CHAPTER

Platform Architecture 5

Each person making . . . four thousand eight hundred pins in a day. But if they had all wrought separately and independently . . . not each of them have made twenty, perhaps not one pin in a day.
Adam Smith in *The Wealth of Nations* (1776)

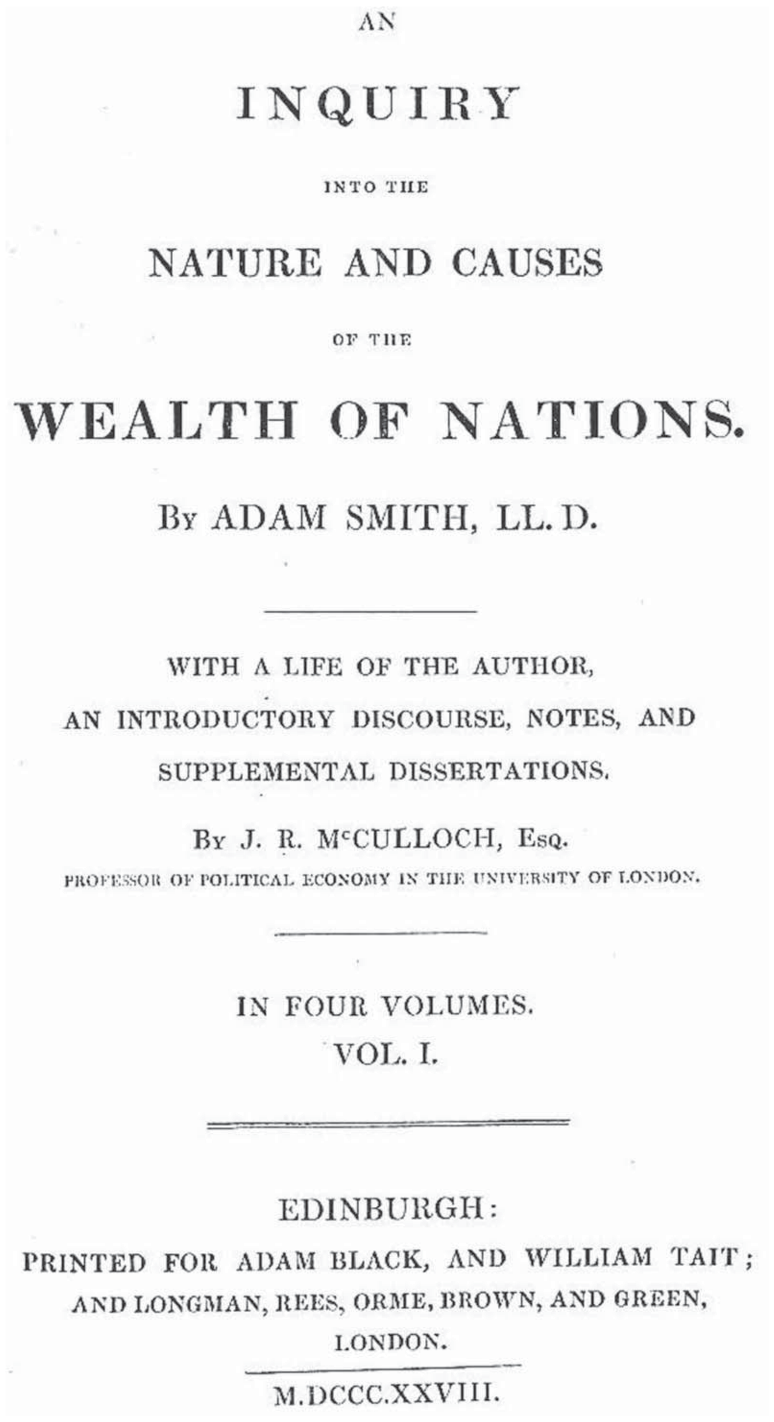
IN THIS CHAPTER

- How complexity paralyzes innovation in ecosystems
- How architectures reduce complexity
- Two functions of architecture: partitioning and reintegration
- Two facets of ecosystem architecture: platform architecture and app microarchitecture
- Four desirable properties of platform architectures
- Modularization
- Two mechanisms for modularizing architectures: decoupling and interface standardization
- Simple rules for what goes into the platform and what stays out
- Tradeoffs in architectural choices

5.1 HOW UNEMPLOYED HAIRDRESSERS BECAME FRANCE'S MATHEMATICAL CHAMPIONS

The inspiration for solving an innovation problem too complex for anyone to grasp comes from an unlikely source: unemployed hairdressers in 18th-century France. Gaspard de Prony (Figure 5.2) was a civil engineer who the French revolutionary government charged with a most unenviable task in 1790: create the largest, most precise set of trigonometric tables ever created (Langlois and Garzarelli, 2008). Baffled by this seemingly impossible task, de Prony was absent-mindedly flipping through Adam Smith's *Wealth of Nations* (Figure 5.1) in a bookstore when he was hit by an epiphany. He could "manufacture" trigonometric tables like Adam Smith's workers manufactured pins in a pin factory, by division of labor. Except, it was going to be division of cognitive labor rather than physical labor. So de Prony began in earnest. He first recruited four of the most eminent French mathematicians—Legendre, Prieur, Cote d'Or, and Carnot—to devise formulae that could be numerically calculated. He then passed along these formulae to about a dozen average mathematicians, who

☆"To view the full reference list for the book, click here or see page 283."



AN

INQUIRY

INTO THE

NATURE AND CAUSES

OF THE

WEALTH OF NATIONS.

By ADAM SMITH, LL. D.

WITH A LIFE OF THE AUTHOR,
AN INTRODUCTORY DISCOURSE, NOTES, AND
SUPPLEMENTAL DISSERTATIONS.

By J. R. McCULLOCH, Esq.
PROFESSOR OF POLITICAL ECONOMY IN THE UNIVERSITY OF LONDON.

IN FOUR VOLUMES.
VOL. I.

EDINBURGH:
PRINTED FOR ADAM BLACK, AND WILLIAM TAIT;
AND LONGMAN, REES, ORME, BROWN, AND GREEN,
LONDON.

M.DCCC.XXVIII.

FIGURE 5.1

Adam Smith's *Wealth of Nations* inspired de Prony's creation of trigonometric tables in 1790.

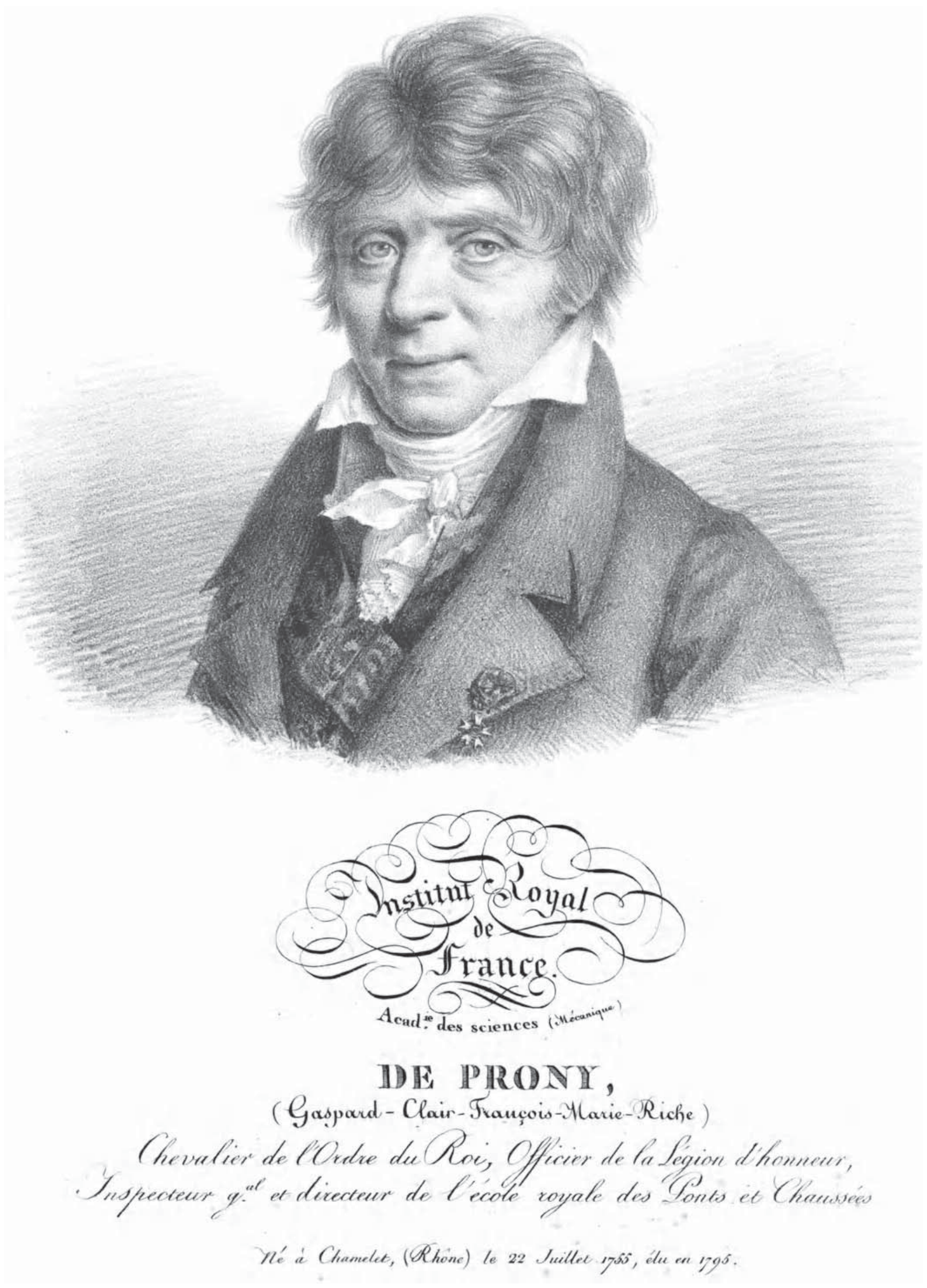

Institut Royal de France.
Acad.ie des sciences (Mécanique)
DE PRONY,
(Gaspard-Clair-François-Marie-Riche)
Chevalier de l'Ordre du Roi, Officier de la Légion d'honneur,
Inspecteur g.al et directeur de l'école royale des Ponts et Chaussées
Né à Chamelet, (Rhône) le 22 Juillet 1755, élu en 1795.

FIGURE 5.2

Gaspard de Prony (1755–1839).

Courtesy of the Smithsonian Institution Libraries, Washington, D.C. Reproduced with permission.

turned them into simple algorithms and created templates of tables to be filled by hand. Filling these tables required little knowledge of math beyond addition and subtraction. Each table could be filled without any knowledge of the other tables. It was simple grunt work at this point.

France in the late 1700s was like the United States after 2008: excesses were frowned upon, a recession was in full swing, and austerity was expected in high society. This gave de Prony the perfect cadre of workers to do the grunt work: about a hundred unemployed French hairdressers. There was no communication from the hairdressers to the mathematicians, and no feedback from the mathematicians to the hairdressers. In four years, this team of hairdressers was producing close to a thousand results a day.

de Prony took an innovation problem too complex for one person to wrap his head around and made it solvable by partitioning it into smaller, independent problems that required little knowledge of the other pieces. Doing this required little central coordination and little ongoing communication, and the individual hairdressers' output could be plugged back into the slowly forming book of tables. de Prony's ingeniousness was in how he partitioned the complex innovation problem into small, independent problems. The magic was in how they could then be reassembled to form the whole. The same approach can also work in organizing innovation in platform ecosystems. The key to this is platform architecture, the focus of this chapter.

Platform architecture is the first gear in a platform ecosystem's evolutionary motor (Figure 5.3). A platform requires an "architecture of participation" to grow its ecosystem (Baldwin and Clark, 2006). Outside app developers must simultaneously be *able* to and be *motivated* to innovate around it. Ability without motivation is as worthless as motivation without ability. Creating the ability is the realm of platform architecture. Platform architecture determines the divisibility of innovation work among app developers and the platform owner. It also influences its subsequent reintegration. Platform architecture is the blueprint. Creating motivation is the realm of platform governance (Chapter 6). Evolution of a platform's ecosystem is therefore predicated in the interplay of its architecture with how it is governed.

This chapter provides a foundation for understanding platform architectures. It begins by explaining how complexity stymies innovation and leads to an unrealistic optimism bias in platform ecosystems. It then describes how a platform's architecture can make growing complexity manageable by serving two functions: partitioning innovation tasks and facilitating reintegration of an ecosystem's parts. It explains how architecture is a platform's DNA that preordains viable organizing logics and irreversibly imprints its evolutionary trajectory. What appears to be a technical decision has huge strategic consequences. A platform's properties are inherited by apps in their own architectures, but imperfectly.

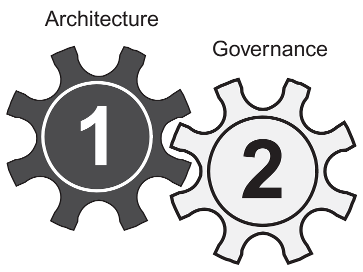


FIGURE 5.3

Architecture is the first gear in a platform ecosystem's evolutionary motor.

We explain the connections between a platform's architecture and the "microarchitecture" of apps (a microscopic view) as well as ecosystem architecture (a telescopic view).

A good architecture must exhibit four simple properties that it shares with the architecture of modern cities: simplicity, resilience, maintainability, and evolvability. We also explain the two mechanisms for modularizing architectures along with practical guidelines to implement them. We also revisit Goldilocks, who cautions that you neither want too little nor too much modularity but something in the middle. We explain this by putting ourselves in the shoes of a platform owner and an app developer, and offer guidelines to help you figure out how much modularity is *just right*. Finally, we segue into platform governance (the topic of the next chapter), which influences the degree to which the potential advantages of thoughtful platform architecture are realized in practice.

5.2 COMPLEXITY: THE ACHILLES HEEL OF PLATFORMS

A platform ecosystem can be envisioned as a complex system. Broadly, a complex system is one comprised of a number of parts that have many unpredictable interactions (Simon, 1962). It is comprised of smaller subsystems whose interactions and interdependencies are difficult to describe and manage (de Weck et al., 2011, p. 186). An ecosystem's complexity is a function of the number of unique subsystems present in it. The more numerous such subsystems, the greater its complexity. In a platform ecosystem, these subsystems are the platform and the apps that interoperate with it. Complex systems that were complex to begin with can become even more complex over time as they evolve.

Complexity is the Achilles heel—a potentially deadly weakness—of platforms. Complexity creates two challenges that worsen over time: incomprehensibility and a gridlock. First, platform ecosystems become increasingly difficult to be comprehensible in their entirety to one person (Baldwin and Clark, 2000, p. 5; de Weck et al., 2011, p. 27). They often stretch the ability of a human mind to grasp their complexity. Platform architects therefore quickly become unable to comprehend the technology that they invent. App developers face a similar challenge because they become increasingly unable to comprehend the complexity of the ecosystem in which their apps must function. Second, it creates a gridlock problem. Managing such complexity can become so daunting that it can paralyze any one ecosystem participant's ability to change a subsystem for which she is responsible. This is because a slight change in one app or in the platform can have unpredictable ripple effects that can potentially break the ecosystem. Like in a house of cards, moving one might do nothing bad or it might bring the entire structure down. The solution to the challenges created by growing complexity is to reduce it.

Remember the old fable of the elephant and the eight blind men (Figure 5.4A). Each was asked, "What do you see here?" Each touched the elephant and drew different conclusions. One concluded that it was a rope, another said it was a wall, another thought it was a fan and another concluded that it was a spear. A platform ecosystem often faces the same problem: Each app developer sees a different image of the ecosystem when looking at the whole from her own perspective. Now look at a different picture (Figure 5.4B). Would the eight blind men be any closer in their interpretations of that object? Likely. The bottom line is that more complex a man-made object gets, the harder it becomes to comprehend for any one individual. This incomprehensibility can become the showstopper in a platform's evolvability.

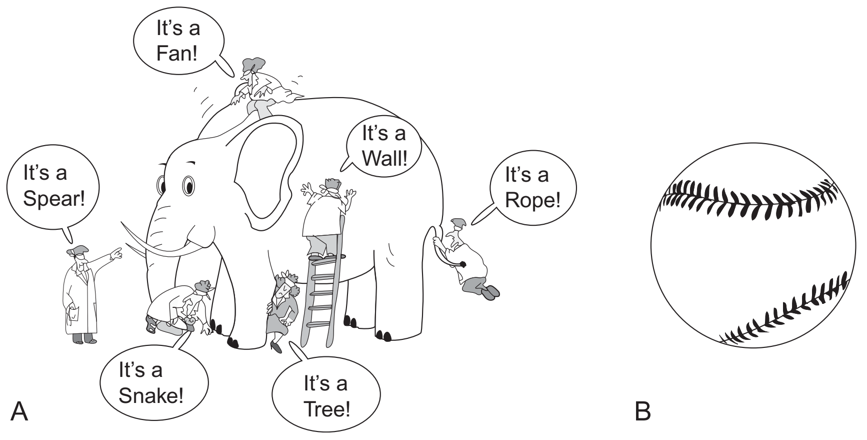


FIGURE 5.4

What do you see in these pictures (A) and (B)?

5.2.1 Two types of complexity

Complexity can be of two types: structural complexity and behavioral complexity. An ecosystem can be both structurally complex (interconnections between its parts are difficult to describe) and behaviorally complex (its aggregate behavior is difficult to predict or control) (de Weck et al., 2011, p. 185). The two types of complexities are often (but not always) correlated: Systems that are structurally complex are also often behaviorally complex. We believe that architecture is the lever to tackle structural complexity and governance (Chapter 6) is the lever to tackle behavioral complexity. Architecture is therefore a tool for simplifying and precisely describing the interconnections between parts of an ecosystem—potentially reducing structural complexity. It does this by reconfiguring the structure of dependencies between the platform and its apps within an ecosystem.

5.2.1.1 How complexity amplifies innovation risk in platforms

Structural complexity matters for innovation in ecosystems because it magnifies what Ron Adner (2012, p. 49) calls co-innovation risk. A useful way to think of dependencies is to compare the difference between joint and independent probabilities (Adner, 2012, p. 48). Consider a simple example in Figure 5.5 where two app developers and the platform owner must contribute to, say, creating a novel smartphone app. If each has an 80% chance of being able to deliver their part, the likelihood of successfully implementing the app is far lower than 80%. This is because realizing an innovation with dependencies among the three parties is governed by joint, not independent, probabilities. The likelihood that the app will succeed is a pitiful 51% ($80\% \times 80\% \times 80\%$). This means that although the three partners are fairly likely to be able to deliver on their promises, the odds of success in their joint endeavor are about 50–50. It is easy for each partner to be confident about the success of the project, when the reality is bleaker than any one of them might realize. As the number of parties involved increases, so does co-innovation risk. But co-innovation risk

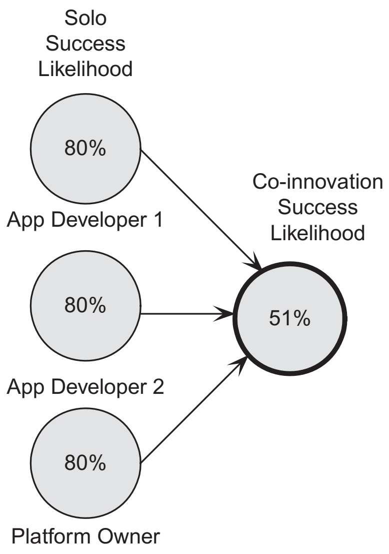


FIGURE 5.5

Co-innovation risk is magnified by complexity.

exists only if the work of one developer relies on the successful completion of the work of the other two in Figure 5.5. A powerful way to reduce this risk is by lowering the dependencies between the contributions of the three parties in this example. Reducing dependencies reduces interactions among them, hence reducing the structural complexity of the system in Figure 5.5. Architecture is a way to reduce such dependencies among subsystems that constitute a complex system. Thoughtful architectures however are not a silver bullet: They cannot eliminate complexity but can make it more manageable.

5.3 THE TWO FUNCTIONS OF ECOSYSTEM ARCHITECTURE

The biggest potential strength—and the biggest weakness—of platform ecosystems is their diversity, both in apps that constitute them and the app developers that develop such apps. Architectural choices by platform owners that quash the autonomy of app developers kill innovation potential. On the other hand, the same architectural choices can also fail to leverage this diversity into a cohesive platform, resulting in unfettered chaos. Platform ecosystems must therefore manage the delicate balance between coordination and autonomy (Iansiti and Levien, 2004, p. 5). Architecture is a tool for balancing the need for autonomy among app developers without compromising the capacity to integrate their work into a cohesive ecosystem.

Architecture, however, lurks in the blind spot of most strategy thinking, largely because it is perceived as the outcome of a technical decision-making process. Two types of costs directly stemming from platform architecture and with strategic consequences enter the picture: (1) the costs incurred by app developers of doing business with platform owners (“transaction costs”)[1] and (2) the costs incurred by them to manage the dependencies between apps and the platform (“coordination costs”). If both parties were part of the same organization as was historically the case in traditional software development teams, these costs would have been lower. However, the moment app developers and platform owners become standalone organizations, these costs can overwhelm any prospect of cost-effective collaboration among them. (They tend to be higher across organizations than inside organizations.) Transaction costs and coordination costs among a platform owner and app developers determine whether a multi-organization ecosystem is even viable around a platform. A well-designed architecture can minimize both types of costs (Baldwin and Woodard, 2009). Architecture in a platform ecosystem ought to serve two overarching functions: partitioning and systems integration.

5.3.1 Partitioning

Partitioning refers to a decomposition of the ecosystem such that each subsystem in it is relatively autonomous from others (Figure 5.6). The primary function of architecture is to provide a framework to decompose a complex ecosystem into relatively independent subsystems. When designers envision a complex system, they quickly hit human cognitive limits due to its complexity. One way to reduce this complexity and make it cognitively manageable is to decompose the complex system into smaller pieces—platform and apps—that can work together to deliver the desired functionality. So, a complex system can be broken down into a collection of black boxes that do specific things. Such decomposition can continue ad infinitum where a complex ecosystem is decomposed into smaller and smaller subsystems. However, it is useful to stop this decomposition exercise when further decomposition no longer aids comprehensibility or no longer enhances autonomy among contributors to a platform ecosystem. When an element in a subsystem can no longer be decomposed meaningfully, it is said to have reached an *atomic* level of decomposition.

A well-partitioned architecture describes how these little black boxes behave and talk to each other, but not how they work. What happens inside a black box remains inside that black box. Once this collection of black boxes is envisioned, each of them can be designed and implemented individually with the hope that when they are all pieced together the much larger complex system will emerge. Unless the

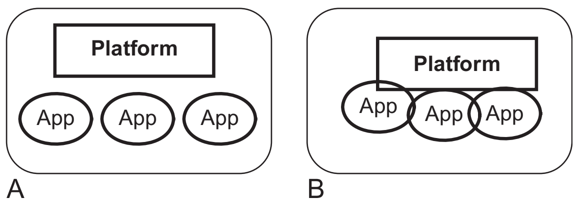


FIGURE 5.6

A well-partitioned architecture decomposes the platform and apps into relatively autonomous subsystems (A, not B).

[1]Transaction costs are simply the overhead costs of doing a trade in the open market between two parties that might not have the same interests. They have a long tradition in the history of economics (see Baldwin, 2008; Williamson, 1987, 1991, 1999, 2010) and technology strategy (see Ang and Straub, 1998; Tiwana and Bush, 2007; Young-Ybarra and Wiersema, 1999).

platform and apps can be readily separated, it is increasingly impossible for them to be developed by independent parties. Partitioning of subsystems through architecture can therefore allow innovation partitioning among organizations that develop them. A “good” architecture should respect this black-boxing because it can pay off handsomely: Ideally, each black box can be implemented by completely independent organizations, motivated by different things, driven by different expertise.

In theory, that’s the promise of platform thinking; instead of one company having to implement all the black boxes, hundreds of thousands of different companies (as in the case of iOS and Android) can create the black boxes. The power of the market then takes over—the valuable black boxes survive and the rest fade away in a competitive marketplace for black boxes. In platform ecosystems, almost all these black boxes are the apps developed by independent entrepreneurs, who take over the innovation role from the platform owner and collectively expand it to a scale that the platform owner cannot even imagine replicating inhouse. Architectures that effectively partition complexity allow these numerous outside organizations to provide the pieces of a larger ecosystem while also ensuring that the parts coherently fit together. Software architecture can therefore enable a divide-and-conquer approach in which a complex ecosystem is divvied into manageable components—the platform and its many apps—that can be developed independently and subsequently brought together. (This decomposition is what we subsequently call *modularization.*)

Partitioning affects the work of both the platform owner and app developers. For the platform owner, effective partitioning has consequences for viable organizational structures around a platform. Effective partitioning largely determines whether the development of complementary apps is best done inhouse by the platform owner (as has traditionally been done for complex software) or by a distributed, multiorganizational ecosystem. Platform architectures therefore mold viable business models (Meyer and Selinger, 1998), both opening and constraining possibilities for platform owners. It also allows outsiders to engage in parallel competing efforts to solve the same problems using a variety of different approaches. Therefore partitioning created by a platform’s architecture permits variety in the apps that can complement the platform. The greater the uncertainty a platform faces about end-users’ needs, the more valuable is such diversity; a greater variety of competing attempts to meet end-users’ needs increases the odds that some attempt will work. Partitioning also reduces complexity: If a complex ecosystem can be divided into separate parts such as each part can be developed by different people, the limitation of complexity disappears (Baldwin and Clark, 2000, p. 5).

Consider how partitioning affects the work of app developers. The architecture of a platform ecosystem specifies how the ecosystem is decomposed into the platform and apps that interoperate with it, and how the two types of subsystems interoperate to provide the overall functionality of the platform ecosystem. Their architecture therefore influences which parts of the ecosystem (e.g., the platform or other apps) must be tweaked to implement a new version of an app, and in turn influences the costs incurred by the app developer for changing the app. If the architecture of an app is less independent of other subsystems in the ecosystem (Figure 5.6B), it is not possible to make changes to an app without also having to make parallel changes in the platform and possibility in other apps. As a platform ecosystem’s complexity grows, interdependencies between the platform and apps can become so numerous that integrated development efforts become impossible (Ethiraj and Levinthal, 2004b). Such a platform can grow into an immensely complicated tangle of interconnections, with each part potentially dependent on every other part. The app developer would need to know beforehand what these dependencies are and which can be prohibitively difficult to understand and keep track of as a platform gets more complex. Therefore, even small changes can have an unpredictable cascade of ripple effects on other parts of the ecosystem. The greater the number of other subsystems that must be tweaked in order to successfully alter an app, the greater are the coordination costs faced by an app's developer (Adner and Kapoor, 2010). If apps are to be successfully produced by outsiders, these costs must be contained.[2] Architectural differences across platforms can therefore explain partly why the costs of innovating can be starkly different even for the same app across comparable rival platforms. Architectural differences can also explain not just the frequency of innovations feasible by app developers but also the types of innovations that do and do not occur in an ecosystem.

Partitioning through architecture influences how much app developers need to understand the insides of the platform and need to be aware of other apps that their own app might interact with. A well-partitioned architecture can provide app developers the benefit of *valuable ignorance*: In doing their own work, they need not know how a platform does what it does. Nor do they need to understand the intricacies of the platform native functionalities on which their app draws. This form of ignorance is valuable because it allows app developers to focus largely on their own work yet be able to subsequently integrate their completed app with the platform. It allows them to sharpen focus on their distinctive knowledge and capabilities for creating and implementing novel ideas that they can pursue relatively autonomously. A well-partitioned architecture can therefore reduce the costs faced by both app developers and platform owners to coordinate their work with each other. This is what economists call *transaction costs* (Baldwin, 2008; Williamson and De Meyer, 2012). Therefore, by simultaneously enabling and constraining individual participants in a platform's ecosystem, architectures influence innovation generation both by the platform owner and app developers. Architectural differences can therefore affect the intensity, quality, and type of app innovation that are critical determinants of the vibrancy of the platform's ecosystem.

5.3.2 Systems integration

The second function of architecture is to enable systems integration. Systems integration refers to coordination of development activities among app developers and the platform owner. Systems integration capability of a platform is a platform's capacity to combine the different competencies of app developers with those of the platform. Although partitioning of apps and platforms allows app developers to pursue their development work building on their own strengths, these apps must eventually interoperate with the platform to deliver value to end-users. Apps mixed-and-matched by an end-user must coherently work together in individual end-user instantiations of the ecosystem (Boudreau, 2010). Integration between a platform and apps is therefore critical to realizing the potential of apps by app developers participating in a platform's ecosystem. If apps seamlessly interoperate with the platform and augment its own capabilities in creative ways, the ecosystem can become a powerful mechanism for a platform to acquire a steady stream of new capabilities from its ecosystem partners.

App developers face two broad types of costs in their ongoing work: (1) *app innovation costs* and (2) *systems integration costs*. App innovation costs are an app developer's costs of actually designing and implementing the changes over the lifecycle of an app. For example, if Skype wants to add new functionality to its iOS app, these are the costs of conceptualizing, designing, and

[2]Generally, costs of coordinating across interorganizational boundaries of different organizations are much greater than they are among groups within the same organization (Rysman, 2009).

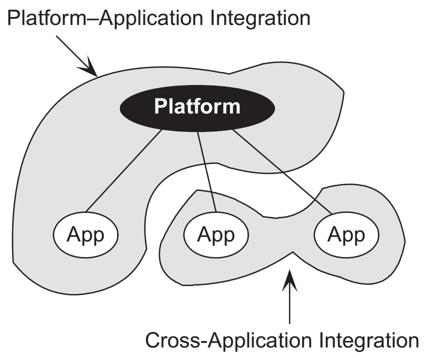

Platform–Application Integration
Platform
App
App
App
Cross-Application Integration

FIGURE 5.7

The two types of systems integration costs faced by app developers.

implementing a revised app with that functionality. Systems integration costs are the effort required by Skype developers to ensure that the revised Skype app will function as intended when installed on an iOS device.

Systems integration costs then refer to the effort required to manage the dependencies among a platform and apps in an ecosystem. The potential advantages of a large ecosystem can easily be wiped out if systems integration costs are high. App developers directly face two types of systems integration costs, illustrated in Figure 5.7: (1) those of integrating an app with the platform (application–platform integration costs) and (2) those of integrating an app with other potentially interacting apps in the ecosystem (cross-application integration costs). The second type of systems integration costs can be reduced by a platform's architecture, but the first type is more challenging for two reasons. First, different apps might evolve at different rates (Baldwin and Clark, 2000, p. 297). This means that app integration with the platform is lumpy rather than predictable. Second, changes in the platform itself can require an app to be changed to maintain its integrity and interoperability. Therefore, it is more useful to think of system integration costs as ongoing rather than a one-shot integration activity. Different platform architectures, however, impose different levels of system integration costs for apps, and in turn can change the rate of investment by app developers in app innovation. Architecture influences the initial and subsequent releases' systems integration costs faced by app developers by altering modifiability of an app. This in turn affects app developers' incentives to innovate rapidly and the extent to which they will be willing to bear the risk of app innovation. High systems integration costs faced by an app developer can therefore discourage innovation by app developers.

The common systems integration approach in the software industry is overt communication between the parties. The two parties communicate, interact, and coordinate their own work to ensure that their subsystems will work together. In platforms, such ongoing interaction between the platform owner and an app developer is one mechanism for ensuring successful systems integration. This approach, however, becomes increasingly infeasible in complex ecosystems involving thousands of app developers, where different apps might also be evolving at a different pace.

An alternative solution is not to maximize communication among them but rather minimize the *need* for it. Platform architecture can potentially reconfigure the structure of dependencies among a platform and apps. It can provide the blueprint that stitches together apps and the platform on a scale where communication-based coordination mechanisms of traditional organizations are simply infeasible. Architecture—by specifying dependencies and interactions between apps and the platform—can then become an invisible coordination mechanism that can substitute for such overt communication-based coordination in platforms ecosystems. Early architectural choices by a platform owner—by influencing the costs of realizing innovations at every level—can therefore either catalyze or discourage experimentation within a platform's ecosystem.

5.4 ECOSYSTEM ARCHITECTURE

Platforms are *purposefully designed* complex systems with an underlying structure that influences how they behave, function, and evolve over time. Like any other complex system, a platform ecosystem can be envisioned as composed of many interacting subsystems. How these subsystems interact is determined by the platform ecosystem's architecture. Two broad types of subsystems here are the platform itself and the portfolio of apps that augment it. Just as the architecture of a building is different from the building itself, the architecture of a platform ecosystem is at a higher level of abstraction than either the platform or the apps. Architecture is to a platform ecosystem what a blueprint is to a building. Rather than a working system, it is a description of the building blocks of an ecosystem and how they relate to each other, what they do, and how they interact (van Schewick, 2012, p. 21). This high-level description specifies the components of the ecosystem, the externally visible properties of these components, and the relationships among them (van Schewick, 2012, p. 21). Ecosystem architecture ideally partitions the ecosystem into two types of subsystems: (1) a highly reusable core platform that remains relatively stable and (2) a set of complementary apps that are encouraged to vary (Baldwin and Woodard, 2009). Architecture therefore describes the early design decisions about the decomposition of a platform ecosystem into a platform and apps. These choices have considerable evolutionary consequences because they are largely unchangeable, as we describe in the next section.

Architecture is meaningful only in relation to other parts that together constitute the whole ecosystem. Architecture is a hierarchal concept: Ecosystems can be decomposed into interrelated subsystems such as apps, which also have architectures (Baldwin and Clark, 2000, p. 413). Ecosystem architecture can be thought of as comprised of two levels: (1) the architecture of the platform itself (*platform architecture*) and (2) that of an app, which we refer to as that app's *microarchitecture*. Platform architecture is like viewing a platform ecosystem through a telescope and app microarchitecture is like viewing architecture through a microscope. This distinction is illustrated in Figure 5.8. Platform architecture includes the core platform *and its interfaces*. Recall that a platform is a set foundational functionality and shared assets made available to apps through a set of interfaces. Platform architecture should tell apps both what the platform does and how to use the platform (Parnas et al., 1985). The latter is a role directly played by the platform's interfaces, which therefore must be treated as an integral part of a platform's architecture.

Although the platform has a specific architecture that all apps see, the architecture of individual apps within the same platform can vary from one app to another. Platform architecture imposes constraints on all apps in a platform's ecosystem; therefore many properties of app architectures are correlated with the architecture of the platform. However, the two are rarely identical because there can be considerable variance among apps developed for the same platform. Therefore, an app's "microarchitecture" will define how each individual app interacts, communicates, and interoperates

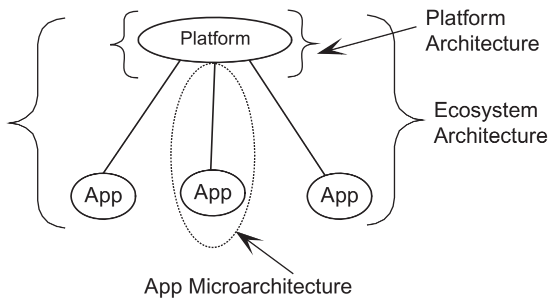


FIGURE 5.8

Ecosystem architecture is comprised of platform architecture and app microarchitecture.

with the platform. Even if the platform owner attempts to impose architectural guidelines on app developers, the extent of compliance by individual app developers with such guidelines can result in different app microarchitectures. A useful way to think of the distinction is to distinguish between envisioned versus realized architecture. Platform architecture is architecture for apps as envisioned by a platform owner; app microarchitecture is the same architecture as realized in the implementation of an individual app by its developer. This inheritance of properties of platform architecture by apps gives them their evolutionary properties.

Such decomposition can continue until the lowest atomistic level is reached. For example, an app can be further divided into subsystems, and subsystems within those subsystems, *ad infinitum*. However, for practical purposes, it is meaningful to stop at the level after which further decomposition no longer aids comprehension. Zooming out—the opposite of architectural decomposition—results in a more aggregate view of a platform ecosystem. For example, individual ecosystems themselves might be embedded within a larger architecture, which can be aggregated to the highest level of the Internet as a whole. They also coexist, compete, and cooperate with rival ecosystems.

5.4.1 App microarchitecture

Each app can have its own internal structure that represents its *internal* microarchitecture. The internal microarchitecture of the app influences its *external* architecture (i.e., how it connects to the platform).

5.4.1.1 The four functional elements inside an app

Any software app's internal functionality can be decomposed into four functional elements shown in Figure 5.9. These four pieces are:

1. *Presentation logic*. An app's presentation logic is where almost all of the interaction with the end-user occurs. It is the part of the application that handles receiving inputs from the end-user and presenting the application's output to the end-user.
2. *Application logic*. The second function is the core work performed by the application that is distinctive to it. This encompasses the functionality of the app that makes it uniquely valuable to its end-users. For example, a video conferencing app's core application logic is the video streaming between two client devices.

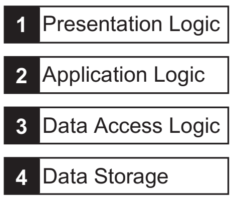


FIGURE 5.9

The four pieces of an app's internal functionality.

3. *Data access logic*. The third function is the processing required to access and retrieve data. This often equates with database queries through which user-specified data is retrieved from data storage. Examples of data access can include tag searches to retrieve emails, flight pricing data used by a travel reservation app, retrieving a media file such as a music file, or specific images from a larger database of images.[3]
4. *Data storage*. The last function is data storage. Most apps require data to be stored somewhere in order to be retrieved. This can be a small text file written by a word-processing app, map data in a navigation app, PDFs in an annotation app, images in a note-taking app, pictures in a camera app, or messages in an email app.

These four functional elements constitute the entirety of the *internal* microarchitecture of any app. These four elements can be placed on either the client side or the server side in several plausible arrangements.

An app's internal microarchitecture (also known as its network architecture) is simply a description of how these four functional elements are distributed between a client and a server connected by the Internet. Note that the client and the server need not be actual computers but can be any device (e.g., smartphone, tablet, object, or appliance) that is connected to the Internet. The five common arrangements that spread the four pieces across a network result in four different types of app microarchitectures: (1) standalone, (2) cloud, (3) client-based, (4) client–server, and (5) peer-to-peer. These five app microarchitectures apply to any networked application.

5.4.1.2 Unique properties of platform-based app functional partitioning

The unique aspect of these app architectures in platform settings, however, is that each individual functional element can be flexibly partitioned between the app and the platform, as Figure 5.10 illustrates. An app developer can rely entirely on the platform itself to build each part of the app, or choose to build part of the client-side and part of the server-side functionality itself and rely on the platform for the remainder. (An app can also invoke third-party Web services using Web service APIs (application programming interfaces) to implement some of its functionality.) The modularity of the connections—defined by decoupling and interface standards compliance—between the app and a platform in such divvying-up of client- and server-side functionality then represents its *external* microarchitecture. For example, 50% of the presentation logic of an app might be implemented within the app and the other 50% might be implemented in the platform. The app then invokes the capabilities of the platform to execute its own presentation logic. The choice of how much of each functional element

[3]The accessed data can be either user-owned or provided by a third-party supplier.

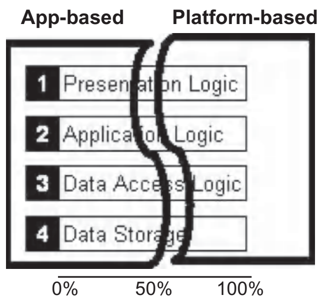


FIGURE 5.10

Each of the four functional elements of an app can be flexibly partitioned between an app and the platform.

is app-based and how much of it is platform-based is largely a decision made by app developers. The connections between the two big blocks in Figure 5.9 that are split between an app and a platform defines its external microarchitecture.

An app's designer has considerable freedom in choosing how much of an app's functionality to pull from the platform and how much to build herself (Figure 5.10). The division of functionality between the app and the platform in app microarchitecture is therefore not a given but rather a choice. This choice influences how much of the app development work is done by the app developer and to what extent the app leverages and invokes the capabilities of a platform in order to function. This results in different app microarchitectures for similar apps even within the same platform. This choice also has nonobvious consequences that can enable and constrain the future evolution of the app in nuanced ways. As the next section of this book explains, such choices have considerable consequences on the evolutionary trajectories open to and closed to individual apps. This type of partitioning of each functional element between the app and platform allows the app developer to avoid duplication of the core functionality of the platform and instead focus on building capabilities unique to the app. In platform-based apps, some part of the functionality for one or more of the functional elements will *always* reside on the platform than in the app. This is the premise of platform-centric models and also the reason for the importance of platform governance (the focus of the next chapter).

5.4.1.3 Standalone microarchitecture

The first app microarchitecture is standalone architecture (Figure 5.11). Here all four functional elements are on the client side and nothing resides on the server side. Internet connectivity is therefore unnecessary for the app to function. This is the model for applications that dominated in the era that preceded Internet-enabled computing beginning in the 1990s. However, these four functional elements themselves can be divvied up between the platform and the app, with varying proportions of each implemented in the app and invoked by the app from the platform. Examples of such apps include a PDF reader or a flashlight app.

5.4.1.4 Cloud microarchitecture

The second app microarchitecture is cloud architecture (Figure 5.12). All four functional elements reside on the server side. The client device then simply becomes a "dumb" terminal that serves only to accept user inputs and display outputs. Although some of the presentation or data storage

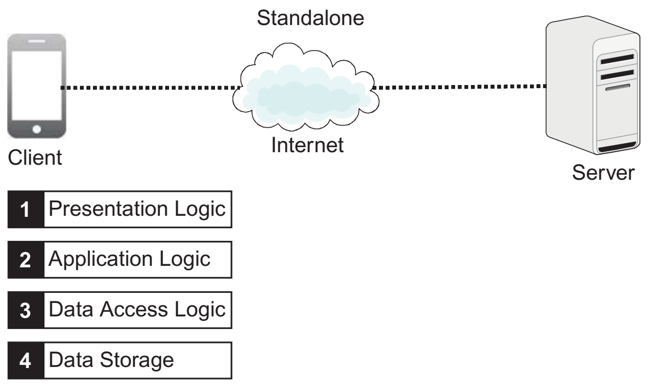


FIGURE 5.11

All four functional elements reside on the client device in the standalone app microarchitecture.

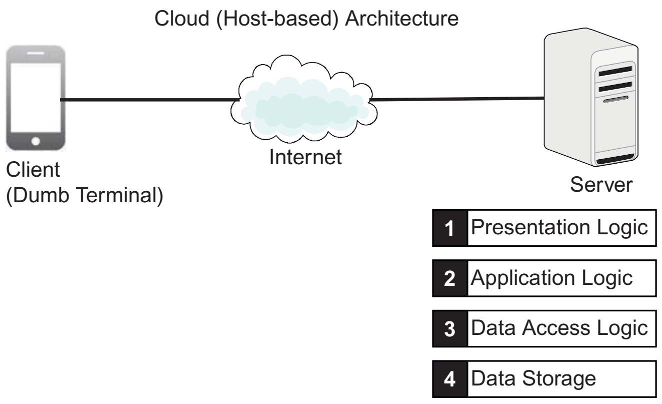


FIGURE 5.12

All four functional elements reside on the client device in the cloud app microarchitecture.

functionality might also reside on the client side, most of it is on the server side. This model mirrors host-based computing architectures from the mainframe era and contemporary "thin client" architectures. Examples of apps using such architectures are most search engines, where almost all of the work of the search application is done on the server side and the client is usually a browser that serves little purpose other than accepting user inputs and displaying outputs. Other examples include Web-based email apps such as Gmail.

Cloud microarchitectures have three advantages. The key advantage of this arrangement is economies of scale: All of the functionality of the app is managed in one centralized location. A second advantage is that rolling out new features and functionality is easier because little or no upgrades are required on the client side. Instead, rolling them out simply requires changes on the server side. This benefit holds only in theory. In practice, however, the server-side functionality that is platform-based is not under the direct control of the app developer. This makes it challenging to roll out new functionality to the app's users as readily as it might appear on paper. Third, cloud microarchitectures are potentially more secure because all of the app's functionality is centralized on the server side. There is only one major point of vulnerability relative to other microarchitectures: the server. Such architectures are particularly attractive in platform apps where the device on the client side does not have much processing power. This is often the case with networked objects that are used as part of the emerging Internet of Things.

This microarchitecture has two big downsides. First, all of the work of the app must be done on the server side. As user demand and usage intensity grows, the server side can become overloaded and slow to respond to requests from the client side. This can potentially result in sluggish response times. Put another way, the app has greater network usage intensity. Second, capacity upgrades on the server side cannot be incremental. Instead, they are large and potentially expensive. In other words, cloud-based architectures are not cost-effectively scalable. And the more the app leverages the platform, the more dependent the app developer becomes on the platform.

5.4.1.5 Client-based microarchitecture

The third app microarchitecture is client-based architecture (Figure 5.13). This arrangement is similar to standalone architectures with the exception that the data storage function is placed on the server side. The key driver of the advent of these architectures is an increase in the processing power of client devices (such as smartphones and tablets) and the advent of high-speed ubiquitous connectivity. This allows the majority of the functionality of the app to reside on the client device. This microarchitecture makes sense if storage capacity is a constraint on the client device. The upside of this arrangement is that the data resides in a centralized location, which makes it easier to secure. The key downside of this architecture is network congestion: All data must travel over the Internet from the server to the client each time the application is used. This includes data that might not be needed by the user, which cannot be separated from the data the user actually needs because the data access logic is on the client side. This can result in larger than necessary bandwidth consumption and can result in sluggish application performance. If the app is data intensive, this microarchitecture also suffers from lack of cost-effective scalability since part of the app's functionality is derived from servers (usually run by the app developer or run by the platform owner but paid for by the app developer). Scaling up can therefore be costly.

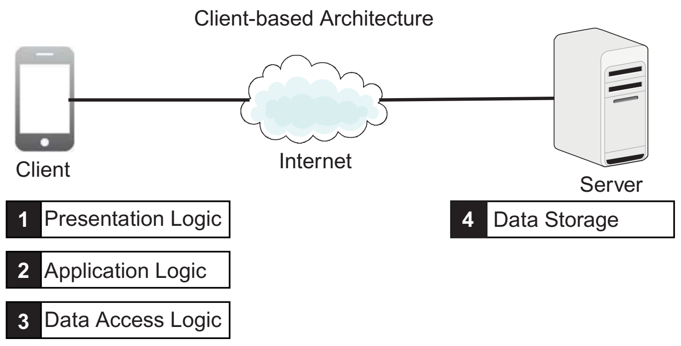


FIGURE 5.13

Only data storage resides on the server side in client-based app microarchitecture.

5.4.1.6 Client–server microarchitecture

The fourth widely used app microarchitecture is client–server architecture (Figure 5.14). This arrangement evenly splits the four functional elements of an app between clients and servers. While data access logic and data storage reside on the server side, presentation and application logic reside on the client side. In practice, the application logic is often split between the client and server, although it predominantly resides on the client. This design balances processing demands on the server by having the client do the bulk of application logic and presentation. It also reduces the network intensity of an app by limiting the data flowing over the Internet to only that which is needed by the user. Placing the data access logic on the server side accomplishes this; the queries from the client are initiated from the client but executed by the server, which only sends back the results of those queries rather than the entire raw data as client-based microarchitectures do. The downside of client–server app microarchitectures is that different types of client devices must be designed to invoke the data access logic on the server side in compatible ways. Depending on how the server-side functionality is split between an app and the platform, this arrangement can potentially free up app developers' attention to focus their attention on developing the core functionality of the app (where most end-user value is generated) and fret less about the data management aspects of the app.

5.4.1.7 Peer-to-peer microarchitecture

The final app microarchitecture is peer-to-peer architecture (Figure 5.15). The same device acts as a client and as a server in this arrangement, with significant elements of each of the four functions of the app present on it. Because each device serves simultaneously as a client and a server, the consolidated device is often referred to as a *servlet*. In its pure form, there is no separate server or centralized point of control. This means that every client also simultaneously acts as a server. Therefore all devices connected to peer-to-peer architecture can simultaneously initiate requests and fulfill requests from each other. The key advantage of this approach is immense scalability: The addition of every new client simultaneously adds server capacity to the network. Scaling the capacity of any other architecture usually requires additional capacity on the server side, the need for which is eliminated by the use of peer-to-peer microarchitectures. Skype is an example of such architecture; it allows tens of millions of users to simultaneously use the service and can readily and automatically scale to meet rising demand. The incremental cost of adding another use is therefore pennies, and adding more users improves app performance unlike all other app microarchitectures where adding more users degrades performance.

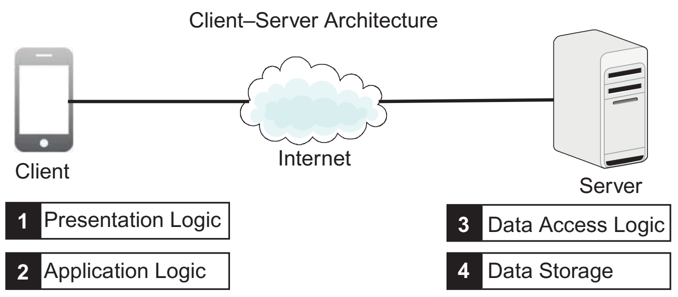


FIGURE 5.14

Client–server app microarchitectures evenly split application functionality among clients and servers.

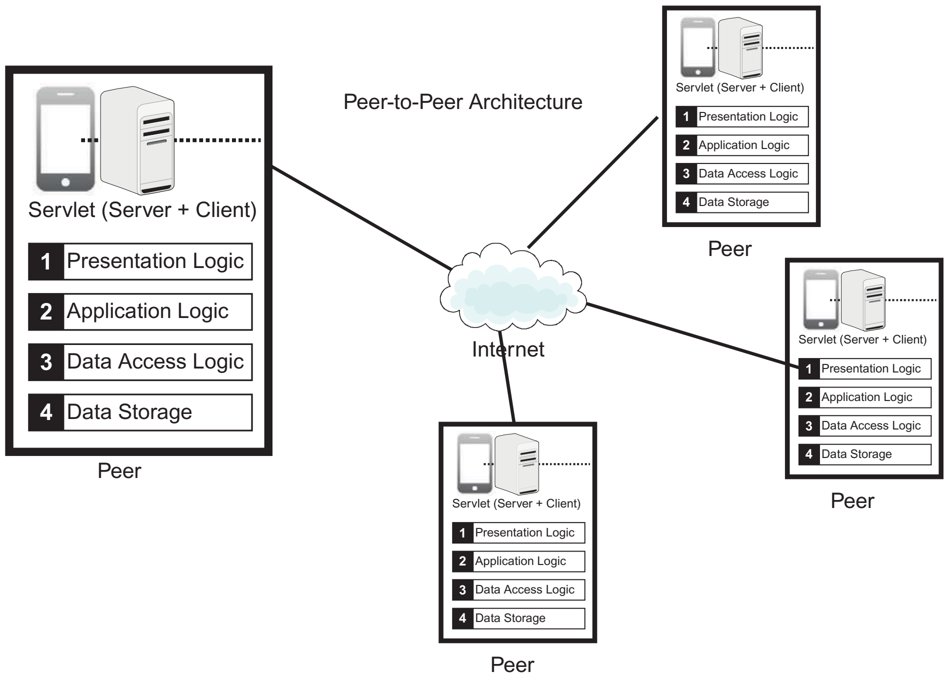


FIGURE 5.15

Peer-to-peer app microarchitecture.

However, this microarchitecture has two caveats. First, there is little or no control that the app developer has over the users of such apps. This limits the utility of this arrangement to only a few types of applications where central coordination and control are not needed and need for scalability is extremely high. Second, this architecture rarely exists in its purely decentralized form. Some centralized control is often needed and accomplished by retaining some of the server-side functionality on a separate server. This hybrid form of peer-to-peer architecture infuses some elements of the other microarchitectures (e.g., adding a central server) into the completely decentralized pure-form peer-to-peer arrangement.

Client–server and cloud-based architectures dominate contemporary platforms. Hybrid microarchitectures that mix-and-match properties of these five are increasingly in use; nevertheless, one architectural approach dominates even in such hybrids. For example, Skype has a predominantly peer-to-peer architecture, with some elements of the client–server model incorporated into it (e.g., monitoring user availability and initiating calls).

5.4.1.8 Tiering in app microarchitectures

The preceding app microarchitectures split the functional elements of an app between two devices: a client and a server. They are therefore called two-tier architectures. This logic can be extended to splitting this functionality among one client but more than one server. As this splitting is done using three devices, it becomes a three-tier architecture (see Figure 5.16); more than three devices makes it an

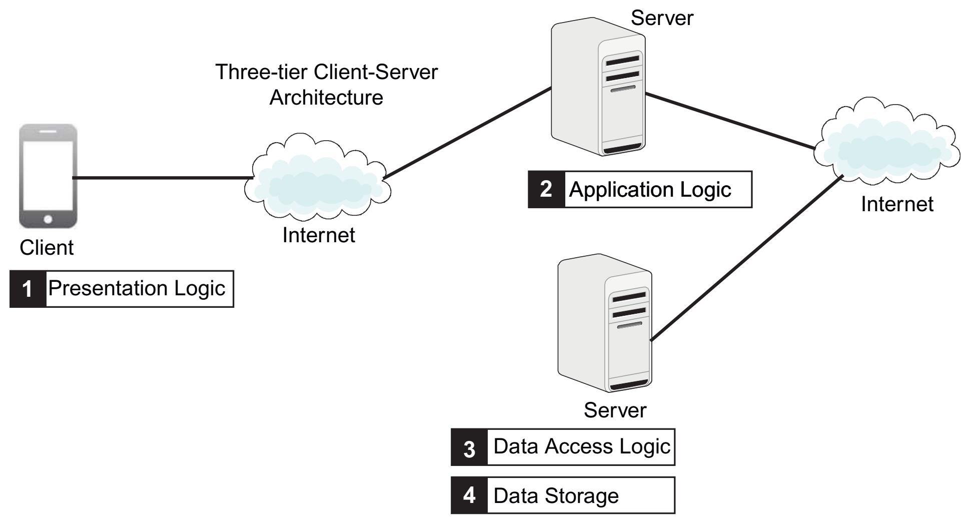


FIGURE 5.16

Tiering.

n-tier architecture. In any such multi-tiered architectures, there is always only one client. (Tiering is conceptually distinct from layering.[4])

The upside of tiering is that it increases scalability of the app. Separating the functional elements of an app into more than two tiers allows just one device associated with specific functionality to be replaced or upgraded. If the demand for, say, application logic increases, just the servers hosting that functional element need to be replaced or upgraded. In platform-based apps, three-tier architectures are often used when an app draws on different servers for part of its functionality. Tiering also provides app developers the future flexibility to more easily move one tier of the app from being app developer-owned to being platform-based or vice versa. This phenomenon was widely observed when Apple introduced its cloud-based data storage service in its iOS platform, which many app developers deployed for the data storage tier. The downside of tiering is that because the data now must travel along more paths, the network bandwidth intensity and the risk of sluggish performance both increase with tiering. The remainder of this book will use a two-tier architecture for simplicity; the logic can readily be extrapolated to multi-tier architectures.

An ongoing evolution in app microarchitectures is that the networks connecting various devices have increasingly migrated to Internet protocol (IP). Independent of the type of network (Wifi, 3G, LTE), the common language spoken by them is IP. This protocol delivers just about any type of information (e.g., voice, video, text) formatted as small IP packets—of fixed-size blocks of data—with a standardized structure between any two devices designed to speak the language, and it does so robustly, error-free, inexpensively, and over long distances using the Internet. This allows increasingly heterogeneous devices and networked "things"—including ones that do not yet exist—to interoperate with each other from the client and server side.

[4]Layering is a special extension of modular design that constrains permissible interactions among modules. A subsystem assigned to a specific layer can use only the subsystems in the lower layers or in the same layer, but not in the layers above it.

5.4.1.9 Strategic and evolutionary consequences of app microarchitectures

An important question for app developers is therefore how to partition an application across the Internet (internal microarchitecture) and across the app and the platform (external microarchitecture). These choices are made early in their lifecycle and are almost impossible to subsequently reverse. Several properties of an app are affected by the microarchitectural choices made by their developers during their initial implementation. Some of these qualities are immediately visible in the short term: speed, security, reliability, scalability, testability, and usability. Performance of an app depends on the complexity of interaction and the amount of communication between the app and the platform ecosystem (including other apps). The amount of communication in turn depends on how the four pieces of functionality are distributed between an app and the platform by its developer's architectural choices.

However, other evolvability-related qualities are visible only in the longer term: maintainability, extensibility, evolvability, and the capacity to mutate and envelop adjacent app market segments. (For example, changes to an app can be made faster and more cost-effectively if few systems outside of the app require tweaking when the internal functionality of an app is changed.) App microarchitecture also enables versioning of apps, as described in Chapter 11. App microarchitecture also has direct strategic consequences for same-side and cross-side network effects, and for how inextricably an app can get locked into one platform. Therefore, evolvability of an app is preordained by its microarchitecture. Unfortunately, architectural choices always involve tradeoffs; maximizing one attribute usually requires compromising on another. The consequences for the app's visible performance are immediate, but the more profound evolvability consequences surface much later. App designers must be cognizant of these tradeoffs so they can make them consciously rather than recognizing them after the fact. (Chapter 11 tackles these issues in detail.)

5.5 FOUR DESIRABLE PROPERTIES OF PLATFORM ARCHITECTURES

Platform architecture is an enduring—often irreversible—choice with profound evolutionary and strategic consequences. Good platform architecture has four desirable properties. These architectural properties always invoke tradeoffs such that dramatically increasing one property will reduce another. It is therefore impossible for any architecture to simultaneously have high levels of all of these properties. On the other hand, some of these properties are correlated; increasing one can help nudge another property upward. A platform architect should aspire for "satisficing" (a mix of satisfactory and sufficient) levels of a mix of these properties. We focus primarily on the architectural properties of the platform rather than of apps. Apps can potentially inherit a platform's architectural strengths, but this usually requires that the platform first have them! Performance is visibly missing on this list, largely because an acceptable level of performance is taken to be a precondition for a platform to be viable in the immediate future. By taking performance off the list, we focus on the core properties of architecture that influence the *evolution* of a platform. The four desirable properties are:

1. *Simple*. The architecture of a platform should be simple enough to be comprehensible at least at a high level of abstraction. This means that the platform should be conceptually decomposable into its major subsystems, the platform's functionality reused by many apps should be identifiable, and interactions between the platform and apps should be well defined and explicit. In short, simplicity pays off.

2. *Resilient*. One defective app should not cause the entire ecosystem to malfunction. The key to such resilience is to ensure that apps are weakly coupled with the platform through interfaces that do not change over time. This approach of keeping platform–app dependencies to a minimum also makes the entire ecosystem more stable in its performance.
3. *Maintainable*. It should be possible to cost-effectively make any changes within the platform without inadvertently "breaking" apps that depend on it. Conversely, changes in an app should not require parallel tweaking in the platform. This is accomplished through partitioning it into standalone subsystems (described elsewhere in this chapter) and then linking them using standardized interfaces. Designing for maintainability also increases a platform's composability (i.e., capacity to integrate with new apps).
4. *Evolvable*. Evolvability means the capacity to do things in the future that it was never originally designed to do. For this, the architecture—particularly the interfaces—of a platform must endure over time. This property allows a platform to be extensible in the near term and exhibit emergent behavior in the longer term. The key to evolvability is stable yet versatile platform interfaces that ensure autonomy between the platform and apps, make the architecture rich in "real options" (Chapter 8), and permit its mutation into derivative platforms (see Chapters 7 and 9).

5.5.1 Architectural lessons from cities

The architecture of platform ecosystems has several interesting parallels with the architecture of modern cities with long histories such as Atlanta or Paris (Table 5.1). Although this chapter focuses primarily on similarities in their structure, we revisit the parallels in their governance and evolution in subsequent chapters.

Table 5.1 Parallels Between the Architecture of Modern Cities and Platform Ecosystems

City	Platform Ecosystem
Structure	
Mix of preserved old buildings and new buildings	Mix of stable platform and new apps
Variety of buildings	Variety of apps
Stable roads and utilities (e.g., water, electricity, and sewage)	Stable interface infrastructure
Shared public facilities and infrastructure	Shared platform services and functionality reused by many apps
Discrete neighborhoods with unique character and purpose (e.g., residential vs. commercial)	Partitioning of functions with commonality and reusability into the platform, and unique functionality with low reusability into apps
Multiple stakeholders (businesses, residents)	Multisidedness (app developers, end-users)
Long lifespan of streets	Long lifespan of platform interfaces
Varied building designs	Varied app microarchitectures

Table 5.1 Parallels Between the Architecture of Modern Cities and Platform Ecosystems—cont'd

City	Platform Ecosystem
Governance	
City ordinances	Platform design rules
City taxes	Pricing policies and revenue-splitting arrangements between platform owner and app developers
Citizens' right to vote	Shared governance (decision-rights partitioning)
Law enforcement by the city	Interface standards enforcement by the platform owner
Autonomy of citizens within the constraints of city laws	Autonomy of app developers, with the constraints of the platform's rules
Evolution	
Retirement of old assets	Retirement of legacy functionality
Expansion into outskirts	Expansion with new interfaces and APIs
Gentrification of neighborhoods	Widespread adoption of once-unique services and functionality by many apps
Renovation of historic buildings	Expansion of platform core functionality over time
Capacity to absorb new migrants	Capacity to scale
Modernization while preserving its character	Emergent properties of the platform

5.6 MODULARITY OF ARCHITECTURES

Modularity is the general property of any complex system. Any complex system can be decomposed into smaller subsystems that are always going to be interdependent to some extent and independent to some extent (Simon, 1962). Modularization then refers to the extent to which the interdependence among these subsystems is intentionally reduced by design (Parnas, 1972; Parnas et al., 1985). Reduced interdependence simply means that changes in one subsystem do not create a ripple effect on the rest of the complex system. When viewed as a complex system, the two distinct types of subsystems in a platform ecosystem are the platform and apps. An ecosystem exhibits modularity if the platform and apps can be designed independently but will work together to constitute the ecosystem (Sanchez and Mahoney, 1996). Modular designs are therefore more Lego-like. An ecosystem can be intentionally *designed* to be more modular, although a less modular design can accomplish the same function. We therefore distinguish the process of *modularization* from the property of modularity. Modularization is the *deliberate activity* of increasing modularity of a system whereas modularity is a system's descriptive property of being more modular.

Consider a simple illustration of two different designs of a screwdriver in Figure 5.17. The first design (A) is a non-modular ("monolithic") design and the second design (B) is a modular design. Both designs provide the same functionality. However, the more modular design can be produced differently vis-à-vis the monolithic design. The monolithic design has to be produced as a single unit by one organization. (The smallest subsystem is the whole screwdriver.) The modular design has four parts (the four subsystems that constitute the whole screwdriver), each of which can be produced by four independent

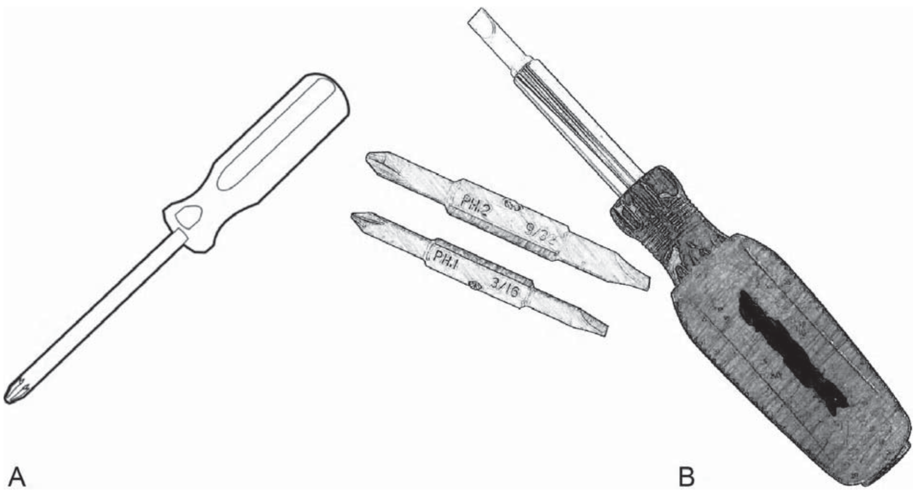


FIGURE 5.17

Monolithic (A) versus modular (B) design of a screwdriver.

organizations. The only information that each of these organizations needs to know is the size and shape of interface for the part for which they are responsible. As long as each of them complies with the specification for their part, fit between the four parts is guaranteed without the need for any additional coordination among them. A modular architecture of the screwdriver (B) therefore allows its production to be distributed among multiple organizations. Put another way, modularization of the screwdriver's architecture makes viable a different organizational design for producing it. The design philosophy is minimizing dependence across the four parts but maximizing it within them. These organizations need not know any internal details of other parts or how they are constructed, just how to connect to them. They can work in blissful ignorance of the other three parts of the screwdriver and focus exclusively on the one part for which they are responsible. The only requirement for compatibility is compliance with the appropriate physical dimensions of the interfaces. This allows each organization to specialize more deeply in improving its own part, engendering a laser-like focus on honing its core competence. If the same set of organizations attempted to jointly produce the monolithic design, they would require intense supervision, iteration, and coordination to produce it. A potential tradeoff is that the monolithic design might initially outperform the modular design. However, the modular design offers one advantage: Any new shape or size for a screwdriver tip can readily be added to the modular design in the future, even if it was not foreseeable by its original designer. This property is what we described in Chapter 2 as emergence. Changes in the architecture of the product can therefore open up new possibilities for who can participate in producing it and whether partitioned production can be accomplished cost-effectively. The process of converting a monolithic design into a more modular design is called *modularization*.

We can draw five useful lessons about design modularization from this example. Modularization of the screwdriver's architecture:

1. Makes its production divisible among many organizations (partitioning)
2. Restricts their interdependence to its connection points (interfaces)

3. Relies solely on compliance with interface specifications to integrate the four organizations' outputs (systems integration)
4. Sacrifices some performance for future flexibility (the so-called modularity tax)
5. Allows unforeseeable capabilities to be added in the future (emergent properties)

5.6.1 Software modularity

The same logic applies to the design of software-based platform ecosystems. Modularity of a platform ecosystem refers to the degree to which the platform and apps can be designed, implemented, operated, and altered independent of each other (de Weck et al., 2011, p. 188). Modularity in software systems is a property that can reside anywhere along a continuum ranging from perfectly monolithic to perfectly modular. A perfectly monolithic system is one where every app is highly interdependent with the platform. An example of a monolithic architecture is one where a platform and an app are integrated into a single system (e.g., the Mail app in iOS). A complex system like this can achieve high levels of integration across the subsystems that constitute it, and can exhibit high levels of performance and an assemblage of components that fit each other like a glove. The iPad would be an example of a monolithic hardware system. The other extreme would be a highly modular complex system such as a desktop PC. Each component connects to others using highly standardized interfaces, and can be replaced with a different component that adheres to the same interface specifications. This sort of plug-and-play architecture is an example of a modular system. Most complex systems fall in between these two extremes of monolithic and modular architectures.

Although the merits of either approach have been debated for years, industry battles have been fought over the philosophies (e.g., the PC vs. Mac dynamics in the 1990s); we will refrain from even attempting to settle that debate in this book. Instead, the important point to remember is that the *degree* of modularization of platform architectures has strategic and evolutionary consequences. Depending on a platform's strategy, different types of architectural choices will lead a platform ecosystem down different forks in the evolutionary road. Some architectures are more conducive to fostering the inventions of these "black boxes" than others. Therefore architectural choices by platform owners, and also by app developers, have decisive consequences for whether one ecosystem out-innovates another. But understanding these dynamics requires first understanding various elements of architecture with a degree of nuance that is not yet common practice among software developers and managers.

5.6.2 Platform architecture: an ecosystem's DNA

Architecture is to a platform ecosystem what your DNA is to you: You cannot change it after the fact. Even though it is possible to change it in theory, it is almost impossible to change it in practice. It puts you on a future trajectory that you cannot completely escape. Likewise, platform architecture preordains the realm of what a platform can and cannot evolve into. Platform architecture—like DNA—imprints future evolvability of its ecosystem. But much like your and my DNA did not guarantee good outcomes in life without our own effort, a platform ecosystem's evolution must be guided, prodded, and nudged. Architecture therefore creates the potential trajectories for a platform's evolution, but much more is needed to realize its evolutionary potential.

Modularity in the platform's design (1) allows disaggregation of its production rather than having one organization produce it and (2) provides future flexibility to extend the platform's native capabilities in ways that cannot be anticipated at the outset by its original designers. These two advantages are much more powerful together than the sum of their parts: Many outsiders can extend the platform's capabilities far beyond what it started out with. This gives a platform emergent properties. Emergence is a property where new capabilities unforeseen by the platform's original designers are added to a platform by an app (Dougherty, 1992). By making apps more self-contained and autonomous, modularization gives them the freedom to evolve and grow. It therefore infuses self-organizing properties into a platform's ecosystem.

Modular architectures rely on stable, well-defined interfaces to ensure interoperability between a platform and apps, minimize unnecessary interdependencies, and ensure that the remaining interdependencies are well understood by the app developer. Platform interfaces are an important leverage point—places in a platform's ecosystem where a small change produces a disproportionately large evolutionary influence—and represent loci of power to nudge its evolution. Modularization entails minimizing dependence between the platform and apps, but maximizing it within each of them (Ethiraj and Levinthal, 2004a). Modularity, however, is not an absolute property of a platform ecosystem. A platform can intentionally be designed to be more or less modular. Platform modularization is a necessary but insufficient precondition for app modularization. If a platform is modular in its design, apps produced for it have the *potential* to be modular as well. Even if the platform is highly modular, it does not automatically follow that apps in its ecosystem will also be highly modular. Different apps for the same platform can have different levels of modularity, depending largely on the microarchitectural design choices made by the app developer. If an app is modular in its design, it can be designed and produced independently of the platform and other apps in the ecosystem. It is therefore important to distinguish between architectures at the platform level and at the app level; these two aggregate to describe the modularity of an ecosystem's overall architecture.

Modularity in platform ecosystems is achieved by (1) decoupling the platform from apps (i.e., minimizing their interdependencies) and (2) codifying the interface specifications for how an app interacts with the platform (Tiwana, 2008a,b; Tiwana et al., 2010). Modularization enables the two core functions of architecture in platform ecosystems: partitioning and systems integration. It facilitates partitioning by reducing design dependencies between the platform and apps, and in turn reduces dependencies in their development tasks. Changes in one app become independent of those in others. Modularization is therefore an antidote—and a vaccine—against growing complexity.

5.6.3 Design precedes production

A platform ecosystem's architecture determines how its production can be organized (Baldwin and Clark, 2000, p. 30). Recall the mirroring principle from Chapter 2. A modular architecture allows a multitude of app developers to participate whereas a monolithic architecture would have required a single integrated organization to do all the development work. The development of the platform and apps can then proceed independently and concurrently in different organizations without much need for day-to-day interactions among them. In economists' lingo, modularity thus reduces "transaction costs" between platform owners and app developers (Baldwin and Clark, 2000, p. 354; Williamson, 2010). Concurrent development of various apps not only means faster development but also potentially translates into market competition among multiple possible solutions to the same user needs by rival app developers. It also lowers the need for the platform owner to constantly monitor and watch app developers, whose alignment with the platform's goals can be guaranteed by self-interest through proper platform governance. The more profound benefits of such partitioning are in the evolvability of the platform, where modularization makes growing complexity tenable and embeds powerful "real options" in the design (Baldwin and Clark, 2000, p. 90; Gamba and Fusari, 2009). (These benefits are described in Part III of this book.)

Modularization also facilitates systems integration by reducing the need for explicit coordination and communication among the platform owner and the app developers to accomplish integration of apps with a platform (Baldwin and Clark, 2000, pp. 77, 216; Tiwana, 2008a,b). By reducing the dependencies between the platform and app to just their interfaces, compliance to interface specification is all that is needed to ensure that they will seamlessly interoperate. In economist's lingo, modularity thus reduces "coordination costs" among platform owners and app developers (Gulati and Singh, 1998). Systems integration costs are ongoing costs potentially faced by app developers every time they make changes to an app. By dramatically reducing them, modularization reduces their disincentives to rapidly innovate.

Keeping transaction costs and coordination costs in control is usually why organizations organize innovation work inhouse rather than outsourcing it. But modularity can wipe out these advantages of traditional organizations and make ecosystem-based models just as viable. It therefore enables partitioning and dispersion of innovation work across multiple organizations where it was not possible before (Baldwin and Clark, 2000, p. 354). This can accelerate the evolution of the entire ecosystem, generating fierce Red Queen competition for rival platforms. Faster evolution, however, leads to survival only under some conditions, which Part IV of this book explains. Modularization can therefore be a powerful organizing strategy that enables decentralized innovation to become viable while preserving ecosystem-wide coordination.

5.6.4 Design modularity enables production modularity

Modularization of platform architectures matters also because it determines whether development around a platform can feasibly be organized as an ecosystem or whether it must be done within the boundaries of a single organization. Different architectures impose different constraints on those who interact with it or are exposed to it (van Schewick, 2012, p. 4). The preceding discussion focuses primarily on modularity in design, overlooking another kind of modularity—modularity in production (Gamba and Fusari, 2009), which refers to modularity in the logic used to organize *production* of a design. A traditionally integrated organization represents a monolithic organizing logic whereas a distributed ecosystem composed on many independent app developers represents a modular organizing logic used for producing the design. Modularity of production is a platform governance decision (the focus of the next chapter), but one whose feasibility is determined by modularity of the platform's design. Lack of design modularity, in contrast, makes production modularity infeasible (Langlois, 2002; Sanchez and Mahoney, 1996). Architectural choices therefore preordain the realm of execution possibilities. This is the essence of the mirroring principle. Design modularity and production modularity are therefore fundamentally isomorphic (i.e., design modularity enables production modularity) (Baldwin and Clark, 2000, p. 12; Hoetker, 2006). A platform's architecture is therefore particularly decisive in the evolution of its ecosystem because it is closely intertwined with the organization of the ecosystem: who chooses to participate, how much they are willing and able to invest in creating complementary innovations around a platform, and their incentives to innovate.

5.7 GOLDILOCKS STRIKES AGAIN

On the surface, it might appear that modularity is then a desirable architectural property at every level of a platform's ecosystem. However, modularity introduces tradeoffs that might not be worth its cost in some circumstances. The key tradeoff between modular and monolithic architectures is that between immediate performance and future evolvability (Figure 5.18).

Monolithic architectures usually can outperform modular ones in the short run but lead to rigidity because they are hard to change in small increments. Modular architectures, in contrast, makes a platform ecosystem more evolvable even though short-term performance will usually be poorer than monolithic architectures. Therefore monolithic architectures optimize immediate performance but offer lower evolvability, and modular architectures compromise immediate performance but offer higher evolvability. An optimal, just-right level of modularity is somewhere between the two extremes that strikes a balance between this performance–evolvability tradeoff. Modular architectures should therefore be favored over monolithic architectures when rapid innovation is more important than overall performance (Ethiraj and Levinthal, 2004b). Let us put ourselves in the shoes of a platform owner and app developer to examine the upsides and downsides of modularization from their perspectives.

5.7.1 Upsides of modularization

Modular architectures have four advantages for platform owners and app developers, as summarized in Table 5.2.

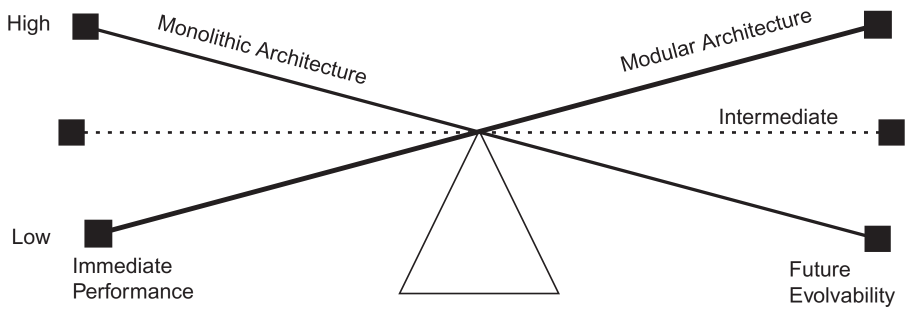


FIGURE 5.18

Tradeoffs between modular and monolithic platform architectures.

Table 5.2 Upsides of Modularizing a Platform for Platform Owners and App Developers

Platform Owner	App Developer
Massively distributed innovation	Less reinvention, more specialization
Increased variety of apps	Valuable ignorance
Greater volume of incremental innovation	Greater app evolvability
Control via architecture rather than ownership	Multihoming in rival platforms more feasible

The upsides of platform modularization for platforms owners are:

1. *Massively distributed innovation.* Modularization, like the de Prony trigonometric tables project, allows a platform owner to massively distribute innovation around a platform among thousands of independent app developers. A modular architecture wipes out the informational and coordination cost advantages of integrated organizations and makes mass collaboration across many organizations viable. Modularization enables a platform owner to leverage the distinctive capabilities of loosely coupled, independent, hungry app developers. Thus a modular architecture shifts the locus of innovation and risk-taking from the platform owner to more numerous app developers. This is particularly valuable when outsider app developers understand end-user needs and have diverse skills and capabilities that the platform owner does not possess inhouse (and cannot readily ramp up). Modularization also increases the future flexibility of the platform's trajectory ("option value" in Part III), which is most valuable when end-user preferences are diverse and unpredictable (Adner and Levinthal, 2001).
2. *Increased variety of apps.* By minimizing platform–app dependencies, app developers do not have to coordinate their own work with the platform owner yet are assured of subsequent systems integration. In other words, decreasing transaction costs between them reduces the costs of using the market to produce apps relative to developing them inhouse. With this come high-powered incentives that attract app developers, potentially increasing the variety of platform-specific apps. (Recall that outsiders usually have higher-powered incentives to perform well than inside employees.) This increases a platform's appeal to end-users and gives it a leg up in Red Queen competition with rival platforms.
3. *Greater volume of incremental and app-level innovation.* Innovation at the ecosystem can broadly be classified into four distinct modes—incremental, modular, architectural, or radical—as shown in Figure 5.19 (Henderson and Clark, 1990). It can involve refinement (incremental

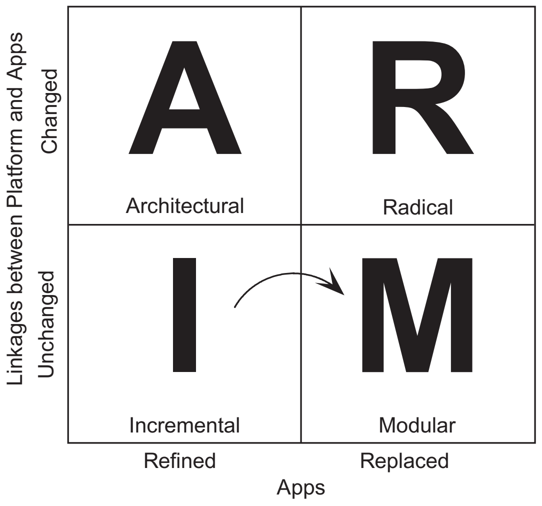


FIGURE 5.19

The four modes of innovation.

innovation) or replacement (modular innovation) of apps but leaving the linkages between the platform and apps unchanged. It can also involve changing the linkages between the platform and all its apps but leaving the apps largely stable (architectural innovation), or completely changing both the platform–app linkages and replacing the apps simultaneously (radical innovation). Modularization accelerates the quantity of innovation in the lower two cells. The reduced dependence of apps on the platform's innards allows the platform owner to make changes freely inside the core platform. Therefore, the threshold for modular innovation is much lower in modularized platforms. This potentially accelerates incremental innovation within the platform. (Modular innovation can be thought of as apps-driven innovation.) As a platform's architecture increases in its modularity, the innovation mode emerging around it shifts its evolutionary trajectory from being incremental tweaking (as in single-organization platforms) toward being modular innovation-oriented. Modularization of the platform architecture simplifies coordination with app developers, which is now achieved through design parameters and interface specifications rather than direct communication. Partitioning of apps also reduces the scope of troubleshooting and testing, reducing app development/revision timeframes (Parnas, 1972). Together, these changes speed up how fast app developers can introduce new apps and refine existing apps. New apps can rapidly extend the platform's own capabilities, and a rapid pace of introduction of new and revised apps continues to augment the usefulness of the platform. In contrast, the threshold for nonincremental innovation is much higher in monolithic platforms. Overall, this drives up the speed of app-level innovation around the platform. Therefore, modular platform architectures provide the advantage of variety today and evolvability tomorrow (Baldwin and Woodard, 2009).

4. *Control via architecture rather than ownership*. Finally, modularization of platform architecture potentially allows the platform owner to maintain some control and substantial influence over the ecosystem's evolutionary trajectory. Ordinarily, this level of control would require ownership of the entire ecosystem and the commensurate risk associated with developing apps to complement the platform. Maintaining sufficient control without compromising app developers' autonomy is a central aspect of governance of modularized platforms that we explore further in the next chapter.

The upsides of platform modularization for app developers are:

1. *Less reinvention, more specialization*. Platform modularization allows app developers to use the platform's services and reusable functionality as the *starting* point for their own work. The cognitive bandwidth freed from not having to reinvent the wheel allows them to invest more effort on functionality that differentiates their app in the marketplace. Put another way, it fosters specialization among app developers and reduces innovation costs.
2. *Valuable ignorance*. Apps in modularized platforms can be written with little knowledge of the code in the platform. This effectively reduces the conceptual design space that individual app developers must grasp.
3. *Greater app evolvability*. App developers can independently upgrade their apps without having to worry about interdependence or ripple effects elsewhere in the ecosystem. Each app is therefore free to evolve independent of the rest of the ecosystem, within the constraints imposed by the platform's interfaces (Gamba and Fusari, 2009). This increases evolvability of apps relative to monolithic platforms.

4. *Multihoming in rival platforms becomes more feasible*. Rival platforms are likely to share some common functionality, which app developers are less likely to reinvent if a platform is modularized. Therefore, more modular platform architectures increase app developers' incentives to join a platform and to remain engaged in developing around it (Baldwin and Clark, 2006). Furthermore, the cognitive bandwidth freed up by modularization and greater specialization in their app domain's functionality makes it more feasible for app developers to develop their app on multiple rival platforms, lowering their risk of being straddled with a losing platform.

5.7.2 Downsides of modularization

Modular architectures have four disadvantages, respectively, for platform owners and app developers, as summarized in Table 5.3.

The downsides of platform modularization for platform owners are:

1. *Modularity is not free*. Modularization is not free for platform owners. A platform owner must incur substantial upfront costs of modularizing a platform and establishing its interfaces. These upfront costs are usually much higher than those for more monolithic platform architectures.
2. *Technical performance takes a hit*. Modular architectures pay a modularization performance tax. The modular decomposition of a platform's ecosystem might split functionality that should have belonged in a single system instead of being split across many subsystems. The increased need for communication among these subsystems can decrease the overall technical performance of the ecosystem. It is usually not possible to fine-tune a modular system as precisely as a monolithic one (Langlois and Garzarelli, 2008). Therefore the technical performance of a modularized architecture is often worse than a comparable monolithic architecture for the same platform. Thus the assumption in modular systems thinking is that the benefits of decreased complexity and greater modification flexibility in apps and platforms offset the loss of overall system-wide performance.
3. *Modularization forecloses architectural innovation*. The choice of a platform's architecture puts it on a different trajectory in terms of the dominant mode of innovation (see Figure 5.20). More modular architectures are more conducive to incremental and modular innovation but often become hindrances to architectural and radical innovation (Figure 5.19). Recall that freezing a platform's interfaces is a precondition for modularization of its architecture. Platform interfaces, once widely adopted by app developers, lock in the platform for a substantial period of time even when a viable superior alternative subsequently becomes available (Langlois and Garzarelli, 2008).

Table 5.3 Downsides of Modularizing a Platform for Platform Owners and App Developers

Platform Owner	App Developer
Modularity is not free	Modularity imposes additional costs
Technical performance takes a hit	App performance takes a hit
Modularization forecloses architectural innovation	Modularity constrains experimentation
Increased risk of imitation by rivals	Leveraging the platform risks getting locked into it

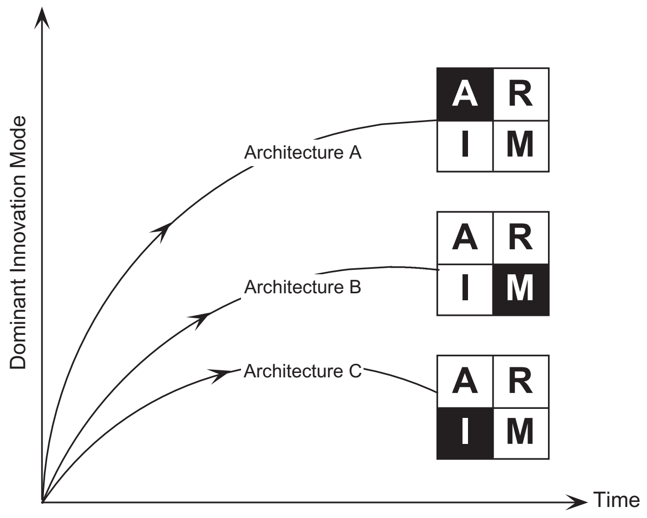


FIGURE 5.20

Platform architecture determines the dominant modes of ecosystem-wide innovation.

(Think of Apple's 30-pin i-device connector replaced by a 9-pin Lightning connector 10 years later in 2012, the QWERTY keyboard, USB, and the 30-year-old VGA connector. Although technologically superior alternatives became available—e.g., the Dvorak layout for keyboards and a plethora of potential successors for VGA interface standards—the legacy interfaces continue to widely persist.) The costs of poor architectural choices can therefore be high in the long run. Over time, modularization also reduces the platform owner's capacity for architectural innovation because balkanized expertise of the platform owner (from deepening its focus on the platform and away from apps) leads to a myopic understanding of the entire ecosystem. This focus serves it well, until the arrival of a different dominant design that offers sufficient improvements over the existing one surfaces. The platform owner's core competencies can then become its weaknesses. There are numerous examples of organizations that were unable to adapt to changes in their industry's dominant design. However, as Part IV describes, major innovations at the platform level are not impossible even with modular designs.

4. *Increased risk of imitation by rivals.* Monolithic designs usually take more time for rivals to understand and copy. Platform modularization therefore can increase the risk of imitation (Ethiraj et al., 2008; Pil and Cohen, 2006). A *nearly* modular design that retains some monolithic properties strikes an optimal balance between reducing ecosystem-wide coordination costs and risk of imitation by rivals (Figure 5.21). Therefore, intermediate levels of modularity are preferable.

Platform modularization has four downsides for app developers:

1. *Modularity imposes additional costs on app developers.* Developing for a modularized platform imposes costs of compliance with its design rules and interface specifications on app developers. These costs are often offset by the reduction in systems integration costs and gains in app evolvability.

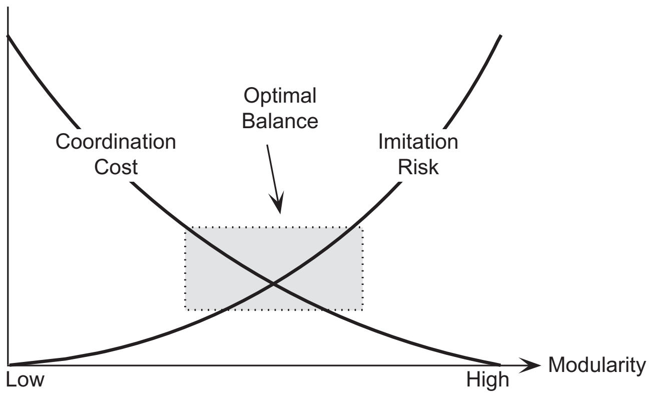


FIGURE 5.21

Modularization must balance the reduction in coordination costs against increased risk of imitation.

2. *App performance takes a hit.* Apps built around modular architectures often trade off technical performance (e.g., speed) by relying on a platform for some functionality. Such performance is likely to be higher in standalone, monolithic apps that are entirely developed by an app developer. Thoughtful app microarchitectures can compensate for this performance hit to some degree.
3. *Modularity constrains experimentation.* Complex modularized platform interfaces and their restrictions can constrain experimentation and flexibility of app developers as they attempt to develop new versions of their apps. Ingeniousness by app developers in app microarchitectures and their degree of tiering can sometimes help overcome such constraints, as we subsequently explain.
4. *Leveraging the platform increases app developers' vulnerability.* Modular platforms give platform owners an increased capacity to innovate *within* a platform. As a platform itself gains new or innovative platform-specific functionality and software services, app developers can leverage it in their own apps. (This is called increasing the *synergistic specificity* between a platform and an app (Schilling, 2000).) But this can be a double-edged sword if the leveraged functionality is unique to a platform because it makes an app more deeply dependent on and integrated with that platform. This increases the risk that an app developer will be locked-in with the platform and less able to multihome that app among rival platforms.

Overall, modularization is worth its downsides when a platform's markets are heterogeneous, riddled with uncertainty about technology trajectories and end-user preferences, and require deep pockets of specialized complementary knowledge that the platform owner does not have inhouse.

5.7.2.1 Why modular enough is good enough

Designing a perfectly modular platform architecture is almost impossible because it requires recognizing, anticipating, and resolving all app–platform dependencies in advance (Baldwin and Clark, 2000, p. 6). This is commonly problematic in practice because a platform's designers quickly run into the very human limits of bounded rationality. But the good news is that modular enough is good enough. Perfect modularity is still a useful ideal to aspire toward because even imperfect modularity reduces costs of software evolution and experimentation. Recent research has indeed shown that intermediate levels of modularity produce the most useful innovations (Ethiraj and Levinthal, 2004b). Architects aspiring to modularize platform architectures will likely end up somewhere short of perfect modularity and away from a monolithic design, which is a perfect place to be, as shown in Figure 5.18.

5.8 TWO MECHANISMS FOR MODULARIZATION

Modularization requires a mix of openness and secrecy. The logic behind modular architectures is to share information about the interfaces but keep the proprietary innards of individual apps and the platform secret. The secrecy in modular architectures comes from its first mechanism: decoupling. The openness in modular architectures comes from its second mechanism: standardization of interfaces. Decoupling facilitates decomposition of the ecosystem into relatively independent apps and the platform; interface standardization facilitates their reintegration after they are independently developed. This ability to separate and integrate subsystems is what software engineers call *composability* (Messerschmitt and Szyperski, 2003, p. 89).

5.8.1 Decoupling

Decoupling (or loose coupling) refers to the degree to which the components of an ecosystem are designed to be independent of each other such that changes *within* one component do not affect others in the ecosystem. At the heart of modular architectures is the idea of creating a “block-independent” structure, where the blocks represent the subsystems in an ecosystem (Baldwin and Clark, 2000, p. 61). The premise is that if details of a particular block of code are consciously hidden from other blocks, changes to one block can be made without having to change the other blocks (Parnas, 1972). Readers with programming expertise will immediately recognize this as the foundation of object-oriented programming languages, which are based on the idea proposed originally by David Parnas. In practice, this means that the platform and app should be designed to minimize interdependencies between them. Changes inside an app should not have an unpredictable ripple effect on the platform or on other apps. Conversely, changes inside the platform or in other apps should not adversely affect the functioning of that app.

Perfectly decoupled designs exist in theory but rarely in practice. Therefore, it is useful to think of decoupling as a slider scale with perfectly decoupled and tightly coupled on either ends of the scale. Decoupling lowers dependencies between the platform and an app, minimizing the need for coordination between the app developer and the platform owner when either the app or the platform is tweaked. Changes internal to one do not have a ripple effect on the other. A decoupled architecture can therefore potentially simplify interactions among app developers and the platform owner. This also means that coupling *within* an app and within the platform can be high such that subsystems within each can have strong dependencies. Therefore, decoupling means weak coupling between a platform and an app but strong coupling within a platform and within an app (Figure 5.22). Put another way, apps and platforms can be as monolithic internally as needed yet be a part of a highly modular ecosystem.

Decoupling is accomplished through a design process known as *encapsulation* (Baldwin and Clark, 2000, p. 63; Zweben et al., 1995). Any subsystem such as the platform involves two types of

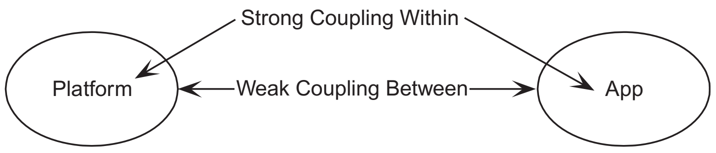


FIGURE 5.22

Weak coupling between a platform and an app and strong coupling within either.

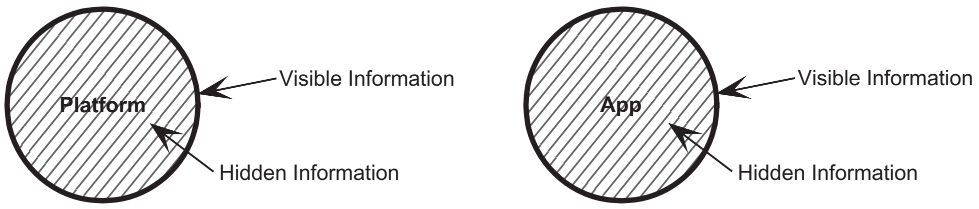


FIGURE 5.23

The distinction between visible versus hidden information in a subsystem.

information: visible information and hidden information (Figure 5.23). Information about what's inside a platform is its architecture's *hidden information*. It is information that is visible only to its developer. A well-decoupled platform should minimize the need for app developers in its ecosystem to have to know any hidden information. Internal details of its implementation should be invisible and untouchable by an outsider. The external view of the platform should display only its essential properties and hide unnecessary details of its internals. It must reveal as little as possible about its inner workings (Parnas, 1972). This property of information hiding is known as encapsulation of that subsystem. The external view is the platform's *visible information*, accessible to outsiders through its *interfaces*. The platform's interfaces are therefore the entirety of its visible information (Baldwin and Clark, 2000, p. 89). This means that only the interfaces of a platform are visible to app developers. The same logic can be applied to individual apps to encapsulate them by separating hidden information from visible information.

Interacting with the platform should require nothing more than access to its interfaces (i.e., the visible information) specified in its architecture. App developers can access the platform's functionality and services through its interfaces without having to know what goes on inside. The platform becomes a black box to app developers. It is much like driving a car: All you need to be able to drive is know how to use the car's controls; knowing how it works under the hood is not necessary. This allows app developers in a platform ecosystem to focus exclusively on their own app development work and have confidence that compliance with the platform's interface specifications will guarantee interoperability with the platform. Encapsulation of the platform therefore hides complexity (Ethiraj and Levinthal, 2004b). The same logic applies to encapsulation of individual apps. Encapsulating complexities and implementation details within subsystems not only preserves secrets about the functioning of the platform and individual apps, it also alters the competitive dynamics with rival platforms, as the next section of this book elaborates.

Decoupling requires creating and maintaining rigorous partitions of design information into hidden and visible subsets (Baldwin and Clark, 2000, p. 413). Decoupling can be accomplished in two ways: (1) splitting the ecosystem into high and low reusability subsystems and (2) specifying permissible assumptions.

5.8.1.1 Decomposition by reusability and variability – the five magic rules

The first way to decouple subsystems in a platform ecosystem is to partition it into stable and variable blocks, and high- and low-reusability blocks of functionality (Baldwin and Woodard, 2009). This requires evaluating each major element of functionality that is a candidate for consideration for inclusion in or exclusion from the platform core. Two premises of platform-based development can help guide this assessment:

- *Economies of scale across the ecosystem.* A platform achieves economies of scale in facilitating the development of complementary apps primarily by incorporating functionality that is reused by many apps.
- *Reduced costs of developing apps for the platform.* When including a function reduces app developers' costs of developing around the platform, it should be included in the platform.

Using these two guiding principles provides the following five heuristics—the magic rules—for deciding what functionality to include from a platform core.

1. *High-reusability functionality goes in the platform.* The subsystems with high reusability (i.e., shared by many apps) should go into the platform and those with low reusability (i.e., unique to a few apps) should be implemented as apps. Keeping the high-reusability functions within the platform allows that functionality to be improved much more rapidly because the subsystems used to implement them are colocated *within* the platform and can be tightly integrated. The platform itself can therefore be highly monolithic internally even though it is externally modular (i.e., when viewed by an app developer). Even the platform core of a modular platform can change over time; only the platform's interfaces need to remain stable. As long as the interfaces described in the next section remain untouched, internal functionality enhancement within the platform can be leveraged by all apps in the ecosystem without the need for much coordination between the platform owner and app developers.
2. *Generic functionality goes in the platform.* Functionality that is generic enough to be useful to many apps should go in the platform. These are the common components and functionality shared by many app implementations (Boudreau, 2010). This includes investments by a platform owner in foundational infrastructure, and shared assets and services that all app developers can potentially leverage. The exception to including only general functionality that can be used by all apps is functionality that a platform owner would want to completely control for strategic reasons (e.g., phone calls in iOS). However, nongeneric functionality that is idiosyncratic to a few apps should be implemented as apps but kept out of the platform.
3. *Any interfaces to the platform are an integral part of the platform.* Both the core codebase of the platform and its interfaces should be treated as inseparable parts of the platform because they have relatively long life spans and meet the criterion of high reusability.
4. *Stable functionality goes in the platform.* Stable functionality should go in the platform but immature, changing, evolving functionality should ideally be implemented as apps. Today's immature

functionality might become tomorrow's stable functionality; strategies for absorbing it into the platform without compromising architectural decoupling are discussed in Part IV of this book.

5. *Functionality with the highest uncertainty remains outside the platform.* The value of diverse experimentation is the highest for unproven functionality, where multiple solutions are viable and it is unclear upfront which solution or approach the market will prefer. Such functionality should be kept out of the platform and instead left to app developers. Including it in the platform risks premature commitment to what might turn out to be an inferior solution. Leaving it out of the platform can attract many diverse solutions from competing app developers aiming to meet the demand for it. The market will determine the winner out of the diverse experiments, provided appropriate incentives are in place. (It might be tempting for a platform owner to eventually subsume the winning solution into the platform, but this breach of trust with app developers can be a slippery slope, as we subsequently discuss.)

There are two exceptions to the fifth rule. First, a platform owner should integrate functionality that it deems critical to the attractiveness of the platform to end-users. If the platform owner also tweaks platform governance to match, it can foster strong healthy competition among app developers to improve the inhouse, native functionality. An example of this in mobile computing platforms is the inclusion of an email app and a Web browser in the iOS, Android, Microsoft Mobile, and Blackberry platforms, both of which were then also opened to app developers attempting fiercely to improve on them. Second, a platform owner should include functionality in the platform that it deems critical in terms of end-user expectations and norms set by rival platforms.

In summary, as shown in Figure 5.24, the platform should emphasize low-variety, high-reusability functions and relegate high-variety, low-reusability functions to apps.

5.8.1.2 Decoupling by specifying allowable assumptions

The second way to decouple subsystems is how a platform owner can explicitly specify assumptions that an app can make about the platform, and assumptions a platform can make about apps. Explicitly specifying allowable assumptions weakens dependencies among them. Neither apps nor the platform

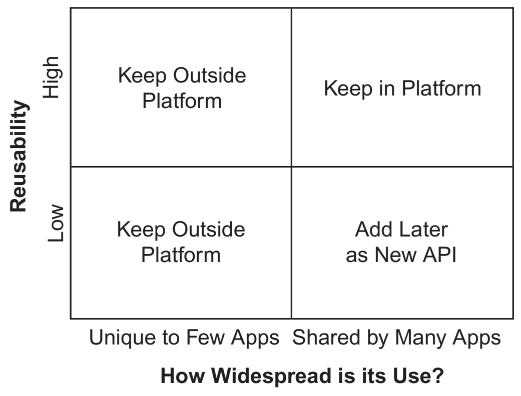


FIGURE 5.24

Strategically distributing platform functionality between a platform and apps.

are allowed to make any assumptions about each other beyond their visible information. Therefore access to a platform's functionality is restricted to its interface. This rule keeps the developers of the platform and the apps from relying on each other's hidden information as they do their own work and keeps them reliant on only the explicit, visible information provided by their architecture. Since the design of any app does not rely on the platform's hidden information, neither app developers nor the platform owner are affected by changes in the other's internals as long as the visible information remains frozen.

More decoupled apps are easier to comprehend, easier to change, and easier to test. If an app is weakly coupled to the platform, its interdependencies are completely contained in its interface to the platform. The app can therefore be implemented with limited knowledge of the platform's innards, relying largely on the interface specifications provided by the platform owner. In contrast, tightly coupled apps are designed to work closely with the platform and cannot be modified in isolation from the platform. This requires the app developer to understand the platform and the app as a whole, and possibly also the dependencies and interactions that the app might have with other apps in the platform ecosystem. Internal changes in the app can then cause errors in the platform's operations and changes in the platform can cause the app to malfunction. This creates a time-consuming coordination overhead that can slow down the app's evolution. Therefore decoupling decreases the likelihood of app malfunctions and speeds up its evolvability. The decreased need for an app developer to coordinate with the platform owner and to understand the inner workings of the platform allows the app to be changed more frequently and at a lower cost to the app developer. Similarly, changes within the platform are less likely to break the app as long as the platform owner remains faithfully compliant with the frozen interface specifications to the platform, as described in the next section.

5.8.2 Interface standardization

Interfaces are like a treaty between a platform and apps in a platform's ecosystem (Baldwin and Clark, 2000, p. 73). They are the platform's *visible* information. They specify the basic set of rules to ensure the technical interoperability of apps with the platform (Baldwin and Woodard, 2009; Boudreau, 2010).[5] A platform's interface standards are therefore sometimes referred to as the platform's design rules. *Interface standardization* refers to the degree to which an app communicates, interoperates, and exchanges data with the platform using predefined, well-specified interfaces, protocols, and rules that are not allowed to change. Interface standardization therefore means that all information about how any app must communicate and interact with the platform is explicitly documented in writing by the platform's owner and then frozen.

From an app developer's perspective, a platform's interfaces are the only visible embodiment of a platform seen and experienced by them. Apps are forbidden from communicating with the platform outside of its interfaces. The entirety of the interaction between a platform and apps occurs through the platform's interfaces. For all intents and purposes, from an app developer's perspective the interfaces *are* the platform. The platform's interfaces describe to all apps what they need to know to invoke and access the services and capabilities of the platform. The interface also specifies protocols, which

[5]Interfaces within the platform and within individual apps are their *intra*system interfaces. We concern ourselves with not intrasystem interfaces but rather solely with *inter*system interfaces—ones that connect an app to the platform, and both to external systems outside the ecosystem boundary—throughout this book.

describe how any back-and-forth sequence of interactions between an app and the platform should occur. A platform's interfaces therefore not only enable communication among the apps and platform, but also govern and discipline it (Langlois and Garzarelli, 2008).

Interface standardization does not require compliance with any public standards (such as CORBA) or joint development, just with the platform's own interface standards. In general, compatibility with rival platforms is undesirable for strategic reasons. Incompatibility among rival platforms' interface standards is a powerful strategic tool that subsequent chapters discuss in detail. Agreeing to common standards and interlinking rival platforms is likely to be profitable for all platforms only if no platform has a technological edge (Farrell and Saloner, 1985, 1992; Rohfls, 2003, p. 197).

Standardized platform interfaces are the key to opening up a platform's architecture. Openness of a platform architecture simply refers to whether outsiders need permission from the platform owner to build on it (Evans et al., 2006, p. 12).[6] If a platform is open, it means that outsiders can access its visible information using its interfaces to build complementary apps that augment the platform. A recent study of major platforms that thrived between 1990 and 2004 found that opening up platforms to outside developers increased the rate of innovation around that platform by 500% (Boudreau, 2010). Platforms can be open or closed to varying degrees, with most platforms neither completely open nor completely closed (Boudreau, 2010). For example, Apple's iOS platform is more closed than Google's Android platform because app developers need approval from Apple to distribute apps for the iOS platform. However, platform openness is as much a platform governance choice (Chapter 6) as it is a technical or architectural choice. But a platform's interfaces must be open to outsiders for an ecosystem to emerge. An interface is open standards-based when it is well documented, available publicly, nonproprietary, and not subject to intellectual property restrictions. Interfaces to a platform need not be open standards-based to foster vibrant ecosystems around them; they can be proprietary and open. Interface standardization therefore allows an app developer to treat the platform as a black box and focus her attention only on her own app. If the platform owner clearly and explicitly communicates how outsiders can build on the platform's capabilities in their own work, it can potentially become the basis for large-scale innovation around the platform (Meyer and Selinger, 1998). Such interface standardization must be ecosystem-wide but need not be applicable outside an ecosystem. Standards for the interface between a platform and app have three desirable properties to make them an effective coordination tool: precision, stability, and versatility.

5.8.2.1 Precisely documented

Interfaces of the platform are the glue that binds apps with a platform. Interface standards to a platform must be clearly specified, unambiguous, well documented, and stable to be useful to app developers. Ideally, the platform's visible information and interfaces are comprehensively and precisely documented, leaving no room for misinterpretation by app developers. They must also clearly lay out the design rules that describe how an app and the platform communicate, interact, exchange information, and interoperate. Information that is of relevance to more than one app must be nearly completely specified in the architecture design phase of the platform. As long as an app complies with the interface standards

[6]Openness of architecture should not be confused with open-source and closed-source architectures. Open source refers to platforms that make their *source* code freely available for others to use, augment, and build on. Google's Android platform is an example of an open-source platform.

prespecified by a platform owner, it should be able to interoperate with the platform independent of the app's internal implementation. A simple example of a standardized interface is a USB port. As long as a device (e.g., a printer, scanner, or camera) complies with the documented USB standards in how it communicates with a PC, it is able to communicate with it independent of the internal design of the device. Design rules provided by interface standards are particularly important after the initial platform architecture is chosen because they guide the implementation of the architectural decisions (such as decoupling). Their precision can substantially reduce the platform-wide inconsistencies between envisioned architecture and realized architecture, discussed earlier.

In platforms, such interface standards are often proprietary to individual platforms. But they must be documented in sufficient detail to provide app developers all the information they need to ensure interoperability with the platform. By isolating the interdependencies of the app from the rest of the ecosystem, platform interface standardization increases the flexibility of what an app developer can do and gives her more freedom to design and implement the functionality of the app. The utility of interface standards is that they allow app developers and a platform owner to leverage each other's strengths without sacrificing autonomy to innovate in their own work. Precisely documented interfaces between a platform and apps therefore help both app developers and platform owners innovate more effectively around the platform, guide interoperability at the app level, and provide an implicit coordination mechanism between the platform owner and app developers (Dougherty and Dunne, 2011; Sanchez and Mahoney, 1996). Thousands of app developers can then simultaneously work on their own apps in blissful ignorance of the technical details of the rest of the platform ecosystem.

A dominant way to provide such "hooks" for apps to interface with a platform are APIs. An API is simply a standardized interface designed to accept a broad class of apps (or add-ons/extensions/modules). APIs allow app developers to use the platform's capabilities without having to concern themselves with how those capabilities are implemented within the platform. App developers know how the platform behaves but do not need to know how it works. APIs play a central role in minimizing duplication of effort by app developers (Evans et al., 2006, p. viii). An API can include specifications for variables, routines, data structures, protocols, and object classes and behaviors. These APIs provide app developers well-defined means to access the platform's shared libraries, protocols, functions, and specific capabilities that they can use as a starting point for implementing their apps. An API specification can be compliant with a public open standard or protocol, or be platform-specific in the form of a platform software framework, libraries of a programming language (e.g., the Java API), or commands to invoke a Web service. Examples of widely used public API technologies include Pragmatic REST, JSON, and OAuth (for authentication without password propagation across the Internet). An open API allows an outside app developer to build on and extend the platform's native functionality by tapping into the platform's generic portfolio of services through a documented interface. An API or interface standards of a platform need not be completely open to all app developers. Openness of an API is a matter of degree; it can reside along a *continuum* ranging from completely open to completely closed. An API therefore enables a one-to-many relationship between the platform and apps. The advantage of APIs is that they allow extensions to the platform that could not even have been envisioned at the time the platform was created. They are also rich in embedded real options (see Chapter 8).

An API can become an industry standard in a platform's market, but can still be under the complete control of a single platform owner. (Maintaining such control, however, requires careful attention to platform governance.) An industry standard is a detailed specification that is agreed on by multiple players in the industry. Such standards can be formal *de jure* standards such as HMTL 5 or arise *de facto* through mass adoption (e.g., PDF format). While *de jure* standards are often created by industry consortia or regulatory bodies (such as IEEE or W3C), *de facto* standards often begin as a proprietary interface, protocol, or API that subsequently gains critical mass. Platform owners who maintain control over a *de facto* platform interface standard can enjoy considerable market power for long periods without getting under the skin of antitrust regulators.

Another way to implement interface standardization is through protocols. A protocol is a defined system of rules and semantics for exchanging messages between a platform and apps. These protocols may be platform-specific or based on public standards (such as HTML 5).

5.8.2.2 Frozen

Interfaces provided by the modularly architected platform must also be stable and unchanging (Baldwin and Clark, 2000, p. 76). Interface standardization provides an implicit coordinative function between the platform owner and app developers by documenting dependencies between a platform and apps. It specifies what assumptions app developers are allowed to make about the platform and what assumptions the platform owner can make about individual apps. These assumptions should not be allowed to change during the implementation of individual apps. Changes in the platform's interface standards can violate such assumptions, and in turn break apps. Even when they don't violate any previous assumptions that app developers were allowed to make, the added overhead for constantly verifying the safety of making an assumption about the platform can impose considerable unnecessary overhead on app developers while accomplishing little. Interface standards usually have long lifecycles and are loaded with legacy baggage. Think of standards that have persisted for decades: the VGA connector, 30-pin iPhone connector, QWERTY keyboard, and ASCII text. The widespread use of an interface standard by its adopters makes it difficult to dislodge even if a new replacement standard is technically superior.

An app must be able to assume that information about how it will interface with the platform will remain unchangeable. As long as a new version of an app (or new apps) complies with the interface specifications provided by the platform owner, any changes within the app cannot compromise its ability to interoperate with the platform. It is therefore important that any possible interdependencies between a platform and apps be captured in the interface specifications.

Paradoxically, this inflexibility in a platform's interface specifications dramatically increases flexibility in what app developers can do within their apps. New functionality that is added to a platform should therefore avoid touching existing interface specifications because stable APIs lead to predictability (Iansiti and Levien, 2004, p. 53). Instead, adding new functionality to a platform usually requires adding new APIs. This is why Apple's iOS had almost 15,000 APIs by 2013. Therefore, stability of interfaces used by the platform is critical to minimizing the interdependence between the platform and apps. They also must remain consistent across successive generations of a platform (Iansiti and Levien, 2004, p. 88).

5.8.2.3 Versatile

However, to support the ecosystem's evolution and the evolution of apps, such interfaces should also be versatile. Versatility means that the interfaces to the platform should be able to incorporate linkages in the future and permit future apps unforeseeable by the platform's original designers (Baldwin and

Woodard, 2009). Although stable interface specifications minimize the need for app developers to coordinate their app design decisions with the platform owner, they can severely constrain all subsequent designs to be bounded by those specifications. The more precise a platform's interface specifications, the more inflexible is this constraint. The more general a standard (HTML 5 vs. Flash), the greater its versatility.

To prevent the inflexibility of interface specifications from stymieing the evolution of the platform, good platform architecture should bundle together the functionalities that can most benefit from subsequent improvement in the platform itself. So functionalities that are likely to change together and are used by multiple apps should belong in the platform codebase itself. Monolithic-ness within the platform is therefore generally desirable. Put another way, the partitioning of the initial architecture of the ecosystem should be done in a way that places the functionality potentially shared by many apps into the platform core that is intended to be stable and the rest into a set of complementary apps that are encouraged to vary (Baldwin and Woodard, 2009). Versatility of a platform's interfaces is therefore inseparable from decoupling in its initial design.

5.8.2.4 Compliance with interface standards

Interface standards are like traffic lights; they simplify coordination only as long as everyone follows the same rules. Each driver must both know and follow the rules. To be effective coordination devices, interface standards must be binding on both the platform owner and app developers; app developers must obey them and also expect others to obey them (Baldwin and Clark, 2006; Ostrovsky and Schwarz, 2005). The degree to which an app actually complies with interface specifications encapsulates its behavior in practice. This in turn can generate variability in compliance among different app developers, with some app developers closely complying with the standards prescribed by the platform owner and others doing it half-heartedly. This variance in interface standards compliance among apps coupled with differences in app microarchitectures often creates variability in the architectural properties of different comparable apps even within the same platform. However, such compliance cannot readily be enforced, is rarely contractible, and is costly to verify (Ostrovsky and Schwarz, 2005).

Compliance with a platform's interface standards requires a carrot—demonstrable value and benefit to app developers—rather than a stick. What's in it for an app developer? Interface standards compliance decreases the complexity that an app developer must cope with, and the depth of knowledge about the platform that she needs to possess in doing her own work. This rationale, however, works better for established interfaces that have already been adopted by other app developers in a platform than it does for new interface standards. Early adoption of a new standard or API by an app developer can be particularly risky, especially one that develops the app on multiple rival platforms (i.e., produces a multihoming app). The platform owner must therefore credibly assure app developers against the risk of being stranded with high dependence on a platform interface standard that could be abandoned at will and also communicate precisely how it can enhance the app developer's own work. The platform owner must make it easier and less costly for app developers to adhere to the platform's interface standards and specifications. This requires investments by the platform owner in creating good app testing mechanisms that app developers can use themselves, and also in tools to help the platform owner determine whether an app complies with the critical interface specifications. Thoughtful platform governance—especially control mechanisms, described in the next chapter—can further alleviate this compliance problem. Enforcing a platform's design rules and interface specification is therefore an important role of governance (Baldwin and Clark, 2000, p. 13).

Table 5.4 Guiding Principles Influenced by Platform Architecture Decisions

Principle Affected	Platform Architecture	App Microarchitecture
Red Queen effect	●	●
Chicken-or-egg problem		
Penguin problem		
Emergence	●	●
Seesaw problem	●	
Humpty Dumpty problem	●	
Mirroring hypothesis	●	
Coevolution		●
Goldilocks rule	●	●

Different architectures can be used to implement the same functional requirements in a platform, and platform designers and app developers have relatively few constraints that keep them from freely picking one set of architectural choices over others at the outset. But these early choices can have strikingly different evolutionary consequences. Architectural decisions by platform owners and app developers therefore influence the evolution of platforms, apps, and entire ecosystems. Table 5.4 provides a preview of which of the key evolutionary principles are affected by their architectural choices. Parts III and IV of this book explain these ideas in depth.

CHAPTER SUMMARY

- *Complexity stymies innovation.* Platform ecosystems are complex to begin with and, as their complexity grows, interdependence among their many parts becomes paralyzing. Co-innovation is not additive but multiplicative, which stymies the prospects of even capable participants producing joint innovations. Architecture can reduce ecosystems' structural complexity but not their behavioral complexity.
- *Architecture is a platform's DNA that imprints evolvability.* Architectural choices irreversibly preordain the evolutionary trajectories open and closed to a platform's ecosystem. Platform ecosystems are intentionally designed complex systems composed of many interacting parts. Platform architecture specifies what these parts are, how they connect, and what they can and cannot do. These properties are inherited by apps in their own microarchitectures as well, but imperfectly so. Envisioning ecosystem architecture requires adopting a telescopic view of platform architecture; envisioning app microarchitectures requires adopting a microscopic view of platform architecture.
- *Architecture precedes organization.* Platform architecture determines whether transaction and coordination costs can be reduced sufficiently to make an ecosystem viable. It shapes the ability of outside app developers to join a platform and provides incentives to join.

- *Architecture partitions and reintegrates a complex system.* Architecture must serve two purposes: (1) it must partition a complex ecosystem so it can be decomposed into relatively autonomous apps and the platform and (2) it must facilitate ongoing systems integration among them so they can be put back together as a cohesive ecosystem.
- *Apps have internal and external microarchitectures.* Their internal microarchitecture is defined by how their four functional elements are split across the Internet. Their external microarchitecture is defined by how these functional elements are divvied up between the app and the platform.
- *Platform architectures have four desirable properties.* They should be simple, resilient, maintainable, and evolvable. The structure, governance, and evolution of modern cities offer useful lessons for designing vibrant platform ecosystems.
- *Modularization endows these desirable properties to architectures.* Modularization of architectures refers to how they can be designed so that changes within the platform or an app do not have a ripple effect on the rest of the ecosystem. Modularization is accomplished by decoupling the platform from apps and then freezing well-documented specifications for how they connect.
- *Decoupling in architecture uses two simple rules.* First, partition the ecosystem into low-variety, high-reusability functions that go into the platform core and high-variety, low-reusability functions that must remain outside it. Second, specify assumptions that apps can make about the platform and vice versa.
- *A platform's interfaces follow three criteria.* They must be precise, frozen, and versatile. Getting app developers to comply with them requires a carrot rather than a stick.
- *A platform's interface standards are like traffic lights.* They are useful as a coordination device only when everyone follows the same rules. Enforcing compliance is an important role of platform governance. A platform owner can invest in tools (such as developer toolkits, reference models, simulators, and module testers) that make it less costly for app developers to comply with them.
- *Goldilocks should not be forgotten.* Modularization has upsides and downsides both for app developers and platform owners. Modularization is usually worth it when a platform's markets are heterogeneous, riddled with technology and market unpredictability, and require eclectic knowledge that the platform owner does not have inhouse. The optimal level of modularity is *just enough* modularity. We discussed how to determine what is just the right level of modularity for a platform and for an app.

Even the most thoughtful platform architecture cannot nurture a vibrant ecosystem unless it is governed effectively. We turn our attention to platform governance in the next chapter. It provides a three-dimensional framework for platform governance that encompasses who decides what, mechanisms of control, and pricing. It also describes precisely how platform governance can be aligned with its architecture to help realize its strategic potential.