Software design as multiple contrasting dialogues

Marian Petre
Open University
m.petre@open.ac.uk

André van der Hoek
University of California, Irvine
andre@ics.uci.edu

David Bowers
Open University
David.bowers@open.ac.uk

Abstract

Software design is a complex pursuit – technically, cognitively, and socially. Understanding that complexity – and managing it effectively – are ongoing challenges. Building on decades of empirical research on professional software design, and on existing literature, this paper presents a new characterization of software design that unpacks that complexity. The characterization drills down to the core of design as a goal-driven activity and expresses it in terms of parallel contrasting dialogues: (1) a dialogue between problem and solution, (2) a dialogue across application, interaction, architecture, and implementation design, (3) a dialogue across the design cycle of analysis, synthesis, and evaluation, (4) a dialogue between pragmatism and fitness-for-purpose, and (5) dialogues among the team members engaged in the work of designing,. There is an inherent tension and interaction between the dialogues, which emphasise different views. This is a mechanism by which effective designers manage the complexity: each dialogue provides a focus (if not a simplification) for design reasoning, but effective design maintains the interaction between dialogues and makes use of the contrasts between them to achieve design insight. This characterisation helps to explain both why existing software engineering methodology does not always work and what an effective 'design mindset' is; the paper discusses some of the implications of viewing current software design practices in this light.

1. Introduction

Software design is a complex pursuit – technically, cognitively, and socially.

We start with a definition of 'design': to decide upon a plan for a novel change in the world that, when realized, satisfies stakeholders as fit for purpose. This is a useful definition that captures three key characteristics: novel change, context in the world, and fitness for purpose.

The technical landscape changes continually, with shifts in technology, scale, and focus. The technology is used to address problems in multiple domains, and to satisfy multiple stakeholders with different expectations. Design is conducted in different social contexts, and at all stages of software development, from greenfield design to product lines to maintenance.

Software developers tend to be clever people, but nevertheless often software is late, it doesn't meet its specification, it doesn't work properly – why? And why, in contrast, are there nevertheless individuals and teams with exceptional track records, who repeatedly deliver software on time, under budget, working first time? In a landscape where technologies and infrastructures change orders of magnitude faster than personnel, one thing remains of constant importance: the ability of developers to be great designers. So what exactly sets expert software designers apart, and what enables them to achieve repeated and enduring design and development success, regardless of the technology or infrastructure of the moment?

The introduction of software engineering and methodology was historically a response to the 'Software Crisis': the increasing need for increasingly complex software, without a population of good developers at the ready to produce it [Haigh, 2010]. As Petre and Damian [2014] argued, software development methodology is about systematising (process) and standardizing (process and outputs) in order to achieve consistency and thereby provide leverage for communication, coordination, and, notionally, quality. But consider: making things consistent is about making things conform to a norm. This is why Damian and Petre argued that methodology can be a driver of mediocrity:

> "If we assume that practitioners are competent, then what drives their decisions? What do they take from methodology – when do they adopt it, and when do they decline it? Methodology affords potentially valuable leverage:
> - structure
> - coordination (standardisation, consistency)
> - re-use
> - communication (common language)
> - sharing artefacts (especially in a potentially diverse context)
>
> The importance of a specified methodology may be greater for less-experienced developers or for organisations that haven't already evolved their own mechanisms for these things.
>
> There are times when developers interpret methodologies strictly:
> - When they are first learning them.
> - When they fit their context well.
> - When the perceived or experienced benefits outweigh the costs.
>
> There are also times when developers deviate from strict interpretation:
> - To adapt to local needs.
> - When the cost of adherence exceeds the perceived benefit.
> - When the methodology (or its underpinning philosophy) is at odds with an effective existing culture.
> - When adherence is too constraining." [Petre & Damian, 2014]

A useful analogy likens the conventional view of software development methodology to driving a juggernaut down a highway. Methodology suppresses variation, because there is some value to be realised from systematic constraint, consistency, convention, and standardization. It is a structured process conceived in terms of driving toward a specified solution. However, when methodology is embedded in a culture of strict adherence, it takes on a momentum of its own. Methodology is a support, not a replacement, for critical thinking. And one of the things that distinguishes expert designers is just that: persistent and effective critical design thinking.

So how do expert software designers manage the complexity? Instead of trying to manage complexity through a fixed methodology, experts manage it by recognizing that different perspectives must be maintained simultaneously. They use different methods and notations as lenses, changing them deliberately, in order to address different perspectives on the design [Petre and Green, 1990]. We characterise this management of perspectives in terms of design dialogues. These dialogues intersect, contrast, and represent different, simultaneous perspectives on design – and accommodate shifts in the design space as the design evolves. The dialogues may be within the designer's mind, between a designer and some external representation, or among colleagues. Doing so both clarifies why design is complex and identifies key core considerations that designers keep in mind.

This characterisation is not wholly novel; it draws on and integrates ways that others have characterised design. It is, however, grounded in empirical studies of expert software designers and high-performing teams which are presented elsewhere (an annotated bibliography is available at: https://softwaredesigndecoded.wordpress.com/annotated-bibliography/).

The following sections introduce each of the dialogues in turn.

2. Problem - solution

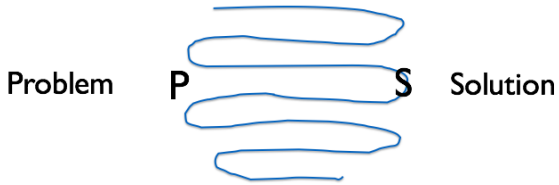

Figure 1: The dialogue between problem and solution

In practice, software designers are presented with a "problem" – perhaps a design brief or presentation or simple verbal cue or a problem ticket in an issue tracker – and are expected to generate a 'design', the plan for change in the world that can be realized to make that change in the world happen (the 'solution'). That 'design' may be realized as a schema, design document, UML diagram, mock-ups, code, etc. – depending on the context.

However, design is typically not a straightforward transition from problem to solution; rather, there is a dialogue that takes place between the design problem and the design solution. Dorst and Cross [2001] articulated this *co-evolution of problem and solution* clearly, and Michael Jackson gave prominence to this dialogue in his work, (particularly with *Problem Frames* [2001]), which stressed the importance of understanding the problem in context and in depth, and identifying and decomposing the requirements, in order to map them to a solution.

The design process is about managing the interplay between the problem and solution, as a 'dialogue' that emerges. This dialogue between problem and solution clearly includes both problem discovery and understanding, and generating a solution, and each of those includes multiple tasks: information-gathering, sense-making, synthesis.

Why is this dialogue necessary? Well, one could try to understand a design problem *ad infinitum* in the abstract. But 'the rubber tends to hit the road' [Anderson, 1998] when designers consider solution and problem in dialogue: new considerations emerge; gaps in the understanding of the problem emerge; the problem may be re-imagined when priorities shift with deeper understanding. Hence Dorst and Cross's emphasis on "co-evolution"; as the understanding of the design problem evolves, so does the solution space in which it is solved. The dialogue provides a basis for evaluating an evolving solution and its fitness for purpose.

3. Levels of abstraction / focus

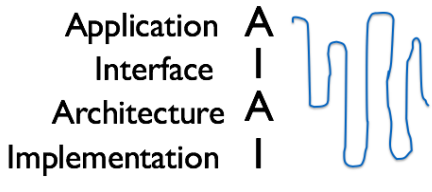

Figure 2: The dialogue between levels of abstraction, that is, between application, interaction, architecture, implementation.

A second dialogue takes place across different levels of abstraction and focus. Design is decision making, involving many decisions of different kinds. Clearly, not all of these decisions are at the same 'level' of focus, and, actually, some of these decisions inform others. For example, if we decide that the functionality of the system is to be fault-tolerant, that should inform the choice of platform we are going to use. If we decide to use MVC (mode—view-controller) as the dominant architecture, that is going to influence the data structures and APIs that we build. Etc.

Various characterisations of different levels of abstraction have been offered by other authors. We find it useful to distinguish four high-level categories of design:

Application design: "what is the software to do?"

Interaction design: "how does one use the software to do that?" (i.e., interpreting the functionality in terms of user interaction)

Architecture design: "how does it principally solve the problem?" (i.e., realization of both application and interaction design into an overall software solution)

Implementation design: "what are all the details that help make it solve the problem?" (i.e., realization of application, interaction, and architecture design into actual code).

The reality is, just as designers engage in a dialogue between problem and solution, they engage in an emerging dialogue among these four levels of design in an evolving solution space.

This emerging dialogue is not necessarily linear, from application to interaction to architecture to implementation. It may shift from application to architecture and implementation, for instance, to see if a certain piece of functionality can technically be implemented. Or it may move from interaction to implementation, coding up a UI independent of what the underlying architectural framework will be. Sometimes, indeed, designers just have to try some things out. The understandings of each level are imperfect, and they co-evolve as designers shift among the perspectives.

As designers continue to make decisions at all levels, decisions at some levels start constraining decisions that can be made at other levels. For example: Ania could not design a report, because the distributed database would be too slow in generating it, because data was distributed and the join was too expensive. A level of inertia emerged from prior decisions. Sometimes, designers are in a position where they can redo, but as more design decisions are made, they will be more constrained by what they have already decided.

Sometimes designers will purposely co-design parts at different levels and actually be working on multiple perspectives at the same time. For example, we often see UI elements next to architecture elements next to a set of functional requirements on a whiteboard, with the discussion rapidly moving among all three of them, often juxtaposing a pair, and then making updates to each.

Sometimes, work at the implementation or architecture level allows designers to realize that new opportunities arise at the application or interaction level (and similarly for other combinations of levels). For example, Fred could be in the midst of coding, and realize that something that he thought could not be built actually could, if he changes the code around, so he returns to the architecture and updates it – and he may subsequently return to the application and 'raise the bar' in terms of the reliability that he wants to achieve.

Those two dialogues – problem/solution and levels of abstraction - co-exist.

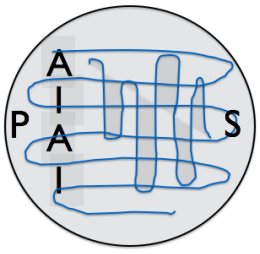

Figure 3: The first two dialogues co-exist and contrast.

4. The design cycle

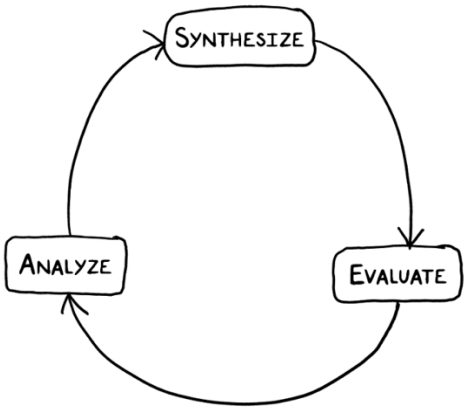

Figure 4: The design cycle.
[Figure by Yen Quach; used with permission.]

Another fundamental concept in design work is the design cycle: analyze -> synthesize -> evaluate [e.g., Lawson, 1997, p. 37]. This can be thought of as an iterative process – or as a set of critical thinking functions or skills – or as another form of dialogue, whose focus shifts in an iterative cycle (and in cycles within cycles), with each focus potentially informing each of the others.

The dialogues between problem and solution, and between the levels of abstraction and focus, occur within iterations of the design cycle dialogue, with the input from one dialogue informing the other.

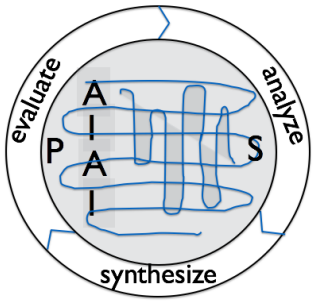

Figure 5: Overlay of the three dialogues so far.

5. Pragmatism and fitness for purpose

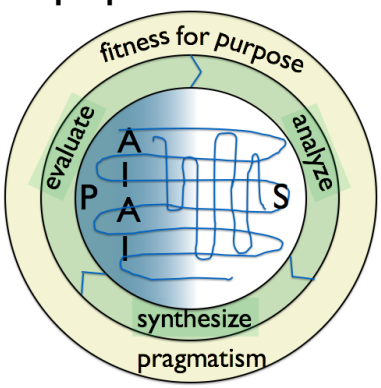

Figure 6: Fitness for purpose / pragmatism

Another crucial conversation takes place: the dialogue of fitness for purpose playing off with pragmatism. It is the dialogue that connects the other two dialogues to 'reality' – toward grounded decision making.

Design involves challenges that ultimately lead to the need for satisficing and making trade-offs. We cannot simply make the perfect design solution under given limitations on budget, time and effort. So, we need to engage in a conversation of when 'good enough is good enough', which is a conversation that plays out along the issue of desirability versus feasibility. This represents the trade-off between what is ideal from the perspective of the audience and other stakeholders, and what we can actually in the end design and build within the budget, time, effort constraints that we are given. This is where the designer ends up being the pragmatist, the maker of choices of 'what is in' and 'what is out' and 'why'.

The fitness-for-purpose/pragmatism dialogue is at the balance point between the well-informed understanding of the problem and the considered choices about what is prioritised in the solution. It takes account of the problem and solution contexts: of the intended purpose, audience, environment-of-use, and of practical constraints. It does not anticipate all possible interpretations and uses; rather it addresses the match between the intended purpose and the feasible solution (and so it relates to the first dialogue – between problem and solution).

6. Dialogues among team members

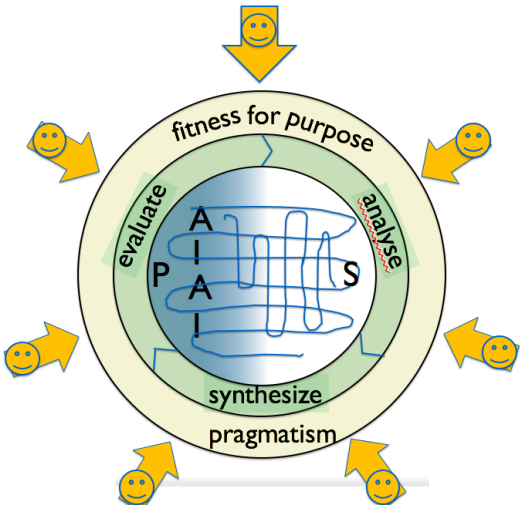

Figure 7: The contrasting design dialogues occur in the context of a design team – and its discussions.

Of course, all of this happens in the context of a design team. The design cycle of analyse - synthesize - evaluate is not only manifest in how one designer thinks in the moment, but it plays an equally important role in how we organize design activities in a broader social design context. It might start at a design meeting at a whiteboard. That conversation may have an overriding design cycle – with smaller design cycles addressing sub-problems or concerns that arise as part of detailed thinking or addressing the concrete ideas and decisions that contribute to progress. That meeting might be part of a sequence of meetings that constitute an over-arching design process. For example, those meetings might be part of an Agile sprint of two weeks. That spring might be part of a larger set of sprints within a given release cycle. And so on.

To sum all of this up: We engage repeatedly the design cycle, and engage in iterations of cycles, and in cycles within cycles. We do this at multiple levels. In doing so, we are engaging in different design dialogues: bridging between problem and solution; considering decisions at four different levels: application, interaction, architecture, and implementation. We integrate these dialogues in terms of the

balancing dialogue - about 'fitness for purpose' and pragmatism. This in terms of trade-offs between desirability and feasibility.

It is no surprise, then, that the software development community – organizations, developers, researchers – has tried to impose structure on it. Why? To simplify, so that designers and teams of designers do not to have to think about everything at once. And yet, maintaining awareness of all the different factors at play is characteristic of expert designers.

Hence this quotation from Nigel Cross: "Following a reasonably structured process seems to lead to greater design success. However, rigid, over-structured approaches do not appear to be successful. The key seems to flexibility of approach, which comes from a rather sophisticated understanding of process strategy and its control..." [Cross, 2003, p. 116] A structured process helps, but not if it's rigid, and only with sophisticated understanding, including opportunism and modal shifts.

Cross is indicating what we refer to as the 'design mindset' – which combines critical thinking, with an appropriate toolset of methods and analytics, and design thinking, with an openness to opportunity and change that drives innovation. Edward Glaser, in his seminal work on critical thinking and education [1941], defined critical thinking as:

> "The ability to think critically … involves three things: (1) an attitude of being disposed to consider in a thoughtful way the problems and subjects that come within the range of one's experiences, (2) knowledge of the methods of logical inquiry and reasoning, and (3) some skill in applying those methods. Critical thinking calls for a persistent effort to examine any belief or supposed form of knowledge in the light of the evidence that supports it and the further conclusions to which it tends."

This contrasts – and interacts – well with Nigel Cross's characterisation of design thinking:

> "… the following conclusions can be drawn on the nature of 'Design with a capital D':
>
> - The central concern of Design is 'the conception and realisation of new things'.
> - It encompasses the appreciation of 'material culture' and the application of 'the arts of planning, inventing, making and doing'.
> - At its core is the 'language' of 'modelling'; it is possible to develop students' aptitudes in this 'language', equivalent to aptitudes in the 'language' of the sciences - numeracy - and the 'language' of humanities - literacy.
> - Design has its own distinct 'things to know, ways of knowing them, and ways of finding out about them'." [Cross, 1982, p. 221]

Together, the two capture the both the critical and the creative aspects of the 'design mindset'.

There is no 'ideal' process – although there is a need for structure. The simplifications and idealisations help and may provide focus, but the 'reality' of the design process is that it is messy, with lots to consider and juggle. The multiple contrasting dialogues (with associated methods and notations) allow expert designers to manage focus while maintaining awareness.

7. Summary – and invitation to discuss

There is an inherent tension and interaction between these dialogues, which emphasise different views on a (changing) design space. This is a mechanism by which effective designers manage the complexity: each dialogue provides a focus (if not a simplification) for design reasoning, but effective design maintains the interaction between dialogues and makes use of the contrasts between them to achieve design insight.

This characterisation helps to explain both why existing software engineering methodology does not always work and emphasises the importance of an effective 'design mindset' – actively critical, with openness to change and opportunity. In expert practice, that mindset – and the dialogues – are supported by socially embedded and reinforced practices that help promote creativity and mitigate bias [van der Hoek and Petre, 2016].

This paper offers this characterisation as a proposition that is grounded in literature in the more general field of design research and in evidence from studies of expert software designers and high-performing teams (although we don't present the evidence here). Our purpose is to open a discussion about the proposition, and its implications for developing software design tools, practice, and education.

References:

Anderson, B. (1998) Where the Rubber Hits the Road: Notes on the Deployment Problem In Workplace Studies. Xerox PARC Technical Report EPC-1998-108, later published in: Paul Luff, Jon Hindmarsh and Christian Heath (eds.), *Workplace Studies: Recovering Work Practice and Informing System Design*, Cambridge University Press.

Cross, N. (2003) *Designerly Ways of Knowing*. Birkhäuser Architecture.

Cross, N. (1982) Designerly Ways of Knowing. *Design Studies*, 3 (4), October, 221-227

Dorst, K. and N. (2001). Creativity in the design process: co-evolution of problem–solution. *Design Studies*, 22(5), 425–437.

Glaser, E.M. (1941) An Experiment in the Development of Critical Thinking, Teacher's College, Columbia University.

Haigh,T. (2010) Dijkstra's Crisis: The End of Algol and Beginning of Software Engineering, 1968-72, http://www.tomandmaria.com/tom/Writing/DijkstrasCrisis_LeidenDRAFT.pdf [accessed 13 May 2014]

Jackson, M. (2000) Problem Frames: Analysing & Structuring Software Development Problems. ACM Press / Addison Wesley Professional, ISBN 978-0201596274.

Lawson, B. (1997) *How Designers Think: The Design Process Demystified*, Third Edition. Architectural Press

Petre, M., and Damian, D. (2014) Methodology and culture: drivers of mediocrity in software engineering? *FSE 2014* (Visions and challenges track).

Petre, M., and Green, T.R.G. (1990) Where to draw the line with text: some claims by logic designers about graphics in notation. In: (D. Diaper et al., Eds.), *Human-Computer Interaction—Interact '90* . IFIP, Elsevier Science Publishers (North-Holland), 463-468.

van der Hoek, A., and Petre, M. (2016) *Software Design Decoded*. MIT Press.