practice-by-practice. A pilot group, consisting of talented and motivated people should be selected. The pilot group should be actively working on the code and it should be dedicated to fully implementing the methodology. To gain the momentum needed to drive an organization-wide implementation of defect prevention, the pilot group must be a well-esteemed team working on an important project. Electing one of the organization's most talented teams as the pilot group yields two main benefits. First, it demonstrates the value placed in the methodology, and consequently helps the organization's most critical projects. Second, it would likely produce a showcase implementation, which provides other groups with real-life evidence that defect prevention can indeed be implemented in the organization's environment and that it actually delivers the promised benefits.[^1]

Chapter 5. How can embedded systems developers benefit on automation?

Testing the software on embedded systems is an issue of its own, as diagnostic
capabilities of such systems are usually limited. In most cases the quality of the testing process, thus the quality of an entire product, strongly relates to capabilities and qualifications of the people involved. Such situation reflects in many disadvantages when comparing to native development, starting from low repeatability of tests results, high impact made by team members changes, finishing on high costs of the process. Good example of automation of such process, by means of ADP, is usage of static analysis tools. Modern state-of-the-art automatic static and unit testing tools can be easily used to test complex technical matters, thus lowering the cost of quality assurance in a company. They also make the process more reliable by eliminating human factor out of the equation. Good example of such pitfalls, waiting for an embedded developers, can be bitwise operations, types conversion or arithmetic calculations: problems often hard to find by a developer (especially inexperienced one), yet easily identified by a dedicated tool (such as static analysis tool).

Experience shows, that most of the automated means of quality assurance should be used together to achieve best results, as they very often complement each other. And so:

— Static analysis: can help to get rid of malicious code as well as can ensure code compliance to standards (such as MISRA or JSF standards). Data flow analysis can be found especially helpful, as it can help you to find run-time errors (division by zero, null pointer usage, or improperly handled resources) early in the development process, without necessity to build the entire application.

— Unit testing: dramatically improves the quality of a code by ensuring its compliance with the specifications. It has also been noted, that consequence in using and maintaining unit tests lowers the costs of maintenance of already shipped product. This is especially important in embedded environments, where costs of error found after the release cannot even be compared to the same bug on native platforms. Coverage analysis is also very important part of unit testing, as it can give you the idea on how well the tests are written and what can be improved to achieve better (more reliable) results. Many certifications also require metrics of unit tests’ code coverage. Currently, there is a number of tools on the market (like Parasoft C++test™), that allow testing directly on the target device, thus being independent from the emulation platforms.

— Manual code reviews: in many cases the only way to find errors in developer’s code is for the code to be reviewed by another developer. Algorithmic flaws can be often found only using this methodology. Moreover, by enforcing manual peer reviews, what a company can gain is a set of developers enhancing each other’s knowledge and skills, plus a better knowledge of code base.

By enforcing these methodologies and monitoring their trends (changes over time) you can quickly and reliably get the information on the status of the project, thus allowing for quick countermeasures in case of any threat to the product. By a simple analogy to the theory of automation: by monitoring trends one has the ability to act like an exemplary PID controller, while by just looking at point-in-time results one would act like a simple P controller. The consequences: one can gain a stable and controlled quality assurance process leading to a stable and on-time product delivery, instead of a series of isolated initiatives targeted towards the purpose of reaching higher quality, that can often lead you nowhere...

Chapter 6. References