Dichotomic Architectural Patterns in Systems Engineering

Gerhard Chroust
Institute for Systems Engineering and Automation
J. Kepler University Linz
4040 Linz, Austria
e-mail: gc@sea.uni-linz.ac.at

ABSTRACT

When conceptualizing a complex system, especially a so-called 'wicked system', its designers have the problem to make initial architectural decisions without having sufficient in-depth information of the final consequences. Such decisions usually entail a choice between basic alternatives (called here *dichotomic architectural patterns*) like "centralized versus decentralized architecture", "optimistic versus pessimistic system behavior", etc. These decisions pre-define the basic system architecture and are very difficult to reverse later. Identifying and isolating these dichotomic architectural patterns and describing their basic properties allows to teach these principles. This helps system designers to understand their options and the resulting consequences for the envisioned system.

In this paper we describe essential dichotomic architectural patterns and typical examples. We classify them according to underlying principles and discuss their commonalities.

Keywords
System Design, dichotomy, dichotomic architectural pattern, design decisions, trade-off

1. DICHOTOMIC ARCHITECTURAL PATTERNS

Today's information systems show a continuous growth in complexity, due to the need to solve what Hermann Kopetz calls "wicked problems" [1], an extrapolation of Lehman's Environment Systems [2]. Besides being large, complex, ill-defined and without clearly identified objective. These systems have the property that *the problem cannot be specified without some concept of its solution*.

Starting from the wrong assumptions not only will lengthen the development process. Under today's pressing economy of software development together with the stringent time-to-market needs, this often will mean the end for a project.

It is therefore necessary, even in the case of insufficient information and badly understood interactions between features to understand the effects of the a-priori basic decisions: *one needs to make some a-priori assumptions/decisions before even starting to conceptualize the system's specification*. Designers have to rely on their over-all knowledge, experience and intuition in order to create a reasonable initial solution which will later be refined and modified.

In this paper we suggest to identify 'dichotomic architectural pattern' which are the basis for such decisions [3][4]. The finally chosen decisions with respect to such patterns can be seen as a point in a continuum between two abstract, though extremal endpoints. A simplistic expression of such know-how is the popular saying *"You can't keep the cake and eat it"*.

A typical (still simple) example is the design of a programming language: one can either design a language for being compiled or being interpreted. This has considerable consequences on the syntax and semantics of the constructs of the language.

In the field of engineering we often find trade-offs between two opposite, often contradictory alternatives for a design: e.g. you can design a system for extreme flexibility or for extreme security. usage this decision is not optimal any more Internet (based on the original DARPA-network was designed for minimizing the effects of severing connections. This decision now causes today's problem with security but this cannot be remedied easily.

Many of the dichotomic architectural patterns, however, are not symmetric: *you can keep the cake today and eat it tomorrow*, but not vice versa. Furthermore in real life a compromise between the extreme positions can be taken: e.g. *you can eat half the cake and keep the other half*.

The cake example is typical for a large class of such patterns: one can perform some action now or later (sometimes only by using a different technology): *"You can cross the frozen lake now or you can cross it in half a year's time"* - but by some other means (e.g. *swimming*).

Each dichotomic pattern has certain characteristics attached to it: cost, implementation time, security, etc. and thus implies the basic behavior of the system to be built. Once such a decisions is made, it can be modified but basically not changed without redoing most of the design.

Discussing the extreme endpoints of such a dichotomic architectural pattern allows to teach their basic properties and thus helps to develop a 'guts feeling' for engineers obliged to make such decisions.

We can make several observations with respect to these patters:

1. The choice has strong, often irreversible influence on the characteristics of the final system, e.g. *once eaten, the cake is gone!*.
2. Many of the patterns are not symmetric, e.g. *you can keep the cake today and eat it tomorrow*, but not vice versa.
3. In the final system some compromise will have to be chosen, e.g. *eating half the cake and keeping the other half*.
4. If circumstances change, a different choice could be better, e.g. *keeping a cake if one is not hungry*.
5. The wrong combination of several such choices might result in a suboptimal system, e.g. keeping a cake and many biscuits.

In chapter 2. we describe characteristic dimensions of these choices. In chapter 3. we show that some of the given examples can be classified into different patterns. In chapter 4. we discuss the interdependence and consequences of the choice of dichotomic architectural patterns.

2. DIMENSIONS OF THE DICHOTOMIES

Several dimensions can be distinguished:

Enactment time WHEN should a foreseen action be performed?

Synchronicity WHAT TEMPORAL RELATIONS do actions have?

Physical Location : WHERE should needed data and programs be placed?

Granularity of Access : HOW MANY/MUCH should be accessed/handled in one step?

Communication Control : WHERE should control be located?

Risk expectancy : HOW PROBABLE is a certain event?

The above dimensions seem to be the most important ones in current information systems, but a closer research will probably reveal more.

2.1. Enactment Time

One of the essential decisions for an action is as to *when* it should be enacted, assuming that it has to be enacted - sometime. Two extremes points can be identified:

anticipative The action is performed as early as possible, as soon as it is identified as necessary, or at least probable. We speak of "work ahead"[5]. This usually provides more time to devote to the action, allowing a more thorough work, with the uncertainty that the work might be unnecessary.

just-in-time The actions has to be performed immediately in order not to risk some other problems of performance (e.g. real-time constraints).

We know that not every action has to be performed immediately after its identification. It may be more advantageous to *delay* it. some Major reasons for delay can be:

- During a time-critical action no time should be "wasted" for lower-priority tasks, especially if there will be some slack time later to do the work.
- At a later time there might be more accurate information available to perform the task.
- Resources might be cheaper later.

We can distinguish two variants of delay:

- The actions can be performed later using the same means and methods
- The delayed action needs different means and methods.

2.1.1. using the same method

Prefetching versus just-in-time fetching of values High-speed computers prefetch data e.g. in order to avoid costly page faults later. This entails the risk of being sometimes wrong and having to throw away some work.

Binding of Variables Memory space has to be allocated to a program's variables ('binding of variables'). Allocation can be done *before* the program starts ('statically'), using considerable memory space, even for variables which might not be used at all. Alternatively binding can be done 'dynamically' when storage for the variable is actually needed at the price of perhaps valuable (perhaps critical) time to be lost.. Binding can also be done at several convenient points in time: at compile time, when the

programm is linked together with other programs or when the program is loaded into the computer system [6].

Compilation/Interpretation Computer programm written in problem-oriented programming languages have to be *translated* into the primitive control statements of the hardware. This can be performed in two ways:

- translate the *complete programm* at once and execute it later (*'compilation'* [7][8]) or
- translate *one statement* (a 'minimally meaningful expression of the language') at a time, execute it and then translate the next statement (*'interpretation'*) [9] [10].

Both compilation and interpretation have their specific advantages. The basic trade-off is whether one invests considerable effort beforehand (writing a compiler and executing the compiler to translate the programs) or one is willing to expend some small amount of effort each time a higher-level language statements is executed.

2.1.2. using a different method

Computation of functions versus table look-up In order to compute mathematical values (e.g. trigonometric functions) one can pre-compute and store all necessary values in a table (needing considerable memory) or compute the requested value whenever needed - using considerable ad-hoc time.

Storage Management versus Garbage Collection During operation the computer system has to provide storage space for data. Data not needed any more and occupying memory can in many cases be removed immediately when they have ceased their useful existence or they can be collectively removed at a later time [11].

Systematic archiving versus search Retrieval of archived information and documents is one of the major problem of bureaucracies. One can either perform elaborate, time consuming indexing operations aforehand or store the data essentially unordered and un-indexed, but develop elaborate (and fast!) search engines.

Formal specification and verification versus testing When trying to establish the correctness of a program basically two approaches are (within limits) available: At the time of writing a program one can attempt to ascertain its correctness by formally defining and verifying it (which is very expensive and theoretically not fully solved) [12][13]. Alternatively one can try to find and eliminate errors in a program by validation and verification *after* the production of the code, by inspections [14] and/or testing (which is very expensive and theoretically not fully possible).

2.2. Synchronicity

With respect to the time relation between several parts of a process also a basic distinction can be made.

Parallel versus linear A task can be executed in a linear fashion or alternatively split into several parallel streams to be executed in parallel [15], often in a networked fashion. Its advantage is a major increase in speed, permitting larger computation tasks in reasonable time (e.g. 'grid computing' [16]) but at the cost of increased complexity and cost of synchronization [17].

2.3. Physical Location

The placement of data or work is a critical decision which has strong impact of in future performance of information systems, but often without having good data on amount and access requirements of future data.

Local versus remote computing In computer networks programs can be stored anywhere. Execution of such programs can either be done in the remote environment ('remote job entry') or programs can first be fetched in order to be execute them locally (Fig. 1). The advantage of local access has to be compared to the effort to fetch programs and keep redundant copies. Remote data can be made accessible either by having a pointer or a link to the remote location or by bringing the data physically to the user (imbedding or copying) [18].

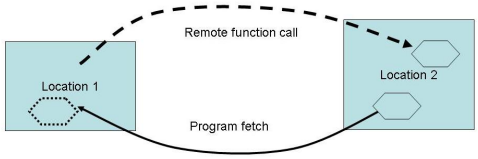

Fig. 1: Remote function call versus programm fetch

Vertical migration In a modern computer system with its multiple architectural levels of functionality (cf. ISO's Open Systems Interconnection Reference Model of ISO/IEC Standard 9834) actions (programs) can be performed on different levels. Transferring a function from one level to another is called 'Vertical migration' [19], cf. Fig. 2. This offers

certain trade-offs: programs near to the hardware level execute very fast, are safe from many security attacks but are difficult to implement and to change. On the user-oriented levels programs can be written, understood and changed much easier at the price of vulnerability and (comparative) slowness.

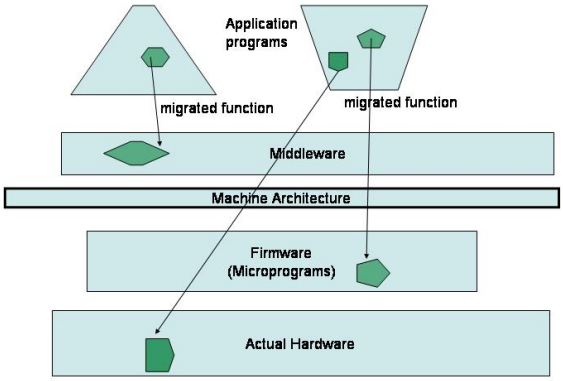

Fig. 2: Vertical Migration

2.4. Granularity of Access

One can fetch (from some storage medium) either exactly the needed items or also some 'neighboring' items in the hope to need them later (Fig. 3).

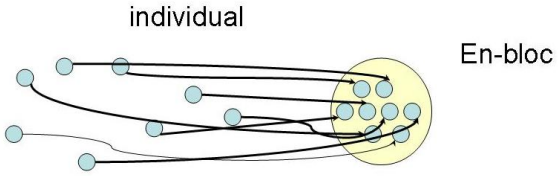

Fig. 3: Granularity

Accessing multiple remote data When operating on remote data one can copy/transfer the complete data set or a large portion of it from the remote location to the location of processing, knowing that only a fraction of the data will actually be used and at the same time perhaps excluding other users from use. Alternatively one can only fetch the needed minimum, knowing that more re-fetches will be necessary later resulting in over-all higher effort. Paging systems are based on the former strategy.

CISC versus RISC In the past considerable speed differences existed between the circuitry of the central processing unit (then in semiconductor technology) and main memories (then still using considerably slower technology like magnetic core and discs). The appropriate strategy was to fetching/store a large amount of data during each computer step in order to conserve fetch time. As a consequence powerful and complex instruction sets were needed for handling the large amount of data ("*C*(omplex) *I*(nstruction) *S*(et) *C*(omputer)"[20][21]). When memory technology achieved the same technology and speed as the central processing unit one resorted to "*R*(educed) *I*(nstruction) *S*(et) *C*(omputers)" with simple, less powerful instructions. RISC imply complex, time-consuming compilers due to more a larger semantic difference between the instructions of the problem-oriented programming languages and the (less powerful) instructions of the RISC hardware [22].

Modularization For dividing systems into components two opposite strategies can be adopted: division into the smallest possible modules, which usually are easy to implement and handle, but entail the problem of resulting in numerous modules and therefore many interfaces, increasing the cost of communication. Alternatively dividing the system into a few large subsystems which in themselves are complex but provide simple, easy interfaces. Usually the complexity of a module grows overproportionally with its size, inducing a small modules, but the overproportional connection complexity consuming the gain from small module size (Fig. 4) arises.

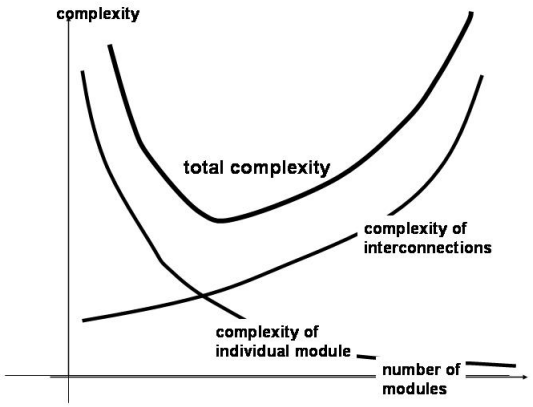

Fig. 4: Tradeoff between number of modules and size of modules

Locking Granting unique access rights to a collection of data [23] can either be done in a coarse-grained fashion, easing administration, but creating a bottleneck situation and preventing many other from access. Alternatively one can provide access rights to single elements, making both administration and simultaneous access to several elements more complicating and increasing the danger of deadlocks (cf. the transaction concept [23]).

2.5. Communication Control

Actors in a multi-actor environment usually need communication and coordination. A key decision is about who is in control of this communication.

Centralized versus Distributed In today's networks control can either be located in a single location or distributed over many (Fig. 5). Different properties are associated with these choices, typically ease and speed of control and update. Lower access costs favor distribution, while questions of consistency and networking costs favor centralized solutions

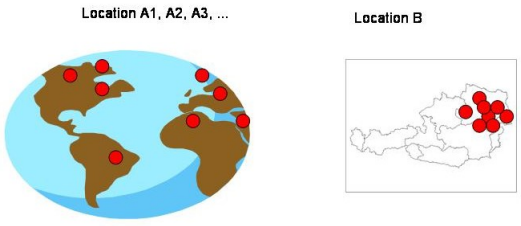

Fig. 5: Work Principles related to Space

Polling versus interrupt A classic decision with respect to a communication protocols is who should be the initiator of a communication. The requester(s) can periodically *poll* the supplier(s) about the availability of data or the supplier(s) can offer this information via *interrupt* (Fig. 6). Polling is easier to program and - seen from the requestor - allows for more control, but cannot cater for preemption of tasks. This is necessary in real-time systems and in systems with an arbitrary large number of suppliers. Interrupting another process causes more administrative work, is more difficult to program and more error-prone.

Fig. 6: Polling versus Interrupt

Master/slave or democratic Assigning control to one communication partner is usually simpler and straightforward, but entails potentially a bottleneck and makes the system more vulnerable to failure or unfairness. The democratic case is more difficult to program due to certain anomalies [24].

2.6. Risk expectancy

Error prevention versus error detection The expected probability of an error to take place (e.g. 2 users updating the same element, a hardware error to occur) can be evaluated in an optimistic or pessimistic fashion. Assuming everything to run well will induce to rely on ex-post repair (at unknown cost) while the pessimistic view (e.g. assuming Moore's Law) will invest a-priori in an attempt to unearthen and prevent problems (cf. 'foolproofing' [25]).

Check-out/check-in versus consolidation When working on some common objects (e.g. individual module designs within a larger development project) parallel changes of the same detail cannot be accepted (cf. the transaction concept of data bases).

One can either use a locking mechanism and give access only to one user at a time ('check-out/check in') or all users can access the data base at the same time. In a subsequent 'consolidation process' potential inconsistencies are detected and a clean-up is initiated. Commercial databases prefer the first approach, while software engineering environments often apply the second approach in order to avoid the bottleneck at check-out time, optimistically assuming a low probability of truly conflicting changes [26].

3. OVERLAPPING PATTERNS

Many of the dichotomies presented above can - if looked at from different view point - also be classified into other categories: typically 'Local versus remote computing section 2.3. could also be seen as a distinction between different ways of communication control. The table in Fig. 7 shows some of these overlaps.

dichotomy	Enact. time (same m.)	Enact. time (diff. m.)	Syn-chron.	Phys. Loc.	Gran. Access	Comm. Ctl	Risk exp.
Prefetching vs. just-in-time fetching of values	X						x
Binding of Variables	X						
Compilation/Interpretation	X				x		
Computation of functions vs. table look-up		X			x		
Explicit Storage Management vs. Garbage Collection		X			x		
Systematic archiving vs. search		x		X			
Formal specification and verification vs. testing		X					x
Parallel vs. linear			X			x	
Local vs. remote computing				X		x	
Vertical migration				X		x	
Accessing multiple remote data	x				X		
CISC vs. RISC					X		
Modularization					X	x	
Locking					X		x
Centralized vs. Distributed				x		X	
Polling vs. interrupt		x				X	x
Master/slave or democratic				x		X	
Error prevention vs. error detection							X
Check-out/check-in vs. consolidation		x					X

Fig. 7: Relationships between dichotomies: 'X' major category, 'x' minor category

4. SYSTEMIC INTERDEPENDENCIES

In a real system several of above dichotomic architectural patterns come into interplay. The following observations can be made:

- The individual choices are not independent from one another but have considerable cross-influences, impacting other choices.
- Each of these dichotomic choices directly or indirectly influences several cost drivers (development time, execution time, development cost, future orientation, sustainability, etc.) and their totality therefore has to be chosen in the light of optimization of the total system.
- The totality of the chosen dichotomic architectural patterns contribute to the 'next higher' function (cost, complexity, implementation effort, ...). In systems design one has to optimize these 'higher' functions. A typical example is the division of a system into components and the resulting complexity, cf. 'modularity' in section 2.4. and Fig. 4. Reducing the size of the components reduces the complexity of the individual module at the price of increasing the complexity of interconnections. Due to the overproportionality of the influences a picture as in Fig. 4 results, which has an optimum somewhere between the extrems.
- Any optimization has always to consider the *whole* system and avoid suboptimization.

5. Summary and Outlook

In this paper we have introduced the concept of *dichotomic architectural patterns* of systems design. They are important in the very early conceptual phases of systems design, especially for so-called wicked systems. For the designing engineers it is important to intuitively grasp the consequences of these basic tradeoffs in order to be able to make stable and sustainable initial assumptions for the design to be created and avoiding any crucial mistake. These dichotomies are seen as choices on a one-dimensional scale between two extremes. The final choice will be a compromise between the extremes.

One can teach these dichotomic architectural patterns and their properties and thus improve choices at the initial stage of system design.

Returning to the proverb *"you can't keep a cake and eat it"*, it is of help to know how to make choices with respect to keeping/eating in such a way to get the most out of it, whatever the situation and your preferences are.

REFERENCES

[1] H. Kopetz. *Real-time Systems - Design Principles for Distributed Embedded Applicatins*. Kluwer Academic Publishers, Boston/Dordrecht/London 1997, 1997. ISBN 0-7923-9894-7.

[2] M.M. Belady L.A. Lehman, editor. *Program Evolution - Processes of Software Change*. APIC Studies in Data Proc. No. 27, Academic Press, 1985.

[3] G. Chroust. The empty chair - uncertain futures and systemic dichotomies. *Systems Research and Behavioral Science, vol. 21 (2004)*, pages 227–236, 2004.

[4] G. Chroust. You can't keep a cake and eat it! dichotomies in systems engineering. In *Kokol, P.: . The IASTED International Conference on SOFTWARE ENGINEERING 2005, Innsbruck, Austria*. Acta Press 2005, paper 455-083, 2005.

[5] R.W. Phillips. State change architecture: A protocol for executable process models. *Tully C. (ed.): Representating and Enacting the Software Process Proc. 4th Int. Software Process Workshop, May ACM Software Engineering Notes vol. 14, no. 4.*, pages 129–132, 1988.

[6] M. Franz. Dynamic linking of software components. *IEEE Computer vol. 30 (1997) no. 3*, pages 74–81, 1997.

[7] D. Gries. *Compiler Construction for Digital Computers*. John Wiley, New York, 1971.

[8] P. Rechenberg. Compilers. *Morris, D. and Tamm, B. (eds.): Concise Encyclopedia of Software Engineering, Pergamon Press 1993*, pages 59–64, 1993.

[9] M.J. Hoevel L.W. Flynn. A theory of interpretive architectures: Ideal language machines. *Stanford Univ., Computer Systems Lab., TR 170, Feb. 79*, 1979. 56 pp. VIP09MI-A664.

[10] W. Mackrodt. Considerations on language interpretation for microprocessor systems. *Microprocessing and Microprogramming 7*, pages 110–118, 1981.

[11] H.P. Mössenböck. Automatische Speicherbereinigung. In *P.Rechenberg, G.Pomberger (eds.): Informatik-Handbuch, 3. Auflage, Kap. D12.6*. Hanser-Verlag 2002, 2002.

[12] W.W. Agresti. *New Paradigms for Software Development*. IEEE Computer Soc. Press, North Holland Publ. Comp., 1986.

[13] J.P. Bowen and M.G. Hinchey. Ten commandments of formal methods. *IEEE Computer*, 28:4:56–63, 1995.

[14] T. Gilb and D. Graham. *Software Inspection*. Addison-Wesley Reading Mass. 1993, 1993. ISBN 0-07062166-7.

[15] T.C. Chen. Parallelism, pipelining and computer efficiency. *Computer Design Jan*, pages 365–372, 1971.

[16] IBM Corp. IBM GRID Computing. http://www-1.ibm.com/grid/, June 2004, 2004.

[17] D.J. Kuck. A survey of parallel machine organization and programming. *ACM Computing Surveys vol. 9, No. 1*, pages 29–58, 1977.

[18] R.M. Adler. Emerging standards for component software. *IEEE Computer*, 28:3:68–77, 1995.

[19] G. Chroust. Orthogonal extensions in microprogrammed multiprocessor systems: A chance for increased firmware usage. *EUROMICRO Journal vol. 6, No. 2*, pages 104–110, 1980.

[20] W. Stallings, editor. *Reduced Instruction Set Computers - Tutorial*. Computer Society Press, 1986, 1986. Order No. 713, ISBN 0-8186-0713-0.

[21] F.L. Williams and G.B. Steven. How useful are Complex Instructions? - A case study using the Motorola M68000. *Microprocessing and Microprogramming*, 29:4:247–259, 1990.

[22] G. Chroust. *Modelle der Software-Entwicklung – Aufbau und Interpretation von Vorgehensmodellen*. Oldenbourg Verlag, 1992. ISBN 3–486–21878–6.

[23] A. Kemper and A. Eickler. *Datenbanksysteme - Eine Einführung*. Oldenbourg Verlag, 1997, 1997.

[24] J. Magee J. Kramer. The evolving philosophers problem: Dynamic change management. *IEEE Tr. on Software Engineering*, 16:11:1193–1306, 1990.

[25] T. Kume H. Nakajo. The principles of foolproofing and their application in manufacturing. *Reports of Statistical Application Research JUSE, vol. 32 no. 2*, pages 10–29, 1985.

[26] K. Kurbel and Schnieder T. Integration issues of information engineering based i-case tools. Working Papers of the Institute of Business Informatics, Univ. Münster, No. 33, http://www.wi.uni-muenster.de/inst/arbber/ab33.pdf, 1994.