Antipatterns: A Compendium of Bad Practices in Software Development Processes

Sandro Javier Bolaños Castro[1], Rubén González Crespo[2], Víctor Hugo Medina García[1]
[1]*District University "Francisco José de Caldas" - Bogotá (Colombia)*
[2]*Pontificial University of Salamanca - Madrid (Spain)*

Abstract — **This Article presents a set of software process antipatterns, which arise as a result of bad practices within application development processes. Process AntiPatterns warn us about the harmful effects that may arise in projects, and also describe the features that identify them. The proposed anti-patterns provide a catalog that serves as a vocabulary for communication among project participants. Such Antipatterns can be implemented through software tools in order to keep better record of their implementation. Additionally, a tool that can operate under GPL (General Public license) is provided for this purpose.**

***Keywords* — Software Process, anti-patterns.**

I. Introduction

The goal of software development is to generate products with high levels of productivity and efficiency that ensure good levels of quality. To achieve this, it is necessary to avoid the risks introduced by bad practices of software. These bad practices have been labeled as anti-patterns, and occur in different areas. The catalog of antipatterns is an important road map, particularly on dark paths that might be followed when precautions are not taken, and of course, that cause problems in projects. This article presents a list of harmful practices that represent anti-patterns in the implementation of software development processes.

II. Understanding Antipatterns

Alongside patterns, the anti-patterns trend is also a major subject of study that should be taken into account. The anti-patterns that constitute harmful practices must be avoided to reduce the risk of failure in software projects. One of the most recognized works on anti-patterns is that proposed by Brown [1], where software development AntiPatterns, software architecture anti-patterns and antipatterns of software project management are put forward. Another work is that of Dikel [2], where a set of anti-patterns for software architectures according to vision, rhythm, anticipation, partnering and simplification is proposed. Unfortunately, it is very easy to be engaged in anti-patterns since they are caused by poor abstraction and poor implementation of the theoretical approaches of software. Usually, "shortcuts" and poor analysis approaches lead to malpractice. The time factor developers always have to compete against does not allow thinking more carefully about good practices; even patterns themselves might become anti-patterns when abusing their implementation. A definition of anti-patterns allows creating a recognizable vocaary that facilitates communication among the participants in software projects regarding dangerous situations you need to be aware of and avoid, or at least reduce their possible impact.

III. Antipatterns of the Software Development Process

The application of a process in a software project is necessary to monitor and control it. There is a wide range of software processes of different kinds; each process holds out a way to track and coordinate activities, resources and knowledge in order to provide a support tool for the operation of a software project. However, it is easy to incur in poor implementation of procedures and protocols as well as poor resource management, especially human resources, resulting in bad practices that might be called anti-patterns of software processes. Below we propose a set of harmful practices that may occur in software development processes.

A. *Top Process*

It is common that whenever a process is needed, the first choice is to pick the in-fashion process, which is generally proposed by a large organization, a community, a research center or a person or group of people who pool their expertise to propose a rescuing formula. Generally the top process is proposed as the only silver bullet [3] with regard to the process. However, what worked for a particular project environment does not necessarily work for every project environment. You must take into account the business conditions of the organization, and ultimately you must be careful about the inherent difference that exists between the application domains of the different processes. The main responsibility for achieving success lies in the process, as an essential tool for software projects, regardless of the software singularity [4], so the top process does not guarantee success (Figure 1).

DOI: 10.9781/ijimai.2011.147

Figure 1. Top Process

B. *Super Process*

Explaining any phenomenon from all angles is an approach that can be adopted. Similarly, using complexity to explain a software process is another way; in the words of Morin: "let us take a contemporary cloth, it uses flax, silk, cotton, and wool of various colors. For that cloth, it would be interesting to know the laws and principles concerning each of these types of fibers. However, the sum of knowledge about each of these types of fibers that form the cloth is insufficient to meet not only the new reality which is the tissue, that is, the qualities and specific properties of the texture, but also to help us understand the shape and configuration"[5]. As systems (or objects of study) become more complex, that is, not just consist of more parts but also the interaction between them becomes increasingly complex, it seems that the explanation of the phenomena presented by the behavior of such systems tends to take into account the "context", the environment, that is, the phenomena’s "totality"[6]. The complexity theory focuses on identifying that we already have enough to work on only by making the activities of a process harmonize, the proposed schemes end up in incomprehensible schemes, which include size, spirals, tables and other notations, often overloaded unnecessarily, becoming a burden that a development team cannot bare (Figure 2).

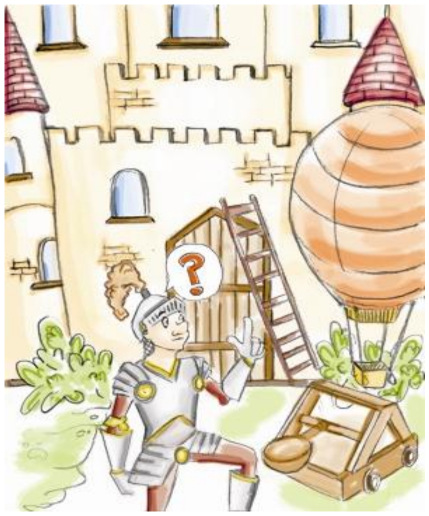


Figure 2. Super Process

C. *Extreme Process*

“There are profound differences between theory and mere computer technological rules" [7], for Popper, it is clear that there are two extremes: on the one hand, the theoretical approach, and on the other hand, instrumentalism. It seems that software processes fluctuate between these frequencies; unfortunately for any project, it is inconvenient to fall in these limits. On the one hand, theorizing about the issue of processes is a task not only valuable but also necessary, but the task itself must take into account that the processes should be practical, and it is at this point where the development steers into the other edge, namely instrumentalism. It is common for a software process to be successful in one project and fail in another, so relative success is not universal guarantor for a process, in this sense, pure instrumentalism runs out of arguments. Finding the right amount of theory, mostly as a result of the a-priori approach of reflection, along with a dose of instrumentalism can be a good combination. In this sense, developers should not rely entirely on a theory without proof, nor shoud they rely just on a test (probably successful but without epistemological foundation) when bearing in mind that processes follow a technical application that does not neglect the theoretical reflection on their problematic core (Figure 3).

Figure 3. Extreme Process

D. Casual Process

Conducting a software process often becomes an ad hoc activity, resulting in improvisation of the tasks. Such type of work takes place when an organization is not aware of the importance of processes and usually ends up diverting all the workload onto development activities. Ad hoc processes arise primarily because there is not a process manager who guides the selection of at least one process to perform. Ad hoc processes are not aware of the roles and end up creating handyman roles, promoting anti-patterns that generally resemble the project management anti-patterns [1]. An ad hoc process ends up extending schedules, repeating efforts and consuming resources. Because the process is not clearly identified, it may end up taking different names from a list given by the participants, which is usually inconsistent. A casual process tends to be confused with an organization's customized process, therefore, care must be taken when the course of the process has features like those listed above. (Figure 4)

Figure 4. Casual Process

E. Slide Process

Adopting a process that encourages the production of outputs from a given input regardless of the way in which workflow occurs is generally counterproductive. Software processes should not be slides, which do not pay interest to the way results are obtained, since in the workflow participate a society of roles that may be sacrificing not only the quality in the process, but most importantly, sacrificing performance conditions and quality of life. In a slide process, it is typical to start at a certain speed and finish with acceleration. In the same way, a process without rhythm [2] starts with extended times in its initial phases and have tight schedules in development and deployment phases. A slide process does not control time, delaying projects; it also accelerates at critical stages, sacrificing product quality. These processes end up adjusting schedules, paying fines, conducting renegotiations, and making considerable losses for the organization (Figure 5).

Figure 5. Slide Process

F. Immutable Process

Thinking that an immutable process represents a great advantage is a problem if you consider Heraclitus paraphrased words regarding his theory of perpetual flow "do not use the same process twice." Proposals such as CMM [8], about the repeated process as one of the levels of maturity, point at a feature that is apparently advantageous; however, such a setting is unfeasible given that the conditions and specific process variables are impossible to repeat; even when in the extreme case where conditions are very simillar, time becomes an impediment. A process, as opposed to be considered immutable, should be treated with high doses of adaptation, as proposed by methodologies like ASD [9]. Each time a software process is conducted, it truly becomes a new process. The fact that a process has a general guide should not be confused with executing the same process over and over again. Considering a process as immutable eliminates the possibility of seeking new knowledge when developing the process, losing the possibility of improving the process (Figure 6).

Figure 6. Immutable Process

G. Process without evidence

Usually, software development processes involve creating documents related to the product being made, such as developing manuals and user manuals, among others. However, a document of the process itself, which at least provides information about what was learnt from the process execution, is a task that is never performed. When the process lacks evidence of its execution, it is highly probable that the same actions will be re-executed with the same fundamental flaws. These side effects result in process delays, repeating and perpetuating defects. Processes without evidence, are a sign that there is no process manager, who leads the process and records its past history for new process implementations (Figure 7).

Figure 7. Process without evidence

H. Process without rhythm

A software development process should try to keep a sort of rhythm in each of its activities so that there are no gaps that hinder efforts and resource investment from efficiently contributing to constituent-workflow tasks. It is common to find elongated-time activities, while other activities are time-constraint, the proportions of time allocation must be fair without causing trauma. The time resource should be one of the main variables to govern the processes, the workflow must balance the periods of time employed in each activity, thus avoiding botched executions. A process without rhythm occurs when other antipatterns are inserted, such as paralysis of analysis or design by committee [1]. In these harmful practices, it is evident that the imbalance in a specific activity impedes the normal execution of the remaining activities (Figure 8).

Figure 8. Process without rhythm

I. Domino Process

A development process tempered by a high interdependence between the activities that constitute its workflow will result in a domino process.Initial activities are critical and cause exponential effects on final activities to the point that it becomes impossible to produce an activity *i +1* if you have not fully completed activity *i*. A domino process leads to stiffness and reduces the possibility of feedback at early stages in the workflow. A problem is detected when the cost has increased considerably, leading to elongation in the schedules, as well as to inefficient use of resources. Unfortunately, when developing software, it is very common to find problems in the requirements phase, given the volatility and ambiguity typical of gathering requirements; under these conditions, if a process does not propose strategies to deal with the activities themselves as well as with the activity-coupling management, a domino process will evolve easily (Figure 9).

Figure 9. Domino Process

J. Perpetual Process

When a process becomes interminable is said to be a *perpetual process*. Generally speakig, the process falls into infinite loops when the workflow is repeated without generating useful products. This type of process is evidence of the immaturity associated to the organization that runs the process as well as of its lack of adequate estimation, its failure to meet the requirements and development. Such immaturity is most obvious when in the testing phase, where developers will need to constantly repair things, with the aggravating circumstance that these repairs might cause further inconveniences. In the perpetuity of the process there is no proper configuration management, and quality control is summarized in trying to fix an accumulation of defects that cause poor reliability [10] of the results obtained at a prticular point of development. When a process becomes perpetual, it usually ends abruptly with negative collateral implications for the participants (Figure 10).

Figure 10. Perpetual Process

K. Headless process

Poorly managed processes, and /or processes with leadership problems in the various disciplines, are referred to as headless processes. This type of process does not define clear functional objectives and responsibilities, there is a poor identification and assessment of the roles and therefore there is no adequate assessment of the disciplines; activities usually focus on the production of code without ensuring appropriate quality conditions; moreover, ad-hoc delegations occur. Headless processes exhibit exaggerated rotation of staff, stalling the workflow and leading to an abrupt end with unfavorable implications for the parties involved (Figure 11).

Figure 11. Headless Process

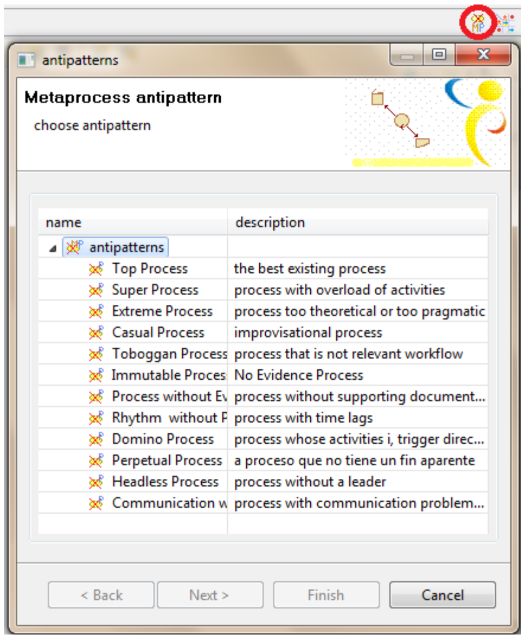
L. Processes without Communication

Communication between the parts of a process is critical to ensure the flow of information and of the knowledge management processes [11]. For a software process it is important to create role networks to integrate the different functions and responsibilities. The lack of communication makes processes slow, consequently, work flow stagnates and redundancy of labor is produced; moreover, resources wear out and delivery times are easily exceeded. Communication must flow in the organization in every possible way, not only from the command roles to subordinates, but also from basic to higher roles. Some agile methodologies, such as daily meetings, propose good practice regarding communication, where project-roles interaction strengthens the processes. This results in the generation of evidence and promotes continuous improvement. Lack of communication promotes the loss of resources and also slows work flow down (Figure 12).

Figure 12. Process without Communication

IV. ANTIPATTERNS SUPPORT THROUGH SOFTWARE PROCESS

One of the advantages of having a catalog of antipatterns for software processes is to implement the catalog using automated tools, which allows timely identification of a bad practice within a process. Th purpose is to generate a labelled-fault control record that helps developers avoid following wrong paths whenever running a process in a software project. In this particular case, we have developed a process antipatterns component for the Coloso platform [10], (Figure 13).



Metaprocess antipattern

choose antipattern

name	description
antipatterns	
Top Process	the best existing process
Super Process	process with overload of activities
Extreme Process	process too theoretical or too pragmatic
Casual Process	improvisational process
Toboggan Process	process that is not relevant workflow
Immutable Proces	No Evidence Process
Process without Ev	process without supporting document...
Rhythm without F	process with time lags
Domino Process	process whose activities i, trigger direc...
Perpetual Process	a proceso que no tiene un fin aparente
Headless Process	process without a leader
Communication w	process with communication problem...

Figure 13. Coloso Software, www.colosoft.com.co

V. CONCLUSION

The anti-patterns generate a vocabulary and a list of risks that can arise when using a software process. This vocabulary facilitates effective communication between the different roles of the process and contributes to failure detection and quick response whenever risks are encountered in a project.

Software processes have been accepted as heroic formulas but processes themselves are part of the problem of software development and although such processes need not be permanently reinvented, it is extremely necessary to see their weaknesses and strengths in order to avoid trauma when conducting projects.

ACKNOWLEDGMENT

Special thanks to designer and art student Paula Milena Sanchez, who made the drawings in this article.

REFERENCES

[1] Brown, W., Malveau, R., Hays, M., & Mowbray, T. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis* . John Wiley & Sons. 1998.

[2] Dikel, D., Kane, D., & Wilson, J. *Software Architecture.* Prentice Hall. 2001.

[3] Brooks, F. *The Mythical Man - Month.* Addison Wesley.1995.

[4] Bolaños, S., Medina, V., & Joyanes, L. Principios para la Formalización de la Ingeniería de Software. *Ingenieria*, 31-37.2009.

[5] Morin, E. *Introducción al Pensamiento Complejo.* Gedisa. 2001.

[6] Johansen, O. *Introducción a la Teoria General de Sistemas.* Limusa. 2001.

[7] Popper, K. *Realismo y el Objetivo de la Ciencia.* Tecnos. 1998

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

[9] Highsmith III, J., & Orr, K. *Adaptive Software Development : A Collaborative Approach to Managing Complex Systems* . Nueva York, EUA: Dorset House.2000.

[10] Meyer, B. *Construcción de Software Orientado a Objetos.* Prentice Hall. 1999.

[11] Nonaka, I. A Dynamic Theory of Organizational Knowledge Creation. *Aroganization Science*, 14-37.1994.