1. The DCI Paradigm: Taking Object Orientation Into the Architecture World

James O. Coplien

Gertrud & Cope

Trygve Reenskaug

Professor Emeritus of Informatics, University of Oslo

Abstract: We find surprisingly strong parallels in a playful comparison of the progression of thought in the architecture of the built world and its namesake in software. While some architectural progression in both fields owes to fashion, much more of it owes to learning, both in the field of design and in collective human endeavour. The authors have been working on a paradigm called DCI (Data, Context and Interaction) that places the human experiences of design and use of programs equally at centre stage. It brings software design out of the technology-laced modern school of the 1980s into a post-modern era that places human experience at the centre. DCI offers a vision of computers and people being mutually alive in Christopher Alexander's sense of great design. DCI opens up a dialogue contrasting metaphors of collective human reasoning and Kay's vision of object computation, as well as a dialog between the schools of design in the built world and in software.

1.1 Introduction

Software architecture started as Fred Brooks' vision of a good metaphor for how we do software, and particularly for the early work of the programming-in-the-large forms of design. Somewhere along the line the metaphor took on a life of its own and lost many of its original roots. The metaphor became a place for non-coders to hang their hats, and architecture too often appears in the development process only as the source of artefacts that are thrown over the wall.

In this chapter we will look at the history of software architecture with a focus on recent history characterized by object-oriented design. Object-oriented design broadly characterizes many historic and contemporary methods that go by many names. All of them share the notion of encapsulation of state and behaviour in a run-time unit with a unique identity, and all of them separate the client of an object from the object itself by deferring the binding of the name of an object operation until its invocation. But, more fundamentally, they return to the basics of architecture foreseen by Vitruvius, as embodying a balance of critical thought and practical application [40]:

> ...[A]rchitects who have aimed at acquiring manual skill without scholarship have never been able to reach a position of authority to correspond to their pains, while those who relied only upon theories and scholarship were obviously hunting the shadow, not the substance. But those who have a thorough knowledge of both, like men armed at all points, have the sooner attained their object and carried authority with them.

Or, as Richard Gabriel notes [25, p. 231]:

> … Vitruvius, the classic Roman architect, characterized architecture as the joining of Commodity, Firmness, and Delight … In software engineering — if there really be such a thing — we have worked thoroughly on Firmness, some during the last 10 years on Commodity, and none on Delight. To the world of computer science, there can be no such thing as Delight because beauty and anything from the arts or the so-called soft part of human activity has nothing to do with science — it is mere contingency.

Perhaps it's noteworthy that, with market cap as a measure of success, the best of the best in contemporary mass consumer computing have the hallmark of being driven by customer Delight.

This chapter refocuses the software architecture discussion on its historical roots, in part by invoking the work of architects whose work has influenced the recent history of software architecture; in particular, Christopher Alexander, and other post-modernists. This is not just for the sake of nostalgia but also to drive beyond the superficial trappings of contemporary methods to the fundamentals that make us human. Most contemporary software architecture efforts remain mired in the modern school though they clumsily strive to apply agile vocabulary and principles. This leads to frequent breakdown in the metaphor. Architecture is used equally often to politically co-opt tangential areas such as knowledge management and project management, reinterpreted in a modernist framework, or to provide a vehicle for one organization to exercise political control over another. On the other hand, the agile position on architecture articulated here not only borrows directly from the post-modern school but can also be reconciled with its principles of balanced, practical human focus on life activity over structure for its own sake.

This chapter places the relatively new DCI (Data, Context and Interaction) paradigm on a firm architectural footing. DCI can be viewed as a culmination of many design goals over the years. In particular, this chapter illustrates how DCI addresses the fundamental issues that have arisen when drawing human users into code design. Such problems have manifested themselves as misfits in the modern-school worldview of object orientation, and we will show how we address them with the DCI paradigm.

1.1.1 **Agile apologia**

It should, but sadly cannot, go without saying that these perspectives on design support what today is broadly called “agile” in software. At the highest level, DCI is a celebration of the human in computing in the sense that the original goals of OO also put the end user at center stage. A post-modern perspective is firmly ensconced in “[i]ndividuals and interactions over processes and tools;” [8] this facet shows through in DCI’s emphasis of interactive software and human mental models. DCI is a boon to code intentionality at the system level, which many hold to be the *vade mecum* of software architecture, in obvious support of “working software over comprehensive documentation” [8]. Embracing human mental models and including architectural to the user interface, developed through the socialization of domain models and use cases, recalls “customer collaboration over contract negotiation” [8]. And a careful separation of the dominant shear layers of software development — domain data and business use cases — is a high-order evidence of “responding to change over following a plan” [8].

1.1.2 **Architecture and DCI**

It’s possible to present DCI as a programming technique that emphasizes object models and interactions between objects rather than classes. At a higher level, however, DCI is more properly considered as a paradigm for system construction that entails fundamentally different mental models than its predecessors. Just as the architecture of the built world progresses through paradigm shifts such as the school of the *beaux-arts* gave way to *art nouveau* and *art deco* in turn, so DCI introduces a new paradigm of software design at the level of software system architecture.

1.2 **The Vision: What is architecture?**

Architecture is a longstanding metaphor for software design and construction, and particularly for programming-in-the-large. Software engineering has largely embraced this metaphor in many forms, ranging from the use of the software title *architect* to the metaphors offered by the pattern discipline.

Architecture is the *form* of any system created through conscious design, and it thus has strong human elements both in its process and its product. The term *form* implies a deep mental model of the essence of some structure. A structure has form; a given form awaits implementation in structure. For example, an image comes into your mind when I invoke the word *chair*. For most people it's not a wholly concrete image: It may not even have a colour until the question causes you to assign it one. I might suggest that I meant to have you think of a five-legged chair and, though you are likely to have envisioned only four legs, you are likely not to protest that such a structure violates the *form* of *chair*.

A software architecture may characterize many different systems with potentially different features implemented in different programming languages. We are likely to say that two different consumer-banking systems have the same architecture even though they offer accounts that differ in many different parameters. Form is the deep essence of what is common between these systems, just as Victorian architecture is the essence of common elements across innumerable houses. Victorian architecture, client-server architecture, and Model-View-Controller architecture are about form. That they drive structure doesn't mean that they can't be conceptualized independent of structure. In fact, the presence of structure obfuscates form with distracting detail and non-essential elements. Architecture drives to the essence of a system.

The term *architecture* broadly touches a host of concerns in the built world, which perhaps best can be summarized in the terms popularized by the late Roman architect Vitruvius: *utilitas*, *firmitas*, and *venustas*. As captured by these terms, much of the classic architecture vision speaks to quality of human life. While architecture's link to fashion and even to aesthetics is controversial [46], commodity and utility (*utilitas*) are fundamental; so is beauty. Architecture is not without an engineering component that encompasses materials and techniques of construction, as good construction must be durable (*firmitas*) and arguably timeless [5]. Last, but certainly not least, architecture should inspire a human sense of delight (*venustas*). We can distil "delight" as comfort, beauty, or awe.

Because form is a result of design, and not of analysis, architecture lives squarely in the space of design. Architecture itself is therefore not principally about knowledge management, though knowledge management activities such as domain analysis and pattern mining often serve as powerful preludes to architecture. It is exactly this confusion in software, however, that often distances architecture efforts from the code and breeds scepticism among coders. Nowhere has this split become more pronounced than in the transition of software to agile software development, which is largely a movement among designers and coders.

1.2.1 Why do we do architecture?

It might be useful to revisit some of the key goals of architecture. As mentioned above, Vitruvius reduces the purpose of architecture to *utilitas* (commodity or utility), *firmitas* (firmness) and *venustas* (delight). These goals echo strongly in software, which has adopted them with its own emphases. More broadly, architecture is, and always has been about *form*. Except among specialists, the English word form is often confounded with structure, and software folks in particular often incorrectly infer that a system's architecture is the *structure* of its artefacts.

The proper architectural usage of the term *form* has historically been more precise. It's important to differentiate form from structure: Form is the essence of structure. We can talk in detail about the form of gothic cathedrals even without having a gothic cathedral at hand. Form is the conceptualization of structure in terms of the relationship between parts, and between the parts and their environment. Many given structures can implement a given form, just as there are many (different) gothic cathedrals, all of which implement the forms of gothic cathedrals.

1.2.2 Into software

These fundamental notions of the built world found parallels in the 1960s world of software construction. The architecture metaphor for software development, and particularly for programming-in-the-large, originated with Fred Brooks in the 1960s. Brooks himself was a bit sceptical of his own brainchild but, after discussions with Jerry Weinberg, became convinced of its metaphoric value for the software world [53].

Software has strongly embraced this metaphor, both for its casual parallels to programming-in-the-large on one hand and for some of its specific techniques on the other. Software engineering tends to emphasize the former, with the strongest parallels relating to the concerns around the coarse or large structure of software and how it relates to the prominent architectural features in the framing out of an edifice in the built world. The pattern discipline [5] is an example of the latter, whose philosophies of local adaptation and piecemeal growth became an alternative to big-up-front-design in the 1990s and flourished in the guise of the agile movement in the ensuing decade.

The architecture metaphor flourishes in software engineering literature today. The engineering and architectural metaphors arose only a few years apart. It should come as no surprise that the architectural metaphor stands out most strongly in the software engineering community, which views software as an extension of the engineering metaphor. Software engineering would expand rapidly as a metaphor in the late 1960s, owing to its popularization by Peter Naur in conjunction with the nascent software engineering community and its first conference in 1968 [37]. As with all metaphors, this one isn't perfect, but it tends to be more strongly flawed than most other metaphors, including that for architecture [19].

Today, the architectural principles of the built world continue to be mirrored in its software namesake in varying degrees. One can most often find the principles of *firmitas* in software engineering's exercising of the analogous English language terms *stability* and, more indirectly, *maintainability*. Software engineering's exploration of *utilitas* is isolated largely to the area of requirements management and formalism, touching the final built product largely through automated requirements translation rather than any act of design. *Venustas* in software languishes in the branches both of software architecture and software engineering, making only an occasional appearance in the human-computer interaction (HCI) and user experience (UX) fields, which have their own communities that are again often distanced from implementation or any human concern. In fact, the industry definition of software engineering itself is rather devoid of any human properties such as *venustas* or even *utilitas*: "The application of a systematic, disciplined, quantifiable approach to the development, operation and maintenance of software" [19].

Architecture thrives in a more humane way in the pattern community, outside software engineering, where beauty is still valued. However, the pattern community has paid little heed to utility: it is still largely a community of architecture for the coders, whose carpenter-like perspective is often indifferent to and sometimes antagonistic to end-user *venustas*. There are noteworthy counterexamples, of course, particularly in the HCI community (e.g., [51]), which struggles to bear the standard of *venustas* for the industry.

1.2.3 Why software architecture?

If building architecture is about *utilitas*, *firmitas*, and *venustas*, what is software architecture about? Here, the parallel between the architecture of the built world, and software architecture, works in good measure.

Most software architecture literature emphasizes *firmitas* in the guise of *maintainability*. Software first must work when it is delivered, and then keep working as requirements change. Even building architects emphasize the role of good architecture in supporting evolution through changing requirements, as Brandt describes in his classic *How Buildings Learn* [11]. Good architecture also offers building blocks, vocabulary, and world models necessary to worthy software.

Some software architecture schools (and the pattern discipline in particular) emphasize the notion of *venustas*: of the beauty of software. Most software literature emphasizes the beauty of the code. We are exhorted to write clean code [36] or, taking the architectural metaphor more literally, habitable code [25]. But there is another aspect to *venustas* that too often goes unheeded in software, and particularly among the software engineering crowd: the *venustas* of the interface. Good interfaces are attractive and usable. This deep kind of beauty goes beyond what just graphic designers do but touches deep mental models of the end user.

This perspective on *venustas* leads us directly into *utilitas*. Does architecture relate to usability? In fact, the program structure and the end-user structure have much to do with each other in an object-oriented system: that is much of the essence of Kay's Dynabook vision and of the Model-View-Controller vision. It's about matching machines to people in much the same way that architecture matches a house to its inhabitants. This fact is lost on most contemporary programmers. The interface is the product: the code is just the stuff that has to go along to make it work [41, p. 5].

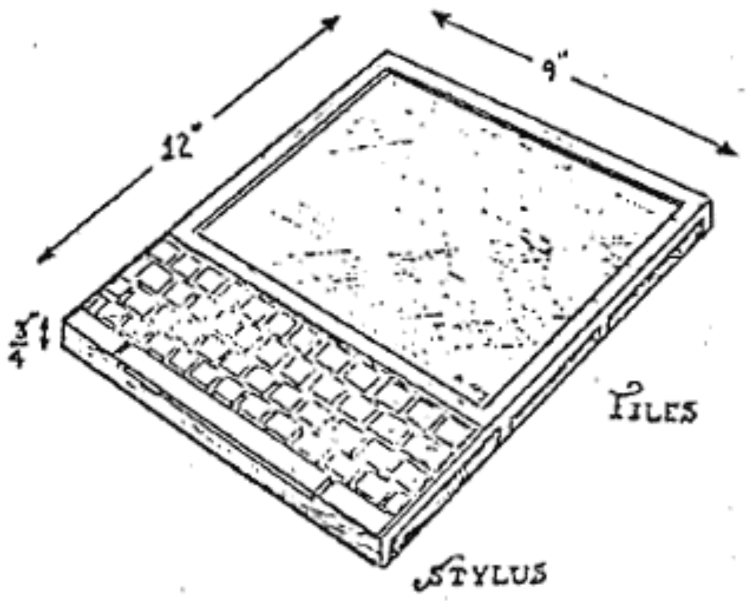


Figure 1: The Dynabook. From [32].

1.2.4 Architecture and the agile agenda

An agile approach is as much a sine qua non of contemporary development as architecture might have been in the 1980s. After the birth pangs of Agile dismissed the value of up-front architecture, or at least marginalized it, the industry is coming around to a more moderate position that accommodates a tenuous co-existence between them. DCI is one of the leading comprehensive approaches that spans this territory, as well as spanning the range of concerns from domain analysis down to coding concerns, and everything in between.

In this chapter, as we examine the relationships between Agile, architecture, and DCI, the question certainly arises about what boundaries to draw around *Agile*. Agile development as a titled movement is young, dating back to the Agile Manifesto in 2001. [8] Nonetheless, like all manifestos, it standardized what at the time was broadly established practice [20]. A look at history and publications suggests that its popularity can be traced back to a turning in the industry that started gaining momentum in the early 1990s. The 1990s became the decade of doubt, during which many sacred cows fell or were at least wounded. Architecture was unfortunately one of the casualties, but it has newfound legs and is regaining credibility as the industry discovers that emergence alone doesn't create good designs in time frames that the market expects.

The original Agile agenda took an anti-architecture turn in reaction to the top-down, overly prescriptive architecture techniques of the 1980s. (*Those* in turn were a reaction to the perceived lack of software discipline in the 1960s and 1970s.) Instead of looking to architecture (form) this new generation would look to *processes*, in the sense of autopoietic (self-maintaining) systems. The hope was that architecture would become emergent, thereby skirting the delay and cost of big up-front architecture efforts. The Agile Manifesto [8] attempted to capture this perspective through its focus on people, communications, pragmatism, market connection, and flexibility over stipulated processes and technologies. Agile would echo and amplify the pattern community's early leanings away from modernism towards post-modernism.

1.2.5 **DCI as an integrative view of the architecture metaphor**

One can view DCI is as a way of integrating the positive contributions of diverse communities such as HCI, software engineering, software architecture, and programming language. DCI respectively strives to embrace the end user experience, the need for low-cost software comprehension and extension, while still maintaining stable software artefacts with long service lives and providing a practical and elegant expression of its practice in accessible programming language technology. DCI didn't come about as an engineered solution to a wish list of such needs, but as a worldview rooted in the broad concerns of the relationship between computers and their users.

What does that worldview look like? It's about thinking of the computer as an extension of self that is, as Raskin says, "responsive to human needs and considerate of human frailty" [41, p. 6] that serves human value. This means being attentive to the mental models both of end users and of programmers. End users care most about what the system *does* while expecting the system to support their mental model. Programmers are also concerned about what the system *is*. In the same sense that the architecture of a village, or a resort, or an individual house is an extension of self — an ecosystem of forms that provide a framework for symbiosis between its inhabitants — so should the computer be an extension of its human subjects. This comes down to simple concepts like clear interaction metaphors, parallelism between programming constructs and the mental constructs of the end users, and clearly understandable program code. Most of these concepts relate to form, and that puts us squarely in the centre of architectural dialogue.

1.3 Form and function in architectural history

It's instructive to locate DCI's place in the march of programming history, extrapolating its trajectory from past practices that will be familiar to most readers here. We find striking similarities to the progression of ideas in the arts and in architecture of the built world, particularly as regards the age-old discussion of form versus function, as well as the place of control versus harmonization, and of technology versus human concern.

The built world and software would jointly consider these questions in the Design Movement, a loose collection of workshops, essays, and books in the 1970s and 1980s [22], [49]. Peter Naur of computing fame was among them, and the building architect Christopher Alexander — whose name would later become synonymous with patterns — was a major contributor to this body of literature. The debates and innovations of this era provide an interesting backdrop against which to discuss the DCI paradigm.

The modernist school of design could be said to dominate most of software history, and certainly its foundational years. Software gained its footing in a 1960s culture that firmly believed in the triumph of technology, including bold visions of artificial intelligence (that seem to resurge every few years with much less accompanying progress) and robots to automate our daily chores. In concert with this shiny, robotic world we find very little *venustas* in software. And while the times shaped our vision of software, it's noteworthy that software also shaped the times. Consider this man-bites-dog 1965 quote from Archer [6]:

> The most fundamental challenge to conventional ideas on design, however, has been the growing advocacy of systematic methods of problem solving, borrowed from computer techniques and management theory, for the assessment of design problems and the development of design solutions.

Software focused on construction, perhaps because it could: notions of coupling and cohesion, apart from their roots in organizational concerns, were easy to understand and to reduce to a number. With the echoes of 1960s modernism resonating from a recent past, the programming community gravitated naturally to these numbers that conveyed a sheen of science. It was all about *technical* goodness. Christopher Alexander would take note of this trap in the software world as late as the mid-1990s [4]:

> Please forgive me; I'm going to be very direct and blunt for a horrible second. It could be thought that the technical way in which you currently look at programming is almost as if you were willing to be 'guns for hire.' In other words, you are the technicians. You know how to make the programs work. 'Tell us what to do, Daddy, and we'll do it.'

This worldview is so strong in software that it is taken for granted. It's important at some point to emphasize that the software world adopted the architectural metaphor selectively, and this would be a good time to raise this issue. Of deeper importance here is that much of its use of metaphors is uninformed. A good example is the cacophony of attempts to automate pattern detection in programs, with a concomitant flurry of publications. None of them cite Alexander's own earlier forays into this territory, their failure, and the fundamentals beneath the failure. [17]

Another longstanding foible of the software community lies in the confusion of form with structure. Early architects turned to platforms and modules first, and protocols and interfaces only second, in their realization of the architectural vision. Architecture has always been closely coupled to the idea of reuse, and reuse almost always played out at the level of masses of software. Though the underlying economic motivations of this position were lost on few, there seemed to be few alternatives. Libraries and platforms flourished. This approach would be tempered somewhat with the rise of frameworks — partially filled-out architectures — only in the 1990s, some thirty years after the rise of the architectural metaphor in software.

Form suffered more subtle slights at the hands of software practice. Form, in architecture, starts either in the eye of the beholder or, as Alexander would have it, in deep processes that transcend even human existence. This notion dominates the conscience of the architectural profession in its use of the term. Software more commonly adapts a more vulgar use of the term rooted in engineering and technique. In the world of class-oriented programming, snippets of system behaviour don't exist outside the form of classes. Even in prototype-based approaches (such as that espoused by **self** [52]) behaviour follows structure (of instances) rather than form. *Venustas* suffers directly, and *utilitas* in a less direct way.

1.3.1 **Historic movements and ideologies**

There are strong parallels between *l'École des beaux-arts* and the primordial hacker culture of programming in 1960s-era MIT. The metaphor continues on the side of the built world into the mass-produced art of the Great Exhibition in the Crystal Palace in London in 1851, and the Arts & Crafts movement; in the advent of Software Engineering at the Garmisch conference in 1968 and the rapid rise of reuse and structured design; and in the Arts and Crafts movement in England in 1861 and the rise of "anthropomorphic" techniques of object orientation in the 1980s. Objects were, in many ways, the art nouveau of the programming world.

Even as architecture would evolve through art nouveau and art deco into the modernism of the 20th century, so object orientation would become a diminishingly human-centric concept in an increasingly technology-based community. The same kind of linguistic focus one finds in James Joyce's literature could be found in the language wars of 1980s computer science. The technological focus of modernism maps to the case tool craze of the 1980s. And the increasing focus on twentieth-century objectivism found a natural home in 1980s programming notion of objects: manifestations of concrete, real-world things.

Software architecture took a strange turn in the 1990s as the object-oriented programming community discovered patterns. The concept of patterns was refined in the built world by architect Christopher Alexander, a post-modernist who detests card-carrying post-modernists. In Alexander's definition, patterns are incrementally built elements of form necessary to the wholeness of, or a kind of quality that defies delineation of, some built whole. Each one transforms the whole from a less whole state to a more whole state. These patterns link together in a grammar that contextualizes each one and that imposes constraints on their ordering of application.

Software practitioners adopted the pattern metaphor to describe what they knew were essential forms of custom construction in specific domains. Because they are customized to a domain or particular problem, they aren't the general fodder of academic literature. The pattern community in fact consciously distanced itself from academic sponsorship and discarded academic mores of originality in favour of broad practices of communities in the wild. [18]

These foundations of patterns constituted a left turn because, first, they were more of a conscious departure from the status quo than a complementary framework to it. Good patterns didn't describe how to do object-oriented programming — that is, they did not take ordinary object-oriented staples such as encapsulation, polymorphism and inheritance as their building blocks. Rather, they tended to describe how to create code when pure object ideals or directly applied language constructs failed. Patterns became a way to describe how to survive software development when saddled with the dire constraints of object orientation, and they gave legitimacy to constructs that consciously violated sacred principles such as identity (most GOF patterns [26] break it), cohesion (most GOF patterns achieve their goal by distributing computation into additionally created objects), sub-typing through inheritance (patterns such as Façade allow simulation of inheritance with cancellation), and so forth.

Beyond this technical redirection we find even deeper ideals. Patterns grew up outside the community of software architecture and largely outside the field of software engineering; you find pattern literature in those fields only late in the maturity of the community. Rather than adhering to the largely technical agenda of those communities, patterns were explicitly about people. Patterns clearly blossomed in part because the early days of object orientation had laid the foundation for a human agenda of programming through approaches such as anthropomorphic design, and through the link that MVC created between objects and the human clients of computation. Elaboration of any true human agenda within object orientation itself was largely muted in the 1980s by the louder voices of programming language (modernism and James Joyce again) and automation (CASE tools).

1.3.2 Enter post-modernism

Computing today is enmeshed in a long-running slog of transition into post-modernism: the triumph of ideas over objects [49, p. 8]. These same terms that are used in the arts apply equally as well to software, and will figure in our dissection of DCI. In software, the pattern discipline of the 1990s published the first tomes of progress in this area. Like its counterpart in the built world and in movements such as art deco, the postmodern software world is focused on software for the masses, on compositional strategies over individual parts, and a focus on change rather than static beauty: "…to live in a perpetual present and in a perpetual change that obliterates tradition." [29] We find these notions in the rise of intentional programming, generative programming, multi-paradigm design, and aspect-oriented programming.

1.3.3 Architecture finds an object foothold

As generations of programmers are born into settings that are increasingly removed from Fred Brooks' environs, year after year, so the mores of the software engineer's version of architecture diverge increasingly from the roots of architecture of the built world.

Grady Booch arguably stood as the original doyen of object-oriented software architecture. It was largely through his extensive work and leadership that the object community came to embrace the architectural metaphor. Booch will best be remembered for his contribution to system modelling and to his cloud-icon notation, affectionately referred to as “Booch diagrams.” Along with Jacobsson’s use case contributions [28], it would later modulate the largely OMT-based semantics of UML.

Most practitioners from the last two decades of the last century will remember class diagrams as the primary useful component of UML, certainly as regards architecture. Jacobsson’s use cases, in the mean time, were relegated an important position alongside of, but not central to, architectural concerns. Architecture became synonymous with *structure*; behaviour was something else. Architecture and class diagrams were for architects; use cases and message sequence charts were for analysts. And it was the job of the programmers — software’s ersatz carpenters — to reconcile these two perspectives. There were, of course, noteworthy exceptions. UML 2.0 would compensate for UML 1.0’s paucity in this area, but did so at the expense of visual verbosity. SOA defined services, but at a level that was usually far removed from the code; it is probably a better metaphor for urban planning than for the architecture of a house.

Thus the object community stumbled into a dichotomy between form and function. Computer practitioners were perhaps predisposed to such a dichotomy anyhow: the previous generations had seen a split of records versus functions; I-space versus D-space; database versus process; entity-relationship versus data flow.

The architects of the built world were no stranger to this dichotomy of form and function, either. Design has often been a question of *utilitas* versus form.

1.3.4 **Software engineering and architecture today**

The same term in software more often relates to engineering practices than to the broader concerns of architecture. While architecture of the built world is indeed concerned about both the form of the whole (and, to a degree, of its parts) and about the engineering concerns of construction, software engineering tends to emphasize the structural, methodical and mechanical concerns. The software architectural landscape is littered with formalisms that speak more to construction than aesthetics. Even when invoking the pattern metaphor, most software patterns are more about engineering concerns than about any explicit nod to *firmitas*, *utilitas*, or *venustas*. Alexander’s original notion of generativity (indirect emergence of form) became confused with a notion of cleverness or obscurity, and patterns took more of a form of “aha” puzzles and their solutions than with human comfort or quality of life.

1.3.5 Measures of the vision

Software adopted architecture with the hope (justifiable perhaps only through revisionist thinking) that it would help teams create software structures that could be reasoned about in respective isolation. These units were informally called *modules*, and their degree of independence, *modularity*. Constantine proposed measures of good modularity based on the internal connectivity of a module (cohesion) and lack of connectivity between modules (de-coupling). Conway proposed that good modularity leads to team autonomy [16], and given that small, autonomous teams were more productive than monolithic groups, architecture would aid productivity.

More informally, architecture was seen as a discipline for the good of discipline. There is a tendency to believe that good architecture leads to systems that perform better and are more secure, but such claims relate less to any given architectural principle than to the timing of big-picture deliberations in the design cycle and to the proper engagement of suitable stakeholders. Architecture was an artefact that encouraged a front-loading of key activities that become awkward if pushed until too late

In fact, the object paradigm was unwittingly created with noble architectural ends: support the creation of built artefacts that could adapt to and better support the quality of human life. Little of this rationale appears to owe to the architectural metaphor or any roots in design theory, but the two roads would cross many times after meandering independently for many years.

1.4 What is object orientation? Achieving the vision

Computers were invented largely as mental aids. In inventing object-orientation, Alan Kay envisioned objects as a recursion on the concept of a computer. His metaphor of objects was that of a large network of interacting objects, each one of which was designed in-the-small to perform its own task well. From the perspective of the system architect one can view such objects as bricks whose individual contributions to architectural semantics are low. Elements of human value would appear at larger scales as emergent properties arising from the interaction of these large numbers of individual objects with integrity.

1.4.1 The Kay model

As inferred above, Kay's vision can be interpreted from an architectural perspective, or system level, as a metaphor for self-maintaining eco-systems. A system's structure is a consequence of its local adaptations over time. The human's place in this system is as the translator of real world nuggets into the language of the computer, at the level of its organs or, perhaps more instructively, of its cells. In the purest form of this system, the end user was removed from the burden of overall system design. Starting with a platform like Smalltalk, an end user could ideally express a few increments of interest where the computer could augment the end user's needs, and could make the system do their bidding by the incremental addition of a few objects.

It's crucial to note that the Kay model is highly distributed: It is in essence a network paradigm of computation. The overlap of this model with parallelism and concurrency is complex and difficult to delineate, and the industry is not yet at a point of integrating these perspectives though there have been numerous research attempts to do so.

We can say that the Kay model expects order to arise as an emergent result from the construction and interaction of individual objects of integrity. This early aspect of object-oriented programming, amplified by the pattern discipline's love affair with emergence, can certainly be identified as one of the roots of the agile ideology.

In what too easily can be considered a side note, Kay was acutely aware of the fundamental dynamic aspects of human mental models. Returning to the original Dynabook paper [32], we find:

> Two of Piaget's fundamental notions are attractive from a computer scientist's point of view.
>
> The first is that knowledge, particularly in the young child, is retained as a series of operational models, each of which is somewhat ad hoc and need not be logically consistent with the others. (They are essentially algorithms and strategies rather than logical axioms, predicates and theorems.)

1.4.2 Mental system models

Doug Englebart had earlier developed even deeper foundations for what later was to become object-oriented programming. Rather than thinking of the computer as an externalized tool or component, his vision incorporated the computer as an extension of human capabilities. Englebart speaks of augmenting the human intellect (though his work doesn't focus on the internal structuring of programs).

Behind Englebart's vision stand human mental models and a hope to extend those models into the computer. It became an early goal of object-oriented programming to capture those models. Kay writes [32]:

> We feel that a child is a "verb" rather than a "noun", an actor rather than an object; he is not a scaled-up pigeon or rat; he is trying to acquire a model of his surrounding environment in order to deal with it; his theories are "practical" notions of how to get from idea A to idea B rather than "consistent" branches of formal logic, etc. We would like to hook into his current modes of thought in order to influence him rather than just trying to replace his model with one of our own.

More prominently, MVC and Kay's brainchild Smalltalk would use objects to capture these mental models in the running program, in the "mind" of the machine.

1.4.3 **Model-View-Controller**

MVC embraced Englebart's vision of computers as an extension of the human mind, and translated that vision into an object-oriented world in which an interactive human interface played a central role. This interactivity was central to Englebart's notion of mental augmentation.

The central architectural paradigm, then, was to maintain synchronization between the end-user worldview and its representation as computer data. As with most design paradigms, the major organizing principle was partitioning. MVC's main partitioning structure is its *views*, each one of which corresponds to some *tool* by which the end user interacts with the computer. At a lower level, each tool comprised a dynamically assembled network of objects. Thus, the architecture had a large dynamic component of changing object connections and changing views. For any given view, there was a relatively stable configuration of objects that could be characterized by the same pattern: the relationships between its *models*, the *view* itself, and the *controller*. The *models* are the computer representation of the end user mental model of some object, and in fact are what programmers usually think of in association with the term *object*. The *view* arranges the presentation of those objects to the end user, usually in a visual form. The *controller* is responsible for creating and coordinating views and, together with the views, handles operations such as selection.

It is important to understand that MVC was not conceived as a library-on-the-side to add interactivity to a working program, but rather as the nervous system of the silicon part of the human-computer system. More broadly, MVC as an architectural paradigm includes the end user as well, and we now use the name MVC-U — where *U* stands for the end user — to emphasize this aspect of its design.

1.4.4 Patterns

The software pattern discipline took major departures from the Alexandrian vision of architecture, and these departures are no more apparent anywhere than in object-oriented practice. The *Design Patterns* book [26] was selective in its application of Alexandrian ideals. On one hand the GOF recognized that software has cross cutting constructs that aren't visible in the code, but are nonetheless part of the design vision of the programmer. This notion of scaling beyond individual objects to relationships takes us firmly into the realm of architecture.

Patterns were arguably one of the strongest foundations of the agile agenda. The ideas of piecemeal growth and local adaptation that are fundamental to pattern-based developments would be taken up almost verbatim by the pattern community. Agilists would embrace Alexander's valuation of human concerns over method less than a decade later.

1.4.5 Use cases

Human users usually approach a system with a concrete use case in mind. When you go up to an ATM machine, you bring your withdraw-money script or transfer-money script with you in your head. You have to learn it from scratch only the first time; the MVC approach helps your right brain train your left brain as you gain repeated experience with the script, or Use Case. The use case eventually becomes part of your left-brain mental model: this is long-term learning. This model has strong links to the right brain and its conceptualization of the "things" of the user world.

These use cases are only *complicated* (short of being *complex* or *chaotic*) in the Snowden taxonomy [47], which suggests that the emergence-based model of object system behaviour is overkill while paradoxically being impoverished in intentionality. Class-oriented code is hard to write and harder to maintain. The programmer cannot reason about how the end-user conceptualizes system functionality, which ends in a modelling stalemate between the end user and the programmer [34]. An example of a consequence of this mismatch is the frustration one experiences with a popular word processor when trying to insert a graphic in the middle of a paragraph: the mental models for the programmer and end user are clearly different.

Contrary to the Kay paradigm, the use case paradigm is a centralized view of computation. Use cases aren't really part of the "object canon." (Jacobsson's use cases indeed have an object model, but it is a meta-model that structures related scenarios rather than the mental models within scenarios.)

UML (the Unified Modelling Language) was an attempt to bring use cases together with the more data-centric facilities of the Booch method [10], drawing largely on Rumbaugh's OMT notation. The result is neither a paradigm nor a computational model, but a language for communicating such models or paradigms.

Use cases have a reputation of being anti-agile because they were widely abused in the 1980s. However, they are curiously suitable to incrementally structuring requirements in agile, and overcome many of the risks of the more popular concept of user story [15].

1.4.6 **Many views of objects and the boundaries of MVC**

The Model-View-Controller (MVC) vision in many ways tried to reconcile the network paradigm of Kay with the use case paradigm. It embraced the communication paradigm that one can extrapolate from Kay's vision: that is, that at its foundation, a system is a collection of many cooperating objects. On the other hand, MVC focused on the link between the objects and human mental models in concert with Englebart's vision of computers (and objects by extension) as human mental adjuncts. The vision goes back to Thing-Model-View-Controller in 1978 [44], which evolved into MVC. By drawing the human being into the world of interacting objects, MVC investigates the nature of interactions between objects — interactions that have their roots in the end-user mental model.

While Kay expressed his vision in terms of networks of communicating objects, he relegated the intelligence of design — of programming — to the level of the individual objects themselves, trusting the structure of their interworking to self-organization. This perspective is much in line with Alexander's vision of emergent structure. This perspective tacitly supported the idea that objects could be designed from the inside looking out instead of precipitating from a wider perspective of their place in system behaviour. Unfortunately, this viewpoint became institutionalized in the class: a way of designing individual objects from their identity as individuals rather than their roles in contextualized system operations.

MVC has only scratched the surface of Kay's communication paradigm. MVC captured the way that people view of the "things" in the computer's representation of their world. In the programmer's world, this is the program-in-the-large or, grossly, the form of the system data. The part of MVC that helps people understand the whole of the data forms necessary to a given set of related tasks speaks largely to the right brain. The brain takes in the screen information as a whole without specifically analyzing each part or its functionality. At any given time we have a static worldview and a static architecture, poised to transition into a successor static worldview after some event (usually from the user) drove the computer through useful business processing. This processing was opaquely relegated to the Model part of MVC, and it was easy to map MVC models onto Kay's autonomous objects.

Once the user has established this connection with the computer — which typically takes 10 seconds [14] — the end user now sets about achieving a business goal. That goal often entails multiple interactions between the user and the system following a script in the user's mind. This script is a gestalt, though it can be chunked along the boundaries between the end users' classifications of the "things" in their world according to use. When in this operational mode we conceive of real-world things according to their use in the moment; in a rain shower, a newspaper becomes a hat; for a motorcyclist, a garbage bag becomes rain gear. The right brain is dominant in carrying out these interactions towards the business goal: a focused, analytical use of the program-in-the-small.

This worldview isn't so easy to map into Kay's model because the end user details of object behaviour do cut across objects but yet are stable in the long term. There was nothing in object architecture that provided a reasonable home for a (static) architectural representation of these dynamics. By contrast the procedural world of FORTRAN, Pascal and C gave a home to these models at the expense of the right brain.

MVC didn't attack this right-brained aspect of user mental models. Other tool metaphors arose for these activities, most of them falling outside the architectural metaphor, and few of them led to concrete engineering practices. One powerful metaphor that combined both these worlds was Laurel's vision of the human-computer interaction through the metaphor of theatre, where the objects in a system become reminiscent of actors in a play and the user becomes a member of the audience [34]. But the most popularized model of the interactions between people and computers came in Ivar Jacobsson's use cases [28].

Sadly, both the FORTRAN/Pascal model and the use case model viewed what-the-system-is and what-the-system-does as separate concerns. That naturally led to the creation of separate artefacts in design and programming. Multi-paradigm design [21] advised us to use procedures for algorithmic-shaped constructs and classes for the more static elements of design; this led to terrible patterns of coupling between the two worlds.

The idealistic Kay vision suggests that individual small methods on small objects would naturally interact to do the right thing. A good metaphor is to compare these objects with people in a room who are asked to divide themselves into four groups of approximately equal size. It seems to work even without any central control. Snowden characterizes such systems as *complex systems* [46]. In summary, the original object vision didn't go far enough to capture the essence of the real world it was meant to model.

1.5 Shortcomings of the models

Software's dance with architecture was initially exploratory and playful, but the years have hardened it either into law or habit. Many of the paradigms of the early years became institutionalized in programming languages, methodologies, and standards. In retrospect, experimentation with the metaphor stopped too early, and today it's difficult to gain acceptance for any notion of "architecture" that lies outside the hardened standards. Many of my previous attempts to describe DCI on an architectural level have fallen on deaf ears because the self-titled architects can't recognize it as falling within their sphere of influence or exercise of power, and so they too easily dismiss it.

Architects speak of *shear layers* in built artefacts. Different parts of a house evolve more rapidly than others: a house needs a new roof every few years but rarely needs a new exterior wall. Good architecture provides good interfaces that separate the shear layers of its implementation: a necessity for evolution and maintenance. Class-oriented programming puts both data evolution and method evolution in the same shear layer: the class. Data tends to remain fairly stable over time while methods change regularly to support new services and system operations. The tension in these rates of change stresses the design.

DCI is an attempt to overcome these elements of structural inertia by returning to first principles and the deep elements of object foundations. Its premises seem to be born out in early experimentation and application. The rest of this paper will focus on the dialog between the pre-DCI world and status quo to help readers hone their understanding of the state of the art in object-oriented programming.

1.5.1 The network paradigm

Kay's original formulation missed the what-the-system-does component of design. It worked fine for simple programs where each unit of business functionality can be encapsulated in an object, but it left no place to reason about system behaviour across objects. Further, Kay and Ingalls rolled out this vision in a language called Smalltalk, which was widely adopted as a way to implement designs based on class thinking and class models rather than object models.

The class model places the programmer inside of the object, cognisant of its internal workings and constructions, but insulated from the interactions between its own objects and other objects in the system. This is a strange man-bites-dog reversal of the normal sense of encapsulation. The same class boundary that protects the internals of a design from concerns outside the interface, so that the programmer can reason about them locally, also insulates the programmer from the crucial design concern with interactions between objects. Each class ignores other classes' design concerns — and since there is nothing but classes in a class-oriented language, there is no locus of understanding relationships between classes.

Programming languages institutionalized this paradigm through encapsulation techniques. Programming environments provide little aid for reasoning about any structure beyond the class. One can argue that good environments express inheritance relationships between classes; however, inheritance is only a syntactic convenience that leaves the computational model untouched. Further, it is a temporary compile-time artefact that lies between human mental models in analysis and object instances at run time. It doesn't change the semantics of any object-oriented program if we flatten all base classes into a single derived class composition.

Design methods also institutionalized this worldview. One of the best known is responsibility-driven design, popularized through CRC (Classes, Responsibilities, and Collaborators) cards. While responsibility-driven design has the strong advantage of starting with scenarios or other use cases, the resulting artefacts ossify the behaviour elements into static relationships between classes, as the name "CRC" exhibits. In fact, end users don't conceptualize system behaviour in terms of classes (which are total classifications of form) but instead in terms of roles (which are partial classifications of form). Experience proved this to be a problematic approach. Rebecca Wirfs-Brock has since wanted to rename them to "RRC Cards" (Roles, Responsibilities, and Collaborators). She has instead kept the original acronym but has replaced "Class" with "Candidate" — like a *role* [54].

Good code conveys designer intent; great code captures end user intent. The ability of code to express intent is called *intentionality*. The embedding of the network paradigm, the class paradigm, and other early architectural metaphors for objects has caused intentionality of system behaviour to dissolve. DCI restores this intentionality to architecture by explicitly capturing use cases in a contextualized form.

1.5.2 **MVC**

Model-View-Controller missed the what-the-system-does component of design. It worked well for simple designs. MVC is better as a virtual machine than as the architecture built on top of it. It encouraged the atomic interaction style of human/computer interaction innate in the Kay world-view: a paradigm that viewed each object as being able to handle the user request atomically without much consideration for *sequences* of tasks between objects and the end user. MVC has been institutionalized with varying degrees of fidelity into many environments, such as Apple's Cocoa framework.

MVC's interests are largely orthogonal to DCI; the two are complementary. Historically, MVC emphasized data over interaction. While most programmers followed this paradigm and took it not only as the primary metaphor but the exclusive metaphor for their system design, it is not exclusive of the use case focus afforded by DCI.

1.5.3 **Patterns**

Though the GOF patterns claim Alexander's vision as their heritage, they are so remote from Alexander's vision of architecture as to be barely recognizable as patterns. Alexander's bore a clear tie to the patterns of events that they supported; there is little of this in the GOF patterns. Alexander's patterns were rooted squarely in the business domain and solved end-user problems; GOF patterns have no mapping to or from the users of the system. Alexander's patterns were fractal in scale; the GOF patterns live largely in the programming-in-the-small world.

Last, while most GOF patterns live in a class world rather than an object world, they hardly represent any uniform paradigm grounded either in objects or in classes. The overview of FAÇADE invokes the word *object* only once; *class* appears seven times [26, p. 185]. ITERATOR, however, mentions *class* 6 times and *object* 9 times [26, pp. 257-258].

Because of their Alexandrian heritage many OO practitioners came to believe that GOF patterns provided software architecture foundations. Software architecture practice embraced patterns, and that usually meant GOF patterns. This perspective reinforces the Kay programming-in-the-small model to this day.

1.5.4 Use cases

In the end a program offers a service. Object-oriented design has poor intentionality for a use case world model. Most object systems express and organize concepts closer to the program data model than to its process or behaviour model. The data part of software architecture is certainly a crucial perspective, but it's only half the story. What's worse is that the data view fails to express most of the client value of a software product. We sell use cases, not classes, and not even objects: end users don't usually conceptualize system behaviour in terms of classes.

From an architectural perspective, this leads the designer — who works at the source code level — out of touch with the dynamics of the whole. The hope held by the network model is that emergence will win out.

From a broader perspective, it's noteworthy that OO became the technology of choice for reusable libraries of containers, user interface components, and other APIs where the programmer can reason within a single class about the consequences of a business operation. While objects took off in these infrastructure areas they rarely thrived in applications with complex workflows.

DCI embraces the power of the emergence as in Alexandrian patterns but adds a focus on the *intent* of the design. A collection of well-constructed objects will no more generate meaningful system behaviour on their own than a collection of building materials will generate a structure suitable for human activity (Figure 2). As such, DCI can be seen as a paradigm that builds on Kay's original vision of socially responsible objects working together to generate collective system behaviour, but which extends that model to explicitly articulate intended behaviour. This inclusion of intent leads us into the arena of system behaviour, *its* form, and the articulation of this form.

One might ask: *whose* intent? The literature of contemporary software architecture is littered with allusions to the architect's intent. DCI holds the end user volition over that of the architect. This is more in line with the agile agendas of "individuals and interactions over processes and tools," as well as the agendas of customer collaboration, working software, and changes in the customer space. [8]

1.5.5 The object canon

Some object fundamentals are basic enough to transcend the schools of object orientation: encapsulation, polymorphism and friends. Each of these design techniques brings its own problems: information hiding is good, but

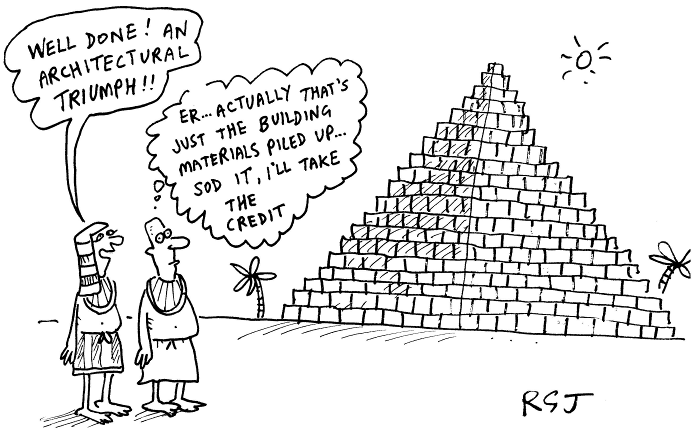

WELL DONE! AN ARCHITECTURAL TRIUMPH!!
ER... ACTUALLY THAT'S JUST THE BUILDING MATERIALS PILED UP... SOD IT, I'LL TAKE THE CREDIT
RGJ

Figure 2: Design emergence

hidden things are hard to find; polymorphism is a form of hyper-galactic GOTO. DCI strives to address many of these problems.

1.5.5.1 *Object-oriented programming isn't about classes*

Few programmers program objects or design objects. The class is most often the unit of design. This is absurd from an architectural perspective. Architecture traditionally has been about creating the artifice delivered to the end user. Carpenters use scaffolding and tools to achieve the architect's vision, and a great architect will be in there with the carpenters swinging a hammer. Most contemporary architecture thinking, however, seems to leave behind any thoughtful relationship between form and function but focuses instead on the tools. This may well be because great architectural talent arises from domain knowledge, and it's difficult to treat architecture as a generic discipline within the (generic) discipline of programming. In the end, architecture has arisen as a generic discipline of tools rather than the result of a quest for beauty and utility.

The preponderance of class thinking in software engineering likely arose from two sources: programming language technology, and interactive computing. Programming languages introduced types as a convenience that helped the compiler generate efficient code, and types were later adopted as a way to communicate a module's intent to its user. Class relationships such as sub-typing, commonly implemented using inheritance, provided an attractive mechanism to link programmer modeling to the compiler type system. This led to programming-by-increment using subclassing, as well as arguments for code reuse based on inheritance, that caught the imagination of software engineering. This was use case heaven.

Interactive computing inverted the traditional batch computational model. There is no human presence in a batch program, so the sequencing of function executions depends only on the data. Latency was not a core concern. Design becomes an issue of sequencing function invocations. In an interactive program, the human presence injects events into the program that result in unpredictable sequences of function invocations, and a quick response is imperative. Function sequencing is unpredictable, and the data model dominates. Classes were viewed largely as “smart data and became the loci of design, with most of the functionality subordinate to the data model. Early object orientation thrived on the noun-verb model of computation, where the “verb” component was usually a simple, atomic operation that could be localized to a class. Use cases were too easily forgotten in deference to the computational model arising from point-and-click.

1.5.5.2 *Class thinking isn’t limited to class systems*

The problem of single-object-think is aggravated by class orientation but is not unique to class-oriented thinkers. Most object methods are curiously reminiscent of a Kantian object world where individual objects act alone and programmers live inside of objects looking out: there is rarely any sense of collective behaviour in object-oriented systems, and there is rarely any degree of behavioural (self-)organisation. We are told that objects are smart data, but a closer inspection of both data and system behaviour shows something profoundly amiss with that characterization.

1.5.5.3 *Lack of locality of intentionality*

Adele Goldberg used to say, “In object-oriented programming, it always happens Somewhere Else.” Part of this owes to the innate thesis of object orientation itself: that intelligence is collective rather than localized. This shows up in three ways: polymorphism, deep object hierarchies, and deep class hierarchies.

Most object-oriented thinkers will link Adele’s quote to polymorphism, which is a kind of hyper-galactic shift in execution context that occurs with each method invocation. Polymorphism hampers our ability to understand code statically: we can follow the sequencing of method invocations only at run time. It’s perhaps no accident that there has been an increased focus on testing and techniques like test-driven development with the advent of object-oriented programming: If you can't analyze, test.

Second, object hierarchies tend to be deep. More precisely, objects usually lack a hierarchical structure but possess more of the structure of the network paradigm of computation. To an architect who bases a system on units that interact via inter-process communication, object orientation has the feeling of message passing and of asynchrony. Objects in fact embraced the message metaphor explicitly; that it might infer asynchrony or parallelism is perhaps unfortunate. That detail not withstanding, object orientation still has the feel of a pass-the-ball style of computation. This is a serious obstacle to program comprehension and intentionality because the program counter passes many abstraction layers on its way to accomplishing its goal.

Object orientation is designed so we are not supposed to know where the program counter will end up on a method call: object encapsulation and method selection insulate us from that coupling. We gain syntactic decoupling; we lose system-level comprehension. The supposed semantic decoupling of objects participating in a use case is largely an illusion, because in the end, each method executes in the business context both of the preceding and ensuing execution. It is difficult to reason soberly about a method in isolation, with respect to business goals.

Third (and closely related to the second) is that class hierarchies are also deep. Let's borrow an example from our good friends in academia who seem wont to employ zoo animals and shapes in their pedagogical examples. Here is the `roundRectPrototype` method of `Rectangle`, from Squeak:

```
roundRectPrototype
  ^ self authoringPrototype useRoundedCorners
           color: ((Color
                r: 1.0
                g: 0.3
                b: 0.6)
                alpha: 0.5);
           borderWidth: 1;
           setNameTo: 'RoundRect'
```

How many classes do you need to understand to fully understand this code? Most programmers will answer that we need just to understand `Rectangle`. In fact, objects of the `Rectangle` class include 7 other `Rectangle` methods, but also reflect a flattening of a hierarchy including `Morph` (with 47 methods) and `Object` (with 30 methods). The illusion exists at compile time that I need understand only this method or perhaps only this class. Programming languages hide the rest.

Much of program design, and programming language design, is in fact about separation of concerns. The lines that separate concerns can be thought of as reasoning boundaries whose goal is to delineate domains of comprehension. It's fine if such boundaries encapsulate the code relevant to a given "endeavour of understanding." But for non-trivial *system* behaviour, class inheritance layering and object layering of object-orientation cut across the fundamental unit of business delivery: the use case. Further, the additional class boundaries along the inheritance hierarchy add accidental complexity from the perspective of reasoning about system operations. And polymorphism de-contextualizes method selectors enough to make it impossible to reason about the behaviour of any contiguous chunk of static source code one writes in a given language and programming environment.

1.5.5.4 ***Summary of the shortcomings***

All of these shortcomings can be summarized as variants on one theme:

> *Traditional object orientation organizes concepts at the extremes either of a rather free-form network structure or of a single, punitive hierarchy of forms.*

The DCI paradigm strives to express a network model rather than a hierarchy, but provides disciplines for intentionality of form rather than leaving it to emergence.

1.5.5.5 ***Epicycles: early visions of relief***

Researchers over the years have recognized this problem and have discussed it in various guises, and a number of attempts have appeared to address it. Most of these solutions somehow relate to removing the limitations of thinking in a single Cartesian hierarchy by introducing richer forms of expression, all with the goal of higher code intentionality.

Howard Cannon's Flavors system [13] was an attempt to move beyond a strict classification that forced every object to be of one class at a time, to one that permitted the class itself to be a composition of multiple class-like things. Multiple dispatch [46] was an attempt to stop classifying methods in terms of their method selector and the single type of a single object, but instead to classify each method as potentially belonging partly to several classes at once. The **self** language [52] tried to destroy the very notion of classification as found in a traditional class, and to return to the object foundations that drew objects from the end-user mental model. Dependency injection [30] strove to blend the functionality of two objects into one. Multi-paradigm design [21], [12] refused to view the world according to a single classification scheme, making it possible to carve up different parts of the system in different ways.

The goal of Aspect-Oriented Programming (AOP) is similar to that of mix-ins, except its crosscutting units are more invasive at a finer level of granularity. Aspects are reminiscent of multi-paradigm design in that they allow a degree of separation of function and structure, but aspects' functional structure is much richer. It is more like having multiple knives carving up the same part of the system at the same time, whereas multi-paradigm design ensured that the knives didn't cross. Further, AOP again is about thinking in classes rather than thinking in objects: it is a very static way to attach a kit of adjustments to a program at compile time, even though it uses reflection to achieve its end.

While most of AOP is about a decorative re-arranging of code, and while that re-arrangement arguably makes it more difficult to reason about aspectualized code, it in fact does provide slightly an enhanced computational model because of its emphasis of reflection. The original AOP vision is in fact rooted in reflection and a desire to apply the kinds of reflection available in Lisp to non-Lisp languages like Java. Still, more than 15 years after its conception, one of AOP's inventors points out that it has failed to live up to even one of the three propositions justifying its potential value [35].

Most of these architectural "advances" can be viewed metaphorically as ornamentation of a base architecture rather than new paradigms in their own right. Rather than fixing the fundamental flaws in the vision of the paradigm, they tended to "patch" the paradigm with respect to singular concerns of coupling, cohesion, or evolution.

These discourses wouldn't be the only time in history that epicycle-like structures would arise to rescue object orientation. Flavors in fact can be viewed as a precursor to the DECORATOR pattern; multiple dispatch and dependency injection, to the VISITOR pattern; multi-paradigm design, perhaps as a weak form of the STRATEGY pattern. None of these approaches underscored the original principles of object orientation; rather, they offered localized repairs to the damage caused by applying the principles of class-based programming.

There are two notable techniques that challenged the hierarchical structures of class-based systems: a return to pure objects, and reflection.

The Actor paradigm [27] is typical of a pure object worldview. Its semantics are expressed in terms of interactions between objects that provide services, and it is a very good approximation to the network model. The **self** language challenged the notion of classes themselves. The **self** language, of course, can be viewed as an unabashed return to the fully network-based metaphor of computation in a way that applied it so uniformly as to minimize the problems of a class-based system. It's hard (but not impossible) to find hierarchy in the **self** world.

In the real world many social interactions are in fact emergent while others (like the course of a train along its track) are designed in advance. Sometimes a design problem arises that is difficult to regularize in any architectural form. The software design-level parallels to this adaptation are reflection and introspection. This is the realm of meta-object protocols (MOPs). MOPs have failed to gain traction over the years for a number of reasons. They require a mindset change that cannot be rooted in static, syntactic program analysis alone; few programming languages have risen to the occasion to express it; and methods that lead to the right reflection structures are elusive.

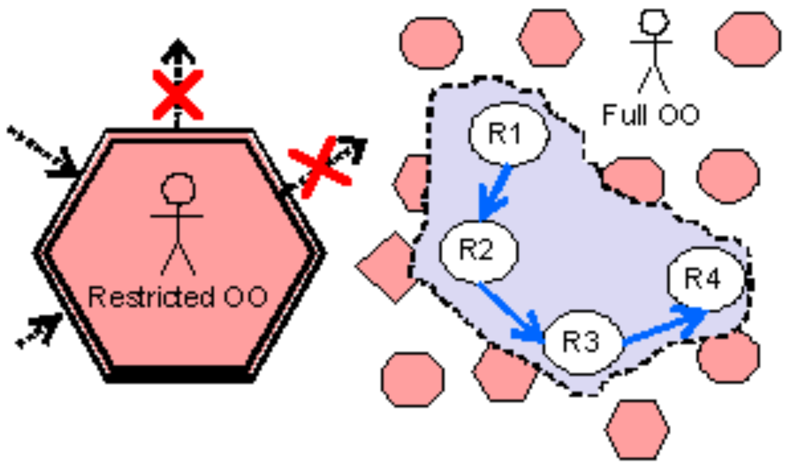
1.6 DCI as a new paradigm

DCI is a new paradigm of software architecture that emphasizes human mental models and improved system comprehension over class-oriented programming. Why is DCI a new paradigm? Many of its rules and tools are reminiscent of the most fundamental practices of class orientation: encapsulation, cohesion, objects with identity and state that represent local areas of domain concern, and so forth.

Part of what makes DCI a new paradigm is that it provides a new expressive way to view some of the same semantics that lie latent in the network computational model. Real-world objects in fact don't interact randomly or with total blindness to their environment, but form communities and chains of responsibility. DCI makes these structures visible in ways that class-oriented design techniques do not.

Many of the resolutions to the single-hierarchy problem mentioned in Section 1.5.5.5 did not fundamentally change the taxonomy of form, and only AOP changed the computational model. DCI is less about decorating or augmenting existing form than about carving a new form out of space itself: the form of function.

DCI is also progressive in how it uses carefully constrained reflection to provide the flexibility necessary to express the kinds of re-associations between objects that arise in dynamic human mental models.



One object seen from inside its abstraction boundary on the left;
a universe of objects seen from the space between them on the right.

Figure 3: Comparison of Restricted-OO and DCI worldviews

1.6.1 A DCI overview

We here give a brief overview of DCI from an architectural perspective. Such an overview cannot be complete in the space allotted. For more detailed information on DCI, see [9] or [42].

In the previous section, we discussed the shortcomings of the class oriented computational models. The shortcomings were related to the free-form network structure of objects and to a punitive hierarchy of forms. DCI employs intentional network structures and restricted classes to overcome these shortcomings.

Figure 3 serves as a background for the discussion. The shapes symbolize a universe of run-time objects. The system can be studied either from the inside of a particular object or from outside the objects in the space between them.

1.6.1.1 *Full OO*

In Full OO, I see the outsides of the objects and the messages that flow between them. Each object appears as a service. Its inner construction is hidden by its encapsulation and does not concern us.

A DCI network has a bound form. It is intentional and is designed to achieve a certain use case. A particular execution involves a sequence of objects where each is responsible for fulfilling its part of the use case. In the figure, a sample sequence is marked R1, R2, etc. Different executions of the same use case may involve different objects, but the network topology will remain the same. The nodes in the network are the *Roles* that objects play and the edges between them are the communication paths. The Roles are wrapped in a DCI *Context*; there is one such Context for each use case. In the figure, the Roles are marked R1, R2, etc. There is an ephemeral bond between the Role and the object behind it.

Communication is now a first class citizen of computer programming.

1.6.1.2 *Restricted-OO*

I am here placed on the inside of an object. I can see everything that is defined by the object's class with its superclasses. The class comprises both data and methods; state and behavior. The class won't appear in the code at run time; the intellectual concept called the class is absent. What exists is run-time objects. As I sit inside my class coding it is difficult to reason about other classes. We already know this from the Kay model, or network model, of OO computation. I can envision those objects but I can't know much about them. In fact, object orientation explicitly prevents me from knowing anything about any other object in my program because interactions between objects are polymorphic. Seeing an invocation of method *bar* on object *foo* doesn't help me find *bar*. There is an explicit abstraction layer between objects that prevents me from reasoning about them in concert. For this reason, we restrict our classes from sending messages to objects in the environment. Such messages are blocked with red crosses in the figure. We call this style of programming Restricted-OO because instances appear as self-contained services that are isolated from their environment.

While a restricted class says everything about the inside of an object and nothing about the objects surrounding it, a DCI Context says everything about a network of communicating objects and nothing about their inner construction.

1.6.1.3 *DCI*

DCI is an acronym standing for *Data*, *Context*, and *Interaction*.

With DCI, we move all methods that relate to object interaction out of the classes, attach them to the Roles in the appropriate Contexts, and call them Role Methods. What remains are the *Data* classes. They are Restricted-OO because all interactions with the environment have been moved out. The Roles are wrapped in a DCI *Context* and their Role Methods collectively specify the *Interaction* between objects that achieves a use case.

In a Role Method, I see an invocation of method *bar* on Role *foo*. I know the method since it is attached to the Role *foo* and there is no polymorphism in a DCI Context. I can, therefore, reason about the chain of methods that realize a use case.

There are three fundamental advantages of DCI. First, the complexity of the class is significantly reduced since it is Restricted-OO and no longer contains any interaction code. Second, Role Methods now appear adjacent to the methods in their collaborator Roles, thus keeping the code for the overall interaction algorithm in one place where I can inspect it and reason about it. Third, the Context is adaptive and self-organizing because it binds the Roles to objects afresh for each use case invocation.

Bank accounts serve as a frequent DCI example, with classes for different kinds of accounts. We want to support a system operation to transfer money between those accounts. As designers we envision ourselves in the run-time system and ask what objects we need and what responsibilities they must support to be able to transfer the money. One possible mental model has three Roles: Source Account, Destination Account, and Transfer Amount. The Role Methods makes the Source Account decrement its balance by the Transfer Amount after which it asks the Destination Account to increase its balance by the same amount.

Role names like Source Account, Destination Account and Transfer Amount come directly from the end user mental model. You can easily reconstruct them by asking anyone around you to give a succinct, general description of how to transfer funds between their accounts, and listen carefully what they say. They will refer to the objects involved in the transaction. More precisely, they invoke the names of those objects according to their roles in the money-transfer transaction. These are the new design entities, the Roles, which form the locus of business logic.

The Context encapsulates the Roles, their logic, and the interactions between them. After all, Roles make sense only in a Context: these Roles make sense only in the Context of Money Transfer. We might call the class `MoneyTransferContext`.

Now we have a system of source code where the Data, defined by the restricted classes, is separated from the business sequencing, defined by the Context. We separate the shear layers of what-the-system-is and what-the-system-does for independent maintenance. System behavior and locally focused class methods evolve at different rates. In traditional architectures, they are linked in a single administrative unit that either can cause the stable parts to inadvertently become dependent on rapidly changing requirements, or make rapidly evolving code overly dependent on code with high inertia.

We need to return once more to run time. DCI depends on a powerful run-time environment that dynamically associates objects with the Roles they play in a given use case. A program instantiates the appropriate Context object at the moment it is asked to enact a use case. Each Context in turn associates each Role with an object that plays that Role for the duration of the use case.

This association between Roles and objects makes each object to appear to support all of the corresponding Role Methods as part of its interface. While the DCI computational model doesn't stipulate how the run-time system should do this, it conceptually can be thought of as extending each object's method dispatch table with the methods for the Roles it plays. This can be done by directly manipulating the dispatch table in single-hierarchy languages (e.g., Python or Smalltalk), and can be done with traits in languages that have a stronger dual-hierarchy tradition (Scala, Ruby, and C++). More advanced implementations affect a just-in-time binding between a Role Method and its object at the point of invocation.

1.6.2 Relating DCI to the original OO vision

1.6.2.1 *How DCI achieves the vision of Restricted-OO*

DCI draws heavily on the domain modelling that one finds in both Lean Architecture [9] and in the original MVC framework [42]. MVC's first foundation is integrated domain services. The data classes in the DCI paradigm correspond almost exactly with the Model classes of MVC.

Furthermore, DCI's primary computational model is based on objects rather than classes. One understands program behaviour in terms of the logic in its roles; those roles are just behaviour-annotated names for the objects involved in the use case. This is reminiscent of the network model of computation germane to the original object vision.

1.6.2.2 *How DCI overcomes the shortcomings of class-oriented programming*

By capturing the system view of state and behaviour, DCI and its grounding in mental models go beyond the more nerd-centric vision of late-1980s object orientation to the visions of mental augmentation and human-centeredness.

Though both DCI and the original object vision take networks as their model of computation, DCI reveals the network structure with more intentionality. Think of a train system as an analogy. Trains and train stations are objects. We can think of train behaviour in terms of its arrival at a station: stopping, letting off passengers, closing the doors, and starting off again. We can think about stations in terms of receiving and sending off passengers. DCI steps up one level to express the regular patterns of station visitations by trains.

1.6.3 DCI and the agile agenda

The Agile agenda discarded many trappings of modernism: the triumph of technology over nature, the notion of form being subservient to function (instead of function having its own form), the notion of automation (automatically generated code) in deference to human craftsmanship, and many more.

DCI is very much in line with these architectural shifts in agile. DCI is much more about mental models than about technology — more about the end user's intent than the architect's intent. Good software, like a good house, suits those who inhabit it. On the technology side, the focus is on thinking and the creation of good separation of form. While some technological underpinnings are of course necessary to support the DCI model of computation, this issue has not risen to the level of language debate or of a battle of technological prowess.

DCI leads the programmer and the user of the code (sometimes the same person) into a dialog that helps capture mental models in the code. DCI offers a home for the end user mental model directly is in the code in Contexts and domain classes. That obviates the need for an additional level of documentation, removing a level of handoff and translation between the end user and the programmer.

DCI audits favourably against the Agile Manifesto [8]. The agenda of "individuals and interactions over processes and tools" is evident in giving the human-computer interface full first-class status in use cases. This "individual" may be the end user whose use cases relate to the business, or the programmer whose use cases are likely classic algorithms. Instead of depending on intermediate documentation to express the requirements, we go directly to the code where we express the mental model directly; that means that we're more likely to get working software than in the more myopic class-centred design. We focus on customer collaboration — both

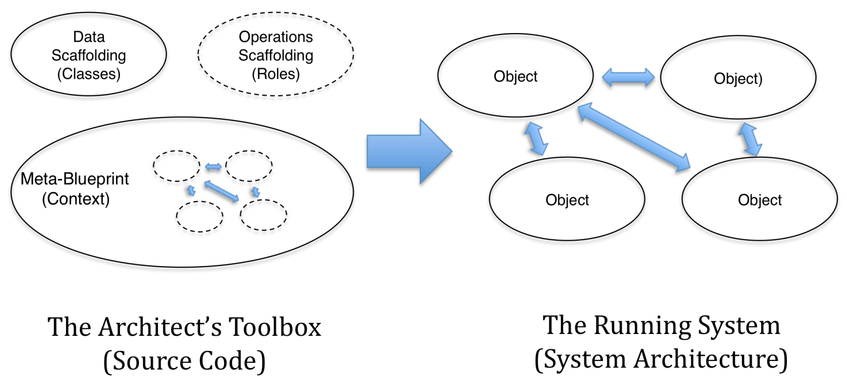


Figure 4: Designing a DCI application

between the team and the client at the level of use cases, and between the product and the client at the level of the mental models. We lubricate change by separating the shear layers of data and function.

1.7 DCI and architecture

DCI is in fact a radical break with the contemporary software architecture canon. Most software architecture metaphors are based on the source code or static (often class) structure. One darling of contemporary design is class hierarchy — a form that is absent at run time. Most contemporary expositions of run-time architecture are metaphors or computational models rather than models of form: Actors [27] and reflection come to mind. The DCI paradigm explicitly captures run-time models in the architecture: the form of function.

We can view DCI as an architectural school firmly ensconced in the post-modern worldview. It breaks with the modernistic notion that the technology (e.g., coupling and cohesion) is primary and takes a more utilitarian, human position. It is less about language (most modern programming languages can express DCI designs) or implementation technology (there are many stylistic variants of DCI) than about the computational model shared across the mind of the end user and the mind of the machine.

Today's class-oriented architect can't easily envision the form of the running artefact because the associations between objects at run time are somehow too dynamic. DCI constrains the run-time connections to a form prescribed by the Context, giving the architect the power to conceptualize and shape the run-time system (Figure 4).

Most important, DCI provides a home for the form of function. A Context encapsulates the interaction of a set of Roles. Each Role describes how its corresponding object interacts with other Roles to carry out a system operation. The network of *i*nteractions between roles (the *I* in DCI) is the form of that function.

1.7.1 **DCI and the post-modern view**

DCI is an approach to system architecture that is characterized by several post-modern notions:

- Value ideas over objects, including the expression of the forms both of data and function;
- Favouring compositional strategies over individual parts
- A broad-based human-centric agenda
- Focus on change

DCI has been embedded in a design approach called Lean Architecture [9] that has other aspects of the post-modern school, most notably the importance of process

1.7.1.1 ***Ideas over objects***

Architects of the built world have long been fascinated with form and function. The phrase "form follows function" is a vernacular English language idiom that stood as a truism for ages. Contemporary architects are wont to critique this posture and offer alternatives such as: "Form follows failure" [39], which evokes the need for change and dynamics in converging on a suitable architecture (section 1.7.1.4).

Returning to object orientation's roots in a worldview of emergent behaviour, we can view the form (think architecture) of a program as the result of accumulated learning over generations of program evolution. The accumulated knowledge is broadly called domain knowledge. Program form, then, has much of its roots in human conceptualizations of work. In the vein of post-modernism, DCI is about ideas over objects — more about the human side of systems than the system materials. These ideas take the shape of algorithms, use cases, or the patterns of arrangement of material beyond individual building blocks. DCI's Role interactions help us reason about these ideas.

1.7.1.2 *Compositional strategies over individual parts*

The modern school emphasized the primacy of structure. In the built world we find buildings like the Pompidou Centre in Paris that let the structure "all hang out." Class-based programming forces functional designers to become structural designers by thrusting them within a class framework. It is difficult to work at the level of pure form (e.g. abstract base classes) because there is no architectural home for functional concerns at the system level: only at the level of the data.

DCI is instead about compositional strategies: how to capture function and form and to reintegrate them under a computational model at run time. That model also integrates objects into a contextualization of their collective behaviour. Programmers can now reason about system behaviour because all code for any given use case is collocated in a single class. Execution hand-offs across objects (represented by roles) are statically bound rather than polymorphic.

1.7.1.3 *A human-centric agenda*

Team Autonomy: DCI data classes correspond to the domain organizational structure, and Contexts correspond to system-level deliverables. Recalling Conway's Law [16], this structuring supports team autonomy. Class-oriented architectures split use case code across the classes that form the major administrative units of object-oriented programming. In DCI, the code for a given use case is in the roles encapsulated by the single Context class for that use case.

End-user focus: DCI moves programming closer to the end users by embracing their mental models. It moves programming beyond the realm of a select few (called programmers) into the realm of the many (called end users). It places programming in an ecosystem of system behaviour rather than a separate area of its own. This recalls post-modernism's shift from art for the elite to art for the masses.

1.7.1.4 *Focus on change*

Change is about the kind of human-centred context and relationships, larger than objects, of the DCI paradigm. Consider this post-modern insight [50]:

> Complex systems are shaped by all the people who use them, and in this new era of collaborative innovation, designers are having to evolve from being the individual authors of objects, or buildings, to being the facilitators of change among large groups of people.
>
> Sensitivity to context, to relationships, and to consequences are key aspects of the transition from mindless development to design mindfulness.

DCI realises the pattern ideals of piecemeal growth and local adaptation to the extent that developers can add new use cases as stand-alone code modules independent of the domain structure.

DCI is not only about "design mindfulness" but more so about systems thinking. Being able to reason about use cases makes it possible to reason about system state and behaviour instead of just object state and behaviour. We can now reason about evolution at the system operation level in the context of supporting knowledge about market and end user needs. This is architectural thinking; class-based programming is limited to organizing kinds of software building materials in class hierarchies.

The evolution of the form takes place at the level of social awareness or progress at the level of the ideas. This *idea* focus (discussed above in section 1.3.2) is the first-order focus of change: Necessity is the mother of invention. Structure emerges from function during design. The functions of human endeavour arise from the supporting forms in the environment. DCI supports this function-centred focus in design as well as a structure-focused awareness in program use.

1.7.2 **Patterns and DCI**

Though patterns were broadly adopted for their power in describing geometric form (classes and objects) they in fact have strong roots in temporal geometry. Alexander prefaces his discussion of emergent structure with geometric patterns, but the geometric patterns are prefaced with a discussion of patterns of events. Indeed, Alexander sees deeply into a time-space relationship that makes it difficult to separate the two [5, pp. 65-66]:

> It is the people around us, and the most common ways we have of meeting them, of being with them, it is, in short, the ways of being which exist in our world, that makes it possible for us to be alive.
>
> *We know, then, that what matters in a building or a town is not its outward shape, its physical geometry alone, but the events that happen there.*
>
> …
>
> *A building or town is given its character, essentially, by those events which keep on happening there most often.*

This aspect of patterns is missing from almost all software practice. DCI is one of the first software architecture approaches to cite and build on this aspect of Alexander's work.

DCI also echoes the Alexandrian agenda in its end-user focus. Alexander put house design in the hands of those who inhabit them. A house must relate to their mental model, to which end the architect must step aside [3, p. 38]:

> On the other hand, people need a chance to identify with the part of the environment in which they live and work; they want some sense of ownership, some sense of territory. The most vital question about the various places in any community is always this: Do the people who use them own them psychologically? Do they feel that they can do with them as they wish; do they feel that the place is theirs; are they free to make the place their own?

The architect should focus on fine craftsmanship and beauty that harmonizes the human perspective with context of use in a fundamental way that transcends culture. DCI is such an architectural framework, and it defers the cultural (domain) questions to the mental models of the occupiers of the code: the end users and the programmers. Classic software architects are likely to find this agile perspective disempowering.

It is possible to view DCI as a unification of two orthogonal architectures: the data architecture ensconced in classes (Restricted-OO), and the behaviour architecture ensconced in roles (the full OO part). This view is particularly suitable to those who take a software construction perspective on architecture instead of thinking ahead to the run-time delivered system. A more subtle and deeper view of DCI notes that it in fact combines these two forms into one at run time, albeit dynamically in a way that is impossible to wholly capture in closed form. Yet the main rhythms and shapes of both the dynamics and statics can be expressed respectively in the Roles and classes of DCI source code.

This view is very close to Japanese models of the relationship of space to time and the way that space provides potential for some happening in time (e.g., 間 or ”ma”, sometimes translated ”space-time”). Such Japanese roots are at the core of Alexander’s worldview. Alexander himself reveals this perspective in his writing [5, p. x]:

> These patterns of events are always interlocked with certain geometric patterns in the space.

and [5, p. 70]:

> The activity and its physical space are one. There is no separating them.

1.7.3 DCI and the network computation view

Perhaps one of the most telling distinctions of DCI is the place it relegates to the human in the design process. If we think of the network model of computation in the extreme, design and intelligence are local: system behaviour is emergent.

Designers often put this network view on a par with patterns. Alexander's works feed this speculation with references emergence as well as to techniques such as automatic writing. However, a closer inspection of Alexander makes the human component of his design world obvious. He speaks more often about the power of community in sustaining a pattern language than he does about the role of the architect.

In some sense, the network computation view was based on well-intentioned individuals, with the metaphor relating to localized design ownership and collective execution. But this model lacked Alexander's notion of patterns — both in time and in space. Patterns, unlike Alexander's more fundamental Theory of Centres [2] are a human and social phenomenon.

DCI provides a vision of the role of human intellect, will, and design above the network model of computation. Humans design the contexts (social interpretations of collected behaviours) and the interaction of their roles to reflect the recurring "patterns of events" between objects.

We can revisit reflection in this context. The network model of computation is rooted in emergent behaviour. True emergence requires flexibility in software architecture that outstrips most architectural techniques, because it becomes difficult to tease out the underlying patterns. You might drive to work via a different route every day based on individual reflection that changes the path of interactions between your car and the intersections it passes.

DCI supports a weak form of reflection whereby Contexts can reason about the binding of role behaviour to objects at run time. This reflection supports a form of emergence in which modules come and go dynamically according to system use. Every use case *enactment* creates a dynamic module (a Context object) as a configuration of interacting objects.

1.7.4 Firmitas, utilitas, and venustas

DCI contributes to stability in its data architecture in the same way as Restricted-OO. Most such approaches will still use classes for the data architecture. But these classes are now freed from the rapid changes in behaviour driven by new business requirements. The architecture becomes more stable, and *firmitas* is served.

An important part of *utilitas* isn't in the software itself but in the relationship between the software and the end user. DCI gives the end-user mental model of behavior a home in the code. That lessens the risk of a translation error as can occur when splitting a use case across the widely separated domain classes implicated in a system operation. The architectural focus turns from rudimentary technical merits to first-class *utilitas*.

In the end, DCI is about an integration of human experience with computer support. Rather than separate man and machine through layers of translation and design, DCI strives for the vision of integrating the machine seamlessly into end user expectations. When used to manage the suitable selection of the computer as a tool of human mental augmentation, DCI can reduce work effort, rework, and the frustrating surprises that make computer life hell. DCI makes the program understandable so the coder can feel at home. It's about making the code habitable, in the direction of *venustas*.

1.8 **Conclusion**

DCI advances software into the human components of the architectural metaphor more deeply than class-oriented programming and other preceding paradigms. Further, DCI explicitly supports the agile agenda at the same level of the architectural values that serve a broader human agenda, with support for:

- end users and programmers as human beings with rich mental models;
- readable code to more easily achieve working software
- creating a home for the customer engagements of domain analysis and use cases;
- clean evolution along the dominant domains of change in software.

DCI is typical of the broader promises of a post-modern approach to architecture and problem-solving. Elements of DCI reflect a broader change in the design climate in software and the broader technical world, and the broader integration of computing systems that go far beyond the business applications of yesteryear to today's social networking infrastructure. Networks of interacting objects reflect the increasing consciousness about networks of interacting human beings through computer systems today and foresee the needs of architectural forms that can express these complex forms. Articulations of such understanding, such as DCI, will enable the leaps of functionality that this new world order will demand of their computing systems.

The interesting aspect of this new world order is that, unlike many software architecture approaches in this book, it is much less about technology than about human mental models of their world. As the great architecture efforts of classic civilizations have always strived to support the social functioning of the cultures in which they arise, so DCI and its related post-modern techniques can lay groundwork with the potential to raise the quality of life in all endeavors connected with computing. In a world where over 20% of people are connected to the Internet, with rapid growth, it goes without saying that a large fraction of human endeavor is at stake.

1.9 References

1. —. IEEE Standard Glossary of Software Engineering Terminology, IEEE Computer Society, 1990.
2. Alexander, Christopher. The nature of order. Volume 1: The luminous ground. Oxford University Press, 2004.
3. Alexander, Christopher. The Oregon experiment. Oxford University Press: 1978.
4. Alexander, Christopher. The origins of pattern theory: the future of the theory, and the generation of a living world. IEEE Software, September / October 1999.
5. Alexander, Christopher. The Timeless Way of Building. Oxford University Press, 1979.
6. Archer, L. B. Systematic method for designers. In Nigel Cross, ed., Developments in design methodology, John Wiley and Sons, 1984.
7. Archer, L. Bruce. Whatever became of design methodology? In Nigel Cross, ed., Developments in design methodology, John Wiley and Sons, 1984.
8. Beck, Kent, et al. The agile manifesto. http://www.agilemanifesto.org, accessed 2 June 2012.
9. Bjørnvig, Gertrud, and James Coplien. Lean architecture for agile software production. Wiley, 2010.
10. Booch, Grady. Software engineering with Ada, 1987.
11. Brandt, Stewart. How buildings learn: what happens to them after they're built. W&N, 1997.
12. Budd, Tim. Multi-paradigm design in Leda. Addison-Wesley, 1994.
13. Cannon, Howard. Flavors: a non-hierarchical approach to object-oriented programming. http://www.softwarepreservation.org/projects/LISP/MIT/nnnfla1-20040122.pdf, 1979.
14. Card, Stuart K., Thomas P. Moran and Allen Newell. The Psychology of Human-Computer Interaction. Hillsdale, NJ: Lawrence Erlbaum, 1983, p. 390.
15. Cockburn, Alistair. Why I still use use cases. http://alistair.cockburn.us/Why+I+still+use+use+cases, accessed 2 June 2012.
16. Conway, Mel. How do committees invent? Datamation 14(4), April 1968.
17. Coplien, James. Coding patterns. C++ Report 8(9), October 1996, pages 18-25.

18. Coplien, James. The culture of patterns. In Branislav Lazarevic, ed., *Computer Science and Information Systems Journal 1*, 2, Belgrade, Serbia and Montenegro, November 15, 2004, 1-26
19. Coplien, James. It's not engineering, Jim. IEEE Careers web log, http://www.computer.org/portal/web/buildyourcareer/Agile-Careers/-/blogs/it-s-not-engineering-jim, accessed 5 December 2012.
20. Coplien, James. Agile: 10 years on. InfoQ seris on the 10th anniversary of the Agile Manifesto, http://www.infoq.com/articles/agile-10-years-on, 19 February 2011.
21. Coplien, James. Multi-paradigm design in C++. http://793481125792299531-a-gertrudandcope-com-s-sites.googlegroups.com/a/gertrudandcope.com/info/Publications/Mpd/Thesis.pdf.
22. Cross, Nigel. Developments in design methodology. John Wiley and Sons, 1984.
23. Evans, Eric. Domain-driven design. Addison-Wesley, 2003.
24. Fowler, Martin. Dependency Injection. http://martinfowler.com/articles/injection.html, 2004.
25. Gabriel, Richard. Patterns of software: tales from the software community. Oxford University Press, 1996, pp. 9-16.
26. Gamma, Erich, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns of reusable object-oriented software. Addison-Wesley, 2005.
27. Hewitt, Carl. Actor model of computation. http://arxiv.org/abs/1008.1459, 2010.
28. Jacobsson, Ivar. Object-oriented software engineering: a use case driven approach. Addison-Wesley, 1992.
29. Jameson, Fredric. Postmodernism and consumer society. In The Anti-Aesthetic. Hal Foster, ed., Port Townsend, Washington, 1983.
30. Jenkov Jakob, Dependency injection. http://tutorials.jenkov.com/dependency-injection/index.html, n.d.
31. Kay, Alan. The early history of Smalltalk. http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html, 2007.
32. Kay, Alan. A personal computer for children of all ages. http://history-computer.com/Library/Kay72.pdf, August 1972, accessed 15 June 2012.
33. Kiczales, Gregor et al. An overview of AspectJ. Proceedings of ECOOP 2001.
34. Laurel, Brenda. Computers as theatre. Addison-Wesley, 1993.
35. Lopes, Cristina. Speech at AOSD 2012. 29 March 2012.
36. Martin, Robert C. Clean Code. Prentice-Hall, 2008.
37. Naur, Peter and B. Randell, eds. Proceedings of the NATO conference on software engineering. NATO Science Committee, October 1968.
38. Neighbors, J. M. Software construction using components. Tech report 160, Department of information and computer science, University of California—Irvine, 1980.
39. Petroski, Henry. Form follows failure. Technology Magazine 8(2), Fall 1992.
40. Pollio, V. Vitruvius: The Ten Books of Architecture. Trans. Morris Hickey Morgan. New York: Dover, 1960.

41. Raskin, Jef. The humane interface. Addison-Wesley, 2000.
42. Reenskaug, Trygve. The common sense of object-oriented programming. April 2009, http:// http://folk.uio.no/trygver/2009/commonsense.pdf, accessed 8 June 2012.
43. Reenskaug, Trygve. Model-View-Controller: its past and its present. http://heim.ifi.uio.no/~trygver/2003/javazone-jaoo/MVC_pattern.pdf, September 2003, accessed 9 June 2012.
44. Reenskaug, Trygve. Thing-model-view-controller. http://heim.ifi.uio.no/~trygver/1979/mvc-1/1979-05-MVC.pdf, 1978.
45. Reenskaug, Trygve. Working with objects: the OORAM software engineering method. Prentice-Hall, 1996.
46. Rybczinski, Witold. Home: A short history of an idea. New York: Penguin 1987.
47. Snowden, D.J. Boone, M. A Leader's Framework for Decision Making. Harvard Business Review, November 2007, pp. 69-76.
48. Steele, Guy L. Common List: The language. Bedford, MA: Digital Press, 1990, chapter 28.
49. Thackara, John. Design after modernism. Thames and Hudson, 1988.
50. Thackara, John. In the bubble: designing in a complex world, n.d., p. 7.
51. Tidwell, Jenifer. Designing Interfaces. Sebastopol, California: O'Reilly Media, Second Edition, 2012.
52. Ungar, David, and Randy Smith. Self: the power of simplicity. http://labs.oracle.com/self/papers/self-power.html, 1987.
53. Weinberg, Gerald. Personal interview with Jerry Weinberg, 31 May, 1999..
54. Wirfs-Brock, Rebecca. Personal E-mail of 14 October 2009.