11-7-2016

Open Source, Modular Platforms, and the Challenge of Fragmentation

Christopher S. Yoo
University of Pennsylvania Carey Law School

Open Source, Modular Platforms, and the Challenge of Fragmentation

Christopher S. Yoo*

Abstract

Open source and modular platforms represent two powerful conceptual paradigms that have fundamentally transformed the software industry. While generally regarded complementary, the freedom inherent in open source rests in uneasy tension with the strict structural requirements required by modularity theory. In particular, third party providers can produce noncompliant components, and excessive experimentation can fragment the platform in ways that reduce its economic benefits for end users and app providers and force app providers to spend resources customizing their code for each variant. The classic solutions to these problems are to rely on some form of testing to ensure that the components provided by third parties comply with a compatibility standard and to subject the overall system to some form of governance. The history of the three leading open source operating systems (Unix, Symbian, and Linux) confirms this insight. The question is thus not whether some constraints will apply, but rather how restrictive those constraints will be. Finally, the governance regimes range from very restrictive to relatively open and permissive. Competition policy authorities should take into account where certain practices fall along that spectrum when enforcing competition law. Exposing the more permissive practices to demanding scrutiny runs the risk of causing operating systems to turn to more restrictive approaches.

---

* John H. Chestnut Professor of Law, Communication, and Computer & Information Science and Founding Director of the Center for Technology, Innovation and Competition, University of Pennsylvania. This paper benefitted from comments at seminars at the Center for Global Communications of the International University of Japan, ICT Law & Economy Institute and Social Science Korea (SSK) Research, Group at Sogang University, the Roundtable on "Platforms and Mobile Competition" co-sponsored by the London School of Economics and Political Science and the University of Leeds, and the University of Pennsylvania Law School. The author also thanks the Milton and Miriam Handler Foundation and Google for their financial support. All views and any errors contained in this paper are the responsibility of the author.

1 Introduction

The past few decades have borne witness to the emergence of two conceptual paradigms that have fundamentally transformed the software industry. The first is the open source movement. Open source is based on the principle that every user should be able to modify software freely. In so doing, open source seeks to mobilize the entire community of end users to volunteer their time to debug the code. The freedom to build freely on existing software also enhances competition by enabling anyone being charged excessive prices to develop their own alternative. The flexibility provided by open source also promises to promote innovation by empowering all users to add new functionality to the system.

The second is the idea of platforms. Platforms are standardized architectures that divide complex systems into modules and define the interfaces that link these modules. Modular platforms represent a break from the traditional approach to managing complexity in which a single actor manages the software development process. In a modular platform, any interested

third party can create a component for the overall system so long as the resulting component complies with the standardized interface. In so doing, platforms allow multiple actors to pursue parallel innovation, which can improve the quality of the technical solution as well as increase the rate of technological change. The standardization inherent in modular platforms also allows device manufacturers and app providers to unlock the economic potential of their innovation by allowing them to reach large user bases without having to constantly create new versions for each new hardware device.

These two concepts can be powerfully complementary in certain settings. Indeed, commentators have long recognized that the distributed development model underlying the open source movement necessarily depends on modularity to divide the system into parts small enough to be improved by individual work groups and programmers and to enable multiple actors to work to improve different parts of the system simultaneously.[1] The compatibility and affinity between these two concepts is demonstrated eloquently by the fact that two of the most important operating systems, Unix and Linux, are simultaneously platforms for third-party apps and open source.

What is less well recognized is the extent to which these two concepts rest in uneasy tension with one another. While the freedom of open source suggests unlimited flexibility to change parts of the system, to function properly modular platforms require that all components adhere strictly to a predetermined set of standards that govern how the different components interconnect and interact. This tension is well illustrated by the problem of fragmentation, which has long been recognized as a major problem for many open source projects. The most extreme form of fragmentation, known as forking, occurs when a contributor to an open source project

---

[1] See, e.g., Feller and Fitzgerald (2002, 76-79, 170-71); Weber (2004, 172-74).

customizes a non-application layer of a platform to an extent that it is no longer fully interoperable with the rest of the project. The result is to divide the system into two distinct and incompatible versions.

Such fragmentation represents a conundrum for open source. On the one hand, users' freedom to customize software is integral to the open source movement. Indeed, absent constraints, the freedom inherent in open source effectively gives users the ability to fragment the system.[2] On the other hand, infinite flexibility creates costs for the open source community by requiring the diffusion of effort and the duplication of work across multiple projects. Fragmentation also harms device manufacturers and app developers by limiting interoperability and by requiring them to adapt their products for what are now separate platforms (a process called *porting*). End users are often disappointed to find that particular software works only on some platforms.

The success of an open source platform thus depends on reconciling the freedom inherent in open source with the compatibility required by modular platforms. Some constraints on the flexibility of open source are thus inevitable. The real policy question is what type of constraints are appropriate.

This Article analyzes the complex relationship between open source and modular platforms by describing the basic principles underlying each approach and examining the extent to which they are simultaneously fundamentally interconnected and in inherent tension. It then explores the history of three leading examples of open source operating systems—Unix, Symbian, and Linux—to illustrate how these dynamics work in practice. It concludes by examining what lessons these histories have for the current debate over the propriety of

---

[2] See, e.g., Weber (2004, 64, 89, 170); Corbett (2011).

restrictions to open source mobile operating systems, paying particular attention to Google's Android Anti-Fragmentation Agreement. It also lays out key features that make such restrictions less problematic from the standpoint of competition policy.

The core lesson is that some restrictions on what people can do with open source operating systems are necessary if consumers are to enjoy the full benefits of competition and innovation. My point is not to suggest that open source software is inherently superior to proprietary software or vice versa. Both approaches have distinct virtues that appeal to different users. Moreover, any attempt to cast the policy debate as a choice between those polar extremes is based on a false dichotomy. Instead, the different modes for producing software platforms are better regarded as occupying different locations along a continuum running from completely unrestricted open source to completely proprietary closed source. Indeed, companies may even choose to pursue hybrid strategies that occupy multiple locations on this continuum simultaneously. The diversity of advantages associated with these different approaches suggests that consumers benefit if different companies are given the latitude to experiment with different governance models, with the presence of one open source platform serving as an important competitive safety valve. Moreover, the analytical framework suggests that a completely open source platform represents an ideal type that is inherently unrealistic. The fact that open source platforms are subject to some constraints is thus not inherently problematic. The proper role for competition policy is to provide a framework for determining when such constraints are reasonable.

2 The Conceptual Underpinnings of Open Source and Modular Platforms

Understanding the simultaneous connection and tension between open source and modular platforms requires an appreciation of the principles underlying each concept.

2.1 Open Source

Open source software is often put forth as a new paradigm in software production. Computer programs can be distributed in two forms.[3] The first is known as *source code*, which is written in a programming language such as Pascal and Fortran, to use two dated examples that have been replaced by newer languages such C++, Python, and Perl. Although source code is quite technical, experienced programmers can read and modify it. Source code is then compiled into *object code* or *machine language*, which consists of a series of 0s and 1s. Object code can be read by computers, but cannot easily be read by human beings.

One of the main triggers for the open source movement was software companies' practice of attempting to protect their software by distributing it only as object code and refusing to release the source code. These companies also copyrighted their code and included clauses in end-user licenses prohibiting customers from modifying it. The absence of the source code and the contractual restrictions on modifying the code made it difficult for end users who wished to customize the code to diagnose and resolve incompatibility problems.

Frustration over the inability to customize code led to the open source movement. Although numerous definitions of what constitutes open source exist, they generally agree that all software should be distributed with its source code or that the source code be made available on request. Open source definitions also generally share the requirement that end users be permitted to modify the code and distribute their modifications. Beyond these basic commitments, open source exhibit considerable variation. For example, the GNU Public License (GPL) contains a *viral* provision that requires that any code that is combined with GPL-licensed

---

[3] The foregoing discussion draws on Marella and Yoo (2007).

code to be governed by the GPL. Because the GPL enforces openness through copyright licenses, the viral provisions are sometimes called *copyleft* requirements. Other open source licenses, such as those used by Berkeley and Apache, take a more academic approach, simply requiring that any modification provide clear notice of the changes and give appropriate credit to the creators of the original code. Other variants exist as well, with the Open Source Initiative currently listing seventy-eight approved licenses.

The existence of multiple licenses reflects a divergence of philosophies within the open source movement.[4] Some early movement pioneers, such as Richard Stallman, emphasize the freedom to tinker and rely on the viral copyleft provisions to prevent proprietary and open source software from being combined. Others, such as Bruce Perens and Eric Raymond, adopt a less hostile, more pragmatic approach that permits open source and proprietary software to be combined.

Beyond these formal attributes, open source projects depend on a vibrant community willing to volunteer their time to improve and extend the project. The belief is that opening up the opportunity to improve the code to the entire user base will increase the total number of person-hours devoted to the project and will identify and fix problems more rapidly. This spirit has been captured by Eric Raymond (1999) dubbed Linus's Law: "Given enough eyeballs, all bugs are shallow."

2.2 Modular Platforms

One of the biggest problems confronting any major software project is how to coordinate the various teams working on different parts of the system. One of the most famous examples

---

[4] *See, e.g.*, DiBona et al. (1999, 8-9); Perens (1999, 80); Stallman (1999, 37); McGowan (2001, 260-65).

arose when IBM was developing the System/360 computer. To make sure that the entire team understood the full intricacies of the design, the project managers required that every programmer maintain a workbook documenting all of the other parts of the system. In just six months, the workbook was five feet long and required filing of 150 pages of updates each day. Worse yet, even when the project was running behind, managers found that adding more personnel actually slowed the project down. This insight has led to the coining of what is known as Brooks's Law (1975, 31), which holds that "adding manpower to a late software product makes it later."[5]

In addition, one of the hallmarks of a complex system is the way that components can interact with one another in unexpected ways.[6] Validating a complex system requires testing every possible combination of states of the world that each of the various components of the system can possibly occupy. If the number of interdependencies is large, the number of unique combinations of parameters that must be tested can rapidly become immense, particularly if each component is permitted to occupy a large number of states.[7] The problem becomes even more difficult if the interdependencies form a circuit that recursively loops back onto itself (e.g., if task *A* depends on task *B*, which in turn depends on task *C*, which depends on task *A*). When that is the case, testing requires exploring not only every possible combination of states of the world, but also cycling through enough iterations until each combination reaches stability.

The traditional approach to managing the inherent complexity of large software projects is for a single actor to coordinate and control all of the design teams working on a project and to use managerial processes to ensure that the communication and testing needed for proper

---

5 Brooks (1975, 31).
6 The analysis of modularity draws on Yoo (2016).
7 Dijkstra (1968, 344).

integration of the design occurs. The tightness of the control means that most firms either produce components themselves or maintain strict control over any third parties on which they rely to produce components of the overall system.

A new approach has emerged that replaces the strong integrated design with a *modular architecture*. Modular architectures minimize system complexities by defining the modules so that highly interdependent tasks are clustered within the same module.[8] Cross-module interdependencies are limited by requiring that modules interact with one another solely through predetermined interfaces that strictly cabin the amount of information that can pass between modules, the details of which are often defined in open standards.[9]

The existence of these standardized interfaces minimizes the need for firms producing components to coordinate with one another. So long as a component manufacturer conforms to the standard, any third party can produce compatible components. All of the information needed to coordinate with other modules is embodied in the standards. This allows third parties to work on different components of the same system without having to worry that any changes made to any one component might create ripple effects throughout the entire system. Moreover, it allows multiple teams working in parallel to experiment with different technical ways to implement a particular module, allowing greater latitude to experiment with different solutions and faster innovation.

---

8 For the classic statement, see Simon (1962).

9 Parnas (1972).

3 The Complex Relationship Between Open Source and Modular Platforms

Open source and modularity are both recognized as important underpinnings of the modern Internet economy. Indeed, modularity is often identified as a critical success factor for any open source project. As Linux founder Linus Torvalds (1999, 108) succinctly noted, the open-source development model depends on "hav[ing] a system which is as modular as possible," because without modularity, "you can't have people working in parallel," and "I would have to check every file that changed."[10]

A closer examination reveals that the relationship between open source and modular platforms is more complex than this simple statement would lead one to believe. Although open source cannot exist without modularity, the infinite flexibility inherent in open source exists in uneasy tension with the strict structural requirements upon which modular platforms depend.[11]

3.1 The Synergies Between Open Source and Modularity

Open source and modularity are widely regarded as complementary concepts. Indeed, modularity is essential for an open source project to succeed. As Torvalds's statement quoted above indicates, decomposing a larger system into subsystems connected by minimal interdependencies isolates each component in ways that make it easier for multiple groups to work on improving different components simultaneously. This allows designers to experiment with improvements to particular parts of the code without having to worry continually about creating problems for other parts of the system.

---

[10] For academic studies of the link between modularity and open source, see, e.g., Bonaccorsi and Rossi (2003, 1247); Feller and Fitzgerald (2002, 76-79); Weber (2004, 172-74); Clark and Baldwin (2006); Midha and Paliva (2012, 903).

[11] The discussion that follows draws heavily on the superb analysis in Weber (2004). For other important accounts, see DiBona et al. (1999) and Feller and Fitzgerald (2002).

Indeed, the rigid logical structure through which modules are interconnected is what allows multiple third parties to work on the same project. Conway's Law (1968), which has long been recognized as a central tenet of software production, holds that the architectural structure of technical systems mirrors the organizational structure that produces them. This means that without the distributed nature inherent in a modular architecture, the distributed organizational production system that characterizes open source could not exist.

3.2 The Tensions Between Open Source and Modularity

Open source advocates have acknowledged that the freedom to innovate that lies at the heart of open source software represents something of a two-edged sword. Open source inherently gives end users complete latitude to customize software as they see fit. Although such unfettered freedom is unproblematic when the code is run in isolation, it becomes more problematic when the code is supposed to interoperate with the other components of an interoperable platform. As noted earlier, modular platforms depend on standardized interfaces that predefine how different modules will interact with one another. Although one can experiment with different configurations of tasks within a module, interactions between modules must strictly adhere with the interfaces. Any code that does not conform to the modular design becomes noninteroperable with the rest of the system.

The tension between flexibility and structure can lead to two characteristic problems with open source platforms. The first is the temptation for people modifying individual components to introduce interdependencies that deviate from the modular architecture. The second is the possibility that a subgroup of an open source project may fragment the project, in extreme cases dividing into two distinct and incompatible branches in a phenomenon called *forking*.

3.2.1 The Temptation to Create Noncompliant Modules and the Need for Testing

As noted earlier, the key design feature of a modular architecture is the clustering of highly interdependent tasks within the same module and ensuring that the interdependencies that are supposed to be encapsulated within that module do not affect other modules. The key to ensuring that these interdependencies remain isolated within a module is to design the module interfaces so that they contain only information associated with interdependencies that are permitted by the design and to require that other modules restrict themselves to interact only with the information made visible by the interface. All information about other independencies remains hidden within the module.

The tradeoff inherent in this approach means that "designers will lose the ability to explore some parts of the space of designs—in effect, the architects will restrict the search, declaring some parts of the design space to be out of bounds."[12] More specifically, the generality inherent in modularity inevitably leads to a degree of inefficiency.[13] There will inevitably be occasions where one module finds that the most efficient way to solve the problem at hand would be to refer to information contained in an adjacent module, despite the fact that that information is excluded from the module interface and is thus associated with an interdependency that is supposed to remain encapsulated within the module. Moreover, generality requires incurring the cost to support features that particular implementations may never need.

---

[12] Baldwin and Clark (2000, 68).
[13] McGee (1959, 2); Clark (1982, 16).

The inefficiency and inflexibility inherent in this result led early scholars to denounce the use of modular interfaces as "radical."[14] Over time, these critics began to concede that using information hiding to implement modularity created real benefits.[15] This concession does not eliminate the reality that inefficiency remains an irreducible part of any modular platform and that open source module developers have both the ability and the incentive to access information associated with interdependencies that they are not supposed to take into account. This dynamic explains why the number of interdependencies among Linux modules has increased exponentially with each release.[16]

A recent dispute between Skyhook Wireless and Google provides an apt illustration of these dynamics.[17] Both Skyhook and Google provide location services, which are apps that identify the latitude and longitude coordinates for the location of the device. Location services determine geolocation data from one of three sources: (1) global positioning satellites (GPS), (2) WiFi access points whose locations have been stored in a manually compiled database, and (3) triangulation on cell tower locations. Of these three, GPS is considered the most accurate, but is typically slower than the other methods.[18] All location services incorporate the data they collect into the existing databases. Because GPS data is considered more accurate, the Android GPS application programming interface (API) reports data collected from GPS, WiFi, and cell towers separately to give developers that rely on this data a clear understanding of its quality.[19]

---

14 Brooks (1975, 78).
15 Brooks (1995).
16 Feller and Fitzgerald (2002, 176).
17 *Skyhook Wireless, Inc. v. Google, Inc.*, 30 Mass. L. Rptr. 417 (Super. Ct. 2012).
18 *Id.* at 418.
19 *Id.* at 418, 419, 421.

Both Google and Skyhook use all three methods to determine the location of a mobile device. Motorola was considering including Skyhook's location service, known as XPS, into one of its devices. Google carefully differentiates between location data based on GPS and location data derived from network information, such as the location of WiFi access points and cell towers.[20] XPS, however, reported both GPS-based and network-based data together.[21] When Google found out about these plans, it informed Motorola that XPS's location services did not comply with the Android compatibility standard, although it did make clear that Motorola was free to include Skyhook if XPS was modified to stop returning network-based data into the GPS database.[22] Eventually, Motorola removed XPS from its devices.[23] Skyhook sued Google for intentional interference with contacts and business relationships. The Massachusetts Superior Court granted summary judgment in favor of Google on all counts.[24]

Modularity theory provides a clear basis for understanding why the court's decision was correct. To function properly, every module must be able to trust that all of the information being sent by other modules comprising the system complies with the design architecture. To ensure that is the case, some means must exist for identifying and excluding noncompliant modules. Compliance and testing mechanisms ensure that each module can rely on the fact that all of the other modules are operating in the manner specified by the design.

When a mobile platform is proprietary, the subgroups designing individual modules rely on the command and control apparatus of the company to ensure that this is the case. In contrast, when a mobile platform is open, there is no single actor exercising control over all of the

---

20 *Id.* at 418.
21 *Id.* at 419.
22 *Id.* at 425.
23 *Id.* at 420–22, 423–24.
24 *Id.* at 418, 424, 427.

modules. Instead, the activities of the different modules are coordinated by the information structure of the architecture rather than a firm. Modules must restrict themselves to sending only the information that the other modules expect if the architecture is to function properly. All actors participating in an open platform depend on the presence of some governance mechanism for ensuring that all of the components created by the various third-party providers comply with the architecture. Thus exclusion of a noncompliant app from the system should not automatically be regarded as a sign of anticompetitive or improper behavior. On the contrary, it may be a necessary part of any open architecture.

3.2.2 Fragmentation

The flexibility inherent in open source software can give rise to a problem more severe than noncompliant modules. Sometimes participants in open source projects go beyond tinkering with the design of individual modules and take the architecture in a fundamentally new direction. In extreme cases, the divergence can create a *fork* in the open source project that causes the project to divide into two different and noninteroperable branches, each pursuing its own path.

Some forms of fragmentation or differentiation are not without redeeming qualities. For example, forking may represent a diversity of interests, typified by the fact that even-numbered Linux are experimental releases filled with new features that have not been fully debugged, while odd-numbered Linux releases constitute stable resales that have been thoroughly tested. The former is designed to appeal to sophisticated developers interested in conducting research on the cutting edge, while the latter is intended to meet the needs of commercial and less sophisticated users who are more interested in reliability and ease of use. Moreover, forking can allow third-parties to reinvigorate open source projects that are stuck in inefficient designs.

More importantly, the flexibility integral to the open source movement in effect gives users the fundamental ability to fragment or fork.

At the same time, fragmentation can greatly impede the likely success of an open source project. Fragmentation can force app developers to develop different version for each noncompliant module, a process called porting. Dividing an open source project into separate forks forces what was once a single community working on one project to divide its energy and duplicate efforts across two separate projects. In addition, the community developing apps for the operating system must now spend the time and effort to make sure that their products are compatible with both branches of the fork. Developers would ideally prefer to operate in an environment in which they can "write once, run anywhere." Excessive fragmentation and noninteroperability would frustrate their ability to do so.

The tension represents what Martin Libicki (1995, 47) has called the "fundamental contradiction" between open source and modularity:

> The more open the system, the more it can be modified by vendors and users to their own ends, which is good. The more a system is modified, however, the more likely that the modifications will be nonstandard. With many nonstandard versions of UNIX available, software vendors need to disperse (perhaps dissipate) their software efforts among many systems, leading to fewer pieces of software available to any one system. . . . This result reduces choice, which is bad.

3.2.3 Mechanisms for Resolving These Tensions: Testing and Governance

How do open source projects manage the inherent tension between open source and modularity? Whereas open source implies flexibility and freedom, modularity requires a highly structured and restrictive environment to ensure conformity with the architecture and to provide a sufficiently stable platform for the developer community. What keeps open source projects from fragmenting in an inefficient manner?

As an initial matter, some open source communities rely on a series of informal governance mechanisms to maintain their projects' coherence. For example, open source communities typically have produced a fairly strong norm against forking. In the words of Eric Raymond, "There is strong social pressure against forking projects. It does not happen except under plea of dire necessity, with much public self-justification, and with a renaming."[25] Furthermore, the incentives confronting a person considering whether to create a fork can be quite daunting. All participants in the new fork would be part of a smaller community, which would mean fewer collective benefits and a greater obligation to do work. The magnitude of these liabilities increases when the existing open source project that is being forked is large. Moreover, if the new fork does not attract sufficient followers, it will fail.

While important, these informal mechanisms are too weak to ensure coherent management of an open source project. With respect to noncompliant modules, modularity theorists regard the existence of a system for testing and verifying the performance of other components as an essential part of any modular system. Harvard Business Professors Carliss Baldwin and Kim Clark (2000, 380) note that "the testable, verifiable dimensions of the module are the foundation that supports arm's length-contracts and market transactions" and that "without tests, there is no way to know what is being bought and sold."

To prevent excessive fragmentation, most open source projects rely on some form of strong formal governance. This comes as a surprise to many observers. The mythology holds that open source projects consist of widely dispersed communities organized from the bottom up, within which all members make their own small contributions to the overall project, excellence

---

[25] Feller and Fitzgerald (2002, 96).

is determined by peer review and who works the hardest, and the community adopts the pragmatic and meritocratic position of "letting the code decide."

In practice, open source projects operate in a much more concentrated and hierarchical manner. In fact, studies have indicated that 85% to 90% of contributed code is discarded.[26] Another study indicated that ten developers (less than 0.1% of the overall universe of developers) contribute almost 20% of the code base for each project.[27]

Decisions about which contributions are accepted are made in a similarly hierarchical manner. For example, the oft-cited article by Harvard Business School Professor Josh Lerner and Nobel Laureate Jean Tirole noted that open source projects are characterized by "a strong centralization of authority."[28] Another early commentator noted, "Open source may sound democratic, but it isn't. Leaders of the best-known Open Source development efforts often explicitly stated that they function as dictators."[29]

In fact, the term dictator has been used to describe the leadership of a wide variety of open source projects, such as Linux and Python (although in Linux, Torvalds has delegated a great deal of authority to two lieutenants).[30] Perl has developed a rotating dictatorship, in which authority is passed among a small inner circle of Perl developers,[31] with Perl creator Larry Wall serving as the final arbiter.[32] Even the Apache server project, which has been called "as close to a democracy as one is likely to find in software development," is controlled by two dozen

---

26 McKusick (1999); Mockus et al. (2000).
27 Ghosh and Prakash (2000).
28 Lerner and Tirole (2002, 221).
29 Bezroukov (1999).
30 DiBona et al. (1999, 12) (calling Linux a "benign dictatorship"); van Rossum (2008) (referring to the founder of Python as "benevolent dictator for life");
31 Weber (2004, 92).
32 Feller and Fitzgerald (2002, 91).

developers, all of whom wield veto power.[33] Many other open source projects are governed by a foundation.

Linux creator Linus Torvalds explicitly acknowledges that the control provided by Linux's hierarchical governance structure allows him to take bolder action: "[T]he fact that there is one person who everybody agrees is in charge (me) allows me to do more radical decisions than most other projects can allow."[34] Conversely, Unix collapsed in large part because no user group or actor had the authority to make decisions for the platform.

The presence of such governance hierarchies is fundamentally at odds with the collectivist mantle in which the open source movement tends to wrap itself. Eric Raymond (1999) famously analogized the differences between proprietary and open source software to the differences between a cathedral and a bazaar. Like proprietary software, cathedrals are top-down projects "carefully crafted by individual wizards or small bands of mages working in splendid isolation, with no beta to be released before its time." Open source communities, in contrast, are more like bazaars: great babbling marketplaces "of differing agendas and approaches," bustling about in apparent confusion.

As the presence of strong hierarchies reveals, the truth lies somewhere in between. The presence of strong formal governance reveals that the so-called bazaar has many cathedral-like qualities and that the sharp distinction between cathedrals and bazaars may represent a false dichotomy.[35] Even the most free-wheeling environments must have some rules and means for settling disputes, particularly if they must conform to a strict set of architectural rules in order to preserve interoperability. Moreover, the type of open-source license can affect the strength of

---

33 Maclachlan (1999).
34 Yamagata (1997).
35 Feller and Fitzgerald (2002, 159-60).

the governance mechanism. The viral copyleft provisions of the GPL ensure that any noninteroperable customizations will be available to the developer and user community. Consequently, open source projects that rely on the GPL have less need for governance mechanisms to protect against fragmentation. BSD/Apache-type licenses permit software developers to assert proprietary control over their modifications. As a result, open source projects relying on the latter type of license typically employ stronger forms of governance to ensure that the ecosystem remains interoperable.

The nature of leadership also takes on a different character in the context of open source. Success of an open source project depends on inspiring a community of people willing to work on it. In a real sense, an open source leader's authority depends on the existence of followers. In a world where all contributions are voluntary and the community is always free to exit the community by forking the project, leaders' ability to retain their positions depends largely on their responsiveness to the needs of those led. These needs include providing fast feedback, serving as an effective moderator of technical disputes and personality conflicts, and realistic interim and long-term goals.

To say that open source projects require a type of leadership that is somewhat different from the leadership that characterizes commercial companies that produce proprietary software is not to say that they need no leadership at all. On the contrary, ensuring that an open source platform does not fragment depends on the presence of an actor with sufficient authority to resolve disputes and to steer the platform in a beneficial direction.

4 Lessons from the Past: Unix, Linux, and Symbian

The concepts of open source software and modular platforms represent something of a paradox. They are inextricably bound together, while at the same time resting in uneasy tension

with one another. Although open source holds out the promise of unbridled freedom, to the extent that the software needs to interoperate with other components on a standardized basis, it is not completely free.

Fortunately, two classic solutions exist to this problem. First, the fact that some components will be provided by third-parties requires the existence of some means to test components for compliance with the architecture. Second, the possibility of forking requires some form of governance to help prevent the platform from fragmenting.

A review of the histories of three well-known open source operating systems—Unix, Symbian, and Linux—provides an eloquent illustration of these dynamics. The case study of Linux serves as an example of how these dynamics can benefit end users. Although Unix and Symbian have enjoyed some degree of success, their ultimate fate consigns them more to the role of cautionary tales.

4.1 Unix

Unix exemplifies both the upsides and downsides of open source software. On the one hand, it represents one of the first successful open source projects. Indeed, some commentators have called it "perhaps the greatest software innovation of all time."[36] On the other hand, it eventually became so badly fragmented that it has become the classic example that everyone uses to illustrate what not to allow to happen to an open source project.

Unix was originally written by Ken Thompson of AT&T Bell Laboratories in a single month to enable him to play a computer game called Space Travel on a then-outdated PDP-7

---

[36] Libicki (1995, 47).

computer. It was designed to be a simple operating system that presented the same interface and functionality across a wide range of different types of machines.

At the time Unix was created, AT&T was operating under a 1956 antitrust consent decree that prohibited the company from entering into the computer business and required AT&T to license its patents. As a result, AT&T initially did not try to commercialize Unix and instead licensed it to universities royalty free. The University of California at Berkeley showed particularly strong interest in Unix, particularly after Thompson spent a semester teaching there in 1975. Berkeley programmers began improving the operating system in the late 1970s and began releasing a package of tools and utilities called the Berkeley Software Distribution (BSD), subject to an open source license requiring clear notice of any modifications and appropriate credit to the creators of the original code.

During the late 1970s and 1980s, the collaboration between AT&T and Berkeley became wildly successful, as users ported it to a wide variety of different machines and it became a key platform for the TCP/IP suite of protocols. Over time, however, AT&T began imposing greater restrictions on the distribution of the Unix source code. In 1982, the settlement of the antitrust case that broke up AT&T led to the spinoff of Bell Labs and AT&T's equipment manufacturing subsidiary, Western Electric, into a separate company that would eventually become known as Lucent Technologies. The revisions to the consent decree lifted the restrictions that prevented Bell Labs from commercializing Unix.

The prospect that Unix might become proprietary led the Berkeley group to recruit a large group of volunteers to expand BSD into a complete version of Unix that was independent of any code created by AT&T. Other companies began creating their own versions of Unix, some based on BSD (such as Apollo, DEC, Integration Solutions, and NSC), others based on

AT&T's version (such as Altos, Apollo, Compaq, HP, IBM, Intel, Microsoft, and Silicon Graphics), and still other entirely new instances based on neither version (such as Cray, DEC, Data General, Motorola, and Unisys). In 1987, AT&T attempted to end the fragmentation by entering into a strategic alliance with Sun Microsystems. In 1988, Apollo, DEC, HP, IBM, Bull, Nixdorf, and Siemens responded by creating the Open Software Foundation with the stated (but ultimately unsuccessful) goal of creating a Unix version that did not depend on AT&T licenses. AT&T and Sun created a rival organization known as Unix International to promote the AT&T version.

By 1990, fragmentation had left the proprietary side of the Unix market in a state of crisis. During the mid-1990s, differences of opinion regarding the technical direction of the platform and sharp personality clashes caused the academic side of the Unix market to fragment as well (with FreeBSD, OpenBSD, and NetBSD emerging as separate forks). The protracted legal battle waged between AT&T and the Berkeley group from 1991 to 1994 over Berkeley's use of the original Unix code added additional uncertainty to the future of Unix.

The result was the coexistence of multiple, incompatible versions of Unix, in direct contravention of the hope that Unix would provide a uniform platform that would not require app developers to port their software to each individual machine. Larry McVoy of Sun Microsystems warned in late 1993 that "Unix is dying," had "become stagnant," and had "ceased to be the platform of choice for the development and deployment of innovative technology," but his attempts to reunify the environment fell on deaf ears.[37] Shortly thereafter, Unix was overtaken by Microsoft on the proprietary side and Linux on the open source side.

---

[37] Weber (2004, 98).

The problems that led to Unix's demise are summed up nicely by a 1985 *Computerworld* article, which asked, "What's Wrong with UNIX?" and concluded that there were too many versions, each with its own unique tweaks. In short, the flexibility that is on the one hand the greatest virtue of open source at the same time became Unix's greatest vice. In the words of one user, "Unix is larger and more flexible than it has to be. Systems with less flexibility can often provide better solutions ."[38]

The collapse of Unix represents a classic example of fragmentation. The existence of multiple versions of Unix forced the software community dedicated to debugging and improving the operating system to disperse its energy across multiple, duplicative efforts. Unix was also dogged by the lack of a standardized and friendly user interface. The lack of a unified platform prevented app developers from leveraging compatibility and forced them to spend the resources needed to create specialized versions for each environment.[39] The Unix universe also lacked a strong leader with the authority to resolve disputes and put the platform back on the right track. The lack of any mechanism or authority for offering some guidance over Unix's evolution prevented the community from creating a solution even after these problems had been recognized.

Unfortunately, these problems emerged at a critical time in the computer industry. The creation of Windows NT in 1993, which was the first version of Windows that was completely free from MS-DOS, led to its widespread adoption in the PC world. IBM, Hewlett Packard, Sun Microsystems, Santa Cruz Operation, Univel, and UNIX System Laboratories made a last-ditch effort to unify the platform, but failed. Novell tried to forestall the inevitable by making Unix

---

[38] Korzeniowski (1985).
[39] Weber (2004, 98)

completely open, but to no avail. At the same time, the developer community left for Linux. The Open Software Foundation attempted to stem the tide, merging first with Unix International and then with a consortium of European Unix system operators known as X/Open to form the Open Group. The Open Group eventually joined with IEEE to certify a unified Unix specification in 2001. By this time, however, Windows and Linux had displaced Unix as the operating system of choice. The near total absence of new adoptions means that Unix's future is quite bleak.

4.2 Symbian

The second cautionary tale is Symbian. Called "Android before Android,"[40] Symbian dominated the early market for mobile operating systems, peaking at a market share of 67% in 2006, and was the favored platform for Nokia, Samsung, Motorola, and Ericsson. It continued to lead the market until 2010, when Android finally passed Symbian in terms of new shipments. Its market position was once characterized as "total dominance," but by 2013 was recognized as "sliding into obscurity."

Symbian began in 1998 as a joint venture between Psion Software (the creator of the predecessor operating system EPOC) and three phone manufacturers, Ericsson, Motorola, and Nokia. From the beginning, Symbian was badly fragmented. The sheer variety of physical form factors and screen sizes meant that distinct versions of the operating systems had to be customized for each individual device.[41] Moreover, although Symbian phones shared the same shell operating system, different groups of phone manufacturers created their own mutually

---

[40] Best (2013).

[41] For an excellent overview of how device diversity leads to fragmentation, see Rajapakse (2008).

incompatible user interfaces. As a result, the Symbian market was dominated by three distinct software platforms—S60, UIQ, and MOAP—with different companies viewing their version as a key differentiator. The result was that apps written for one platform would not run on the other platforms. This noninteroperability not only frustrated end users and increased app developers' costs; it also meant that no unified app store could ever develop for Symbian.

The emergence of competition from the iPhone in 2007 signaled the beginning of Symbian's demise. In 2008, Nokia bought out its co-venturers' interests in Symbian and created the Symbian Foundation in an unsuccessful attempt to turn Symbian into a royalty-free open source platform. Symbian's origins as a proprietary operating system made it difficult for it to attract the type of robust user and developer community upon which open source projects depend. In addition, the Symbian Foundation did not release the operating system's source code for another two years. The Symbian Foundation folded shortly thereafter, and Nokia abandoned Symbian in February 2011 for Windows Phone. On June 22, 2011, Nokia outsourced further development of the Symbian operating system to Accenture through 2016 and terminated support for Symbian on January 1, 2014.

Symbian's history offers a number of warning signs for future efforts. First, although support for a wide variety of form factors and screen sizes greatly enhances competition and consumer choice, it also presents significant challenges in terms of fragmentation. Second, left to their own devices, the various Symbian device manufacturers each attempted to use aspects of the operating system and user interface as key differentiators instead of investing in compatibility and the viability of the platform as a whole. The emergence of multiple user interfaces, each with its own mutually incompatible APIs, required app providers to undertake

the effort to port each app for each manufacturers' device, which fragmented the Symbian ecosystem still further.

4.3 Linux

In 1991, Helsinki University student Linus Torvalds released the kernel of a Unix-like operating system that he had developed based on a Unix clone called Minix. His efforts dovetailed perfectly with the effort initiated by former MIT researcher Richard Stallman in 1984 to create a completely open source operating system, which he called GNU (for "GNU's not Unix"). By 1991, Stallman finished the most of the operating system, but was unable to finish the kernel until 1996. Torvalds stepped into the breach by combining the two. In the process, he invited others to join him in improving the kernel and to help him reconfigure utilities created for Minix for the new operating system. Torvalds released the first official version of Linux in 1994.

Linux proved to be a tremendous success. It filled the void left by the collapse of BSD, as former Unix vendors began to shift their emphasis to Linux. That said, Linux has been faced with persistent concerns about forking, often phrased in terms of whether Linux would fall into the same trap as Unix. As of August 2014, hundreds of different Linux distributions existed, and many of them contained different (and incompatible) program "libraries" used in running applications. Calls for reducing the number of Linux distributions were met with criticism from those arguing that the right to experiment with software freely was the essence of the open source movement. The fragmentation of Linux has been mitigated by the rise of for-profit companies such as RedHat and VA Linux, which help users manage the distributions.

Commentators have been struck by the limited extent to which Linux has fragmented.[42] The primary reason is that, in stark contrast to Unix, Linux had a natural leader: Linus Torvalds. As Linux's founder, Torvalds was the natural person to exercise authority over the system. He bolstered his authority by adopting a self-deprecating manner, going to great lengths to document and justify his decisions, and being willing to admit when he is wrong.

When necessary, Torvalds has not been afraid to take action. For example, in 1992, when complaints arose that Fred van Kempen's efforts to incorporate TCP/IP into Linux were taking too long (mostly because of his determination to make it work with all networking protocols and not just TCP/IP), Torvalds sanctioned a parallel coding effort by Alan Cox and ultimately declared Cox's TCP/IP-only solution the winner by admitting it into the core Linux distribution. The episode effectively anointed Cox as Torvalds's de facto lieutenant for networking. Although van Kempen could have forked the code by continuing to work on his version, the developer community remained loyal to Torvalds.

A second threat arose in 1998, when the operator of a mirror site complained that Torvalds was taking too long to accept patches to the code. Torvalds obviated the threat by agreeing to a pyramid structure, which deputized key lieutenants to take the lead in reviewing submissions, while retaining Torvalds as the final authority to resolving disputes. A similar dispute in 2002 led to the creation of additional layer of organizational decisionmaking. Torvalds's status as Linux's creator, the goodwill he earned for his dedication and good judgment in managing the community, and the deft touch he exercised in handling the interpersonal dynamics gave him the authority to prevent major forks from emerging.

---

[42] *See, e.g.*, DiBona et al. (1999, 12); Weber (2004, 158-59).

Thus, although Linux has achieved some success, it does not represent the world of total freedom, bottom-up spontaneous ordering, and technical meritocracy that the collectivist rhetoric surrounding open source might lead people to believe. The history of Linux reveals that prevention of the fragmentation that can have such a devastating negative impact on open source projects was the result of an elaborate system of governance. The result is a process that is quite formal and hierarchical, notwithstanding the fact that participation in any open source project is completely voluntary.

5 Implications for Modern Smartphone Operating Systems

Taken together, the histories of Unix, Linux, and Symbian provide a number of insights into the dynamics surrounding open source operating systems. As an initial matter, the desire to support multiple physical devices increases porting costs and causes a significant risk of fragmentation. In addition, the participation of multiple device manufacturers, each pursuing its own interests, creates additional pressures towards fragmentation. The case studies also illustrate the point made above that the best way to prevent fragmentation is through strong governance. Linux was able to resist these pressures because of the leadership of Linus Torvalds. For Unix and Symbian, the absence of clear leadership led to a more difficult environment for both end users and app developers in terms of systems integration and maintaining a consistent end user experience.

Acknowledging the propriety of some form of governance leaves open the question of how much governance is appropriate. The spirit of open source requires that any governance regime leave substantial room for experimentation. In addition, the voluntary nature of open source projects and the example set by Linus Torvalds both counsel in favor of asserting as light a touch as possible. The real question is not whether some actor should have been allowed to

exercise some degree of guidance over the platform, but rather how much and what type of governance should be considered a reasonable step to ensure that the mobile operating system achieves its potential.

5.1 The Role of Hardware Diversity

The Symbian experience teaches us that hardware diversity is an important source of fragmentation. Although a broad selection of phones creates real benefits to end users and can enhance competition, hardware variations create significant differences in operating context in terms of screen parameters (size, color depth, orientation, aspect ratio), memory size, processing power, input devices (keyboard, touch screen, etc.), cameras, and connectivity options (WiFi, Bluetooth, Infrared, Global Packet Radio Service), just to name a few.[43]

5.2 The Need for Testing and Governance

Apart from the differences in hardware platforms, third-party provisioning and the divergence of incentives requires that end users must have some means for verifying that components comply with the design. As noted above, some means for testing devices is necessary to have functioning markets for third-party provision and for end users to know that they are getting what is promised. In most cases, the incentive structure and social norms surrounding open source projects discourage module creators from deviating from the architecture or creating incompatible forks. That said, the desire to reduce costs by omitting certain APIs or other features suggests the existence of circumstances that can lead participants in the platform to deviate from the architecture. The possibility exists that the cost of generality

---

[43] Rajapakse (2008).

and the incentives to make noncompliant devices may become sufficiently strong that device manufacturers cannot be expected to abide by the honor system. This danger makes some system for verification essential. Requiring end users to conduct such tests would be burdensome and unnecessarily duplicative. The result is that some degree of provider-based testing appears inevitable.

Within provider-based testing, a range of alternative approaches to testing and governance exist, each one varying in terms of restrictiveness. (1) The most restrictive approach is for the platform sponsor to manufacture all of its own smartphones. While this approach gives the platform complete authority to ensure compatibility, this approach imposes limits on the variety of hardware devices and on competitive entry. (2) A less restrictive approach would permit third parties to produce devices, but require that manufacturers to submit their devices to the platform sponsor for certification and testing. While more permissive than the first option, this approach risks giving the platform sponsor gatekeeper control over all devices. (3) An even less restrictive approach would provide a compatibility standard along with open testing tools for device manufacturers to self-certify that their devices comply with that standard. This approach provides device manufacturers the most flexibility and offers the greatest benefits in terms of variety of devices and ease of competitive entry. (4) In addition, a platform sponsor may adopt a hybrid approach that gives module creators a choice between (2) and (3).

In addition to testing regimes to ensure compatibility, open source projects must rely on some form of governance to prevent fragmentation and forking. A nonexclusive requirement to maintain compatibility would seem to be the least restrictive approach.

The Android platform that is the current subject of antitrust scrutiny generally falls within the last and least restrictive of these options. The following, more detailed review of Android's

licensing practices reveals that within this regime, device manufacturers can choose among a range of possible licensing alternatives.

5.2.1 The Android Open Source Project (AOSP) License Agreement

The least restrictive option is to license the Android Open Source Project (AOSP) source code without making any commitments as to the modification or implementation of the code. The software is royalty free under an Apache open source license, although some hardware is subject to patent licenses. The Apache licenses ensure that device manufacturers remain free to modify the source code as they see fit. They also remain free to produce other devices using other operating systems if they so choose.

The most prominent provider to go this route is Amazon, which has created its own operating system known as Fire OS, which is based on AOSP and runs the Amazon's Kindle and Fire Phone. Other prominent examples include Nokia's X platform and the open source CyanogenMod operating systems, among others. Samsung is attempting to avoid Android altogether by basing its new Tizen operating system on the original Linux kernel.

The Android license agreement for AOSP places device manufacturers under no obligations to carry any Android apps and leaves them free to add whatever apps they choose. Because there are no restrictions on the level of customization, the resulting devices may not be compatible with apps written for other Android devices.

5.2.2 The Compatibility Definition Document (CDD) and the Compatibility Test Suite (CTS)

The second level of compatibility is for a device manufacturer to guarantee interoperability by ensuring that its device satisfies a published compatibility standard. The

compatibility standard for each version of Android is embodied in a Compatibility Definition Document (CDD). Google also provides a free Compatibility Test Suite (CTS) that device manufacturers may use to determine whether their device is compatible. The goals of the CDD are to:

- Provide a consistent application and hardware environment to application developers
- Enable a consistent application experience for end users
- Enable device manufacturers to differentiate while being compatible.
- Minimize costs and overhead costs of compatibility.[44]

Devices that comply with the CDD must include nine core applications: Desk Clock, Browser, Calendar, Contacts, Gallery, GlobalSearch, Launcher, Music, and Settings. These applications tend to provide basic functions on which other applications draw, so their presence provides a consistent set of resources on which the app developer community can draw. Device manufacturers can satisfy this requirement either by using the versions of these apps provided by Google or by providing their own apps so long as they satisfy the interoperability requirements. The CDD also requires that the device include a complete set of Android APIs and Android developer tools to ensure that the device will operate properly. Device manufacturers remain free to develop and distribute their own APIs in addition to those required by the CDD.

The goal is to create a baseline of interoperability that helps app developers by creating a stable set of resources and by eliminating the porting costs/multiple versions for different builds, while at the same time preserving a degree of flexibility. Devices that demonstrate compliance with the CDD by passing the CTS are regarded as Android compliant. Importantly, device manufacturers may choose to comply with the CDD without signing any agreements.

---

[44] Android (n.d.).

It bears noting that mandatory apps do not include any Google proprietary apps or apps alleged to be give rise to market power by competition regulators. CDD compliant devices are under no obligation to install any of these services and remain free to include apps that compete directly with these services. Device manufacturers may also use whatever search service they would like.

5.2.3 The Anti-Fragmentation Agreement

Device manufacturers who would like greater certification of compatibility can sign the Anti-Fragmentation Agreement (AFA), which was first created in early 2008, when Android was nascent and one year prior to the launch of the first Android smartphone. The AFA requires signatories to promise (1) that all Android devices it makes will fulfill the CDD requirements and (2) not to take any actions that may cause the fragmentation of Android.

As noted above, the first provision, requiring that all Android devices made by the signatory fulfill the CDD requirements, only requires the installation of basic services, such as Desk Clock, Browser, Calendar, Contacts, and Settings. It does not prevent device manufacturers from substituting their own versions of the required apps so long as they pass the compatibility test. The CDD also does not place any restrictions on the device manufacturer's ability to market non-Android devices (i.e., devices based on other operating systems, such as Windows Phone, Blackberry, or Linux).

The second provision, prohibiting device manufacturers from taking any actions that would fragment Android, is effectively a reiteration of the first provision in that it prohibits manufacturers from creating Android phones that do not comply with the CDD. The rationale is that permitting device manufacturers to sell both CDD-compliant and CDD-noncompliant Android phones can increase app developers' costs by requiring them to port their apps to

multiple platforms and can create potential confusion among consumers over which phones are Android compliant and which ones are not.

Signatories that satisfy these requirements are eligible to declare their devices to be "Android Compatible Devices." The AFA requires pre-installation of only those basic apps included in the CDD (Desk Clock, Browser, Calendar, Contacts, Gallery, GlobalSearch, Launcher, Music, and Settings). As noted above, AFA signatories can use any version of these apps (Google's, their own, or a version provided by a third party) so long as they fulfill the basic functions. AFA signatories can also benefit from additional technical support in the form of information about upcoming Android features, new APIs, Android security and performance, and new form factors. as well as assistance to patch bugs, address CTS failures, and implement new features.

Together, these provisions represent a fairly unrestrictive form of governance that ensures a minimum level of compatibility and interoperability across Android devices. Importantly, the compatibility requirements covered by the AFA refer only to APIs and basic apps and do not contain any obligations with respect to Google Mobile Services (GMS) suite of apps, such as Google Play, YouTube, Maps, and Gmail, that have been the primary source of regulatory concern. In essence, the AFA enables signatory device manufacturers to join together in a partnership committed to promoting a particular version of the Android open source project by creating mutually compatible devices and limited sharing information.

5.3 Safety Valves

The terms of the AFA contain a number of features that make it much less likely that its terms can properly be regarded as problematic. As an initial matter, the basic licensing

agreement for AOSP, the apps that the CDD requires to be installed, the CTS tool for evaluating compliance, and the AFA are all royalty free.

In addition, the AFA is nonmandatory: many device manufacturers (especially in China) opt to comply with the CDD without signing the AFA. Moreover, both the CDD and the AFA are nonexclusive in that device manufacturers can market any non-Android devices (see, e.g., Samsung and ZTE) and can substitute their own apps or otherwise customize the hardware and software so long as they comply with the CDD (see, e.g., Xiaomi, Huawei, ZTE, and HTC).

Finally, the presence of meaningful market options makes it unlikely that either the CDD or the AFA will harm competition. Because Android's is based on open source, any device manufacturer that is unwilling to comply with the CDD or sign the AFA remains free to produce its own version of Android.

Interestingly, both Apple and Microsoft are invoking Android's greater supposed vulnerability to fragmentation as a potential reason to buy iOS or Windows Phone instead of Android. This suggests that fragmentation is a product feature on which various operating system providers are competing. Limiting any of these companies' ability to manage fragmentation would place artificial limits on product features and would reduce one dimension of competition.

Indeed, focusing undue regulatory attention on open systems may have unfortunate unintended consequences. If antitrust scrutiny restricts providers of open platforms from using less restrictive governance mechanisms to protect against fragmentation and noninteroperability, those providers may well be left with no choice but to adopt more restrictive alternatives. Specifically, adopting too restrictive a stance on the use of agreements like the AFA to limit fragmentation may force mobile operating systems seeking to avoid fragmentation either to

adopt Apple-style vertical integration or to require that all device manufacturers submit their phones for testing. This would effectively bar mobile operating system providers from employing the least restrictive of these three alternatives. If so, this would reduce the diversity and competitiveness of mobile operating systems and would substantially increase the barriers to entry for OEMs, developers, and other platform participants.

In short, all of the approaches have advantages and disadvantages that appeal to different types of consumers. In addition, providers face a considerable amount of uncertainty over the best way to strike the proper balance the benefits to innovation associated with flexibility and concerns about fragmentation. This underscores the error in regarding the selection of the ideal governance regime as an either-or choice. On the contrary, end users benefit the most by being able to choose among different options that are exploring different approaches to preventing fragmentation.

6 A Brief Comment on User Interfaces and the MADA

Although this analysis focuses primarily on the relationship between open source software and modularity theory and its implications for the AFA, I thought I would offer a few words about the other principal governance instrument for Android: the Mobile Applications Distribution Agreement (MADA).[45] MADAs typically require device manufacturers to preload all of the apps contained in GMS, including Play, YouTube, Maps, and Gmail. MADA also require that the Google Search widget and the Play icon be accessible with at most one phone tap away from the home screen and that Google Search be the default for in-app searches, although

[45] Although the terms of specific MADA are confidential, two MADA were made public as part of the record in *Oracle America, Inc. v. Google Inc.*, 872 F. Supp. 2d 974 (N.D. Cal. 2012).

the MADA does not require that Google Search be the default search engine for the web browser. MADA signatories may also use the Android green robot trademark. All apps are provided royalty free except for patent royalties imposed by outside parties. Device manufacturers are free to preload their own versions of these apps alongside the Google versions.

Unlike the AFA, the MADA is not designed to address the problems of fragmentation. Instead, it is designed to address another major weakness of open source systems: the difficulty in providing a consistent user interface.

Open source is often described as "hackers writing for hackers."[46] It is known to function best in horizontal domains where there is widespread agreement on the design architecture and the general shape of the software requirements is well known and not problematic. It is less effective in vertical domains where requirements are a function of domain specific knowledge acquired over time. The sparse documentation and field support has long made open source better suited for experts operating on the server side than for end users.[47]

Because of these qualities, open source projects have faced particular difficulty with end-user interfaces. The design of end-user interfaces is dominated by tacit information that is hard to modularize. Moreover, the acquisition of this information is associated with focus groups and physical contact, which contrasts starkly with the email chains that characterize open source projects.[48]

As a result, it comes as no surprise that Unix, Symbian and Linux have often been characterized as "esoteric and hard to use."[49] In particular, Unix was criticized for its lack of a

---

46 Weber (2004, 237-38).
47 Feller and Fitzgerald (2002, 175-76).
48 Feller and Fitzgerald (2002, 132, 175); Weber (2004, 237-38).
49 Feller and Fitzgerald (2002, 22).

friendly interface easily implementable by a nontechnical person.[50] Symbian suffered from similar problems. As noted earlier, different groups of manufacturers each made their own proprietary enhancements to Symbian, which in turn created three distinct Symbian software platforms (S60, UIQ, and MOAP), each with its own user interface. The result was an end-user interface that was quite inconsistent and did not provide app developers with a consistent platform for which to design their products.[51] Linux users must choose from dozens of end-user interfaces, although Linux is attempting to standardize around the K Desktop Environment (KDE) and GNU Network Object Model Environment ("GNOME").[52] These challenges have prevented Linux from achieving significant penetration into the home desktop and laptop market.

While the diversity of form factors, screen sizes, and user interfaces among Android devices yields many benefits, the end-user experience also varies widely from device to device. Android users must adapt to every new device. In contrast, the consistency of the end-user interface is often regarded as one of the primary advantages of the Apple iPhone.

The fragmentation of end-user interfaces poses difficulties for the developer community. The variations in code and format often force them to create versions for each environment in which they would like to operate, increasing their costs and making it more difficult for them to enter new markets.

The ideal solution would be to introduce some element that permits greater consistency to the end-user experience without losing the benefits of product diversity and entry by new firms associated with an open source environment. The MADA represents one attempt to strike a balance between these two competing considerations. Requiring signatories to use specific

---

50 Korzeniowski (1985); McKusick (1999, 56).
51 Gilson (2012).
52 Feller and Fitzgerald (2002, 22); Weber (2004, 102-03, 238).

Google-provided apps helps unify the end-user experience. The goal is to provide end users with a consistent baseline of out-of-the-box functionality. At the same time, the GMS apps provide a more consistent platform for the app community.

The need to remedy the lack of a consistent end-user experience that has long plagued the open source operating system provides a strong justification for the type of restrictions contained in the MADA. The fact that the apps required by the MADA are royalty free and nonexclusive reduces the likelihood of any anticompetitive effects. Most importantly, preventing open source projects from using agreements like the MADA to address this key weakness would leave them incentives to assert more direct control over end-user interfaces by adopting policies that place more restrictions on device manufacturers and app providers. Thus, overly vigorous antitrust oversight imposed in the name of promoting competition and protecting consumers runs the risk of actually reducing competition and consumer choice.

7 Conclusion

Open source operating systems thus something of a conundrum. On the one hand, open source requires that developers have absolute freedom to modify the software as they see fit. On the other hand, the software must obey certain architectural rules if it is to serve as a platform that can bring together different types of hardware and applications. The flexibility inherent in open source can lead to incompatibility. In extreme cases, it can even cause the open source project to fork into two or more different branches.

Such fragmentation dissipates the economic benefits of being able to access a large customer base through a single platform and forces app developers to expend the cost to make their products compatible with multiple versions of the operating system. One classic solution to these problems is to rely on some form of testing to ensure that the components provided by third

parties are configured to comply with a compatibility standard. Another is to subject the overall system to some form of governance. Although both alternatives may seem to be somewhat inconsistent with the philosophy of open source, the academic literature indicates that both are a necessary aspect of any modular platform in which multiple parties provide separate components. The question is thus not whether such restrictions must exist, but rather how restrictive they have to be.

The history of the three leading open source operating systems (Unix, Symbian, and Linux) confirms this insight. Moreover, an approach that permits third parties to self-certify represents the least restrictive way to implement such requirements. Any restrictions are also less likely to be problematic if they are royalty-free, nonexclusive, and open source. It thus appears that solutions such as Google's Anti-Fragmentation Agreement represent one way to strike a reasonable balance between ensuring that the operating system serves as a platform that brings together mobile devices and applications in a way that promotes the ability to "write once, run anywhere" and giving device manufacturers and app developers as much flexibility as possible. Given the lingering uncertainty about the best way to balance these concerns, end users and technological progress would best be served by giving operating system providers considerable latitude in determining the best way to promote freedom without creating undue risks of fragmentation.

References

Android. n.d. Compatibility Program Overview. https://source.android.com/compatibility/overview.html.

Baldwin, Carliss Y., and Kim B. Clark. 2000. *Design Rules: The Power of Modularity* Cambridge, MA: MIT Press.

Best, Jo. 2013. "'Android before Android': The long, strange history of Symbian and why it matters for Nokia's future." *ZDNet*. http://www.zdnet.com/article/android-before-android-the-long-strange-history-of-symbian-and-why-it-matters-for-nokias-future/

Bezroukov, Nikolai. 1999. "Open Source Software Development as a Special Type of Academic Research (Critique of Vulgar Raymondism)." *First Monday* 4(10). http://firstmonday.org/article/view/696/606.

Bonaccorsi, Andrea, and Cristina Rossi. 2003. "Why Open Source Software Can Succeed." *Research Policy* 32(7): 1243-58.

Brooks, Frederick P. 1975. *The Mythical Man-Month: Essays on Software Engineering* Boston, MA: Addison-Wesley Professional.

——— 1995. *The Mythical Man-Month: Essays on Software Engineering,* 20th Anniversary Edition. Reading, MA: Addison-Wesley.

Clark, David D. 1982. "Modularity and Efficiency in Protocol Implementation." Network Working Group Request for Comments 817. http://tools.ietf.org/pdf/rfc817.

Clark, Kim, and Carliss Y. Baldwin. 2006. "The Architecture of Participation: Does Code Architecture Mitigate Free Riding in the Open Source Development Model." *Management Science* 52(7): 1116-27.

Conway, Melvin E. 1968. "How Do Committees Invent?" *Datamation* 14(4): 28-31.

Corbett, Jonathan. 2011. "Android, forking, and control." https://lwn.net/Articles/446297/.

DiBona, Chris, Sam Ockman, and Mark Stone. 1999. "Introduction." In Chris DiBona, Sam Ockman, and Mark Stone, eds., *Open Sources: Voices from the Open Source Revolution,* 8-15. Sebastopol, CA: O'Reilly Media.

Dijkstra, Edsger W. 1968. "The Structure of the 'The'-Multiprogramming System." *Communications of the ACM* 46(11): 341, 343.

Feller, Joseph, and Brian Fitzgerald. 2002. *Understanding Open Source Software Development.* Boston, MA: Addison-Wesley Professional.

Ghosh, Rishab, and V. V. Prakash. 2000. “The Orbiten Free Software Survey.” *First Monday* 5(7). http://www.firstmonday.org/ojs/index.php/fm/article/view/769/678.

Gilson, David. 2012. “The History of Symbian’s Secret Fragmentation.” *All About Symbian*. http://www.allaboutsymbian.com/features/item/14405_The_History_of_Symbians_Secret.php.

Korzeniowski, Paul. 1985. “Users Laud UNIX Portability, Call Flexibility a Weakness.” *Computerworld* 11.

Libicki, Martin C. 1995. *Information Technology Standards: Quest for the Common Byte*. Newton, MA: Digital Press.

Lerner, Josh, and Jean Tirole. 2002. “Some Simple Economics of Open Source.” *Journal of Industrial Economics* 50(2): 197-234.

Maclachlan, Malcolm. 1999. “Panelist Describe Open Source Dictatorships.” *TechWeb.com.* http://web.archive.org/web/20060313204003/http://www.techweb.com/wire/story/TWB19990812S0003.

Marella, Fabrizio, and Christopher S. Yoo. 2007. “Is Open Source Software the New Lex Mercatoria?” *Virginia Journal of International Law* 47807–36.

McGee, W. C. 1959. “Generalization: Key to Successful Electronic Data Processing.” *Journal of the ACM* 6(1): 1-23.

McGowan, David. 2001. “Legal Implications of Open-Source Software.” *University of Illinois Law Review* (1) 241-304.

McKusick, Marshall Kirk. 1999. “Twenty Years of Berkeley Unix: From AT&T-Owned to Freely Redistributable.” In Chris DiBona, Sam Ockman, and Mark Stone, eds., *Open Sources: Voices from the Open Source Revolution*, 31-46. Sebastopol, CA: O’Reilly Media.

Midha, Vishal, and Prashan Palvia. 2012. “Factors Affecting the Success of Open Source Software.” *Journal of Systems and Software* 85(4): 895–905.

Mockus, Audris, Roy Fielding, and James Herbsleb. 2000. “A Case Study of Open Source Software Development: The Apache Server.” *Proceedings of the 22nd International Conference on Software Engineering* 263-72.

Open Source Initiative. 2007. The Approved Licenses. http://www.opensource.org/licenses.

Parnas, D. L. 1972. "On the Criteria To Be Used in Decomposing Systems into Modules.” *Communications of the ACM* 15(12): 1053–58.

Perens, Bruce. 1999. “The Open Source Definition.” In Chris DiBona, Sam Ockman, and Mark Stone, eds., *Open Sources: Voices from the Open Source Revolution*, 79-86. Sebastopol, CA: O’Reilly Media.

Raymond, Eric S. 1998. “Homesteading the Noosphere.” *First Monday* 3: 10.2. http://www.firstmonday.dk/issues/issue3-10/raymond/.

———. 1999. *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. Sebastopol, CA: O’Reilly Media.

Simon, Herbert A. 1962. “The Architecture of Complexity.” *Proceedings of the American Philosophical Society* 106(6): 467-482.

Skyhook Wireless, Inc. v. Google, Inc., 30 Massachusetts Law Reporter 417 (Super. Ct. 2012).

Stallman, Richard. 1999. “The GNU Operating System and the Free Software Movement.” In Chris DiBona, Sam Ockman, and Mark Stone, eds., *Open Sources: Voices from the Open Source Revolution*, 31-38. Sebastopol, CA: O’Reilly Media.

Torvalds, Linus. 1999. “The Linux Edge.” In Chris DiBona, Sam Ockman, and Mark Stone, eds., *Open Sources: Voices from the Open Source Revolution*, 101-11. Sebastopol, CA: O’Reilly Media.

van Rossum, Guido. 2008. “Origin of BDFL.” All Things Pythonic Weblogs. http://www.artima.com/weblogs/viewpost.jsp?thread=235725.

Weber, Steven. 2004. *The Success of Open Source*. Cambridge, MA: Harvard University Press.

Yamagata, Hiroo. 1997. The Pragmatist of Free Software: Linus Torvalds Interview. http://www.tlug.jp/docs/linus.html.

Yoo, Christopher S. 2016. “Modularity Theory and Internet Policy.” *University of Illinois Law Review* (1): 1–62.