Software Development Activity Cycles

Collaborative Development, Continuous Testing and User Acceptance

Robert F. Rose

Apress®

Software Development Activity Cycles

Collaborative Development, Continuous Testing and User Acceptance

Robert F. Rose

Apress®

Software Development Activity Cycles: Collaborative Development, Continuous Testing and User Acceptance

Robert F. Rose
Alexandria, VA, USA

ISBN-13 (pbk): 978-1-4842-8238-0 ISBN-13 (electronic): 978-1-4842-8239-7
https://doi.org/10.1007/978-1-4842-8239-7

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Shiva Ramachandran
Development Editor: James Markham
Coordinating Editor: Jessica Vakili

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, New York, NY 10004. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Printed on acid-free paper

Books, books! There are dozens of books, hundreds of books, on every subject imaginable. So please, if you must write a book, make it a small one.
–Anonymous Professor of History

If you can't describe what you are doing as a process, you don't know what you're doing.
–W. Edwards Deming

Table of Contents

About the Author

Robert F. Rose has provided services to both private and public sectors including telecom and healthcare, NavAir, the Environmental Protection Agency (EPA), and Housing and Urban Development (HUD). His experience includes pioneering design and development of a warehouse system for storing and analyzing medical records, design and development of an early prototype logistics tracking system for the V22 Osprey, and design and implementation of a complex enterprise-wide web-based directory system. Among his accomplishments, he was Technical Project Manager for the Presidential Commission's Inquiry on the Challenger Disaster. The DPAC model is the product of independent efforts both in management and in preparation of the technical approach section for various responses to requests for proposals (RFP). Now retired, Robert has pulled together the sum of his experience with the process of developing software into the DPAC framework. It is entirely original work and not derived from other approaches. He can be contacted at Robert.F.Rose.77@gmail.com

Author's Preface

I should say from the outset, as of this writing, the DPAC model has not been used in practice. This is a concept paper, applied to a hypothetical example. That being said, I contend that the "Cycles of Activity" represented in the DPAC paradigm are representative of human activity in every information system development effort. DPAC makes a clear separation between human activity and the progress of software throughout the process.

While the model is entirely original, my interpretation of the cycles has been informed by the long list of works contained in the Resources by Category and Alpha Listings. I have included books only. There are only a few articles cited in the book, most of them in Chapter 2. This belies the very large number of articles that were reviewed to find the best of the best, most relevant to the current discussion. Too many, in my opinion, to be listed for productive assistance.

Regarding citations of materials found online, rather than using a web address, which changes for a specific item even as we speak (so to speak), I have cited "Google" which means Google the title to find the item at its current location. Not only do web addresses change, but in the current period companies are being gobbled up by other companies such that entire .com addresses are changing. Every effort has been made to contact the source of materials used herein – seeking permission for use. Some have been unresponsive quite possibly due to the fog of a merger. I apologize if I have given offense. Attribution has been provided nonetheless.

In this interpretation, DPAC is a fully "shift left" model, taking current trends to their "logical and absurd" conclusion. If nothing else, it should have heuristic value, although I believe it has merit in and of itself.

I started building my technical library in 2012 and copyrighted the DPAC model with the present Chapter 1 in 2015. My earliest versions of the model stretch back to the mid-1990s. In the intervening years, my library was built up following industry trends – an expensive endeavor at best. I think we are now (2022) at a plateau where "stovepipe" thinking can be reduced or eliminated. I hope DPAC will help turn our thinking about software development around the corner.

I would like to thank the folks at Apress – Jessica, Jim, and Shiva – for stewarding the manuscript through the travails of creating a book and for their patience with my occasional outbursts of panic as I leapt into the unknown. And special thanks to my support group for their kindness and encouragement over the years my "writing project" took to come to fruition. Carolyn, Diane, Doran, Jim, Jolene, Pam, and Richard – you helped more than you know.

Introduction

Intent and Purpose

The first computer I ever saw was playing Jingle Bells. It was 1958, on the ground floor of the DuPont engineering building near Wilmington, Delaware. The Univac computer was built around vacuum tubes and solenoids that buzzed and hummed while computing. A group of engineers, clearly making use of any free time, wrote a nonsense program that matched the noises to Christmas carols. I was mesmerized.

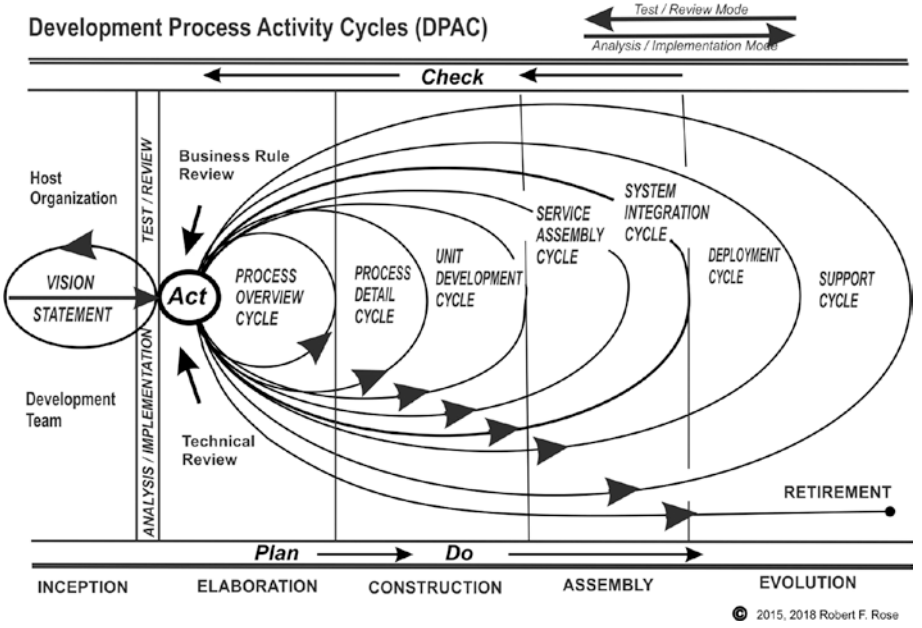
My career in IT did not begin until a decade later in 1969 as a FORTRAN programmer. I started building *systems* in 1976. My first system was a small Management Information System (MIS) that was concluded with great success. My second effort was a system an order of magnitude more complex. After one flop at the start, it was completed but never deployed – giving me a strong sense of project risk.

I started thinking about the *process* of developing software in the 1980s to respond to the technical approach section of various RFPs (requests for proposals). The DPAC model has evolved from that process through my direct participation over 35 years in six software development projects. There was no direct influence from other models. DPAC reflects my thinking since circa 1995; the DPAC model in its present form was copyrighted in 2015.

Having also participated as a technical troubleshooter cleaning up systems after the developers "left the building," I experienced firsthand that many issues in support are directly related to the quality of the software development effort.

The model described herein represents a continuum beginning with inception into ongoing evolution. While retirement seems inevitable, there are systems that are still going strong through decades of use. Legacy systems may need to be *enhanced* to enjoy the delights of user-facing web and mobile experiences.

Another thing the DPAC model is designed to address is the derivation of functional requirements throughout the development process beginning with a Vision Statement and continuing into the Support Cycle. To that end, the cycles of activity in the DPAC model are represented as recursive and re-entrant, nested contiguous ovals. Each activity cycle is overlaid with an interpretation of the Deming quality control phases of Plan, Do, Check, and Act (PDCA).



The DPAC Model

The DPAC Model and Chapters at a Glance

A model, as used in this context, is a graphic representation of the development processes. A tour through the DPAC model begins in Chapter 1. The larger case for development to keep software support in mind is presented in Chapter 2. Chapter 3 defines the Inception Stage, while Chapters 4–7 are walk-throughs of the Elaboration, Construction, Assembly, and Evolution Stages, respectively. Each of these chapters begins with the model diagram at the top highlighting the part of the model the chapter describes and where it fits in the overall process.

Chapter 8 presents a summary of personal experiences with risk. Chapter 9 is a short exposition on Engineering Software Quality. Chapter 10 contains Final Remarks.

Until now there has been no generalized model of the course of activities over *all* the application life stages. *The DPAC model brazenly claims to fill that gap*. In fact, I contend that it is the optimum model of geometric efficiency. It also describes a singular path of development, including recursive flows.

> *DPAC does for agile what Waterfall does for traditional methods of development.*

The Tower of Babel

> *5 But the LORD came down to see the city and the tower that the men were building. 6 The LORD said, "If as one people speaking the same language they have begun to do this, then nothing they plan to do will be impossible for them. 7 Come, let us go down and confuse their language so they will not understand each other."*
>
> Gen 11:5-7
>
> New International Version

Vocabulary informs our view of reality.

As used herein, the term "User Acceptance-Driven Development (UADD)" is a superset of specification by example, behavior-driven development (BDD), and acceptance test-driven development (ATDD). Each of these methods has one thing in common. They are all dependent upon agreement by the user that the subject of analysis is covered. They all can use the triad approach of developer, tester, and client to achieve those objectives (collaborative development). UADD contends that it is *user feedback* that drives the *development effort*. Test-driven development (TDD) and refactoring are techniques used to reduce error and produce clean code.

Where, please, will the miracle occur?

The odds are low that the needs of the user will be satisfied by a single meeting – no matter how long it may take, it is more likely than not that the process of Plan, Do, Test, and Review will require more than one pass. This is to say that the "little gray cells" need time to marinate on the problem. The problem of meeting the needs of the user continues through the Unit Development Cycle, Service Assembly, System Integration, and User Acceptance Test (UAT) just before the system is deployed.

The lack of a consistent terminology creates, to some degree, a waste of energy.

In writing this book, one of the things I want to accomplish is to use a consistent vocabulary for describing a cyclical, iterative process. The DPAC model begins with the concept of Stages. Within each Stage are one or more cycles of activity. A Cycle is composed of phases, and a phase may have more than one part. "Step" can be used colloquially for any succeeding element.

QE, QA, CM, and Test

There are a number of issues regarding vocabulary rooted deeply into the thinking and practice of development. The first is to disambiguate Software Quality Assurance (SQA) from test. Software Quality Assurance in the domain of Quality Engineering (QE) is the maintenance of a desired level of quality in a service or product, especially by means of attention to every stage of the process of delivery and production. In the QE paradigm, "Test" is an element of quality control (see breakout, SWEBOK v. 3, and relevant ISO standards).

Software Quality Assurance (SQA) officers play a crucial role. The most important of the Process QA tasks is to assure the integrity of the traceability matrix by ensuring that work products are visible, traceable, and accountable – including audits – and that standards are met.

Use of the Term "Quality Assurance" in the Private and Public Sectors

All this is muddied by the difference in the use of the term Quality Assurance (QA) between the private and public sectors. Many federal agencies (e.g., Department of Defense (DoD), Small Business Administration (SBA), NASA) as well as guidance from the General Services Administration (GSA), Office of Management and Budget (OPM), and Federal Acquisition Requirements (FAR) require a separate SQA function.

Similarly, state (e.g., NV, CA, IA) and local governments may be obliged to use the term QA for monitoring of the entire development process. Private sector entities, on the other hand, which tend to be focused on the Support Activity Cycle, may have a "QA Department" (meaning test) and no dedicated Quality Engineering function.

Herein after, I will use Software Quality Assurance (SQA) to represent Process Quality Control and will avoid the use of the term QA as a synonym for test.

Configuration Management

Another clarification in thinking is to disabuse the notion that configuration management (CM) is only about code version control. CM applies to control of each class of work products from the evolution of requirements, tracing test scripts, a traceability matrix, to maintenance of "lookup" tables such as a personnel roster or task list. It is governed by the "rule of immutability" – whereby each version of each work product is regarded as unique and is stored accordingly. In most efforts, however, *code* version control is under the supervision of the technical team.

Use of the Term "Service"

I have borrowed the term "service" to indicate a complete and independent component of code. "Service" implies function, while the term "module" is used from the perspective of the programmer. A *service* may comprise one or more units of code where a unit is the smallest stand-alone system component. For example, a web-based ecommerce site might include (at least) four services: (1) catalog, (2) cart management, (3) checkout, and (4) customer.

(The term "service" in this regard from Max Martynov and Kirill Evstigneev, *Continuous Delivery Blueprint*, Grid Dynamics, np 2017–2018)

Units, Components, Services, and Systems

A unit is the smallest piece of stand-alone code. A component comprises one or more units of code as a stand-alone generally with a GUI (graphical user interface). One or more components comprise a service, and one or more services comprise a system. ***All changes take place at the unit level of code.***

Systems of Record vs. Systems of Engagement

There is a significant difference between a System of Record (SOR) such as a Management Information System (MIS) and a System of Engagement (SOE) such as a mobile phone application. A SOR is developed from the inside out, where requirements are unknown or only sparsely defined up front, resulting in a set of user interfaces (or screens). A SOE is developed (or should be developed) from the outside in, where the user interface is described up front and the SOR to support it is defined therefrom. DPAC as applied to the example used herein is for a System of Record. A System of Engagement may be served by a data structure supplementing the System of Record, or, for security purposes, may be isolated as a separate system with available data updated from the SOR accessible to the SOE on a periodic (even hourly) basis. Keep SOR data for the SOE separate from the LAN and intranet. The mobile app will want to retrieve current billing information. The SOE would have the means for the user to pay the current bill.

Quality Assurance, Quality Control, and Test. What's the Difference?

June 23, 2016 – Blog Post – Functionize.com (Google)

Organizations throw around terms quite a bit and sometimes interchangeably, even if they aren't really synonymous. A prime example of this is with such terms like Quality Assurance (QA), Quality Control (QC) and Testing. Though they're closely related, they are, ultimately, different.

If you work in the IT industry, you've probably come across them. You've also probably noticed that many executives – and customers – don't understand the difference between these terms. They most likely go as far as referring to them as the same processes, which they're definitely not. Let's figure out the difference.

***Quality assurance** is process oriented. It is all about preventing defects by ensuring the processes used to manage and create deliverables works. Not only does it work, but is consistently followed by the team. Moreover, QA is about engineering processes that assure quality is achieved in an effective and efficient way.*

For instance, if a defect is found and fixed, there is no guaranteeing it won't pop back up. The role of QA is to identify the process that allowed the error to occur and re-engineer the system so that these defects won't appear for the second time. The QA process verifies that the product will continue to function as the customer expects.

Though QC is absolutely necessary, QA is perhaps more important. By the time you reach the QC stage, for instance, fixing bugs becomes an expensive issue. Because of that, focusing efforts on improved QA processes is one of the best investments an organization can make.

Examples of QA include process definition and implementation, training, audits and selection of tools.

***Quality control**, alternatively, is product oriented. It is the function of software quality that determines the ending result is what was expected. Whereas QA is proactive, QC is reactive. QC detects bugs by inspecting and testing the product. This involves checking the product against a predetermined set of requirements and validating that the product meets those requirements.*

Examples of QC include technical reviews, software testing and code inspections.

***Testing** is a subset of QC. It is the process of executing a system in order to detect bugs in the product so that they get fixed. Testing is an integral part of QC as it helps demonstrate that the product runs the way it is expected and designed for.*

To summarize, think of everything as an assembly line. QA can be thought of as the process to ensure the assembly line actually works, while QC is when the products coming off the assembly line are checked to verify they meet the required specifications.

Ultimately, both QA and QC are required for ensuring a successful product. When used together, they can help detect inefficient processes and identify bugs in the product. Moreover, QA and QC can help to develop and deliver a consistently high-quality product to your customers.

Evolution vs. "Production"

The term "production" is another carryover from the manufacture of "things" to "software" development efforts. It belies the true nature of software development as a process of evolving requirements to meet a set of generally defined objectives. "Turnover to Production" or "Transition to Production" largely set the view that software is delivered in a complete and final form. As anyone with experience in "Maintenance" is aware, this could not be farther from the truth – if only to complete system parts that were left unfinished by the development team. DPAC labels this "Stage 5. Evolution."

Support vs. "Maintenance"

One performs *maintenance* on a car to restore it to its original condition. One *supports* the *evolution* of software. "Maintenance" is yet another carryover from the manufacture of things.

The DPAC model recognizes software as an organic medium providing a conduit for the processing and transformation of information. The Evolution Stage is the same as the initial development stages except in slow motion as the system continues to evolve. Requirements morph into additional requirements and/or changes or additions to basic business rules. Again as used herein, the Support Activity Cycle includes software support (including evolutionary changes), operations (providing support and provisioning for hardware, LAN, configuration, and access to the system), and Database Administration (DBA).

Scrum is defined by Scrum.org as "a *framework* within which people can address complex adaptive problems, while productively and creatively delivering products of the highest possible value." DPAC is an alternative "framework," although Chapter 5 may be used to explain what goes on inside a "Sprint."

The "Application Life Cycle"

I have a problem with the term "application life cycle," with *phases,* instead of a linear model with "*stages*." It's not really like we are going to start all over again on the same ground. *Lay the progress line flat.* It's about *one* system, not many. If the "application life cycle" does mean anything, it means that the first one flopped so hard that another contractor was called in to finish the job. And one after that, and one after that… ad nauseam until somebody declares the project "Dead" and pulls the fiduciary plug. So much for the *Cycled Life* of the Application.

Interestingly enough, the activities that emerged from each Activity Cycle mostly "fell out" of the model as I have described them herein (Chapters 3–7, the stages respectively). It may well be that the "tasks" to which the *model* must respond define the most efficient course of the activities of the developers of the system.

The Software Development Industry

DPAC proposes a different world view of software development activities from that of the "Software Development Industry." The System Development Life Cycle (SDLC), a competing model, faces the same demise as the "application life cycle," that is, *you do it over, and over, and over…" etc.*

The SDLC model has, however, spawned a legion of serial "software developers." That is the essence of build/fail, build/fail, until either you get it right or the Contracting Officer advises the Budget Director, and funds are pulled from the project. Then as someone said on a failed federal contract – "You don't think the Government is going to take the blame for this, do you?" And another contractor bites the dust.

The DPAC Model

I have picked up bits and pieces from a long list of "resources," a set of techniques from literature on the subject, and influences from the various agile methods and methodologies. But it was ultimately the DPAC model that has emerged from decades thinking about the process of developing software that changed my own "world view" of the development process. It is different in three ways:

(1) The moderation of words and references emanating from the "production of things"

(2) The recognition that systems *evolve,* and software support services need to support that activity

> (3) The DPAC model itself – which I posit as the most efficient geometric representation, including recursion, through the "software development" maze

DPAC is a conceptual framework on which one can "hang the ornaments," that is, the techniques, methods, and procedures appropriate to each stage or activity cycle of the model.

As interpreted herein, DPAC depends on inspections, exercised in the Act Phase of the PDCA quality model, that cogent internal documentation – in stream comments – exists. (Trust but verify.) The external documentation comprises adherence to standards, guidelines, and procedures which have the secondary importance of having a new person come up to speed faster than without them.

All the rest is "tribal knowledge," which can verbally tie up two people in conversation to bring a new person up to speed. Put the standards and procedures up on Wiki accessible to all. Demand that procedures known only to developers by tribal knowledge be written down where it is accessible to all. These are responses to risk mitigation, especially the loss of a key person. See my experience with this matter in Chapter 9. "How do I file a Change Report (CRpt)?" without a written procedure is a question that could tie up two or more persons' time. The "standard" need be no longer than a few sentences: "Hand it to the Configuration Manager who will log it in and turn it over to the Project Manager for evaluation. Expect a response within two days."

To be sure, the DPAC model is informed by my participation as the Software Quality Assurance (SQA) officer, on a 50-55 person development effort. Scrum thrives on a considerably smaller scale. But DPAC can also work in smaller schemes such as several agile teams of about nine persons each assigned to a specific service, but each following the methodology suggested by the DPAC model. (See definition of a service above.) The model can be iterated to accommodate larger projects.

Developers vs. Support Personnel

Finally, software support personnel make poor developers. Development is raising something from scratch and needs several specialized skills – an architect, programmers, business analysts, and testers. Support, on the other hand, builds "add-ons" like building a sun porch to an existing structure. Something *new* to support might be the addition of a Configuration Management Database (CMDB) to support a help desk, or a new service: "inventory." And developers make terrible support personnel. Support is not the thrill of making something out of nothing other than of the intent of the users.

I believe that the DPAC model, taken as a framework for developing software, resolves the dissonances that cost time figuring out which way to go. The interpretation of DPAC presented herein takes a Lean-Agile Collaborative Development approach to these issues; test is built into each activity cycle as part of the Check Phase of PDCA as is Review with User. So that's the broad description, but

> ***From such crooked wood as that which man is made of, nothing straight can be fashioned.***
>
> **—Immanuel Kant**

All that being said:
Onward through the fog…