# Increasing the Level of Abstraction as a Strategy for Accelerating the Adoption of Complex Technologies

Willy C. Shih
Harvard Business School

**Abstract.** Many new technologies are complex and embody high levels of technical sophistication, and applying them should require significant knowledge and experience. Yet, the rapid adoption and incorporation of these technologies into other innovations seems inconsistent with the expertise needed to make them work. In this paper, we propose increasing levels of abstraction as a strategy for speeding the adoption of new technologies. Higher-level abstractions package complexity in ways that makes them easier to understand and recombine, and they decrease the resources needed by firms to deploy sophisticated technical know-how. Increasing the level of abstraction is a way to push forward the innovative frontier by making such difficult-to-use technologies readily accessible to other innovators. Although this framing has been used in engineering and software development to describe modular encapsulation and cumulative innovation, we propose its use in the management literature to describe more broadly the uptake of new technologies and their facile recombination. This framing casts a different light on cumulative innovation and exposes new managerial questions to explore.

Technological change lies at the heart of economic growth (Romer 1990), and we live in an era in which many new technologies are incorporated into products and services at an extraordinary pace. New technologies build on a foundation of knowledge and inventions laid out by earlier innovators (Nelson and Winter 1982, Scotchmer 1991, Caballero and Jaffe 1993), and many innovations today are remarkably complex and embody high levels of sophistication. For example, 4G-LTE or 5G wireless telecommunications protocols employ fl exible and spectrally efficient radio-link technologies (Larmo et al. 2009, Agyapong et al. 2014, Gupta and Jha 2015) that require highly trained radio engineers many years to develop. Yet, once they are embodied in semiconductor chip sets, smartphone manufacturers can incorporate them within a single

design cycle, usually lasting less than a year. Innovations like smartphones that rely on these technologies, in turn, have become platforms upon which a multitude of other new products and services are built, and their speed of adoption is much faster than would be expected if firms had to develop in-depth understanding and working knowledge of the underlying computing and communications technologies before being able to replicate them. Innovators, thus, are able to shortcut the requisite organizational learning and capability building that faced the pioneers, and this fosters broad and faster diffusion of new technologies. This paper seeks to illuminate a mechanism for this knowledge transfer and bridge a gap in the management and innovation literature by arguing that increasing the level of abstraction can be framed as a deliberate strategy that innovators use to enable the rapid adoption of new technologies and facilitate recombinant innovation.

The innovation literature is rich with the analysis of learning and knowledge transfer. Central to the adoption and use of new technologies is the development of an understanding of the technical foundations coupled with the ability to internalize and use the knowledge embedded within them (Cohen and Levinthal 1990). Knowledge transfer is a central competitive dimension of what firms do (Nelson and Winter 1982, Kogut and Zander 1992, Zander and Kogut 1995, Schumpeter 2013) and contributes substantially to organizational performance (Epple et al. 1996, Argote and Ingram 2000). Such transfers represent efforts to create partial or exact replicas of complex practices (Lippman and Rumelt 1982) and often entail understanding a web of relationships that connect specifi c resources (Szulanski 2000). Difficulties experienced in such transfers connote stickiness (von Hippel 1994, Szulanski 1996), which drives the often-heavy resource costs associated with such transfers (Teece 1981).

The codification of such knowledge is a prerequisite to its effective use. This involves the transformation of experiences and information into some kind of symbolic and easily communicable artifacts, such as blueprints, images, formulas, or computer instructions (Teece 1981). To the extent that the knowledge can be codified, which is the process of converting that knowledge into messages that can be processed as information, less of it remains idiosyncratic to a person or a few people, and it is transformed into something that can be communicated at low costs (Cowan et al. 2000). The degree of codifi cation and how easily capabilities are taught has a significant influence on the speed of transfer (Teece 1981, Zander and Kogut 1995). Yet, as legions

of innovators employ many new complex technologies in their products, they seem to have bypassed the traditional need to understand the codification. This is where abstraction comes in.

## Abstraction

Abstraction is a concept that has been extensively used in fields outside of management research. Its use in philosophy began with Locke in 1706, who described it as a process of separating ideas from the spatial or temporal qualities of particular things (Colburn and Shute 2007). In mathematics, abstraction is used as a verb and signifies the extraction of the underlying patterns and structures of a concept, while removing dependence on physical objects to which they might have been connected, leading to a generalization. In the cognitive sciences, it has been more associated with the separation of generalities from specific facts (Ericsson et al. 1993, Ohlsson and Lehtunen 1997).

In examining the adoption of new technologies, the usage of the term in engineering is most helpful. In software development and engineering design, producing an abstraction means identifying a pattern, naming and defining it, analyzing it, finding ways to specify it, and providing a way to reuse it (Shaw 1989). It is a process of generalization, deciding what details need to be highlighted and which details can be ignored in order to retain and make visible only the key relevant information for performing a particular task (Wing 2008). Abstraction in software development utilizes the concept of layers: the layer of interest and the layer below. Well-defined interfaces between layers then enable the building of more complex systems (Wing 2008).

In the technology-innovation literature, abstraction has been broadened to encompass the division of innovative labor, wherein the process enables the representation of phenomena using a limited number of " essential" elements, rather than in terms of concrete features (Arora and Gambardella 1994), providing a mechanism to generalize a thought process for wider application. If we think of a technological innovation as a black box, an abstraction characterizes the transfer function or behavior of the system contained within, along with all the inputs and outputs that would be needed in order to completely describe and use its functionality. In this context, the abstraction would not necessarily render explicit the inner workings of the black box. In contrast, codification would be concerned with how it worked, the internal mechanisms or mechanics, the core technological underpinnings, and how it delivered the transfer function.

Abstraction can be a useful framing in understanding innovation and the adoption of complex technologies. Providers of advanced technologies package them often as a demarcated bundle with complete specifications and behavioral models that make it easy to incorporate as a building block for a more complex system. By economizing on the information that the mind has to respond to in order to use the technology (Teece 1981), it enhances the trialability of a new technology (Rogers 1995) and lowers the barriers and reduces the costs of adoption, increasing the speed of diffusion (Hall 2004).

The concept of modularity is central to abstraction. Modularizing a system creates a partitioning of functionality across component modules (Baldwin and Clark 2000). Specifying a system' s architecture, how functionality is divided across modules, and the interfaces that govern how the component parts interact (Baldwin and Clark 2006) creates building blocks that each embody some subset of overall system behavior. Several authors have pointed out that when complexity reaches a certain threshold, it can be isolated by defining a module as a separate abstraction with a simple interface (Baldwin and Clark 2000, Ethiraj and Levinthal 2004). Modularity can promote specialization and more facile innovation, and, in this way, some modules can be viewed as abstractions. Yet not every modular partitioning leads to components that are abstractions in the general sense. If the partitioning is driven by an effort to decouple system components and distribute development efforts, individual modules may not have standalone value for recombination. Abstractions are intended to stand alone and are intended to be used in unpredictable future recombination.

## Raising the Level of Abstraction

Our central proposition is that raising the level of abstraction can be an effective strategy for facilitating and increasing the pace of adoption of a complex technology. This encompasses increasing the scope of the innovation, often by incorporating more of the dependencies or integrating complements so that a user is relieved of the task of orchestrating all of the necessary pieces. In so doing, it increases product development productivity by reducing the need for deep understanding of implementation details in exchange for limiting the ultimate degrees of design freedom.

This is a concept that is well known from the earliest days of the development of computer software. Digital computers carry out sequences of arithmetic and logical operations that are

controlled by binary data bits (ones and zeros). Machine-language programs use these binary data bits to tell a computer precisely what to do at each step, but the instructions are extremely primitive and only do things like move a number between storage locations or perform simple arithmetic or logical operations. If programmers had to worry about this level of detail, they would find it difficult to rise above the minutia and think about the big picture (Návrat 1994, Návrat and Filkorn 2005). Computer scientists eased their task by developing symbolic " assembly languages." Assembly language programs were the earliest " higher-level language" and were symbolic representations that were translated into the ones and zeros that the computer actually uses by a program called an assembler. They introduced a layer that was the next higher level of abstraction. They were followed by increasingly higher-level languages, such as FORTRAN, which stood for formula translator, developed by John Backus and a small team at IBM from 1954 to 1957 (Backus 1979). This was a revolutionary innovation because it enabled programmers to express what they wanted to do in a more easily understood form, like "c = a + b ." The programmer ran a " compiler" to translate the high-level language program into the primitive machine instructions.

More sophisticated languages, such as C++ and Java, operate at progressively higher levels of abstraction, helping designers to segregate the concepts that a programmer wants to implement from its instances of implementation. These levels are layers that facilitate the introduction of new unforeseen hardware below the language layer, or new unforeseen applications above the language layer. Abstraction suppresses the details and simplifies the understanding of the result (Shaw 1989). Over time, abstraction has been associated with language constructs; specification techniques; program structures, such as algorithms and data types; and strategies for modular decomposition. Software developers have been able to focus on high-level system architectures and less and less on the details of implementation.

In computer software, abstraction today is looked upon as the isolation of a software subsystem or module of a larger system into a reusable component. Development of abstraction techniques has been a major source of improvement in programming practices (Shaw 1989). Some even argue that entire history of software engineering can be characterized as progressively rising levels of abstraction (Aaronson 2006). This has been evident in the development of new computer languages, platforms, methods, and tools. Today, reusable components and software-container systems play a critical role in software innovation.

The same concept has also been applied to hardware, but in a more limited way. Early computer systems required programmers to understand how each piece of attached hardware, such as a printer, worked. This became a significant burden as the variety of devices proliferated. Software developers solved this problem by adding a layer of abstraction, which hid or encapsulated knowledge of the device within a module. For example, Microsoft introduced a hardware abstraction layer into its Windows operating system to insulate software writers from the wide range of hardware that was developed by third parties. Programmers then wrote standardized instructions, such as for printing a page, and sent them to the operating system. The operating system, in turn, communicated with the device, using a "device driver." Device-specific knowledge was not communicated to other parts of the system, which also enabled independent development of modules and better comprehensibility (Parnas 1972). The physical printer plus its associated driver abstracted the printing function from the point of view of the operating system software. More importantly, the users of the abstraction were relieved from needing to know the underlying mechanism of how the device functioned.

These methods have spread to hardware at the microchip level as well, with hardware compilers that can implement component descriptions into physical designs of circuit components, along with associated simulation and modeling tools that enable designers to work at a high level (see, for example, Van Rompaey et al. 1996 and Baghdadi et al. 2001).

Raising the level of abstraction can describe the strategy behind facilitating the adoption of complex technologies, especially those that rely on cumulative innovation (Hargadon and Sutton 1997) and recombination. Innovation requires a broad search for information and recombination of different kinds of knowledge (Nelson and Winter 1982, Levinthal and March 1993). When new technologies emerge, they often face challenges in adoption (Anderson and Tushman 1990), especially by incumbent firms (Nelson and Winter 1982, Levinthal and March 1993, Tushman and O' Reilly 1996), as firms have to first recognize their value and then assimilate them (Cohen and Levinthal 1990). The recombination of distant or diverse knowledge is essential for breakthroughs because research confined to narrow domains might lead to intellectual lock-in and foster incremental innovation (Gavetti and Levinthal 2000, Fleming 2001, Ethiraj and Levinthal 2004, Kaplan and Vakili 2015). Thus, bridging distant or diverse knowledge is important to enhancing creativity (Hargadon and Sutton 1997, Audia and Goncalo 2007, Kaplan and Vakili 2015), and the use of general and abstract knowledge in innovation fosters the division of innovative labor

(Arora and Gambardella 1994). Explicitly seeking to raise the level of abstraction should facilitate easier incorporation and reduce barriers to adoption.

Some examples are instructive. Many electronic device manufacturers raise the level of abstraction on a complex new technology in order to speed its incorporation into their customers' designs. They do this by creating and publishing " reference designs." Reference designs offer complete example designs, which include wiring schematics, electronic computer-aided design and computer-aided manufacturing files intended to be used as direct input to manufacturing systems, and software that could be customized with a customer's logo and branding. Implementation details are hidden, even though they are critical to the underlying device' s proper functioning. The only things revealed to the adopter are interface details and the abstraction's functional behavior, a black box whose inputs, outputs, and behaviors are fully characterized. Compared with doing their own *de novo* design, a reference design spares users from having to understand implementation methods and details and makes the adoption of a complex new technology much easier.

A historically important example was how Intel Corporation sped the adoption of next-generation chip technology in personal computers. In 1995, the company was experiencing difficulty getting its PC customers to keep pace with its new microprocessor releases. It wanted companies like Dell and Compaq to offer models that incorporated its latest chips as soon as they became available, but found they were lagging because of the delays in absorbing new design information. It solved the problem by providing its "ATX" reference designs, which were matched to its latest CPU chips as soon as they became available. All of the complex electrical signaling and timing were hidden— PC manufacturers had no need to understand the implementation details. The reference designs abstracted all of the electrical signaling needed to assemble and mass-produce the heart of a PC, making the embodied microprocessor innovations simpler to recombine. The reference design included the locations of mounting holes and specification for every electrical and signal connection. Designers of products like industrial controllers for machine tools, cash registers, and other devices that needed inexpensive computation and user interaction could recombine their own innovations with the ATX abstraction of a basic computing engine that included provisions for a simple user interface and connectivity. The ATX innovation enabled a huge wave of new market entrants with no prior experience in the technology, including companies like Lenovo.

Innovators also use abstractions as platforms that promote recombination. A smartphone is an example of a platform that abstracts Internet connectivity, computing power, and a touchscreen-display user interface in a battery-operated package. The abstraction includes application programming interfaces (APIs) used in an associated software-development kit, for example, to provide touchscreen interface actions or camera image capture. The staggeringly complex details of implementation are hidden. Although designing it as a building block for others to use might not have been the original objective, it facilitates more efficient cumulative innovation and recombination. New personal medical test devices use smartphones as Internet edge devices to log and send data. They are used as subsystems in infrared cameras, measurement tools, and other applications that need generic computing and communications functionality.

Abstractions can extend beyond hardware and software to the provision of services. Cloud computing services such as Amazon Web Services (AWS) raise the level of abstraction for generic connected computing and communications, enabling innovators to purchase networked computing capability and capacity on demand without having to be concerned with the details of the hardware or software provisioning. Traditionally, the provisioning is performed by a large IT infrastructure organization, but AWS abstracts all of those complements and makes them invisible, making adoption easier.

Progressively higher levels of abstraction are often visible in tiered product innovations. Google Maps is a consumer-oriented geographic information system. After acquiring the original software, Google converted it into a web application (app) with a JavaScript API that allowed embedding in third-party websites. Google Maps abstracted real-time geolocation information, including complex technologies like traffic-sensing, and enabled recombination with services like the ride-sharing offerings of Uber, Lyft, and others. In the next higher layer, the Uber app delivered an innovative new service to consumers, but also raised the level of abstraction by offering a ride-request button that others could incorporate into their own new services, for example, to provide food delivery. Uber marketed this as a way to build on-demand delivery solutions for local retailers and online shops. The "Ride there with Uber" button brought everything in the Uber application— fare estimates, a pick-up time estimate, and location— into the new service with just a few lines of code. Built on top of Google Maps, which, in turn, was built on top of smartphone platforms, Uber abstracted a modular ride service for firms like

OpenTable, Starbucks, and United Airlines to recombine delivery services within their own service apps at a next higher tier.

Innovators can use a strategy of raising the level of abstraction to enhance network effects on their platforms by bringing more parties onboard. Voice assistants, such as the Google Assistant or Amazon Alexa, speed the incorporation of voice recognition into new services. The Google Assistant provides a voice user design interface and framework for conversational design, and Amazon' s Alexa offers access to specific functional groups, like a video-skill API or music-skill API. Both of these abstractions package voice-recognition functionality in a way that allows others to easily build this functionality into their products without any of the formidable research and development capabilities needed to commercialize the core speech-recognition technology. This encourages more parties on all sides to join the platform.

### Higher-Level Abstractions Replace Lower-Level Abstractions

Higher-level abstractions that simplify or further reduce the need for technical expertise often replace lower-level ones in the market and can shift the dominant design paradigm. An example is the adoption of software container systems. An important innovation in the software operating systems for computers was the virtual machine, a programming environment that abstracted server hardware and made it appear as if it was a dedicated machine from the point of view of an application developer. Everything that was necessary to run the application was contained within the virtual machine, and multiple instances of the virtual machine could run on a single hardware computer system. This is now a mature technology and is available from a number of vendors.

Software containers like Docker virtualize the operating system running on a computer and make it appear like the operating system and all dependencies required to run a program are within a container image. This is a higher level of abstraction than the virtual machine, and it has that advantage that only one copy of the operating system is needed, improving program-execution efficiency, as well as speed of deployment. Software containers have their roots in primitive constructs like the chroot system call, which was introduced in 1979, and FreeBSD Jails, a way of partitioning the FreeBSD operating system into independent systems introduced in 2000 (Hope 2002). This was followed by process containers introduced by Google in 2006 (Menage 2007), but even those required a high level of skill to implement. Docker emphasized ease of use

and delivered a higher level and more complete abstraction that offered an ecosystem for container management. Following its introduction in 2013, its popularity took off (Merkel 2014, Rubens 2017). As a higher-level operating system abstraction, it has started to substantially replaced virtual machines in new deployments. A 2018 survey of 576 IT industry leaders reported that 11.98% of enterprises had deployed them and 47.05% planned to (Diamanti 2018). Google Trends, which reports relative search interest, shows a clear displacement of " virtual machines" by " Docker" over a 15-year period as software implementers switched to the higher-level abstraction.

## Case Study: Reducing Resources Required to Apply New Technologies

We examined the Taiwanese company MediaTek, who raised the level of abstraction for 2.5G mobile handset makers with a design package that included everything needed to assemble the hardware and software (Shih et al. 2010). On the hardware side, it provided a reference design in electronic form so that customers could easily modify them. Customers could also choose to use them directly without modification. In that case, all they had to do was add a plastic case, and they would be able to offer a complete, albeit basic, product.

On the software side, the company included a proprietary operating system, a man– machine interface code base, and key design tools. MediaTek's customers could use these to customize the software that ran on the handset and quickly produce a custom look and feel with tailored functionality, often over the course of a lunch break, and then could download it into the handset. Handsets also incorporated other complementary components that the company did not supply. The company did extensive testing and qualification of components and published the data for its customers. For example, MediaTek did not make the camera modules for phones, but, rather, it tested modules that were available on the market and optimized drivers and software for them. Color rendition and image clarity were very dependent on the software used with them, so MediaTek invested extensively in R&D and testing to ensure that its designs could produce good results for its customers who were far less skilled. This resulted in a design robustness that could survive the customers who chose to use inexpensive lenses instead of better quality ones, or customers who wanted to take design or engineering shortcuts.

Although traditional contemporary feature phone firms, such as Nokia and Motorola, employed hundreds of engineers to develop and produce a product over the course of a nine-

month product cycle, a typical MediaTek customer design team might have as few as 10 employees and could produce a new feature phone in several weeks. The company's abstraction efforts led to the emergence of nearly a thousand small phone makers in the Shenzhen, China area.

MediaTek was particularly interesting because its company strategy was based on raising the level of abstraction in different consumer electronic device markets. Prior to its work with mobile-phone handsets, it had done the same thing by reducing the electronics for making a CD-ROM to a single silicon chip, followed by doing the same thing for DVDROMs, DVD/Blu-Ray players, and television sets. In each of these product categories, the company enabled a large number of new entrants.

## Strategic Implications

The innovation model described here provides one explanation for the rapid uptake of sophisticated new technologies. An innovator can raise the level of abstraction of a technology to make it easier to incorporate and economize on the know-how, both tacit and explicit, that is required to use it. This frame also raises some implications as well as questions for further research.

For firms, raising the level of abstraction is away to lower the barriers to adoption of a new technology, as exemplified by the use of reference designs. The MediaTek example highlighted both benefits and consequences of the approach. By making its technology more accessible with the objective of increasing adoption, it commoditized products that incorporated it. Interestingly, Intel produced a similar outcome with its ATX reference designs, and, in both cases, the promulgators retained a key value-extraction mechanism in the form of the semiconductor chip at the center of their respective platforms, and both derived considerable benefits. Thanks to network effects associated with usage of the Intel platform, ATX became the dominant design for personal computers. MediaTek, on the other hand, faced competition from other chip firms, including Qualcomm and Samsung, so although it was able to sell over 350 million chipsets in a year, the main effect was to commoditize lower tier feature phones on which producers made very little profit.

For firms who operate matching platforms, abstracting a service, such as the " Ride there with Uber" example, can simplify adoption and bring more users to the service provided on one of

the platform sides. Again, the focus is on how to encourage other innovators to build on the technology or service by making it easy to incorporate.

Innovators operating at the higher levels of abstraction don't have detailed understanding of inner workings at lower levels in the cumulative innovation pyramid. Thus, although raising levels of abstraction lies at the heart of economizing on bounded rationality, innovation could become more limited beneath the abstraction layer. We asked a senior manager at Microsoft how many of her employees actually knew how to write a compiler, something that was once a rite of passage for computer science undergraduates. Her answer was that none of her new hires did, and if anyone needed to know how, they had to train that individual. When there is no longer the detailed understanding of the base technologies and how to implement them de novo, there is little incentive to go back and revisit the technology that the abstraction is built upon. This can become an issue when the balance point in core system design tradeoffs changes because of a new application type that might disproportionally favor use of one resource over another. This is quite evident today in the evolution of domain-specific microarchitectures for new application areas like machine learning. One author commented:

> It adds a dimension that most of the industry is not used to. We are not really being taught that kind of thing in school, and it's not something that the industry has decades of experience with. So it is not ingrained in the programmers. (Bailey 2020)

Said differently, a higher-level abstraction reinforces dominant designs to the exclusion of new approaches (Anderson and Tushman 1990). The commercial success of the ATX motherboard abstraction promulgated by Intel stifled innovation in the PC system architecture for many years. ATX motherboards achieved de facto standardization, and the very high volumes and low market pricing made it difficult for competitors to reap any profit. This channeled innovations primarily into cost reductions and cosmetics, with little architectural innovation. Intel competitor AMD achieved some success by replicating the external interfaces of ATX in its own version of the reference design and then changed the memory controller architecture inside to elicit better performance. But even then, it had trouble delivering consistent market success. Computer architects have long pointed out that memory-system performance was crucial to better overall system performance, but the market weight of ATX stifled innovation within the confines of the abstraction. We discussed this with leaders of the Taiwanese computer industry

and memory chip manufacturers who recognized the opportunity for relieving a system-performance bottleneck, but they argued that the economics were overwhelmingly against making significant changes. It was only the emergence of hyperscale datacenters and their high consumption of computer systems that provided the economic motivation for firms like Google, Amazon, and Facebook to revisit the design assumptions and make fundamental architectural changes that were more suited to their own specific usage cases (Thusoo et al. 2010, Hazelwood et al. 2018).

In the same vein, the Intel ATX example also raise questions about technological learning more broadly. Researchers who have studied learning by example point out that it is a well-documented process for cognitive-skill acquisition. People gain a deep understanding when they can follow a worked-out example (Renkl 2005); a reference design is essentially this. For novices who don't have deep prior exposure to the material, an absorption process that relies on studying worked examples is more effective for learning, as well as more efficient with less investment of time and effort during acquisition (Atkinson, et al. 2000), and is a more efficient approach than trial and error (Sweller and Cooper 1985, Cooper and Sweller 1987). But does reliance on abstractions confine learning to narrower domains? Their more widespread use might explain the increasing complexity and deeper tiering found in modern supply chains. Our study of the Chinese motorcycle industry found that firms who used reference designs had great difficulty innovating beyond superficial attributes like paint schemes, shapes of the handlebars, or shape of the gas tank (Shih and Dai 2010). Firms who offer successful abstractions reduce incentives to invest in their layer of a value network, which can provide an ancillary benefit of protecting tacit or proprietary knowledge. Their customers tend to maintain focused expertise above the abstraction layer with a correspondingly greater reliance on suppliers below them.

We think that raising the level of abstraction by a firm as either an implicit or explicit strategy provides a useful lens that helps explain the rapid adoption of some new advanced technologies as well as their speed of diffusion. It complements the established body of literature on modularity by adding a temporal element and the notion of higher levels, at which an innovator interfaces. It also raises questions about the stratification of innovative activities, as the pressure from lowered entry barriers relieves innovators from the need to develop an understanding of the technical foundations that might limit more basic innovation down the road.

## References


Aaronson L (2006) Q&A with Grady Booch, *IEEE Spectrum*, https://spectrum.ieee.org/geek-life/profiles/qa-with-grady-booch.

Agyapong PK, Iwamura M, Staehle D, Kiess W, Benjebbour A (2014) Design considerations for a 5G network architecture. *IEEE Communications Magazine*, 52(11), 65-75.

Anderson P, Tushman, M (1990) Technological discontinuities and dominant designs: A cyclical model of technological change. *Administrative Science Quarterly*, 604-633.

Argote L, Ingram P (2000) "Knowledge transfer: A basis for competitive advantage in firms," *Organizational Behavior and Human Decision Processes*, 82(1), 150-169.

Arora A, Gambardella A (1994) "The changing technology of technological change: general and abstract knowledge and the division of innovative labour." *Research Policy*, 23(5), 523-532.

Atkinson R, Derry S, Renkl A, Wortham D (2000) Learning from examples: Instructional principles from the worked examples research. *Review of educational Research*, 70(2), 181-214.

Audia, P, Goncalo J (2007) Past success and creativity over time: A study of inventors in the hard disk drive industry. *Management Science*, 53(1), 1-15.

Backus, J (1979) The history of Fortran i, ii and iii. Ann. Hist. Comput. 1(1):21–37.

Baghdadi A, Lyonnard D, Zergainoh N, Jerraya A (2001) An efficient architecture model for systematic design of application-specific multiprocessor SoC. *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001* (IEEE, New York), 55-62

Bailey, B (2020) An Increasingly Complicated Relationship with Memory, *Semiconductor Engineering*, https://semiengineering.com/an-increasingly-complicated-relationship-with-memory/#comment-326042

Baldwin C, Clark K (2000) *Design rules: The power of modularity* (Vol. 1). MIT Press.

Baldwin C, Clark K (2006) Modularity in the design of complex engineering systems. In Complex engineered systems. 175-205. Springer, Berlin, Heidelberg.

Caballero R, Jaffe A (1993) How high are the giants' shoulders: An empirical assessment of knowledge spillovers and creative destruction in a model of economic growth. *NBER Macroeconomics Annual*, 8, 15-74.

Colburn, Timothy, and Gary Shute (2007) Abstraction in computer science. *Minds and Machines* 17(2), 169-184.

Cohen W, Levinthal D (1990) Absorptive capacity: A new perspective on learning and innovation. *Administrative Science Quarterly*, 128-152.

Cooper G, Sweller J (1987) Effects of schema acquisition and rule automation on mathematical problem-solving transfer. *Journal of Educational Psychology*, 79(4), 347.

Cowan, R, David P, Foray D (2000) The explicit economics of knowledge codification and tacitness. *Industrial and Corporate Change*, 9(2), 211-253.

Diamanti (2018) Container Adoption Benchmark Survey, Diamanti.com, https://diamanti.com/wp-content/uploads/2018/07/WP_Diamanti_End-User_Survey_072818.pdf

Epple D, Argote L, Murphy K (1996) An empirical investigation of the microstructure of knowledge acquisition and transfer through learning by doing. *Operations Research*, 44(1), 77-86.

Ericsson, K. Anders, Ralf T. Krampe, and Clemens Tesch-Römer (1993) "The role of deliberate practice in the acquisition of expert performance." *Psychological review* 100(3), 363.

Ethiraj S, Levinthal D (2004) Modularity and innovation in complex systems. *Management Science* 50(2), 159-173.

Fleming L (2001) Recombinant uncertainty in technological search. *Management Science*, *47*(1), 117-132.

Gavetti G, Levinthal D (2000) Looking forward and looking backward: Cognitive and experiential search. *Administrative Science Quarterly*, 45(1), 113-137.

Gupta A, Jha R (2015) A survey of 5G network: Architecture and emerging technologies. *IEEE Access*, 3, 1206-1232.

Hall B, (2004) Innovation and diffusion. No. w10212. National Bureau of Economic Research

Hargadon A, Sutton R (1997) Technology brokering and innovation in a product development firm. *Administrative Science Quarterly*, 716-749.

Hazelwood K, Bird S, Brooks D, Chintala S, Diril U, Dzhulgakov D, Law J (2018, February) Applied machine learning at Facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA*), IEEE, 620-629.

Hope P (2002) Using Jails in FreeBSD for fun and profit. *login: The Magazine of USENIX & SAGE*, 27(3).

Kaplan, S, Vakili K (2015) The double-edged sword of recombination in breakthrough innovation. *Strategic Management Journal,* 36(10), 1435-1457.

Kogut B, Zander U (1992) Knowledge of the firm, combinative capabilities, and the replication of technology. *Organization science*, 3(3), 383-397.

Larmo A., Lindström M, Meyer M, Pelletier G, Torsner J, Wiemann H. (2009) The LTE link-layer design. *IEEE Communications magazine*, *47*(4), 52-59.

Levinthal D, March J (1993). The myopia of learning. *Strategic Management Journal*, 14(S2), 95-112.

Lippman S, Rumelt R (1982) Uncertain imitability: An analysis of interfirm differences in efficiency under competition. *The Bell Journal of Economics*, 418-438.

Menage P (2007) Adding generic process containers to the linux kernel. In *Proceedings of the Linux symposium*, 2, 45-57.

Merkel D (2014) Docker: lightweight Linux Containers for Consistent Development and Deployment. *Linux Journal*, 239, 2.

Návrat P (1994) Hierarchies of programming concepts: abstraction, generality, and beyond. *ACM SIGCSE Bulletin*, 26(3), 17-21.

Návrat P, Filkorn R (2005) A Note on the Role of Abstraction and Generality in Software Development. *Journal of Computer Science*, 1(1), 98-102.

Nelson R, Winter S (1982) *An evolutionary theory of economic change*, 929-964.

Ohlsson Stellan, Lehtinen E (1997). Abstraction and the acquisition of complex ideas. *International Journal of Educational Research,* 27(1), 37-48.

Parnas D (1972) On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, *15*(12), 1053-1058.

Renkl A. (2005) The worked-out-example principle in multimedia learning. *The Cambridge handbook of multimedia learning*, 229-245.

Rogers E (1995) *Diffusion of Innovations*, Simon and Schuster.

Romer P (1990) Endogenous technological change. *Journal of Political Economy*, *98*(5, Part 2), S71-S102.

Rubens P (2017) What are containers and why do you need them?. *CIO*. URL: https://www. cio. com/article/2924995/software/what-are-containers-and-why-do-you-needthem. html.

Schumpeter JA (2013) The process of creative destruction. Capitalism, Socialism and Democracy (Routledge Taylor & Francis e-Books, Milton Park, Abingdon, UK), 81–85.

Scotchmer S (1991) Standing on the shoulders of giants: cumulative research and the patent law. *Journal of Economic Perspectives*, 5(1), 29-41.

Shaw M (1989) Larger scale systems require higher-level abstractions. *ACM Sigsoft Software Engineering Notes*, 14(3), 143-146.

Shih W, Chien C, Wang J (2010), "Shanzhai! MediaTek and the White Box Handset Market," Harvard Business School Case no. 610-081, Harvard Business Publishing, Boston, MA.

Shih W, Dai N (2010), "From Imitation to Innovation: Zongshen Industrial Group," Harvard Business School Case no. 610-057, Harvard Business Publishing, Boston, MA.

Sweller J, Cooper G (1985) The use of worked examples as a substitute for problem solving in learning algebra. *Cognition and Instruction*, 2(1), 59-89.

Szulanski G (1996) Exploring internal stickiness: Impediments to the transfer of best practice within the firm. *Strategic Management Journal*, 17(S2), 27-43.

Szulanski G (2000) The process of knowledge transfer: A diachronic analysis of stickiness. *Organizational behavior and human decision processes*, 82(1), 9-27.

Teece D (1981) The market for know-how and the efficient international transfer of technology. *The Annals of the American Academy of Political and Social Science*, 458(1), 81-96.

Thusoo A, Shao Z, Anthony S, Borthakur D, Jain N, Sen Sarma J, Liu H (2010) Data warehousing and analytics infrastructure at facebook. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*, 1013-1020.

Tushman M, O'Reilly III C (1996) Ambidextrous organizations: Managing evolutionary and revolutionary change. *California Management Review*, 38(4), 8-29.

Van Rompaey K, Verkest D, Bolsens I, De Man H (1996) CoWare-a design environment for heterogeneous hardware/software systems. *Proceedings EURO-DAC'96. European Design Automation Conference with EURO-VHDL'96 and Exhibition*, IEEE, 252-257

Von Hippel E (1994) "Sticky information" and the locus of problem solving: implications for innovation. *Management Science*, 40(4), 429-439.

Wing J (2008) Computational thinking and thinking about computing, *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences,* 366(1881), 3717-3725.

Zander U, Kogut B (1995) Knowledge and the speed of the transfer and imitation of organizational capabilities: An empirical test. *Organization Science*, 6(1), 76-92.