On architects – a conservative view

A proposal for a cyclical model of programming and design

Vincent J. Mak
TU Delft 2013
The value of architects

Is the architect more than the combination of a number of separate professionals: e.g. graphic designer; building engineer; and process manager? This essay argues he is. Moreover, it argues that the design process is more than a succession of predetermined phases, and that with an architect actively shaping the process, using his expertise and communicative skill, the resulting project will gain analysis and quality and thus provide better buildings.

Critical reflection on the profession

The financial crisis, now going into its fifth year, puts a palpable strain on architects, in particular fresh graduates such as my friends. The bursting of the financial bubble in 2007 that caused a domino effect on the entire economy is intimately linked to bank lending and housing mortgages. So the construction industry, which did so well before 2008, has to endure some of the hardest blows. Architects seem to notice this more than anyone. The number of architects employed in the industry has halved since 2008, and the amount of work left for the remaining architects is still declining (Architectenweb.nl 2012aa; 2012bb). We always need to ask: What value do we add to our societies as architects? But in a situation where work is scarce and opportunities are dwindling, the question becomes urgent.

Cesal (2010) adds the following, more pragmatic questions: Why are architects getting marginalized in the building process, why do they get laid off more than any other building professionals, and why do both architects and non-architects seem to have some trouble delineating the architect’s area of expertise? Underlying these questions is the troubling observation that architects are not only getting laid off as a result of a shrinking market, but that they are being more fundamentally undermined: their role in the project team is questioned. No longer can architects rely on a universal understanding that they are required, in the tradition of Master Builders, to oversee and guide the building process. Management techniques and project protocols have taken over much of that, leaving the architect with design tasks in the strictest sense of the word: the shaping of things. I intend to show that protocols and management techniques are no substitute for the critical and complex tasks that architects do.

The crux of the problem, according to Cesal, is found in the architect’s inability or unwillingness to explain what it is they do exactly. Hence, they cannot communicate what their value is, and that means that others cannot easily see it either – which results in the marginalization and layoffs he observes. However, there is plenty of value to architects, as he goes on to show in 10 segments, each highlighting a critical function of architects. As an analogy, we could say that Cesal is describing all the parts of a car (the wheels must be soft, but durable; the engine must be powerful yet fuel efficient), without ever describing what the car is for. Not all clients need the fastest car or the largest one, or the one made of the toughest materials. More than making sure the parts are the most optimal, the architect needs to show how these optimal parts work together in one thing that works best for the client. Matching the demands of the client to what is possible to build is a complex process that goes beyond optimizing some variables, for reasons that I will attempt to expose below. It is in actively managing the complexity of this task that makes architects have a crucial contribution to any building project.
The type of complexity of programming

The unique value of the architect is to dynamically manage the requirements, which is on the one hand balancing and eliminating conflicting demands, which greatly overlaps the process formally known as programming, and on the other hand the architect is to discover what the client wants where he doesn’t know yet, which involves actively proposing solutions through designing. Programming and designing are not necessarily either opposite or complementary to each other, as one involves the other and vice versa. The terms refer to the different types of documents that are produced during the course of building projects, a program being more verbal, and recording more or less abstract wishes and conditions, while a design tends to be more graphic.

Many sources on programming have taken this difference in presentation as a difference in process, presenting programming as one or two distinct phases, strictly preceding design (White 1972; Preiser 1978; Palmer and American Institute of Architects 1981; Sanoff 1992; Spekkink 1993; Cherry 1999). The break between programming and design has been made most prominent in the proposal of Peña and Parshall (2001), where programming becomes a separate project, preferably executed by specialized programmers, rather than architects. In doing so, they try to reflect a project-based approach, where the task or “problem” is first thoroughly analyzed and then formulated in a “problem statement”, before any solution (design) is proposed. This solution can then be tested against the problem statement, and if it complies in full, the design is ready. This method has the advantage of being straightforward, and it allows a project manager to split up the tasks, and easily shelf the whole process between phases, because the process parts relate to finished products.

**fig.1.** Some representations of project processes where program and design are clearly separated, and have a unilateral relation, with all programmatic tasks always preceding design. Top left: Walter H. Moleski - 2 programming phases called “diagnosis” and “strategy”. Top right: William T. Peña - 3 phases, exhaustive method includes fieldwork. Bottom left: Edward T. White - 2 main phases and a pre-programming phase. Bottom right: Jay Farbstein - includes post-occupancy evaluation as a core task for programmers.
However, more recent publications acknowledge a more complex relationship between the processes of designing and programming, arguing that feedback loops and direct interdependency between the two are necessary for a good formulation of the project’s goals (Valkenburg 1998; Barrett and Stanley 1999; Blyth and Worthington 2010). The reason, they argue, is that the exact problem statement may be too vague or conflicting, which must be discovered and negotiated through iterations of design proposals. Indeed, the client may have a difficult time seeing the consequences of certain decisions until they have been worked out and visualized for him. Thus, the initial question, no matter how thoroughly researched, may no longer be accurate as the project continues to take shape: design produces new questions and possibilities that were inconceivable at the start. In fact, a program that is too thorough in its description in the early stages of the project may prove to be a problem for the architect: if the document is too big, architects report ignoring it, relying on personal communication instead (Heintz and Overgaard 2007). Architects prefer to-the-point programs, though often the point has yet to be discovered: again, it is possible that the real concept of the project is discovered through design.

The initial wishes of the client need to be analyzed, tested, processed, structured and even amended. This makes the whole process inherently complex and, in a technical sense, “chaotic”. A chaotic system can be produced when outputs depend on earlier outputs, rather than on inputs only, as in a project-based method. Chaos theory was first proposed to explain how small deviations in the input in a weather prediction system could cause an increasingly different outcome of the prediction the further on ahead they calculated (Johnson 2001). Though the name “chaos” invokes connotations of randomness and negativity, chaos theory in fact proposes that deterministic systems with some simple repetitive rules can cause wide ranging results when they have some kind of feedback loops, which are also at play in making architecture, i.e. step N+1 depends on the information gathered in step N. Given a certain set of transformative rules (or algorithm), a certain input will always result in a corresponding output. Hence, this outcome is not random at all, but guided by the application of the algorithm.

An important difference with the theoretical chaotic model, of course, is that the architects who manage the design process, and the tasks that they do, are not reducible to some simple algorithmic rules. They too can be influenced by the input gathered in the course of the project. The way architects respond to the information they have at any point, can be guided by their past experiences, expectations, skills, routine, intuition and so on, and is hence at that level as difficult to model as any human thought process. But there are some more general things we can say. For example that an individual architect tends to be at least somewhat consistent in his approach to a project: this is why portfolios are valued in most architect selection procedures. We can also analyze the methods architects use to gain understanding of the project’s relevant issues. Many books have been written about this, and I have collected a sample of the more creative techniques in the sub-process of programming. Understanding that these techniques were not restricted to any single phase allowed me to propose a reading of the design process in which feedback loops shape the course of the project itself, with the architect collecting all the data, performing reflection on them, and deciding the next step(s) on the basis of this new understanding. This is where the pivotal role of the architect becomes most apparent. Only when all the data – both externally gathered and internally generated – is considered in this reflection, decisions about both the goals and proposals can be consciously made. These decisions have to be continually made to be able to see effectively and efficiently what the next step ought to be, which is dependent again on the evolving goals and
proposals. The procedure is similar to the learning cycle described by Kolb (1984), in which understanding is also expanded in iterations. In fig.2. and 3. I have adapted the Kolbian learning cycle to a project-oriented terminology.

![Diagram](image)

*fig.2. Kolbian learning cycle in a (building) project*

![Diagram](image)

*fig.3. Kolbian learning cycle expanded to include some products and processes in a design process. Products and processes in every phase become the inputs for the next phase in the Kolbian cycle. Physical products, such as the program document, result from the processes that the cycle goes through. They can aid in accumulating knowledge, or serve as a record, or both.*

**The many tactics through which one may navigate**

During the programming and designing process, the architect needs to uncover what the project requires, but he also needs to find a structure in what he finds so that it becomes useful, and he needs to record those findings in a way that he or others may find it again. Furthermore, he will need to test the new findings against the old ones, and suggest a direction for further inquiry (which, again, can be through design or program), embedding it in a constant stream of increasing sophistication, through feedback and reflection. To illustrate that the individual tasks are non-
hierarchical, I produced a matrix of functions that programming can have. On one axis, I placed actions (what you can do by programming). On the other, I placed goals (what you can have by programming). Combining an action with a goal, yields tactics.

Actions:

- Initiating (starting contacts and ideas);
- Assessing (needs, wishes etc.);
- Planning (saying a path forward);
- Reporting (writing down what has been learned).

Goals:

- Structure (for ordering the information in some fashion);
- Continuity (for recalling previous decisions);
- Inspiration (for revealing new programmatic issues);
- Collaboration (for communicating deliberations to other building participants).

<table>
<thead>
<tr>
<th></th>
<th>Initiate</th>
<th>Assess</th>
<th>Plan</th>
<th>Report</th>
</tr>
</thead>
<tbody>
<tr>
<td>Structuring</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Continuity</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Creative</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Collaborate</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

*fig. 2. Matrix of programmatic functions*

By combining both lists point by point, I created 16 areas of possible tactics an architect could use to further the project, through the program. With the exclusion of the initiation phase – which is often for some reason separated from programming in the literature, I have collected some techniques and methods that fit well into the cyclical understanding of programming – and not so much in a linear model. I do not wish to propose that this is an all-encompassing guide – there are in fact thousands of good suggestions in all the books that have been written about programming over time, and it is no point to repeat those. The purpose of this collection of techniques is to give a number of the options one has at any point, to illustrate that there is no single path through them, and furthermore, that not every tactic is required for each project, which becomes obvious when one examines them more closely.
Insert: 4 sets of cyclical techniques

- 1 Structuring
- 2 Continuity
- 3 Creativity
- 4 Collaboration

The following pages detail the 4 sets of activities in the programming process that were used to get to the graph in figure 2, which formed the basis of my proposal for a “chaotic” cyclical system that uses feedback from the project’s content in order for the designer to decide what action(s) to try next. I will now include them below to give some examples of what a cyclical process could mean more or less practically. Each set corresponds to one of the “actions”, and each set has a number of sub-chapters that correspond to the various “goals”, though the content of each set and chapter can of course endlessly be expanded with other techniques and considerations – the examples I’ve collected are just some reference points in that category.

Vincent J. Mak
TU Delft 2013
Set 1: Programming is structuring

Programming is not only about collecting relevant data, in the same way that constructing a house is not only about collecting all the building materials in one place. Let’s say we have a load of bricks and plenty of wood and shingles, some concrete, a very large diesel engine, several windows and doors, and a bunch of wallpaper. In order to do anything with those, we need to think about how and when each thing becomes relevant if we want to turn this stuff into a house. We can use the concrete for a foundation on which the walls and floors will rest; then the bricks and doors and windows need to be stacked with a plan for where the entrance goes and where light must come in; the roof we can build from the wood and the shingles; and on the walls we can put a wallpaper that pleases us. This is all quite intuitive, if you have a small house in mind: a house has a structure that has been proven through the ages with the foundation at the bottom and wallpaper on the walls, and not the other way around. We also know to disregard the very large diesel engine. Though it might have its value, it is of no use in making a simple, standard house. The idea of the house makes us see where the materials should go.

In programming, one encounters wishes, ideas, conditions, possibilities, rather than building materials. And it is not immediately clear which of those are foundational, and which are not; which ones should be dealt with first and which ones can be incorporated later; what is useful information and what is not, etcetera. The architect needs to create a virtual plan of the structure of these ideas: to know for himself, and to convey to others, which ones are the “foundation”, which ones are “wallpaper”, and what is “very large diesel engine” – useful but irrelevant.

There is not a universal structure that applies to all projects: It depends on who and what you are structuring for. Besides organizing thoughts and concepts in some kind of structure, one can also think of structure in the sense of how to allocate time, and the process of designing itself. What do you need to know first, and when do you need to make certain decisions? Especially in larger, more complex projects one requires a more extensive kind of organization than in a small one: figuring out an appropriate process structure can save lots of time, effort and money. In this chapter, I will present several methods of thinking about structure that can improve the effectiveness of programming.

Structuring content [report function]

Architects usually do not design buildings for themselves. The drawings they produce have to be read and understood by the client, who will be using the building, and by the contractor, who is supposed to construct it. And there may be even more stakeholders in any given project. Each of them may contribute unique and indispensable advice to the architect, if they understand what they are seeing on the drawings. What they see there are lines with some annotation: it is unlikely that they will have anything to say about those, until they have some understanding of the program. For example: Is the foyer so large because it is supposed to be intimidating, or inviting, or does it have some other purpose? Perhaps it is used by many people at the same time? A good program should have something to say about these kinds of questions, because it relates to how people will use and experience the building on the one hand, and the required investment(s) on the other – a large foyer costs more money and space than a small one. But the program might also say something about fire escapes, air vents and electric sockets: If they are going to be part of the building, what (visual) quality do they have? This can quickly become overwhelming to anybody trying to understand the building at a basic level, which makes a shared understanding of the program difficult (Cherry 1999; Bogers, van Meel et al. 2008). Hence, clear labeling, a graphical hierarchy, or any of the other basic typographic tools can help the designer to make his program clear to any audience, which in turn can help the audience make it clear to the designer what they think should be kept or changed.

However, many programs are not designed to be shared. This tendency stems from the idea that the program is a direct translation of the wishes and conditions of the client. For example, White (1972) assumes a strict
one-directional linearity: The architect extracts or receives all the relevant information from the client in the form of the (written) program; then, the program is translated into a design; and the design has a one to one relation with how it is constructed. In this view, the program concerns only the architect and the client, and that invites a lot of private understandings, which make the program inaccessible to others. That is not necessarily a bad thing, but the designer should be conscious of that restriction.

If, however, it is decided that others will also need to be party to the underlying ambitions to certain designs and drawings, thinking about a structure may save both designer and audience a lot of work. Every stakeholder will read the program differently, and they will have a different interest. So as the diversity of readers increases, the amount of information they may require will increase, too. And the more specialized these stakeholders are, the more they will need information that has little relevance to others. Structuring is then about managing information.

Managing information does two things: First, it organizes information, classifying it, and bringing a hierarchy into it. At the same time, it should allow the reader to quickly find the required piece of information. Organizing demands can also help the architect see which are core demands, and which demands are more flexible (Bogers, van Meel et al. 2008), and it also allows them to communicate this understanding back. Understanding how the question is layered is really the starting point for any design.

To conclude: Imposing a structure on the program forces the designer to think about the ordering of needs and ambitions, and makes the document and the project more accessible to other stakeholders.

**Structuring fixed programs [assessment function]**

Sometimes, the architect is not involved in the creation of the client’s program document. If structuring is indeed a first step in understanding the project, having no influence in its creation clearly has to be a problem. Clients usually do not have the knowledge and experience to see which of their demands and restrictions are directly relevant from an architectural point of view, or how the layering may inform the project at the level of design. Nor is such scrutiny expected of them: this kind of experience is what they hire an architect for.

In the case where an architect receives such a “ready-made” program, we see that it has to be processed. They will often (partially) rewrite the program, or write a synopsis, or do initial design explorations on it, to “get the project in your fingers” (Heintz and Overgaard 2007). This nice phrase evokes the image of coaxing liquid inspiration from the pages of the program into the architect’s fingers. But looking at the types of activities these architects report doing, it seems they are doing more “earthly” things: producing little (graphic) experiments, storyboards, summaries, and then reflection with the client. It seems they are exploring the structure of the information: testing where key ambitions and key difficulties are; and echoing their findings back to the client to see if they interpreted it correctly. In an informal way, they impose a (new) structure on the information. I call this a shadow program. The official program, created by the client, is the starting point for the structured shadow program.

It may be helpful to present this new understanding to other building partners. It can save time, confusion, and disappointment to get other contributors to the project – and stakeholders – as much on the same page as possible, especially on the fundamental building blocks that are established in/by the program. Certainly in a participatory project, a clear presentation of the structure can be invaluable. The different participants are not to be expected to be involved in the project at all times, and they may forget how the challenge was understood. The program can help them remember the conceptual bearings that have been found in the previous iterations. Furthermore, as the number of participants increases, the chances that they are replaced in their function increases. A focused and accessible structure can help the newcomers come up to pace with the more veteran members of the team.
Concluding, a structure in the program works on multiple levels: First, it is a way to understand and prioritize the issues that the project deals with. Second, it can help to efficiently introduce other participants to the current state of the project. However, it is unlikely that the structure of the project remains the same. It is very likely that a certain design solution creates a new chokepoint or key issue. Therefore, as with all other parts of the program, the structure has to be seen to some degree as an unfixed thing that requires scrutiny as the insights progress. That means that a fixed program, which cannot evolve, and may not even have an effective structure that the architect/participants can work with, is likely to create unnecessary friction in a design process, and should be accepted as an integral part of the project with extreme reserve.

**Structuring programming [plan function]**

Much of the quality and depth of the program is dependent on the feedback of all stakeholders in the building project. Not all stakeholders will be freely available – sometimes involvement of the end-user may even be prevented by management (Farbstein 1993). For those who are able to contribute, it may be important to know (far) in advance when they need to reserve some time. But how do you plan ahead when the design process is a continuous cycle with no clear borders?

Common phase divisions (such as 4 phases: structural/schematic design, pre-design, definitive design and lastly construction drawings as in (Wiecherink 2011) or 7 in the newest RIBA plan of work (RIBA 2012)) tend to offer a framework for time estimations by breaking the process up into smaller parts. Based on an estimation of the project’s complexity and past experiences with similar projects, the time for each phase can be calculated, and appointments can be scheduled between periods of project development. Most authors choose to indicate the expected products for each phase. RIBA for example chooses to finalize the program document in step 2. The conventional structure usually assumes that every progressive phase includes more detailed information, starting from a zoomed out rough exploration of possibilities, smaller and smaller until the smallest details.

The grouping of these levels of detail into 4 progressive phases or 7 is rather arbitrary. Other authors have suggested 3 phases (Diagnosis, strategy, design. Attributed to Walter Moleski in Sanoff (1992)), or an infinitely expansive system of phases (System attributed to Gerald Davis in Sanoff (idem)). You could even have parallel tracks for programming and design, each with their own phases; or parallel tracks for specified sub-problems in design/programming (Working with multiple “frames” as in (Valkenburg 1998)). Often, the end of each phase is used as a sign-off point with the client, and each phase has a self-standing sub-result, as far as possible – for example a “preliminary design” with some specific demands. More sign off points means more strain on the architect, as he needs to make the drawings understandable for the other participants. Too few sign off points, on the other hand, may lead to the architect missing crucial feedback.

The reason for deciding that a specific phase will have some pre-determined products is part practicality and part habit. Architects usually have some assumption about how a project evolves. It is indeed a very common idea is that a project starts from the largest scale and then develops to ever smaller scales. That assumption means you can measure where the project is in its path of development by the scale at which it is currently being developed. There are other schools of thought, however, who wish to work at multiple scales at all times, or even from the smallest scale to the big scale. But then how do you organize your meetings and planning?

Spekkink (1993) offers an alternative. In his proposal, you start with a comprehensive checklist of things to do and consider. During the design process, you will try to address the things on the checklist, in any order. You can measure progress by how much of the checklist is done, yet work on the things that matter most at any moment, unrestricted by some external order of things. Furthermore, by keeping track of what issues on the checklist still need to be resolved, you can find out what areas need more immediate attention. This is somewhat how Peña (2001) approaches his checklist, though he of course restricts it to programmatic issues only: In the end, all categories need to be filled in, but the order of doing it is undecided.
Knowing how much is done, and more crucially, how much is yet to be done, is a vital part to planning future work, and the timely engagement of external experts. This can be done by dividing up the tasks into pre-determined phases, and completing those phases successively, but it can also be done more “organically”, for example by employing have-done and to-do checklists. However, when the latter strategy is employed, it may still be helpful to retro-actively label the products of certain tasks into groups, so that the information contained is easier to find.

RIBA (2012) "RIBA Plan of Work 2013: Consultation document ".
Set 2: Programming is research and continuity

Programming is part of any project, but documenting it has particular value to projects that are 1) complex, 2) long-running or 3) repeated over time (Palmer and American Institute of Architects 1981). That is not to say that writing down the particularities of a highly unique and relatively simple project is useless, but the further away we get from this, the more the storage and transferability of the information in a program becomes convenient and even essential. For example to prevent us from having to reinvent the wheel and hitting the same obstacles or to reveal what the considerations of the architect have been when he decided to implement this or that design feature that we are thinking of altering or copying. There are also other reasons why we might want to recall the original motivations, which I will discuss below.

Human factors [assessment function]

People are quite apt at producing patterns out of complex data and applying that to a project. However, they are generally not consistent at it: 1) it can be radically different between two people and 2) it can also be different in the same person at a different time. The first difficulty is therefore to come to an agreed understanding of the relevant data and assumptions between all the participants of a building project. This requires good communication, described for example in Design Through Dialogue (Franck and Howard 2010). The second difficulty is to sustain this understanding over time, so that decisions taken at one time will be consistent with the understanding and resulting decisions at a later time – and also to remain consistent with the consensus formed earlier within the group of participants, saving a lot of disappointment. The simple remedy is writing stuff down. “The program serves as a log, a memory […]” (Sanoff in Palmer 1981).

Besides simply forgetting how the project is understood between participants, it may also be that the specific people involved are replaced over time. The impact of the client organization on the programming process is further described by Farbstein (1993), but one thing we can intuitively understand is that an elected representative is more likely to be replaced during the course of a (long) project than someone investing in an expansion of his family business. In the case that some – or even all – of the participants are replaced during a project, a good program can facilitate two things: On the one hand, it should explain to the new participant(s), the previous understanding of what the project is about. The new participant can then tack on from there, without needing to go through the exact learning process that the others went through. From there, the program can also become a vehicle for discussion: the new participant may not agree with all the values that the previous participant held, and then a new consensus may need to be sought. The program is an active memory that records what the future of the project looked like before. Hence, when the client organization no longer agrees with that vision on the future, a new understanding needs to be agreed upon and recorded.

However, writing these considerations down serves a more immediate task as well: they can be checked for consistency with the other demands of the project, as Bogers et al. (2008) and Spekkink (1993) recommend. Writing things down can be a means of reflecting for the architect/programmer, and also for the client. Simply making an organized list of certain decisions can uncover inconsistencies that weren’t obvious in drawings, as well as vice versa: a drawing could prove that two or more statements that were written down are contradictory or perhaps redundant. Testing out certain considerations in different media can help the participants make decisions and get agreement.

Another advantage of writing the program is to circumvent psychological effects such as “recency error” or “priority error”, among others, in the architect. The human brain tends to prioritize the first and last issues that were brought up (Reason 1990), although those issues might not be the most important or urgent. Documenting all issues offers a level of objectivity. It allows one to test the design against the program, and base contracts on the basis of the challenges described. The program can become an evaluative tool during design (White 1972; Bogers, van Meel et al. 2008), or during occupation, as the program document tells you what the building is supposed to do, which you can then test for effectiveness. The latter belongs to the
category of Post Occupancy Evaluation (POE). These evaluations are useful for both architects (learning from the hits and misses) and clients (figuring out possible improvements to their real estate for the short term or the long term) (Voordt and Wegen 2005; Dursun 2007).

Adaptability [plan function]
If we take into account that the objectives of a program may change during design, and that design and program are interdependent (Blyth and Worthington 2010), the relationship wherein the program is the benchmark for design becomes somewhat problematic. Moving goalposts are not a reliable test. On the other hand, testing against outdated considerations is no good either. But we need to bear in mind that the program and the design do not change at the same time. If a design-iteration proves a more promising way ahead, it can be found that is incompatible with the program. One can then decide to amend the program.

This change needs to be argued and checked for consistency with the rest of the program. It may be advisable to have a small protocol in place for such amendments, for communicating and discussing the changes with the other parties involved at that point. Chiu (2002) describes a framework for such a protocol in the context of collaborative design, though it is not the only conceivable framework. However, without provisions that allow amendments to the program, while the content of the program is invisibly changed as the project goes on, one potentially creates a number of difficulties downstream. 1) It becomes very difficult for new participants to grasp how the design is understood by the other participants, because the expressed goals and considerations in the program do not match the design any more. 2) The program cannot be used to measure the effectiveness of the completed building in the program’s stated terms, which could make POE relatively arbitrary – after all, the most important function of programming is to pick the values that are relevant to the building, which are the same ones that you will want to test the building on. That is presumably why a number of architects including Farbstein and KMD (Sanoff 1992), put POE among the responsibilities of the programmer. Independent value assessment of the building may still be possible, but must be based on other terms, such as a comparison with similar typologies.

Proponents of “fixed programs” (i.e. a program that is predetermined and cannot be changed during the subsequent design phase(s)) have argued that the necessary leeway for inserting new values can be achieved by simply keeping to a level of abstraction or indeterminacy in parts where the architect is directly involved (Peña and Parshall 2001). Here, we see a different interpretation of the program than for example in Blyth and Worthington, who would choose to include much deeper programmatic information, along the entire design process. Abstraction can be good early on, but a building project is about producing an architectural form out of the abstract intentions. At some point, the program needs to be fleshed out with details on how to get from abstract intentions to physical qualities – and many of those considerations go beyond design(-drawings). These can certainly be decisions on the level of the program, and in many cases it is valuable to record them there, for reasons described earlier. A secondary programmatic document supplementing the document produced by Peña’s method may be the solution: this introduces a hierarchical structure to the programmatic information, with abstract and “vague” intentions at the top, and then more exact decisions pertaining to the actual design below, which can be continually tested against those intentions. As for determining an area of vagueness: the architect has an effect on all parts of the program, as the product, architecture, is one of form and functionality. If it wasn’t, it would be either sculpture or machine, and there are very few architects who subscribe to either idea, and probably even fewer outsiders (public; users). It follows there is no territory where the programmer has exclusive say: and the areas where it might be seen to be so probably have no place in the architectural program.

Research [report function]
In order for the program to remain relevant during and beyond project design, the architect needs to adapt it, as his understanding of the design task increases. By testing the program adaptations against the earlier body of project knowledge, and testing the design against the program, the program documents a growing understanding. Thus, it functions akin to science. Values, ambitions, and wishes are all laden with emotion and
subjectivity, which are not generalizable. However, what you do with them is, and so recording them makes them an objective basis from which new knowledge about the effectiveness of processes and strategies can grow. I have already mentioned the POE (Post occupancy evaluation) above, which can test how well buildings are at what they are supposed to be doing. Besides testing whether the design was good, this information taken together with the POE data of a lot of similar buildings may teach us which design features work in general, and which ones are more likely to fail. This argument goes back to the early days of architectural POE in the 1960’s. It continues: We could have more targeted architectural experiments, and analyze and publish the results, making any experiment much more valuable the architectural field and society at large. This may offset the cost of the risk in itself, and even diminish the risk, through the availability of much comparable and reliable data.

Set 3: Programming is creative

Creativity has many readings, so it is essential to define this word before discussing how programming can be it. Though we have to dig a little deeper than a dictionary definition (e.g. “the ability to transcend traditional ideas, rules, patterns, relationships, or the like, and to create meaningful new ideas, forms, methods, interpretations, etc.”) to understand how creativity is applied in architecture. According to Fitchett (1998), in architecture, creativity is often measured as the “often formalist response to a verbal discourse”, with an esoteric and philosophical language that alienates those outside the architectural profession. Though it may be true that the creative process is chaotic or mysterious, the problems Fitchett describes with that follow from the interpretation and language used to describe the product. It is a problem of interface.

A knee jerk reaction would be to try to re-educate architects on their use of language or force them to be stricter and more accurate in their wording. But this is a dead end. Rather than improving the verbal side in order to properly reflect the formal, we need to question the dichotomy itself between the verbal and formal. In practice, a verbal response to existing or newly generated formal conditions can be equally productive as the other way around, and thus it can be creative. Furthermore, the verbal and formal, and program and design, go back and forth. They are interdependent on each other, as Blyth and Worthington stated in the introduction to “Managing the brief for better design” (2010). In the following chapter, I will explore how programming can be more like “creative” design, rather than merely an administrative layer.

Creative techniques in programming [assessment function]

It is clear that the difficulty of programming is to make problems and ambitions appear. Some of these will be very obvious; others may be hidden or forgotten, as users found ways to cope with a certain lacking. Or perhaps some ambitions and problems are overstated or exaggerated. It is essential to tease these things out in the program. Many ways to do this have been suggested; most of them rather playful and (semi) informal.

For example, in Palmer (1981):

- Brainstorming – Wildly and uncritically generating solutions and ideas with an absolute emphasis on quantity. An evaluation process follows either in group or by the session leaders.
- Synectics – Similar to brainstorming, except the emphasis is on producing fantastic, ridiculous, impossible analogies that suspend even the most basic laws.
- Buzz/Rap session – A light version of brainstorming.
- Role playing – Having people play act relations and situations. Though this is perhaps more useful to explore human relations and conflicts, and therefore not often directly relevant to the architecture.
- Gaming – Enacting situations through some simple rules.

Many of these creative games have been developed to support idea generation and programming. Some gaming strategies are especially useful if there are conflicts: For example priority games, where people have to negotiate with each other, using some type of currency or points, to get the things they want to have included in the program and/or reach a consensus elsewhere. But games can also be used to study the priorities and assumptions of participants, for example in urban planning games (one of many examples is the BCN Formula Game developed at the TUDelft). For the architect/programmer, there are dozens of games to choose from, both online and offline* (Tan 2011). Their strength comes from the combination of fun and effectiveness. Games can be employed to “get the ball rolling”, or to get some group cohesion, though the best games invite unexpected or unpredictable situations from which the planner can extract certain values.

A close cousin of gaming is simulation. Simulations, either in real life, or through the computer, can be used to test what happens if one assumes one or the other thing. The most obvious example is of course the computer generated 3d model which allows one to experience spaces before they are built. Koutsabasis et al. (2012)
went a step further and allowed the client to walk through the building and interact with the design in real-time, by adding textual comments or graphic aids where they would like to see changes, and the architect could implement minor changes on the fly. Larger requests are documented and worked on outside the interactive session.

This last example is a very direct translation of the concept of reflection-in-action, which was originally explored by Schön (1984). He argued that this way of working is vital to architects in a broader sense, not always with the client present: architects often draw something on a suspicion, just to see what happens, and then evaluate this new situation, like the client does in the virtual simulation. What happened; what is its value; did we expose any obstacles? When the architect asks himself these questions, he will get different kinds of answers, than when the client responds to his “guessed” sketches, though both are valuable and may offer vital clues to the program, in a very direct manner. The new insights can be recorded, while the sketch may be abandoned. Here we clearly see a design act serving programming, which is of course very common in an iterative (design) process.

*see graphic at http://www.slideshare.net/ekimtan/treecitydesigngamesworldofcitycraft

**Programming and a technology of foolishness [plan function]**

Much of human behavior can’t be explained properly by assuming only a strictly consequential logic in subjects (i.e. the actor takes a certain decision because the expected outcome of that decision is the most optimal.) (Hudson 1999; March and Olsen 2004; March 2006). Even though the subject seeks an optimal outcome, it is often not clear which specific course of action will result in that outcome. The problem is dual: on the one hand it is very difficult to predict the outcome in a complex or chaotic situation, which makes the options hard to see. On the other hand, the desired outcome has to be a function of the available resources and vague (dormant) ambitions, in order to be achievable. For example, world peace is always a good ambition to have, but there is no set of architectural resources that can lead to that as a consequence.

In architecture, the outcome is the building plan. Programming is all too often seen as the intermediate step between ambitions and resources, and the finished design. Virtually all books written in the early days of programming, roughly between 1960 and 1985, describe programming as if it was its cousin in computer technology: a set of instructions that mechanically processes raw data in a way that the user (architect) can work with it. (White 1972; Preiser 1978; Palmer and American Institute of Architects 1981); and all of these methods fail to a large extent to capture the architect’s attitude (Latham 1994; Barrett and Stanley 1999; Heintz and Overgaard 2007). It seems architectural programming is just not that straightforward, with the procedure (or algorithm in computer language) depending on both the design and the inputs: which makes computer programming an unsustainable analogy.

Rather than predicting the future and drawing the result, a lot of the creative work is in fact working in the opposite direction. First, they guess a plan, and draw some of it; then they predict what will happen and how they can formulate the ambitions; and then they readjust the drawings based on what they have learned. This process is echoed in the research of Sarasvathi and Dew (2005), which examined just how successful product developers tackle a problem. They too begin with a clean sheet, and just the skills and resources they know. Their first step is to ask something along the lines of: “who am I, what do I know, whom do I know?” The next question is “what can I do?” which involves conjuring up a new customer who might not even exist yet, for a product that may have no clear definition yet. Based on this vague notion and target audience, they will expand their own capacity by calling up people who may be able to expand the product. Those who are interested to contribute to the project may have goals of their own, and extra means, which expand the original possibilities and perhaps also the target customer(s). This is called “technology of foolishness”, named thus to indicate the differentiation from rational decision making. It starts from investigating your means and possibilities; and then
you invent a problem you might be able to solve; which in turn defines the ambition level of the project, ensuring a realistic ambition, because it starts from the possible.

Architects intuitively have to do this, because a rational analysis starting from a complete problem assessment, as suggested in some of the older books on programming, will lead to listing a lot of problems that turn out to have no architectural solution. That is not useless, per se, and the client may thank you for doing a complete analysis of all their problems (or they may find you extremely intrusive), but it is not the task or the competence of the architect/programmer: Those things which do result in a new building design strictly do not belong in the architectural program. If you are committed to a complete analysis, let a specialist (corporate) analyst do it, and then let the architect/programmer extract the passages that may be valuable in thinking about the design solution. Otherwise, it is best to ask specific questions, and let reflective feedback loops progressively inform you.

The checklists that have been developed to make the analytical task more specific, for example in Peña and Parshall (2001), may help you gather some initial insights, but its value will diminish as you increasingly understand the project more during design. Again, even though the checklist may be exhaustive, we cannot expect the ambitions to be clear or the resources and problems adequately explored. The primary task of programming is not to map all the problems and ambitions, but to see which ones can be tackled by a specific design. A comprehensive checklist approach may be useful, to make sure that towards the end, you have covered all possible aspects of architectural design, but if insight is progressive and iterative, consider it a living document that may give you a clue to where you have to give some extra attention/ask some extra questions during design.

Programming is exploratory. Any early drafts of it, even if they are a definitive program version, should be treated as indications of where the project will go, if it is to fit with the exploratory nature of design(ing). A program with a legal/contractual status may be difficult to increasingly “ignore” like that during the course of designing, which makes them problematic. A time planning in which the program comes up for review a few times with the client, as suggested by John M. Kurtz (as described in Sanoff (1992)) may alleviate this inflexibility, while retaining the benefits of having a contract based on the architect’s merit in addressing the client’s stated issues.

A technology of foolishness approach can be visualized as such: It starts with an exploration of what the means of the client and architect are. The questions again are: who are we, what do we know, whom do we know, and then what can we do? Since an architect is involved, the first answer to the latter question will always be that we can design a building. But then: is there land; is there a budget or revenue that can fund the project; how much time is available etc.? To take the budget: In the rational approach it is a resultant of all the other actions if you follow its logic consistently enough. Simultaneously, the budget is a limit above which the project is disqualified. This puts the budget in a rather awkward position, and it makes it the locus of much debate and disappointment in the final stages of design, as the design has been made to meet analyzed problems, which then turns out to be too expensive. Last minute cutting may not be completely avoided by a technology of foolishness, but it is psychologically different: From the point of view of the architect, it is easier to accept that the means cannot sustain the solutions than it is to accept that the problems you painstakingly identified cannot be met due to a too limited budget.

Adopting this kind of attitude to programming is not (yet) conventional, so it is essential to inform the client how you see the process evolving (see also Set 4: programming is collaborating). Though in reality, at least behind the scenes, it seems not that much changes for the architect, as this is all inspired by current practices and observations. What changes is the formal presentation of information, with the recognition that it evolves over time, and not entirely predictably. Next steps can be delineated in the program, and discussed with the client to give some structure to both client and architect: what is important to explore next; how much time is this expected to require, and how does that fit in the larger scheme; who are we going to meet with and when;
etc. Here, practicality plays a role, as actors will need to schedule things. If an actor has very limited time, they may become a milestone object in time for the process – the process is means driven after all.

Hence, we can see that the act of recording the means and ambitions becomes a creative driver and planning structure for the design process.

The program as advertisement [report function]
Programming is at once recording what you are going to do, and a report of what you have done, though the difference between those two states is in the time during which the plans are realized. That is to say, we plan what to design, then we design it, and then the program is the record of what we were planning to design in the first place, but that is of course still the very same document. In the designed product, there will be inconsistencies and new findings, and we may adapt the program, and start the cycle again and again. But in the end, the two “counterparts” programming and design should be in tune, which implies equivalence.

However, programming and design have different qualities, though these qualities do to some extent overlap:

<table>
<thead>
<tr>
<th>Programming</th>
<th>Design</th>
</tr>
</thead>
<tbody>
<tr>
<td>More or less abstract; prescriptive</td>
<td>Concrete; descriptive</td>
</tr>
<tr>
<td>Mainly text</td>
<td>Mainly graphic</td>
</tr>
<tr>
<td>Usually includes information on all aspects of the design, from time planning and organization to demands about the form and materiality. (note: synthetic/integrative)</td>
<td>Information usually separated for different categories of receivers (e.g. details for contractors; computer rendered images for potential investors; etc.) (note: in a sense analytic/differentiating)</td>
</tr>
</tbody>
</table>

Perhaps the program could be used to advertise a project, to attract investors for example. In brochures we often see that graphics are much more heavily used, because they seem to work better at getting people to buy your “product”. Only when the customer is already interested in a product are they willing to read beyond a few titles and key words. This resembles the advertisement business, and yet many architects do not seem to be “selling” architecture as if it was toothpaste. When they give presentations about their designs in competitions and such, many tell a story that explains how the building is experienced, or used, or works in other “programmatic” ways that make the building stand out. We find that although graphics are very appealing, describing the qualities of even very simple images can be much more effective than trying to prove them in the image. The quasi-mythology of commercial giant Apple, for example, seems to be the product-industry analogy of that. Their product design or advertising graphics are not so exceptional in their markets, but it seems they manage to use them to evoke an experience – much of it through story-telling and the clever use of words. Perhaps versions of these sorts of invigorating mythologies can be written as part of the program, and then used to attract investors on the one hand, and inspire the project’s genesis on the other (as has been suggested in Heintz and Overgaard (2007) when recommending the use of suggestive prose in programming).

Set 4: Programming is a collaborative process

Building is a fundamentally collaborative enterprise, involving many people and disciplines. The architect has a complex advisory role, which involves listening to many stakeholders and figuring out a form that is achievable within certain constraints. On the one hand, there are the users and clients who will somehow profit from the building after it is completed; on the other hand there are building partners who can offer advice on constructing the thing, and there are people who will actually build it. This is not necessarily an adversarial position in which the architect must negotiate, though it can be as Tijhuis and Maas describe (1996). Their research, comparing Dutch and German contracting culture, shows that a trusting understanding between contractors and architects/clients produces fewer problems, and furthermore that an integrated building organization with as few parties as possible is the most efficient way to eliminate costly conflicts. (Note that this research precedes the exposing of the widespread building fraud practices in the Netherlands, so especially the first point may have to be taken with a grain of salt.) However this relation is shaped, the architect must translate the client’s wishes to the builders, and similarly, translate the builder’s demands to the client in terms of costs and benefits to the finished product. The program here serves as a tool for consistency and clarity between the various deliberations, as it records the core values and agreements. As the project moves from inception to completion, there are various aspects in which those two words, consistency and clarity, can be interpreted in the program.

The program as a schedule [plan function]

Though the program is usually described as to deal with the physical demands of the finished product, the date of its completion may be included in that as well. It is no use for example to finish the design of an office building in 6 years for a business that is expected to require the extra space one year from now. Hence, one has to plan ahead and make this final date central to a certain strategy. And planning time means negotiating people’s agendas, which in turn produces conditions for the project, which may be communicated to all parties in the program.

The simplest strategy for a time deadline is to say the new project has to be fully completed at that point. If it is a large project with many different actors involved, one may need to plan pivotal meetings and steer the project towards those. This is not necessarily recorded in the program, though certain appointments may be so critical to the chosen project path that it influences even the design. It may be that the timing of this appointment is not ideal, but the architect has to manage the capacity he is able to muster for the project. If the benefit of this particular appointment outweighs the possible cost of being too early or too late for example, it may be a strategic decision, and you may be able to circumvent some of the possible problems by strategizing around those. Scenario thinking as described by Blyth and Worthington (2010) could be applied to the design process itself to prepare design alternatives in case any likely problems occur.

So, managing key points in time can be one thing that is considered in the program. But when faced with a timetable for construction, such as the example above of a business that needs extra capacity in a year’s time, one could also suggest strategies for transition. In this example, the extra capacity could be constructed first and be finished within the year, and then a second phase adds the previous capacity to the new building. This type of strategy has a direct impact on the physical building, or at least the building method. This allows more time for preparation and hence more flexibility in the partners you choose to work with, which may have considerable financial benefits, if not also new opportunities for interesting design.

In the case that there is time, architects may consider spending some of that to research some intricate pattern or user’s attitudes regarding the new building or its location. The findings of this research of course are a direct input for the program, or it may provide invaluable inspiration to the design, so creating the time for this is seldom a bad idea. Again, the principle of weighing the cost of finding out versus the cost of not knowing has to
apply (quoted from Cherry (1999)), although it is immensely difficult to estimate the related value of something not yet known.

As I said before, these kinds of schedule arrangements do not necessarily have to be taken up into the program. But one must be aware that they may influence the project very much, as in the example above about the growing business. And if that is the case, it makes sense to include it, to make clear to all participants that this appointment or date is of key importance to the process – just as much as a budget or a specific (zoning) law can be.

The program for opinion gathering [assessment function]
I started discussing the time tables, and mentioned that certain appointments with certain people can be essential to a process. The reason is that each project has specifics that the architect needs to learn, before he can adopt his architectural skill set to the problem. Thus, the inputs are crucial, and they come from the other participants. From the user, he will get unique information about the expected use of the new facility. From the client, he may get information about the available budget and location. From the contractor, information about how they prefer to build things and what materials or systems might be suitable.

It is important to organize the information coming from the various stakeholders, as the amount may be overwhelming, and the content contradictory. Almost all literature seems to agree that untangling and organizing this information is one of the most central difficulties in programming. The purpose of doing this is to make the information workable, readable, consistent, and concise: that is, to present it as a coherent thing – or perhaps even as a narrative “story” (Palmer and American Institute of Architects 1981). Hence, it matters who you are presenting it to, and then what is relevant.

Peña and Parshall (2001) argue that this input gathering should be done in advance of any other work, and by a person with a strictly analytical gaze (as opposed to a designer’s view). They propose the programmer(s) use an extensive matrix to gather all relevant variables of the architectural “problem”, and they include in their programming schedule a 1 or 2 week “squatting” tactic that involves being either on-site or in the existing facility to examine the conditions up close. These may prove very useful tactics to get a whole range of impressions and facts in a relatively limited amount of time, though the crucial step is of course after that: organizing and representing what you have found.

Peña and Parshall propose the programmers evaluate all the factors they identified and distill the most pertinent ones, which will be presented preferably in a simple graphic. This, at least in theory, is a tactic for collaborative practice, because he assumes that architects prefer graphic representations to fire their (graphic) thought processes, and thus the end of his problem seeking process connects with the next phase, which is that of design. However, this assumes two things, namely: that architects are graphic thinkers (which does not seem to be based on much), and that drawings are an adequate means of translating often abstract and complex dilemmas. Hence, I trust that the extra documentation they provide will contain a clearer account of their findings than just the graphics, though it is obviously a good thing that they attempt to consider the audience. The advantage of drawings is that they leave room for interpretation, and that is a good thing in the creative process; but it is a problem perhaps when you are trying to convey exact findings. Another problem with a graphic program may be that it is only accessible to graphic thinkers (i.e. architects under their assumption), which perhaps makes it a document with limited value in the later process. This doesn’t have to be a problem if you take the view that the design process is neatly sequential, a view that is made more explicit in White (1972), who describes the various stages (program, design, building), as translations of an original demand of the client and/or users. But if these phases overlap, as they seem to do to some extent, then other actors, including the client and users, may be interested to learn the underlying considerations that are stated in the program, and perhaps challenge them, which asks for a more accessible program (see also Set 1 on structuring the program).
There is a concern with people challenging and changing preferences and considerations that they or others stated earlier, namely, that it costs time to implement those changes, and that this time might not be accounted for in advance. Furthermore, constant changes of assumptions may demoralize the designers and frustrate the entire project. Especially near the start of the project, views are expected to change often and sometimes radically, as the implications of the assumptions become clear (Barrett and Stanley 1999). A designer should be expecting this and be apprehensive about sketching an all too detailed plan while programmatic demands might still be crystalizing. But design sketches or other exploratory methods may assist very effectively to make clear what kind of consequences certain decisions may have. An example from Barrett is to visualize the size of a proposed office space by taping the measurements down on the floor and putting some furniture in it. Choosing the right visualization techniques may help them understand us, and in that way provide feedback that helps us understand them – which goes for users and clients as much as constructors and specialists.

**Strategies for cooperation [plan function 2]**

I have talked about gathering information from other stakeholders, who are usually external to the design process. However, the same problems of sharing information and organizing tasks and resources apply when multiple actors, primarily designers, are working on the same project at the same time. In any more linear idea of the design process, a clear problem description can be used simply to divide up the tasks and work towards this static set of problems and sub-problems. The program then becomes the organizing principle for the organization of the collaboration. However, as the original problem will change character as more information will become available in time, and as a result of the design actions, disappointment is to be expected. Some actors will be aware of the change, while others will work towards a solution that has been rendered obsolete or problematic by the shifting goal. However, with the increased speed of communication and the variety of methods to do so, Chiu (2002) suggests it is possible for a team to have dynamic tasks and goals, with the right framework for actively sharing goals.

Chiu states: “‘Structured collaboration’ refers to a team within a collaborative framework for sharing the same goals.” He then goes on to analyze the function, frequency, and processes of communication in collaboration; how tasks can be distributed; and the role of computer technology. He concludes that the “current or collaborative ‘groupware’ technologies (such as shared whiteboard, email, internet phone, desktop conferencing systems and general-purpose software) are inadequate for the particular needs of simultaneous, multi-user discussion and co-production of architectural document[s].” As a solution, he suggests emphasis on shared protocols of interaction that are independent of implementation and storage schemes. That independency is the key to maintaining project integrity in a team, while the project goals may be shifting. And that importantly frees the program up from its organizing role in a collaborative project.

Chiu’s framework model consists of 2 distinct modules: A Communication Module, in which designers share themes and ideas with each other and coordinators, is at the base. These coordinators monitor the total workspace, and bring whatever is produced in accordance with the design tasks, which are located in the second module. This second module is called the Modeling Module, and deals with representations. The Modeling Module has two levels. The design tasks at the bottom level of the module are connected to a top level called design information, which is information captured for example in DXF files, text files, and raster files. So in Chiu’s framework model, there is a dialogue between communicating and reflecting actors in the Communication module, and representations in the Modeling Module. Adaptations may go back or forth, depending on the interpretation of the coordinator: so either the current tasks are adapted due to new insights coming from the designers, or the designers will adapt their work to the task.
It is counterintuitive in this model that the designers are separated from the design products by four levels of negotiations, but this should not be read as a design framework. The designers are presumably directly involved in changing and updating the design information. The framework described above works parallel to that, like a Voltage meter, keeping track of what everybody is doing and steering where necessary. Because this framework is supposed to work with computer technologies, it might give coordinators extra tools to reflect on the design process as it is happening, which circumvents some of the “distance” problems of reflection-in-action that are shared by analyzing own design activity: that is, being unable to critically reflect on the progress of the design process because you are in the middle of it. (Pedgley 2007)

End of insert.
Modeling programming in practice

So rather than following a predetermined route through various phases, going through the tasks in some described manner (fig.3.), the path the architect will follow is constantly based on the inputs he is getting from each current position. I have indicated decision points at the circles in fig. 4, at which the architect must decide where to focus attention next. He will try out tactics, probably based on what has worked for him in the past, or inspired by new examples, such as in Barrett, Blythe, Cherry, or any other seminal work. Executing those various preferred actions, he will gather new information, which will help him determine the next step. We find a similar approach in the Action Research method (Susman 1978), though in design, the "action taking" will usually be in the form of making a proposal or committing to a plan, rather than perhaps implementing some type of new policy. The evaluation, learning, and action planning (or framing (Schön 1984; Valkenburg 1998)), however, still apply to the new state of the design project, which is made "real" through visualizations such as renderings or stories. We could draw the key moments (in this case imaginary) on the tactics matrix and connect subsequent decision points with a line. Because of the lack of structure in the order in which an architect can progress through a project, the line will seem messy, though in reality it is informed by the feedback within the project, which means that the content (or lack thereof) will drive the investigative process.

Because the architect steers the "path" of the project on the basis of content that is constantly being added, the path will look chaotic, but it will not be random, firstly because it will be steered, and second because there are some general principles in design that are useful conventions and practices. For example, though he may revisit the site plan (the largest scale) at some advanced point in the project, it is generally considered a good idea to start from some assumptions on this scale and work down. Even more logically, setting up a rough timetable, or establishing the scope of the project makes extremely little sense at the end of a project. Hence, I attached some likely labels along the path of some physical products that might roll out at that stage. Stretching the path out...
along a horizontal line, we get fig.5. We can compare that path with the RIBA plan of work (2013) (See also fig.6.), which also focuses on physical products, to see some overlap in the succession of products, roughly based on the principle of working from a large, global scale to a very detailed small scale.

*fig.5. Same line as in fig.4., but stretched out.*

*fig.6. The path of fig.5. (left) compared to the RIBA plan of work 2013 (right). Various phases may still be recognized in terms of physical products, but the exact procedure is dependent on the content of the data gained, and the decisions of the architect on how to proceed with that.*
Recommendation

To test if the “complex” or “chaotic” model of the programmatic process indeed works as proposed above – with feedback loops enriching the content of the program (and therefore the design also), while at the same time informing the choice for the next pragmatic technique(s) to be applied – one might suggest to plot out the techniques used in several different monographs of a design project onto the graphic table such as in fig.2. These are preferably projects from the same architect, to see if indeed the feedback changes his design process “path”. To do one of them for this thesis may have illustrated much better how this would work, which I now had to infer from different literary sources. Unfortunately, I have been unable to find a suitable reference work.
Conclusion

The shortest possible conclusion from this research would be to say that architects seem to have found an effective, intuitive way to integrate the task of programming into their design practice. However, the interesting part of this research was to uncover how the modeling of this process has transformed from a straight linearity (i.e. initiate -> program -> design -> build) in the 1970s to 80s, to a more complex, rich model involving feedback loops and interdependency, especially in the programming and design phases. This understanding seems to have evolved in the 1990s, but it may have simply been a reaction against the rising popularity of the models that emerged in the decades before, which ascended along with project management strategies that focused on setting progressive goals and then achieving them, in that strict order.

The model of a simplistic project management strategy with predictably progressive phases based on a linear understanding of the process of design is hardly possible to achieve in architecture, as Barrett shows, and he then proposes different approaches to improve the workflow. I have done a somewhat similar thing in this research by collecting a number of (creative) proposals for how to make the programming process more cyclical, allowing the architect to learn more about the project they are working at. A cyclical approach, rather than a linear one, means that some steps can be skipped or repeated in any given cycle, as every “phase” can be revisited. From a linear point of view, this seems highly inefficient, because it can be seen as taking equal steps forwards and backwards – never getting ahead. But in fact, the cyclical process is similar to a Kolbian learning cycle, which is a very effective way of both gaining insight, and at the same time applying it – which can’t quite be said about the linear method that puts all data acquisition and question finding before the application and testing and therefore opens the possibility of missing essential bits of the complex assignment, which can be in the form of feedback from other participants, or feedback from the design products themselves.

Another important distinction between the linear and complex process models, is that linear models place certain activities at certain times (most strikingly in Molesky (Sanoff 1992)), whereas a complex cyclical model such as the one I propose in this research lets the order of activities depend on the output of previous cycles. That is, after every cycle, it is possible to alter the upcoming activities somewhat. There is of course then a danger of a process that is out of control, but again, in the “sets” included in this research paper, I have collected a number of strategies to keep sight of strategic (sub-)goals within the project. It should be noted that a number of the tools used by, for example, Peña, an advocate of a very linear system of working, can also be of use in a cyclical model. One of those tools is Peña’s checklist that allows a designer to keep track of which issues still need attention, or his emphasis on using appropriate levels of detail for your different audiences.

In fact, most of the literature on programming focuses on these sort of tools and techniques – though it seems the framework of the process has only recently been starting to be described. The theories of “reflection in action”, “action research”, “technology of foolishness” and the Kolbian learning cycle appear to provide a good basis on which to build a specifically architectural model of the programming process: these have in common that they see the interaction between process participants and products as an important input itself, and they also allow for a reversibility of question and solution, which are both certainly essential for arriving at a thorough architectural program. So the apparent complexity of the process is very valuable, because it results in a good
product: A good program that asks the right questions and demands the right values for the project, which in turn makes an architect a conscientious one in doing any project.

And yet, it appears that many architects do this intuitively, probably without reading any of the more recent academic publications on the subject of complex programming and design processes: they adapt their strategies to the specifics of a project, and cycle through reflection and experiment etcetera, which is possible to do without so much study because the cyclical process, although difficult to model and graph in specific detail, is in fact relatively simple to execute for any single agent who coordinates and manipulates the various inputs that drive the cycle – Which brings me back to Cesal and Down Detour Road. Architects do not seem to be able to explain what it is they do. Cesal, in his book, felt he needed to explain how architects can benefit a client in a manifold of ways. Perhaps that is not quite enough: When architects understand how they are essential to shaping and controlling the design process in its complexity, they should be able to explain how they are necessary to making the right building for any client.

**Sources:**


Susman, G. I.,
