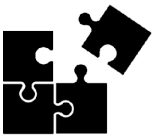


The Open Components Ecosystem

Collaborative Innovation in Bible Technology

https://opencomponents.io

Draft 4 – 20 July 2021

Executive Summary: This paper proposes that the development of Bible technology will benefit from a shift in production model from one where each app is essentially the custom creation of a guild of artisans to one where Bible technology is largely composed of interchangeable components. The paper argues that the benefits of this transition will be similar to that of interchangeable parts manufacturing. Key functionality will be easier to reuse and maintain, more developers from more regions of the world will be able to collaborate together efficiently, and the overall creativity and innovation potential of the global Bible technology network of developers will significantly increase. The design and intentional fostering of a globally collaborative ecosystem of Bible technologies composed of interchangeable parts ("open components") will create a context for nonlinear innovation while maximizing "constructive competition" (and avoiding "destructive competition"). This, in turn, will result in more people able to produce, distribute, and use Bible translations and biblical content in any language, on any technology, and in any format needed.

Introduction

One of the most important factors contributing to the Industrial Revolution is also one that is easily overlooked: the invention of **interchangeable parts manufacturing**. Prior to Eli Whitney's invention in the early 1800s of a means of manufacturing identical components for muskets, production of each tool and machine was essentially the unique work of an artisan.[1] Until manufacturing techniques enabled the mass production of identical components, complicated tools were a one-of-a-kind possession and, as such, could only be repaired or improved by a skilled craftsman.

Once interchangeable parts manufacturing became the standard means of production, several important things happened in the world of manufacturing, including: repair and replacement of parts became much easier, the bar for entry into manufacturing was lowered such that many more engineers and inventors became involved, overall production increased, and many new inventions came into existence that could not otherwise have happened.[2] Interchangeability "changed the industrial revolution and thus changed the world. Every single other invention that came out of the industrial revolution benefited from interchangeability: the steam engine, sewing machines, telegraphs, and more."[3]

The thesis of this paper is that the development of Bible technology will benefit from a similar shift in "manufacturing process" from one where each app is essentially the custom creation of a guild of artisans to one where Bible technology consists largely of interchangeable "open components." The first part of the paper provides a high-level overview of the Open Components Ecosystem. The second part of the paper is oriented toward software developers; it transitions from a theoretical overview of the ideas to the practical realities of Open Components and creating software as part of an ecosystem. The technical details of Open Components—including tutorials, documentation, and code samples—are not included in this paper but are available online: https://opencomponents.io.

Part 1: An Overview of the Open Components Ecosystem

In order to understand the rationale behind the Open Components Ecosystem, we begin by considering the objectives of Bible technology in light of certain strategic considerations. Then we consider different development models for achieving the objectives, which we follow with a description of Open Components. Finally, we consider the advantages and disadvantages of this approach.

Rethinking Bible technology development in the 21st Century

The rapid expansion of the global church in recent decades has significantly changed the context of Bible technology development and use. The changing context indicates an opportunity to take a step back and reconsider Bible technology development. We review certain strategic considerations affecting the development of Bible technology and various approaches to achieving the stated objectives.

---

1 As is the case with many inventions, while Whitney is often credited with the invention of interchangeable parts manufacturing, many others were involved in the development of the invention before it finally went mainstream.

2 As such, the invention of interchangeable parts manufacturing opened up a vast array of adjacent possibilities that could not otherwise have been conceived of. In the same way that the concept of a horse-drawn wagon became possible with the invention of the wheel, mass production of identical tools became possible with the invention of interchangeable parts manufacturing. (See ch. 1 of Johnson, *Good Ideas* for more on adjacent possibilities.)

3 English, "Interchangeable."

Strategic considerations affecting the development of Bible technology

In general terms, Bible technology is intended to accomplish some combination of the following objectives:

- **Production of Bible translations** – Translation drafting, checking, formatting, and publishing (in text, audio, and/or video formats) in thousands of languages, and translation revision as needed over time.
- **Production of Bible content** – Creation of biblical resources integrated with translations of the Bible, including study resources, interlinear translations, lexical and encyclopedic resources, teaching resources, etc.
- **Distribution of Bible translations and content** – Widespread (re)distribution of Bible translations and the biblical content integrated with them.
- **Consumption of Bible translations and content** – Reading, hearing, watching, and deep study of translations of the Bible and the biblical resources integrated with them.

In this section we consider the factors that affect the development of effective Bible technology.

1. How does the complexity of the global context affect the development and use of Bible technology?

The global church is now alive in thousands of people groups, speaking thousands of languages, across many different cultural contexts—and is expanding constantly. This has triggered a corresponding increase in Bible translation activity by a growing number of new translators who are translating the Bible into their own languages. Before this sudden influx of new participants began, developers of Bible technology could make reasonably accurate assumptions about their users. In general, users were more similar than dissimilar, and their technology needs were known and largely uniform.

Now, however, the world has become more complex and the default assumptions about users of Bible technology need to be revisited. The users we know of are often quite diverse in their patterns of technology use, and their Bible technology needs are often in conflict with each other. The needs of one set of users are often the exact opposite of the needs of another set of users—and these are only the needs that we know of. For example, some need an app that is online, synchronously facilitating management of users and roles, making collaboration and sharing easy. Others need an app that never goes online, never identifies the user (even pseudonymously), is not easily discovered, and leaves no trace when uninstalled. Even when technologies are built for different sets of users, it becomes increasingly difficult to respond to their needs when the developers differ from their users in language, culture, time zone, technological reality, and assumptions about how Bible technology should be designed and used.

Some well-known Bible technologies have large numbers of users. The developers of these technologies are to be congratulated for development of apps that are undeniably useful and effective for many users. But the success of such technologies should not obscure the immense number of Bible technology needs that are *not* being met by them. This is not because there is a problem with the technology, nor is it because potential users have not discovered or tried the technology. It is simply that a single app cannot meet the vast array of different technological needs of the global church.

If the global context is shifting from one that is generally uniform and predictable to one with many conflicting and unknown factors (and which impact each other in unpredictable ways), how might this

affect the development of Bible technology?[4] At a minimum, it suggests a greatly increased need for many diverse and innovative Bible technologies that equip the entire global church for the production of Bible translations and biblical content in these languages, as well as distribution and use of the content in different formats and on many different technologies.

2. How does ownership design affect Bible technology?

Have you ever browsed a website or used an app that was obviously created by people who were of a different culture and language? When the content is the manual for a vacuum cleaner, the strange UI and odd font choice are of trivial consequence. But if the content is the translated Scriptures and the technology is the means by which the user seeks to study and understand the truth, these things matter. The medium may not actually *be* the message in this case, but the inadvertent sense of foreignness in the very design of the technology can undercut the user's trust and even undermine the gospel message—without the user consciously realizing it. For some, the ability to trust the technology (and, one assumes, the content it serves) is eroded when updates to the technology are unilaterally made by faceless entities in another part of the world with whom they have no relationship.

The point is not to suggest inadequate effort on the part of current technology providers. The point is that "ownership design" may be a much more important factor in Bible technology going forward than what we have experienced to date.[5] Successfully extending Bible technology into every people group and language may involve equipping technologists in those people groups to meet their own needs using technology that they own and develop, and using processes that they control in order to maximize trust in both the technology they design and the biblical content it makes available.

3. How might the growing technical ability of the church of the global south contribute to meeting their own Bible technology needs?

As the global church continues to expand, the number of skilled technologists who are motivated to employ technology in service to the church is also increasing. Technologists all over the world are finding that existing Bible technologies are good, but that they do not meet certain use cases or criteria unique to their own region of the world. Consequently, they develop their own Bible tools to meet the needs of their own people.

We may benefit from considering an important principle from the domain of relief work. One of the most important considerations regarding the effectiveness of relief and development efforts is that of ensuring that those who are experiencing the need also have the right to meet their need. Thus, if the goal is "ending poverty," it is more effective to ensure the poor have the rights to meet their own needs rather than to rely on a technocratic approach where "experts" are the only ones with the right to "end poverty" for those experiencing it. In the same way, if the goal is "ending Bible technology poverty," it is important

---

[4] The differences between such contexts can be further illustrated by contrasting a Ferrari (complicated) with the flow of traffic (complex). No matter how complicated a vehicle might be, a skilled mechanic can fully deconstruct and reconstruct it exactly. The individual function of each part and its relation to the whole can be known and controlled. Traffic, by contrast, is dispositional (only general predictable) as it is composed of elements (e.g., drivers' attitudes, road conditions, weather, accidents, etc.) that interact in unpredictable ways such that traffic cannot be controlled, only managed. Many of the assumptions and solutions that work in a complicated context tend to hinder effectiveness in a context that has shifted to one that is complex. For more on this contrast, see Snowden and Boone, "Framework."

[5] Ownership design is used here, not in the sense of intentional manipulation of a user's actions for the benefit of a corporation, but in the sense that the design of Bible technology steers behavior *whether the designer is aware of it or not*. If this is true, then it follows that technologists intent on extending Bible technology globally must consider the psychology of ownership together as part of the function and design of the technology.

to consider how equipping those with that need to meet their own need could avoid technocratic approaches to Bible technology development that might inadvertently limit its effectiveness.[6]

As the scenario of technologists around the globe meeting their own Bible technology needs becomes increasingly common, it presents an opportunity to intentionally equip the global church to that end. If there is agreement that anyone should be able to meet their own Bible technology needs, it suggests the value of a Bible technology ecosystem that predisposes and invites collaboration and sharing, lest the result be a fragmented (albeit well-intended) approach.

4. How might we cultivate optimal conditions for innovation in Bible technology?

One of the most essential points about innovation is easily overlooked: it cannot be planned or predicted. If innovation could be predicted in advance, "it wouldn't be innovation. If the invention is already known, it does not need to be invented. What this means is that the path of innovation is always a surprise. Who will be the innovator is equally a surprise."[7] The best way to predispose innovation is to cultivate a context that provides favorable conditions for it. Attempting to innovate without assessing and optimizing the conditions for innovation is like planting a field without first cultivating and fertilizing it—it might produce some results, but the true potential may not be realized. In this regard, "the conditions facing entrepreneurs determine how much innovation happens."[8] We propose these factors that help optimize the conditions for innovation:

- **Lower the bar for entry** – Make it as easy as possible for new contributors to Bible technology to get started. "The higher the costs of entry, the fewer will be the ideas that can vie for attention ... This is why lowering the cost of entry is so critically important for anything that is critically important."[9]
- **Seed the technology field** – Provide as much technology as possible in reusable formats, so as to minimize unnecessary redundancy and provide maximum functionality to the greatest number of contributors in the least amount of time possible. "The key prediction is that innovation is usually faster the more technology one already has."[10]
- **Maximize the flow of ideas** – Most innovation tends to happen through novel recombinations of existing ideas and technologies. It follows, then, that decreasing the friction in the flow of ideas and reuse of technology will increase the innovation potential of the community involved. The most effective "patterns of innovation … do best in open environments where ideas flow in unregulated channels. In more controlled environments, where the natural movement of ideas is tightly restrained, they suffocate."[11]

5. How might it be possible to "disruptively innovate" without unintentionally "destructively competing" against other apps (or developers)?

In software development, collaboration is frequently in tension with innovation. Furthermore, the prioritization of one value tends to produce a diminishing effect on the other. If collaboration is valued

---

6 Easterly argues that the history of relief and material development has generally followed the technocratic idea of "focusing on development instead of on the rights of those to be developed" (Easterly, *Tyranny*, p. 49). He notes the emphasis on material development has focused on "'what must we do to end global poverty?' while neglecting the unequal rights for blacks and whites and the unequal rights in the West and the Rest" (p. 95).

7 Easterly, *Tyranny*, p. 299.

8 Wu, *The Master Switch,* p. 146.

9 Wu, *The Master Switch,* p. 122.

10 Easterly, *Tyranny,* p. 284.

11 Johnson, *Good Ideas,* § Conclusion.

higher, then innovation may be curtailed and constrained to the point that it does not disrupt any of the collaborators. If innovation is valued higher, then collaboration becomes endangered by disruptive innovations that can easily turn into a detrimental form of competition—particularly when users begin migrating to the app that has the innovative breakthrough.

To be clear, we do not consider competition to be intrinsically detrimental. To the contrary, constructive competition can be an effective way of increasing innovation and solving intractable problems. An example of this form of "constructive competition" might be a hackathon where several developer teams compete to design the most effective solution to a shared technological challenge and where each is able to incorporate the winning solution into their technology.[12]

When developer teams are (or perceive themselves to be) in competition for scarce funding, it can lead to a less altruistic form of competition. In such contexts, teams may be less inclined to share their technological innovations with others, in order to maintain greater competitive advantage. While there is a time and place for this kind of "destructive competition," we suggest that it is less than ideal for the development of technology pertaining to the translation, distribution, and use of Bible translations in thousands of languages.

A Way Forward

The proposed solution to the tension we have considered above is the intentional and collaborative development of an open ecosystem of Bible technology development. This approach acknowledges the complexity of the global Bible technology need and meets it with a modular approach that can be easily reconfigured in any number of ways, as needed. It agrees with the assertion that "large-scale social change [e.g., translating and distributing the Bible in every language and people group] comes from better cross-sector coordination rather than from the isolated intervention of individual organizations."[13] It acknowledges that those who are best able to meet the Bible technology needs are generally those closest to the needs, especially when they are connected to others working together with them as part of a collaborative network. It provides an optimal context for innovation and enables contributors to innovate together without destructively competing against one another. In order to better understand the Ecosystem approach to software development, we will first compare it to a Centralized approach and a Fragmented approach.

Three approaches to development of Bible technology

What approaches to software development might we use to accomplish the objectives of Bible technology?[14] There are at least three approaches to software development by which the objectives of Bible technology could potentially be achieved. It may be helpful to consider each of these approaches in light of some of the factors mentioned above, including:

---

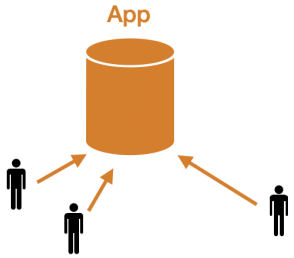
[12] A more organic example is the progression of innovations in Bible translation editor interfaces that have emerged as developers of early Open Components have learned from the work of others in order to develop new technologies. An early component, `usfm-js` encouraged a new way of parsing USFM in `usfm-grammar`, both of which have spurred on `Proskomma`, converting USFM to a graph.

[13] Kania and Kramer, "Collective Impact." The authors go on to observe that "the nonprofit sector most frequently operates using an approach that we call isolated impact. It is an approach oriented toward finding and funding a solution embodied within a single organization … there is scant evidence that isolated initiatives are the best way to solve many social problems in today's complex and interdependent world. No single organization is responsible for any major social problem, nor can any single organization cure it."

[14] The scope of this paper is limited to software development, though many of the same principles described herein could apply to Bible technology hardware, e.g., dedicated Scripture audio players, streaming devices, etc.

- **Ownership**: decentralized (multiple apps owned by multiple developer teams to meet multiple needs) vs. centralized (single app owned by a single developer team).
- **Collaboration**: *organic* (immediate, as needed, relatively easy) vs. *formal* (possible, but requires certain official documentation and overhead).
- **Innovation**: *nonlinear* (unconstrained, potentially divergent, "outside the box") vs. *limited* (constrained to avoid disruptive innovations, linear and incremental, "inside the box").
- **Adaptability**: *adaptable by anyone* (unlimited, anyone with the technical ability can adapt the technology to meet their needs, multiple versions/variations exist) vs. *managed adaptability* (very limited adaptation, only possible for the owner of the technology, a limited number of different versions exist).
- **Portability**: *optimal reusability* (technology elements can be reused across multiple different applications) vs. *limited reusability* (technology and functionality generally exists only in the original application for which it was developed).

Centralized (single app)



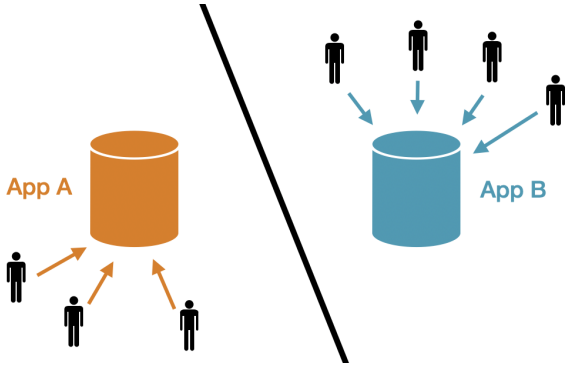
Centralized development: a single app attempting to meet all user needs

In this approach, all development is focused on a single application, with the goal of reducing redundancy and streamlining development. While this approach may produce short-term gains, it has the adverse effect of greatly reducing the number of technologists and innovators who can meaningfully contribute to the expansion of Bible technology.[15] The number of Bible technology needs of the global church is immense and growing—expanding over thousands of languages and dozens of technologies. The growing number of needs, together with the increasing number of Bible technologists and software developers already working globally to address those needs, suggests that the notion of a single application being able to accomplish everything needed by the global church is becoming decreasingly

[15] Wu notes that there is "an undeniable efficiency" that attends to centralized development models. He notes that "what such well-oiled machines do not do so well, however, is initiate the sort of *creative destruction that revolutionizes industries and ultimately multiplies productivity and value*. And where information is the ultimate commodity, the multiplier effect is incalculably great" (*The Master Switch*, 195, emphasis added). The need to multiply productivity and value regarding the translation, checking, distribution and use of the ultimate information—the Scriptures—in every language suggests that a centralized approach could inadvertently sacrifice long-term innovation breakthroughs for short-term gains in efficiency.

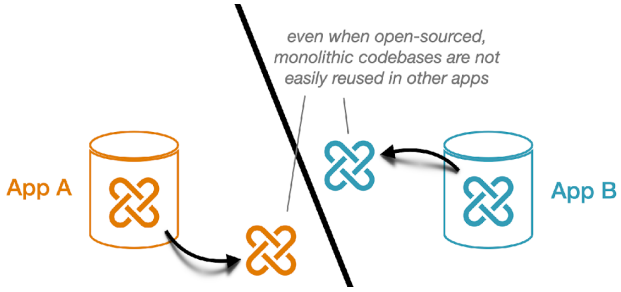
plausible.[16] As such, we will focus our considerations in this paper on the remaining two development models: Fragmented and Ecosystem.

Fragmented system (multiple apps)



Fragmented development: when multiple players build centralized apps

In the fragmented approach to software development, independent apps proliferate in competition for desirable functionality and largest user base. This is the de facto standard across most software development domains, particularly where increasing market share for the company via direct competition is desirable. This is also the approach that best represents the current Bible technology environment, even though such competition is generally considered detrimental to the missional objective.



Fragmented development: suboptimal collaboration and innovation

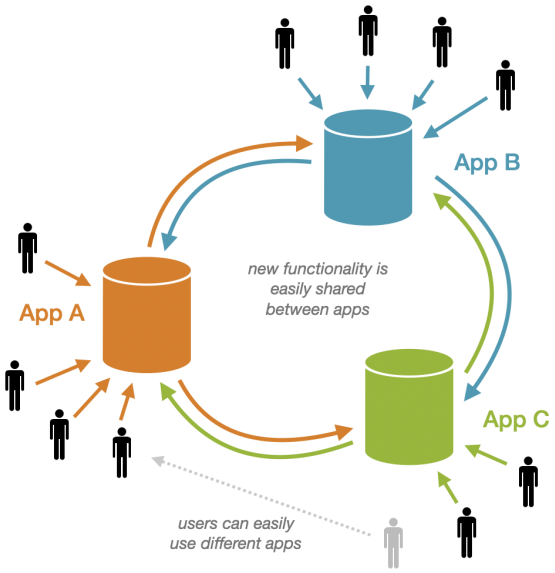
As it is, many Bible apps accomplish generally similar functions, but even when developers desire to collaborate together across organizational boundaries and codebases, they lack an effective means of doing so (see diagram above). Furthermore, to the extent that technologies are not able to be easily reused (whether due to the structure of the codebase or the license or both), each app in the fragmented system eventually needs to recreate the functionality of every other app in order to not fall behind in terms of their competitive advantage or their ability to serve their userbase. The result is widespread increase of unnecessary redundancy and cost, and a diminishing of collaboration and innovation.[17]

---

[16] The number of Bible technology development teams is expanding globally at an exciting pace. A partial list of Bible technologies under development by teams around the world who are known to the authors of this paper include: the Autographa Bible translation suite (by Bridge Connectivity Solutions (BCS) in India), Vachan Online (Bible study, by BCS in India), FABIST (Bible translation, by 222 Ministries for Iranian languages), V-Draft (Bible translation, by GeCraft in Central Asia), VietBible (Bible study, in Vietnam), SABDA (Bible study, in Indonesia), Bible Study App (GeCraft in Central Asia).

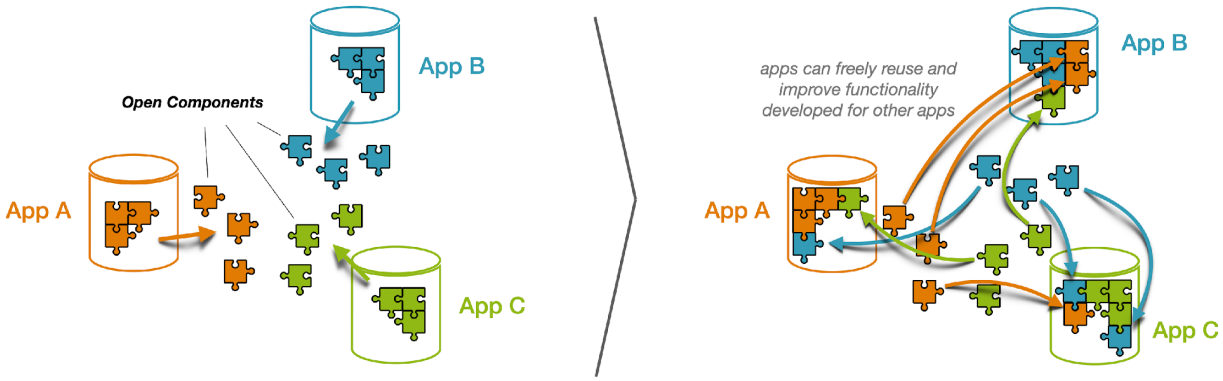
[17] In a fragmented system, it is not uncommon for the larger incumbents to attempt to return things to a centralized system with only one app permitted. The rationale for doing so usually pertains to reducing redundancy and cost, as well as streamlining development efforts. These arguments are not without merit, as it is true that, compared to a fragmented system of multiple independent apps, there is less waste. Instead of “ten companies competing to develop a better

Ecosystem (collaborative innovation)



Ecosystem: when multiple players collaboratively cultivate an efficient environment for rapid innovation

We argue in this paper for a decentralized model of Bible technology development. This approach looks similar in some regards to the fragmented approach, but with one important difference: Bible technology developers are not merely focused on the development of their own unique app, but **they build in such a way as to also nurture the health of the Bible technology ecosystem of which they are a part**.



Ecosystem: Comprised of open-source components intended for reuse

By creating open source, modular software that is reusable in other applications, developers all over the world can innovate more efficiently in a massively collaborative model that maximizes the effectiveness of Bible technology while maximizing constructive competition (and minimizing destructive competition).

---

telephone—reinventing the wheel, as it were, every time—society's resources can be synchronized in their pursuit of the common goal. There is no duplication of research, with many laboratories chasing the same invention" (Wu, *The Master Switch*, 110). But centralized systems are notoriously restricted in what they can innovate. Wu states that they are constrained by a serious "genetic flaw" in that "they cannot originate technologies that might, by the remotest possibility, threaten the … system." Centralized systems are "practically *restricted to sustaining inventions; disruptive technologies, those that might even cast a shadow of uncertainty over the business model, [are] simply out of the question*" (Wu, *The Master Switch*, 107, emphasis added). Wu notes that such practical inventions as the answering machine had been invented early on by Bell Labs, but were terminated by executives who feared the slightest disruption to AT&T's business model. Many such useful inventions only came to market decades later, after the monopoly had been dissolved.

The result of this decentralized and collaborative technology development model is the Open Components Ecosystem.

Understanding Open Components

Open Components meet these three technology-design criteria: Extensible, Portable, and Open source.

- **Extensible** architecture is designed to enable expansion of functionality by incorporating code developed by a third party, most commonly in the form of a plugin. In this regard, other developers are invited to bring their code and "play in the sandbox" provided by the base technology.
- **Portable** architecture is designed to go in the other direction, by providing functionality that can be incorporated into other technologies, commonly in the form of libraries or simple apps. Developers are invited to "create their own sandbox" by incorporating the portable components into their own apps.
- **Open source** technology consists of source code that is made available under an open license, such that developers can repurpose, expand, improve, redistribute, and otherwise reuse the technology without restriction.[18]

By considering the interplay of these factors, we can see how different technologies and tools provide different functionality, as in the diagram below:

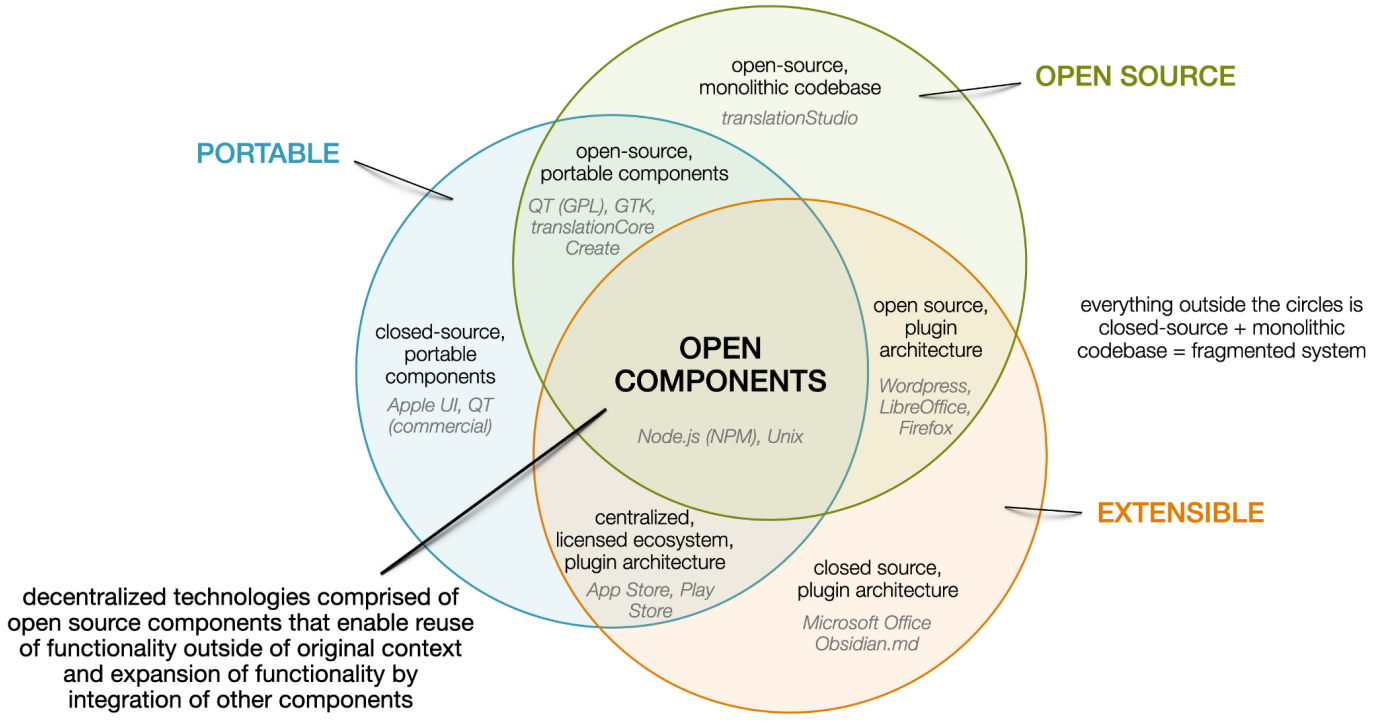


Diagram: The Open Components Ecosystem

[18] The connection between open-source code and the potential for nonlinear innovation is well documented. One of the classic examples is the 10-day MATLAB programming conference, during which contestants are required to submit algorithms to solve a problem. The contest is gamified and all source code from each contestant is open for others to see, reuse, and tweak in order to create a better solution to the problem. Howe notes that "the extraordinary aspect of MATLAB … in which *all intellectual property is thrown into the public square to be used and reused at will* turns out to be an insanely efficient method of problem solving … On average … the best algorithm at the end of the contest period exceeds the best algorithm from day one *by a magnitude of one thousand*" (Howe, *Crowdsourcing*, emphasis added).

Open Components exist in the overlap of all three of these criteria: it is the sum of decentralized technologies comprised of open-source components (Open) that enable use of functionality outside of their original context (Portable), as well as the expansion of functionality by integration of other components (Extensible). A "component" refers to a portable code module that provides separation of concerns around a particular problem set (e.g., a particular UI element, a headless function like a tokenizer, etc.). An "open component" is one that has been released under an open-source license that maximizes reusability (e.g., MIT, AGPL). The "open components ecosystem" is composed of open-source components, the technologies that integrate the components together to solve a diverse range of problems, and the people involved with the technology.[19] In this ecosystem, new technology is **designed and built with two concurrent perspectives in view**: meeting the immediate need (the "application" perspective) *and* doing so by means of open components that can be reused by other technologies in the ecosystem to meet the same need (the "ecosystem" perspective).

Open Components are **more than modular software**. Many software codebases are designed to be modular for ease of maintenance, but the modules themselves are not made available to others, nor is the codebase extensible by third parties.[20] Portable and extensible code is necessarily modular, but not all modular code is portable and extensible. The Open Components Ecosystem includes technologies composed of modular codebases that are also designed to be improved by reuse of other components (extensible) and to improve other technologies by permitting the reuse of the new components created (portable).

Open Components are **more than open-source software**. Releasing source code under an open license does not thereby create an open ecosystem. Open-licensed source code is a necessary element of an open ecosystem, but it is equally important that the code itself be designed for the ecosystem. Code that is monolithic—whether closed source or open source—may be created more quickly, but is beneficial only for the application for which it was designed.[21] Even when made available under an open license, monolithic code can be difficult to understand and cumbersome to improve or reuse. Other developers may be able to reuse it by forking and modifying the code to meet their needs (e.g., by adding new features and fixing bugs) and they may contribute these improvements back to the existing project. It is when both projects need to diverge to meet their respective use cases that the limitations of a monolithic, open-source application becomes a hindrance. While they may share as much as 99% of the same code, future bug fixes and patches to one project are so tedious to apply to the other that they

---

19 The Open Components Ecosystem is an example of *composability*, a system design principle that deals with the inter-relationships of components. A highly composable system "provides components that can be selected and assembled in various combinations to satisfy specific user requirements." In information systems, the essential features that make a component composable are that it be self-contained (modular, in that it can be deployed independently and may cooperate with other components, but dependent components are replaceable) and stateless (it treats each request as an independent transaction, unrelated to any previous request). See "Composability" (https://en.wikipedia.org/wiki/Composability).

20 Modular code contrasts with monolithic code in some important ways. Slootweg, in "Monolithic", notes that modular code "rarely needs to be changed—once it is written, has a well-defined set of events and methods, and is free of bugs, it no longer needs to change. If an application wants to start using the data differently, it doesn't require changes in the component; the data is still of the same format, and the application can simply process it differently." Furthermore, the cost of a modular codebase may be greater at the outset, but it is "a one-time cost. After your first project, you already know where to find most modules you need, and building on a modular framework will not be more expensive than building on a monolithic one." In addition, modular code is generally easier to understand and maintain, and it makes it relatively easy to swap out components without needing to significantly refactor the codebase.

21 Many—though not all—technologies in a fragmented system are composed of monolithic codebases, meaning the code is tightly coupled (highly interdependent on other code) and makes a lot of assumptions about how the code interacts with each other. For more on monolithic code and coupling, see Lin, "Monolithic" and Beard, "Microservice Architecture."

usually don't happen—in spite of good intentions. For this reason, many times developers avoid collaborating on monolithic applications, even when they are open source.

Open Components are **not limited to User Interface elements**. Open Components can provide frontend, backend, data processing, server-side, client-side, etc. functionality. As we shall see below, backend components that do not include a UI element benefit greatly from the provision of a visual sandbox whereby the functionality of the component can be explored.

Advantages

The Open Components Ecosystem provides several important **advantages** over traditional software development models. These advantages exist at the level of the writing and deployment of software code (code-level advantages), and also at the level of the ecosystem as a whole (ecosystem-level advantages).

Code-level Advantages

- **Easier to Develop** – The separation of concerns in this approach enables a developer to focus on a single functionality in a given component, making it easier to develop.[22]
- **Easier to Prototype** – In a robust ecosystem, an application can be rapidly assembled from existing components in order to provide a framework for development of new component prototypes. Given the ease of enabling/disabling components in such a framework, multiple prototypes can be tested independently or in parallel. This ability to focus on prototypes leads to more rapid innovation, as there is no need to build functionality provided by existing components.
- **Easier to Test** – In the same way, testing a component is easier since the separation of concerns enables the developer to focus on the component and more rapidly iterate through the build ➔ measure ➔ learn cycle.[23]
- **Easier to Reuse** – With good documentation and input/output design, a component can be easily reused in other applications. This provides immediate functionality with virtually no redundant development of existing functionality.
- **Easier to Improve** – A component can be improved more easily, as its singular focus and standalone design make understanding and improving the code easier. As use of the component extends across multiple applications, more developers are able to identify innovative modifications and suggest (or write) improvements to the code that increase its functionality.
- **Easier to Maintain** – In the same way, as a component is used in other applications, more developers are able to identify bugs and submit patches that fix them to the component's maintainer. Consequently, long-term maintenance of an application's codebase is cheaper, as the cost of maintaining it is shared across the ecosystem. Effectively, other developers participate in the maintenance of other applications by maintaining the components used in those applications. *Note: this does incur a review cost for application developers, as they will need to invest time learning how functionality of updated components has changed, and determining if/when/how to integrate them.*
- **Easier to Sustain** – The life of a component can easily outlive the usefulness of the application for which it was designed. This is especially important in rapidly changing domains such as Computer-Assisted Language Technology, where applications tend to become obsolete much sooner than the key functionality they were designed to provide. The longevity of technology

[22] As mentioned below in "disadvantages," the design of components may require more time, at least initially.

[23] For more on lean development, see Ries, *Lean Startup.*

implemented as an open component is decoupled from the relatively short lifespan of a single app, making it easier to sustain its usefulness in the ecosystem longer than would otherwise be possible.

Ecosystem-level Advantages

- **Enables nonlinear innovation without destructive competition** – By virtue of the fact that all innovations are available under open-source licenses and also designed as components that can be reused in other applications, innovations of the most disruptive nature do not disrupt the other contributors to the ecosystem. If the breakthrough invention of another developer is useful to the users of another application in the ecosystem, it can be included in that app with relatively little effort. The net result is a development context strongly predisposed both toward nonlinear innovation and also toward constructive collaboration.
- **Enables more cost-effective technology development** – As mentioned above, the facility of including components created by one development team into the application of another greatly reduces the amount of resources wasted on recreating existing technology.
- **Enables effective participation of new additions to the global network of Bible technology developers** – The ecosystem is not only open to new participants, but actively welcomes them, as the more contributors to the network of ideas, the greater the innovation capacity of the ecosystem as a whole. Furthermore, as more technology developers join the ecosystem from within church networks involved in Bible translation and theological formation, the more effective the entire network becomes at extending useful Bible technology into every people group and language.
- **Enables effective participation of part-time and volunteer developers** – One of the greatest challenges facing developers who are not able to supply many hours of time to the development of Bible technology is that they spend much of the time they do have merely getting up to speed on everything that changed in the source code since their last opportunity to help. In the Open Components Ecosystem, a part-time developer can meaningfully serve the entire ecosystem by developing and maintaining even a single component that other applications depend on.
- **Enables reuse of functionality across different Bible technology domains** – Often, different applications are focused on either the production of Bible translations or the distribution of Bible translations. There is, however, some degree of overlap between the technological needs of both kinds of applications (e.g., tokenization of text strings in different languages, display of interlinear translations, etc.). When implemented as an open component, the same functional need can be met in different technology domains. Cross-domain connections are highly fertile sources of innovation and creativity—the more it is catalyzed, the more nonlinear and divergent innovations are possible.
- **Enables for-profit companies to participate in the ecosystem** – In general, the world of Bible technology is split between the non-profit sector (often intending to serve many (or all) of the 7,000+ minority languages spoken in the world) and the for-profit sector (usually focused on relatively few major languages). The difference in focus, and the growing trend in non-profit Bible technology toward open-source technology, can make it difficult for those in different sectors to collaborate. The Open Components Ecosystem provides a meaningful way for developers from both sectors to collaborate together on technology that serves both.[24]

[24] Kania and Kramer note the importance of meaningful cooperation between nonprofit players and those outside the nonprofit sector for solving complex social problems: "Social problems arise from the interplay of governmental and commercial activities, not only from the behavior of social sector organizations. As a result, *complex problems can be solved only by cross-sector coalitions that engage those outside the nonprofit sector*" ("Collective Impact," emphasis added).

Disadvantages

The Open Components Ecosystem is not without some **disadvantages**, including the following:

- **Requires developing with a view to serving the ecosystem** – Developers in the Open Components Ecosystem need to adjust their development practice to take into account the parameters and conventions of the ecosystem (i.e., Portability, Extensibility, Open-source), instead of merely focusing on their application and meeting the immediate need. This can prove to be a challenging adjustment for developers accustomed to only considering the context of their own application and for applications that are composed of monolithic code that is not available under an open-source license.
- **May require more time to develop component libraries than would otherwise be needed** – Writing code for a traditional application is often less time-intensive than writing code that can be used in many applications in an ecosystem. Developing technologies that serve the ecosystem requires a short-term time investment (slowing down to some extent the provision of the functionality to one's own application) for the long-term increase of value to the ecosystem (providing the functionality as a component that all other applications in the ecosystem can also use). *Note: observations suggest that as developers become accustomed to creating components instead of traditional code, their efficiency increases to the extent that the time difference may become negligible in some cases and may even result in greater efficiency in others.*
- **Generally requires use of the same programming language** – The Open Components Ecosystem is effective to the extent that applications are built using the same programming language. For example, a component built in C#/.NET will not be immediately useful in an application written in JavaScript, and vice versa. There are three important things to note about this observation. First, this does not mean that the Open Components Ecosystem is limited to a single programming language—it is not. Second, through the use of cross-compiling and transpiling, components written in one programming language may be made usable in other technology stacks (e.g., Rust ➔ JavaScript; many programming languages ➔ Web Assembly). Third, even when components cannot be directly reused in another technology, standalone components that maintain a separation of concerns can be relatively easy to port across different programming languages/environments. Thus, the same component can be recreated in more than one programming language, improving the portability of component functionality across the entire ecosystem.

Considerations for Funding

The Open Components Ecosystem provides a robust and cost-effective context for development of Bible technology. First, it **minimizes the amount of funds wasted** on recreating functionality that already exists in other applications. By creating functionality as components that can be reused in multiple technologies, developer teams spend less time recreating what already exists. Second, the Open Components Ecosystem **maximizes use of resources to solve new problems and invent new technologies**. By creating a context for global collaboration, a growing network of technology developers can coordinate together with minimal friction to explore new opportunities and innovate toward meeting the immense Bible technology needs of the global church.

In addition to these, one of the greatest advantages of the Open Components Ecosystem is that it helps **resolve the tension between developer teams as they pursue funding** for Bible technology. The traditional model for funding technology tends to incentivize either a centralized (single app) or fragmented (multiple apps) technology-development model. Effectively, technology developers compete for funding by showcasing the functionality of their own applications, often with at least implicit contrast to other "competitors." This makes meaningful collaboration and sharing between developer teams

complicated and conflicted, to put it mildly. By contrast, the Open Components Ecosystem can completely alter the funding landscape and avoid these conflicts. However, this is only possible to the extent that those who fund Bible technology understand how the ecosystem works and adapt their resourcing of technology development accordingly. To that end, we suggest the following considerations.

Understand the ecosystem

Funders are often accustomed to funding an organization to build a particular application. It is important that those who resource the development of technology understand the highly interconnected nature of the Open Components Ecosystem and that *functionality is shared across many applications.* Consequently, funders will benefit from understanding the functionality that they are funding, not merely the application where the functionality is first being developed.

Not all contributors to the Open Components Ecosystem will necessarily have an application that showcases their work. Instead, some developer teams may create components that are in use across several applications built by other developer teams. It is important that funders recognize these dynamics and understand how funding a technology ecosystem is different from merely funding development of an application. As funders recognize the immense value of funding those working behind the scenes to create the technology on which the whole ecosystem depends, the entire ecosystem will be strengthened.

Incentivize the creation of the ecosystem

Closely tied to understanding the Open Components Ecosystem is the opportunity funders have to incentivize its creation. To the extent that funders do so, they will be able to rapidly expand the network of participants and increase the innovative potential of the ecosystem as a whole. Practically, this could happen in at least the following ways:

- **Fund the "componentization" of existing technologies** – Some functionality that is needed by the entire ecosystem currently exists in only one application. Funders could greatly strengthen the ecosystem by providing resources to adapt the technology into an Open Component that would then be useful across the entire ecosystem.
- **Fund the creation of new components** – Funders can encourage developer teams to build new technology as Open Components that benefit the entire ecosystem, rather than benefiting merely their own application.

Resource the leadership and infrastructure of the ecosystem

As an intentional collaboration between many different participants, the success of the Open Components Ecosystem is directly correlated with adequate communication and coordination between the participants. To some extent, a supporting infrastructure is necessary, providing such things as a repository of available components, tutorials, documentation for participation, communications platforms, hackathons, joint project management, etc. Without this, the expectation of effective collaboration is unrealistic.[25] Thus, we propose that this need be acknowledged and that appropriate resources be allocated to provide effective leadership and infrastructure, so as to mitigate risk and provide for effective collaboration in the ecosystem.[26]

---

[25] Good intent all too easily falls by the wayside as each organization deals with demanding timelines to meet disparate goals and objectives—no matter how similar they seem to be to another. "The expectation that collaboration can occur without a supporting infrastructure is one of the most frequent reasons why it fails" (Kania and Kramer, "Collective Impact").

[26] This is a similar change to what is required for funding Collective Impact Alliances. "This requires a fundamental change in how funders see their role, from funding organizations to leading a long-term process of social change. *It is no longer*

Summary

In the first part of this paper, we have proposed a transition in the development of Bible technology from a fragmented environment to a globally collaborative ecosystem of Bible technologies composed of interchangeable parts (Open Components). We have suggested that the net result of this transition will be similar to that of interchangeable parts manufacturing: reuse (and maintenance) of key functionality will be easier, more developers from more regions of the world will be able to collaborate together efficiently, and the overall creativity and innovation potential of the global Bible technology network of developers will significantly increase—while avoiding destructive competition. This, in turn, will result in more people able to produce, distribute, and use Bible translations and biblical content in any language, on any technology, and in any format needed. We now turn to the second part of the paper in which we consider the practical implementation of the Open Components Ecosystem.

Part 2: Toward an Open Components Ecosystem for Bible Technology

The remainder of this paper is written with a view to addressing certain technical matters that may be of interest to software developers. Tutorials, templates, and code samples are available online at https://opencomponents.io.

The Open Components Ecosystem is already in existence, with several entities collaborating together to create reusable components pertaining to production and use of Bible translations. In this section, we will take a look at the state of the ecosystem today and then consider the principles and practices ("simple rules") that lead to a healthy Bible technology ecosystem.

Open Components in Real Life

Here we consider examples of open components from industry, as well as from the early stages of the Open Components Ecosystem itself.

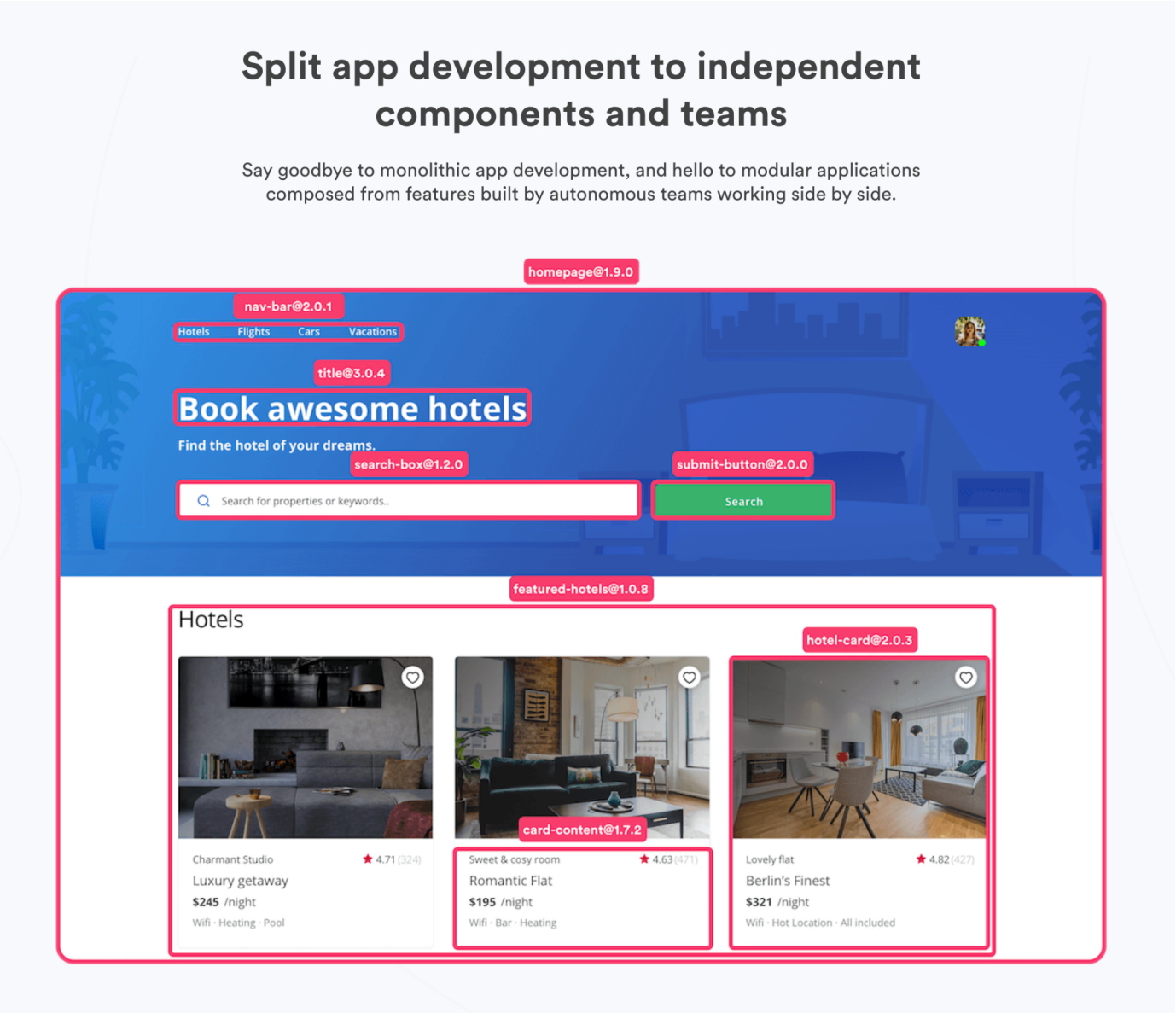
Examples from Industry

A well-executed example of collaborative component development in the wider software development industry is https://bit.dev. This site resembles NPM and makes it easy for software developers to describe and share their JavaScript components.[27] A screenshot from their website gives a visual hierarchy of components that were used to make a popular website. The site provides tutorials for component development with a web-centric focus, including popular frameworks such as React, Vue, Angular, but also Node.js for downloadable applications. Although there is no coordinated market segment that is targeted by bit.dev, it is a good example of how generic components can be reused for multiple segments. The Open Components Ecosystem has an advantage in that both funders and implementing organizations want to actively collaborate with one another in a narrower market (Bible technology).

---

enough to fund an innovative solution created by a single nonprofit or to build that organization's capacity. Instead, funders must help create and sustain the collective processes, measurement reporting systems, and community leadership that enable cross-sector coalitions to arise and thrive" (Kania and Kramer, "Collective Impact").

[27] NPM is an online repository for the publishing of open-source Node.js projects (https://www.npmjs.com/).



Examples from the Open Components Ecosystem

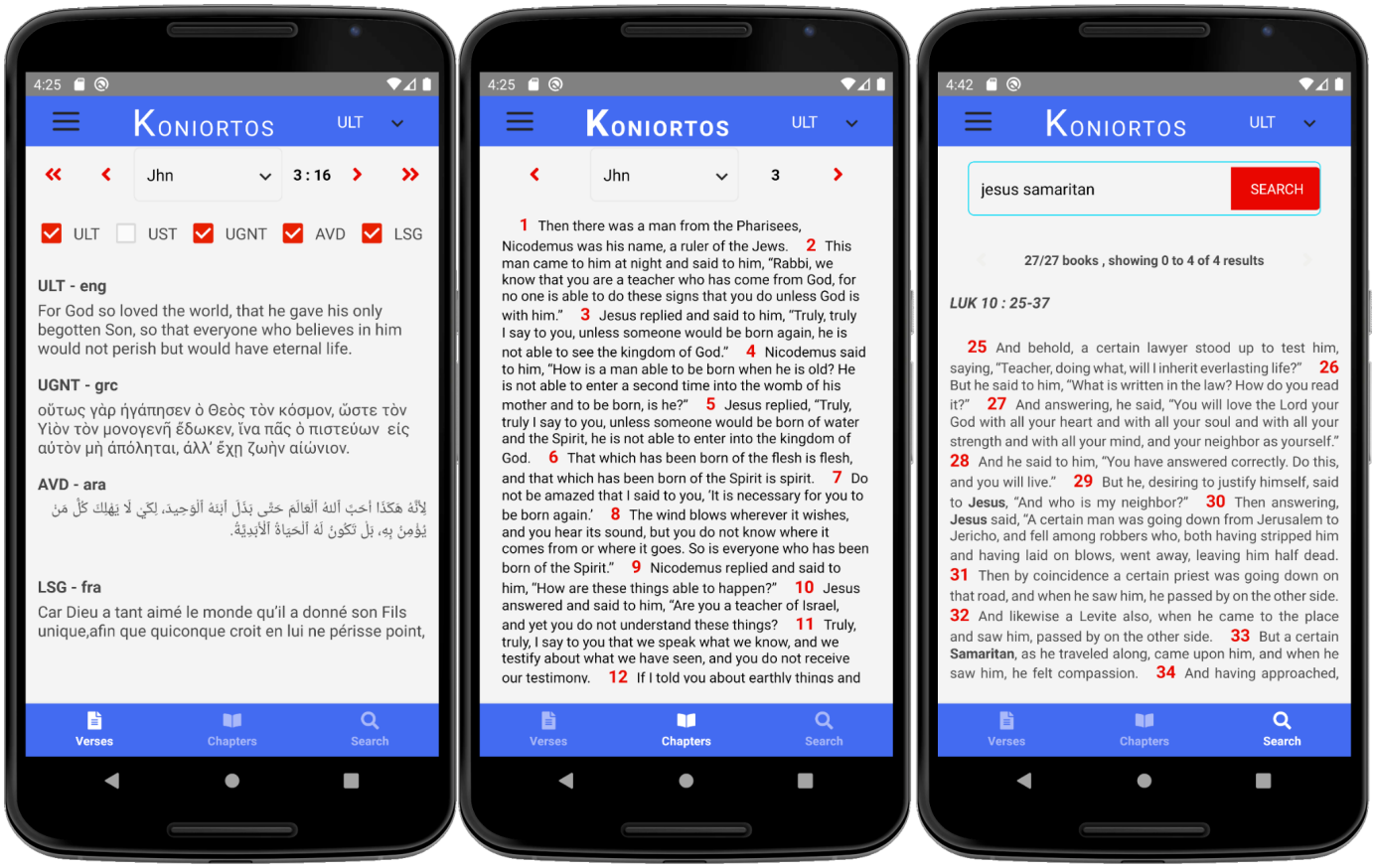
As of the writing of this paper, the Open Components Ecosystem is in its early stages and growing rapidly. Several organizations and developer teams are working to assemble components that provide essential Bible technology functionality. Most of these Open Components are React Component Libraries, often intended for use in both web applications and JavaScript-based frameworks (e.g., Electron).[28] The following abbreviated list gives a sense of the breadth of the ecosystem at this early stage, as well as some of the core functionality provided as components.

[28] Components built as React Component Libraries provide significant advantages, notably the number of platforms on which they can be used (including web, desktop, and mobile applications) and the ease with which the components can be integrated into an online "sandbox" for in-browser testing of use cases. Note: several organizations have independently selected this stack for Bible technology development.

Component	Type	Team
USFM Editor – *WYSIWYG Bible editor interface*	React	BCS
Preference Settings – *persist user preferences*	React	BCS
Grapha UI – *UI components*	React	BCS
Grapha Sync – *drag-and-drop UI for file-syncing*	React	BCS
Interlinear Editor – *WYSIWYG alignment editor*	React	Clear.Bible
Bible Reference Picker – *Book-Chapter-Verse navigation*	React	GeCraft
Scripture ePub – *export to EPUB*	Javascript	MVH Solutions
Proskomma – *Scripture runtime engine*	Javascript	MVH Solutions
Graphite – *smart font system for complex scripts*	C++	SIL
WordMap – *multilingual word alignment prediction*	Javascript	unfoldingWord
String punctuation tokenizer – *string token classifier*	Javascript	unfoldingWord
Headless Workspace Layout – *UI component layout*	React	unfoldingWord
Content Validator – *USFM, Markdown, TSV*	Javascript	unfoldingWord

Note: The current list of components and developer teams is online at https://opencomponents.io.

Recently, a software development intern was able to create a feature-rich mobile Bible app prototype in less than 6 weeks as a result of existing components. Having no prior experience with either the Bible or software development, Imad developed Koniortos based on the Proskomma JavaScript component and React Native. Because Proskomma handles all the complexity of processing USFM text and then presents it to the developer as a graph, there was very little development time spent on a thorny problem that has already been solved.



Principles for a collaborative Bible technology ecosystem

In the section "Rethinking Bible technology development in the 21st Century" we suggested that the global Bible technology context has shifted from one that is "complicated" (lots of moving parts, but still predictable and structured) to one that is "complex" (elements interact in ways that are unpredictable). As the network of contributors to the Open Components Ecosystem continues to grow, the complexity of the network and the number of technologies invented to meet diverse needs also increases. Consequently, it is important to define and communicate the parameters that lead to a healthy ecosystem. Otherwise, the increase in the complexity of the interactions between all the different technologies could result in decreased efficiency and compatibility. Simple rules can be used in complex environments to maximize collaboration.[29]

Principles for building the Open Components Ecosystem:

The principles that address practical aspects of designing software for the Open Components Ecosystem can be encapsulated in one sentence: *Focus on less complicated tools that are usable by more developers to meet the needs of more users.*

1. Focus on less complicated tools… Here we note the "less is more" principle: it is more advantageous to design specialized tools that tackle individual problems.

29 Sull and Eisenhardt use the concept of principles as "simple rules" and argue that "meeting complexity with complexity can create more confusion than it resolves." They go on to ask, "How can people manage the complexity inherent in the modern world? Our answer, grounded in research and real-world results, is that simple rules tame complexity better than complicated solutions" (Sull and Eisenhardt, *Simple Rules*, 12).

2. … that are usable by more developers … Do this by creating modular and reusable open-source code that is well documented (and preferably easy to test interactively by means of an online "sandbox")

3. … to meet the needs of more users. Prefer simple interfaces that increase reconfigurability in order to maximize usefulness across the broadest range of users.

Principles for creating Open Components

In order to help illustrate the distinctive features of the Open Components Ecosystem, consider the manufacture of motor vehicles. A completely customized vehicle would be composed entirely of non-interchangeable functionality. Everything from spark plugs to doors and wheels would be custom-made for only that vehicle. There would be some advantages to this approach (e.g., extreme fine-tuning of the vehicle design at every point). But it would be difficult for anyone apart from the original designer to understand how all the components fit together in the vehicle. Consequently, it would result in a vehicle that would be cumbersome and expensive to upgrade or repair.

This is why the motor vehicle industry today is generally characterized by interchangeable components that can be individually repaired and upgraded. By designing for an ecosystem, different vehicles can be assembled from the same components for a variety of different uses, from motorcycles to sedans to pickup trucks. Innovators can improve individual components without needing to design an entire vehicle around it—the new or improved component can be integrated into a vehicle composed of other pre-manufactured components. In a similar way, the Open Components Ecosystem can be defined by three principles that set it apart from other technology-development models: easy to reuse, easy to change, easy to learn.

1. Easy to reuse

A key intent of the ecosystem is to maximize the number of projects that can take advantage of existing code while minimizing the amount of duplicated effort. The well-known book about good software design, *The Pragmatic Programmer*, says this about reusability:

> What you're trying to do is **foster an environment where it's easier to find and reuse existing stuff than to write it yourself.** If it isn't easy, people won't do it. And if you fail to reuse, you risk duplicating knowledge.[30]

These are key considerations that maximize the reusability of a component.

- **Build for the ecosystem:** With the proper separation of concerns, a component can be reused not only in a single organization's suite of software tools but also shared among multiple organizations. No more copy and pasting of files and lines of code across projects or hunting down each project that does the same thing to apply patches and bug fixes. The main point with reuse is keeping things DRY, as in Don't Repeat Yourself. This holds both for the same application and for different applications by the same team and even across other teams. Proper versioning of the modules using Semantic Versioning can communicate backwards compatibility and allow each project to lock down and limit unexpected changes.[31]

[30] Thomas and Hunt, *Pragmatic*, ch. 10 § "Topic 9: DRY—The Evils of Duplication, Tip 16: Make It Easy to Reuse", emphasis added.

[31] See Semantic Versioning (https://semver.org).

- **Do one thing well:** Smaller components are easier to reuse, so it is ideal to limit the scope of each one. Use the Unix philosophy: do one thing and do it well. Complex modules that do too much can be tempting, especially on related functionality, but come at the cost of limiting reuse.[32]
- **Standardize I/O:** To maximize reuse among disparate systems, it is helpful to standardize the input and output, especially for data formats.[33] This can be challenging for developing new functionality as there may not be an existing standard. Much time and effort can be spent exploring the standardization, so be careful not to be paralyzed by this. You can always release a new version of the module with a refined I/O standard. Each iteration should strive to improve this.
- **Default to stateless:** Stateless modules and components (or those that maximize the amount of state internally managed) are the easiest to reuse. Some functionality requires state and can be abstracted so that the code is broken down into stateless portions as well as wrapped with optional state management. Going stateless can decouple the state-management concern from the application and increase compatibility.
- **Open source:** All components in the ecosystem should be clearly licensed under an open-source license that permits the broadest reusability of the component (e.g., MIT Public License or similar).

2. Easy to change

A second key design consideration of the Open Components Ecosystem is that components should be designed for maximum flexibility to adapt to changing user needs and contextual factors. This means writing code that is intended to be easy to upgrade and replace, as described in *The Pragmatic Programmer*:

> A thing is well designed if it adapts to the people who use it. For code, that means it must adapt by changing. So we believe in the ETC principle: Easier to Change.[34]

By way of analogy, think of the last time that you had a flat tire on your car. Imagine what it would feel like if the wheel was not easily removable and was integrated with your axle. Now imagine what it would feel like if your axle was a part of the chassis.There would be no way for you to replace your tire on the side of the road. There is a reason that wheels are easily changeable. In a similar way, components in the Open Components Ecosystem should be made easy to change by implementing as many of the following characteristics as possible.

- **Separation of concerns:** The most useful components in the ecosystem are designed such that they can be swapped out easily, in order to easily implement new or improved functionality. The goal is to find the right balance in terms of the granularity of the component, like changing a car tire on the side of the road, rather than needing to replace the wheel, axle, and part of the drivetrain with the help of an expert mechanic. Most of the time, it is easier to err on the side of a component being too

---

[32] The UNIX philosophy can be summarized as: write programs that do one thing and do it well, write programs that work together, and write programs to handle text streams, because that is a universal interface (Salus, *UNIX*, 52).

[33] A promising solution for interoperable data formats for Bible projects is Scripture Burrito (https://docs.burrito.bible/).

[34] Thomas and Hunt, *Pragmatic*, ch. 2 § “Tip 14: Good Design Is Easier to Change Than Bad Design.” The authors go on to note: “given that you’re not sure what form change will take, you can always fall back on the ultimate “easy to change” path: try to make what you write replaceable. That way, whatever happens in the future, this chunk of code won’t be a roadblock.” Later in the book, they expand on the importance of code modularity and changeability in light of future complexities: “The more you have to predict what the future will look like, the more risk you incur that you’ll be wrong. Instead of wasting effort designing for an uncertain future, you can always fall back on designing your code to be replaceable. Make it easy to throw out your code and replace it with something better suited. Making code replaceable will also help with cohesion, coupling, decoupling, and DRY, leading to a better design overall” (ch. 4 § “Topic 27: Don’t Outrun Your Headlights”).

small, as one can always combine the smaller pieces together to ease the implementation and compatibility.

- **Functional isolation:** This provides a guideline for logical interchange between related parts. This has multiple benefits in areas such as ease of testing, reduced side effects, clearly defined boundaries, and ease of replacement.
- **Decouple UI from business logic:** Decoupling the user interface from the business logic is also important, as it allows the UI and data manipulation to be interchangeable. Sometimes you like the UI but want to provide your own functionality. Other times you like the functionality but want to use your own UI.[35]
- **Minimize data migrations:** Minimize data migrations to ease upgrades and reduce the long-term maintenance. Avoid such things as persisting intermediate data formats including databases and using custom data formats when possible. It can be tempting to store or cache an intermediate data representation for performance reasons but it comes at a steep price. Many times this seems as though it is the only solution, but rarely is the optimization worth the tradeoff in the long term. If this rule is followed, effective solutions eventually present themselves, with the added benefit of minimizing long-term maintenance costs.
- **Maximize learning through iterative development:** Architecting and designing software that is easy to change can take time and many iterations to get right. One approach to creating applications that are usable while teasing out the proper separation of concerns is the progression of a Proof-of-Concept → Prototype → Minimum Viable Product (MVP). Starting with a proof of concept, you can set the vision for what needs to be built but in a quick, sloppy way. No need to waste time worrying about how to do things the right way. The goal is just to get it done the quickest way possible. The next phase is rewriting the code in a way that starts to detangle the separation of concerns in a modular way. Break down the code to what seems reasonable. Once the prototype has a few iterations, the architecture and design for the MVP will become more clear and apparent.[36]

3. Easy to learn

Every developer knows that documenting code is important, both for their own reference in the future, as well as to inform others who may need to maintain the code. *The Pragmatic Programmer* addresses this directly in **Tip 13: Build Documentation In, Don't Bolt It On**:

> It's easy to produce good-looking documentation from the comments in source code, and **we recommend adding comments to modules and exported functions** to give other developers a leg up when they come to use it.[37]

If a picture is worth a thousand words, a **visual sandbox** that enables a developer to test the functionality of a component in their web browser is worth at least that much traditional documentation! Even headless components can often be wrapped in a visual layer that enables stakeholders, project managers, and developers to test their own data with a component and compare the pros and cons of various options.

---

35 It should be noted that many times the separation of concerns happens during the prototyping phase (see below). It is not always optimal or clear how to abstract effectively until after you prove the functionality. It can be an iterative process.

36 For more on increasing learning by iterating through PoCs, prototypes, and MVPs, see Klapp, "Lean Expectations."

37 Thomas and Hunt, *Pragmatic*, ch. 1 § "Topic 7: Communicate".

Conclusion

The transition to replaceable-parts manufacturing made the Industrial Revolution possible and helped catalyze the invention of virtually everything we use today—from automobiles and airplanes to mobile phones and the Internet. In a similar way, we have an opportunity to develop a new approach to Bible technology—one built on open components that simultaneously encourage nonlinear innovation and meaningful collaboration across a network of Bible technologists. By creating technology components that are easy for others to learn, reuse, and change, developers will create favorable conditions for innovation and widespread use of Bible technologies. New developer teams will find a low bar for entry into the Bible technology space as more functionality is made available through a growing number of components. These factors help maximize the flow of ideas, leading to more innovative solutions for more people involved in production, distribution, and use of Bible translations and other biblical resources in any language.

Learn more about the Open Components Ecosystem, explore existing components, learn how to create new components, and connect with others who are part of the movement at **https://opencomponents.io**.

References

Beard, Ross. “Microservice Architecture” – “Why a Microservice Architecture Is Important (4 Reasons).” Shadow-Soft, April 9, 2018. https://shadow-soft.com/why-microservice-architecture/.

“Composability.” In Wikipedia, June 26, 2021. https://en.wikipedia.org/wiki/Composability.

English, Trevor. “Interchangeable” – “The History of Interchangeable Parts in the Industrial Revolution,” August 31, 2019. https://interestingengineering.com/the-history-of-interchangeable-parts-in-the-industrial-revolution.

Johnson, Steven. *Good Ideas – Where Good Ideas Come From: The Natural History of Innovation.* Reprint edition. New York: Riverhead Books, 2011.

Jore, Tim. “Established” – “From Unreached to Established.” unfoldingWord, May 16, 2018. https://unfoldingword.org/established.

Kania, John, and Mark Kramer. “Collective Impact.” Stanford Social Innovation Review, 2011. https://ssir.org/articles/entry/collective_impact.

Klapp, Christopher. “Lean Expectations — PoC, Prototype, MVP.” Medium, September 8, 2018. https://medium.com/@klappy/lean-expectations-poc-prototype-mvp-140749383fd4.

Lin, Robert. “Monolithic” – “Monolithic vs Modular.” Medium, November 3, 2016. https://medium.com/@berto168/monolithic-vs-modular-9b6d69684a2c.

Preston-Werner, Tom. “Semantic Versioning 2.0.0.” Semantic Versioning. Accessed June 28, 2021. https://semver.org/.

Ries, Eric. *The Lean Startup: How Today’s Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses.* New York: Crown Business, 2011.

Salus, Peter H. *A Quarter Century of UNIX.* Addison-Wesley Publishing Company, 1994.

Slootweg, Sven. “Monolithic vs. Modular - What’s the Difference?” Gist. Accessed April 6, 2021. https://gist.github.com/joepie91/7f03a733a3a72d2396d6.

Snowden, David J., and Mary E. Boone. “A Leader’s Framework for Decision Making.” Harvard Business Review, November 1, 2007. https://hbr.org/2007/11/a-leaders-framework-for-decision-making.

Sull, Donald, and Kathleen M. Eisenhardt. *Simple Rules: How to Thrive in a Complex World.* Reprint edition. Mariner Books, 2016.

Thomas, David, and Andrew Hunt. *The Pragmatic Programmer: Your Journey to Mastery, 20th Anniversary Edition.* 2nd edition. Addison-Wesley Professional, 2019.

Wu, Tim. *The Master Switch: The Rise and Fall of Information Empires.* Reprint edition. Vintage, 2010.