Programs = Data + Algorithms + Architecture: Consequences for Interactive Software Engineering

Stéphane Chatty

ENAC, Laboratoire Informatique et Interaction,
7 avenue Edouard Belin, 31055 Toulouse Cedex, France
and
IntuiLab, Prologue 1, La Pyrénéenne, 31672 Labège Cedex, France
http://recherche.enac.fr/~chatty

Abstract. This article analyses the relationships between software architecture, programming languages and interactive systems. It proposes to consider that languages, like user interface tools, implement architecture styles or patterns aimed at particular stakeholders and scenarios. It lists architecture issues in interactive software that would be best resolved at the language level, in that conflicting patterns are currently proposed by languages and user interface tools, because of differences in target scenarios. Among these issues are the contravariance of reuse and control, new scenarios of software reuse, the architecture-induced concurrency, and the multiplicity of hierarchies. The article then proposes a research agenda to address that problem, including a requirement- and scenario-oriented deconstruction of programming languages to understand which of the original requirements still hold and which are not fully adapted to interactive systems.

1 Introduction

Niklaus Wirth, renowned computer science teacher and programming language designer, wrote in 1975 a reference book entitled “Algorithms + Data structures = Programs” [1] that has influenced thousands of programmers. It may be that his equation was incomplete though. Software architecture, that is the way of organising software into interconnected parts, has progressively become recognized as a central issue in programming and software engineering, to the point where students now spend more time learning about patterns and frameworks than data and algorithms. Yet, software architecture is still mostly considered a separate issue from programming languages. We contend that this is a serious issue for the software engineering of interactive systems. Short of being able to write "Programs = data + algorithms + architecture" and addressing architecture issues at the language level, the architecture of interactive software may be doomed to inconsistency and complexity.

The architecture of interactive software has been heavily studied and many influential results in software architecture were obtained by researchers with a background in interactive software, or derived from their work. Compare for example the authors and topics in the following list of publications: [2-10]. Still, very few actors of the

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-3-540-92698-6_37

J. Gulliksen et al. (Eds.): EIS 2007, LNCS 4940, pp. 356–373, 2008.

domain consider that the situation of interactive software architecture is satisfactory: teaching these issues is still awkward, and programming interactive software remains complex as soon as one does not stick to common WIMP interfaces. The author's personal experience in selling interactive software design and solutions was a very instructive field study of that problem: most potential customers of interactive software technology are put off by perceived incompatibilities between the processes of user interface design and traditional software engineering, or even more explicitly by software incompatibilities [11]. For instance, customers had to renounce implementing the chosen design when finding that implementing it with Java Swing would cost four times the cost of a WIMP interface, just because of architecture mismatches.

In this article, we propose an analysis of the relationships between software architecture, programming languages and interactive software, based on the principles of requirements and usage scenarios. We highlight a strong coupling between languages and architecture, and propose that languages can be studied using the same methods. We then use this analysis to identify some requirements and scenarios where current programming languages and interactive software conflict and thus favour inconsistent or costly architecture solutions. User interface toolkits act as architectural patches to languages, but the result is not always consistent. Finally, we propose a research agenda for addressing that issue, considering that user interface development brings at the same time new problems and techniques for addressing them. Architecture issues can be addressed by identifying the underlying usage scenarios more explicitly before applying the body of knowledge created for programming languages. Doing so, in addition to helping to understand interaction architecture, could help improve programming languages.

2 Of Programming Tools, Scenarios and Architecture

The software engineering and the user interface design communities have come up with similar models of requirements engineering and design for software products. With some differences in vocabulary, they share the concepts of stakeholders, external requirements or goals, technological choices or constraints, scenario-or usecase-based design, task or process analysis, and iterative design [12,13]. These design models have proven effective over the years for designing tools and (in many cases) improving the efficiency of the final users.

These models can be applied to the design of a special category of tools: the tools made for software builders themselves. Programming languages are tools for programmers; development environments are tools for programmers and project managers; user interface toolkits are tools for programmers and interface designers; some specialized languages are aimed at non-professional programmers, and so on. Some of these tools are developed with a focus on a given technology and aimed at specific tasks, for instance logic programming for knowledge management. Some have to take into account constraints such as the performance of compilers or computers. But all of them were designed, explicitly or not, with stakeholders and usage scenarios in mind. That is, they take into account all the persons that are concerned with the product because they build, manage, or use it and they try to capture the multiple activities around the product through concrete stories called scenarios or use cases. Many

language designers used themselves as the target users, made their own scenarios mentally, and performed initial iterations by testing the candidate designs against their mental scenarios. Others, such as the designer of Perl, used the whole user community for a vast participatory design process. In all cases, understanding the underlying scenarios and requirements provides a powerful means for analysing architectures, languages and other tools.

In the following sections, we identify the types of stakeholders and scenarios that underlie the state of the art in software architecture, programming languages and interactive software architectures. We will later use that analysis to detect some plausible causes of the problem of interactive software architecture.

2.1 Software Architecture

One definition of software architecture is "the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their evolution over time" [14] or in other words, how to split programs in smaller parts and glue them together. In their seminal paper on software architecture, Garlan and Shaw analyse architectural styles by focusing on the nature of components and the glue that links them [15]. Software architectures are not tools for building software, but rather rules, guidelines, or patterns for the same purpose. Nonetheless, the above reasoning on scenarios applies, in that an architecture style is a design aimed at supporting some scenarios of software building for stakeholders of the software industry. Programming tools are complete and implemented designs, whereas architectures styles are partial designs. Some architecture styles come with supporting tools. Others are more theoretical and let their users choose how to implement them, either because they address issues orthogonal to those addressed by available tools, or because they conflict with them (see the section on Interactive software architecture below for examples).

Architectures, like tools, are aimed at sparing their users from some design choices by providing a good solution adapted to their goals. For instance, a "pipes and filters" architecture like that of the Unix shell focuses on the needs of three types of stakeholders involved in the production of data analysis software: the programmers of basic analysis algorithms, who are encouraged to isolate their algorithms in separate programs, thus avoiding the details about how their algorithms will be used; the shell programmers, who are encouraged to implement a simple interface for connecting program inputs and outputs, and know that their shell will be usable in various situations; and finally power users who can build custom analysis chains at a very low cost.

The role of scenarios is recognized by the software architecture community [16]. Admittedly, no architecture style is well adapted to all situations. The identified stakeholders include the end user, framework programmers, administrators, and maintainers. Scenarios include development, debugging, parameterising, all sorts of software reuse, and even off-shoring. It is recognised that the type of application (databases, interaction, AI, etc) is an important aspect of scenarios too [15]. It is interesting to note, however, that most of the literature on software architecture focuses on scenarios and techniques beyond a certain granularity of code. Most proposed definitions of software architecture suggest that it deals with medium and large-scale software components. Garlan and Shaw present software architecture as the third level of a scale where the first two levels are high-level programming languages and abstract data types.

2.2 Programming Languages and Hardware Design

It is also interesting to analyse languages and even computers through the looking glass of scenarios and architecture. Actually, many constructs in programming language are aimed at software architecture rather than algorithms or data structure. Most literature shows that all programming languages and even computing hardware enforce certain architecture styles and were built with certain stakeholders and scenarios in mind. It also hints that expressions in programmer lore such as “clean”, “elegant” or “orthogonal” are actually scenario-based architecture quality statements.

In the prehistory of computing, Jacquard looms were machines that executed programs coded on punch cards. The system was split into two components (machines and cards) so as to support a standard scenario involving two actors: the same machine built by a maker could afterward be used by an operator to produce different weaving patterns by changing cards. That architecture style where the central engine is fixed and smaller parts of the execution process can be changed at will was very influential on Ada Lovelace. She built upon the idea to propose that Babbage's analytical engine could be used to tabulate different mathematical functions by using different cards. She also used it to suggest that functions could be computed several times with different data [17]. Later Turing invoked similar computing scenarios to propose splitting the sequence of operations executed by the Automatic Computing Engine into “subsidiary operations” [18]. He also proposed instructions named BURY and UNBURY and a stack structure to support that architecture, thus laying out the foundations of the call stack. Support for implementing it was soon built into computers and since then has been present in the microcode of most processors.

Just like computing hardware, programming languages have been deeply influenced by these historical scenarios: a fixed engine executing interchangeable computations, or programmers splitting their code into several sequences so as to call the same sequence several times. The concept of function, procedure or subroutine borne from these scenarios is present in most languages. The design rationales written by language designers are dense with references to such scenarios. For instance Stroustrup [19] mentions “communication between designers and programmers” (p. 114) as a goal, states that “the issue of how separately compiled program fragments are linked together is critical” (p. 34), and that “C with Classes was explicitly designed to allow better program organisation; computation was considered a problem solved by C” (p. 28). Actually, languages such as Pascal, C++ or Java abound with features aimed at facilitating the splitting of programs into reusable parts: functions, name scoping, namespaces, typing, classes, etc. These features implement a style that is strongly suggested to programmers: split your programs in functions so that you can reuse them at will. Hence we claim that languages support the “Programs = data + algorithms + architecture” equation, and we observe that most languages are still based on the historical computation scenario.

True enough, the evolution of mainstream languages has been focused on supporting more and more complex software engineering scenarios. First it was observed that the functions paradigm could be used to support such uses as documenting, reading and maintaining code, or detecting errors. Then came more complex scenarios: a first programmer develops a library of functions that other programmers will reuse in their programs; or a programmer builds a computation engine in which other programmers

later insert their own computation functions; or a programmer builds a specialisation of a library and inserts it into a computation engine, etc. These scenarios are supported by features such as separate compilation, late binding, interfaces or exceptions. This evolution was possible because clever engineers always found how to extend the basic paradigm to support these scenarios: they were compatible with the historical architecture.

Alternate programming paradigms have been proposed: functional, logical, reactive or parallel programming. But usually the proposed justifications were about the expressive power of languages for a given programmer, not about architecture or scenarios involving multiple stakeholders. If some of these paradigms induce architecture styles that diverge from the historical style, this is apparently just a side effect. For instance, when Backus criticised "von Neumann languages" and proposed the functional style [20], he did it at the level of programming instructions, not at the level of combining larger parts of programs. Some of his arguments used architectural concepts ("language framework versus changeable parts"), but his concern was at a very fine grain and that did not lead him to challenge the functions paradigm. And the truth is that the ability of this paradigm to be applied to all situations is apparently unlimited.

2.3 Interactive Software Architecture

Nevertheless, after nearly 30 years of research history, interactive software does not seem to be part of that success story:

The user interface research community periodically debates about the reasons why so little of its successful research makes it to commercial products, and software issues are among the proposed explanations;

Programming rich user interfaces is still considered a highly complex task, and teachers still look for solutions to make their students able to create working interactive components during their courses;

Researchers working on new interaction styles often express frustration at current tools or build their own;

Many works have been devoted to software architecture, models and patterns for interactive software, which confirms that there are stills problems that need solving; the fact that research in the domain has considerably decreased is most likely not due to a sense of successful achievement;

Very few results have been integrated into programming languages, unlike with other software engineering works;

Industries in the defence, aerospace, automotive, or home automation industries are still looking for technologies that combine the results of user interface research and their current development tools;

The implementation of many interactive systems uses some sort of middleware, which frees architects from the constraints of languages by creating their own language (the middleware protocol) to glue components; the fast evolution of Web user interfaces is probably an example of this.

We propose to analyse causes of this situation by comparing the architecture styles induced by languages and those proposed for interactive software. We first try to identify the software engineering scenarios behind the proposed interactive software architectures, before identifying some conflicts in a later section.

One of the most cited reference is the Seeheim architecture model, proposed at a time when the problem at hand was retrofitting existing software with new graphical user interfaces [21]. This scenario was new because it required to organise software along two dimensions. The first axis was as usual a split into one fixed and one interchangeable parts: the functional core and the user interface. The second axis dealt with the varying location of execution control, which depends on the nature of the user interface: control is split between the functional core and the user interface for text user interfaces, and it resides within the user interface when it is graphical. These requirements led to propose a four-tier architecture pattern. However that was done at a very high level of abstraction, not explaining how that was related to programming constructs, probably because there was no obvious solution for that. When the Seeheim model was refined later into the Arch model, new tiers were added to accommodate more complex reuse scenarios including multiview user interfaces, but once again no relationship with programming languages was set forth [22]. This means that programmers are free to implement the architecture as they wish. But this freedom comes at a high cost, just as if programmers of classical programs had kept on coding in assembly language. More detailed architecture styles have been proposed. PAC [23] had the same aims as the Seeheim and Arch model, but with more concrete handling of concerns such as the hierarchical organisation of components. However it was no more based on programming language constructs.

In contrast with these architecture styles aimed at changing user interfaces, a series of architecture styles or patterns have been proposed and implemented as toolkits or frameworks to address more programmer-oriented needs [24]. The “Inversion of Control” (IoC) or “Dependency Injection” pattern recently gained popularity [26]; it captures the fact that containers are usually coded before the objects they contain even though they pass control to them at execution time. Earlier, a series of graphical toolkits have used the callback pattern or the late binding technique provided by object-oriented languages [4,5,25]. The MVC (Model-View-Controller) pattern focused on graphical rendering and input handling, relying on constructs of Smalltalk, a rare language built with user interaction scenarios in mind [9,27]. Some authors proposed to connect program components through one-way constraints [28] or dataflow connections [29] so as to support program readability and interchangeability of components, or even adaptation to execution platforms, in the context of direct manipulation and animation. With similar use scenarios in mind, but with a focus on graphical rendering, others have proposed to isolate graphical computations in components linked together by a hierarchical glue named a scene graph [30]. Others have proposed to isolate states and reactions to events in components based on finite state machines, Statecharts or Petri nets [31]. Others have noticed that architecture styles proposed by alternative programming styles matched some scenarios of interactive software development: tools were developed using the functional programming [32], the reactive programming [33], or the parallel programming paradigms. Some even tried to merge user interface programming deeply into the syntax of existing languages to try and force the compatibility of user interfaces and programming languages, see for instance the Ubit toolkit that makes heavy use of the operator overloading feature of C++ [34].

The theoretical architecture styles such as Seeheim, Arch or PAC could not fail: they represent real concerns and do not face “implementation details”. The more

implementation-oriented solutions were not as successful, even though most of them strike by their elegance. Apart from MVC and the Smalltalk environment, they all fall into one of these two categories:

The general purpose tools, which are widely used but considered as yielding complex architectures and limiting the evolution of user interfaces;

And the more specialized tools, which are not widely used, probably because the local help they provide conflicts with the requirements of the other parts of the software or the architecture style of the underlying language. In the rest of this paper we attempt to analyse the reasons behind this mixture of success and failure, and we propose a research agenda to address them.

3 A Multi-level View of Software Architecture

We observe that all the tools and architecture styles mentioned in the previous section are concerned with architecture at different levels of granularity. All levels propose to split applications into components in a way that efficiently supports scenarios where parts of the software are created at different times by different persons, but they deal with components of different sizes.

3.1 Four Levels of Architecture

Architecture can be considered at four levels with growing component sizes:

1. The lowest level is that of *programming instructions:* how can they be grouped and reused, for instance in iterations? We are used to juxtaposing instructions, but Turing identified that as a design choice: "A simple form of logical control would be a list of operations to be carried out in the order in which they are given" [18] Lisp or Occam do not rely on that implicit semantics of grouping. As for control structures, patterns are proposed that favour different reuse scenarios (using an assignment in a test, for instance). This level of architecture is handled by languages and processors: they define a data model, a set of instructions, and ways of organising them. All underlying usage scenarios have one stakeholder: a programmer who writes, reads, and debugs a small piece of code, usually at the scale of one page.
2. The next level deals with *structuring chunks* of programs: how do I split my code in sequences that are at most one page long and that can be reused at several places? That level deals with the needs of programmers or groups of programmers working on the same part of a program. It deals with scenarios such as documenting code, communicating about it, optimising or debugging it. Most languages handle it through functions or classes, or through alternate constructions such as continuations.
3. Then comes the level of *software reuse*, customisation and extension by different actors. Common stakeholders are groups of programmers that either split work and integrate it later or reuse libraries and frameworks built earlier. Others are project managers, maintenance managers and technical writers. Recently, engineers who deploy and parameterise software, or even users, have become stakeholders at that level. For classical software, that level has been handled by tools like preproces-

sors and linkers, then by languages, then more recently by architecture patterns and systems of plug-ins. For interactive software, it has been the focus of user interface management systems, toolkits or frameworks. Interactive software has been a great provider of research on that level, and the works listed previously are solutions pending for consideration. For instance, events were recently included in C# [35].

4. The highest level is *software planning*, concerned with reusing whole applications or groups of applications. It deals with stakeholders such as information directorates in companies, computer providers, software houses and scenarios such as product line management, deployment, etc. Expressions such as "software urbanism" [36] have been coined for this level, which we do not address here.

Taking the perspective of tool design, the first two levels are aimed at single users (the programmers), and the third level is more about groupware design (development teams). These levels cannot be handled independently.

3.2 Managing Compatibility

All levels cannot be addressed by a single tool. For instance it was decided to handle in operating systems issues that were best not handled in languages. However, a lot of research has been aimed at handling more and more of the higher levels in languages. The step from level 1 to level 2 was made very early; the step from level 2 to level 3 has started with FORTRAN II (the introduction of separate compilation) and is probably not over. Two probable reasons for that tendency are:

- A wish to minimise the number of concepts or patterns manipulated by programmers; once they are in a programming language or a processor, they can be used at all levels with no additional cost;
- Once a pattern has proved its value and compatibility with the language, a desire to encourage programmers to use that pattern rather than invent others which might prove incompatible and dangerous.

Compatible patterns. These two points highlight the importance of having compatible patterns throughout the four levels and especially within a given level. Patterns are compatible when they can be combined so that all scenarios they support individually are supported by the combination, without adding complexity. For instance, functions and object-oriented programming can be made compatible by deciding that object methods are functions. This allows to combine components written with either pattern. If compatibility is not retained programmers are led to creating code that has not the expected behaviour because the programmer had wrong expectations. At best this necessitates special documentation and training for programmers; at worst, programmers may try to introduce new concepts or syntaxes, succeeding only in masking the problems. For instance, message passing and functions can appear similar for architecture purposes but are based on different synchronisation models; mixing them is dangerous because the programmer's code may be executed in an unexpected way. Consequently, an architecture level should only use a subset of the connectors provided by the lower level (or compatible connectors), and its component types should be refinements of component types of the lower level. When incompatible patterns are identified at different levels, one can build middleware that adapts connectors: a RPC

library or a message bus, for instance. The additional cost is acceptable between levels 2 and 3, or 3 and 4, but not within level 2 or 3.

Pattern lifecycle. Another consequence of the two points above is the lifecycle of architecture patterns that they describe. Solutions are first proposed to programmers in tools that act as additions or modifications ("patches") to the underlying language. When an addition or modification proves safe and beneficial to a large audience, it ends up being part of a new language. Most user interface toolkits or frameworks provide both additions and modifications. The additions are interactive objects and algorithms: graphics, interaction management, gesture recognition, etc. The modifications are new level 3 or even level 2 architecture patterns: data-flow, scene graph, continuations, etc. The same holds for operating systems. Consider for instance the `select` call of Unix or the message queues of Windows: they provide mechanisms that are not native to the C (resp. C++) language and that allow asynchronous communication.

In the above lifecycle, additions usually stay out of the language. As for modifications, three states are possible:

Compatible modifications waiting for inclusion in a language, if someone can devise a clever way of including them;

Modifications that have been identified as incompatible and either force the use of a middleware layer or limit the usefulness of the toolkit.

Modifications that have not been identified as incompatible, and make the toolkit difficult or even dangerous to use.

Compatibility as a goal. Ideally of course, one would be able to design compatible architecture patterns that answer all known software engineering scenarios of a given domain, and thus ultimately build a language that supports that domain. That language would offer a component model and a linking mechanism that would hold at all levels and allow to build "fractal" software where the architecture patterns would be the same at all levels of hierarchy of the software. That would, among other things, make middleware useless. That would also allow the implementation of multilanguage solutions at level 3, such as Microsoft's .Net which allows the use of different languages for addressing different application parts. But it seems that the current situation today is that most proposed solutions for interactive systems are in the second or third state above. As stated before, this makes programming interactive systems more difficult and error-prone than necessary. This also has dire consequences on project management and user interface quality, encouraging to develop user interfaces at the end of projects when constraining architectures are already in place.

An exception to this situation would be the Smalltalk environment, which was explicitely designed along the lines of architecture consistency: "Smalltalk's design — and existence— is due to the insight that everything we can describe can be represented by the recursive composition of a single kind of behavioral building block (...)" [37] Even then, the limited industrial success of Smalltalk suggests that some key scenarios where not taken into account, the foremost being probably the interconnection with non-interactive software. C++ took the opposite stance, making it harder to develop interactive software. That shows how much understanding the possible architecture mismatches is important.

4 Understanding Mismatches

We now propose a few reasons why architecture patterns proposed at level 3 for interactive software display incompatibilities with those offered by most programming languages. Most reasons listed below stem from the same cause: interactive software involves new stakeholders and generates new engineering scenarios. If we except project managers, maintenance managers or technical writers, most scenarios described earlier in this article involved programmers who build their own programs by including components written by others, or insert their components into existing computation engines. User interface design and development multiplies the roles: it introduces interaction designers, graphical designers, developers of low fidelity prototypes, developer of the final application, framework developer, developers of device drivers, interactive component developers, users setting parameters of their application, etc. All these stakeholders have different backgrounds and use different tools, and they generate complex development scenarios. The complexity is similar to that of very large systems, even though a single program is produced. This partly comes from a new step of software engineering: it focused on programmers, then on software engineering groups, and now needs to focus on multidisciplinary software engineering groups [38].

4.1 New Reuse Patterns

Software reuse defines a partial order relation between components: to reuse a component, a programmer must know how to address it, and uses that in the newly written component. This relation fostered many constructs in programming languages: names given to functions or variables, typing, encapsulation to hide details, name rewrite to provide growing levels of abstraction, etc. This binary relation is well adapted to scenarios where programmers add layers upon layers of code. It is not to scenarios involving other types of stakeholders, because in that case there are more than one reuse relations. That challenges many mechanisms, starting with encapsulation:

An interface designer or a user who changes a font in an application accesses a property name defined by the programmer of a text field; that name is not accessible to other programmers; consequently, components should have several interfaces depending on the type of stakeholders: developers of new interaction modalities, interactive component developers, application programmers, graphical designers, users;

Even among programmers, the order relation may vary; for established concepts, the language and its core library reuse and encapsulate the operating system (see for instance the standard input in C); but with innovative user interfaces the application programmer is often also a device driver programmer, who for instance configures a wireless remote control to behave as a mouse; this requires framework developers to provide extension mechanisms for all operating systems, and breaks the traditional encapsulation hierarchy;

Encapsulation usually supposes that the reused component is complete, whereas interface skinning or the multidisciplinary development of components leads to splitting components in halves that are managed independently: a programmer will develop the behaviour and a graphical designer the looks, for instance. This lessens the added value of class derivation.

4.2 Contra-Variance of Reuse and Control

One of the most common reuse scenarios in interactive software is that of event sources: picking a target in graphics scenes or interpreting speech is hard enough that one prefers to reuse existing libraries. Reusing these components has led to event-driven programming and to the progressive replacement of graphical libraries by programming framworks. This reuse pattern is fundamentally different from the historical reuse scenarios. Consider the partial order relation introduced in the previous section (reuse relation) and compare it with another partial order relation: that which relates two components when one transfers control to another one (control relation). In the historical reuse scenarios, the two relations are covariant: the caller knows the callee, because the main program is written after the libraries or at least linked later. With interactive software, the main program is still written last but initiative always comes from external sources: timers, network peers, or input devices. The two relations are thus contra-variant.

This contra-variance has been accounted for in diverse ways: event-driven dialogue, main loops, callbacks, programming frameworks, IoC pattern, are all toolkit-level solutions for supporting it. However, we believe that it should be handled at a more basic level, because it is characterises the most important reuse pattern in interactive software. Apart from their initialisation, there are few situations where components are in a "covariant reuse" situation; actually, it is possible to describe fairly rich user interfaces without the concept of function, whereas it is impossible without a solution for the "contra-variant reuse".

Apart from the additional cost and complexity induced by this inversion of priorities between languages and interactive software, it causes several problems:

Event emission is a good basis for encapsulating components: a button emits either `press` or `release`, a dialogue box with two buttons only emits `ok` or `cancel`, and so on; managing it outside of languages deprives programmers from that encapsulation;

There are solutions for providing both dataflow and event emission with a unified model; having function calls as the predominant paradigm in programs makes it difficult to implement, in particular because of diverging semantics as for sequencing;

Using the functions paradigm creates a misunderstanding with functional core programmers: it does not help them to detect that user interfaces cannot be programmed as mere function calls, and pushes many teams to restrain to interface components that can be used with the functions paradigm;

And finally it plays a role in the "inversion of calendar" problem that strikes many large projects: when a user interface design is chosen towards the end of a project, managers realise that the architecture chosen years before does not allow it. Indeed, it is logical to choose an architecture early enough: at the beginning, the interface is still in the iterative design phase and there are other developments to start. But with no knowledge of the interaction styles that will be chosen one can only resort to the common denominator, which currently appears to be the function call, whereas the only certain thing is that it will not be the function call. It is therefore necessary to promote a basic pattern that accommodates the contra-variant reuse pattern, and if possible the covariant one for the commodity of functional core development.

4.3 Locality of State and Computations

When reading software or locating errors, locality of behaviours is an important feature: having one page per algorithm makes it easier to use a divide-and-conquer approach. Functions are fit for that purpose when programs mostly consist of algorithms: each function implements a computation, which in addition makes computations reusable. However, computations and algorithms play a more minor part in interactive software. Most behaviours consist in managing a state, its modifications upon events, and the associated actions. For instance, leaving the graphical objects aside, a visual button is essentially made of a state (disabled, idle, pushed, etc) and ways of changing it. In computation-oriented programs functions are essential and data can be hidden in the call stack, and that led to functional programming. With interaction, state is essential in behaviours and the locality principle would require that all code that changes it is grouped. That pushed researchers to propose programming patterns based on finite state machines, Statecharts or Petri nets, but:

When using a computation-oriented language, the transitions are implemented as functions or methods and the principle of locality is not met;

Functions and transitions are not as easy to match as functions and methods: all uses of function arguments do not easily transpose to transitions, and the expected sequencing properties are not always the same;

In the same way as functions can be combined in complex ways, many development scenarios involve the combination of several behaviours; for instance, a blinking icon has two orthogonal behaviours: the blinking, and the ability to be dragged across the screen; state management should allow to separate and combine states at will, just like for functions;

States and behaviours are an important part of reuse scenarios and thus should be part of the reuse patterns: with interactive systems, programmers do not reuse components by adding functions to them; they add event reactions or animations as much as they would change the graphical looks;

In addition to be combined or reused, behaviours sometimes need to be structured hierarchically: levels in a game or steps in a wizard are high level states that influence lower level behaviours such as the speed of targets or the enabling/disabling of buttons; hierarchical state machines are a local solution that mixes badly with the software reuse scenarios;

Finally, not all behaviours have the same focus on state transitions; some, often represented by dataflows, are made of successive computations that alter quantitative states. Animation, for example, relies on combining algorithms to compute the positions of graphical objects. This creates a continuum between computations, dataflows, and state-transitions that would require a uniform organisation pattern.

4.4 Architecture-Related Concurrency

Interactive systems require concurrency in few situations only. When reading large documents, the user should be able to interact with the system even when the program is busy loading the file. For most other situations, one only needs to rely on the interleaving of external events which all occur asynchronously. However, software engineering scenarios and architecture induce some form of concurrency that needs to be handled properly.

Consider a program that emits events when the user clicks on an icon. Classical interactive software engineering scenarios lead to providing that component in a library, so that programmers can reuse it and bind their code to events it emits. It may happen that several components are connected to this event source. For instance, an application programmer can bind both the modification of a text field and the opening of a dialogue box, both obtained from two widget programmers. Suppose the box emits a sound then an animated feedback when opening, and the text changes with an animation. Then for all purposes, these two widget programmers are in a concurrent situation: neither knows about the actions coded by the other, and nevertheless the application programmer may want to ensure a sequencing order: sound first then animations, for instance. That requires that the programming environment allows to express sequencing constraints on the actions triggered by events. This requirement is rarely fulfilled, and many commercial programs exhibit strange behaviours with that regard.

As usual, one may be tempted to handle this requirement with the concepts or the syntax of the underlying language. For instance, the author used an animation library that encapsulated sequencing in a functional programming style. It was very elegant to use, except that it had to be implemented through nested event loops, and when sequencing more than two animations, the first animation might get stuck and the program continued its execution with two nested mainloops. Trying to hide the concurrency only made it bite programmers later. The safe solution is to use a concurrent language or a system of threads and semaphores, which forces user interface programmers to absorb complex concepts and does not make it easy to explicit sequencing properties of their code.

4.5 Multiple Hierarchies

Programming languages manage two hierarchies in programs. First, they give an important role to the lexical hierarchy of code to manage components. Most names are visible only within a given lexical scope, which plays an important role in defining reusable functions and components. Languages like C++ associate the lifecycle of objects to their lexical scope. Some languages, like Occam, even use lexical scopes to define the concurrent or sequential execution of instructions. Second, most languages introduce a hierarchy or types or classes that is often used to represent a hierarchy of domain concepts. Interactive systems require that other hierarchies are managed by the language or toolkit, and can rely very little on syntax. When a component is made of sub-components, these can:

- Be created in a given lexical scope and use the names defined in that scope;
- Be derived from another type of components, using the class hierarchy proposed by object-oriented languages;
- Belong to a given modality (graphics, speech, etc) and occupy a certain position in a modality-specific hierarchy (scene graph or widget containment for instance); that is the hierarchy seen by the specialist of that modality;
- Influence the execution of their parent and sibling components, for instance because their sizes is used by the layout algorithm, because their current state influences the behaviour of another component, or because their mere presence changes the nature of the user interface: consider for instance a graphics layer that removes

all colours from the interface whenever a modal dialogue box is displayed. There are multiple independent behaviour hierarchies, relatively independent from each other. For all these hierarchies, it is tempting for programmers either to map them to the existing hierarchies in languages, or to build one's own set of graphs. The first option often yields conflicts. For instance, it is tempting to use a class hiearchy to represent the nature of components: a hierarchy of graphical object classes, a hierarchy of speech object classes, etc. This potentially leads to very complex class hierarchies when containers are present: can graphical groups contain speech objects? can windows contain animation trajectories? The latter option creates less complexity but forces programmers to build their own hierarchy management system, which cannot benefit from services provided by the language for its own hierarchies, such as renaming and encapsulation.

Furthermore, language hierarchies are limited to the scope of programs. They do not scale up to applications built as several programs. To do so, one needs to use middleware such as Corba, which provides a multi-program class hierarchy but at a very high cost. Ideally, a language should provide a hierarchy management that supports the hierarchies found in interactive systems, and valid at all levels of granularity, thus enabling to handle programs like components.

5 Related Work and Research Agenda

This is not, by far, the first attempt at analysing the nature of programming languages and their issues. To begin with, all language designers appear to have carried out a critical analysis of existing languages. As already discussed in this paper, most did it with programmers in mind. Examples include Backus on functional programming [20], Kay on Smalltalk [37] or Stroustrup on C++ [19]. Prominent software engineering essayists often carry out the same type of analysis, based on their experience of industrial development; see Graham for a recent example [39]. Some researchers have tackled the issue of dealing with more complex software engineering scenarios. Aspect programming [40] and the meta-object protocol [41] are examples of that approach. Software architecture specialists have identified the problem of architecture mismatch [14] and analysed their causes and consequences, at a generic level. Several researchers from the interactive software community worked on resolving some mismatches posed by interactive software. For instance, Prospero is aimed at solving issues between different levels of tools in CSCW software development [42]. Wegner even goes further and challenges the very fact that algorithms should be central in programming, proposing interaction as the key construction [43].

Our approach focuses on architecture and relies on the conviction that user interface development brings both problems and techniques for addressing them. A first list of problems has been presented in this article. The techniques are those of user interface design: requirements engineering and design techniques for usercentred design. We are convinced that an explicit use of these techniques, often used implicitly by language designers, can help understand the needs of interactive software stakeholders, the solutions proposed, and how to match them. Our experience with the user-centred design of the graphics module of a user interface environment [38] strengthens that conviction. We therefore propose a research agenda that could help

understand to what extent solutions currently proposed by programming languages can be used for or adapted to the efficient development of interactive systems, or how they could be modified to support the expected development scenarios without forfeiting their other qualities. This agenda includes:

Reviews of the software engineering and programming language literature to identify all stakeholders and scenarios taken into account in these domains;

Identification of stakeholders and scenarios with modern and/or future interactive software;

Measurements of how these scenarios are handled in current software;

Identification and classification of requirements and properties expected from interactive software development tools and languages;

Deconstruction of programming languages and theories to identify the supported architecture patterns and the underlying scenarios;

Identification of the patterns in traditional or alternative languages that support the desired scenarios, and those that potentially conflict with them; this may lead us to discover that some works in interactive software architecture have exact equivalent in programming language research;

Research of compatible patterns that support the scenarios from interactive software; in other words, re-application of the working methods of the language and software engineering communities once the deconstruction has been performed, including formal methods;

Construction of a set of basic instructions and patterns adapted to interactive software, so as to build the equivalent of Microsoft .Net for developing interactive software with languages adapted to each part (graphical interface, functional core, speech interface, dialogue, etc).

6 Conclusion

In this paper, we have proposed to analyse programming languages and interactive software in terms of software architecture and in terms of stakeholders and scenarios supported by architectures. We have suggested that software architecture is present at several levels of granularity, the finest grain being handled by programming languages. We have described user interface toolkits as providing modifications to the architectures proposed by languages. We have listed several issues where languages and interactive software bring conflicting patterns, causing complexity that must be managed by programmers and that impedes innovation in user interaction. Finally, we have proposed a research agenda based on the identification of stakeholders, scenarios and architecture patterns that involves the application of language design techniques to interactive software tools or even interactive software languages. User interface design teaches us that humans are able to adapt to various designs, sometimes accepting systems that make them relatively inefficient. How much of this coadaptation is at work when we build user interface tools based on languages? So far, the user interface community has mostly focused on “getting the job done with the tools provided”, that is producing the expected user interfaces and taking the rest of software tools as immutable. Maybe we need some usability experts for ourselves!

Acknowledgements. This article finds its roots in a long conversation with M. Beaudouin-Lafon in Palos Verdes, CA in 1994. Some ideas came from there or my later work with P. Palanque and J. Accot at CENA. The rest came from an extreme experience at IntuiLab in 2002-2004: trying to apply to ourselves participatory design as taught by W. Mackay in the design of IntuiKit. Finally, marketing work with D. Figarol helped me articulate the arguments. S. Conversy, P. Dragicevic and M. Beaudouin-Lafon helped to improve the paper.

References

1. Wirth, N.: Data structures + algorithms = programs. Prentice Hall, Englewood Cliffs (1975)
2. Kruchten, P., Sotirovski, D.: Implementing dialogue independence. IEEE Software 12(6), 61–70 (1995)
3. Kruchten, P.: The Rational Unified Process — an Introduction. Addison-Wesley-Longman (1999)
4. Linton, M.A., Vlissides, J.M.: The design and implementation of InterViews. In: Proceedings of the USENIX C++ Workshop (1987)
5. Weinand, A., Gamma, E., Marty, R.: ET++ -an object-oriented application framework in C++. In: OOPSLA 1988 Proceedings (1988)
6. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, Reading (1995)
7. Bass, L., Coutaz, J.: Developing software for the user interface. The SEI Series in Software Engineering. Addison Wesley, Reading (1991)
8. Bass, L., Clements, P., Kazman, R.: Software architecture in practice. Addison-Wesley, Reading (1998)
9. Krasner, G., Pope, S.: A cookbook for using the Model-View-Controller user interface paradigm in Smalltalk 1980. Journal of Object-oriented programming 1(3), 26–49 (1988)
10. Barrett, R., Delany, S.J.: OpenMVC: a non-proprietary component-based framework for web applications. In: Proceedings of the 13th international WWW conference (2004)
11. Chatty, S., Sire, S., Lemort, A.: Vers des outils pour les équipes de conception d'interfaces post-WIMP. In: Actes d'IHM 2004, pp. 45–52. ACM Press, New York (2004)
12. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: ICSE 2000: Proceedings of the Conference on The Future of Software Engineering, pp. 35–46. ACM Press, New York (2000)
13. Muller, M.J., Kuhn, S.: Participatory design. Commun. ACM 36(6), 24–28 (1993)
14. Garlan, D., Allen, R., Ockerbloom, J.: Architectural mismatch: Why reuse is so hard. IEEE Software 12(6), 17–26 (1995)
15. Garlan, D., Shaw, M.: An introduction to software architecture. In: Ambriola, V., Tortora, G. (eds.) Advances in Software Engineering and Knowledge Engineering. Series on Software Engineering and Knowledge Engineering, vol. 2, pp. 1–39. World Scientific Publishing Company, Singapore (1993)
16. Kazman, R., Abowd, G., Bass, L., Clements, P.: Scenario-based analysis of software architecture. IEEE Software 13(6), 47–55 (1996)
17. King, B., Lovelace, A.: Notes by the translator of the Sketch of the Analytical Engine invented by Charles Babbage, by L.F. Menabrea. Scientific Memoirs 3, 666—731 (1843)
18. Turing, A.M.: Proposals for the development in the mathematics division of an Automatic Computing Engine (ACE). Technical Report E882, Executive Committee, NPL (1946)

19. Stroustrup, B.: The Design and Evolution of C++. Addison-Wesley, Reading (1994)
20. Backus, J.: Can programming be liberated from the von Neumann style? A functional style and its algebra of programs. Communications of the ACM 21(8) (1978)
21. Pfaff, G.E. (ed.): User Interface Management Systems. Eurographics Seminars. Springer, Heidelberg (1985)
22. Bass, L., Pellegrino, R., Reed, S., Seacord, R., Sheppard, R., Szezur, M.R.: The Arch model: Seeheim revisited. In: CHI 1991 User Interface Developers Workshop (1991)
23. Coutaz, J.: PAC, an implementation model for dialog design. In: Proceedings of the Interact 1987 Conference, pp. 431–436. North Holland, Amsterdam (1987)
24. Myers, B.A.: A brief history of human-computer interaction technology. Interactions 5(2), 44–54 (1998)
25. Beaudouin-Lafon, M., Berteaud, Y., Chatty, S.: Creating direct manipulation interfaces with X_{TV}. In: Proceedings of EX 1990, London, pp. 148–155 (1990)
26. Martin, R.C.: Agile Software Development: Principles, Patterns and Practices. Pearson Education, London (2002)
27. Goldberg, A.: SMALLTALK 1980, the Interactive Programming Environment. Addison-Wesley, Reading (1984)
28. Myers, B.A.: Creating user interfaces using programming by example, visual programming and constraints. ACM Transactions on Programming Languages and Systems 12(2), 143–177 (1990)
29. Chatty, S.: Defining the behaviour of animated interfaces. In: Proceedings of the IFIP WG 2.7 working conference, pp. 95–109. North-Holland, Amsterdam (1992)
30. Strauss, P.S.: Iris inventor, a 3d graphics toolkit. In: OOPSLA 1993: Proceedings of the eighth annual conference on Object-oriented programming systems, languages, and applications, pp. 192–200. ACM Press, New York (1993)
31. Palanque, P., Bastide, R.: Petri net based design of user-driven interfaces using the interactive cooperative object formalism. In: Proceedings of the DSV-IS 1994 workshop, pp. 383–401. Springer, Heidelberg (1994)
32. Dannenberg, R.B.: Arctic: A functional language for real-time control. In: Proceedings of the ACM Conference on Lisp and Functional Languages, pp. 96–103 (1984)
33. Clement, D., Incerpi, J.: Programming the behavior of graphical objects using Esterel. In: Díaz, J., Orejas, F. (eds.) TAPSOFT 1989 and CCIPL 1989. LNCS, vol. 352. Springer, Heidelberg (1989)
34. Lecolinet, E.: A molecular architecture for creating advanced GUIs. In: Proceedings of the ACM UIST, pp. 135–144 (2003)
35. Venners, B., Eckel, B.: The C# design process. a conversation with Anders Hejlsberg (2003), `http://www.artima.com/intv/csdes.html`
36. Desreumaux, M., Oudrhiri, R.: Information and software systems: from architecture to urbanism. In: Proceedings of the 1st IFIP Working Conference on Software Architecture. Chapman & Hall, Boca Raton (1998)
37. Kay, A.C.: The early history of Smalltalk. ACM SIGPLAN (3), 69–75 (1993)
38. Chatty, S., Sire, S., Vinot, J., Lecoanet, P., Mertz, C., Lemort, A.: Revisiting visual interface programming: Creating GUI tools for designers and programmers. In: Proceedings of the ACM UIST. Addison-Wesley, Reading (2004)
39. Graham, P.: Hackers and Painters: Big Ideas from the Computer Age. O'Reilly Media, Sebastopol (2004)
40. Kiczales, G.: Aspect-oriented programming. ACM Computing Surveys 28(4) (1996)
41. Kiczales, G., des Rivières, J., Bobrow, D.G.: The art of the meta-object protocol. MIT Press, Cambridge (1991)

42. Dourish, P.: Using metalevel techniques in a flexible toolkit for CSCW applications. ACM Transactions on Computer-Human Interaction 5(2), 109–155 (1998)
43. Wegner, P.: Why interaction is more powerful than algorithms. Communications of the ACM 40(5) (1997)

Questions

Prasun Dewan:
Question: Regarding the influence of programming languages, all programming languages are Turing complete. Just because you find a language difficult to use could mean you don't know how to use the language.

Answer: I have lots of examples, but indeed it is really hard to prove it.

Helmut Stiegler:
Answer: The language related notion of a control stack goes beyond a data-driven way of a processing model according to "last-in-first-out". The notion is based on a "processing context" of a unit of processing (usually called a "procedure") being automatically handled by an implicit mechanism and being independent from data visibly accessed by the unit of processing. This kind of control stack was introduced by Bauer and Samualtou in the Algre language, and a patent was granted to them.