TechRxiv
Powered by IEEE

DNAP: Dynamic Nuchwezi Architecture Platform - A New Software Extension and Construction Technology

This paper was downloaded from TechRxiv (https://www.techrxiv.org).

LICENSE

CC BY 4.0

SUBMISSION DATE / POSTED DATE

02-11-2020 / 03-11-2020

CITATION

Lutalo, Joseph Willrich; Steven Eyobu, Odongo; Kanagwa, Benjamin (2020): DNAP: Dynamic Nuchwezi Architecture Platform - A New Software Extension and Construction Technology. TechRxiv. Preprint. https://doi.org/10.36227/techrxiv.13176365.v1

DOI

10.36227/techrxiv.13176365.v1

THIS WORK HAS BEEN SUBMITTED TO THE IEEE FOR POSSIBLE PUBLICATION. COPYRIGHT MAY BE TRANSFERRED WITHOUT NOTICE, AFTER WHICH THIS VERSION MAY NO LONGER BE ACCESSIBLE.

DNAP: Dynamic Nuchwezi Architecture Platform - A New Software Extension and Construction Technology

Joseph Willrich Lutalo, Odongo Steven Eyobu and Kanagwa Benjamin

Abstract—The need to improve or build new software systems to solve new and old business challenges is a persistent challenge in the software consumer and development industry, yet costly. To minimize these costs, the construction method should be designed with the following qualities in mind; software portability, extensibility, and simplicity. To achieve these qualities, this paper proposes the Dynamic Nuchwezi Architecture Platform (DNAP), which is a new software construction and extension technology. DNAP offers a visual programming paradigm with a capability of generating production-ready business automation software for both mobile and web. It also offers a simple mechanism for the extension of existing softwares using embeddable components. To evaluate and justify DNAP, eight Software Operating Environment (SOE) metrics have been developed and together with the SOE model, are used to contrast DNAP against four alternative software construction technologies namely; Android Platform, .NET Framework, Java SE Platform and Python. The performance evaluation results show that DNAP offers an average of 33% reduction in software construction complexity and an 11% enhancement in language efficiency when compared to alternative technologies.

Index Terms—Software Construction Optimization, Visual Programming, Cross-Platform Development, Platform Engineering, Mini-Programs, Software Metrics

1 Introduction

TODAY, the smartphone is the most ubiquitous computing electronic device with a high level of use among consumers worldwide. This trend has seen the development of many mobile-oriented information technology solutions with the aim of enabling or enhancing business automation. Currently, the most popular mobile software environment in use globally is the Android Operating System, or simply "Android Platform". However, robust software products should be designed to support several operating systems and platforms since consumers have the freedom of choice and there exist several competing platforms to choose from. Such cross-platform software engineering is not widely adopted though, and neither is it easy to achieve even though it is much needed. Designing software construction methods that offer software portability as a core feature becomes very important then; it offers the ability for software to be used over a wide array of hardware and software configurations. The processes of software extension and adaptation to suit new environments or new business needs is a rigorous process. In most cases, it requires a full reiteration of the software development life cycle, so as to realize the necessary new requirements, design updates and their implementation. Typically, this entire process is costly and time consuming. In a nutshell, traditional software evolution ofttimes requires the intervention of expert software designers and developers ready to study and work through existing software codes and designs, and then tailor them to fit new software and hardware environment specifications.

For both business owners and their technical teams, a simpler approach that doesn't sacrifice productivity and reliability would be the most desirable. Such an approach should at least achieve software portability, extensibility and process simplification, while delivering the desired end product or software solution. By portability, we mean that the developed software should be readily usable on different software and hardware environments. By extensibility, the software solution should be readily adaptable to new ends. By process simplification, the software construction method should not require software domain experts to realize a software product, improve on its functionality, make changes to its interfaces or deploy it for use. These three key software construction method qualities are what this study sets out to address.

It is traditional and quite popular that software construction methods employ the use of high level textual programming languages. However, this approach can readily become challenging for non-experts. There exist a few approaches devised by the programming community to improve on this software development methodology. Among these is the use of visual programming interfaces that eliminate or lessen the need to explicitly write code, so as to simplify software development. Some notable solutions include Pocket Code [1] which leverages a block-based programming paradigm similar to MIT Scratch [2] and Berkeley's SNAP [3], aimed at producing non-trivial mobile applications directly on phone. Note that all three projects are not suitable for the construction of production-ready business automation software in the wider sense, and only Pocket Code can output mobile-targeting software. Another

- J.W Lutalo, O.S Eyobu and K Benjamin are with the Department of Networks, School of Computing and Informatics Technology, Makerere University, Kampala, Uganda.
E-mail: {j.lutalo, sodongo, bkanagwa}@cis.mak.ac.ug
- Correspondence : sodongo@cis.mak.ac.ug

Manuscript received ... .., 2020; revised ..... , ...

notable advance in this area is Block Pictogramming [4], which champions the use of pictograms to specify functionality and generate software. As with many other block-based programming approaches today though, it is still most useful as an educational or prototyping method, and not as a real-world software development solution usable in the day to day operations of a business.

Some existing software construction approaches do not attempt to eliminate the need to explicitly write source code during development but aim at simplifying the process as much as possible. This is especially true for web software such as content management systems and data collection tools. Notable technologies in this category include platforms such as WordPress [5] for website and content management software, and Google Forms [6] for data collection tools. The former simplifies the software construction process by offering low-configuration, plug-and-play extensions, while the later allows a configuration-free, visual development approach for data collection tools. It is worth noting though, that despite simplifying development, both technologies aren't suitable for the development of standalone mobile or web apps usable in offline scenarios or independently of the authoring platform.

Works on software extensibility have come up to aid the evolution process of software. Projects such as Spinel [7] have been proposed to attempt to make the construction of readily extensible mobile applications easy. The approach makes use of a special plugin-architecture that allows an existing data-driven app to offer new functionality without re-write. The method however currently limits such extensibility to only data browsing applications on Android, and any new functionality required would involve development of new plugins - which the non-expert might not afford.

On the side of software extensibility for web applications, the simplest approach in use today is the employment of embeddable components via iframes [8]. This approach is successfully used to add media playback functionality to existing web applications using such platforms as SoundCloud, YouTube or MixCloud. These make it possible to edit or preview documents – such as is possible using services like Scribd or CodePen, which offer embeddable widgets leveraging the standard HTML iframe tag. However, it is a known issue in the security community that this approach can sometimes be abused so that problems such as cross-frame scripting or differential context vulnerabilities become possible. In such cases, malicious code is run in the context of the hosting application via the imported components in the iframe [9]. Therefore, this approach to software extension though practical for web applications, needs to be hardened where security is of critical concern.

The issue of cross-platform development has been traditionally addressed using different approaches for mobile, web and desktop software. In the most common scenario, a cross-platform technology is designed to address portability across only one of these environments – that is, portability across several different native mobile platforms, or across several desktop environments or only portability across different web/browser environments. It is only a few technologies that have attempted to support portability across both mobile and web or web and desktop or all the three.

One of the most ambitious cross-platform technologies known today is Java. Java has been around since the closing years of the 20th century. It offers a widely adopted solution to cross-platform development that leverages the idea of compiling software to a virtual machine as opposed to directly targeting native platforms. The standard virtual machine for Java software is the Java Virtual Machine (JVM), which takes portable byte code and executes it as native instructions on the specific platform where the JVM is running [10]. The portability of software gained by using Java almost comes as a given since the language was designed to be portable from the start. However, it does not always address the needs of every software construction or business automation task, and therefore, exploring other cross-platform solutions is justified.

The language C# offers a compelling alternative to cross-platform development away from Java, and one of the very popular software development technologies leveraging this language is Xamarin [11]. These two languages, together with JavaScript offer a mature and generic platform-agnostic solution to cross-platform development for those with software engineering expertise, however, to the amateur and hobbyist software engineer, software portability leveraging these approaches remains difficult.

In this paper, we propose a new approach to software extension and development referred to as the Dynamic Nuchwezi Architecture Platform (DNAP). DNAP is aimed at addressing all the three highlighted major challenges in modern software engineering; software portability, extensibility and the simplification of the software construction process. Essentially, this paper makes the following key contributions:

1) Introduces a simple visual programming paradigm called the persona pattern, for the design and construction of production-ready business automation software for the web and mobile via a special integrated development environment called the DNAP studio.
2) Introduces a new mini-program specification language called Cwa Script, for the specification of rich client applications using a syntax that is both lightweight and readily parsable.
3) Introduces a simple mechanism for the extension of existing web, mobile and desktop software using embeddable DNAP components such as the DNAP histrion and diviner.
4) Introduces a mechanism for large-scale discovery and distribution of both software and data over the web using a publish-subscribe pattern leveraging DNAP channels.
5) Introduces the Software Operating Environment (SOE) model for the quantitative comparison of software construction technologies, and uses this to contrast DNAP against existing alternative technologies.
6) In the analysis of software construction technologies, it is found that in general, interpreted languages have a simpler software construction process than compiled languages, and this idea is expressed mathematically in section 4.4.

The rest of this paper is organized as follows; Section II

highlights what other projects there are like DNAP in the literature, Section III goes into detail about the structure, defining properties and potential applications of DNAP, Section IV exhibits what data there is about the relative performance of DNAP against the competition. Finally, we conclude with what further work there is to be done in this direction, in section V.

2 RELATED WORKS

In the following three subsections, the three major software development challenges of programming complexity, software extensibility and portability in modern software engineering, are explored at length.

2.1 Programming Complexity: Visual Programming Interfaces and Languages Simplifying Software Construction

When it comes to software construction using non-textual methods, two dominant approaches prevail: the use of visual programming interfaces (VPI) and then visual programming languages (VPL). VPL involves approaches to software construction in which the specification of a program relies on the manipulation of graphical artifacts [12]. VPI on the other hand does not eschew textual specification of a program entirely, but offers some graphical interface that allows the textual code to be generated automatically via manipulation of graphical artifacts.

Examples of VPL include RAPTOR, a flow chart based VPL, Alice and Scratch [12]. Note that VPL approaches to software construction involve not just graphical interfaces or environments for producing the program, but that the program itself might be thought of as an executable graphical artifact – what traditionally have been called "executable graphics" [13].

Considering VPL approaches, the most outstanding in the industry today are block programming languages. These are special and different from the traditional programming which employs numeric opcodes and parameters in bare-metal coding. Traditional programming came up with higher level languages that abstract away the low-level instruction set via human-readable alpha-numeric textual syntax. Another approach is block programming which makes coding feel less like a technical undertaking and more like an artistic endeavor.

The most famous block programming technologies include Snap [3]. Snap originated from Berkeley, and was inspired by an earlier block programming environment called Scratch [2]. Both offer browser-based environments for coding and running applications – which run using standard web technologies such as JavaScript, HTML and CSS. However, they make it unnecessary to know about the underlying implementation details, and nor does one need to explicitly write any JavaScript or HTML while using these environments – except perhaps, for the expert engineer needing to develop new blocks or extend the environment itself [14].

It should be noted that the use of drag-and-drop interfaces in programs like Microsoft Visual Studio to design and specify the look and behavior of traditional Windows Forms (WF) applications or modern XAML powered Windows Presentation Foundation (WPF) applications, does not entirely eliminate the need to code business logic and the application's event handling instructions in text, away from the visual interface. In fact, it is possible to code an entire WF or WPF application in a pure text-editor such as Notepad or Vim while avoiding the visual paradigm entirely.

One of the reasons business automation, and software construction in general are not yet accessible to the masses, is because the tools and languages used to build software themselves create a high barrier to entry for the majority of prospective coders and programmers. VPI and VPL however simplify the process of translating ideas into workable software, by employing graphical metaphors and languages that feel more like how building and manipulating things in real life is done. Block programming for example, makes building software feel and look like child's play with traditional LEGO blocks or bricks. This means that more stakeholders and not just software engineers or formally trained programmers can engage in software construction with little or no training at all.

2.2 Plugin-Architectures and Software Extensibility Mechanisms

Building extensibility into most consumer software such as mobile applications – or rather "apps", is inspired by the need to offer end-users the means to customize their apps or apply them to new ends after installation. Also, because many apps perform functionality that is fundamentally similar - for example displaying data fetched from a remote API or prompting for data that is submitted to a remote destination, it is better and simpler to design "platform-apps" that users can install once, and then use to do many distinct, but related tasks without explicitly installing any extra standalone, often monolithic apps.

In exploring how extensibility has been realized in modern software, especially mobile software, we shall start with a project known as Spinel [7]. It is a plugin-architecture for Android that makes it easy to extend an existing application with new functionality powered by new data sources, without having to explicitly reprogram the core application. Spinel was designed to bring extensibility to applications that offer list-creation, list-viewing, and visualizing geo-tagged data – essentially, the display of data from an API over a map [7].

Spinel offers extensibility by offering a mechanism to hot-load new data-source configurations specified in JSON files. These files can be written by hand or are generated automatically via the online Spinel plugin generator tool. The user needs to manually download these files onto their device, and then use the Spinel plugin mechanism to load them into the app at runtime to start using the extra functionality. Spinel also allows developers to write new plugins via custom Java code, but this has been found to be the least used approach and can only be recommended for experienced developers. Finally, Spinel offers a library that developers can use to add such extensibility to their apps, though it is currently limited to only Android apps.

In more recent times, the justification for building platform-apps has gained attention. The media is talking

of the "death of mobile apps" in relation to a growing trend that has seen major vendors design their apps in such a way that users spend as much time inside a single app as is possible; the app provides access to services and functionality beyond what its core and developers provided, without requiring the user to exit the app – via what are being called "mini programs" [15].

The best example of a platform-app in the literature is WeChat, which is a social messaging platform with no less than 1 billion active users as of 2018 [16]. WeChat exemplifies the state-of-the-art in designing extensibility inside mobile apps, with the introduction of lightweight "apps within apps" that are better referred to as "mini-programs" [16]. WeChat's mini-programs, also called "applets" [8] are built independently of the main WeChat application, and are mostly developed by the community of users and businesses that leverage the social media application itself. An example of such mini-programs is explored in a paper about We2Book [17].

These mini-programs fit nicely into the category of software plugins because they extend the core application with new functionality without requiring the users to download or install a new version of the host application. Further, they are viewed and used in the context of the main application, and thus, their management, update and security can likewise be controlled via a single, main, host application.

Unlike Spinel, WeChat's plugin mechanism does not require users to manually download anything onto the target mobile device for the extra functionality to be used. Also, the feature scope of WeChat's mini-programs is much wider and more sophisticated with support for arbitrary asynchronous network and local on-device storage I/O via the Java powered MINA framework. WeChat's mini-programs are composed from a minimum of 3 files – app.json, app.js and app.wxss which specify the mini-program's meta-data and structure, logic, and style respectively [18].

The mini-programs approach is appealing because new functionality not related to the core purpose of an application can be added, on-demand, by the user, without involving the host-application's developers and without the need to install any new native software on the device. However, it is not just mobile applications that exhibit or need such extensibility mechanisms.

Among the most widely used software today worldwide, is the web browser. These programs, which are the default clients used by billions of users around the world to search, manipulate and share petabytes of information on the internet, are present on virtually all consumer computing platforms such as smartphones, tablet computers, laptops, desktops, smart televisions and more. The core functionality of a web browser is to accept a resource address, locate the resource, fetch it, and present it to the user for preview or download. However, over time, it has become necessary for browsers to offer a means to extend their functionality beyond just this base use-case [19].

The typical approach to adding extensibility in web browsers has been via the use of browser extensions and browser plugins [19]. Browser plugins are typically used to render content types that the web browser cannot natively process, or which are proprietary. Since the native content types handled by most browsers are HTML, CSS and JavaScript, browser plugins are used to render such custom content as Java applets or Macromedia Flash animations. Browser extensions on the other hand are plugins too, but are useful in a wider array of purposes including modifying the content displayed in a browser or altering the look and behavior of the browser itself.

The Firefox web browser offers one of the most powerful extensibility mechanisms documented, and it offers users and developers a means to write their own extensions to the browser via packages leveraging JavaScript and the Extensible User Interface Language (XUL) [19]. The power of its extension mechanism is hinged on the fact that the same technology used to render the core browser features and interface is also available to the extension environment, so that experienced users can turn the web browser into custom applications for purposes other than browsing the web. A good example of such power at work is in the Topaz extension used in turning the Firefox web browser into a client for the GridFTP protocol [20].

2.3 Cross-Platform Mechanisms spanning Web and Mobile

As mentioned in the introduction, the need for portability across operating systems or execution environments might vary from project to project, however, we can speak of portability across the web, mobile and desktop to keep things simple. Of course, there is also the case of embedded software such as with solutions in robotics, the Internet of Things or nano-technology. However, in this paper, we are content with cross-platform development mechanisms spanning mostly the web and native mobile environments.

Simplifying the construction of software, while catering for portability is an important factor in the choice of cross-platform development approaches. When considering portability with power and feature-completeness – or rather, the ability to write arbitrary solutions that are cross-platform, then leveraging general-purpose programming languages that offer portability as a major feature is key. In this category, we can consider such technologies as Java, C#, and JavaScript as the current leaders. Each comes with its merits and demerits; we shall briefly consider each, and then see why we need alternatives still.

Java, which is still proprietary, and which was originally developed by Sun Microsystems, offers one of the most widely documented, most widely deployed approaches to cross-platform development still in use today. Java programs gain their portability from the fact that compilation targets a special runtime – the JVM, and not the native, hardware-specific runtime upon which the application runs [10]. This means that writing code in Java guarantees portability to each and every environment where a compatible JVM has been implemented – it does not matter if it is web, mobile or desktop, and there are a multitude of such active and inactive virtual machines for various environments [21]. Typically, user-facing Java programs are packaged as Java archives (Jars) – which is a binary format for the language's assemblies, and can then be executed directly on the target platform where a Java Runtime Environment (JRE) and corresponding JVM exist [10]. On the web, Java programs have traditionally been executed inside of a special sandbox

via browser plugins, and with limited access to the native platform, in a format known as Java Applets.

The other popular technology in the same category as Java is the general-purpose language originally developed by Microsoft, known as C#. One major allure of C# is the vast software development tool-chain offered by Microsoft towards simplifying development in the language, and especially for desktop and web development [22]. This later point makes the language a favorite for both novices and professionals looking for a less painful approach to constructing basic and complex business automation software with compelling user interfaces. The simplification of development using drag-and-drop interfaces in Visual Studio, and the many utilities included in that integrated development environment (IDE) for the C# language greatly contributes to its success over Java and several other alternatives. However, as with Java, the C# technology stack relies on many proprietary components, and this is one of the reasons projects like Mono [23] were developed, to offer an open-source implementation of the .NET standard which specifies C#. Further, projects such as Xamarin [11], which is closely related to and leverages .NET and Mono are an ambitious attempt to increase the portability of C# beyond the proprietary Microsoft ecosystem. Xamarin offers a framework for using the same C# codebase to target Android, iOS, Windows, Linux, Mac, and several other platforms [24].

With the emergence of web browsers in the early nineties came the most ubiquitous approach to implementing platform-agnostic software – the World Wide Web, or simply the “web” [25]. Software running on the web gains its portability from the same principle making Java programs portable – basically, everywhere a web browser is implemented for a specific, native target environment, web software shall be able to run without modification. In this case, the web browser is to web languages, what the JVM is to Java. Of the technologies used to implement web software, the general-purpose language, JavaScript, is key. The other important web technologies include Hypertext Markup Language (HTML), which allows to specify the structure and layout of web applications, and Cascading Style Sheets (CSS), which offer a language for specifying the look and feel of web applications. JavaScript, despite being platform-agnostic, does not necessarily solve cross-platform development for the web on its own. This is especially due to the fact that differences exist across browsers, in the sense of; which web standards are supported, how they are implemented, and which extra features the browser supports for the construction of and execution of web software [19].

The relative ease of constructing web software and the ubiquitousness of web browser implementations across virtually all mobile and desktop environments supports the observation that JavaScript ranks high on the list of widely adopted programming languages in the industry [26]. This explains why many developers have chosen to approach cross-platform development by leveraging web technologies; building software in such a way that the same codebase is used to render software for desktop and web via especially the standalone web browser, and for many mobile scenarios, via embedded browsers – in what are commonly called “hybrid applications” [27]. Such flexibility is mostly realized by the use of cross-platform development frameworks, and the most notable ones include PhoneGap or Apache Cordova [28], the Ionic Framework and React Native [29].

In section III we describe how the newly proposed DNAP offers compelling advantages that overlap some of the merits of these explored technologies.

3 The Dynamic Nuchwezi Architecture Platform

First of all, if we had to classify the Dynamic Nuchwezi Architecture Platform (DNAP) from the perspective of engineering processes possible on the platform, then it shall be found that DNAP is a software construction, software distribution, data engineering, data distribution and data analysis technology.

The core DNAP suite consists of four major, decoupled, but closely related components referred to as; the Studio, the Theatre, the Histrion and the Diviner. The formal definition, structure, and purpose of each are described in the rest of this section.

3.1 The Architecture

Fig. 1 highlights the key components of DNAP from an architectural perspective, while Fig. 2 shows how each component functions using a mindmap diagram. Moreover, Fig. 2, shows the concepts which are present in all four core modules that make up the DNAP system – four because the mobile and web histrion are considered to be two different variations of the same core module – the DNAP histrion. Some of the more important concepts from the perspective of data engineering include the persona, acts and channels. We shall therefore explore each of them in turn, and see what they are, and how they relate to each other and the whole.

In the formal DNAP nomenclature, information generated from interactions with user-facing software interfaces, and that is then transferred to short or long-term storage, is what is called “data”. DNAP leverages two special kinds of structured data - the "persona" and the "act", also known as "personas" and "acts" in plural, respectively. The Persona is formally defined in the next section, while the Act is defined in section 3.1.4.

3.1.1 The Persona

Definition 1: Persona: *A data structure, holding the definition, constraints, and metadata of a mini-program on the Dynamic Nuchwezi Architecture Platform.*

The persona is the core concept in the DNAP ecosystem, and at the implementation level, it is expressed using a simple domain specific language called Cwa Script. Cwa Script derives its syntax from JSON [30] and further adopts some of the ideas from JSON Schema [31] — which is a standard for annotating and validating JSON data.

A typical persona, “persona script”, “persona source code” or “persona file” in Cwa Script, has the general structure shown in Listing 1. For demonstration purposes, in the rest of this paper, we shall use as reference, a persona designed to crowd-source local weather updates and

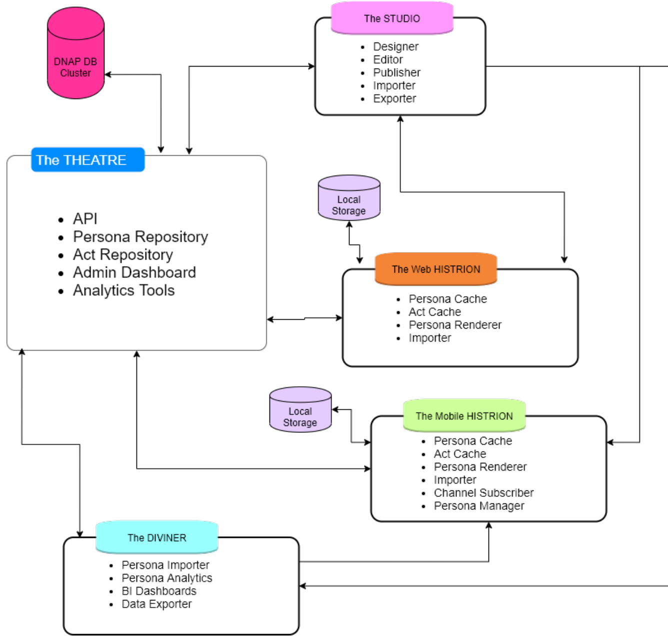

Fig. 1: Dynamic Nuchwezi Architecture Platform

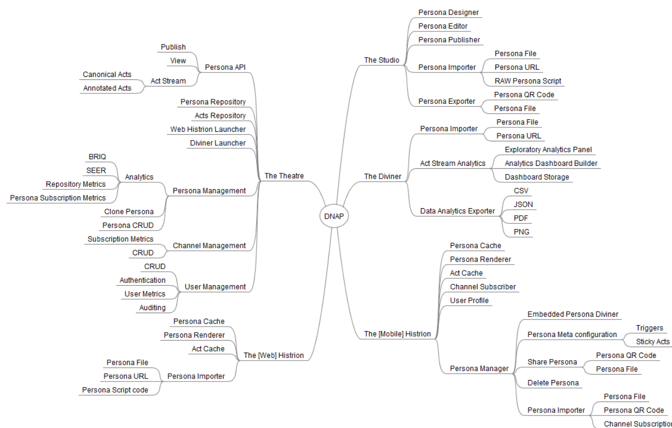

Fig. 2: DNAP Overview

Listing 1: A Persona Structure

```
{
        "app": {
                "name": "<APP_NAME>",
                "color": "<HEX_ENCODED_COLOR>",
                "theatre_address": "<URI>",
                "channel": "<PERSONA_CHANNEL>",
                "transport_mode": "<POST|GET|SMS|EMAIL>",
                "description": "<APP_DESCRIPTION>",
                "brand_image": "<APP_BRAND_IMAGE_URL>",
                "uuid": "<PERSONA_UUID>"
        },
        "fields": [{
                        "field_type": "<FIELD_TYPE>",
                        "cid": "<CID>",
                        "pattern": "<REGEX>",
                        "required": <BOOLEAN>,
                        "label": "<FIELD_LABEL>",
                        "field_options": {
                                "description": "<FIELD_DESCRIPTION>"
                        },
                        "meta": "<FIELD_META_PARAMETERS>"

                }, ...
        ]
}
```

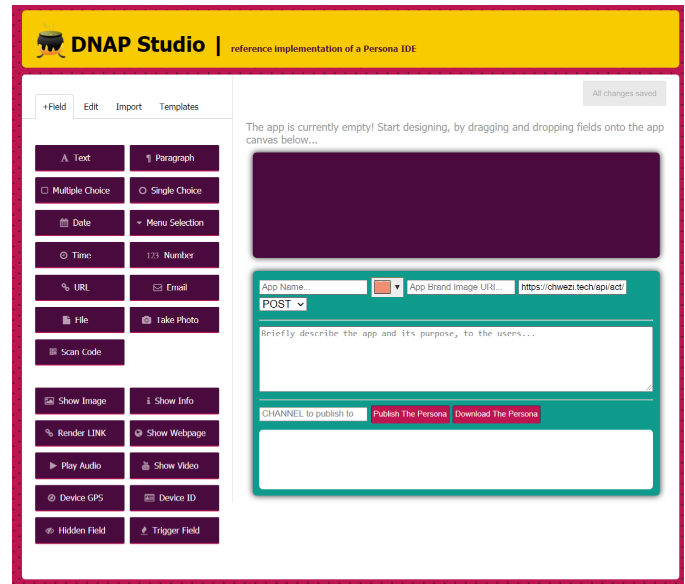

Fig. 3: The DNAP Studio — an online IDE for personas

especially occurrences of rain. This persona we have called “NJURA”, and its full definition can be studied in Appendix A

Personas are executable data - executable because when interpreted inside of a DNAP histrion, they render programs that are called "persona applets". The following section defines what a persona applet is, while the DNAP histrion is formally defined in section 3.1.3.

Definition 2: Persona Applet: *A mini-program running on a DNAP histrion, and whose behavior and looks are derived from its associated persona.*

3.1.2 The Studio and Channel

Definition 3: Studio: *An integrated development environment (IDE), for the design, editing and publication of personas.*

The DNAP reference implementation of a studio runs on the web and looks as shown in Fig. 3. We shall also need the following key concept:

Definition 4: Channel: *A namespace identified by a non-empty alpha-numeric string, to which personas can be published at design time, via the studio.*

3.1.3 The Histrion

Once a persona has been constructed — via the studio, or by hand, a means to turn it into an app that someone can then interact with is necessary. The “histrion”, which is available both as a mobile and web platform-app in the current reference implementation of DNAP, is what plays this role in the DNAP ecosystem – turning personas into persona applets. It is the “front-end” of DNAP, and precisely specified is:

Definition 4: Histrion: *A platform-app that can parse and interpret Cwa Script from which it then renders persona applets.*

The above definitions imply that the DNAP histrion is a parser, interpreter and runtime for Cwa Script. In a way, a histrion is akin to a web-browser, only that instead of HTML, CSS and JavaScript, the histrion processes Cwa

Script, and instead of web pages, it renders or outputs persona applets. Fig. 4 highlights the key components of a histrion from an architectural perspective, to make it clearer what a DNAP histrion is.

As its name implies, this module behaves more like an actor in a theatre; capable of assuming new roles and looks on-demand, based on the currently active script — in the DNAP case, the "script" being the active persona script. Looked at this way, it could be argued that most (if not all) existing web-browsers are a kind of histrion — in which case a persona script is analogous to a web document driving the browser, but that's where the similarity stops.

As of this writing, there exist two reference implementations of a histrion — one for the Android operating system, and another for the world wide web (www). Both can take a persona and output a compatible persona applet that a user can interact with. Currently though, the mobile implementation of the histrion surpasses the web version in terms of feature completeness.

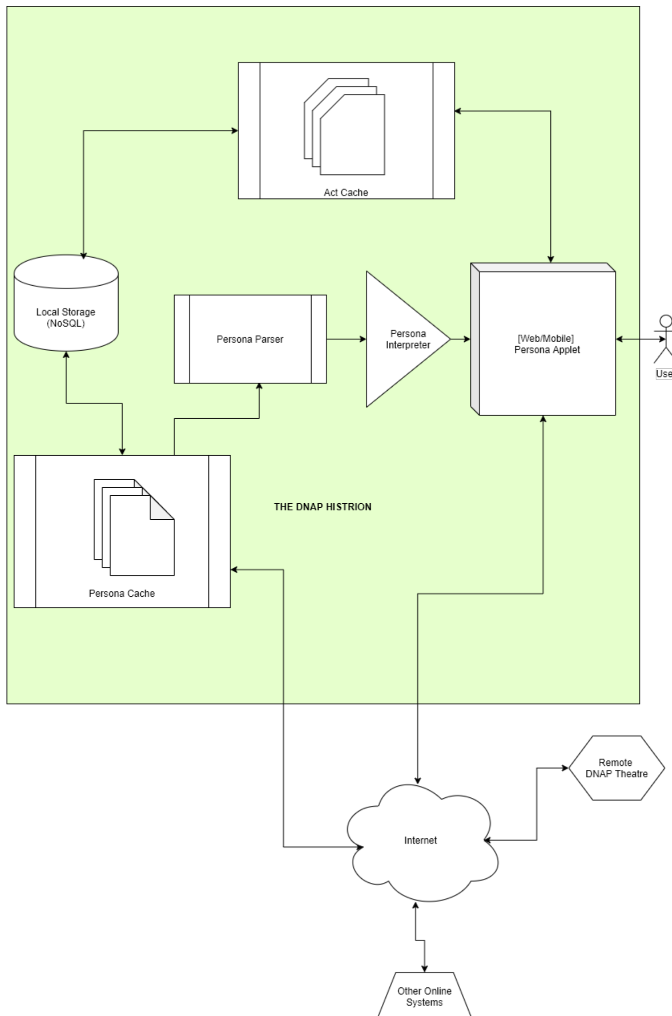

Fig. 4: The Histrion Architecture

The key highlights of a histrion include:

1) It is a standalone app, but whose purpose is to load and run other apps — the personas applets. This is what makes the histrion a platform app, and the persona applets are the mini-programs that run inside a histrion.
2) Personas published via a compatible theatre can be discovered and can be locally cached in a histrion, via user-configured subscriptions to channels.
3) It can load personas from a local source (persona file) or remote/online source (via persona QR Code or channel subscriptions).
4) It renders a persona as a native app depending on where the histrion is running — mobile or web for now.
5) Its persona cache allows a user to run or interact with a persona originally fetched online even if they go offline.
6) It can allow offline data collection — data records thus collected also known as "saved acts".
7) Via a linked, online diviner, it allows a user to browse or analyze submitted acts - including acts submitted from other histrions, on other devices across a network, for any loaded personas.

In addition to these core features, the current mobile implementation of the histrion allows for basic user identification and tagging — at installation time and or via the default histrion settings. The user specifies a name and contact, which information is automatically included in each and every act, so that later, together with the use of the "Device ID" field type on a loaded persona if present, it becomes possible to unambiguously distinguish and attribute data submissions from multiple devices to their respective owners/authors. This is meant to enhance traceability of origin or simplify auditing, where users, such as data enumerators are to be rewarded say based on their actual submissions. This mechanism allows for "non-repudiation", a very important factor in data and information security.

3.1.4 The Act

Once a persona is used to capture information via some set of pre-defined data fields on a persona applet, the resulting data record, which is encoded using JSON, is what is called an "act". More formally:

> **Definition 6: Act:** *A data record whose structure is a flat JSON dictionary with keys corresponding to fields on a persona, and the values holding data recorded against each field during a data collection session.*

Typically, acts are generated via persona applets, but it is also possible to compose an act by hand. Thus we know that given an act, we expect there to exist a persona specifying its data semantics. In general, an act has the structure as shown in Listing 2.

In that listing, the extra fields not labeled <FIELD_ID> are "meta fields", that help to classify or track certain aspects of the data collection session that produced the data held in the <FIELD_VALUE> entries.

For example, given a persona such as "NJURA" (refer to Appendix A), the sample in Listing 3 is a corresponding

Listing 2: Act Structure

```
{
        "<FIELD_ID>": "<FIELD_VALUE>",
        "<FIELD_ID>": "<FIELD_VALUE>",
        "<FIELD_ID>": "<FIELD_VALUE>",
        ...
        "OBSERVER": "<USER_NAME>",
        "OBSERVER_ID": "<USER_ID_OR_CONTACT>",
        "CACHE_TIMESTAMP": "<YYYY_MM_DDTHH:mmZ>",
        "UUID": " <PERSONA_UUID>"
}
```

valid act. It is noticeable that this act instance is not readily decipherable without proper context or extra information — say, access to the specification of the persona applet or the persona via which this act was generated.

> **Definition 7: Cannonical Act:** *An act in which each data record is referenced using a key that is a Content Identifier (CID) corresponding to a field on the parent persona, and where the act also contains a "UUID" record among its meta records that points to the DNAP persona UUID that can uniquely identify this parent persona.*

Listing 3: Canonical Act Example

```
 {
        "c2": "1 N Road.",
        "c10": "NORTH",
        "c22": "8",
        "c18": "YES",
        "c6": "0.0903518,32.5429456",
        "OBSERVER": "P.",
        "OBSERVER_ID": "0110100000",
        "CACHE_TIMESTAMP": "2019_10_10T00:25Z",
        "UUID": "c102391918-23d9-4ebc-8753-3da5aa5f4cd7"
}
```

> **Definition 8: Annotated Act:** *An act in which each data record is referenced using a key that is a human-readable label corresponding to a field on the parent persona, and where the act might or might not contain any meta records or persona UUID.*

Note that in canonical form, an act does not make it immediately obvious what the data means. Thus, to interpret the data held in such an act, it would be necessary to consult the parent persona specification – which one can obtain by fetching the persona referenced in the contained persona UUID of a valid act.

Given such a persona UUID, one can retrieve the necessary specification via a standard API call in which the UUID is used to tell the theatre to fetch the persona with the specified UUID value. The theatre is formally defined in section 3.1.5, and an example of a persona, for the shown canonical act with UUID c102391918-23d9-4ebc-8753-3da5aa5f4cd7 is outlined in Appendix A.

In some cases though, an act can be expressed in an annotated form, in which case, the field CIDs are replaced by the actual labels or names of the fields associated with each value, on the parent persona. This annotated form is what one sees when they access the act cache via the web (see Appendix D) or mobile (see Appendix E). In Listing 4 is an example of the canonical act in Listing 3 as an annotated act.

Listing 4: Annotated Act Example

```
{
        "LOCATION": "1 N Road.",
        "WIND": "NORTH",
        "CLOUDS": "8",
        "RAINING": " YES ",
        "GPS": "0.0903518,32.5429456"
}
```

3.1.5 The Theatre

This is the core back-end engine of DNAP. More formally:

> **Definition 9: Theatre:** *Any web server implementing a REST API that satisfies the following minimum requirements:*
>
> 1) *It can accept and store personas published via an authorized studio.*
> 2) *It can allow such published personas to be accessed or referenced from a compatible histrion.*
> 3) *It can accept and store canonical acts associated with these published personas.*
> 4) *It can accept and serve requests for canonical or annotated acts for published personas when given the associated persona UUID.*

As you can tell from the above definition, the theatre acts as the "glue" that ties together the other components of the Dynamic Nuchwezi Architecture Platform. It is possible to use the studio and histrion without a theatre – for example, one could design a persona, and instead of publishing it, merely download and manually load it into a compatible histrion. But without a theatre, management and security of the tools and data thus generated becomes a nightmare!

The key role of the theatre in DNAP is to make management of tools, data and the users that operate on them much easier and straight-forward, and all this, done securely and via standard protocols and interfaces. As you shall see in the typical DNAP workflow in Fig. 5, once a persona has been designed via the studio, one has the option to first publish it onto a theatre, before they use it. This is the recommended approach and highlights the first role played by the theatre in the DNAP ecosystem.

Once a persona has been published — to a theatre that is, it is associated with a globally unique identifier, the "Persona UUID", which the theatre will know about, and using which, one can then fetch the persona specification by URL, so as to load it into a compatible histrion as a persona applet (see Fig. 4). Such access is possible via a standard REST API of the form:

```
URL-1: https://<THEATRE_FQDN>/api/persona/<PERSONA_UUID>/
```

The above URL, which is accessible using HTTP GET in the reference implementation, returns a Cwa Script that is the specification of the persona referenced by the specified PERSONA_UUID. An example of such a specification is in Appendix A.

The third core role of a theatre is that it allows data captured via these personas, on compatible histrions, to be collected and stored in one place for later use or reference. This is the reason there is a *theatre_address* parameter in each persona specification; at design time, the author of a persona needs to specify how data acquired via the persona shall be transported (the reason there is a *transport_mode* parameter) and where that data shall be submitted to – the

theatre_address, which in the default case (in which acts are submitted using HTTP POST), is the URL of a REST endpoint on a compatible theatre. For example, in the reference implementation of DNAP, the default theatre address for personas built via the reference studio implementation is of the form:

```
URL−2: https://<THEATRE_FQDN>/api/act/create/
```

It is important to note that not all personas need specify a URL for a theatre address; where the *transport_mode* parameter is specified as “EMAIL” or “SMS” for example, the *theatre_address* parameter expects an email address or phone number respectively. In that case, it is expected that the *theatre_address* be specified as a valid URI using the “mailto:” or “tel:” schemes respectively – though, in the reference implementation of the histrion, the scheme part can be eliminated, so that this parameter merely takes in a Uniform Resource Name (URN).

Finally, the theatre should be able to accept HTTP GET requests for, and serve any data submitted to it for the specified persona, given the persona UUID. In general, such a request involves making a call to a REST endpoint on the theatre, using a URL of the form:

```
URL−3: https://<THEATRE_FQDN>/api/persona/<PERSONA_UUID>/acts/
```

Which returns all currently stored data for the persona specified using PERSONA_UUID as annotated acts. Alternatively, there is

```
URL−4: https://<THEATRE_FQDN>/api/persona/<PERSONA_UUID>/racts/
```

which serves the same data, but as canonical acts. Note that in either case, the result of performing such a request on the theatre returns a JSON array, whose only contents are the acts or which is empty – in the case where no data exists yet for the given persona or where access to the act stream is restricted without proper authentication.

This mechanism allows data acquired via DNAP to be immediately consumable from almost any standard data processing interface – web-browsers; text-editors; command line tools such as wget, curl or jq; spread-sheet programs such as Microsoft’s Excel or Google’s Sheets, or even professional data analysis and business intelligence platforms such as PowerBI, Tableau and others.

By requiring that the theatre conform to the above specification, it then makes building of data-driven software, or business automation software, such as data acquisition, data sharing and data analysis tools via DNAP – or rather using the “Persona Pattern” for data engineering, much more straight forward, and standardizes such a process; once a tool is published, it is immediately and readily obvious how data clients shall reference and or access data associated with the tool. Further, this data-access interface can be relied upon without worrying about how or where the actual data storage happens.

The theatre REST API abstracts away the underlying web and database technologies required to actually implement a robust distributed data storage and retrieval platform, and given that these features come straight out of the box for those using a standard DNAP instance, designing, constructing and utilizing data processing apps with the persona pattern becomes so simple, even for non-technical users or those with no experience building client-server, web-based or database-powered software systems. This later point is one of the original key motivations for the Dynamic Nuchwezi Architecture Platform.

In relation to the above, it is important to note that for purposes of keeping things simple, DNAP further standardizes how access to the data querying and data analytics interfaces of a given persona are to be done. This especially applies to web-browser-oriented use-cases. Thus, once published to a theatre, and once the persona UUID is known, then access to the web histrion and diviner of the associated persona is possible via the standard URI forms:

```
URL−5: https://<THEATRE_FQDN>/persona/<PERSONA_UUID>/histrion/
```

and

```
URL−6: https://<THEATRE_FQDN>/persona/<PERSONA_UUID>/diviner/
```

For the histrion and diviner, respectively. The histrion has already been discussed in the previous sections, next we shall consider the diviner.

3.1.6 The Diviner

Definition 10: Diviner: *A platform-app that can parse and interpret Cwa Script from which it then renders act stream analytics applets.*

With reference to this definition, we note that like the histrion, the diviner is a platform-app, except that instead of rendering data acquisition or data sharing mini-programs, it renders data analysis tools for the target persona(s) - what are called "act stream analytics applets" in the definition, to differentiate them from persona applets (see Definition 2). We may also refer to act stream analytics applets as simply "persona analytics applets".

The motivations for the diviner are simple; for the typical use case, one wants to collect data to later browse or analyze it. Thus, instead of merely offering the user a means to collect and share data - which the histrion and theatre does, the diviner further offers them a means to readily browse and analyze such collected data in the simplest way possible.

As of this writing, the only diviner implementation available runs on the web, and as such, is accessible via a web browser just like the studio, theatre, and web histrion components of DNAP. In the reference implementation of the mobile histrion, the diviner is accessible directly from within the histrion, via an embedded webview that loads the analytics applet associated with a specific cached persona.

In the previous section, we have already seen how when given a published persona’s UUID, and the theatre via which it is published, it becomes possible to access its diviner (refer to URL-6). However, like the web histrion, there exists a standalone diviner reference implementation that one can access directly, and then, equipped with a valid persona URI, one can load and analyze incoming or existing data associated with the persona.

3.2 How DNAP Works

In this section, we wish to illustrate how DNAP works. To do that, we shall explore how DNAP is used, with the

data engineering case as a focus scenario to help keep this exploration simple. Fig. 5 shows the typical work-flow using a flow-chart. Essentially, an end-to-end data engineering task involves designing a data collection tool such as are traditionally called "forms", publishing it somewhere, capturing data with it, storing the data, then later analyzing, or sharing the collected data.

In summary, here is how an end-to-end data engineering task proceeds on DNAP:

1) Design and compose your persona via a studio.
2) Publish the persona to a theatre.
3) Load the persona into a histrion (could fetch the persona from the theatre or use a downloaded copy from the studio).
4) Activate the persona on the histrion.
5) Interact with the persona – e.g. capture data using the persona.
6) Verify the data (automatic step upon submission or saving).
7) Save the data (useful when working offline or in areas of poor connectivity)
8) [optionally] Edit the saved data (currently only for the mobile histrion via the "clone" feature)
9) Submit the data to the theatre (requires an active data connection).
10) Visit the diviner instance of your persona (link shared via studio at persona publication time or via the theatre after publication — refer to URL-6).
11) Fetch, browse and analyze any data found on the persona's act stream endpoint (see URL-3 and URL-4)
12) Export, download or share visuals and any other information generated from this analysis.

Note that in practice, it is possible to kick-start this process by starting from an already existing persona instead of creating one from scratch. This facility is in-built on the reference implementation of the studio, and it allows one to kick-start their project in one of several ways:

1) You can clone existing personas via their resource URLs. The essential persona cloning request URL follows the pattern:

```
URL−7: https://<STUDIO_FQDN>/?purl=<PERSONA_URL>
```

2) You can clone an existing persona by finding and loading its *.persona file (e.g. one previously downloaded via a studio or a histrion).
3) You can clone a persona whose specification you have as raw Cwa Script code (e.g. copying and pasting the code in Appendix A into a studio).
4) You can clone a persona by selecting it from the list of template personas shown under the "Templates" section on a studio.
5) You can clone a persona by selecting it from the list of published personas on the reference implementation theatre, via the "Clone Persona" action.

The current implementation of DNAP's Studio and the associated Cwa Script language for the design of personas allows at minimum, the design of programs that can do the following:

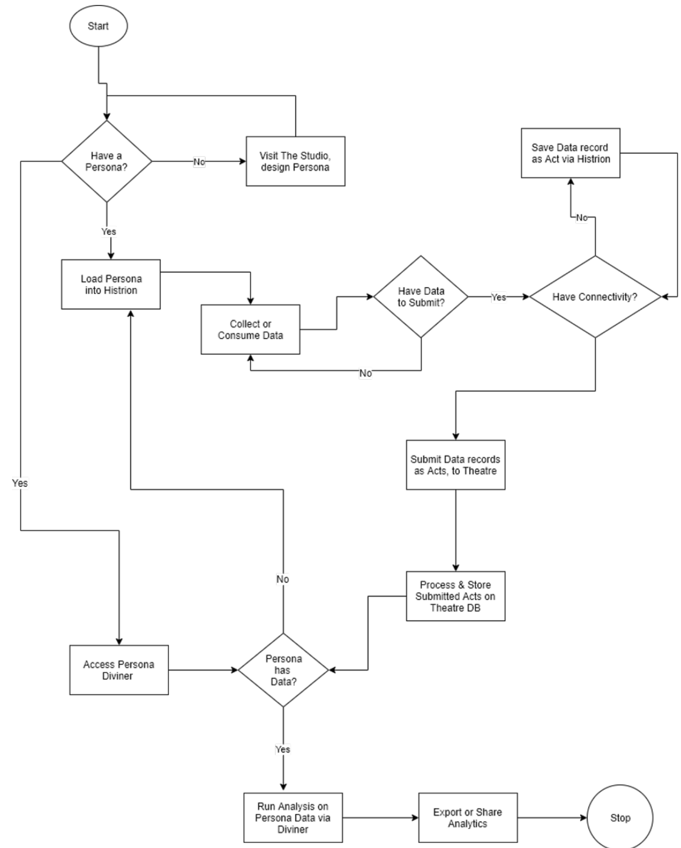

Fig. 5: DNAP Work Flow

1) Interactive data capturing mechanisms: text, numbers, email, phone contact, date, time, etc.
2) Sensor-based data capturing mechanisms: currently only GPS implemented.
3) Multimedia data dissemination mechanisms: audio playback, video playback, web page preview, GIF playback, etc.
4) Simple data dissemination mechanisms: display image, display text, display web page, etc.
5) Relational data compositing mechanisms: auto-populated menus via API, linking from persona to persona at runtime, etc.
6) Meta-data mechanisms: hidden fields, user-tracking fields, auto-timestamps, timed program launching triggers etc.

Finally, it should be noted that the current DNAP implementation includes mechanisms for integrating DNAP's components into other existing software so their existing functionality and looks can be extended or augmented - this is a means to readily evolve existing softwares for example. The following are the key approaches currently possible , spanning web, mobile and desktop systems:

1) Extending existing web applications via embedded web histrion – leverages the standard HTML `iframe` tag. This method can allow adding new data collection or data dissemination capabilities to an existing system.

THIS WORK HAS BEEN SUBMITTED TO THE IEEE FOR POSSIBLE PUBLICATION. COPYRIGHT MAY BE TRANSFERRED WITHOUT NOTICE, AFTER WHICH THIS VERSION MAY NO LONGER BE ACCESSIBLE.

2) Extending existing mobile applications via embedded mobile histrion – leverages an opensource Android Library for the histrion, that is simply added to an existing mobile application project using a single import statement in the project's Gradle file. This method can allow adding new data collection or data dissemination capabilities to an existing mobile application.
3) Extending an existing web application via an embedded DNAP web diviner – leverages the standard HTML `iframe` tag. This method can allow adding new data analysis or business intelligence capabilities to an existing system.
4) Extending an existing mobile application via an embedded DNAP web diviner – leverages the standard WebView component on the associated mobile platform. This method can allow adding new data analysis or business intelligence capabilities to an existing mobile application.
5) Extending existing desktop applications via embedded web histrion – leverages the standard WebView or Browser components on the associated operating system and language. This method can allow adding new data collection or data dissemination capabilities to an existing system.
6) Extending existing desktop applications via embedded web diviner – leverages the standard WebView or Browser components on the associated operating system and language. This method can allow adding new data analysis or business intelligence capabilities to an existing system.

The full specification and discussion of the supported data types, capabilities, and extension mechanisms of DNAP and Cwa Script in particular, shall be expounded upon in a future publication, to keep the current from becoming too lengthy.

4 EVALUATION OF DNAP

In this section DNAP is evaluated and validated against other technologies of a similar kind, using a model termed the Software Operating Environment.

4.1 The Software Operating Environment (SOE) Model

For purposes of simplifying the analysis, a set, Ψ of related, but distinct technologies shall be considered. We shall limit our evaluation set to only five technologies, codified thus:

$$\Psi = \{T_1, T_2, T_3, T_4, T_5\} \quad (1)$$

and where Ψ is a set of SOEs. A SOE can be thought of as an ecosystem of related software technologies that can allow one to design, build, and run software in a given programming language on a given operating system. More formally;

Definition 11: Software Operating Environment (SOE): *A SOE, `T`, over a programming language, `P`, on a host operating system family, `O`, also written `T(P):O`, is a system satisfying all the following properties:*

1) *`T` has a software construction environment for `P` in `O`.*
2) *`T` accepts `P` source code and outputs a `P` executable file in `O`.*
3) *`T` has a software execution environment for `P` executables in `O`.*
4) *`O` has at least one operating system instance where all properties (1), (2) and (3) hold.*

In our analysis of DNAP using the SOE criteria, the set Ψ is constituted as depicted in Table 1.

And the analysis of Ψ shall be based on the following SOE metrics:

1) **Construction Process Complexity Lower Bound (CPCLB)**: When a SOE's simplified software development process is modeled using a state-transition graph, with each definite step in the process mapped to a state in the grap, then CPCLB is the minimum number of possible state transitions required, from the program design state to the program execution state.
2) **Supported Direct Target Platforms (SDTP)**: The number of distinct operating system families that a given program can be executed on without re-write or the number of distinct run-time families supporting direct execution of a given program. In the SOE model, SDTP is the count of distinct operating systems families `O`, that satisfy property SOE(2) and SOE(3) for a given SOE `T` and language `P`.
3) **Lines of Code for the Minimum Basic Output Program (LOCMBOP)**: the minimum number of lines of code (LOC) required to write the Most Basic Output Program (MBOP), which is defined as: a minimal program that only does the following:
    a) Prints "Hello World" (HW) to standard output
    b) Returns
4) **Number of Characters for the Minimum Basic Output Program (NOCMBOP)**: the total number of characters or symbols required to express the MBOP in a given language, ignoring unnecessary characters – for example trailing or padding whitespace before, after or within the body of an instruction, and comments in the program's source file.
5) **Minimum Basic Output Program Brevity-Ratio (MBOPBR)**: The ratio of the length of HW to the length of the entire MBOP in a given programming language, where length is based on NOC.
6) **Lines of Code for the Minimum Basic Input Program (LOCMBIP)**: the minimum number of lines of code (LOC) required to write the Minimum Basic Input Program (MBIP), which is defined as: a minimal program that only does the following:
    a) Prints a prompt, "What is your name please?"
    b) Accepts and stores input from the user.
    c) Prints the captured value.
    d) Returns
7) **Lines of Code for the Minimum Complex Output Program (LOCMCOP)**: the minimum number of

TABLE 1: Assessed Software Operating Environments

$T \in \Psi$	Name of SOE, T	Hosting Operating System Family, O	Software Construction Language, P
T_1	Android Platform	Windows, Android, MacOS, Linux	Java
T_2	.NET Framework	Windows	C#
T_3	DNAP	Windows, Android, Linux, MacOS, iOS	Cwa Script
T_4	Java SE Platform	Windows, Linux, MacOS	Java
T_5	Python	Windows, Linux, MacOS	Python

lines of code (LOC) required to write the MBOP, and that puts terseness ahead of simplicity.

8) **Lines of Code for the Minimum Complex Input Program (LOCMCIP)**: the minimum number of LOC required to write the MBIP, and that puts terseness ahead of simplicity.

4.2 Concerning the LOC Metric and an intepretation of the LOC* metrics used in the SOE model.

Before we proceed with our analysis of the set Ψ, it is important that we have a clear understanding of what some of the employed metrics mean. For example, one justification for using the LOC* metrics in the comparison of software operating environments is because they capture an important fact about a programming language; the simplicity of its programming interface and its expressive power at the source code level.

In Table 2, are shown links to the Minimum Basic Input-Output Program (MBIOP) for several programming languages including those spanned by our analysis set Ψ. Further, the LOC metrics associated with the MBIOP and the Minimum Complex Input Output Program (MCIOP) also discussed later in this section, are shared, so as to help illustrate some key points about the SOE model and the metrics used in our analysis.

TABLE 2: Languages and their LOCM*IOP metrics

Language	MBIOP	LOCMBIOP	LOCMCIOP
Android (Java)	Appendix H.2	27	1
Cwa Script	Appendix H.3	13	1
Bash	Appendix H.4	2	1
Python	Appendix H.5	2	1
C#	Appendix H.1	18	1
Java SE (Java)	Appendix H.6	10	1

It shall be noticed that for languages such as Bash, C and Cwa Script, the use of whitespace to delimit statements in the source of a program is optional. Further, concerning the LOCMC*P and LOCMB*P metrics, it can be argued that whereas in implying that a program is "basic", the simplicity of the programming interface includes the elimination or limiting of parsing difficulties for both the human – programmer or reviewer for example, and machine – parser or executor, in the "complex" case however, we assume that a programming interface need only eliminate parsing complexity for the machine – the implication being, in such a case, the programs thus expressed are not necessarily meant for humans to parse or interpret readily at the source level.

In the "complex" case then, brevity and succinctness are favored over simplicity and legibility, and the reverse happens in the "basic" case. This idea of the complex program source is similar to the concept of "program minification" as employed in the optimization, compression and sometimes obfuscation of scripts as found in web development [32] – typically motivated by a desire to save space or bandwidth, and also secure the code. In our case though, obfuscation is not a goal, but the other ideas apply.

To put the above philosophy into perspective, consider that when comparing two programming languages, P1 and P2 using the LOCMBOP metric, then if language P2 supports the use of whitespace statement delimiters in any programming scenario, such as for Perl, Python and Bash, then the basic case supposes that each distinct statement in the program's source code be delimited from the others by whitespace – typically a new line character or carriage return or combination of both. The effect is increased legibility of the program source, and elimination of or reduction in manual parsing complexity. This implies that in expressing the MB*P for example, the following predicates hold:

Predicate 1.

$$LOCMBOP \geq 1 \tag{2}$$

Predicate 2.

$$LOCMBIP \geq 1 \tag{3}$$

Further, note that the definition of the LOCMBIP metric implies that the Minimum Basic Input Program (MBIP) spans both an input and output operation at minimum. By this line of thought then, the MBIP is equivalent to or is a subset of the Minimum Basic Input Output Program (MBIOP), and it is not hard to show that

Predicate 3.

$$MBOP \subset MBIP \subset MBIOP \tag{4}$$

And that

Predicate 4.

$$LOCMBIOP \geq 1. \tag{5}$$

In compressing the above ideas, we have the following result:

Predicate 5.

$$LOCMBIOP \geq LOCMBIP \geq LOCMBOP \geq 1 \tag{6}$$

On the other hand, if we consider the "complex" case, and if we suppose that P1 supports only non-whitespace delimiters; such as the traditional ";" used in languages such as Java, C and C#; then simplicity and legibility can be sacrificed, and the MBOP can be expressed in P1, in such a way that all, and any distinct executable statements required to express the program lie on a single line in the source file. It can then be shown that the following results are true and equivalent for P1 in the complex case:

Predicate 6.

$$LOCMCOP = LOCMCIP = LOCMCIOP = 1 \quad (7)$$

Or equivalently. . .

Predicate 7.

$$\alpha = 1 \mid \forall \alpha \in \{LOCMCOP, LOCMCIP, LOCMCIOP\} \quad (8)$$

Further, in the complex case, for language P2:

Predicate 8.

$$\alpha \geq 1 \mid \forall \alpha \in \{LOCMCOP, LOCMCIP, LOCMCIOP\} \quad (9)$$

4.3 Concerning the MBOPBR Metric and measuring programs by their number of characters (NOC) instead of LOC in the SOE model.

It shall be noticed that in the definition of the MBOPBR metric, reference is made to two strings – the HW constant, whose length is 11 in almost all languages, and the MBOP, whose length depends on the programming language being analyzed.

Note that unlike the very traditional, popular metric used to compare the size of software expressed as source code – the Lines of Code (LOC) metric, the Number of Characters (NOC) metric ignores the expression of the program as a sequence of lines, and focuses on the expression of the program as a sequence of any supported symbols – or rather elements from the character set ω of the language – with unnecessary characters and commented-out sections being ignored as was explained in the NOC definition (see SOE metric #4). Both measure length, however, the NOC metric, especially as used in the brevity metric MBOPBR, better expresses how verbose a given language is relative to another, with respect to a given, fixed, reference operation – the printing of the constant HW to standard output in the case of MBOPBR. Other brevity ratio metrics could be thought of, that deal with other operations such as the multiplication of two constants, the input and output operation as in MBIOP etc.

To appreciate the power of these proposed brevity ratio metrics, and especially the MBOPBR, let us build a simple mathematical model to help relate the size of a program's source file to the size of task the program needs to accomplish. In the case of the MBOPBR, assume there is a linear relationship between the number of characters one needs to output in MBOP, denoted α, and the actual total number of characters in the entire source file for the MBOP, already defined as NOCMBOP. Then we can mathematically say that

$$NOCMBOP = \alpha K + C \quad for \quad K, C \in \Re, \alpha \in \mathbb{N} \quad (10)$$

Note that C would represent any unavoidable code required in the program's source file irrespective of what is being done - for example the mandatory declaration of the *main()* function in C-family languages at minimum - required of any program that is. K would represent how much the program's source file needs to grow relative to what is being done - say size of text being printed in the case of MBOP. In the simplest case, $K = 1$. Then, from the definition of MBOPBR, we know that

$$MBOPBR = |HW| : NOCMBOP \equiv \alpha : (\alpha K + C) \quad (11)$$

In interpreting the MBOPBR metric then, the following summarizes the important inferences one can make:

1) **MBOPBR = 1**: means the language allows to print to standard output the constant HW without any other boilerplate or overhead information in the program's source file. Put another way, the language will print to standard output the message "Hello World" as long as that string, and nothing else, is written in the source file (implies $C = 0$ and $K = 1$). Subtle, but very important – from a programming language design perspective. One such language is HTML.
2) **MBOPBR = 0**: means the language being used to express MBOP is overly verbose or rather bloated. Using the model in Equation 10, it means that either of the two numeric parameters, K and C, are much larger than α. Expressed mathematically: $|K| \gg |\alpha| \vee |C| \gg |\alpha| \rightarrow |\alpha| \ll |\alpha K + C| \rightarrow MBOPBR \approx 0$
3) **0 < MBOPBR < 1**: simply means printing HW to standard output in the language involves some explicit commands or directives other than merely writing the HW constant in the program's source file. Most, if not all major programming languages in use today fall under this category, including the newly proposed Cwa Script used in DNAP. Also, this scenario often means reading the program's source code makes it unambiguous how the language expresses the standard output mechanism under analysis.

4.4 Concerning the CPCLB Metric and Software Development Processes

In this section we shall bring our attention to the matter of how software operating environments relate with respect to their software development processes. A method for quantitatively ranking SOEs is required, and it should be language agnostic. In the SOE metrics introduced in this section, is the CPCLB metric, and it has the qualities we need.

To illustrate how analysis of SOEs proceed using the CPCLB metric, consider the Minimal Software Development Process (MSDP) supported on each of the SOEs in our analysis set Ψ. The following state-transition diagrams illustrate the minimal software development process across all five technologies in Ψ, and we then compute the CPCLB metric for each SOE based on its MSDP graph.

In Figs. 6, 7, 8, 9 and 10 are illustrated the MSDP graphs for each of the SOEs in Ψ, one after the other.

THIS WORK HAS BEEN SUBMITTED TO THE IEEE FOR POSSIBLE PUBLICATION. COPYRIGHT MAY BE TRANSFERRED WITHOUT NOTICE, AFTER WHICH THIS VERSION MAY NO LONGER BE ACCESSIBLE.

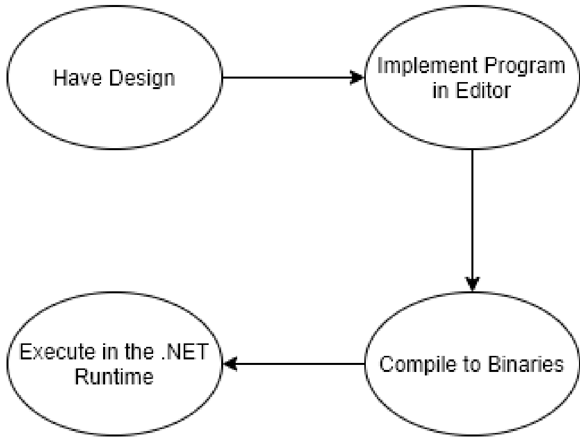

Fig. 6: .NET Framework MSDP

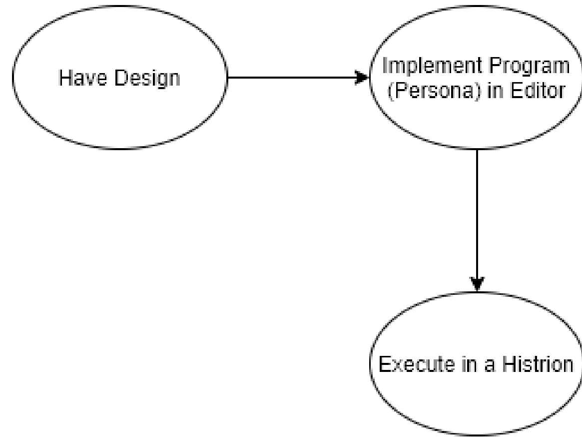

Fig. 8: Cwa Script MSDP

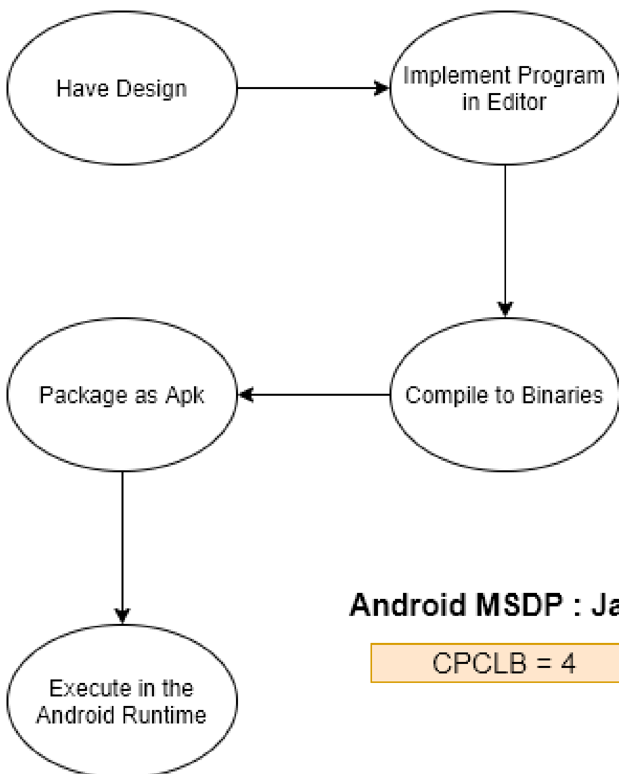

Fig. 7: Android MSDP

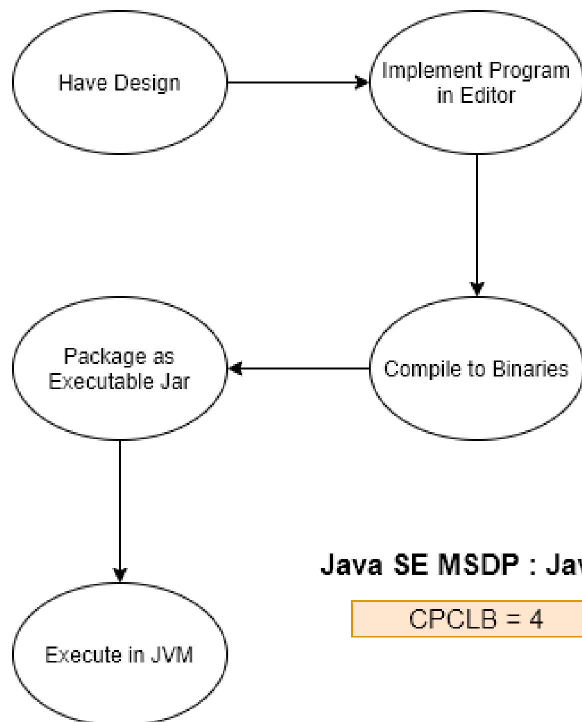

Fig. 9: Java SE MSDP

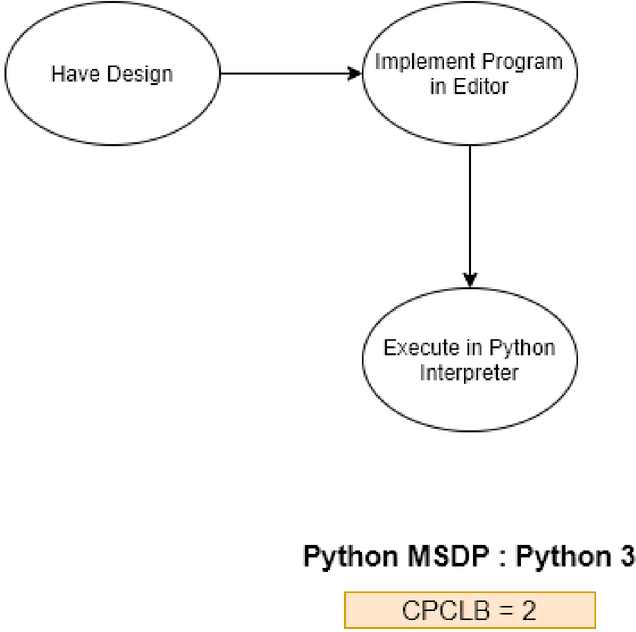

Fig. 10: Python MSDP

The theory behind the computation of the CPCLB metric is simple; given that an ideal software development process (SDP) involves a series of activities and transitions between them, from the conception of a program design, through its implementation to its eventual execution, it is possible then, to model such a process using a state transition graph or basically a directed graph.

Essentially, for a given SOE, we can express the SDP as:

$$SDP : \theta^N \rightarrow [\theta_i, \theta_j]^C \quad (12)$$

where θ^N is the set of all N distinct states in the SDP, and that there are C edges in the SDP where pairs of distinct states such as $[\theta_i, \theta_j]$, represent an edge, or equivalently, a state transition from state θ_i, to θ_j in the SDP graph.

It can then be said that for two distinct SOEs T_i, and T_j with MSDPs having CPCLB values C(T_i) and C(T_j) respectively, then

1) If **C(T_i) = C(T_j)** then T_i and T_j are equivalent in their software construction complexity.
2) If **C(T_i) < C(T_j)** then T_i is simpler than T_j with respect to software construction complexity.

The analysis of the SOE set Ψ with respect to the CPCLB metric is conducted in the next section.

4.5 Evaluating DNAP using the SOE Metrics

In our adoption of the SOE model to quantify the relative performance of DNAP against other similar technologies, we have assumed that the host operating system, O, is the same across the set Ψ, and that it is Windows 10 so as to keep the data collection process simple.

Apart from metrics LOCM*IOP, it shall be assumed that the program used in the SOE evaluation is the Minimum Basic Output Program (MBOP), which is defined in metric LOCMBOP above, and which traditionally is called the "Hello World" program.

Considering the operating system families that have been considered in our SOE model – for example when determining the value for the SDTP metric, the following common, five, user-oriented operating system families are considered; Microsoft Windows, Apple MacOS, Linux, Android and Apple's iOS.

Table 3 presents our findings after applying the SOE model to the set Ψ above:

TABLE 3: SOE Metrics across Ψ

SOE Metric \ $T \in \Psi$	T_1	T_2	T_3	T_4	T_5	**Average**
CPCLB	4	3	2	4	2	3
SDTP	4	3	5	3	5	3.6
LOCMBOP	3	6	6	5	1	4.2
$\|HW\|$	11	11	11	11	11	11
NOCMBOP	134	76	40	74	19	68.6
MBOPBR	.082	.145	.275	.149	.579	0.246
LOCMBIOP	27	18	13	10	2	14
LOCMCIOP	1	1	1	1	1	1

We shall proceed with the analysis of the set Ψ, and especially contrast DNAP with its peers based on the given SOE metrics and their values as summarized in Table 3.

First, it shall be noticed that Android (T_1) and Java SE (T_4), have the highest software construction complexity, with CPCLB values at 4, the highest in the set. These are followed by the .NET Framework, with a CPCLB value of 3, and we find that the simplest SOE with respect to software construction complexity are Python and DNAP, with only a single intermediate step between program design and execution (CPCLB = 2), both below the average (CPCLB = 3).

It shouldn't be hard to notice that one explanation for the low CPCLB value in the case of Python (T_5) and DNAP (T_3) is that they require no compilation, no special packaging of source code for a program to be ready for execution in its final environment. Thus, we can conclude that interpreted languages in general will have a low software construction complexity than compiled languages, and that Cwa Script falls in this category too.

Considering the SDTP metric, which measures how cross-platform an SOE is with respect to the set of operating system families considered, we find that .NET Framework (T_2) and Java SE (T_4) have the lowest score (SDTP=3), and one explanation for this is that neither SOE have support for direct development and execution of their programs on either Android or iOS. Python and DNAP score highest (SDTP=5), particularly because they are readily executable anywhere an interpreter for the languages can be hosted, and in the case of DNAP, this is solved with the mobile or web-based Histrion. The latter being platform-agnostic, thus the high SDTP score.

Next, we consider the LOCMBOP metric. This metric helps estimate how simple it is to write a basic program in a given language. We find that Python offers the simplest approach, with the entire MBOP fitting on a single line, and in a single instruction (see Appendix L.3). DNAP's Cwa Script and .NET Framework come in last, with a LOCMBOP=6. Given that MBOP only involves a single command

printing the "Hello World" string, we can conclude that a high LOCMBOP value implies the languages requires some unavoidable source-level boilerplate code or structure even for the simplest of scenarios. It shall be interesting to note that despite Android (T_1) and Java SE (T_4) having almost similar dialects of the Java programming language, the low LOCMBOP value of 3 for the former is because of the special hack that allows a minimal program in Android, such as the MBOP, to leverage only the program's Manifest file - which uses XML and not Java (see Appendix I.1).

The brevity ratio metric, MBOPBR, is derived from two other metrics - $|HW|$, which is the length of the string constant "Hello World" as measured in a given language, and which we find to be constant at 11 for the entire set Ψ, and NOCMBOP, which as we have seen previously, depends on how MBOP is written in the language being analyzed. First, we notice that none of the technologies in Ψ support the special case MBOPBR = 1, and neither the other special case MBOPBR = 0. However, knowing that a high brevity ratio means the language is very precise, or terse, we then can conclude that in the set Ψ, the most precise SOE is Python, with MBOPBR = 0.579. Ranked by their preciseness then, with respect to the MBOP, we have the order; Python, DNAP, Java SE, .NET Framework and lastly Android.

Finally, we have LOCM*IOP metrics, that compare the SOEs in Ψ by their lines of code for the most basic program that performs both input and output - the MBIOP and MCIOP defined earlier on. We find that in the "complex" scenario - in which code readability is not important, all SOEs in Ψ can have their MCIOP expressed as a one-line program, thus LOCMCIOP = 1. However, in the basic case, where readability matters, we find that Android's MBIOP, written in Java, requires the longest program (LOCMBIOP = 27), and unsurprisingly, Python requires the shortest - only two lines of code. Note that DNAP, using Cwa Script, is outperformed by only Python and Java SE as per this metric.

5 Conclusion

In this paper, we have explored three important problems in modern software engineering: portability, extensibility, and construction complexity of software systems. We have found that much as the research community and the software industry have contributed several well documented solutions to some of these problems, that there is still room for improvement, and that some of the outstanding problems justified new, alternative solutions.

DNAP, which is formally specified and then contrasted against many of these existing solutions for the first time in this paper, has been found to be a compelling, and realistic solution to all three of the core problems identified. In particular, using the SOE model, it has been found that DNAP offers a simpler software construction process than many of the prevailing software construction methods in use today, and that DNAP's Cwa Script programming language performs relatively well on several platform-agnostic metrics as explored in the SOE model first introduced in this paper.

It has been shown how DNAP works at a high level, and how it can be applied in real-world business automation problems hinged on the collection, aggregation, analysis and sharing of data using mobile and web technologies especially. As DNAP's Persona Pattern allows the creation of not only data acquisition software, but also data dissemination tools, including the construction of multimedia and sensor-powered mini-programs running inside of a histrion, it is expected that DNAP's usage will expand much further into many or all domains of business automation, so that currently complex systems such as e-learning systems, ERPs, SOA services and more shall be possible to build in a fraction of the time, and in a manner much more simpler than is currently possible.

We have seen how DNAP's mobile and web histrion can be integrated in existing software via standard methods such as use of iframes, browser components and an open source library. Further, it has been shown how the diviner, which offers data analysis and business intelligence capabilities can likewise be integrated into existing web, desktop and mobile applications via the iframe or embedded browser approaches. There is still room for improvement in how DNAP can be leveraged to extend or evolve legacy and new software systems irrespective of operating system, and this is a very promising area in terms of bringing rapid prototyping to legacy software evolution.

DNAP is still under active development, with only a few reference implementations in place – such as a web-based IDE for personas, a single runtime for personas on the web, and one for personas on mobile. However, future research could extend DNAP's reach so the histrion is natively implemented for all major operating systems – which would eschew the need to run personas in a browser (which currently requires network connectivity) or leverage Android emulators (for running the mobile histrion say on a desktop operating system). Further work needs to be done to improve the Cwa Script language's support for the creation of advanced programs in a basic and precise way using the same, simple, visual paradigm currently employed, so as to make modern software construction accessible to more users, without sacrificing on performance and robustness of the developed systems.

Lastly, it is envisaged that since DNAP's theatre not only offers a means to centrally manage and serve large collections of mini-programs – especially community contributed ones, but also the discovery, access to, transformation and sharing of the data from these programs, that the use of ideas from Service Oriented Architectures (SOA), and Artificial Intelligence shall make this approach more useful as the technology grows and gains more adoption.

Acknowledgment

Special thanks to African Leadership Institute (AFLI) and Advocates Coalition for Development (ACODE), both based in Uganda, for providing part of the support that made a lot of the prototyping and feasibility assessment of DNAP possible. Last but not least, credit is given to the Department of Better Technology (DoBT), USA, for their open-source formbuilder project that first inspired DNAP's concept of a persona studio.

References

[1] M. Müller, C. Schindler, and W. Slany, "Pocket code-a mobile visual programming framework for app development," in *2019*

IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft). IEEE, 2019, pp. 140–143.

[2] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[3] B. Harvey, D. D. Garcia, T. Barnes, N. Titterton, D. Armendariz, L. Segars, E. Lemon, S. Morris, and J. Paley, "Snap!(build your own blocks)," in *Proceeding of the 44th ACM technical symposium on Computer science education*, 2013, pp. 759–759.

[4] K. Ito, "Work in progress: Block pictogramming a block-based programming learning environment through pictogram content creation," in *2020 IEEE Global Engineering Education Conference (EDUCON)*, 2020, pp. 1669–1673.

[5] B. Messenlehner and J. Coleman, *Building Web Apps with WordPress: WordPress as an Application Framework*. O'Reilly Media, 2019.

[6] M. Mallette and D. Barone, "On using google forms," *The Reading Teacher*, vol. 66, no. 8, pp. 625–630, 2013.

[7] K. S. Chang, B. A. Myers, G. M. Cahill, S. Simanta, E. Morris, and G. Lewis, "A plug-in architecture for connecting to new data sources on mobile devices," in *2013 IEEE Symposium on Visual Languages and Human Centric Computing*, 2013, pp. 51–58.

[8] H. Han, J. Guo, and T. Tokuda, "Towards flexible integration of any parts from any web applications for personal use," in *First International Workshop on Lightweight Integration on the Web (ComposableWeb'09)*, 2009, p. 69.

[9] G. Yang, J. Huang, and G. Gu, "Iframes/popups are dangerous in mobile webview: studying and mitigating differential context vulnerabilities," in *28th {USENIX} Security Symposium ({USENIX} Security 19)*, 2019, pp. 977–994.

[10] T. Singh, "The hotspot java virtual machine: memory and architecture," *Int. J. Allied Pract. Res. Rev*, pp. 60–64, 2014.

[11] D. Avdic, "React native vs xamarin-mobile for industry," 2019.

[12] M. Noone and A. Mooney, "Visual and textual programming languages: a systematic review of the literature," *Journal of Computers in Education*, vol. 5, no. 2, pp. 149–174, 2018.

[13] O. Brandes, "Tersus visual programming platform," in *2011 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 2011, pp. 245–246.

[14] S. Dasgupta, S. M. Clements, A. Y. Idlbi, C. Willis-Ford, and M. Resnick, "Extending scratch: New pathways into programming," in *2015 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*. IEEE, 2015, pp. 165–169.

[15] L. Ng, "The end is near for mobile apps," 2018. [Online]. Available: https://medium.com/s/story/mobile-apps-will-disappear-soon-4b4e54f46eb8

[16] K. Cheng, M. Schreieck, M. Wiesche, and H. Krcmar, "Emergence of a post-app era–an exploratory case study of the wechat mini-program ecosystem," in *15th International Conference on Wirtschaftsinformatik*, 2020.

[17] D. A. Normann, E. Þ. Aðalsteinsson, and Þ. Þormóðsson, "We2book wechat mini program development," Ph.D. dissertation.

[18] L. Hao, F. Wan, N. Ma, and Y. Wang, "Analysis of the development of wechat mini program," in *J. Phys.: Conf. Ser*, vol. 1087, 2018, p. 062040.

[19] A. Grosskurth and M. W. Godfrey, "Architecture and evolution of the modern web browser," *Preprint submitted to Elsevier Science*, vol. 12, no. 26, pp. 235–246, 2006.

[20] R. Zamudio, D. Catarino, M. Taufer, B. Stearn, and K. Bhatia, "Topaz: a firefox protocol extension for gridftp based on data flow diagrams," 2006.

[21] Wikipedia contributors, "List of java virtual machines — Wikipedia, the free encyclopedia," 2020, [Online; accessed 27-July-2020]. [Online]. Available: https://en.wikipedia.org/w/index.php?title=List_of_Java_virtual_machines&oldid=954623743

[22] J. O. Ogala and D. V. Ojie, "Comparative analysis of c, c++, c# and java programming languages," *GSJ*, vol. 8, no. 5, 2020.

[23] G. Blajian, R. Eggen, M. Eggen, and G. Pitts, "Mono versus. net: A comparative study of performance for distributed processing." in *PDPTA*, 2006, pp. 45–51.

[24] Microsoft, "Xamarin.forms supported platforms," 2020. [Online]. Available: https://docs.microsoft.com/en-us/xamarin/get-started/supported-platforms?tabs=windows

[25] T. Berners-Lee and M. Fischetti, *Weaving the Web: The original design and ultimate destiny of the World Wide Web by its inventor*. DIANE Publishing Company, 2001.

[26] K. Kumar and S. Dahiya, "Programming languages: A survey," *International Journal on Recent and Innovation Trends in Computing and Communication*, vol. 5, no. 5, pp. 307–313, 2017.

[27] M. Popa, "Considerations regarding the cross-platform mobile application development process," *Academy of Economic Studies. Economy Informatics*, vol. 13, no. 1, p. 40, 2013.

[28] C. M. M. Pinto, "From native to cross-platform hybrid development: Codegt, design and development of a mobile app for erp," Ph.D. dissertation, 2018.

[29] A. Biørn-Hansen, T.-M. Grønli, G. Ghinea, and S. Alouneh, "An empirical study of cross-platform mobile development in industry," *Wireless Communications and Mobile Computing*, vol. 2019, 2019.

[30] T. Bray *et al.*, "The javascript object notation (json) data interchange format," 2014.

[31] H. Andrews and A. Wright, "Json schema: A media type for describing json documents," in *draft-handrews-json-schema-02 ed*, 2019. [Online]. Available: https://json-schema.org/latest/json-schema-core.html

[32] Y. Sakamoto, S. Matsumoto, S. Tokunaga, S. Saiki, and M. Nakamura, "Empirical study on effects of script minification and http compression for traffic reduction," in *2015 Third International Conference on Digital Information, Networking, and Wireless Communications (DINWC)*. IEEE, 2015, pp. 127–132.

APPENDIX A EXAMPLE PERSONA SPECIFICATION: NJURA

Listing 5: Example Persona Specification: NJURA


```
{
    "fields": [{
            "field_type": "text",
            "cid": "c2",
            "pattern": "",
            "required": true,
            "label": "LOCATION",
            "field_options": {
                    "size": "small"
            }
        }, {
            "field_type": "dropdown",
            "cid": "c10",
            "pattern": "",
            "required": true,
            "label": "WIND",
            "field_options": {
                    "options": [{
                            "checked": false,
                            "label": "NORTH"
                    }, {
                            "checked": false,
                            "label": "NORTHEAST"
                    }, {
                            "checked": false,
                            "label": "EAST"
                    }, {
                            "checked": false,
                            "label": "SOUTHEAST"
                    }, {
                            "checked": false,
                            "label": "SOUTH"
                    }, {
                            "checked": false,
                            "label": "SOUTHWEST"
                    }, {
                            "checked": false,
                            "label": "WEST"
                    }, {
                            "checked": false,
                            "label": "NORTHWEST"
                    }
                    ],
                    "description": "Wind direction",
                    "include_blank_option": false
            }
        }, {
            "field_type": "dropdown",
            "cid": "c22",
            "pattern": "",
            "required": true,
            "label": "CLOUDS",
            "field_options": {
                    "options": [{
                            "checked": false,
                            "label": "0"
                    }, {
                            "checked": false,
                            "label": "4"
                    }, {
                            "checked": false,
                            "label": "8"
                    }
                    ],
                    "description": "0=totally cloudless\n8 oktas=sky fully clouded",
                    "include_blank_option": false
```

THIS WORK HAS BEEN SUBMITTED TO THE IEEE FOR POSSIBLE PUBLICATION. COPYRIGHT MAY BE TRANSFERRED WITHOUT NOTICE, AFTER WHICH THIS VERSION MAY NO LONGER BE ACCESSIBLE.


```
                    }
          }, {
                    "field_type": "radio",
                    "cid": "c18",
                    "pattern": "",
                    "required": true,
                    "label": "RAINING",
                    "field_options": {
                              "options": [{
                                        "checked": false,
                                        "label": "YES"
                              }, {
                                        "checked": false,
                                        "label": "NO"
                              }
                              ],
                              "description": "seen any rainfall from where you are right now?"
                    }
          }, {
                    "field_type": "devicegps",
                    "cid": "c6",
                    "pattern": "",
                    "required": true,
                    "label": "GPS",
                    "field_options": {}
          }
     ],
     "app": {
          "theatre_address": "https://chwezi.tech/api/act/create/",
          "name": "NJURA",
          "color": "#545465",
          "transport_mode": "POST",
          "uuid": "c102391918-23d9-4ebc-8753-3da5aa5f4cd7",
          "brand_image": "",
          "channel": "LABS",
          "description": "Track weather changes for later analysis"
     }
}
```


Appendix B
Example of a Persona Running on the DNAP Web Histrion: Njura

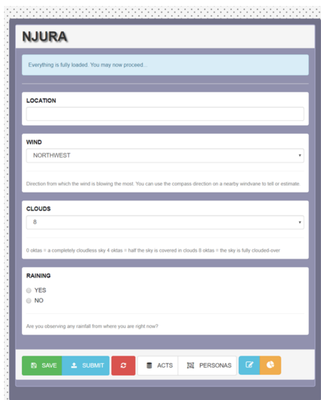

Fig. 11: Web Persona Applet: Njura

The above screenshot is of the same persona defined in **Appendix A**, as rendered on a web based histrion hosted on the reference DNAP theatre[1].

1. https://chwezi.tech/persona/c6e47f79-afd9-4ebc-8753-3da5aa5f4cd7/histrion/

Appendix C
Example of a Persona Running on the DNAP Mobile Histrion: Njura

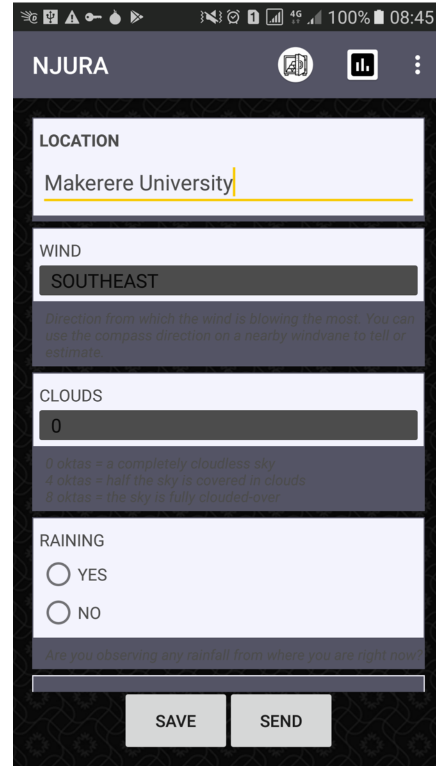

Fig. 12: Mobile Persona Applet: Njura

Appendix D
Example of an Annotated Act on the DNAP Web Histrion's Act Cache

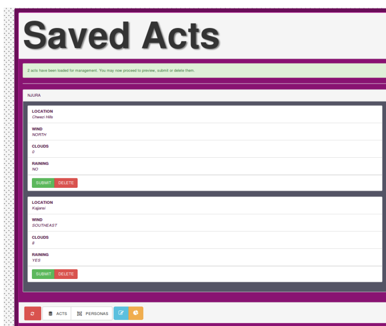

Fig. 13: Web Persona Acts: Njura

THIS WORK HAS BEEN SUBMITTED TO THE IEEE FOR POSSIBLE PUBLICATION. COPYRIGHT MAY BE TRANSFERRED WITHOUT NOTICE, AFTER WHICH THIS VERSION MAY NO LONGER BE ACCESSIBLE.

Appendix E
Example of an annotated act on the DNAP mobile histrion's act cache

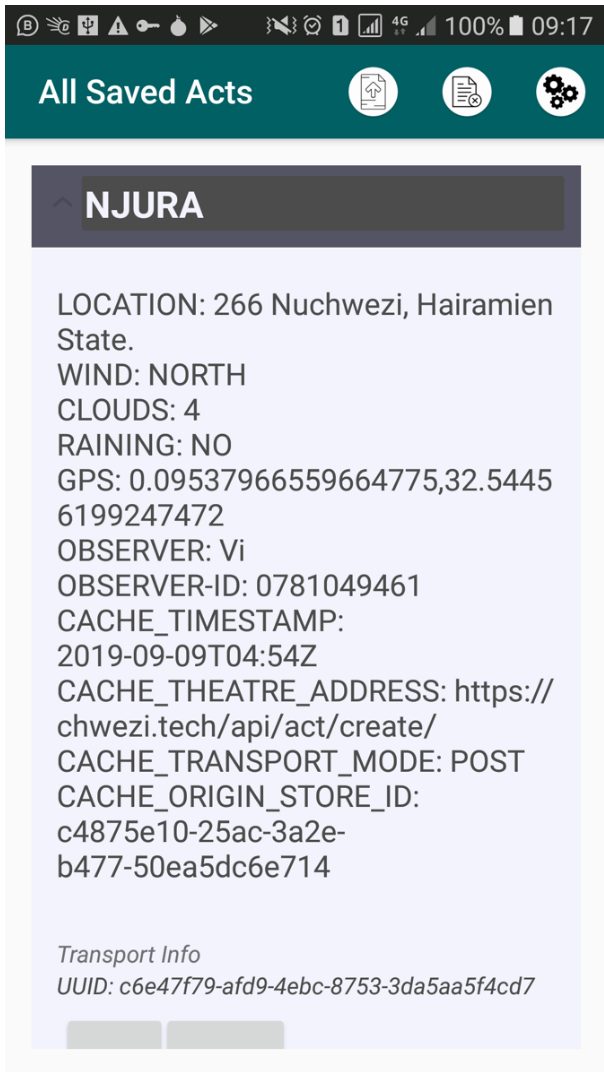

Fig. 14: Mobile Persona Acts: Njura

Appendix F
Examples of an EDA session on DNAP's data diviner

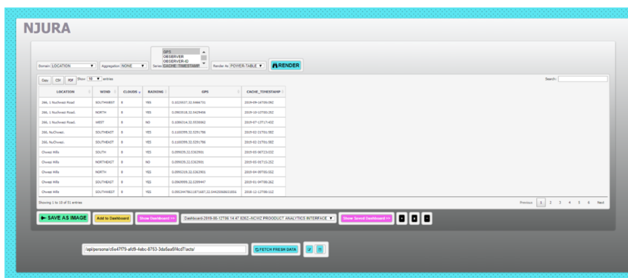

Fig. 15: Diviner Applet: Njura data table

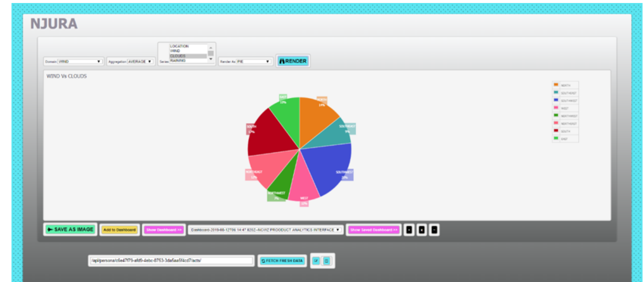

Fig. 16: Diviner Applet: Njura basic categorical data visualization

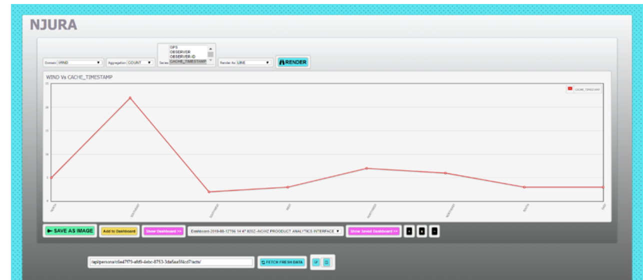

Fig. 17: Diviner Applet: Njura basic time-series data visualization

Appendix G
Examples of an EDA dashboard session on DNAP's data diviner

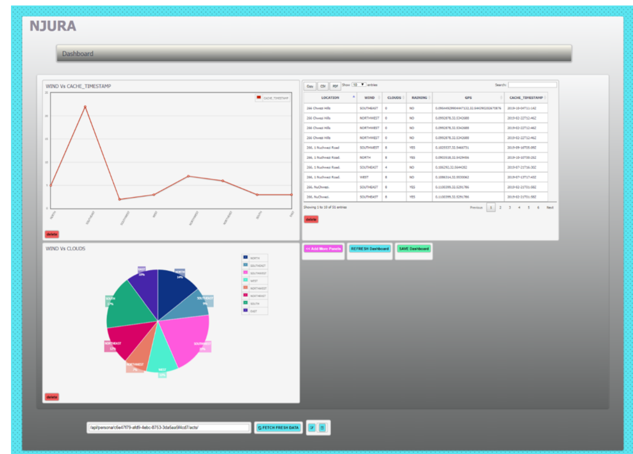

Fig. 18: Diviner Applet: Njura data analysis dashboard

Appendix H
Most Basic Input Output Program (MBIOP)

H.1 .NET Framework (C#)

```
using System; // MBIOP in C#
namespace MBIP___Most_Basic_Input_Output_Program
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.Out.WriteLine("What is your name please?");
```

```
            String name;
            name = Console.In.ReadLine();
            Console.Out.WriteLine(string.Format("Hello {0}",name));
        }
    }
}
```

H.2 Android Platform (Java)

```
using System; // MBIOP in C#
namespace MBIP___Most_Basic_Input_Output_Program
package com.nuchwezi.mbop_android;
import android.app.Activity;
import android.os.Bundle;
import android.text.Editable;
import android.text.TextWatcher;
import android.widget.EditText;
public class MainActivity extends Activity {
        @ Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                EditText eTx = new EditText(this);
                eTx.setHint("What is your name please?");
                setContentView(eTx);
                eTx.addTextChangedListener(new TextWatcher() {
                        @ Override
                        public void beforeTextChanged(CharSequence s, int start,
                                int count, int after) {}

                        @ Override
                        public void onTextChanged(CharSequence s, int start, int
                                before, int count) {
                                  MainActivity.this.setTitle("Hello " + s);
                        }

                        @ Override
                        public void afterTextChanged(Editable s) {}
                });
        }
}
```

It is interesting to note that for the Android platform, the idea of a MBOP can be reduced to a program containing totally no explicitly written Java or any sourcode for that matter, except a simple XML file referred to as the Android-Manifest.xml. Arguably, the MBOP for the Android SOE is that in Appendix I.1.

H.3 DNAP (Cwa Script)

```
{
        "fields": [{
                        "field_type": "text",
                        "cid": "1",
                        "label": "What is your name please?"
                }
        ],
        "app": {
                "name": "Hello",
                "color": "#fff",
                "uuid": "1"
        }
}
```

H.4 Bash

```
read -p "What is your name?" x
echo "Hello $x"
```

H.5 Python

```
a = input("What is your name please? ")
print("Hello %s" % a)
```

H.6 Java SE Platform (Java)

```
import java.util.Scanner;
class Main{
        public static void main(String[]args) {
                System.out.println("What is your name please?");
                Scanner scanner = new Scanner(System.in);
                String tokens[] = scanner.nextLine().split(" ");
                System.out.println("Hello " + String.join(" ", tokens));
                scanner.close();
        }
}
```

APPENDIX I
MOST BASIC OUTPUT PROGRAM (MBOP)

I.1 Android Platform (XML)

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android" package="a.b.c
        ">
    <application android:label="Hello World" />
</manifest>
```

I.2 DNAP (Cwa Script)

```
{
        "app": {
                "name": "Hello World",
                "uuid": "1"
        }
}
```

I.3 Python

```
print("Hello World")
```

I.4 .NET Framework (C#)

```
using System;
class P{
        public static void Main() {
                Console.WriteLine("Hello World");
        }
}
```

I.5 Java SE Platform (Java)

```
class A {
        public static void main(String[]s) {
                System.out.println("Hello World");
        }
}
```