CODE VALLEY – A PEER-TO-PEER SOFTWARE ENGINEERING SYSTEM

Noel Lovisa
Julie Lovisa

Abstract: By enduring the software crisis for so long, we have largely become numb to (and accepting of) its effects. Large scale projects fare the worst with crippling losses and blown schedules occurring more often than not. To date, the very essence of software – its complexity – has proven all but impossible to manage. In direct contrast, other legitimate *industries* routinely manage their complexities. Coincidentally, in 1968 McIlroy observed that the "*software industry is not industrialised*" and the intervening decades have only served to reinforce this observation. Ironically, whilst it is easy to specialise in software, it is virtually impossible to build a viable business as a specialist, thus denying industrialisation its most vital basis – specialisation. We propose to shift developers from the 'code-domain' to a decentralised peer-to-peer network operating wholly in a 'design-domain', where engineers now co-operate to make *design-contributions.* This shift offers a viable medium for capturing their "*special capability in a product"*, thus preserving prospects for repeat business, and ultimately reinstating specialisation. With its most vital basis restored, McIlroy's long-awaited industrialisation of software can begin.

Introduction

> *"There are no inventions that will do for software productivity, reliability, and simplicity what electronics, transistors, and large-scale integration did for computer hardware."*
>
> (Brooks Jr. 1987)

For nearly half a century, the software industry has been in a perennial state of crisis. In 1968, a NATO-sponsored conference was called to chart a way out. While little changed as a result, the term 'Software Engineering' was coined for the disciplined approach it represented and as a tacit recognition of the legitimate engineering disciplines that had left software so far behind. While many attempts have been made to realise this aspiration, and some improvements to lessen the sting of the software crisis have transpired, the fact remains that these methods are reactionary rather than preventative, and have addressed the "*accidental difficulties*" of software rather than the "*essential*". As long as the essence of software remains insurmountably complex, 'Software Engineering' will forever remain a term of aspiration.

> *"The software industry is not industrialised."*
>
> (McIlroy 1968)

Like Ford and Whitney before him, McIlroy recognised that mass-production was the key to industrialisation. History reveals that mass-production is built upon principles of standardisation, interchangeability and reductionism. Whilst it is true that 'standardisation', through the advent of software libraries, modularity, object-oriented programming and the like, has permitted some degree of interchangeability, a mass-produced software components industry has yet to emerge.

> *"The real price we pay is that as a specialist in any software technology you cannot capture your special capability in a product."*
>
> David A. Fisher (in Gibbs 1994)

Inherent in the concept of reductionism is specialisation – an industry specialist is one who is an expert at constructing their respective interchangeable part. At present, the commercial viability of a specialist is undermined by the very essence of software. Ironically, whilst it is very easy to specialise, it is virtually impossible to build a viable business as a software specialist. Thus, with no commercial incentive to become a specialist, software developers are forced to languish as pre-industrial generalists, left in the dust by their hardware engineer counterparts. Sadly, a software industry composed of generalists can never enjoy its own industrial revolution – the crisis will persist.

In this paper, we offer a fresh approach to software development, which permits the capture of Fisher's "*special capability*" in a product, thus reinstating specialisation for McIlroy's industrialisation of software, and thus clearing a path for Brooks Jr.'s order-of-magnitude improvements in software productivity, reliability and simplicity.

Industrialisation

The Industrial Revolution is considered a revolution for good reason. The shift from pre-industrial methods to machines heralded such improvements in productivity, quality and costs that today, economic historians agree that the onset of the Industrial Revolution is the most important event in the history of humanity since the domestication of plants and animals (McCloskey 2004).

McIlroy was wise to pursue industrialisation for the turning point it offered the software industry. However, this pursuit of industrialisation was never fully realised, and McIlroy's 1968 observation that the software industry was not industrialised remains valid decades later. Current methods of software development, while more sophisticated in their arrangement and deployment, differ little from those of the late 60s. Clearly, a formidable barrier to industrialisation exists.

In a truly industrialised software system, software would be developed using a supply-chain, where supply and demand, coupled with competition and innovation, drive production. Typically, a client would contract a specialist supplier and provide software requirements. The supplier would then design and deliver software satisfying those requirements in return for suitable remuneration. However, unbeknownst to the client, the product delivered by the supplier would actually be an assemblage of other products provided by sub-contracted suppliers and so on. Such supply-chains are spectacularly successful in other industries, where the application of reductionism, interchangeability and standardisation, has led to specialisation, automation and intense competition. As a result, costs are driven down while quality, performance and speed are driven up in a continuous cycle that rewards innovation.

Intellectual property exposure

The barrier to industrialisation is an inability to specialise, or as Fisher puts it, "*you cannot capture your special capability in a product*" (Gibbs 1994). We contend that the problem is not so much that you cannot capture your special capability in a product, but that you leak it when you make a sale. When a software specialist's intellectual property predominantly lies within the *code* of a software component, the specialist, out of necessity, exposes their intellectual property upon component delivery. By harming prospects for repeat business, the business model is compromised, along with any opportunity for viable monetisation.

The source of this intellectual property leakage can be traced to three main facets of component delivery – *integration, portability* and *re-usability*.

Under the incumbent software development doctrine, simplifying integration and making components re-usable and portable are worthy objectives when striving for improved productivity and lower costs. However, any support provided by the supplier to assist with a component's *integration* can be viewed as exposing intellectual property rightly belonging to the supplier – an unavoidable consequence when it is the client's responsibility to integrate the components.

The inclusion of additional functionality to determine some of the run-time context permits a more context-independent component with a simpler interface, making the component easier to use. However, this functionality casts some of the supplier's intellectual property into the component, leaking intellectual property rightly belonging to the supplier – an unavoidable consequence when seeking *portability*.

Lastly, making a component interchangeable fosters competition, as it permits more than one supplier of a particular component. Unfortunately, it also requires standardising and publishing the component interface, thereby exposing intellectual property rightly belonging to the supplier – an unavoidable consequence when encouraging *re-usability*.

If a software development process can be determined in which the client integrates the component without knowledge of the component interface or assistance from the supplier, whilst also ensuring the component is interchangeable but not economically portable or re-usable, then we will have a solution for viable specialisation and the race to industrialise can begin.

Intellectual property protection

Protecting the supplier's intellectual property is a formidable challenge. On the one hand, components should be interchangeable and easy for a client to integrate. On the other, it is necessary to withhold interface information and assistance from the client and even stymie portability and re-usability in the interests of preserving supplier economic viability. To break this impasse, something essential has to change.

Simplifying integration

Component integration requires a good understanding of the component interfaces, combined with sophisticated programming skills to develop the glue code necessary to complete the design.

> *We propose to simplify component integration to its irreducible minimum – concatenation.*

We recognise that if the client is required to do more than concatenate in order to integrate their purchased components, then there is leakage of intellectual property.

Reducing portability

While reducing component portability will make the component harder to use, it also reduces any associated intellectual property leakage as less component code is required to determine its run-time context.

> *We propose to sacrifice portability by embracing context-dependence.*
>
> *We propose the run-time context be provided to the supplier at design-time so that each component may be designed for its intended run-time context, strengthening its context-dependency.*
>
> *We further propose to eliminate component interfaces altogether by allowing suppliers to co-operate over their design, thus rendering the component completely context-dependent.*

While these proposals appear to make a component design more difficult, they do provide the substantial simplification of component integration necessary in order to halt exposure of intellectual property.

Reducing re-usability

While component re-usability should be limited in order to reduce intellectual property leakage, it should not impede in any way the interchangeability of the components, as many suppliers competing for a specific component business is vital for industry progression.

> *We propose to harness strong context-dependency to render uneconomical any prospects for component re-use.*
>
> *And finally, we propose to eliminate all glue code and even the concept of a component altogether by synthesising components on-the-fly.*
>
> ***Thus, we propose to replace the context-independent component with a context-dependent fragment.***

Integration via concatenation can only be supported by on-the-fly synthesis of fragments sourced from co-operating suppliers. This co-operation results in strong context-dependency, which is harnessed to stymie reuse, as shown in Figure 1.

With these changes to integration, portability and re-use, the supplier effectively builds the fragment into the client's project, thereby satisfying the objective of 'component delivery.' In effect, the client receives the benefit of the fragment without any burden of integration and the supplier effectively delivers the fragment without the risk of proprietary intellectual property leakage. Finally, with the removal of the component's additional context code and interface, their associated run-time performance penalty is also recovered.

Design-domain

With these changes to integration, portability and re-usability, an industrialised 'component' supplier can contribute to a design without resorting to writing code.

The industrialised supplier can be viewed as making a *pure* 'design-contribution,' as each contribution is an assemblage of

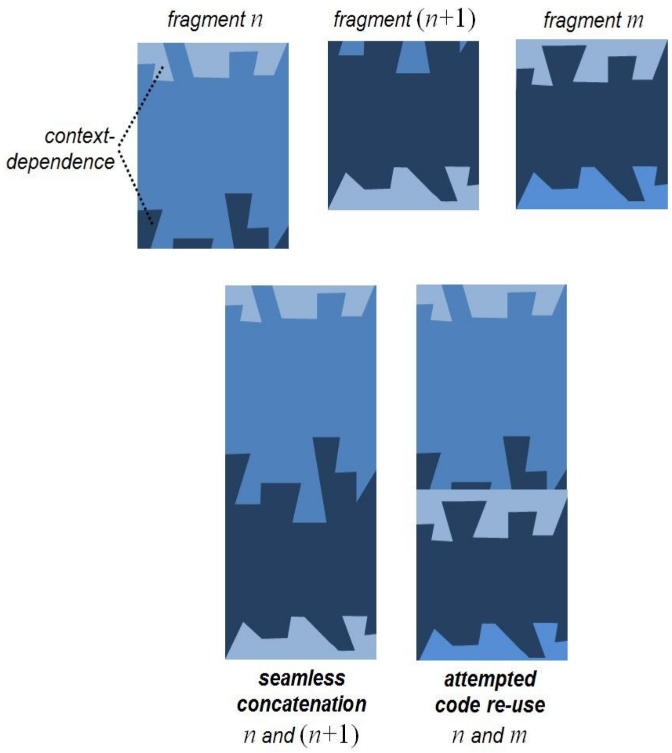

Figure 1 – (a) fragments provided by agents, and (b) attempted code reuse with incompatible fragments

smaller design-contributions from its suppliers.

Such a supplier can be viewed as operating wholly in a 'design-domain.'

> *We contend the decoupling of the design from code effectively shifts a developer from the incumbent code-centric code-domain to a 'design-domain.'*

This shift to the design-domain effectively *reverses responsibilities* that form part of the incumbent relationship between client and supplier.

Reversing design responsibility

In the code-domain, component integration and glue code design are the responsibility of the client, a situation that is reversed in the design-domain. This reversal effectively shifts the *burden* of integration to the supplier, reducing the client's integration responsibility to concatenation, a task so basic that no supplier assistance (i.e. intellectual property) is required. To facilitate this, the supplier must be granted access to a small portion of the client's project.

> *Instead of a supplier delivering a fragment to the client, we propose the client delivers a metaphorical construction-site to the supplier (into which the supplier builds their fragment), in keeping with the reversal of design responsibility.*

Now, the rigid code-domain component gives way to a flexible fragment that is synthesised on-the-fly for the client, and custom-designed according to the client's requirements. During fragment synthesis, the client is now afforded an opportunity to tailor the fragments, thus incorporating some of what would have been glue code, while directing suppliers to co-operate to determine their mutual interfaces incorporates even more. Thus, the role of the client is merely to provide requirements, co-ordinate suppliers and concatenate construction-sites in order to arrive at a workable design.

Reversing legal responsibility

The contract typically drawn up between a client and incumbent software developer ensures that the software product belongs to the *client*. Surprisingly, "*if the software developer re-uses a component of one client's product in a new product for a different client, this essentially constitutes a violation of the first client's copyright*" (Schach 2008). It appears the legal underpinnings of code-domain software development also undermine the commercial viability of a supplier, who is obligated by law to relinquish intellectual property rights to the client. By decoupling the design from the code, the expression of work (the code) is irrevocably separated from the ideas underlying the work (the design). In so doing, the supplier retains proprietorship over the design process while still delivering a code fragment.

> *Instead of protecting the legal rights of the client at the expense of the supplier, both client and supplier are equally protected.*

Thus, the debate over ownership is now moot, and no legal encumbrances or good-will on the part of the client are needed, as the supplier simply does not deliver their intellectual property to the client with the product.

Reversing requirements responsibility

> "*The hardest single part of building a software system is deciding precisely what to build [...] For the truth is, the clients do not know what they want.*"
>
> Brooks Jr. (1987)

In the code-domain, a software project typically begins with the establishment of a Software Requirements Specification, a document that requires input from both client and supplier. During this phase of the project, the supplier is responsible for gleaning the necessary information from the client (who often has only a vague idea of what they desire). This can lead to gaps in communication, which, coupled with revised requirements and changes requested during the course of the project, can cause subsequent delays and development errors.

> *Instead of the client delivering requirements to the supplier, we propose the supplier delivers degrees-of-freedom to the client, in keeping with the reversal of design responsibility.*

Industrialised software engineering reverses the responsibility for requirements specification so that now the supplier presents the client with an array of feasible and clearly defined degrees-of-freedom from which the client must express their requirements.

This process is already standard practice in other legitimate industries, where components not supported by the industry are simply not available. Experienced designers are familiar with the 'degrees-of-freedom' offered by the suppliers and will not produce a design for which parts are not supported, often upgrading to a higher specification or over-designing so as to arrive at a constructable design.

The shift to the design-domain successfully removes legal encumbrances, demarcates design responsibilities and clarifies the expression of requirements. Most importantly, this shift guarantees intellectual property protection, thereby restoring the most vital basis of industrialisation – specialisation.

Peer-to-peer software engineering system

By harnessing the industrial mechanism of mass-production through standardisation, interchangeability and reductionism, the ad-hoc generalist approach to software development can give way to the disciplined system sought in 1968 – Software Engineering. The opportunity to comprehensively overhaul software development uniquely positions this emerging industry to engineer its own revolution by cherry-picking proven methods from legitimate industries which have had the time to refine and mature. Now, rather than industrialising at a national scale and over decades or centuries, this industrial revolution, being unbound by any physical laws or constraints, can occur at a global scale and in Internet-time.

Supply-chain of agents

At the core of this emerging software industry is a layered supply-chain comprising a decentralised, peer-to-peer network of software engineers. In keeping with the shift to the design-domain, a software engineer is expected to provide a *design-contribution* – code is no longer their specific responsibility. Instead, engineers from each layer of the supply-chain contribute to the overall design until an executable coalesces as if by magic from their combined efforts. Such coalescence evokes the phenomenon of emergence. Indeed, when code is no longer the responsibility of a single entity in the peer-to-peer network, yet code is collectively produced by such a network, the entire process can be defined as *emergent coding*.

Agents

An industrialised system will require many such design-contributors, known as *agents*, each a specialist in their own right. When contracting an agent, the client is presented with the agent's degrees-of-freedom, each requirement of which the client is obligated to satisfy in order for the agent to render their design-contribution. Once all requirements have been acquired, the agent is in a position to sub-contract suppliers according to a combination of these requirements and the agent's own proprietary knowledge. (The agent's organisation of suppliers plays a key role in the delivery of the final executable.) These suppliers then present their own degrees-of-freedom to the agent, and so on. This process recursively descends through the layers, with each new iteration of suppliers providing a new level of detail to the design whilst stripping away a layer of complexity. In this way, design-contributions can be *globally complex yet locally manageable*, since each agent can rely on its suppliers to manage their complexity.

This process should not be confused with top-down design which is managed in the code-domain on a *macro level* with the large-scale overview in mind (an approach that favours the generalist). In this industrialised system, reductionism ensures the design framework is rendered on a *micro level* by specialist agents, who are concerned only with completing the task for which they have been contracted and do not require (nor desire) any information about the overall software design. In fact, since the selection and sequencing of suppliers actually embodies the intellectual property of the specialist, any attempt to map the entire design framework would necessitate inappropriate access to each agent's intellectual property. Thus, the success of this design paradigm is reliant upon each agent being able to deliver their design-contribution without any need for global contextualisation.

Construction-site

Of course, the question remains; if software engineers now operate wholly in the design-domain, how does the final code executable materialise?

By shifting industry focus to the design-domain, a software engineer is relieved of the responsibility of *translating* the design into code, arguably one of the more difficult issues faced by code-domain software developers. In fact, as the design increases in complexity, code-domain methods see the difficulties in translating design into code increase exponentially, imposing limits on the scale of software that can be reliably delivered.

In the design-domain, code construction does not take place until all complexity and context has been dispersed, a point at which construction becomes as simple as *placing bytes into what becomes the native binary executable*.

The binary executable coalesces in a design construct called a scaffold, using an in-built and entirely automated protocol known as *construction-site*. As the design proceeds from principal client to 'byte' agents, an elaborate network of ordered connections between agents is strategically established as *part of their design-contributions*. These connections, in their entirety, form a design scaffold which is rendered at a local level and remains *unseen at a global level.*

With each subsequently contracted agent, the overall context of the design becomes more dispersed so that the context provided to each individual agent becomes proportionally simpler. This contextual dispersion continues with each extension to the scaffold until the last vestiges of context have been *removed*. At this point, no further extension is necessary, as byte agents, using the construction-site protocol, simply place bytes into what will eventually become the resulting binary executable. In fact, once an agent has completed their design-contribution, they may be remunerated, with the system *completing the fragment integration and delivery at a later stage on their behalf.*

The construction-site protocol consists of three stages;

1. request for space (code/data),
2. address assignation (code/data), and
3. delivery (code/data) as shown in Figure 2.

Once the design has reached the byte agents and the scaffolding is complete, a request for space is automatically returned to the preceding scaffold intersections (client) where these requests for space are amalgamated and returned to the previous scaffold intersections and so on up the scaffold.

Once the final amalgamated request for space reaches the pinnacle of the scaffold, the start address for the subsequent amalgamated requests for space can be computed (according to each client's careful and strategic ordering of its scaffold connections). These addresses are then passed down the scaffold, and new addresses are automatically computed in a similar fashion at each scaffold intersection.

When the propagated addresses reach the byte layer, byte agents can complete their design-contribution, and fill their requested spaces with bytes, forming the *smallest fragments.* At each intersection, guided by the agents' ordered connections, these fragments are concatenated and/or passed upwards to form larger fragments until the largest fragment – the final binary executable – reaches the principal client. Each intersection's

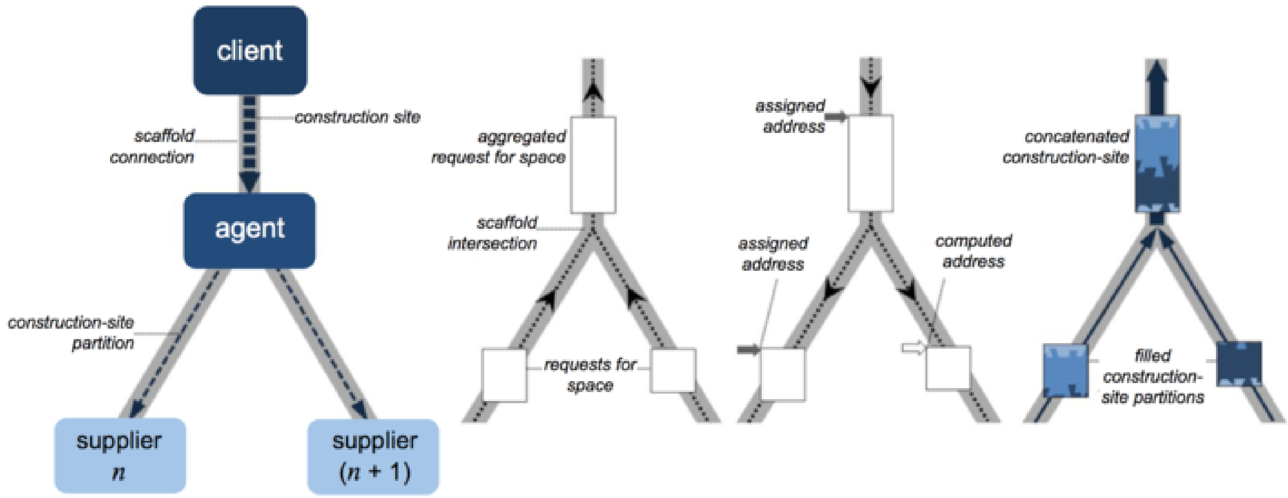

Figure 2 – Construction-site protocol

amalgamation of requests, computing of addresses and concatenation of fragments, serve to protect agents from exposing the number and arrangement of their suppliers.

Metaphorically speaking, the construction-site protocol can be viewed as a physical construct in which the final binary executable is erected, where the scaffolding acts as its temporary structural support. At the commencement of a software project, the principal client is in possession of an 'empty' construction-site. After sub-contracting agents and establishing the first series of scaffold connections, the principal client divides and distributes the figurative construction-site to each of these agents along scaffold supports.

This process occurs in fractal-like fashion until the partitioned construction-site is scattered across numerous byte agents, ready for placement of code/data bytes. Once filled with bytes, these construction-sites are aggregated using simple concatenation at scaffold intersections. The scaffold then retracts and larger construction-site portions are concatenated upwards until the intact construction-site arrives at the principal client. In this way, the scaffold is effectively dismantled to reveal the final product, ready for acceptance testing by the principal client.

Agent communications

A software engineer's intellectual property is now cast in the agent that delivers their design-contribution, and as this intellectual property is the livelihood of the engineer, it should be carefully protected. *A correctly operating agent will not leak intellectual property.*

Each degree-of-freedom that the agent presents to the client will either be resolved directly by the client or will require co-operation with another peer (or peers) as authorised by the client. This co-operation may require a negotiation with the peer (or peers) to arrive at an agreement, thereby resolving the mutual requirement.

Typically, an agent's design-contribution will be an assemblage of design-contributions from other suppliers. The agent selects, orders and ultimately remunerates suppliers according to requirements acquired directly from (or via) the client, in order to complete the contract and receive *their own* remuneration. The selected suppliers will present their degrees-of-freedom, which the agent is obligated to fulfil, for awarded contracts to be acceptable. The requirements delivered to the supplier (in accordance with their degrees-of-freedom) are derived from a combination of the agent's own degrees-of-freedom and internal knowledge. An agent may choose to resolve a supplier's requirement directly or by placing the supplier in contact with one or more of its peers so that they may co-operatively resolve the requirement.

As an agent will receive access to the global construction-site via the scaffold connection, it is obligated to further partition this access for distribution amongst suppliers. The order of connections can be crucial, as the agent may be indirectly assigning the order in which the supplier's fragments will be concatenated upon delivery. Surprisingly, the code fragment returned through the agent could quite feasibly bear *little relation to the design-contribution for which it was contracted to provide.*

Committees

Committees are an in-built protocol and the principal form of communication between agents in the supply-chain. *The purpose of a committee is to resolve a requirement.* Each degree-of-freedom presented to the agent by a supplier takes the form of a representative tasked with policing that degree-of-freedom. It is the agent's responsibility to appoint all awaiting representatives to appropriate committees. If an agent can satisfy a degree-of-freedom directly, a simple two-member committee (with self- and supplier-representatives only) is established to resolve the corresponding requirement. Alternatively, the agent may be presented with a degree-of-freedom that requires negotiation with a fellow supplier representative to resolve the requirement. In this case, the agent simply appoints the supplier and their relevant peer(s) to the same committee. If the agent requires no representation on a committee it has established, it is implied that the agent will accept the unseen outcome, thus leaving the committee to resolve a requirement in the best interests of its representatives. Similarly, the agent resolves its own degrees-of-freedom by presenting self-representatives to *its own* client.

Negotiations

The principal form of co-operation between representatives within a committee is a negotiation. Negotiations may be as simple or as complex as determined by the respective objectives and capabilities of the committee representatives. The negotiation can be viewed as a distributed constraint solver, while a degree-of-freedom can be viewed as a negotiating position. Once a negotiation yields an agreement, the agents share the terms that attend the agreement, which may take the form of additional sub-committee representatives awaiting appointment. With an agreement in place, agents are free to proceed with their design, safe in the knowledge that other representatives will abide by the agreement.

A feasible business

For the first time, engineers can expect payment for every contract their agent wins and successfully completes, secure in the knowledge that operating in the design-domain not only protects their intellectual property but preserves opportunities for repeat business. It follows then that engaging the supply-chain itself to construct software will also involve payment, which serves to indirectly pay each participating agent (in much the same way that purchasing an auto-mobile will indirectly pay each automotive part supplier). Engineers are also free to price their agent at will, assisted by the regular price discovery mechanisms afforded by a competitive market.

Agent directory

To become operational, each agent must register with the *global agent directory* and assume an appropriate classification such that potential clients may peruse and evaluate suppliers from the advertised selection available. By maintaining a range of metrics, the directory also serves to provide information about the reputation of agents. Any unfulfilled contracts due to equipment failures, insufficient resources or the like, may tarnish an agent's reputation, and have repercussions such as loss of market-share.

Transition

In order to begin operating in a design-domain, a minimal working supply-chain is necessary – *a prime supply chain.*

> *A prime chain of agents must encapsulate design expertise of sufficient sophistication so as to construct agent-software.*
>
> *Transition from the code-domain to a design-domain requires a prime chain capable of synthesising all agents it contains.*

A causality dilemma arises – how can a prime chain be constructed in the absence of any existing prime chain? Without an already established prime chain, there is little alternative but to develop each and every agent of the prime chain using legacy code-domain methods. When this 'bootstrap' supply-chain is operational, each engineer is free to rebuild their agent-software by engaging the bootstrap supply-chain with their same agent requirements. The resulting set of agents form a prime chain that is *operationally equivalent* to the bootstrap supply-chain but built using emergent coding methods.

The first wave of *agent engineers* to join the supply-chain will engage the prime chain (by contracting top-level prime agents) to construct their agent-software, and then peer *with* the supply-chain. As more agents peer with the supply-chain, the scope and quality of software that can be engineered is widened (beyond that of agent-software). Engineers can engage other top-level agents from the expanded supply-chain to build other types of software, much the same way an agent-engineer engages prime agents to build agent-software. New agents, as well as competition amongst existing agents, will enhance the supply-chain's overall capabilities (including that of the prime chain). Engineers can rebuild their software (including agent-software) to field any new features offered by the enhanced chain, fuelling a *geometric growth in capability.*

Crossing over to the design-domain requires significant adjustments for a prospective engineer. Fortunately, protocols such as construction-site, committee and negotiation are automatically built into agent-software by the prime chain. These globally powerful yet locally simple protocols are easy to master. In addition, reductionism narrows the scope of concern, which reinstates the usefulness of simple conceptual models such as flow charts and the like. Finally, an engineer can direct their agent to effortlessly deliver complex design-contributions by contracting emergently powerful (and eager) suppliers. Thus, any prospective engineer can quickly and simply cast their innovation into a new agent by engaging the prime chain. By subsequently peering with the global supply-chain, the engineer not only enjoys the rewards afforded by their newly cast innovation, but also contributes to the geometric growth of that supply-chain.

Essence and accidents of Software Engineering

When examining Software Engineering, Brooks Jr. divided software engineering into essence – the difficulties inherent in the nature of software – and accidents – those difficulties that attend its production. In 1987, Brooks Jr. observed that most improvements to software development continued to address the accidental difficulties of software rather than the essential, an observation that remains valid decades later.

> *We contend that the proposed industrialised system addresses the essence of software.*

Essence

Any claim to a method that addresses the essence of software can be assessed by examining the method's impact on the inherent properties of complexity, conformity, changeability, and invisibility.

Complexity

Brooks Jr. makes a compelling case that "*complexities are the essence*" and abstracting them away often abstracts away this essence.

> *We observe that the offered method thrives on complexity and enjoys powerful mechanisms for acquiring it, utilising it, extending it and being remunerated for it.*

When an agent provides a design-contribution in return for remuneration, the client enjoys a delightful ignorance of the agent's process. The client is also shielded from the complexity that is harnessed during the process, including the hierarchy of suppliers *unseen behind the agent.* In effect, the client enjoys the services of a powerful agent without managing the equivalent complexity of the agent's process. Similarly, the agent enjoys the services of their own suppliers without managing the complexity of their processes and so on. Remarkably, the agent inherits all the functionality and capability of their suppliers without inheriting any of the associated difficulty, demonstrating *emergent complexity*.

In a literal sense, an agent is able to reliably deliver a custom-designed fragment totalling many millions of bytes in size and many millions of function-points in capability while still enjoying a process that is locally simple – this agent merely has powerful suppliers.

The complexity of team member communication dissolves in a similar way. Indeed, "team members" are actually only a necessary feature of the generalist approach to software development. In an industrialised system, there are no "team members" much the same way that all the suppliers to the automotive industry are not considered team members. The "team members" give way to a global pool of agents dedicated to some aspect of software engineering rather than a particular project, and as such are now dedicated to potentially *all* software projects. Agents also co-operate with peer specialists (who are experienced problem-solvers within their field), where communication facilitates design decisions. It is clear that while there is substantial communication involved in an industrialised system, all communications are between experts in their respective field and are therefore experienced and adept at communicating within that field.

According to Brooks Jr., when a code-domain software project is scaled up, it is not merely a repetition of the same elements in larger size. Rather, it is an increase in the number of different elements, with complexity of the whole increasing much more than linearly. When software is constructed using an industrialised system, large numbers of contracts are precisely let for design-contributions during the course of a project. Larger projects have proportionally more contracts. The project is never conceptualised at a global level, rather each agent manages their contribution at a local level. With complexity now fixed at a localised contract level, complexity can only increase in proportion to the number of contracts.

> *We contend that project complexity increases linearly with number of contracts, while remaining fixed at a contract level.*

Conformity

Brooks Jr. argues that much of the complexity to be mastered is arbitrary complexity that differs from interface to interface and that it "*cannot be simplified out*" by any redesign of the software alone. In this regard, (putting aside for one moment that there are now no interfaces) we observe that in order for a client to utilise a code-domain component, the client must comply with the interface as *dictated* by the supplier. In the design-domain, interfaces are now defined through co-operation between peers, assisted with the full might of industrialisation. In addition, as all software is mass-produced but custom-designed for each application, conformity is a natural feature of the design-domain.

Changeability

As fielded software is "*embedded in a cultural matrix of applications, users, laws and machine vehicles*" it is constantly subject to pressures for change. Manufactured things are infrequently changed after manufacture, certainly much less frequently than modifications to fielded code-domain software. However, fielded design-domain software is far less accessible, as the software is only known from its initial requirements. Thus, any modifications to fielded software will therefore require a rebuild, incurring the associated cost to modify. Thus, the "*high costs of change, understood by all*" will apply to fielded design-domain systems to "*dampen the whim of the changers*" (Brooks Jr. 1987).

Invisibility

Brooks Jr. argues that software is invisible and unvisualisable and thus geometric abstractions, while powerful, are largely unavailable. Fortunately, in the design-domain, the scope of concern has been considerably narrowed. Thus, there is little need to understand the whole to the same detail as before. Rather, the system relies upon an agent delivering a design-contribution without any need for global contextualisation. Specialisation therefore provides ample opportunity to use geometric abstractions in the quest for improvements to agent processes, operating as they do at a local level, where the problem-space is small and manageable.

Accidents

Accidental difficulties arise during the representation of the conceptual construct of a software system, whereas the essence of software lies within the conceptual construct itself. An industrialised system will naturally provide its own accidental difficulties such as constructability, accountability, change-management and competition among others. These new accidental difficulties open rich workings for any industry striving for order-of-magnitude improvements in productivity, reliability and simplicity.

Constructability

Code-domain methods of software development have enjoyed a smooth requirements specification space. One of the few benefits of the code-domain is that, given enough time and resources, a developer is able to fashion software to satisfy *any* reasonable requirement. However, the design-domain requires a more disciplined and restrictive approach to requirements specification, one that is limited to what 'parts' are available. When a situation arises in which a requirement is outside a degree-of-freedom advertised by any available supplier, the choice presented is one of either compromising on the design (in order to remain within the degree-of-freedom) or bringing a new 'part' into existence (by creating a new supplier or incentivising an existing supplier to expand their degrees-of-freedom).

> *We propose that the requirements specification space now be confined by industry-supported degrees-of-freedom.*

In the design-domain, we can no longer build any application or satisfy any requirement for which the industry is currently deficient in supporting. Fortunately, the law of supply and demand ensures that deficiencies are merely viewed as market opportunities whose growth will inevitably attract the enterprising engineer.

Accountability

With the de-emphasis on code structure and readability, how then are we to find and fix 'bugs'? In the design-domain, there are no 'bugs', only requirements non-conformance. In contrast with the code-domain, where defects are sought on a global level and with global visibility, in the design-domain, non-conformance is identified at a local level by the relevant specialist engineer. When a software program fails its acceptance testing, the principal engineer simply identifies which requirement is not satisfied, and therefore which supplier is in breach of contract, a feat that is only possible now that requirements are discretised by degrees-of-freedom. When notified of their non-conformance (and the behaviour that caused the fault in particular), this supplier then examines their job history to isolate the contract in breach and inspects their design to determine whether the fault was due to their internal knowledge, or a supplier of their own. If it is the latter, the process of recursive non-conformance notification continues.

> *We observe that accountability, while largely absent from the code-domain, strongly manifests in the design-domain.*

An astute engineer will realise that, while their agent inherits the powerful design capabilities of its suppliers, it also inherits their reputations. When an agent's reputation is directly tied to the reputations of its suppliers, the astute engineer will naturally choose these suppliers wisely.

Change-management

As a consequence of operating in the design-domain, code is now structured for performance rather than maintainability. As such, the change-management process now consists of the far more readable and appropriate maintenance-of-requirements. In the code-domain, adding a feature during late stages of the design has always proven problematic, as the code-base is typically not provisioned to incorporate the feature. Further, in the interests of cost control, any new feature is likely to be added in an improvised manner rather than risk an extensive redesign to properly integrate the feature. "*History shows that very few late-stage additions are required before the code base transforms from the familiar to a veritable monster of missed schedules, blown budgets and flawed products*" (Brooks Jr. 1987).

By contrast, engineers in the design-domain can summarily add new requirements (as long as they stay within the bounds of industry-supported degrees-of-freedom) which, with rebuilding, seamlessly incorporate the new features. Any number of late-stage (or post-deployment) additions can be added in this way with little risk of the project becoming unmanageable.

Competition

With intellectual property protection, an agent is denied visibility of a competitor's process. A prospective competitor, while entitled to field a competing agent, must therefore develop their own intellectual property. Where there is competition, the better design tends to win-out, rewarding innovation and forcing improvements in order to retain or regain market-share. Because of the fractal-like nature of the supply-chain, where 'parts' are made up of 'sub-parts', agent competition at a 'part'-level is amplified by supplier competition acting at a 'sub-part'-level and so on.

> *We observe that competition, while largely absent from the code-domain, intensely manifests in the design-domain.*

The size of the competing pool also has significant bearing on the intensity of competition within the pool. With the advent of technologies such as the Internet, with its cross-border reach, and Bitcoin, which provides border-less and instant wealth transfer, *a global competing pool* becomes practical. With careful design of the industrialised system, competition can be focused on key pressure points of software development. If agent performance metrics such as speed, cost, performance, resource usage and rate of non-conformance are advertised to prospective clients, then competition will inevitably be driven by those metrics.

The variance in quality of agent performance is transparent in the design-domain and is to be celebrated – after all, without excessive cost, non-conformances, waste, and the like, there is little for competition to effect. It must be accepted that a supply-chain will not contain agents uniform in calibre, and that there will be varying degrees of quality in a given project. (It would be arrogant to assume that every part of an auto-mobile has been designed to the same standard.)

> *Intense competition delivers a software supply-chain with a fast path for improvement.*

An industrialised system takes software that is largely immune to competitive pressure and creates perhaps the most competitive environment *ever* devised. This acute competition arises for two reasons, both almost unique to software; software's intangibility and the global reach of the Internet. The irony it appears, is that despite being bypassed by the industrial revolution for so long, software development is uniquely suited to an industrialised system.

Emerging Software Industry

McIlroy's industrialisation, with its magnificent ability to manage complexity, is precisely the disruptive innovation that the software industry has long sought. To consider the proposed method a turning point for the emergence of a software *industry*, we examine the extent to which the method complies with industrialisation's reductionism and mass-production, through the principles of standardisation and interchangeability.

Reductionism

In order for a system as complex as a software supply-chain to emerge, conditions such as viability and scaling of its constituent parts must be satisfied. We note that scaling components using code-domain methods poses substantial design challenges. Logically, the larger the component, the more it must interact with its host application. When designing an interface for ease-of-use, the developer finds the magnitude of the required interaction often works against interface simplicity. Furthermore, the client must contend with an ever larger and therefore more complex interface, and must design proportionally more glue code for proper component operation. Thus, code-domain scaling issues limit the size and practical value of any software supply-chain. We also contend that it is virtually impossible to build a viable business around code-domain methods due to intellectual property leakage. With the challenges of component scaling limiting a supply-chain's *practical* value and the inability to specialise limiting a supply-chain's *economic* value, any application of reductionism is not only correspondingly limited, it is functionally inhibited. The design-domain is not limited by any such scaling or economic viability issues. Agents are free to fashion arbitrarily sized fragments through the use of sub-contracting, effectively inheriting a good portion of design and integration effort from these suppliers. Additionally, by removing component interfaces and instead allowing seamless integration of fragments, their effective interface scales with the size of the fragment. The principle of reductionism is clearly satisfied, as scaling of the software entity now becomes "*merely a repetition of the same elements in larger size*" (Brooks Jr. 1987).

Mass-production

In a client-supplier relationship, it is the supplier who has innate knowledge of the possible scope of requirements that exist within their specialist purview, rather than the client. Yet surprisingly, in the code-domain, it is the client who is primarily responsible for dictating these requirements. In the design-domain, this responsibility is now shifted to the supplier, thereby ensuring requirements are *captured* by the expert. This notion is taken to the extreme, a point at which all ambiguity can be eliminated, by introducing *quantised* degrees-of-freedom. Now, whenever an agent is contracted, its client is presented with a finite suite of degrees-of-freedom, each of which the client must satisfy in order to enjoy the commissioned design-contribution. This concept of quantised degrees-of-freedom not only makes the process of requirements-capturing a deterministic one, but more importantly, it opens up the opportunity for automation. If each agent in the supply-chain advertises their degrees-of-freedom in advance, an engineer may now effect automation in all of their agent's operations, as committee appointments and negotiations are driven by internal knowledge (which can also be automated).

It is indeed fortunate that the design-domain affords unprecedented automation opportunities for the simple reason that each fragment is to be synthesised for the client on-the-fly. In fact, when the design of a single software program alone can require (in total) many millions of contracts, it is almost impossible to consider this peer-to-peer technology *without* the prospect of automation.

Standardisation

The proposed software engineering system is a service-oriented one, where each agent makes a design-contribution. Design-contributions in the order of millions can be expected for a single design project. Without some standard outlining agent interactions, the cumulative collaboration required for just a single project would be prohibitively time- and cost-intensive. As each agent's degrees-of-freedom are advertised in a shared directory, the logical application of standardisation would dictate that these degrees-of-freedom adhere to a formalised set of standards.

When contracted, each degree-of-freedom presented by an agent to the client will be in the form of a representative awaiting appointment to a relevant committee. Once appointed, these representatives are free to compatibly arrive at an agreement beneficial to all representatives, the terms of which may result in further representatives awaiting appointment to sub-committees. By formalising committee and sub-committee *types*, this engineering system can operate within a framework of automated services where each contribution is seamlessly integrated *without manual intervention* during a live project.

Interchangeability

The offered software engineering system, while not enforcing interchangeability, nevertheless expects the application of interchangeability to emerge due to market pressures. As in standardisation, an easy way to reduce the client's burden in switching to a new supplier is for the supplier to make their product interchangeable. This is achieved by adopting the same requirements degrees-of-freedom as the competitor. What will quickly emerge are pre-defined *suites* of degrees-of-freedom. These suites can be advertised under classifications in the directory, along with a comprehensive list of compliant agents. As each agent is automated to present these degrees-of-freedom when contracted, an engineer can direct (in advance) their own agent to deal with the suite not the supplier. After the agent has been built and is active in the supply-chain, the engineer may substitute a supplier for another from the suite list. An engineer may wish to interchange suppliers in this manner in order to take advantage of the natural reductions in cost and increase in quality that inevitably emerge within a competitive system.

Each agent that exists in the supply-chain is a simple expert system formed by capturing and rendering an engineer's specialist knowledge as a design service of value. These agents are capable of seamlessly interacting to design and construct highly complex software programs. Through the introduction of standardisation and interchangeability, all agents enjoy high levels of automation which instantly elevates the design-domain to a status only seen upon maturation of industrialisation – *mass-customisation*.

In contrast with preceding industries, whose agility in mass-production was hard-earned through evolved maturity, the software industry can *begin* its industrial revolution with near perfect agility, as every 'component' is mass-produced yet custom-designed.

Conclusion

The software industry is not industrialised. Not surprisingly, industrialisation has been touted as the solution to the software crisis. Proposed is an innovation for protecting intellectual property, the lack of which denies industrialisation its most vital basis – specialisation. Thus, a revolution may be upon us – *a software industrial revolution* – mobilised by the intangibility of software and fuelled by its software factories synthesising its software factories. With this revolution, a new breed of engineer can join the ranks of other legitimate engineering disciplines, ply their skills, and peer with a globally connected supply-chain, founding their own Silicon Valley, a virtual valley, a Code Valley.

Acknowledgements

My chief engineer Jesse Gileppa and his engineering team Dao Zhou and Julie Lovisa for their dedication, loyalty, and determination in driving the design-domain into existence.

My wife Bernadette Lovisa for engineering simply one of our best assets.

References

Brooks, Jr., F.P. (1987). "No silver bullet: essence and accidents of software engineering," *IEEE Computer*, 20(4), pp. 10-19.

Gibbs, W.W. (1994). "Software's Chronic Crisis." *Scientific American*, September 1994, pp. 86-95.

McCloskey, D. (2004). Review of *The Cambridge Economic History of Modern Britain*, edited by Roderick Floud and Paul Johnson, Cambridge University Press.

McIlroy, D. (1968). "Mass-produced software components," *Software Engineering: Report of a conference sponsored by NATO Science Committee*, Garmisch, Germany, 7-11 Oct., Scientific Affairs Division, NATO, p. 79.

Schach, S. (2008). *Object-oriented Software Engineering*, McGraw-Hill Higher Education, New York.