The Diffusions of Components

Richard Veryard, November 2000

Status

Paper for invited talk at the IFIP Working Group 8.6 Working Conference on Diffusing Software Product and Process Innovations, to be held in Banff, Canada, April 7-10, 2001.

Abstract

This paper takes an ecological perspective on diffusion factors within the software component market. It analyses the characteristics of software components that are favourable to diffusion, and poses a radical critique of traditional notions of software requirements and software quality. It also suggests a strategic view of software components and other technological artefacts as evolutionary envelopes rather than fixed collections of properties.

Introduction

Diffusions

This paper is about (the study of) technology diffusing across a landscape. The landscape is a complex ecosystem, in which the technology we have chosen to study is competing for attention and resources with a range of other similar and diverse technologies. (Perhaps we should study not Diffusion but Diffusions – to emphasize the complexity and diversity of the diffusion phenomenon itself.) The ecosystem may involve a community of intelligent agents – which may be human or artificial, individual or collective – with a structure of relationships including delegation as well as mutual obligations and responsibilities.

We typically study the diffusions of a specific technological device or artefact – or perhaps a class of similar devices. In this paper I'm going to talk about software components. My reason for choosing to restrict myself to this class of artefact is that they present themselves as having certain properties that will help to simplify our discussion.

The fundamental notion of software componentry is the separation of a description of the **services** offered by a component from the description of the internal mechanisms (which software engineers sometimes call **implementation**) by which these services are delivered. (Note that the word "implementation" literally means that the job is finished. A software engineer may indeed think that the job is finished when the program code is written – and perhaps tested – but there are other stakeholders who regard this as only the beginning of a much larger and more difficult job.)

Components and Commodity

I shall start with an account of technology derived from Borgmann, which I characterize thus: All technology aspires to the status of **commodity**. As I see it, this means at least two related things: firstly that the technology is usable – and used – to the greatest possible extent; and secondly the technology is packaged (encapsulated) in some set of products and services. (The notion of commodity also has some specific implications in economics, which we won't go into here.) In Borgmann's account, the drive to greater commodity can be seen in terms of ever-increasing **availability** of some technologically mediated benefits: easy and safe, wherever and whenever you want it.

Within software engineering, the logic of commodity is found most strongly in the notion of **component-based development**. Software components are expected to be reusable – and this reuse is supposed to be a good way of achieving economies of scale and increased productivity in the development of software systems. The expectation of software reuse is what I call a **design mandate** – it represents a complex amalgam of aspiration, prediction, justification, imperative and quality judgement, as shown in Table 1.

Aspiration	Software designers want their components to be widely used. This desire may be reinforced by extrinsic motivation, such as financial reward.
Prediction	Given certain conditions, certain levels of software reuse and productivity are expected.
Justification	Component-based software engineering, together with any infrastructure needed to enable the mandated level of software reuse, is cost-justified against the predicted productivity benefits.
Imperative	Designers are required to produce software components that have the requisite characteristics for reuse.
Quality judgement	A "good" component is one with the potential for wide reuse.

Table 1: Elements of the Design Mandate

Design methodologies are typically based upon several such design mandates. The epistemological and sociological status of these mandates is far from straightforward – as I have argued elsewhere, the software engineering industry is awash with claims that are difficult if not impossible to prove convincingly [Veryard 2001].

Surely then, within this engineering discourse, there should be a keen desire to understand which characteristics of software components and other artefacts are likely to be associated with wide dissemination and use – particularly as there are many knowledge-based artefacts (such as methodologies) that claim to be predicated upon reuse. And yet there seems to be little general understanding of diffusion within the software engineering community. (There is some understanding, of course, but it is itself poorly diffused – for reasons that even the technology diffusion community has somehow failed to master.)

Software Success

Software engineering is a complicated game, with many players playing different roles. There are many stakeholders interested in the "success" of particular software artefacts within this game. In particular, many players wish to attach themselves to successful components, in one capacity or another. This entails a desire to predict and control software characteristics that might be related to success.

What is a successful component? What does success mean? One fairly obvious notion of success is in terms of diffusion and use – a successful component is one that is widely disseminated and used. This often brings financial and other rewards for the originators of the component – but of course success for the component doesn't guarantee success for the producer. A component may be given away in the hope of achieving some other advantage, or may be stolen by software pirates – the software may be on everyone's computer but the producer is penniless. Exactly the same principle applies to viruses and worms – a successful virus is one that infects millions, regardless of any consequences to the producer.

Unit of Adoption

Agent Adopts Device	• Granularity of the adopting agent • Granularity of the decision to adopt • Granularity of the adopted device

Table 2: Dimensions of Granularity

Perhaps the event of greatest interest to the student of technology adoption is the adoption decision. This occurs when an actor (which may be a person or community, or an agent or artefact to which some responsibility has been delegated) decides to adopts a device. A manager with sufficient authority may make this decision on behalf of a large community of users.

In many situations, the adoption decision is a joint one. A central planner makes a recommendation, which individual users are encouraged to accept, perhaps by an alteration in the cost-benefit equation for the individual user. Perhaps the central planner negotiates a price reduction, or absorbs some elements of the total cost of ownership.

This adoption decision may also be tentative or contingent. A person may decide to adopt a device on trial, perhaps for use in a pilot project. Further adoption decisions may be held off until more information is available.

This decision is made on the basis of some view of the potential utility of the device for the user(s), as compared with the expected total cost of ownership. The decision may also be dependent on the degree of confidence of these estimates, together with an assessment of any relevant risks.

This supposedly rational decision is highly sensitive to the way the overall system is conceptualized, as well as the costs, benefits and risks that are deemed relevant to the decision. An observer that takes a different view of these things may be critical of the decision that is made, and may regard it as a manifestation of a defective rationality. In particular, when a potential adopter declines to adopt something, which someone else (such as the vendor) thought he ought to have adopted, this is often characterized as **resistance**.

Note also that an adoption may be involuntary – and this is related to the granularity issue. You accept an email or email service, or download some software from the Internet – and find that you've also unwittingly accepted a virus. Or perhaps a colleague gets infected first, and then passes it around the company. There are thousands of software components on my computer – from cookies to DDL files – and many of them have been loaded automatically as a consequence of some other installation or interaction. I hope they're all benign – but I can never be sure.

Artefacts and Agency

In this piece, I'm talking about technological artefacts, including systems and components, as if they possessed intention and value. This is a fairly common trope, and many readers will pass it without comment. However, some readers might find this manner of speaking anthropomorphic or animistic: surely **things** can't have intentions – shouldn't we be careful to attach intentions and values only to specific **stakeholders** – that is, people or communities of people?

Let me say straightaway that I'm not unsympathetic to this objection. In my consultancy practice, I frequently encounter floating statements of intention and value (or belief or risk or cost/benefit), and I often find that it helps to anchor them by attaching them to specific stakeholders. So it might appear that I'm contradicting this practice here by allowing artefacts to have intentions and values. Surely the intentions and values really belong to the system owners – whoever they are.

But there's a problem here. Firstly, it isn't always obvious who the system owners actually are – and even when it is, we shouldn't be dependent on this identification for our analysis. After all, an artefact may continue to manifest certain patterns of intentionality, regardless of a formal transfer of ownership. Often there are many stakeholders, with conflicting and overlapping interests, and the behaviour of an artefact represents a complex balancing of these interests.

Furthermore, if we attribute intentionality to artefacts, we can then compare the intentionality of the artefact with the intentionality of its stakeholders. From a practitioner perspective, this is extremely useful. For example, we may be able to predict errant behaviour in unusual circumstances, without having to engineer or await test conditions.

Most importantly, we can talk about the artefacts in purely ecological terms, as if their owners were completely out of the picture. This is a very useful simplification – provided we don't forget that it's only a simplification and not the whole story. (The map is not the territory.)

Ecological Model

In this section, I propose an ecological model for understanding the characteristics of successful software components. More details of this model can be found in my book [Veryard 2000].

For a software component to be viable, it needs to be simultaneously viable in two separate but connected "ecosystems". It needs to be viable as a black box delivering some set of services to a community of users. And it needs to be viable as a white box, with some configuration of devices executing some set of mechanisms on some platform.

In the service ecosystem, it is services that are competing for survival. Service viability depends on three key principles.

Pleasure Principle	Value comes from achieving an appropriate level/balance of excitement and attention. Foreground components gain value if they are interesting and attention-absorbing; background components gain value if they are routine and require little or no attention.
Connectivity Principle	Value comes from the number of other users of the same service or component, within some domain.
Availability Principle (also known as the Martini principle: *Any time, any place, anywhere.*)	Between two otherwise equivalent services, the more available service will usually win over the less available. Some aspects of availability are as follows (depending on the nature of the service): Global 24-hour access. Instant response. Any hardware and software platform. Available in Arabic, Chinese, English, Hindi, Russian and Spanish. Easy to use. Low entry cost. Good support. Minimum learning curve. High reliability. Safe and secure. Low risk.

Table 3: Principles of Service Viability

In the device ecosystem, it is devices and mechanisms that are competing for survival. Device viability depends on four additional principles.

Energy Conservation Principle	Competitive survival depends on delivering the greatest quantity of service with the smallest amount of work. This is often called reuse; software reuse should be focused on achieving economies of scale in software, based on effective asset management and knowledge management. Energy conservation also entails an interest in the operational efficiency or performance of a component.
Consistency Principle	Getting the expected services (and their associated benefits) from a given configuration of devices. This in turn relies on an ability to predict and control the behaviour of components-in-use, including the emergent properties of large distributed systems. Covers reliability.
Flexibility Principle	The ability to easily substitute devices and reconfigure systems. Covers maintainability and portability.
Biodiversity Principle	The robustness, flexibility and evolution of the ecosystem depends on a reasonable heterogeneity of software and services.

Table 4: Principles of Device Viability

Note that this set of seven principles covers and extends the quality characteristics of software products identified in ISO 9126: functionality, maintainability, efficiency, usability, reliability and portability – with the possible exception of functionality. (We'll come back to that.)

If there are multiple ecosystems, where are the components? As a working hypothesis, I'm going to suggest the following answer. A successful component should have a place in each ecosystem. Just as a frog must be viable both in the pond and on the shore, so a successful component probably needs to be viable and meaningful in both ecosystems, and this in turn means that the component probably respects all seven ecological principles. Such a component is viable and meaningful, and is likely to survive and develop. I'm going to call this the **component viability hypothesis**. It leads to the following definition of component quality: **A viable component is one that respects all seven ecological principles.**

That doesn't mean that every component – or even every successful component – must have these characteristics to the ultimate degree. But a component that fails

to respect one of the principles is vulnerable to attack, and can be swept away by another component that is stronger in this characteristic.

The trend towards components is too recent to provide conclusive evidence for this hypothesis, but there are some early signs of its plausibility, as well as arguments from analogy.

Note that while some components may seem viable in isolation, other components may only be viable as a member of a kit or family or tribe of components. As in biology, the unit of viability may not be fixed. So sometimes we'll apply the ecological principles to individual components, and sometimes we'll apply them to groups of components. This may sound like an unnecessary complication, but it reflects what happens in real life – components are sometimes designed or evaluated in isolation, sometimes in groups.

Here's a well-known example. One of the factors that made Apple Macintosh successful was the common look and feel of Macintosh applications, although these applications were developed by independent teams in lots of different software companies. This consistency is a property of the tribe, not just of a single application. Thus the viability of a single Macintosh application is bound up with the viability of the whole tribe. That's ecology for you.

Here's another example. In the early years of laptop computers, travellers with laptops faced enormous difficulties if they wanted to connect these computers into hotel telephone systems. Although many hotels have now greatly improved the services available to the business traveller, and the telephone systems themselves are much more reliable, an increasing number of businessmen now use their own mobile telephones rather than the hotel telephone for such purposes. A few hotels may invest huge amounts in making provision for business travellers, but this investment is wasted if the businessmen don't bother using these services. And now that they have found a satisfactory alternative, they might not start using these services again until the majority of hotels offer them, and perhaps not even then.

Implications

Requirements

The biological approach to requirements engineering is radically different to the traditional approach, and is based on biological and ecological metaphors.

- First we identify an **ecosystem**, which may contain both human users and existing artefacts.
- Then we identify **services** that would be meaningful and viable in this ecosystem.
- Then we procure **devices** that enable the release and delivery of these services into the ecosystem.

Table 5: Ecological approach to software requirements

This may be contrasted with the traditional approach to software requirements, which I call solution-driven, which may be either specific to a single situation (Table 6) or generic across some domain (Table 7).

- First you identify a group of users who need a software solution for an identified business problem.
- Then you define the requirements on the software system. (For example, this may be specified as a set of use-cases.) These requirements may be based on a model of the business process, and are negotiated with the users.
- Then you design the software system as a set of interacting components.

Table 6: Specific solution-driven approach to software requirements

Of course, if you are trying to build generic components for multiple use, there may not be a specific business process to analyse, or even a specific software system to design. Furthermore, there may not be any specific users to negotiate requirements with. Undaunted by this, software engineers typically adopt the same approach but at a different level of abstraction. A **domain** is defined, which is a generic business process or generic area of automation. Many software artefacts are designed as generic solutions, including frameworks and platforms.

- First you identify a group of domain experts, who are supposed to stand proxy for a class of potential users.
- Then you define the requirements for the domain, in collaboration with the domain experts.
- Then you design a generic kit of interacting components, which will be usable for any system or business process that satisfies the generic domain description.
- Then you assemble systems from these components that satisfy the specific needs of particular users within the target area.
- Real business components need to be provided with staff, resources and infrastructure.

Table 7: Generic solution-driven approach to software requirements

The solution-driven approach seems to imply a division of labour: in the software industry, some engineers shall specialize in the creation of small lumps of functionality (called software components); while other engineers shall specialize in assembling these components to produce large lumps of functionality (known as software applications or systems).

The solution-driven approach assumes that it is meaningful to think about requirements in terms of a fixed lump of functionality or capability, delivered to a fixed community of users. In business, this is known as **the** business; in software, this is known as **the** software system or application. It also assumes that one person or team has design control over this lump. In business this is supposed to be the CEO and her direct reports; as for software, there are several possible job titles, including system architect.

The limitations of this approach emerge when we are faced with large open distributed dynamic networks of business and software. It is both a business imperative and a technological imperative for business organizations to connect their business processes into these networks. These networks lack central design authority or architectural control, and evolve organically. Overall functionality and structure may change unpredictably from one day to the next. Connecting to these networks raises a number of difficult management dilemmas, including control, security and stability.

Taking our cue from Kevin Kelly, these networks are “Out of Control”. Traditional engineering approaches are inadequate for operating effectively in this environment.

As Kelly has shown, biological and ecological metaphors seem to have more relevance than engineering metaphors.

Quality

Viability versus Quality

The ecological model focused on viability rather than quality, although there is undoubtedly some relationship between these concepts. There is an enormous literature on software quality – both in terms of the desirable characteristics of software artefacts and in terms of the development processes that are supposed to guarantee (or at least promote) these characteristics. From a traditional quality management perspective, it might seem appropriate to define a "good" component as one possessing some set of measurable intrinsic characteristics, which are somehow correlated to some set of needs. After all, this correlation between characteristics and needs is crucial to the official ISO definition of quality. (Although this definition is an holistic one – ISO 8402 refers to the totality of characteristics of an entity – quality management practice often attempts to treat these characteristics partially and separately.)

But when we try to discuss the diffusions of components, such notions of quality or "goodness" get us into difficulties.

For a start, in an open distributed world, there is no single value system or global intentions against which the goodness of a component – or anything else for that matter – can be evaluated. There are many stakeholders, and many perspectives. Any fixed position on component quality is going to be arbitrary.

Ecology takes no moral position

When we talk about diffusion of software components in terms of quality and purpose, we find ourselves apparently forced to choose between competing notions of goodness, based in turn on competing (if implicit) notions of ethics or rationality. If we talk about viability instead, this allows us to side-step this choice – and we can then reason directly about the characteristics of software components that are correlated with diffusion, regardless of who thinks these components are "good" or "bad", from whatever perspective.

Utility and Hedonics

The Roman architect Vitruvius, who lived at the time of Jesus Christ, defined quality as commodity, firmness and delight. Bill Gates has quoted this definition, and explained it as shown in Table 8.

Vitruvius	Gates Gloss
Firmness	"Consistency"
Commodity	"Be worthy of the user's time and effort in understanding it"
Delight	"Engagement, fun"

Table 8 Quality from Vitruvius to Gates.

In their study of the adoption of home computers, Viswanath Venkatesh and Sue Brown make a distinction between utility and hedonics, in order to account for a

degree of subjectivity in the adoption decision. Hedonics seems to correspond roughly to the Vitruvius concept of Delight, while utility corresponds to the seemingly more objective notions of Firmness and Commodity.

In domestic purchases of computers, a person may be willing to admit that they have purchased a more expensive model simply for aesthetic reasons. Some cheap home computers are bulky and unattractive. We might therefore expect to find particular emphasis on utility factors among purchasers of cheaper models, while hedonic factors would be stronger among purchasers of the more expensive models.

When the same person is buying computers for his/her company, however, there may be a reluctance to admit the influence of hedonic factors. Instead, purchasers will explain the selection of more expensive models by claiming higher utility – even though sometimes these claims seem fairly thin or optimistic.

From the standpoint of a software producer (such as Mr Gates), it is important to understand and quantify the impact of all factors on the adoption and diffusion of software components, including the hedonic ones.

This brings us to a different notion of hedonics, which includes all aspects of quality, rather than being contrasted with utility. Economists have developed hedonic pricing models, to account for the costs and benefits of various aspects and characteristics of quality in the prices of goods and services. Hedonic pricing methods are also used by environmentalists for attaching value to public goods and ecological assets. [Freeman, 1993].

Hedonic pricing is a method for assessing the price-contribution of each quality characteristic, by analysing a class of similar products with differing quality characteristics. It is used, among other things, to adjust productivity and other macroeconomic data – since in markets where there is a constantly rising level of product quality (both input and output) it would otherwise be impossible to compare productivity figures over time. (This is of course particularly relevant in economic measurement of the IT industry.)

Diffusion theory demands something very similar to this. If we want to compare the diffusions of various components across some landscape, over time, then it seems desirable to factor in the improvements in "quality" or other relevant characteristics that take place during our study. If I wait for six months before installing a new version of something, is this because (a) I'm waiting for the early bugs to be fixed, (b) I'm waiting until lots of people start sending me documents I cannot read without upgrading, or (c) I'm waiting until the price drops.

Resistance

One starting point for discussing resistance is that of a change agent, who has a set of change goals, and regards anything or anybody who gets in his/her way as a nuisance. Resistance is stupid and has to be overcome, using force, guile, patience or whatever other strengths and resources the change agent can access. This is all defined in terms of the change agent's goals.

At our Ambleside conference, Linda Levine offered a more balanced view of resistance – where it is rational to resist if something is either intrinsically flawed or not suitable for the intended purpose. Resistance can thus be interpreted as **due critique**. I see this interpretation as leading to a stratification of resistance: resistance at the first level might count as cooperation at the second level. Conversely, cooperation at the first level can sometimes be interpreted as resistance at the second level: mechanical compliance that avoids real learning.

Evolution

How do components improve over time? Many software producers seem to believe that software gets better by adding functionality. This often results in a component's becoming baroque and more difficult to use. Sometimes there are hundreds of functions that nobody wants – adding greatly to the size and complexity of the component and reducing its reliability, efficiency and flexibility. Such a software component would be vulnerable to another rival component that would offer the essential functions more simply, cheaply and reliably. One extremely popular suite of office software has a very high market share mainly because of the convenience of connectivity – it has become a defacto standard for document exchange – but this advantage might quickly disappear if rival products were able to use the same document formats.

Often the adoption of a software component entails a judgement about the future of the component. A short-term tactical decision merely evaluates the component with the characteristics it currently has. But if I'm going to build a component into my systems, or into my life, I need to know more than its present state – I also need to have some sense of how it is likely to evolve. Are lots of other people going to use it, what kind of people, and how will this influence the producers when issuing new versions – or perhaps abandoning the product altogether? Will it get more reliable as it matures, or will it get more complicated? Will it be made available on new platforms? These are strategic procurement issues, and depend crucially on the expected patterns of diffusion for the component. The component can be understood as an evolutionary envelope – a set of forking paths, leading to a range of possible future properties and configurations – and this is the true object of a strategic adoption decision.

Concluding Remarks

Towards a unified diffusion theory

Good and Bad Components

At present, there are two largely separate fields of diffusion theory. Security specialists study the diffusion of components that represent security threats – software viruses and worms, among other things. Meanwhile, diffusion theorists mostly study the diffusion of "respectable" and "well-behaved" technologies.

An ecological theory of diffusion doesn't take a moral stance towards components. We should expect the same forces and patterns to manifest themselves, regardless of how we sort the components into "good" and "bad" ones. This represents a unification of two previously separate fields of diffusion theory.

Resistance takes on an entirely different aspect in these two fields. Within a security context, as in the human body, resistance implies an effective set of defences against penetration or attack. Within a change management context, resistance is often seen as a problem to be overcome. A unified theory of diffusion could lead to a unified theory of resistance. Practitioners could use this theory to intervene in a more balanced way in the patterns of resistance.

Objective and Subjective Value

The study of diffusion can also usefully draw from economics and environmental science. Hedonic pricing seems to offer a way of putting all factors relevant to

adoption and diffusion – whether objective or subjective – into a unified framework of preference and value.

Hedonic pricing is relevant to an empirical test of the ecological model. It would be extremely interesting to measure the hedonic weight of the seven ecological principles – in other words, their relative importance for the rate and extent of diffusion. It would also be interesting to measure the hedonic weight of alternative principles – for example, to prove my hunch that functionality has comparatively little direct effect on diffusion. This would provide empirical validation of the ecological model as a whole.

Hedonic methods would also be relevant to a measured study of resistance, since we would then be able to differentiate more precisely between different aspects of resistance.

Further work

Practical research in this area to date has been limited in scope. I hope that future research will be able to provide some empirical verification of the model proposed in this paper, both to feed into theoretical work on diffusion, and also to provide much-needed support to practitioners of all kinds.

Further Reading

For loads of material about software componentry, go to the CBDi Forum website – www.cbdiforum.com.

Albert Borgmann, ***Technology and the Character of Contemporary Life.*** Chicago University Press, 1984.

A Myrick Freeman III. ***The Measurement of Environmental and Resource Values: Theory and Methods.*** Resources for the Future, Washington DC, 1993.

Kevin Kelly, ***Out of Control: The New Biology of Machines.*** UK edition, Fourth Estate, 1994.

Linda Levine, "The Ecology of Resistance". in Tom McMaster, Enid Mumford, E. Burton Swanson, Brian Warboys & David Wastell, ***Facilitating Technology Transfer through Partnership: Learning from practice and research.*** Proceeedings of IFIP TC8 WG 8.6 International Working Conference on Diffusion, Adoption and Implementation of Information Technology, 25th-27th June 1997, Ambleside, Cumbria, UK. Chapman & Hall, 1997.

Richard Veryard, ***Plug and Play: Towards the Component-Based Business.*** Springer-Verlag, 2000. See also www.component-based-business.com.

Richard Veryard, "Reasoning about Systems and their Properties". To be published in Peter Henderson (ed), ***Systems Engineering for Business Process Change Volume 2***, Springer-Verlag, 2001.

Viswanath Venkatesh & Susan A. Brown, "A longitudinal investigation of personal computers in homes: adoption determinants and emerging challenges" to appear in ***MIS Quarterly*** (forthcoming).

