

LOCALISM IN AMERICA



**Why We Should Tackle
Our Big Challenges at
the Local Level**



Localism in America

WHY WE SHOULD TACKLE OUR BIG CHALLENGES AT THE LOCAL LEVEL

FEBRUARY 2018

SAMUEL J. ABRAMS

JAY AIYER

KARLYN BOWMAN

WENDELL COX

ROBERT DOAR

RICHARD FLORIDA

TORY GATTIS

NATALIE GOCHNOUR

MICHAEL D. HAIS

JOHN HATFIELD

FREDERICK M. HESS

HOWARD HUSOCK

ANDREW P. JOHNSON

LEO LINBECK III

THOMAS P. MILLER

ELEANOR O'NEIL

DOUG ROSS

ANDY SMARICK

ANNE SNYDER

MORLEY WINOGRAD

EDITED BY JOEL KOTKIN AND RYAN STREETER

© 2018 by the American Enterprise Institute. All rights reserved.

The American Enterprise Institute (AEI) is a nonpartisan, nonprofit, 501(c)(3) educational organization and does not take institutional positions on any issues. The views expressed here are those of the author(s).

Refactoring for Subsidiarity

Leo Linbeck III

If debugging is the process of removing software bugs, then programming must be the process of putting them in.

—Computer scientist Edsger Dijkstra¹

Over the past 60 years, America grew vast federal agencies and spent trillions of dollars to fight wars on drugs, poverty, crime, and terror and to expand federal control over education and health care. The results of centralization and expansion of the federal government raise serious questions about the ability of the large-scale federal government to solve problems. In fact, many problems are worse, some dramatically so, than when the expansion began.

Along the way, virtually all institutions—whether in politics, government, business, religion, or civil society—have lost significant amounts of credibility and trust.² Elites in our society—those responsible for these very institutions—are baffled by this turn of events and are at a loss about what to do.

What is needed is a new approach to changing human institutions that reverses relentless centralization. This essay proposes one. This new approach uses concepts and frameworks borrowed from software engineering but is inspired by the principle of subsidiarity, which has its origins in the ecclesiology of the Catholic Church. The *Oxford English Dictionary* defines subsidiarity as “the principle that a central authority should have a subsidiary function, performing only those tasks which cannot be performed at a more local level.”³ In the language of software engineering, control should be pushed to “lower-order components” (e.g., a municipal government or

individuals), and “higher-order components” (e.g., the federal government) should not interact or interfere with those lower-order components except through a clearly defined interface.

To implement subsidiarity, we can look to software engineering for tools and patterns. In particular, the concept of refactoring provides an alternative way to think about change and adaptation in institutions. Changing large-scale software systems is hard, as is changing large-scale human institutions. For policymakers to decentralize power, the only viable approach is to employ a greenfield strategy. Policy architects should stop trying to fix problems from the center and instead design greenfield strategies that create regulatory competition.

Models of Change

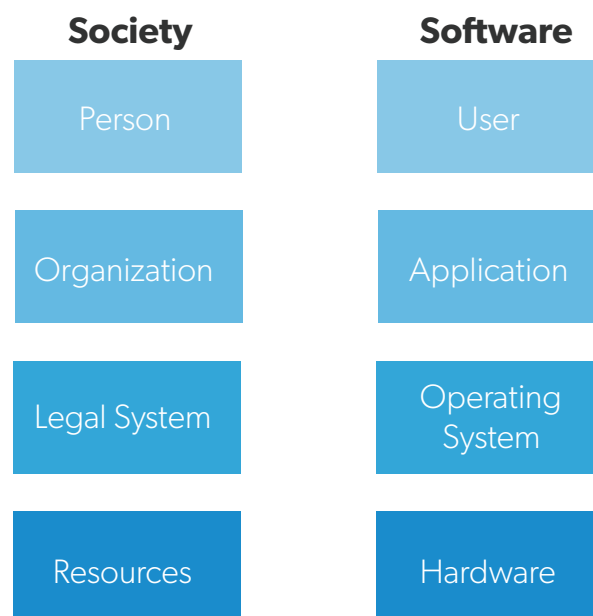
Our current political dysfunction is less a product of evil intentions, as is sometimes alleged, than misunderstanding the consequences of our remarkable success as a nation. When we consider any human organization—whether a business, city, or republic—a cycle emerges: Success leads to growth. Growth leads to scale. Scale leads to centralization. Centralization leads to complexity. Complexity leads to failure. Failure creates the imperative to change.

Figure 1. The Cycle of Society and Software



Source: Author.

Figure 2. Society Versus Software



Source: Author.

(See Figure 1.) This is the cycle of society. It is also the cycle of software.

Change is hard, especially in large-scale, complex, centralized systems. In society, there are two modes of change: reform and revolution. Reform identifies problems or opportunities and creates, updates, or deletes text in the existing set of laws to fix the problem or take advantage of the opportunity. Revolution throws out the existing set of laws and substitutes a new set of laws.⁴

In software, those two modes also exist (Figure 2). Reform is like fixing bugs or adding features. Problems (bugs) are patched or opportunities (features) are added to the existing program by creating, updating, or deleting lines in the existing code. Revolution is like switching from a personal computer to a smartphone.

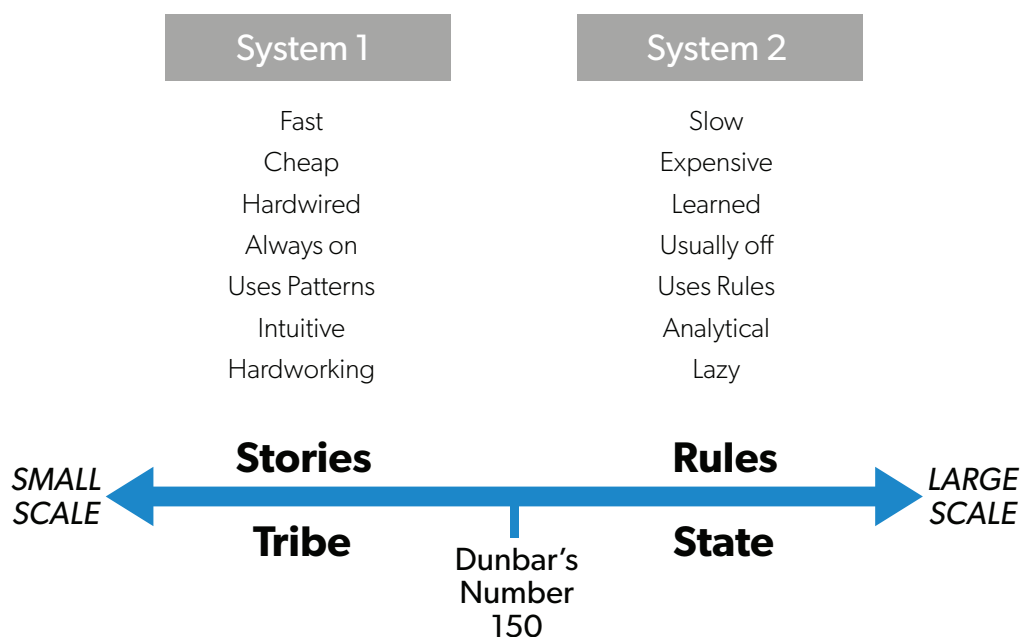
But in software, there is a third mode of change, one that software engineers call *refactoring*. This mode is largely unknown to nonprogrammers and usually completely invisible to users. Simply put,

refactoring changes how a software system works without changing what it does—its external behavior. In software engineering, examples include improved readability and reduced complexity of code. Refactoring does not have an analog today in the political and policy world, which tends to categorize change as either reform or revolution. And as we explore the refactoring concept, it will be clear—especially to software engineers—where the analogy breaks down. But the refactoring concept is a new way of thinking about creating policies of sustainable, maintainable human flourishing.

Centralization, Scale, and Complexity

David Lilienthal, former chairman of the Tennessee Valley Authority, once remarked on the perils of centralization, noting, “Centralization at the national capital or within a business undertaking always glorifies the important pieces of paper. This dims the

Figure 3. Systems of Thinking



Source: Author.

sense of reality.”⁵ According to Daniel Kahneman, our brain has two systems of thinking. He refers to these as “System 1” and “System 2.”⁶ System 2 is our slow, rational, reasoning, rule-based system, and System 1 is our fast, intuitive, pattern-matching, narrative-driven system. System 2 is crucial to our ability to adapt, but most human action is driven by System 1. This is in part because narratives are the language of the shared values that drive communities and businesses to change. As renowned management educator Peter Drucker noted famously, “Culture eats strategy for breakfast.”⁷

Our brain has another important characteristic: We can maintain stable social relationships with around 150 people—Dunbar’s number, named after British anthropologist Robin Dunbar.⁸ This number shows up everywhere in society: in the military (the size of a company, the core unit of an army since Roman times); business (the upper limit on the number of shareholders of a family business); politics (the size of Congress when political parties first emerged); social networks (Dunbar showed that even if you had a million Facebook “friends,” you really have about only 150 real friends); and so on.⁹ For better or worse, we live System 1 narratives in Dunbar-sized tribes.

But as human organizations scale, we go from using System 1 (stories) to System 2 (rules) because we interact more with strangers—people outside our tribe do not know our tribal identity (expressed as our tribe’s stories, so outsiders do not know how we will behave (Figure 3). In addition, scale leads to the emergence of hierarchy as a way to control the number of relationships so organizations and groups’ networks maintain fewer than 150 people—Dunbar’s number.

A hard truth is that a successful organization may grow, but the number of people with whom we can maintain stable relationships does not. This truth means that organizations naturally centralize as they scale—the central tribe maintains control and power through loyalty and trust. The “center” promulgates rules that everyone else is expected to follow while managing its own affairs according to the tribal narratives it maintains—rules for thee, discretion for me.

Thus, throughout human history, as organizations grow larger, they naturally centralize power and authority. In theory, this centralization might be fine, if it were not for another problem: As organizations grow in scale, they grow in complexity, and that complexity soon exceeds the ability of any person or tribe to manage the organization from the center.

The Problem with Complexity

Niklaus Wirth, a pioneer in computer science and software engineering, often noted the perils of misinterpreting software complexity as sophistication and convenience: “Increasingly, people seem to misinterpret complexity as sophistication, which is baffling—the incomprehensible should cause suspicion rather than admiration. Possibly this trend results from a mistaken belief that using a somewhat mysterious device confers an aura of power on the user.”¹⁰

When the United States Constitution was written in 1789, the US population was about four million people.¹¹ By 1860, the population exceeded 30 million and by 2010 was more than 300 million. Along the way, the number of laws and regulations has exploded (as has the number of lawyers). The first Congress passed laws that totaled 225 pages.¹² By 1936, the *Federal Register* was published for the first time, with a length of 2,620 pages. In 2015, the *Federal Register* had grown to 81,402 pages. Each page has about 100 lines, so it is probably more than 10 million lines of text.¹³

When Linus Torvalds published the first version of the Linux operating system in 1991, it had 10,239 lines of code.¹⁴ By 2001, Linux had grown to 3,377,902 lines of code, and version 4.1 of Linux, which was released in June 2015, had more than 19.5 million lines of code and 14,000 contributors.¹⁵

It is safe to say that no human being has read every page of the *Federal Register* or every line of Linux code. Both of these systems are way beyond the scale of a single person. And this simple and indisputable fact means that no one can deal directly with these systems. Instead, this complexity must be managed by employing a number of strategies.

The first strategy is to break a big, complex system

into smaller, simpler subsystems and carefully define the way those subsystems interact. Even after such a breakdown, if a system continues to grow, the subsystems themselves will become too large and must be broken down further into smaller sub-subsystems. Through this subdivision process, we not only reduce the complexity of the subsystems but also increase the number of people who can deal with the problem.

In political theory, this strategy gives rise to concepts such as localism, federalism, and the compound republic. Higher levels of government have clear responsibilities and scope while lower levels of governance are smaller, simpler, and closer to the people; focused on their own local issues (e.g., snow removal is more important in Boston than Houston; the opposite is true for hurricane preparation); and easier to change.

Yet creating a hierarchy of subsystems is not enough. There must be a commitment to subsidiarity—that is, pushing control as low in the hierarchy as possible. We do not reduce complexity if we create additional subsystems but still control everything from the center. In fact, it makes the complexity problem worse. In programming, interaction between components is managed through an *interface*. Higher-order components, for instance, cannot directly access and modify the properties of lower-order components—they must access those properties through the interface of that lower-order component. This rule—which, perhaps counterintuitively, limits the power of the higher-order component—is a way to keep complexity under control and is especially important when debugging programs.

The First Amendment is an example of such a restrictive interface; it begins with the words “Congress shall make no law”—a beautiful example of the software principle called “separation of concerns.”¹⁶ Separation of concerns is needed because scale and centralization lead to complexity. They do this naturally and without any intention or effort on the part of those in charge. But complexity inevitably leads to failure, so we must expend effort to control complexity.

That effort to control complexity in the face of scale—the effort to maintain subsidiarity while growing—is what refactoring is all about.

Refactoring

Refactoring, defined by the “father” of refactoring, is said to be “the process of restructuring existing computer code without changing its external behavior.”¹⁷ The idea is to make changes that improve the code without changing the way the user interacts with the software system.

A real-world, non-software analogy may help. To drive a car, a driver uses three basic controls: an accelerator pedal to go, a brake pedal to stop, and a steering wheel to control direction. These are the control elements of the “user interface” of a car (at least one with an automatic transmission).

Refactoring an engine can lead to many changes: replacing a carburetor with a fuel injection system, a gasoline engine with an electric motor, a hydraulic with an electronic steering system, and so on. Those changes can be large (gas versus electric) or small (single-point versus sequential fuel injection), but the key is that they do not change the way the driver interacts with the vehicle. There can be radical changes under the hood, but there is still just the gas, brake, and wheel.

In software, refactoring has a similar goal: make improvements under the hood but do not change the way the system works from the user standpoint. There are dozens of refactoring techniques, and software engineers come up with new ones from time to time. Decades of experience in software development have given us a powerful tool kit for developing and—more importantly—maintaining high-quality software. And all new software systems have been designed to support some degree of continuous refactoring.

But what about “legacy” systems? Those systems were originally developed before refactoring became widely understood and supported and are riddled with bugs and shortcuts, and attempts to change the code often create more bugs than they fix. These are important systems that millions of people use; what should we do with them?

The general consensus is that legacy software systems should be left pretty much as is, and we should just have a few experts focus on fixing the really egregious bugs that pop up from time to time.¹⁸ They

work, and the short-term cost and risk of breaking them are not worth the long-term benefit of refactoring the code.

Our “legacy” political systems are much the same, creating a nearly uncountable number of laws, rules, and regulations at the federal, state, and local level that are so big and complex that we hardly know where to begin. The short-term cost and risk of changing those systems are almost always too big compared to the potential long-term cost. Sure, eventually it will fail (perhaps catastrophically), but that will be someone else’s problem to deal with.

So are we just stuck with legacy systems that are doomed to fail?

The Greenfield Strategy: Removing Complexity

Not necessarily, but we have to do something different. We need to use a special type of refactoring strategy, one that creates a new system that provides the same functionality to the user, but with a completely different code base. This strategy does not affect the user—the gas, brake, and steering wheel are still there—but it does make big changes under the hood.

But there is a problem with making these big transitions from older systems: They often break special features that users have added to their system. As a result, software companies rarely if ever force users to upgrade their system. Instead, they announce the availability of the upgrade and then allow users to choose when to make the switch and allow application developers to update and upgrade *their* systems to adapt to the new code base.

This refactoring strategy—which we will refer to as a “greenfield” strategy—is a way to make big changes to legacy systems. Put simply, a greenfield strategy is a way of creating an alternative system to an existing system and then letting users—or citizens, in the case of governance—choose between the systems. Engineers can start with a blank sheet of paper and design the best way to give users what they want, but at the end of the day the user decides whether and when he or she wants these additional benefits.

Greenfield strategies are perhaps the most important sources of innovation in software. Each major advancement in software engineering—including the IBM System 360/Operating System (OS), VAX, CPM, MS-DOS, Mac, Windows, Netscape, Google, Facebook, iOS, and Android—started as a greenfield project that ultimately replaced an older legacy system because users liked it better.

The opportunity in governance is similar. We have failing legacy systems because they are too large and ossified to change. Big changes are needed, but these systems are also crucial components of our society, and the long-term benefits do not justify the short-term risks and costs of change—especially from the perspective of the elected official, government bureaucrat, or fee-seeking consultant.

With a different strategy—a greenfield strategy—we can change that cost-benefit ratio and make it possible to do a major refactoring of governance. The essential elements of a greenfield governance strategy could be as follows:

1. It maintains the existing legal authority structure. There are strategies that move authority from one level of government to another—from federal to state, state to federal, local to state, and so on. A greenfield strategy does not do this; it takes the existing authority structure as a given. Using our software analogy, this is like keeping the same hardware platform.
2. It gets enacted through legislation. There are strategies that go through the courts, a referendum, or civil disobedience. A greenfield strategy does not do this; it uses the existing legislative process to make the necessary changes. Using our software analogy, we make changes in our code, not in how users behave or the groups that oversee computer standards.
3. It competes with an existing “brownfield” regulatory regime. There are strategies that try to maintain a single regulatory regime and improve that regime. A greenfield strategy does not do this; it sets up a separate

regulatory regime that is fiscally and administratively independent and allows competition between the two regimes to drive innovation and improvement. Using our software analogy, this is like releasing a new operating system while continuing to ship and support the existing system.

4. It leaves the choice of regime and timing of when to switch (if ever) to the groups subject to regulation. There are strategies that change regulations and have those changes apply to every regulated party. A greenfield strategy does not do this; it gives regulated parties (“customers”) the right to switch to a different regulatory regime if they do not like the change. By giving customers the choice of

regime, it forces regulators to deliver a superior service or see those customers switch to the other regime. Using our software analogy, this is like giving users the choice of whether and when to upgrade their operating system.

These four criteria are essential if we are to undertake a major refactoring of governance. And for that refactoring to be successful, the alternative regulatory regime must have a more modern, decentralized, and competitive architecture. It must embrace the principle of subsidiarity, moving decision-making from higher- to lower-level components. And it must deliver a superior product—a truly better operating system—to attract users to its system.

Table 1. Greenfield Strategies

Domain	Brownfield	Greenfield
K–12 Education	School Districts	Charter Schools
Pensions	Defined Benefit	Defined Contribution
Utilities	City Utility Department	Municipal Utility District
Trash Collection	City Solid Waste Department	Private Trash Services
Building Code Compliance	City Building Department	Third-Party Building Inspectors
Securities Anti-Fraud Regulation	Federal Securities and Exchange Commission	State Blue Sky Laws
Health Care	Affordable Care Act	Block Grant and Waivers, Health Care Compact
Dispute Resolution	Litigation	Binding Arbitration
Lawmaking	Centralized State	Charter City
Commercial Banking	Federal Reserve Bank	State Chartered Banks
Automobile Infrastructure	State and Federal Highways	Private Toll Roads
Criminal Justice	Trial by Jury	Victim-Offender Mediation
Housing Development	High-Density Urban Rental	Master Planned Community
Package Delivery	US Postal Service	Federal Express

Source: Author.

Greenfield and Subsidiarity Strategies Already Work

Interestingly, we do not have to guess how this strategy might work. It turns out that many if not most successful large-scale policy innovations in the past few decades followed exactly this pattern.

Perhaps the most well-known such innovation is public charter schools. The original idea behind charters was to establish a separate system in which schools would receive public funds and deliver a free public education but have the freedom to try different operating models. In states such as Texas, charter schools are established and operate under a largely separate section of the state education code. But perhaps without exception, charter schools are subject to less regulation than district schools.

Charter schools are a greenfield strategy, as they meet all four tests:

1. Charter laws do not change the authority structure. Education is still controlled by the state government and subject to federal regulations.
2. Charter laws were created through legislative action, not through the courts or by popular vote.
3. Charters compete with the existing school district system.
4. Parents can choose to send their children to a charter school or remain in the district system—same for teachers and principals.

The competitive regulatory structure charter schools created—particularly as they increase their market share in local areas—is a powerful force for change. Districts that must compete with charter schools are forced to innovate, improve their human resources practices to retain talent, and be more responsive to parents who now have the ability to leave and take their public funding with them. Studies have shown that districts, while often slow to adapt because

of their large size and bureaucratic inertia, do respond to competition from charters.¹⁹

Education might be the most salient example, but in fact many different greenfield strategies have been proposed or used to great effect at every level of government. Table 1 provides a few examples, but others could be added to this list.

Avoiding System Failure

Our society, characterized by massive-scale and mind-boggling complexity, has a classic legacy system problem. Fixing the system seems impossible, and all involved are filled with a sense of dread. Experienced software developers will recognize the challenges facing our governance system and understand policymakers' plight. But as Winston Churchill once said, "Out of intense complexities, intense simplicities emerge."²⁰

The answer is *not* despair. Rather, we should take many of the hard-learned lessons of software engineering and adapt them to the governance challenge. The knowledge we have of software code needs to be applied to the legal code.

But we first need to accept that the current centralized system must undergo a major refactoring. Too much power is concentrated at the center, and this has led to a dysfunctional mess. You can see this concentration by following the money. In 2015, the federal government made up about 61 percent (\$3.4 trillion) of total revenue across federal, state, and local government receipts. Even after accounting for transfers, the federal government still maintains approximately 51 percent of all spending.²¹

One of the major reasons why our politics is so divisive is that we have established a winner-take-all game decided in Washington, DC. There will always be partisanship and vitriol in politics; it is like the pollution that comes from the decision-making process. As more decisions are sucked inside the Beltway, pollution is too.

The solution to pollution is dilution. This dilution can be achieved by applying the principle of

subsidiarity: putting the decisions about the commons as close to the people as possible. Some might even call this “draining the swamp.”

Applying subsidiarity will require a major refactoring, which is where greenfield strategies can come into play. No other peaceful strategy for devolving power has ever succeeded because brownfield regulatory regimes are monopolies, and no monopoly in the history of mankind has reformed itself from within.

That is why we need to switch our policy change paradigm. We need a new political language, a way to

think and talk about policies in a way that acknowledges the real-world challenges we face. And we need to simplify and de-escalate our politics by returning more control to local communities.

In short, we need to refactor our governance for subsidiarity. And we need to do it before the massive, tangled, bug-ridden spaghetti code in our centralized legacy system crashes and cannot be restarted.

After all, you can always replace a broken computer. It is much harder to replace a broken society.

Notes

1. Mickie Krause, *Information Security Management Handbook* (Boca Raton, FL: CRC Press, 2006).
2. Gallup News, “Confidence in Institutions,” Gallup, 2017, <http://news.gallup.com/poll/1597/confidence-institutions.aspx>.
3. *Oxford English Dictionary*, s.v. “subsidiarity,” <http://www.oed.com/>.
4. Max Weber, *Economy and Society*, Vol. 2 (Berkeley, CA: University of California Press, 1922), chaps. 14–15.
5. David Lilienthal, *Democracy on the March* (New York: Harper & Brother Publishers, 1944).
6. Kahneman, *Thinking, Fast and Slow* (New York: Farrar, Straus and Giroux, 2011).
7. The phrase is often attributed to Peter Drucker.
8. Dunbar, *Grooming, Gossip, and the Evolution of Language* (Cambridge, MA: Harvard University Press, 1996).
9. R. I. M. Dunbar et al., “The Structure of Online Social Networks Mirrors Those in the Offline World,” *Social Networks* 43 (October 2015): 39–47, www.sciencedirect.com/science/article/pii/S0378873315000313.
10. Frederik Lundh, *Python Standard Library* (Sebastopol, CA: O’Reilly Media, May 2001), <http://shop.oreilly.com/product/9780596000967.do>.
11. US Census Bureau, “Pop Culture: 1790,” 2017, www.census.gov/history/www/through_the_decades/fast_facts/1790_fast_facts.html.
12. Charles C. Little and James Brown, *Public Statutes at Large of the United States of America*, ed. Richard Peters (Cambridge, MA: Metcalf and Company, 1845), www.loc.gov/law/help/statutes-at-large/1st-congress/c1.pdf.
13. *Federal Register*, “Federal Register Pages Published, 1936–2015,” May 2016, www.federalregister.gov/uploads/2016/05/stats2015Fedreg.pdf.
14. Linus Benedict Torvalds, “Free Minix-Like Kernel Sources for 386-AT,” Google Groups, October 5, 1991, <https://groups.google.com/forum/#!msg/comp.os.minix/4995SivOl9o/GwqLJlPSICEJ>.
15. Dan Marinescu, *Cloud Computing: Theory and Practice* (Oxford: Newnes, 2013); and Michael Larabel, “Linux Kernel at 19.5 Million Lines of Code, Continues Rising,” Phoronix, June 2015, www.phoronix.com/scan.php?page=news_item&px=Linux-19.5M-Stats.
16. Legal Information Institute, Cornell University Law School, “First Amendment,” www.law.cornell.edu/constitution/first_amendment.
17. Martin Fowler, *Refactoring* (Boston, MA: Addison-Wesley Professional, 1999).
18. Jesús Bisbal et al., “Legacy Information Systems: Issues and Directions,” *IEEE Software* 16, no. 5 (September/October 1999): 103–11, <https://pdfs.semanticscholar.org/446b/85482609887bbb7e28d4ecfb0294914dbc15.pdf>.
19. Marc Holley et al., “Competition with Charters Motivates Districts,” *Education Next* 13, no. 4 (2013), <http://educationnext.org/competition-with-charters-motivates-districts/>.
20. Winston Churchill, *The World Crisis, 1911–1918* (London: Penguin Classics, May 2007), 263.
21. Tax Policy Center, “Federal, State, and Local Government Receipts and Transfers, 2015,” 2016, www.taxpolicycenter.org/briefing-book/what-breakdown-tax-revenues-among-federal-state-and-local-governments.