Biol Philos (2014) 29:293–313
DOI 10.1007/s10539-014-9425-3

Engineering and evolvability

Brett Calcott

Received: 9 October 2013 / Accepted: 7 January 2014 / Published online: 1 March 2014

Abstract Comparing engineering to evolution typically involves adaptationist thinking, where well-designed artifacts are likened to well-adapted organisms, and the process of evolution is likened to the process of design. A quite different comparison is made when biologists focus on evolvability instead of adaptationism. Here, the idea is that complex integrated systems, whether evolved or engineered, share universal principles that affect the way they change over time. This shift from adaptationism to evolvability is a significant move for, as I argue, we can make sense of these universal principles without making any adaptationism claims. Furthermore, evolvability highlights important aspects of engineering that are ignored in the adaptationist debates. I introduce some novel engineering examples that incorporate these key neglected aspects, and use these examples to challenge some commonly cited contrasts between engineering and evolution, and to highlight some novel resemblances that have gone unnoticed.

Keywords Evolvability · Adaptationism · Teleology · Engineering · Evolutionary systems biology · Evo-devo

Introduction

Drawing parallels between engineering and evolution typically begins with the following thought:

> The fit between the design of an engineered artefact and its purpose resembles the fit between an evolved organism and its adaptive environment.

B. Calcott (✉)
Center for Advanced Modeling, Emergency Medicine Department, Johns Hopkins University, Baltimore, MD, USA
e-mail: brett.calcott@gmail.com

This *adaptationist* idea suggests a number of enticing but problematic theses: that evolution is a design-like process; that it produces optimal designs; and that it is possible to locate (somewhere) the "blueprint" for an organism's design. The validity and implications of these ideas have been widely debated (see, for example, Dennett 1995; Lewens 2004; Lewontin 1996; Pigliucci and Boudry 2010), and the prominence of the debates sometimes suggests that any mention of engineering in the realm of evolution implies taking an adaptationist stance (see, for example, Boudry and Pigliucci 2013; Griffiths 1996, p. 514).

In this paper, I argue that focusing on *evolvability* rather than adaptationism provides a different perspective on comparing evolution and engineering. I introduce some new engineering examples that highlight these differences and challenge many of the commonly accepted contrasts between engineering and evolution.

Comparing engineering and evolution with respect to evolvability begins with a very different thought:

> Complex integrated systems, whether evolved or engineered, share structural properties that affect how easily they can be modified to change what they do.

This idea, rather than adaptationism, is at the heart of many recent claims about the resemblance between engineering and evolution. For example, biologists have proposed the existence of "… underlying principles that are universal to both biological organisms and sophisticated engineering systems" (Kitano 2004, p. 826), or "… universal principles, relevant to biology, linking modularity with the robust yet fragile nature of complex systems" (Csete and Doyle 2002, p. 1665). Others frame the connection in terms of theories—thus, a theory of evolutionary innovation "… should also apply to non-biological systems" (Wagner 2011, p. 3) or a theory of facilitated variation "… may have some application to engineering and institutional design" (Kirschner and Gerhart 2006, p. 245). In each case, these claims are not about adaptive fit, but refer to properties that affect how both evolved and engineered systems change over time.

I show why invoking the existence of such properties is distinct from making adaptationist claims, and introduce several novel examples that shed new light on when such properties might be shared by evolved and engineered systems. The novel examples play an important role, as debates about the useful parallels between engineering and evolution have been driven largely by simple examples from mechanical engineering (see, for example, Jacob 1977; Lewontin 1996). These examples emphasise design for current function, but tell us little about how complex systems *change over time*, which is the focus in evolvability. The examples I provide are from modern software engineering, a discipline that deals with constant incremental change in large complex systems. Software is occasionally mentioned in the literature on evolvability, but the focus has been on evolutionary search algorithms (Wagner and Altenberg 1996; Boudry and Pigliucci 2013; Pigliucci 2010), rather than the actual practice of building software.[1] These examples I use demonstrate the key role that change plays in some engineering disciplines, and

[1] Work on evolutionary search algorithms is largely done in computer science, rather than software engineering. For some discussion on the difference between the two, see Connell (2009).

highlight various ways that engineers deal with constructing complex systems that must function effectively now, and yet be easy to modify in the future.

The paper proceeds as follows. In the following section, I outline the idea of shared engineering principles, and show how to make sense of such principles without engaging in adaptationist claims. Next, I introduce the notion of diachronic engineering principles in software engineering, and connect this to evolvability. The next two sections identify some different ways of thinking about change in evolution and engineering, and show how the choice of engineering examples can influence our assessment of similarity across the two domains. Following this, I take one candidate shared principle—modularity—and examine to what extent the concept in software engineering maps to that in evolutionary developmental biology. I then turn my attention to the evolution of evolvability, and explain why (despite some initial doubts) engineering may have something to say about this issue. Finally, I recap the benefits of examining the resemblance between engineering and evolution from an evolvability perspective.

Shared engineering principles

This first section lays some groundwork, but does not directly address evolvability. The key message is this: asserting the existence of common structural principles across evolved and engineered systems need not invoke adaptationism. I take the time to make this point as much of the philosophical literature presumes that any mention of engineering (especially reverse-engineering) invokes adaptationism. I use some recent work on gecko's feet to illustrate how the notion of shared engineering "design principles" makes sense even in the absence of adaptationism. These ideas are extended to evolvability in the following sections.

A brief interlude on Geckos

> Using a nanostructure to create an adhesive […] is a novel and bizarre concept. It is possible that if it had not evolved, humans would never have invented it (Autumn and Gravish 2008, p. 1584)

Geckos walk up walls and run across ceilings, they stick to glass as easily as they stick to trees, and they stay sticky even when they are dead. Until recently, the basis of this stickiness remained elusive, as the standard explanations—glue, suckers, and hooks—were shown not to work. Geckos turned out to stick in a way human engineers had never seen before.

Gecko stickiness derives from the structure of their feet, and this structure allows them to make use of inter-molecular, *Van de Waals* forces. The sole of gecko foot is covered with a branching structure ending in many flexible hair-like *setae*. The setae can bend to fit snugly against any surface—close enough for the limited range of inter-molecular forces to work. Each gecko has around 6.5 million of these setae, so although each seta only exerts a tiny force, the sheer number of them generates enough force to support the weight of a gecko. More than a gecko, in fact: Autumn and Peattie

(2002, p. 1085), extrapolating from tests on the stickiness of a single seta, calculated that the amassed setae of one gecko could support the weight of two humans!

Shared principles

> Effective design of gecko-like adhesives will require deep understanding of the principles underlying the properties observed in the natural system. (Autumn and Gravish 2008, p. 1585)

Engineers clearly learnt something from studying geckos' feet, for they now know how to build sticky things in an entirely new way (Greiner 2010; Pugno 2007, 2008). But what exactly was learnt? It was not the existence of some previously unknown physical force—*Van de Waals* forces, and the role they play in other biological processes, are well known. Neither was it simply "how geckos' feet stick", for the materials built by engineers are not exactly like gecko's feet: engineers use polyimide and carbon nanotubes to build their sticky materials, whilst geckos use keratin (Geim et al. 2003; Pugno 2007). Engineers did not blindly copy the geckos' feet, they isolated the relevant properties of geckos' feet that made them sticky, and then built those features into their materials.

Michael French, in his book "Invention and Evolution", calls these *principles of design* "… derived by reflection, or by abstraction from a wide range of practice …" where *practice* includes what appears in living organisms, as well as what engineers have designed (French 1994, p. 180). These principles are not universal laws, but guide engineers seeking to build systems with particular capacities. Engineers building sticky materials now have a new option: they can structure their material according to the principles learnt from studying geckos' feet.

Learning how gecko's feet stick is an example where engineers benefited from principles gleaned from biology. But the reverse is possible too, as principles worked out by engineers can be used to analyse and understand how biological systems work. According to Steven Vogel, this happens more often (at least in biomechanics):

> … biomechanics has mainly been the study of how nature does what engineers have shown to be possible. Nature may have gotten there first, but human engineers, not biologists, have provided us with both analytical tools and practical examples (Vogel 2003, p. 11)

It is no surprise that there are principles that explain existing capacities and guide the construction of similar capacities. Giving good explanations is tightly coupled with our ability to manipulate and control the world (Woodward 2010, p. 315). The better we understand the results of various manipulations on some system, the better we can explain how it works. And the better we understand how to control a system by manipulating its parts, the better we can design and build a mechanism[2] with the precise capacities we desire.

[2] The notion of "mechanism" has become a hotly debated term recently (Dupré 2013; Woodward 2013). I use the term loosely here to mean a collection of parts that act in concert to reliably produce some effect.

Capacities questions and evolutionary questions

Despite talk of *design principles*, the biologists (and engineers) studying geckos' feet make no claim about the adaptive function of the stickiness or the environment that produced it, nor do they claim the stickiness is optimal. Their explanations describe the crucial details that enable a system to produce the stickiness, rather than looking to the evolutionary events that led to the existence of the stickiness, or the conditions that maintain it.

Elucidating the basis of a capacity—such as the stickiness of gecko's feet—is what many biologists think of as "reverse engineering" (see, for example, Swiegers 2012). This contrasts with the way it is often thought of in philosophy, where the task of reverse-engineering is to infer "… the adaptive problem from the solution which was adopted" (Griffiths 1996, p. 514). Even when this alternative way of thinking about reverse-engineering is recognised (for example, Lewens 2004, p. 42), philosophical discussion on design and engineering has almost always focused on an adaptationist reading.[3]

The difference between these two conceptions of engineering parallels some common distinctions in biology, such as Mayr's proximate–ultimate distinction (Mayr 1961) or Tinbergen's "four questions" (Tinbergen 1963). These distinctions come with their own ambiguity, however, especially with regard to evolvability (Calcott 2013a), so I shall state it like this: the engineering principles central to this paper are those used to explain the *capacities* of a biological system rather the *origins and maintenance* of these capacities.

The principles underlying geckos' feet also demonstrate that the resemblance between biology and engineering is not always metaphorical, as some have envisaged the connection (Lewontin 1996; Pauwels 2013; Pigliucci and Boudry 2010). If an engineer builds gloves with a spiderman-like capacities (Pugno 2008), and then remarks that "the gloves stick like a gecko", they are not gesturing toward some vague and evocative similarity, like calling Juliet the sun (Calcott 2013b). They mean that the material in their gloves is structured according to the very same principles that explain how geckos' feet stick.

Engineering the capacity for change

The principles derived from studying geckos' feet explain the capacity for a system to achieve some particular task *at a time*. Evolvability, in contrast, is about principles that explain a systems' capacity to *change over time*. Engineers deal with change too, and look for principled ways to structure systems that change over time. The issue is particularly salient in software engineering.[4]

[3] Bill Wimsatt's work is one exception to this claim. His work on generative entrenchment is meant to span both evolved and engineered systems, and clearly has a connection to how systems change over time.

[4] Much (perhaps all) of what follows may apply to other engineering disciplines. I take it this would count as *more* evidence of the link, rather than an objection to the core ideas in the paper. I focus on software engineering as it is a particularly powerful example, and one that I have some background in.

Anyone using a modern computer, phone, or tablet will have encountered multiple versions of the same software; new features and bug fixes arrive with increasing frequency (along with new bugs, inevitably). This stream of constant updates mean software engineers working on something now often have no idea what features and modifications will appear in future versions. At the same time, engineers build each successive version by modifying the existing software (entirely rewriting software is very rare, though it does occur). So software engineers are both blind to the future, and lumbered by the past.

Because the very software being worked on now will inevitably be modified in the future, a diligent software engineer has two distinct goals: a *synchronic* goal, to make the software do something useful now; and a *diachronic* goal, to make the software easy to modify when the inevitable—but, as yet, unknown—changes arrive.

The diachronic aspect to engineering is often overlooked, as ideas about engineering typically focus on how a mechanism serves an immediate task. Failing to account for this diachronic engineering goal has had a powerful affect on how similar engineering and evolution are thought to be. It is striking, for example, that perhaps the most well-known and influential article comparing evolution and engineering—François Jacob's "Evolution and Tinkering" (Jacob 1977) portrays engineering as having a clear vision of the future, and always starting afresh with ideal materials and tools. Neither is true in software engineering, yet Jacob's view continues to influence those making comparisons between evolution and engineering (for example, Boudry and Pigliucci 2013). Jacob's famous alternative to engineering—tinkering—actually bears a closer resemblance to building software than his portrayal of engineering does, and as a result software often exhibits features held to be distinctive of evolved organisms, such as vestigial parts and haphazard design.[5]

Building software with this diachronic goal in mind is no trivial task. Software must be *robust*, so changes do not break what already works. It must also be *open-ended* so new, unspecified or, as yet, poorly understood, functionality can be integrated with minimal change. All this must be done whilst still making the software do something useful now. Faced with this challenge, engineers have sought general guidelines for constructing robust, open-ended software. The successful ideas now suffuse many aspects of software engineering—from the structure of the programming languages and algorithms, to the tools used to write and test software, and to practices followed during all parts of software development. The use of general principles (often called "design patterns") to guide the construction of robust, open-ended software is now commonplace (see, for example, Gamma et al. 1994).

The way modern software is built emphasises two things about engineering that are easily missed when our ideas stem solely from considering automotive engineering or the like.[6] First, the structure of the system affects not only the task it does now, but also how easily the system can be modified to do other tasks in the

[5] For an interesting discussion of just how ubiquitous this is, see http://www.laputan.org/mud/.

[6] Car manufacturing is a better example. I say more about this in the next section.

future. Second, there are guidelines, or *diachronic design principles* for structuring systems in ways to make these changes easier to accomplish.

Evolvability in biology and software

It is time to link these ideas to evolution. Engineers seek principles to guide them when building complex integrated systems that frequently change. Biologists (some, at least) seek to understand how the structure of complex organisms might constrain or enhance their ability to change over time. In biology, these interests often fall under the rubric of *evolvability*.

Evolvability captures a range of concepts (Pigliucci 2008) or, more accurately, evolvability can be realised in a number of different ways (Brown 2013). The appropriate conception of evolvability to link with engineering is found in evolutionary developmental biology, where it plays a central role (Hendrikse et al. 2007), but also in evolutionary systems biology. In this context, the focus is on the structural properties of individual systems, and how they affect the way new variation can be generated. A clear articulation of this conception of evolvability is Marc Kirschner and John Gerhart's theory of "facilitated variation".

Kirschner and Gerhart, in a 1998 article, identified a number of properties[7] they surmised would confer evolvability (Kirschner and Gerhart 1998) that they now group under a theory of *facilitated variation* (Gerhart and Kirschner 2007; Kirschner and Gerhart 2006).[8] Properties that confer evolvability—or *facilitate variation*—affect the kind of variation a system produces when changed. Kirschner and Gerhart focused on properties that (i) reduce the lethality of changes, (ii) increase the variety of results produced by the changes, and that (iii) reduce the number of changes required to produce adaptive variation (Gerhart and Kirschner 2007, p. 8582).

The goal of finding properties that facilitate variation—and hence make organisms evolvable—resembles the goal engineers have when building robust, open-ended software. Indeed, we can find parallels to each of the general properties that Kirschner and Gerhart's sought. Software should (i) be robust, so changes do not break the existing functionality, (ii) careful construction should permit a variety of future changes to be easily implemented, and (iii) these changes should require as little work as possible. It is unsurprising, then, that the concept of "evolvability" is often deployed in software engineering to capture these goals (see, for example, Breivold et al. 2008; Cook et al. 2000; Lüer et al. 2001).

As before, it is important to maintain the distinction between the capacities a system has and why those capacities are there. As the gecko example demonstrated, engineers and biologists can profit from identifying structural principles that govern a system's capacities to do something *at at time*, whilst ignoring the question of how those capacities got there. The same approach is possible when looking at principles

[7] I sometimes talk of properties rather than principles. Here is the connection: Design principles describe the properties needed to make a system evolvable. But the principles might say more, such as the environmental conditions required for the properties to confer evolvability.

[8] I suspect the change in terminology was made to avoid confusion over the different uses of the term "evolvability".

that govern the capacity for complex integrated systems to *change over time*. Asking why these capacities evolved is, of course, an important question, and I return to it in the section "Two ways to explain the evolution of evolvability".

Same goals, same principles?

Both engineers and biologists seek principles that govern how complex integrated systems change. But do they seek the same principles? Perhaps the way we build things is too different to way evolution builds things. For example, both planes and bees fly, yet the principles that explain how they stay aloft differ (Altshuler et al. 2005). To secure a connection between engineering and evolvability we need to look more closely at how change occurs in both processes and the kinds of structures that enable that change. In the following sections I look at some contrasts that are commonly thought to hold between engineering and evolution. In each case, I show that these contrasts are highly influenced by the engineering discipline we have in mind. A more suitable choice of engineering examples leads to a much closer match.

Mechanics and manufacturers

Here is a thought experiment from Neil Shubin's *Your Inner Fish*, a book that traces the human lineage back to its fishy origins:

> Imagine trying to jerry-rig a Volkswagen Beetle to travel at speeds of 150 miles per hour. In 1933, Adolf Hitler commissioned Dr. Ferdinand Porsche to develop a cheap car that could get 40 miles per gallon of gas and provide a reliable form of transportation for the average German family. The result was the VW Beetle. This history, Hitler's plan, places constraints on the ways we can modify the Beetle today; the engineering can be tweaked only so far before major problems arise and the car reaches its limit (Shubin 2008, p. 185).

According to Shubin: "In many ways, we humans are the fish equivalent of a hot-rod Beetle". He describes a number of human ailments—hernias, hiccups, and choking, for example—that exist because of the limits met when trying to jerry-rig a fish into a human. Shubin draws our attention to an important fact we encountered in the previous section: how things were built in the past affects how they can be modified in the future. Shubin gets his point across, but his example misrepresents how modification occurs in evolution.

When a car mechanic tries to hot-rod a Volkswagen Beetle, they start with a car, change its parts, and end up with the same car (albeit modified). Mechanics thus *modify-in-place*. Evolutionary change, however, begins by copying of a set of developmental resources—largely, but not only, genes. This copying process introduces modifications, and these modified resources then develop into a new organism. Thus evolution, unlike a mechanic, engages in a *copy-modify-and-generate* process.

Modifying the manufacturing of cars, rather than modifying cars directly, is a closer fit to evolution. Here, instead of starting with an actual Volkswagen Beetle, you would begin with the design sketches for one. To change the car, you photocopy the sketches, scribble adjustments on them, and then use these modified sketches to manufacture a completely new car.

The procedure may be more evolution-like, but it is less feasible as a weekend engineering project. Making incremental changes to a car is already time-consuming; making incremental changes to a manufacturing process and repetitively rebuilding a car just to see how fast it goes would be a very slow and laborious process. Software engineering, in contrast, is all about manufacturing.

Manufacturing software

A software engineer is not like a mechanic. They do not act directly on the working parts of a program. Instead, they copy the existing *source code* and modify it. The source code comprises a complex series of instructions in a computer language. A computer language allows the software engineer to capture what they want the program to do. But the source code is not the same thing as a program that runs on the computer. A program must be built, or *generated*, from the source code, much like a car is manufactured from design documents. Software engineering, like car manufacturing and evolution, is a copy-modify-and-generate process.

Unlike manufacturing a car from design documents, however, manufacturing software is fast, cheap, and largely automatic. The process is so easy, in fact, that it becomes part of the design process. Every software engineer effectively modifies the design documents, manufactures the software, and then test drives it; often many times a day. As Jack Reeves (1992) points out, in software engineering, the source code is the design document.

Failing to notice the importance that manufacturing can play in an engineering discipline can also affect how similar we judge engineering and evolution to be. Richard Lewontin, in his paper "Evolution and Engineering", recognises that engineered systems, like organisms, each undergo an individual historical process: organisms develop and artifacts are manufactured. But he goes on to say that, in the case of artifacts:

> … a description of the process of their manufacture is irrelevant to their use and maintenance. My car mechanic does not need to know the history of the internal combustion engine or possess the plans of the automobile assembly line to know how to fix my car (Lewontin 1996).

Software engineers, however, must know how the assembly line works, for they manufacture software every day. Moreover, just as knowing the history of human evolution can help us understand hiccups and hernias, knowing the history a particular software product is often essential for understanding why it works or, more often, why it fails. Software engineers even possess tools for examining the history of changes to software, for this history can often help identify the where the "hiccups and hernias" come from.

Differentiating manufacturing from mechanics is not a novel claim. But it is important to maintain the distinction (which Shubin does not) and to realise the central role manufacturing plays in some engineering practices (which Lewontin does not). When parallels between organismal development and manufacturing are made, however, they face a number of very powerful objections. Dealing with these objections is the subject of the next section.

Blueprints, build systems, and genotype–phenotype maps

I've suggested that evolutionary change can be likened to change in manufacturing—design documents are changed, and then used to regenerate a modified system. The "design document", in this way of thinking, is the genotype (and perhaps some other stably inherited resources), and development is the process that maps a genotype to a phenotype.[9] Calling the genotype the "design document" will, to many, sound over-simplistic and have the whiff of genetic determinism. Again, however, our notion of a design document is deeply influenced by the kind of engineering we have in mind.

A blueprint is the common example of a design document, but it is a poor analog of the way a genotype maps to a phenotype. Blueprints have a simple, one-to-one, isomorphic relationship with the finished artifact. For example, blueprints often have a scale, suggesting all measurements on the blueprint have a simple scaling relationship to the final measurements of the artifact. This simple relationship permits us to immediately grasp the effect any change made to the blueprint will have on the finished artefact. Changes can be simply "read off" the design documents with no understanding of the intervening processes. In contrast, the complexity of the process that maps a genotype to a phenotype means our ability to predict phenotypic change by merely looking at a genetic change is rare.

Furthermore, the complexity of this process that maps the genotype to a phenotype contributes to the evolvability of a system (Pigliucci 2010). It can, for example, make the system much less brittle under change, as the intervening process can buffer change in various ways. So the "generate" part of the "copy-modify-and-generate" process plays a key role in what makes a system evolvable.

Not all design documents look like blueprints, however. In the next section, I argue that the relationship between source code and a large complex software system looks nothing like a blueprint. The generative process used to construct a complex software system has some features in common with organismal development and, furthermore, these features play a role in what makes the system evolvable.

Build systems

In the early days of programming, all programs were laboriously written using the same instructions that computer processors used (even earlier, programming was

[9] For a discussion of the connection between development and the genotype-phenotype map, see Pigliucci (2010).

done by modification-in-place, by fiddling with the hardware). Today, however, almost all work is done in "high-level" languages, whose structure has many features with no simple correlate in the set of instructions that actually run on a computer. The process that maps the source code into computer instructions is no longer a simple one-to-one mapping. Things are even more complex when we shift from building simple programs to the construction of large integrated software systems—the kind that may run in a large bank, or government department. These systems often comprise many related programs, written in different computer languages, and perhaps running on different operating systems. Generating all of these integrated programs from source code requires a suite of processes—sometimes called a *build-system*—responsible for turning the various pieces of source code into the final integrated set of programs (for some examples, see Smith 2011).

A build-system can have many stages, and complex interactions between the stages. Some stages run tests to ascertain what further steps they need to do, or to change how others stages in the build process will proceed. The chain of steps can become long, with many different transformations taking place before the final usable system is produced. Depending on the environment in which the software is built, it may also produce a number of different *build-variants*—versions of programs suited to different operating systems or optimised for different tasks.

In these systems, the mapping between what the software engineer modifies and the resulting program is extremely complex, and the relationship between the source code that is copied-and-modified and the final complex running system has little in common with the relationship between a blueprint and a car. It is far from direct, and understanding the downstream effect of changes to the source code in large systems requires an in-depth knowledge of how the build process works. More importantly, however, the complexity of this build process is partly the result of engineers trying to make software more evolvable.

Juggling synchronic and diachronic goals

The synchronic and diachronic goals in software engineering often constrain one another—what makes a system do its current job well is not always the same as what makes it easy to modify over time. The complex mapping between source code and program is used to relax the constraints between these two goals so both can be achieved together. Here are two examples.

- *Example 1* A common synchronic goal when building software is to make it run *fast*. The traditional way to optimise code was to work with a "low-level" language. A low-level language has simple, direct mapping between the code and the final program, and allows engineers to fine-tune a program to run as fast as possible. But low level languages are verbose, extremely fragile, and often rely on assumptions about the hardware that can change. These features make modification difficult, and hence affect the diachronic goal. In more recent years, the process of optimisation has become automated as part of the build process. Engineers can then focus on writing simpler code that is more robust to change.

This makes the mapping between the source code and the resultant program far more complex, for the build process must transform the simple representation coded by the engineer into something very different, often targeted specifically to make the most of the particular processor being used.

- *Example 2* Often, several parts of a program interact with one another, so that change in one part of the program requires change in related parts of the system. To modify the system in these cases, engineers must make several simultaneous coordinated changes across the system. Inevitably, these changes fail to be coordinated, and the system does not work. This problem can be cured by putting an extra step into the build-chain, which provides a single central place where the change can be made. The build system then automatically generates the coordinated changes across the system.

In both cases the increased complexity of the mapping makes the simultaneous pursuit of a synchronic goal and a diachronic goal far easier, making the system more evolvable.

Neutral change and code refactoring

There is a further, more general, connection between software engineering and recent ideas about evolvability. Redundancy, or neutral change, in the genotype-phenotype map is important to understanding evolvability (Munteanu and Solé 2008; Wagner 2007). Although a change to the genotype may have no impact on the phenotype, the change may nonetheless be important, as can make further change more easily attainable. A series of small neutral changes may eventually make an adaptive change available. It is for this reason that neutral change is central to Andreas Wagner's theory of innovation (Wagner 2011).

Software engineers take advantage of neutral change too, when they engage in "code refactoring". Code refactoring changes the underlying source code without changing what the program actually does. Refactoring in software is possible because redundancy exists in the mapping between source code and program. There are many different ways to realise the same functionality, so software engineers can modify the source with no impact on the way the resulting program runs.

Redundancy explains why code-refactoring is possible, but why do engineers bother changing code if the behaviour of the program stays exactly the same? Typically such change occurs prior to a functional change. Here is a common case. A software engineer wants to re-use some existing functionality, but the desired section of code for performing this functionality is mixed together with other code whose functionality is not required. To re-use the code, it must first be separated into two functional parts. These parts can be recombined to do exactly what they did before. Now, however, it is now possible to re-use the separated functionality independently of the other code. Again, this emphasises the importance of considering both synchronic and diachronic perspectives in engineering: two different structures may be equivalent with respect to synchronic goals (they provide the same functionality), but not to diachronic goals (re-using functionality is easier when the functionality is separated).

How close is the fit?

The mapping between source code and a running software system bears little resemblance to the way a blueprint works. Furthermore, the complexity of the mapping can play an important role in making software more evolvable. But does this mapping resemble anything biological?

To assess the similarity, we need some ways to characterise the kind of mapping going on. Pigliucci, following others, distinguishes *direct* and *indirect* coding, as a way of picking out the difference between the way a blueprint works and the relationship between genotype and phenotype (Ciliberti et al. 2007; Pigliucci 2010). So is the complex process for building large-scale software systems direct or indirect? These concepts are, unfortunately, not well-defined. For example, feedback and plasticity are present in many build systems; but it isn't clear whether these are sufficient to make the mapping relationship indirect.

A more promising approach for characterising the mapping process is Andreas Wagner's work on neutral spaces—a subject I touched on above. Wagner offers a broad theory of innovation that is explicitly meant to capture both evolved and engineered systems (Wagner 2011, see also Raman and Wagner 2011). Neutral change is central to this theory, for it explains "… how biological systems can preserve existing, well adapted phenotypes while exploring myriad new phenotypes" (Wagner 2011, p. 3). Neutral change is possible because many different genotypes map to the same phenotype. In software we saw the same phenomenon—code is changed by refactoring, yet it maintains the same function.

Although neutral spaces provide a unified way of thinking about change in complex systems, they cannot provide the whole story. This is because the capacity for neutral change can be realised in many different ways.[10] Thus, even if engineered and evolved systems both exhibit neutral change, we still need to establish that they gain this capacity *in the same way* if we are to identify common structural principles that are applicable in both engineered and evolved systems. This is far from obvious, for there are clearly some features of organismal development, such as the robust self-organising capacities of cells (Newman and Bhat 2009) that can contribute to neutral change, yet has no analog in human engineered software. In the next section, I explore one particular property—modularity—that is thought to exist in both evolved and engineered systems.

Interfaces: integrating and re-using modules

Modularity is an important property in software engineering, but the reason for its importance has changed over time. Early programming practices decomposed a system into modules based on related chunks of processing, often guided by a flow chart of the required operations. This decomposition was important for the simple reason that it was expedient to divide labour when constructing a large system. In

[10] The fact that neutrality can be realised in many different ways is a positive feature of the theory, as it enables Wagner to apply these ideas across many different levels of organisation.

1972, David Parnas wrote an influential paper that identified an alternative way to decompose a system into modules, focusing on what he called "information-hiding" (Parnas 1972).

Information-hiding in software engineering is captured by distinguishing between *interface* and *implementation*. The interface defines how the module interacts with the rest of the system. The implementation defines specifically how the module goes about its job.

Here is a simple example. Consider building (as part of a larger project) a software module that sorts a list of words. The interface defines what input the module accepts (a list of words, in this case), and what output the module produces (the same list, sorted in alphabetical order). The definition of the interface, however, says nothing about the implementation—how the actual sorting takes place. These details are "hidden" from other parts of the system. There are many algorithms that can sort a list, and any one of these might be used in the implementation of the module.[11]

Isolating change

Distinguishing the interface from the implementation confers an obvious advantage: the implementation can be modified without affecting the interface, as its workings are "hidden". A faster sorting algorithm might be found, or perhaps different algorithms could be selected that are most appropriate to the size of the list provided. As long as the interface remains consistent, making these changes will not break how the rest of the system works. By making the sorting modular, and hiding the implementation behind the interface, we decouple the dependencies between different parts of the system. Changes to our sorting algorithm are thus *isolated*, and can be made without affecting the global behaviour of the system.

As Parnas noted, this choice of decomposition is guided by thinking about how the system might change. A decomposition that favours information hiding, rather than simply the flow of processing, will be easier to modify in the future, assuming the decomposition correctly identifies the way change is likely to occur in the system. Parnas's paper marks out a significant shift from synchronic thinking (modular decomposition in terms of the processing) to diachronic thinking (modular decomposition in order to simplify change in the future).

These ideas should sound familiar to biologists interested in modularity and evolvability. Modularity is often thought to be important for it allows some traits to change without affecting others. Wagner and Altenberg, for example, describe modularity as a way of reducing pleiotropy (Wagner and Altenberg 1996). The lack of pleiotropy allows one set of genes affects only one character, whilst another set of genes to affect only a second character. This kind of modularity permits the "independent genetic representation of functionally distinct character complexes" (p. 971).

[11] Programmers might note that this simple example could describe a "function", rather than a module. True, but I am trying to keep things simple here. Just imagine it is a module with a single function.

If we look closely, however, an important feature in software modularity is absent from the conception of modularity presented by Wagner and Altenberg. In software engineering, the implementation of one module can be changed without affecting another module, because the source code that represents each module is independent, or "hidden" from the other. This much is the same as Wagner and Altenberg's ideas. In software development, however, whilst the implementation can change, the interface remains constant, and this interface defines how the module *interacts* with the rest of the system. Wagner and Altenberg's conception of modularity does not appear to include any notion of interaction between modules. So although both conceptions of modularity allow for independent change of modules, in software engineering a key idea is the ability to independently modify modules whilst they remain functionally *integrated* with the rest of the system.

The stability of the interface in software engineering modules permits independent change within modules, without requiring the modules be functionally distinct. Integration becomes even more important when we consider a further reason that modularity is thought to affect evolvability in both engineering and evolved systems.

Re-using modules

A second important reason for constructing modules in modern software development is for *re-use*. Rather than writing a sorting module every time we need to sort something, we re-use the same sorting module. The re-use of modules also allows them to be *recombined*, permitting novel functionality to be constructed by organising a number of different modules to work together.

Similarly, in biology, modularity is often thought to be significant for this second reason: modules can be re-used and recombined, allowing new functionality to be constructed from old parts. Like engineering, the thought is that structures that have already proved useful can be re-deployed in a new context to produce some novel feature (for example, Monteiro 2012).

So modules are not just used to isolate change—sometimes they enable re-use and recombination. These two roles for modularity have different applications. Isolating some function in a system within a module makes a system robust to change, whilst making a module reusable makes it easy to create new functionality. Sometimes these roles are distinguished (Sterelny 2004, pp. 496–497), but sometimes they are not (Wagner 2007, pp. 88–89).

The distinction between isolating change and re-using functionality is worth making, however, for these two goals place different constraints on the modules. In software engineering, for modules to be re-used, and especially for them to be effectively recombined, an interface needs to be both standardised and simple to use.

To see why, consider our sorting module again. The interface expects a list of words. In a computer, the characters of a word are represented by numbers—for example, the number 65 might be an 'A', 66 a 'B', and so on. The way that the characters map to a number is called an "encoding", and there are many different encodings that can be used—each with different ways of mapping numbers to

letters. For an algorithm to sort a set of words correctly, it needs to know what encoding has been used. Now, if the sorting module is only ever used in one place in a single system, we can easily ensure we deliver an encoding that it can work with. But if we want the module to be re-used and recombined in the future, we need to ensure that all uses of the module deliver their word list in the same encoding as the module expects. Alternatively, the module itself may detect and then translate the given encoding to one appropriate for sorting.

Here is the point: a module that already isolates function will often require extra work to make it re-usable, for the interface must now interact reliably with *multiple* other parts of the system. What matters for re-usability is not simply the isolated implementation, but how easily the interface can be subsequently re-used by the system.

Interfaces in biology?

Modules in software engineering have two distinct uses: to isolate change, and to allow re-usability. Both ideas can be found in the literature on evolvability, but they are not always clearly demarcated. This is important for, at least in software engineering, the two roles demand overlapping, but not equivalent properties. Furthermore, in software engineering, the notion of an interface is essential for integrating a module within a system, and for permitting the re-use and recombination of modules.

Although modularity is frequently mentioned, interfaces have rarely been discussed in the literature on evolvability. Marie Csete and John Doyle are one exception (though they call them "protocols"). They claim that "[…] protocols are far more important to biologic complexity than are modules." (Csete and Doyle 2002, p. 1666). They suggest a simple analogy with Lego building bricks: each brick is a module, but what makes it possible to construct a huge variety of structures is the patented way they snap together.

Can we find anything in the evolvability literature that looks like an interface? One idea that captures something like the notion of information-hiding is the distinction between "instructive" and "permissive" causes in developmental biology (Gilbert 2000, pp. 142–143). Instructive causes are signalling processes that provide key information about how the downstream developmental processes should work. Permissive causes, in contrast, are signalling processes whose presence merely releases the self-contained capacity of some downstream developmental process. We might say that a permissive cause says, "do it", whilst an instructive cause relays "what must be done". Something like an interface-implementation distinction is here, for a permissive cause could initiate a downstream process, even if what the downstream process does (its "implementation") was changed.

I said above that this kind of re-use and recombination relies on simple interfaces. This idea may be even more important in evolution. If some existing functionality requires a very complex and precise context to operate, then it is unlikely these conditions will arise through random mutation. If, in contrast the initiating

conditions (the "interface") are relatively simple, it may well be possible for the re-use of such functionality to arise by chance mutations.

These ideas have been picked up in the evolvability literature, though not in relation to modularity. Kirschner and Gerhart's notion of "weak-linkage"—a property that facilitates variation—is sometimes likened to an interface, or protocol. They suggest that weak-linkage resembles the ease with which electrical plugs and sockets, because of their standardised design, can be swapped and interchanged. This is how interfaces are thought of in software. Weak signalling thus appears to have two ideas related to the interface/implementation distinction. First, how some downstream process, or implementation, actually works is independent from how it is initiated, and also that initiating conditions are undemanding, and can be "easily broken or redirected for other purposes" (Kirschner and Gerhart 2006, p. 111).

Clarity about modularity

Exploring the role that interfaces play in software modularity provides a number of insights that are not well explored in the biological literature. In both software and biology, modularity can make a system more robust, by isolating change to certain parts of the system. But interfaces in software permit the different parts of the system to remain integrated whilst they change. This is an important idea, but one that is often absent from much discussion about modularity in biology. Second, modules that isolate change are not necessarily fit for re-use or recombination, as this second role often requires standardising and simplifying the interface. Again, there is little discussion of these two distinct roles in the biological and literature, or the suggestion that they may place different constraints on the way modules are constructed.

The concept of modularity in software engineering clearly has some important connection to ideas about evolvability in biology. Although a precise match between any principles was not found, the attempt to find such a link exposed a number of areas that could benefit from further investigation.

Two ways to explain the evolution of evolvability

I've argued that we can make sense of evolved and engineered systems sharing properties that affect their evolvability, but I have specifically avoided discussing where such properties might come from. It might seem that engineering has little to say on this subject, for even if some property—such as modularity—appears both in software and in nature, we cannot explain how it go there in the same way. Engineers put modularity into software, but no agent put modularity into evolved systems. Shared engineering principles might tell us about the capacities of complex systems, but they say nothing about how those capacities got there.

As I have argued elsewhere, however, there are two different questions we might have in mind when we ask where a trait comes from (Calcott 2009, 2013a). One question asks about the population dynamics that drive and stabilise a particular trait. A second question asks about the trajectory of gradual change that needs to be

available for evolution to act. This second question often arises in the context of explaining how complex adaptations or novel traits could evolve. An engineering example can show why distinguishing these two questions is important for understanding evolvability.

Poorly written code is common in software engineering. It might be poorly written for synchronic reasons (perhaps it runs slowly), or it may be poorly written for diachronic reasons (perhaps interacting parts of system are not properly isolated). Enhancing or updating software that is poorly written from a diachronic perspective can be frustrating task, but it presents an opportunity for a diligent software engineer. As well as making the required modification to update what the software does, it may also be possible to modify the structure of the software so that further modifications are easier to make. For example, we could modularise parts of the code, isolating parts that are likely to change in the future.

This second task demonstrates how an engineer can respond informatively to the question: "how did modularity get in the system?" They can do so by citing the series of changes that turned a previously non-modular system into a more evolvable modular system.

This response shows there there are two different questions we can ask about the evolution of evolvability. The first question is the one most commonly addressed in the literature: what population-level processes might change or stabilise some candidate property, such as modularity. If we take an engineering perspective, however, a second question arises: what structural changes could gradually increase (or decrease) the prevalence of modularity (or some other relevant property) in a system? This question *does* have a parallel in engineering. If evolved and engineered systems can share properties that affect their evolvability, it is also possible that the structural modifications that increase or decrease evolvability are shared too. In principle then, ideas from engineering can contribute to understanding how evolvability evolves, and they can do so without making adaptationist claims.

Conclusions

Comparisons between engineering and evolution have a long history, but the debate has largely centered on adaptationist ideas. The central importance of optimality and current function in adaptation has meant the engineering examples used for comparison have focused on synchronic design, whilst the importance of change and the diachronic challenges faced by engineers has been ignored. As I've argued throughout the paper, neglecting these aspects has perpetuated a number of contrasts between engineering and evolution that are far less convincing once we broaden our range of engineering examples.

These arguments have two obvious implications for further debates. Those promoting or dismissing similarities between evolution and engineering should be careful to both (a) articulate whether the context of their claims is adaptationism or evolvability, and (b) use examples that best portray the variety and complexity that engineering has to offer in the relevant context.

There are more far-reaching implications, however. I've argued that a number of rather blunt, black-and-white distinctions between engineered and evolved systems cannot be sustained. Engineers don't always begin afresh, nor do they always have a preconceived plan, and so optimal functioning in engineered systems may be impeded by previous choices or traded off against future flexibility. Moreover, manufacturing is central to some engineering disciplines, and design documents needn't be a simple as blueprints. The failure of these dichotomous views suggests that there is both an opportunity for finding a far richer set of connections between the structure of evolved and engineered systems, and also a far better set of resources for reflecting on what genuinely distinguishes the two. I suggested that self-organisation is an important property not found in engineered systems; perhaps noise (Eldar and Elowitz 2010) and certain kinds robustness are other such properties (Wagner 2007, p. 313). Clearly, there may also be properties in engineered systems that have never evolved as well. But the discovery of a concept in engineering that doesn't exist in the evolutionary literature may provide a new approach, or new twist on an old idea. The notion of interfaces (or protocols, as Csete and Doyle call them) come directly from engineering, but has not yet been well-explored in the literature on evolvability.

Lastly, although I've sought to distinguish these ideas from adaptationism, the examples I've discussed do provide a perspective on adaptationism that contrasts with the standard story of "good design". In software engineering at least, synchronic goals and diachronic goals both play a role in design, suggesting that a complete understanding of current structure cannot be obtained simply by focusing on current function, as is typically the case in adaptationist analysis (see Lewens 2002). Does this idea translate to an evolutionary setting? Some recent work on evolvability suggests it might. In models of both RNA folding and Boolean networks, it is possible to select for complex structures that can be fully explained only by reference to both how they function at a time, and how they possess the capacity for rapid change over time (Parter et al. 2008).

Acknowledgments This paper has a long history. Early formulations benefited from feedback at the Philosophy of Biology at Dolphin Beach conference, at ISHPSSB, and at the "Progress by Design" conference in Bielefeld. A draft paper by Ian Wills, and a long café discussion with Dan Nicholson prompted me to think more deeply about the connection between engineering and evolved systems. Arnon Levy, Michael Weisberg, Maureen O'Malley, Emily Parke, Kim Sterelny and two anonymous reviewers provided useful comments (and words of encouragement) on later versions. This work was supported by a Australian Research Council Postdoctoral Fellowship and a Visiting Fellowship at the Konrad Lorenz Institute for Evolution and Cognition Research.

References

Altshuler DL, Dickson WB, Vance JT, Roberts SP, Dickinson MH (2005) Short-amplitude high-frequency wing strokes determine the aerodynamics of honeybee flight. Proc Natl Acad Sci 102:18213–18218

Autumn K, Gravish N (2008) Gecko adhesion: evolutionary nanotechnology. Philos Trans R Soc A Math Phys Eng Sci 366:1575–1590

Autumn K, Peattie A (2002) Mechanisms of adhesion in geckos. Integr Comp Biol 42:1081–1090

Boudry M, Pigliucci M (2013) The mismeasure of machine: Synthetic biology and the trouble with engineering metaphors. Stud Hist Philos Sci Part C Stud Hist Philos Biol Biomed Sci 44:660–668
Breivold HP, Crnkovic I, Eriksson PJ (2008) Analyzing software evolvability. Presented at the computer software and applications, 2008. COMPSAC'08. 32nd Annual IEEE International, pp 327–330
Brown RL (2013) What evolvability really is. Br J Philos Sci doi:10.1093/bjps/axt014
Calcott B (2009) Lineage explanations: explaining how biological mechanisms change. Br J Philos Sci 60:51–78
Calcott B (2013a) Why how and why aren't enough: more problems with Mayr's proximate-ultimate distinction. Biol Philos 28:767–780
Calcott B (2013b) Engineering: biologists borrow more than words. Nature 502:170
Ciliberti S, Martin OC, Wagner A (2007) Innovation and robustness in complex regulatory gene networks. Proc Natl Acad Sci 104:13591–13596
Connell C (2009) Software engineering !=computer science. Dr Dobb's: the world of software development. Retrieved 4 Mar 2011, from http://www.ddj.com/architecture-and-design/217701907
Cook S, Ji H, Harrison R (2000) Software evolution and software evolvability. University of Reading, UK
Csete ME, Doyle JC (2002) Reverse engineering of biological complexity. Science 295:1664–1669
Dennett DC (1995) Darwin's dangerous idea. Simon & Schuster, New York City
Dupré J (2013) I-living causes. Aristot Soc Suppl 87:19–37
Eldar A, Elowitz MB (2010) Functional roles for noise in genetic circuits. Nature 467:167–173
French MJ (1994) Invention and evolution. Cambridge University Press, Cambridge
Gamma E, Helm R, Johnson R, Vlissides J (1994) Design patterns. Pearson Education, Pearson
Geim AK, Dubonos SV, Grigorieva IV, Novoselov KS, Zhukov AA, Shapoval SY (2003) Microfabricated adhesive mimicking gecko foot-hair. Nat Mater 2:461–463
Gerhart J, Kirschner M (2007) The theory of facilitated variation. Proc Natl Acad Sci 104(Suppl 1):8582–8589
Gilbert SF (2000) Developmental biology, 7th edition, 9th edn. Sinauer Associates, Sunderland
Greiner C (2010) Gecko-inspired Nanomaterials. In: Kuma CSSR (ed) Biomimetic and bioinspired nanomaterials. Wiley-VCH, New York
Griffiths PE (1996) The historical turn in the study of adaptation. Br J Philos Sci 47:511–532
Hendrikse JL, Parsons TE, Hallgrímsson B (2007) Evolvability as the proper focus of evolutionary developmental biology. Evol Dev 9:393–401
Jacob F (1977) Evolution and tinkering. Science 196:1161–1166
Kirschner M, Gerhart J (1998) Evolvability. Proc Natl Acad Sci 95:8420–8427
Kirschner MW, Gerhart JC (2006) The plausibility of life: resolving Darwin's dilemma. Yale University Press, New Haven
Kitano H (2004) Biological robustness. Nat Rev Genet 5:826–837
Lewens T (2002) Adaptationism and engineering. Biol Philos 17:1–31
Lewens T (2004) Organisms and artifacts. Bradford Book
Lewontin RC (1996) Evolution as engineering. In: Collado-Vides J, Magasanik B, Smith T (eds) Integrative approaches to molecular biology. MIT Press, Cambridge, MA
Lüer C, Rosenblum DS, van der Hoek A (2001) The evolution of software evolvability. Presented at the proceedings of the 4th international workshop on principles of software evolution, pp 134–137
Mayr E (1961) Cause and effect in biology. Science 134:1501–1506
Monteiro A (2012) Gene regulatory networks reused to build novel traits: co-option of an eye-related gene regulatory network in eye-like organs and red wing patches on insect wings is suggested by optix expression. BioEssays 34:181–186
Munteanu A, Solé RV (2008) Neutrality and robustness in evo-devo: emergence of lateral inhibition. PLoS Comput Biol 4:e1000226
Newman SA, Bhat R (2009) Dynamical patterning modules: a "pattern language" for development and evolution of multicellular form. Int J Dev Biol 53:693–705
Parnas DL (1972) On the criteria to be used in decomposing systems into modules. Commun ACM 12:1053–1058
Parter M, Kashtan N, Alon U (2008) Facilitated variation: how evolution learns from past environments to generalize to new environments. PLoS Comput Biol 4:e1000206
Pauwels E (2013) Communication: mind the metaphor. Nature 500:523–524
Pigliucci M (2008) Is evolvability evolvable? Nat Rev Genet 9:75–82
Pigliucci M (2010) Genotype-phenotype mapping and the end of the "genes as blueprint" metaphor. Philos Trans R Soc B Biol Sci 365:557–566

Pigliucci M, Boudry M (2010) Why machine-information metaphors are bad for science and science education. Sci Educ 20:453–471
Pugno NM (2007) Towards a spiderman suit: large invisible cables and self-cleaning releasable super adhesive materials. J Phys: Condens Matter 19:395001
Pugno NM (2008) Spiderman gloves. Nano Today 3:35–41
Raman K, Wagner A (2011) Evolvability and robustness in a complex signalling circuit. Mol BioSyst 7:1081–1092
Reeves JW (1992) What is software design. C++ J 2. Retrieved from http://user.it.uu.se/~carle/softcraft/notes/Reeve_SourceCodeIsTheDesign.pdf
Shubin N (2008) Your inner fish. Pantheon Books, New York
Smith P (2011) Software build systems. Addison-Wesley Professional, Boston
Sterelny K (2004) Symbiosis, evolvability, and modularity. In: Schlosser G, Wagner GP (eds) Modularity in development and evolution. University of Chicago Press, Chicago
Swiegers GF (2012) Bioinspiration and biomimicry in chemistry. Wiley, New York
Tinbergen N (1963) On aims and methods of ethology. Zeitschrift für Tierpsychologie 20:410–433
Vogel S (2003) Comparative biomechanics. Princeton University Press, Princeton
Wagner A (2007) Robustness and evolvability in living systems. Princeton University Press, Princeton
Wagner A (2011) The origins of evolutionary innovations: a theory of transformative change in living systems. Oxford University Press, Oxford
Wagner GP, Altenberg L (1996) Complex adaptations and the evolution of evolvability. Evolution 50:967–976
Woodward J (2010) Causation in biology: stability, specificity, and the choice of levels of explanation. Biol Philos 25:287–318
Woodward J (2013) II-Mechanistic explanation: its scope and limits. Aristot Soc Suppl 87:39–65