RPA Coding Standards and Practices

[Summary 1](#_Toc530754582)

[Documentation 2](#_Toc530754583)

[Understand the limitations 2](#_Toc530754584)

[Planning 3](#_Toc530754585)

[Modularity 3](#_Toc530754586)

[Avoid hard coding 4](#_Toc530754587)

[Keep It Simple 4](#_Toc530754588)

[Readability 4](#_Toc530754589)

[Naming conventions 5](#_Toc530754590)

[Code review 5](#_Toc530754591)

[Testing 5](#_Toc530754592)

[Comments 6](#_Toc530754593)

[Decoupling and Loose Coupling 6](#_Toc530754594)

[Error Handling 7](#_Toc530754595)

[Security 7](#_Toc530754596)

# Summary

Coding standard and best practices are a set of informal rules that the software development community has learned over time which can help improve the quality of code. Following a consistent coding standard helps improve the quality of the overall Bot design. The more readable source code is, the easier it is for someone to maintain that code. By following a consistent style, it allows other developers to step in and help with maintenance or new development.

The goals of a good coding standard are: Improved code maintainability, code quality. In particular: testability, supportability, portability, security, efficiency, robustness, loose coupling

# Documentation

Begin with the end to end in mind which will help to execute Bot lifecycle.

In the context of RPA, this means you should start off by documenting the process that you will be automating.

Let’s call this the Process Definition Document (PDD), and it minimally comprises of the process map, as well as a detailed step-by-step walkthrough of the process. These process steps are usually augmented by screenshots for clarity purposes, and may include pertinent information such as possible exceptions and error handling.

Other useful information to include inside the PDD are:

* Contacts of the process SMEs
* “As is” process metrics, e.g. transaction volumes, average handling time (AHT), etc
* In scope and out of scope

One hidden advantage of doing this is that the PDD can serve as part of your organization’s Business Continuity Plan (BCP). For example, if there is an outage in the production bots, your SMEs should be able to pick up the PDD and follow the instructions there to perform the process manually (as a last resort, of course).

# Understand the limitations

No matter which RPA platforms you use, there is no one perfect tool.

There are always limitations to what all these commercially available RPA software can do, despite what your vendor may tell you. Vendor can provide capacity or limitation cheat sheet which can be reference for your Bot design.

Some of these include:

* Do not be overly reliant on the desktop recorder. The desktop recorder can be a very useful tool for you to understand how to construct your RPA script for a certain process, but rarely is the generated script resilient enough to be used reliably in a production environment.
* The truth, unfortunately, is that the robots, not matter how carefully designed and developed, is unlikely to work all the time. Hence, in RPA, it is extremely important for the robots to be able to handle errors and exceptions. Logging should be done so that further investigation, if warranted, can be carried out.
* Image recognition should be used only as a last resort when all else fails. This is because image matching only works when the image is fully visible on screen, i.e. it cannot be in the background. Also, image recognition is sensitive to the screen resolution, which can be sometimes tricky to handle.
* Most RPA tools allow you to extract text from desktop or web applications, either natively or using an OCR engine. If you are using the later, do be mindful that OCR is not hundred per cent accurate. In cases where accuracy is important, you might not want to rely solely on OCR.

# Planning

If you fail to plan, you plan to fail. The same goes for RPA development as well.

It is generally not advisable for one to jump straight into RPA development without first considering the high level design. What this means is that you need to identify beforehand the various components that altogether make up your solution, and how these components interact with one another.

One great hack – create a generic and process agnostic framework that enables you to deal with any use cases in a consistent and structured manner.

# Modularity

Implied from the planning point is the importance of modularity.

Modularity in this context means simply to break down a complex script/program into bite-sized (more manageable) components.

There are two important benefits of doing so.

Firstly, there are often common steps that occur across different processes, for example, logging into the SAP system. Rather than generating these steps from scratch every time you automate a new process, a smarter and more efficient way is to create a separate workflow that does so, and add this to your automation library.

So the next time you have a process that requires logging into SAP, you can simply invoke this workflow file as follows:

This greatly enhances the reusability of your content.

The second benefit is that modularity supports fine granular development and testing. This is especially useful when dealing with complex processes, as you can independently assign the development and testing of specific components to individual developers.

# Avoid hard coding

The idea is to minimize unnecessary changes to the script, especially if it has already been deployed to production, to avoid introducing additional bugs or errors. Also, if these settings are used multiple times within the script, if you hard code, you need to know exactly where the changes are required.

Where possible, you should avoid hard coding all the external settings (e.g. file paths, URLs) within your script. By hard coding, it means that the only way to change these settings is to amend them from within the script itself.

Hard coding should be avoided because these settings are **prone to changes over time**. For example, the file location where you download the daily report to might change due to business requirements.

The most efficient way to store these parameters is to use a configuration file.

# Keep It Simple

In programming/scripting, simple is beautiful (it is not always easy).

Where possible, try to reduce or eliminate unnecessary complexities in your script. This will help reduce the number of bugs or programming errors, which in turn means that you need to spend less time trying to debug or resolve them.

# Readability

Ignoring the readability of your RPA script is a common mistake made by many new RPA developers.

Readability in this case means how easy is it for someone else (other than the developer) to take the script and understand what the software robot is being programmed to do. In fact, for complex processes involving hundreds or thousands of steps, the developer himself often lose track of what is going on if he does not take active steps to ensure readability. Ensuring good readability will also drastically reduced the amount of time you spend during troubleshooting and debugging, as well as during bots maintenance.

To do so is fairly straightforward. Mostly, you need to make a conscientious effort to provide meaningful names for workflow files, activities, arguments and variables. Moreover, as a best practice, all your variables and arguments should be aligned to a naming convention:

* Variables should be in camelCase, e.g. firstName, lastName
* Arguments should be in TitleCase, e.g. FileName, DefaultTimeOut

Also, all your workflow names should contain the verb describing what the workflow does, e.g. GetTransactionData, LoginToSAP.

Last one, you should make judicious use of comments and annotations to describe sections of your code which are not as intuitive. It is important to be selective though as you do not want to end up clustering your program with redundant comments.

# Naming conventions

Use of proper naming conventions is considered good practice. Sometimes programmers tend to use X1, Y1, etc. as variables and forget to replace them with meaningful ones, causing confusion.

In order to prevent this waste of time, it is usually considered good practice to use descriptive names in the code since it’s about real data.

Example: A variable for taking in weight as a parameter for a truck can be named “TrkWeight” or “TruckWeightKilograms”, with “TruckWeightKilograms” being the more preferable one, since it is instantly recognizable.

# Code review

These are very helpful when it comes to refactoring your code. Code review is a phase in the Bot development process in which the authors of code, peer reviewers, and perhaps quality assurance (QA) testers get together to review Bot script.

Others might see better implementation to optimize your code or just make your code more elegant. It also ensures that developers adhere to standards and the work is double checked. In addition to all that, it is a wonderful way for developers to learn from each other.

The role of independent code reviewers should go beyond the compliance aspects (detection of errors and malicious code): they need to ensure that your Bot development is reusable. Code review is the key to ensure that you have highly maintainable and expandable bots

# Testing

One common mistake made by new developers is not testing their code enough (or at all). This is especially true in the case of RPA where developers normally start off working on the happy path.

Unfortunately, in reality, the scenarios are usually much more complicated than that. There could be many variants of a single process, or the input data might not come in the format that you expect. Hence it is important to put in place a robust test plan with comprehensive test cases, so that we can ensure the robot is performing exactly the way we want it to be.

A failed test should not lead to finger pointing. Rather, it should be embraced in the sense that the error is discovered before the bots go into production (where far greater damage can be caused).

Also, you should strive to test as frequently as possible. Do not wait until the entire development is completed before you start testing. If you have implemented modularity in your design, you should be able to independently test the various components and see if they are working as intended.

For the best results, RPA development should not be an individual pursuit. In an ideal case, the developer, the peer reviewer and the tester should all be different persons.

# Comments

Due to time restrictions or enthusiastic programmers who want immediate results for their code, commenting of code often takes a back seat. Good code should be understandable without a line of comments.

But what to do to save time for new developers? — Write simple inline documentation describing what and how Bot script works. This will save much time for understanding and even more — it will give people more chances to come up with better implementation of particular Bot script. And also it will be good start for global code documentation and helps where more than one person may work on a particular Bot script.

Following are the common comments Bot developer should incorporate in the comments are.

* Description of the Bot
* Bot Author & Date
* Bot Modified & Date
* Parameters

Decoupling and Loose Coupling

When possible, sub-tasks should not have a dependency on the calling task. In automation this is often unavoidable. However, developing with this in mind can improve the overall architecture and maintainability of automation.

* Using the login sub-task as an example, if the login task can only be called by one single master task, then it is tightly coupled to that master task. If the login sub-task is designed in such a way that the URL of the page it uses has to be set by the calling task, then it cannot run by itself. It cannot be unit-tested alone, and other tasks cannot call it without knowing the URL of the login page before calling it. If the calling task must provide the login page URL to the login sub-task, then all tasks that use the login sub-task are more tightly coupled to that login sub-task. And if the URL changes, more than one task must be changed.
* However, if the login sub-task contains all of the information it needs to login to the web application, including the URL, then it is a truly stand-alone sub-task. It can be unit-tested, and it can be called by any other task without the need to be provided the URL. It is then “decoupled” from other tasks and is much more maintainable.

# Error Handling

The implementation of exception handling in programming languages typically involves a fair amount of support from both a code generator and the runtime system accompanying a compiler.

The idea is to provide a more rigorous basis for exception handling by defining precisely what “normal and abnormal” behavior is. Specifically, the approach is based on two concepts:

* Failure: the inability of an operation to fulfill its contract. For example, an addition may produce an arithmetic overflow (it does not fulfill its contract of computing a good approximation to the mathematical sum); or a routine may fail to meet its post condition.
* Exception: an abnormal event occurring during the execution of a routine (that routine is the "recipient" of the exception) during its execution. Such an abnormal event results from the failure of an operation called by the routine.

Error Handling enable handling unwanted situation graceful and helps to avoid break the code/system.

# Security

Secure coding is the practice of developing computer [software](https://en.wikipedia.org/wiki/Software) in a way that guards against the accidental introduction of security vulnerabilities.

Identify and document security requirements early in the development life cycle and make sure that subsequent development artifacts are evaluated for compliance with those requirements.

Develop and/or apply a secure coding standard for your target development language and platform for e.g. masking password during inputs, pass sensitive data as encryption, refer credential vault if it is exist.