Software architecture as a wicked problem

A. F. M. van der Sijpt

February 2007

1 Introduction

During my Master's project, the feeling arose that software architecture is a typical 'wicked problem'. This was not the goal of the project, though it could make a nice conclusion for the thesis. This document contains a full analysis of the wickedness of the activity of software architecture.

Section 2 explains the nature of wicked problems, Section 3 gives a characterization of software architecture, and Section 4 maps these two field. Section 5 shows some conclusions and suggestions.

2 On wicked problems

In 1973, Rittel and Webber published 'Dilemmas in a General Theory of Planning' [RW73], introducing the notion of *wicked problems* in social planning. The core of the argument is the lack of optimal solutions for policy issues in social areas such as education, public health care and urban planning.

The social problems intended by the authors have a number of defining characteristics, some of which are inherent to today's complex society, in which there is no undisputable public good.

A good example is the 'fairness' of health insurance contributions. Is it fair to charge an equal sum from everyone? Should the contribution be a fixed percentage of one's income? Should people with a healthy lifestyle pay less, or those with chronic illnesses pay more? Clearly, no single solution, not even a balanced mix of ingredients, will make *everyone* happy.

'Not having a definite solution' is one of the characteristics that makes a wicked problem. This distinguishes wicked problem from hard (or complicated) ones: creating an algorithm for deciding primality of a number in polynomial time (the AKS primality test) is a complicated task, but its goal is very clear. In the health-insurance example, not even the problem itself can be defined in a definite way: how does one define 'fair'?

Besides, the problem might not be self-contained: changing people's social security situation might have implications on their lifestyle, influencing the number of illnesses in the population. Even more indirect effects may happen. It is not unimaginable that charging more for health insurance will bring people into financial difficulties, resulting in a slowdown of the economy and an increase of stress, leading to increased mortality rates younger age due to heart-related diseases, followed by a decline in demand for geriatric services. This indicates that resolving a problem in a given way may have unrelated effects, since the problem itself is a symptom of another one.

Note the use of *resolution* instead of *solution*. The authors use this term to indicate that proposed actions do not necessarily solve the problem at hand, but seem to be adequate given the current limitations and insights.

2.1 Criticism

The term 'wickedness' has been adopted in very diverse fields, often without justification beyond 'this problem has a social context'. Although I do reject some statements from [RW73], such as the classification of 'provide housing for the majority of the population' as a tame problem or the

notion that problems should be defined as a 'desired condition', I do believe that the characteristics give a good representation of what makes certain problems 'look different every time you look at them'. Following the characteristics, I believe that some problems cannot be solved in a formal manner, but only 'pretty good' or 'sufficiently correct' resolutions can be sketched.

Apart from many followers, there has been some criticism in the past. One notable example has been published in the same journal as [RW73].

Archie Bahm accuses Rittel and Webber of 'hiding' (their) incompetence behind the notion of wicked problems [Bah75]. His main claim is that it is 'easier' to blame the problem than to blame the practitioner. He believes the incompetence is caused by lack of a philosophical basis of science and social issues present in our society.

As such, he may be right. There indeed is a lack of philosophical understanding of the fabric of science, both in our society, and in the professional and academic realms. However, his claim that wicked problems are a phantom created by incompetence overshoots its goal. Rittel and Webber do not claim that classifying a problem as wicked is a justification for giving up, as is Bahm's interpretation. Rather, they notify the reader of the existence of a class of problems which are susceptibel to over-simplification.

As stated in Section 2, ten characteristics are provided in [RW73]. Bahm strengthens his claim by refuting six of them. He chooses to provide counter-examples: problems classified as wicked by Rittel and Webber, for which he shows that a given claim does not hold.

However, he uses different examples for all claims. Thus, while refuting a claim for some problem, he refuses to make any statements about the other claims, which just *might* hold for that problem. By including the observation that not all wicked problems should have all characteristics[1], it can safely be stated he refuted *some* claims for *some* problems, but certainly not *all* claims for *all* problems.

3 On software architecture

3.1 What is architecture, anyway?

The term *architecture* has been used in construction for millennia[2], referring to the science and art of designing structures for human use. Over the ages, architecture evolved from a method of joining needs and means (e.g., need for housing and the availability of sticks and leaves) to the current profession of designing structures in relation to their environment.

In a more general setting, architecture refers to *translating a vision, based on needs and constraints, into artifacts which can be used as a basis for further development*; this gives architecture a significant subjective component.

Ever since programming became a discipline of increasing scope and complexity, terms as 'structure', 'design' and 'architecture' have been used to describe a high-level overview of the system at hand. John Zachman is one of the first to give a solid analysis of the similarities between architecture in construction and architecture in software [Zac87]. He presents two cases of architecture (building a house and constructing an airplane), followed by a mapping to the field of information systems.

Software engineering holds a great promise of reuse. Even at the level of architecture, designs can be reused for similar problems, e.g. an online store will have different characteristics each time, but after developing some, patterns start to emerge. From this point on, a standard architecture can be reused and tuned for each application. The architecture process Zachmann considers, only applies to the first iteration of such a process, although the line between the two is fuzzy.

[1] It can be agreed that a problem statement in the form "that this nation should commit itself to achieving the goal, before this decade is out, of landing a man on the moon and returning him safely to earth" has some wicked elements, but it certainly has a stopping rule.

[2] First in ancient Greece, meaning 'a master builder'.

Software architecture is hardly the only –architecture out there. All fields dealing with creating[3] complex structures rely on a form of architecture. Information systems, enterprises, buildings, tools, ... can all have a high-level view which allows relating the artifact to its environment.

3.2 Software architecture in a social context

In a sense, the (software) architect is an interpreter, working between the users (stakeholders) and the builders of a system. Just as an interpreter in a courtroom needs to have knowledge of both languages, the architect needs knowledge of both fields, each having its own 'languages'. He must understand the needs (and potential future needs) of the client, and the possibilities (and limitations) of the builders. Combine this with stakeholders wanting the best solution for the lowest cost in time and money, and we end up with a complex system of inherently contradictory interests.

However, not all interests are directly related to the problem the solution is intended to solve. Many social factors enter the stage, like company politics. It is the architect's job to find an optimal balance of these interests, even though most of which cannot be described by some formal method; many interests are not even outspoken or 'visible'.

3.3 What problem?

Many design methodologies start from a well-defined problem, and from that point on, work toward a solution in well-defined steps; this can be automated pretty well. However, the complexity described above shows that *describing the problem is the problem.* It is also hard to 'understand' the problem; it may well be possible to pinpoint some 'load-bearing' factors of the problem (which is an art in itself), it is not possible to exhaustively sum up all the influences a solution should try to balance.

Instead of trying a formal approach, an architect relies on experience and intuition (which, arguably, may be the same thing, just on a different level of consciousness). This, still, does not lead to a full understanding of the problem: rather, it leads to a mental model of a *solution* to the problem at hand.

3.4 What solution?

As stated before, it is impossible to word an architecture problem in all its richness. The reverse applies to a solution, however: since a solution is turned into a tangible artifact (likely, code), all details have been sorted out. However, there is no definite method to prove that a given solution is actually the *best* solution under the given circumstances, or even that it does actually resolve the problem.

Methods for evaluating a given set of solutions do exist; still, these only include *given* solutions. They will return the solution *from the set* that is likely to achieve the best balance among the *named* interests.

4 Software architecture as a wicked problem?

Rittel and Webber provide ten characteristics for wicked problems. Upon closer inspection, a number of these overlap. They will be grouped into three broad categories, which will be shown to apply to problems of software architecture.

This grouping is inspired by the observation that many characteristics are caused by the same source.

[3] The 'creation' is a key element: many other fields of science deal with complex structures, but feel no need of describing an architecture. Examples include astronomy and medicine.

Not *all* wicked problems have *all* characteristics; I believe any problem that has one or more characteristics from each category can be safely classified as a wicked problem, and should be dealt with accordingly.

4.1 On formulation and interpretation

Two characteristics have a notion of 'problematic formulation', being (1) "There is no definitive solution to a wicked problem." and (9) "The existence of a discrepancy representing a wicked problem can be explained in numerous ways. The choice of explanation determines the nature of the problem's resolution.".

Software architecture problems, as described in Section 3.3, are usually open for many different interpretations. A courtroom interpreter may translate (intentional) ambiguities from one natural language to another. An architect translates from a natural language domain, including unspoken elements, to a formal domain, usually resulting in software. Hence, the ambiguities must be settled.

Certain ambiguities may not be caught by the architect, and end up as an unresolved issue in the specification of the problem. To prevent this, formalisms are used that describe the problem without any ambiguities. These abstract away from some details of the problem, overcoming part of the problems' complexity, thus allowing the architect to concentrate on the left-over core of the problem. However, herein lies a more grave situation: formalisms may both "guide and blind". The 'guide' part is clear: formalisms provide a certain way of thinking about a problem, proposing an interpretation, that can 'incidently' be expressed in that formalism. This guidance also 'blinds': once a certain avenue of thought has been chosen, it is hard to deviate from that and consider alternative interpretations. This chosen avenue brands certain issues as 'not important'. They are likely to be forgotten about altogether, since formalisms have no way of tracking the elements they intend to abstract! They are being ignored, possibly without knowing their significance or impact.

With a fully understood problem, these problems are likely not to occur; a good designer will spot these pitfalls, and choose appropriate formalisms. With the class of problems considered here, however, the problem is only fully understood when a formalized version has been created, or, in essence, an avenue for solving the problem has been chosen. The interpretation can then be heavily influenced by the formalism. Hence, *describing the problem is the problem.*

4.2 On completion and correctness

Knowing when to stop solving a problem, the assessment of a solution's correctness, and the 'borders' of problems are handled by no less than 5 characteristics; (2) "Wicked problems have no stopping rule.", (3) "Solution to wicked problems are not true-or-false, but good-or-bad.", (4) "There is no immediate and no ultimate test of a solution to a wicked problem.", (6) "Wicked problems do not have an enumerable (or an exhaustively describable) set of potential solutions, nor is there a well-described set of permissible operations that may be incorporated into the plan" and (8) "Every wicked problem can be considered to be a symptom of another problem.".

Software engineering projects are rarely set up for their own sake. Software is a means of doing, or at least supporting, business. On the other hand, software has its own influence on the way business is done: an automated system can track purchasing behavior of customers, or make more complicated reward-programs possible. More directly, e-business, the delivery of services and goods using the internet, has its influences on the requirements for doing business.

This means that software is a player in the field of forces that influence each other, together making up an organization. Consequences of this are requirements that may change regularly, a group of stakeholders with very diverse or even contradictory needs, and a theater of operation with changing bounds and rules.

Combining this with the inherent intellectual freedom a software architect has leads to a highly dynamic context. With equal material and personnel costs, many different tools can be used

for resolving the problem at hand, including both tangible tools, such as software development environments, and mental tools, such as description techniques or software paradigms. It cannot be predicted which tools will yield the best results. Furthermore, no tools exist which can assess the adequacy of a given resolution.

4.3 One shot endeavors

Another group of characteristics deals with constraints and the size of implication of a potential solution, comprising (5) "Every solution to a wicked problem is a 'one-shot operation'; because there is no opportunity to learn by trial-and-error, every attempt counts significantly.", (7) "Every wicked problem is essentially unique." and (10) "The planner has no right to be wrong.".

Given enough time and resources, it should be possible to devise an optimal architecture, being most acceptable to all stakeholders. However, innovative software projects have in common that they are on a tight time- and money budget, and have a potentially large impact on the organization(s) involved. So, possibilities for 'trying out' solutions are few; potentially beneficial, but risky, propositions cannot be explored.

While finding 'load-bearing influences' is one of the main traits of an architect, the uniqueness of innovative projects creates opportunities for important factors to stay hidden for a very long time. Choosing a formalism or direction of investigation early on in the project creates even more, as does the tendency of finding analogies to earlier projects.

5 Conclusion

From the statements above, I believe it safe to classify software architecture, or at least 'first iteration' architecture, as a wicked problem. Both ignoring this and taking it overly serious can have adverse effects: the consequences of ignoring have been shown above, but overestimating its impact can lead to a 'formalizing does no good, let's only trust our gut feel' attitude. A combination of sensible use of formalisms and the awareness that some details might get lost, and should therefore be checked regularly, seems to be a good way of 'resolving' these problems. This is nothing new; for example, [BH95] pleads the use of formal mechanisms, but warns the reader not to be blinded by them.

No matter the great tools used, software architecture is still a human activity. Instead of trying to factor this out, making good use of this is much more sensible. We should not try to develop techniques that makes human intellect obsolete. Instead, we should make use of the powerful possibilities the human mind has, and augment this with tools that allow a designer to focus on certain essential elements of the problem at hand.

References

[Bah75] Archie J. Bahm. Planners' failures generate a scapegoat. *Policy Sciences*, 6(1):103–105, March 1975.

[BH95] Jonathan P. Bowen and Michael G. Hinchey. Ten commandments of formal methods. *Computer*, 28(4):56–63, 1995.

[RW73] Horst Rittel and Melvin Webber. Dilemmas in a general theory of planning. *Policy Sciences*, 4(2):155–169, June 1973.

[Zac87] John A. Zachman. A framework for information systems architecture. *IBM Syst. J.*, 26(3):276–292, 1987.