# Towards a Pattern Language for Sound and Music Computing

by

**Pau Arumí**

Submitted in partial fulfilment of the requirements for
the degree of Diploma of Advanced Studies
Doctorate in Computer Science and Digital Communication
Department of Technology

Tutor: Dr. Xavier Serra

Universitat Pompeu Fabra

Barcelona, July 2006

"We adore chaos because we love to produce order."
M. C. Escher

# Abstract

Sound and music computing is all about... computing. But despite the large number of reusable software components and models-of-computation available for this domain, audio developers face new problems all the time. Most often, however, the same or similar problems have already been solved by others, but they lack the means for reusing those solutions. This dissertation proposes a design pattern language for data-flow systems: a technique that bridges the gap between the type of functionality provided by libraries and data-flow frameworks, on one side, and data-flow models-of-computation, on the other. The problem with libraries is that they often are too specific. Frameworks cover a broader scope but often overlimiting how to do things. On the other hand, the data-flow models-of-computation favors analysis versus design and implementation.

A pattern is a proved solution to a recurring design problem. It pays special attention to the context in which it is applicable, to the competing "forces" it needs to balance, and the consequences of its application. A pattern language, moreover, suggests a sequence of patterns to be applied depending on the current context. Patterns provide a better solution when the focus is on customizing a new design on an existing code. However, neither data-flow systems nor other audio-related areas have yet received relevant additions of domain specific patterns.

We demonstrate that the proposed catalog provides useful design reuse in the audio domain, and we show that they can be found in many different applications. As an example, we present the design of an object-oriented framework that uses all the patterns. We also show that patterns are useful to communicate, document and compare designs of audio systems. We believe that the incorporation of pattern-driven design will allow audio software to "grow up" out of the craftsman state and into a more mature state.

# Acknowledgements

I would like to thank Xavier Serra, my supervisor, for giving me the opportunity to work on this topic and in the great environment of the Music Technology Group.

I specially thank Xavier Amatriain who has been my advisor, mentor and friend. His vision and guidance helped me learn a lot, as well as find and focus on an exciting research topic.

I am very grateful to David García for our fruitful discussions on patterns, and for being so patient to my millions of programming and Linux questions. Further thanks to my workmates Maarten de Boer and Oscar Celma.

I also thanks to all the people who have contributed to the development of the CLAM framework, beyond Xavier Amatriain and David García, a non-exhaustive list should at least include Maarten de Boer, Ismael Mosquera, Miguel Ramírez, Xavi Rubio, Xavier Oliver and Enrique Robledo.

The members of Xavier Serra's Music Technology Group have provided a first-class environment for discussing and exploring research ideas. Jordi Bonada, Alex Loscos, Emilia Gomez, Bram de Jong, Lars Fabig and many other researches in the group have provided positive feedback during my research.

Thanks to Dietmar Schetz, from Siemens, who was my "shepherd" during the process that leads to presenting a pattern catalog to the PLoP conference. He revised early drafts of the patterns and gave insights from his experience writing patterns, which caused a huge —and worth while— restructuring effort. Dietmar comments provided a reality check for my ideas.

I'm also indebted to the Linux audio community and open-source community in general, which gave me the chance to examine the code of many audio systems.

This research was funded by a scholarship from Universitat Pompeu Fabra and by financial support from the STSI division of the Catalan Government.

Finally, I am grateful to my girlfriend, Matilda, who provided love, advice and patience.

# Table of Contents

# List of Figures

# Chapter 1

# Introduction

After obtaining the grade in computer science in the Universitat Politècnica de Catalunya, specializing in software engineering, I had the opportunity to join Xavier Serra's Music Technology Group so I was able to combine two of my passions: computers and music. During the last 5 years I've been very involved on the development of the CLAM framework (C++ Library for Audio and Music) [Amatriain and Arumí, 2005]. Which has been a fruitful environment to learn, experiment and put together many aspects of sound and music computing, and software engineering.

Recently I was asked for advice on how to implement certain features in three different audio related software under development in my university. It happened that all three problems were familiar to me because they were issues solved during the years of experience developing and evolving the CLAM framework.

Direct reuse of code was not feasible in these cases. Mostly because existing code was not readily usable and needed a big integration effort, or because it would incorporate unneeded complexity making the solution overkilling —thus, not compatible with the *simple design* quality[Beck, 1999]. But also for non-technical reasons like incompatible software licenses.

I found that because of having seen —or directly implemented— similar problems before, I was able to predict the resulting context and the trade-offs that the solution carried, so we were able to discuss it's applicability and optimization trade-offs before hand. And we were able to implement the similar design on a new system very quickly. I roughly estimate that an order of magnitude quicker

—and less defective also— than the first time I approached the similar problem.

Actually I see this happen around me over and over again. To give a concrete example of a design reuse in our domain: In the last month, I've seen my colleague Maarten de Boer implementing a real-time application that processes audio samples being read from the disk on-the-fly. This is not easy at all to do right because of a myriad of potential pitfalls: It involves non-blocking multi-threading techniques and sharing buffers between a real-time thread in charge of the processing and a low-priority thread that reads chunks of audio from the disk.

However, he was able to effectively implement it in few hours. And, he recognises, this was possible because he was reusing the same design idea that had worked for him several times before in similar situations. Never mind that the application at hand was about mosaicing, and the previous similar applications were, fundamentally, mere audio file players. Also, the new implemented solution varied in several aspects from the previous experiences. Finally, he comments, the first time he approached such problem it took him several weeks to find a good design, and it was after going through a trial of discovering the consequences of each design decision and correcting many design flaws.

I've checked with other developers and they have confirmed that the design reuse, specially in the sound and music field, is mostly reduced to personal experience and that is very hard to learn from others experience.

This thesis start from the observation that sometimes reusing existing domain libraries and frameworks is not possible. Instead or reuse at code level, we might take advantage of reuse at design level. *Design patterns*, introduced by [Gamma et al., 1995] is a technology that allows recording best design practices. But the field of sound and music computing has been quite impermeable to these recent advances.

Design patterns are not the only mechanism for conceptual reuse in our domain. Existing data-flow models are very useful for developing many kinds of audio systems. These models allows a thorough analysis of the system under development. However, there is a gap between analysis and an effective design and implementation. The research reported in this work goes toward bridging this gap, and also provides the scope for a future PhD thesis.

## 1.1 The Problem

Software development is a central activity in sound and music computing. It is important not only for practitioners in the industry developing audio applications for end-users. But also for academic researchers for who the primary resort for carrying our experiments is programming.

Experienced developers are *many* times more productive and successful than novices. The formers find that when they are trying to solve a new problem, the situation usually has something in common with a solution they have already either created or seen. The problems may not be identical, and an identical solution will rarely solve a new problem, but the problems are similar, so a similar solution should work. This similar solution generalized and formalized, is called a design pattern and provides reusable design that can be applied to new problems.

Fostering code and design reuse can drastically reduce the costs of audio software development and maintenance, therefore they represent good research opportunities.

This thesis approaches the general problem of lack of design reuse within the sound and music computing field. However, most part of this work narrows the scope and focus into the problem of how to translate general data-flow models into well crafted and understood designs.

Now we are going to dive a little deeper into both the general and concrete problems.

### Lack of design reuse in sound and music computing

Reusing software components like libraries and frameworks often results in a big productivity boost. However this is not always possible. Libraries are very limited in scope and they solve very specific problems. For example, loading audio files, converting sample rate, Fast Fourier Transform (FFT), audio routing, MIDI and audio I/O, etc.

Frameworks, on the other hand, have a much broader scope, allows the creation —instantiation— of complete applications with little programming effort, and provides mechanisms to be parametrized and extended. Actually, when you use a framework you are not only reusing code through an Application Program-

mer Interface (API) but you are also reusing its design. The disadvantages of frameworks are that they limit the ways to do things, in a way that, in some occasions, makes it hard or impossible to adapt to your needs. Moreover, frameworks are much more difficult to design and construct than applications, even though they greatly simplify application development.

Therefore, in many occasions code reuse is not feasible. Then, developers have no choice but to fall back to ad-hoc and "creative" solutions. Is in such cases when developers tend to reuse similar solutions that worked well, and, as they gain more experience, their repertoire of design experience grows and they become more proficient. Unfortunately, this design reuse is usually restricted to personal experience and there is little sharing of design knowledge among developers [Beck et al., 1996].

*General* design patterns —like the ones from the Gang of Four [Gamma et al., 1995], and the POSA [Buschman et al., 1996a] catalogs— are being more and more widely used in sound and music computing. This can be appreciated, for instance, in academic papers describing audio systems, where there is a growing tendency of documenting the overall system design in terms of *general* design patterns. But also in open-source projects, both in discussions on projects mailing-lists and in code documentation.

Nevertheless, the fact is that do not exist any (published) catalog of design pattern in the sound and music computing domain. The present work is an attempt to change this situation and goes in the same direction as other very recent efforts: Aucouturier presented several patterns for Music Information Retrieval in his thesis [Aucouturier, 2006]. And Roger B. Dannenberg and Ross Bencina presented several patterns on audio and real-time in a ICMC 2005 workshop —though we are not aware that they have been published. In the border line of the domain we found music composition patterns [Borchers, 2000] and, finally, a pattern language for designing patches for modular digital synthesisers [Judkins and Gill, 2000].

Apart from spreading the design *best practices*, collecting audio patterns can serve to another goal: record *innovative* software designs for critical examination which might, eventually, become new *best practices*.

Analysis reuse in the form of models have been actively used, mainly in models of computations (or metamodels), and have been object of formal study during

the last decades [Hylands et al., 2003]. Though being useful for the analysis of the system under development, models give little, or no clue at all, on how their requirements can be designed and implemented successfully.

## Translating data-flow models into concrete designs

Data-flow (meta)models and the process paradigm uses mathematical graphs for modeling its functionalities. They have a relatively long tradition in the area of system engineering, and they are suitable for formal study, providing each model with a set of guarantees like bounded memory, scheduling properties, etc. Since sound and music systems have a strong heritage on DSP systems, also do have it on data-flow models. Specially on Data-flow Process Networks and its variants [Parks, 1995].

There have been recent efforts for customizing these traditional metamodels into the multimedia systems requirements, using the object-oriented paradigm. The Metamodel for Multimedia Systems (4MS) is readily-usable for analysis in the sound and music domain [Amatriain, 2004]. Though the object-oriented approach of 4MS facilitates the translation of models into concrete implementations, in practice there is still a big leap to do. Our experience developing data-flow frameworks for sound and music indicates that many design problems and insights can not be effectively captured only using models. They are abstractions by definition and have to favor a holistic view of system features versus the design aspects.

During the last 10 years have been attempts to write patterns related with data-flow architectures. They were compiled and extended in a Data-flow Pattern Language [Manolescu, 1997]. The patterns from Manolescu are quite high-level (not implementation oriented) and, in general, applicable in our domain. However, problems in sound and music systems tends to be quite specific, and thus, require specific patterns. Each pattern defines a set of forces or quality-of-service to be optimized and its solution results in optimizing a set of forces while de-optimizing another set. Since the sound and music domain often imposes specific quality-of-service (for instance, real-time constrains and efficient management of audio samples), this implies that specific patterns are needed.

Another related problem is the lack of common vocabulary for expressing designs of frameworks in our domain. One one hand, a framework can be viewed as the implementation of a system of design patterns, so design patterns may be also employed in its documentation [Appleton, 1997]. On the other hand, several frameworks exists for sound and music computing using a data-flow architecture. For example, Supercollider, CSL or CLAM [McCartney, 2002, Pope and Ramakrishnan, 2003, Amatriain and Arumí, 2005]. Though, sometimes, they already incorporate general patterns in their documentation, the domain specific designs are not easily documented —if documented at all— because of a lacks of appropriate patterns. In consequence, frameworks are not as easy to understand and to compare as it might, should they share a common design vocabulary made of domain patterns.

## 1.2 The Solution

Our solution to the previous problems is the proposal of a pattern language for sound and music computing.

Simply stated, a pattern is a proven solution to a recurring design problem. It pays special attention to the context in which is applicable, to the competing "forces" it needs to balance, and to the positive and negative consequences of its application. A pattern language, as described in [Borchers, 2000], is a comprehensive collection of patterns organized in a hierarchical structure. Each pattern references higher-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after the current one to further refine the solution.

All the patterns presented in this catalog fits within the generic architectural pattern defined by Manolescu as the Data Flow Architecture pattern (figure 1.1). This pattern addresses how to design a system which performs a number of sorted operations on similar data elements in a flexible way so they can dynamically change the overall functionality without compromising performance.

It is important to note that the Data Flow Architecture pattern does not impose any restrictions on issues like message passing protocol, module execution scheduling, data token implementation, etc. All these aspects should be addressed in other

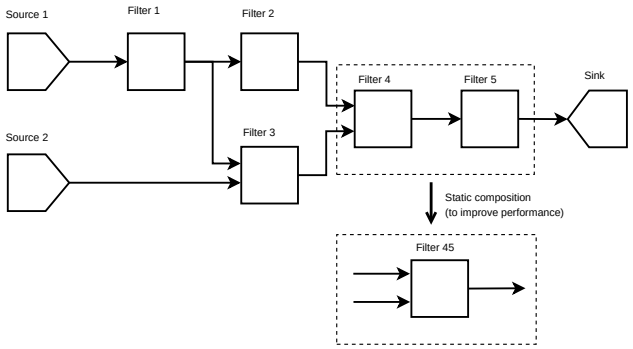


Figure 1.1: Data-flow Architecture

fine-grained and orthogonal design patterns, like the 10 patterns proposed in our pattern language. Our patterns focus on the following three aspects:

1. How to organize different kinds of modules connections.
2. How to transfer tokens between modules allowing a great flexibility.
3. How humans can interact with the data-flow networks.

An example of a system that may be optimally designed (and documented) almost completely using these patterns —plus the general or "traditional" ones— is the following: A system performing real-time Spectral Modeling Synthesis (a technique described in [Serra, 1990]) which implies performing two parallel FFT/IFFTs each consuming audio sample blocks of different size. Of course each module of the data-flow network encapsulates specific algorithms and techniques that falls out of the scope of this thesis. However, such specific techniques are not considered software design and are traditionally well recorded in scientific papers using appropriate means for documenting algorithms and signal processing techniques.

The main limitation of our solution is that our patten language is not comprehensive. Several important aspects of data-flow audio systems are not covered, for example the question of latency calculation (how long it take for input to propagate to output), or the propagation of tokens time between modules. The relevance of such gaps, however, will depend very much on the requirements of the application at hand.

Therefore, if we are strict with the definition, our pattern catalog should not be considered a pattern *language* because it does not cover all the problems of a domain.[1] However, the other property of a pattern language do apply: our patterns are organized hierarchically, making explicit the order in which they can be "instantiated" (figure 1.2). Moreover, they have been crafted in a fine grained way and work synergistically among them and with the existing ones from Manolescu. Therefore, we expect that new pattern will also be easily incorporated into the pattern "language-in-progress".

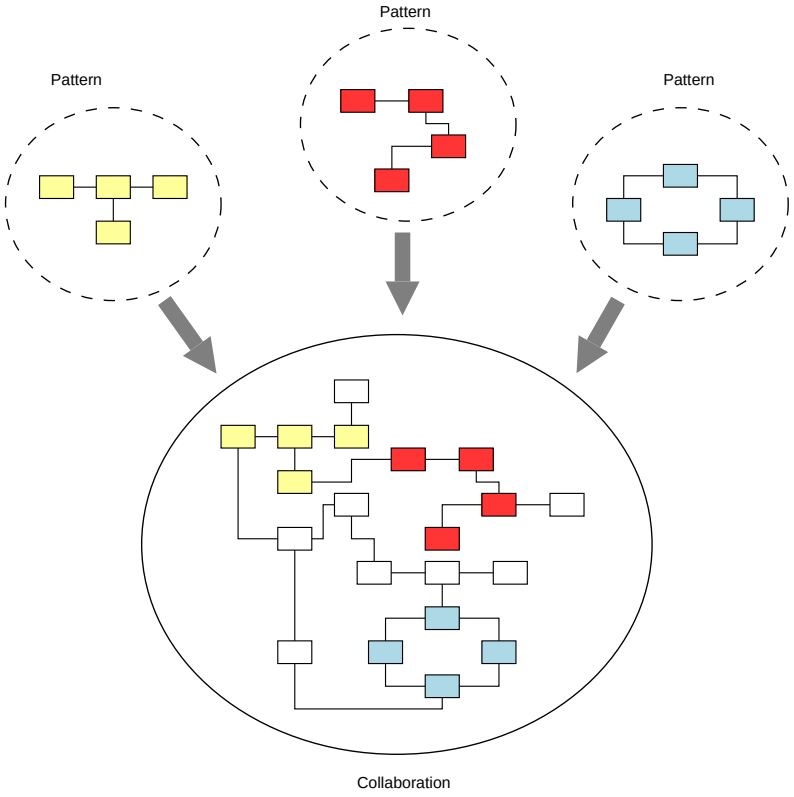


Figure 1.2: Pattern Instantiation

The difficulties of creating new patterns should not be underestimated. The teaching component of patterns —mostly corresponding to the description and resolution of forces and the consequences of application— is the most important

[1]For simplicity, we are not adhering to the strict definition; though we do in the tile.

and also the hardest part [Vlissides, 1998]. Moreover, growing a collection of highly-related patterns, while keeping them independent, involves many iterations and rewritings. Quoting [Beck et al., 1996]:

> The availability of a catalog of design patterns can help both the experienced and the novice designer recognize situations in which design reuse could or should occur. Such collection is time-consuming to create, but it is our experience that the invested effort pays off.
>
> (...) The pattern community is sufficiently enthused about the prospective advantages to be gained by making this design knowledge explicit in form of patterns, that hundreds of patterns have been written, discussed and distributed.

## 1.3 The Method

"Engineering disciplines have large bodies of theory accumulated behind them. But software engineering has a much shorter history than most engineering fields. Consequently, software engineers don't "calculate" software designs. Instead, they follow guidelines and good examples of working designs and architectures that help to make successful decisions. Therefore in the context of software engineering, communicating experience, insight, and providing good examples are important tasks." [Szyperski, 1998]

Our research provides elements of reusable design for building object-oriented sound and music data-flow systems. Consequently we have taken an approach that is appropriate for this objective. The process of creating new patterns starts when you have in depth experience on a particular area. You need to understand the trade-off of forces (quality-of-services) to be optimized in a particular area. But also you need sufficient breadth to understand the general aspects of the solutions to abstract them into a generalized solution. This process is called pattern mining (see figure 1.3).

"Pattern mining is not so much a matter of invention as it is of discovery —seeing that this solution in some context is similar to that solution in another context and abstracting away the specifics of the solutions. To be considered a use-

ful pattern, it must occur in different contexts and perform a useful optimization of one or more qualities-of-service." [Douglass, 2003]

Our patterns have been mined studying several open-source domain tools for application building: Aura, SndObj, OSW, STK, CSL, Supercollider and Marsyas [Dannenberg and Brandt, 1996a, Lazzarini, 2001, Chaudhary et al., 1999, Cook and Scavone, 1999, Pope and Ramakrishnan, 2003, McCartney, 2002, Tzanetakis and Cook, 2002] respectively. But mainly, they came from our experience building and evolving the CLAM framework and its many applications.

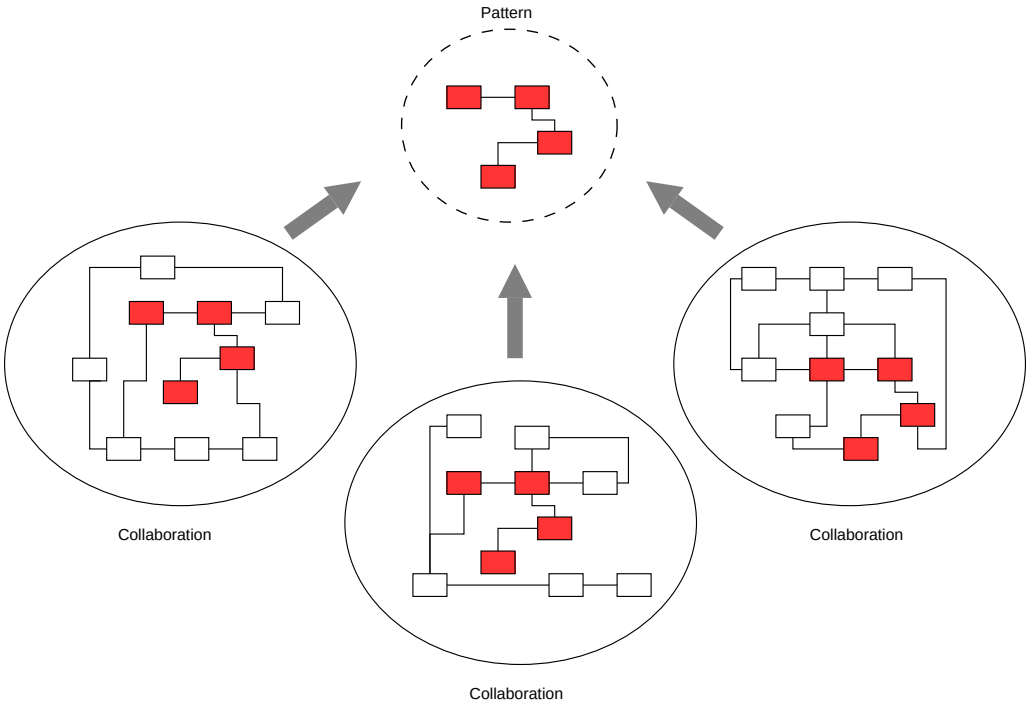


Figure 1.3: Pattern Mining

The patterns are evaluated assessing their usefulness in several use cases in different real-life applications. We show that most of the patterns can be found in different tools while few others only in the CLAM framework. However, we also show that the CLAM framework has demonstrated its adaptability —with many specific applications— to several scenarios within the sound and music domain such as real-time processing and synthesis and off-line audio analysis.

## 1.4 Contributions

The data-flow pattern language for sound and music computing allows software developers to use tested design solutions into their systems. It enables applying systematic solutions to domain problems with predictable consequences (trade-offs), as well as efficiently communicate and document design ideas. These characteristics represents a significant departure from other approaches that focus on reusing code (libraries, frameworks) or reusing analysis (metamodels).

In this thesis I will:

- Propose a (non-comprehensive) pattern language addressing the following aspects of sound and music data-flow architectures:
  - *General Data-flow Patterns:* Address problems about how to organize high-level aspects of the data-flow architecture, by having different types of modules connections.
  - *Flow Implementation Patterns:* Address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general data-flow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in those patterns.
  - *Network Usability Patterns:* Address how humans can interact with data-flow networks without compromising the network processing efficiency.
- Demonstrate that design patterns provides useful design reuse in the domain of sound and music computing.
- Show that all the patterns can be found in different applications and contexts.
- Show how design patterns are useful to communicate, document and compare designs of audio systems.

## 1.5 Thesis Organization

This thesis is structured as follows. Chapter 2 introduces the necessary background on object-oriented, patterns and graphical models of computation. Chapter 3 introduces related work somehow tries to solves the stated problem. It discusses sound and music specific models and an existing data-flow pattern language. Chapters 4, 5 and 6 discuss the proposed data-flow design pattern for sound and music. Chapter 7, organizes the catalog using pattern relations. Chapter 8, use case studies to provide qualitative evaluation of patterns. Finally, Chapter 9 draws conclusions and discusses open issues and future lines for a PhD thesis.

# Part I

# State of the Art

# Chapter 2

# Background

This chapter introduces the object technology which we used in the proposed design patterns. Moreover, design patterns born whithin the object technology community so they are related concepts. Since this is a thesis about audio patterns, this chapter introduces patterns and delineates what a pattern is and what is not. Our contributions focus on data-flow systems for sound and music. Thus, here we introduce the graphical models of computation.

## 2.1 Object Orientation

Booch defines Object-oriented programming as "a method of implementation in which programs are organized as cooperative collections of objects, each of which represents an instance of some class, and whose classes are all members of a hierarchy of classes united via inheritance relationships." [Booch, 1994]

Objects act on each other, as opposed to a traditional view in which a program may be seen as a collection of functions, or a list of instructions. Each object is able to receive messages, process data and send messages to other objects. Thus, objects can be regarded as actors with a distinct role or responsibility.

An *object* is a real-world or abstract entity made up of an identity, a state, and a behavior. A *class* is an abstraction of a set of objects that have the same behavior and represent the same kind of instances. The object-oriented paradigm can be deployed in the different phases of a software life-cycle and the *UML* language

supports most of the activities contained in them.

An object-oriented language supports two characteristic features: encapsulation and inheritance. Abstraction is the process of identifying relevant objects in the application and ignoring the irrelevant background. Abstraction delivers re-usability and information hiding through encapsulation. Encapsulation consists in hiding the implementation of objects and declaring publicly the specification of their behavior through a set of attributes and operations. The data structures and methods that implements these are private to the objects.

Object types or classes are similar to data types and to entity types with encapsulated methods. Data and methods are encapsulated and hidden by objects. Classes may have concrete instances, also known as objects.

Inheritance is the ability to deal with generalization and specialization or classification. Subclasses inherit attributes and methods from their super-classes and may add others of their own or override those inherited. In most object-oriented programming languages, instances inherit all and only the properties of their base class. Inheritance delivers extensibility, but can compromise re-usability.

Objects communicate only by message passing. Polymorphism —having many forms— means the ability of a variable of function to take different forms at run time, or more specifically the ability to refer to instances of various classes.

The benefits arising from the use of objects technology are summarized by Graham in [Graham, 1991]

> Reusability, extensibility and semantic richness. Top-down decompositon can lead to application-specific modules and compromise reuse. The bottom-up approach and the principle of information hiding maximize reuse potential. Encapsulation delivers reuse.
>
> Polymorphism and inheritance make handling variation and exceptions easier and therefore lead to more extensible systems. The open-closed principle is supported by inheritance. Inheritance elivers extensibility but may compromise reuse.
>
> Semantic reichness is provided by inheritance and other natural structures, together with constraints and rules concerning the meaning of objects in context. This also compromises reuse and must be carefully

managed.

Furthermore, proponents of the Object Oriented Programming claim that is easier to learn, simpler to develop and to maintain, lending itself to more direct analysis, coding, and understanding of complex situations and procedures

As Allan Kay observes in [Kay, 1993]

> Though it has noble ancestors indeed, Smalltalk's contribution is anew design paradigm —which I called object-oriented— for attacking large problems of the professional programmer, and making small ones possible for the novice user. Object-oriented design is a successful attempt to qualitatively improve the efficiency of modeling the ever more complex dynamic systems and user relationships made possible by the silicon explosion.

## 2.2 Design Patterns

A *design pattern* is a particular form of recording design information such that designs which have worked well in the past can be applied again in similar situations in the future by others. The form consists of structured prose and diagrams —usually UML diagrams: classes, objects, sequences, etc. The pattern identifies a problem, a set of *forces* or constraints —usually in conflict—, a solution that resolves the forces, and consequences that makes explicit how the forces are resolved and what is the resulting context.

A definition for patterns that become quite popular is the one inspired by Christopher Alexander "A pattern is a solution to a problem in a context.". However, Vlissides points out [Vlissides, 1998] three relevant things are missing from this definition:

1. *Recurrence*, which makes the solution relevant in situations outside the immediate one.

2. *Teaching*, which gives you the understanding to tailor the solution to a variant of the problem. (Most of the teaching in real patterns lies in the description and resolution of forces, and/or the consequences of its application.)

3. A *name* by which to refer to the pattern.

### 2.2.1 A Brief History of Design Patterns

In the 1960's, building architects were investigating automated, computerized building design. The mainstream of this movement was known as modular construction, which tries to transform requirements into a configuration of building modules using computerized rules and algorithms. The architect Christopher Alexander broke with this movement, noting that the great architectures of history where not made from rigorous, planned designs, but that their pieces were custom-fit to each other and to the building's surroundings. He also noted that some buildings were more aesthetically pleasing than others, and that these aesthetics were often attuned to human needs and comforts. He found recurring themes in architecture, and captured them into descriptions (and instructions) that he called patterns and pattern languages [Alexander, 1977]. The term "pattern" appeals to the replicated similarity in a design, and in particular to similarity that makes room for variability and customization in each of the elements. "Thus *Window on Two Sides of Every Room* is a pattern, yet it prescribes neither the size of the windows, the distance between them, their height from the floor, nor their framing (though there are other patterns that may refine these properties)." [Coplien, 1998]

Over the decade of the 1990's, software designers discovered analogies between Alexander patterns and software architectures. The first work on design pattern had their origin in the late 1980's when Ward Cunningham (the father of the wiki) and Kent Beck (best known for its extreme programming agile methodology) documented a set of patterns for developing elegant user interfaces in Smalltalk[Beck, 1988]. Few years later, Jim Coplien developed a catalog of language-specific C++ patterns called *idioms*. Meanwhile, Erich Gamma collected recurring design structures while working on the ET++ framework [Weinand et al., 1989] and his doctoral dissertation on object-oriented sofware development. These people and others met at a series of OOPSLA workshops starting in 1991. Draft versions of the first pattern catalog were matured during 4 years and eventually formed the basis for the first book on design patterns called

*Design Patterns* [Gamma et al., 1995] that appeared in 1995. It was received with enthusiasm and the authors were given the name of Gang-of-Four. In the summer of 1993, a small group of pattern enthusiasts formed the "Hillside Generative Patterns Group" and subsequently organized the first conference on patterns called the "Pattern Languages of Programming" (PLoP) in 1994.

Patterns have been used for many different domains: development processes and organizations, testing, architecture, etc. Apart from *Design Patterns*, other important pattern books include *Pattern-Oriented Software Architecture: A System of Patterns* [Buschman et al., 1996b] —also called the POSA book, authored by five engineers at Siemens; and the book series entitled *Pattern Languages of Program Design* with five volumes to the date.

### 2.2.2 Pattern Misconceptions

One of the most recurring misconceptions about patterns is to try to reduce them to something known, like rules, programming tricks, data structures...

John Vlissides, one of the Gang of Four, comments in his book *Pattern Hatching* [Vlissides, 1998]:

> Patterns are not rules you can apply mindlessly (the teaching component works against that) nor are they limited to programming tricks, even the "idioms" branch of the discipline focuses on patterns that are programming language-specific. "Tricks" is a tad pejorative to my ear as well, and it overemphasizes solution at the expense of problem, context, teaching, and naming.

Since software patterns grew inside the object-oriented community to record object-oriented design principles, they are often seen as limited to object-oriented design. However, patterns capture expertise and the nature of that expertise is left open to the pattern writer. Certainly there's expertise worth capturing in object-oriented design — and not just design but analysis, maintenance, testing, documentation, organizational structure, and on and on. As Vlissides recognises: "the highly structured style the GoF used in *Design Patterns* is very biased to its domain (object technology), and it doesn't work for other areas of expertise.

Clearly, one pattern format does not fit all. What does fit all is the general concept of pattern as a vehicle for capturing and conveying expertise, whatever the field."

Not every solution, algorithm, best practice, maxim, or heuristic constitutes a pattern; one or more key pattern ingredients may be absent. Even if something appears to have all the requisite pattern elements, it should not be considered a pattern until it has been verified to be a *recurring phenomenon*. Some feel it is inappropriate to call something a pattern until it has undergone some degree of scrutiny ore review by others. [Appleton, 1997]

Documenting good patterns can be an extremely difficult task. To quote Jim Coplien [Coplien, 1998], good patterns do the following:

- It solves a problem: Patterns capture solutions, not just abstract principles or strategies.
- It is a proven concept: Patterns capture solutions with a track record, not theories or speculation.
- The solution isn't obvious: Many problem-solving techniques (such as software design paradigms or methods) try to derive solution from first principles. The best patterns generate a solution to a problem indirectly — a necessary approach for the most difficult problems of design.
- It describes a relationship: Patterns don't just describe modules, but describe deeper system structures and mechanisms.
- The pattern has a significant human component: All software serves human comfort or quality of life; the best patterns explicitly appeal to aesthetics and utility.

### 2.2.3 Patterns, Frameworks and Architectures

The practical nature of patterns themselves and the people writing and using patterns should not be underestimated. As Ralph Johnson observed [Beck et al., 1996]:

> One of the distinguishing characteristics of computer people is the tendency to go "meta" at the slightest provocation. Instead of writing programs, we want to invent programming languages, we want to

> create systems for specifying programming languages. There are many good reasons for this tendency, since good theory makes it a lot easier to solve particular instances of the problem. But if you try to build a theory without having enough experience in the problem, you are unlikely to find a good solution. Moreover, much of the information in design is not derived from first principles, but obtained by experience.

Kent Beck and Ralph Johnson points to the reasons why patterns are powerful tools in the design process. [Beck and Johnson, 1994]

> Design is hard. One way to avoid the act of design is to reuse existing designs. But reusing designs requires learning them, or at least some parts of them, and communicating complex designs is hard too. One reason for this is that existing design notations focus on communicating the "what" of designs, but almost completely ignore the "why". However, the "why" of a design is crucial for customizing it to a particular problem. We need ways of describing designs that communicate the reasons for our design decisions, not just the results.

One approach to improving design, currently receiving attention primarily outside the object community, is the idea of "architecture" An architecture is the way the parts work together to make the whole. The way architectures are notated, applied, and discovered are all topics of active research.

A closely related idea inside the object community is that of "framework". A framework is the reusable design of a system or a part of a system expressed as a set of abstract classes and a way instances of (subclasses of) those classes collaborate. Frameworks are a particular way of representing architectures, so there are architectures that can't be expressed as frameworks. Nevertheless, the two ideas overlap. Both are attempts to reuse design, and examples of one are sometimes used as examples of the other.

Beck and Johnson were pioneers of object-oriented frameworks. They observed that frameworks, could be explained as a set of interrelated patterns. Thus, frameworks are a good source for pattern mining. [Beck and Johnson, 1994]

> The HotDraw architecture is not magic, but is the logical result of a set of design patterns. In the past, we have explained the architecture

> as "Drawing, Figure, Tool, and Handle". The pattern-based derivation puts each of theses classes in perspective. It explains exactly why each was created and what problem it solves. Presented this way, HotDraw becomes much easier to re-implement, or to modify should circumstances so warrant. This is a completely different approach to describing the design of a framework than more formal approaches like Contracts. The more formal results only explain what the design is, but a pattern-based derivation explains why. When we attempted to derive HotDraw from the patterns described in the Design Pattern Catalog [Gamma et al., 1995], we immediately realized that the Catalog had none of the graphics patterns that HotDraw would need. It turned out that it did not have the Editor pattern, either, so the derivation of HotDraw showed us that we need to describe a new object-oriented design pattern. This is similar to the proof process in mathematics, where the presentation of a proof hides most of its history, and where advances in mathematics are often caused by break-downs in proofs. Catalogs of design patterns will mature as people try to explain designs in terms of patterns, and find patterns that are missing from the catalogs.

### 2.2.4 Empirical Studies

The need for reliable software has made software engineering an important aspect for industry in the last decades. The steady progress recently produced an enormous number of different approaches, concepts and techniques: the object oriented paradigm, agile software development, the open source movement, component based systems, frameworks and software patterns, just to name a few. All these approaches claim to be superior, more effective or more appropriate in some area than their predecessors. However, to prove that these claims indeed hold and generate benefits in a real-world setting is often very hard due to missing data and a lack of control over the environment conditions of the setting.

In the join paper *Industrial Experiences with Design Patterns* [Beck et al., 1996] authored together by Kent Beck (First Class Software),

James O. Coplien (AT&T), Ron Crocker (Motorola), John Vlissides (IBM) and other 3 experts, authors describe the efforts and experiences they and their companies had with design patterns. The paper contains a table of the most important observations ordered by the number of experts who mentioned them. This can be interpreted as the results of interviewing experts. The top 3 observations mentioned by all experts where:

1. Patterns are a good communication medium.
2. Patterns are extracted from working designs.
3. Patterns capture design essentials.

The first observation is, indeed, the most prominent benefit of design patterns: In Design Pattern book [Gamma et al., 1995] by the Gang of Four two of the expected benefits are the design patterns provide "a common design vocabulary" and a "documentation and learning aid" which also focus on the communication process. The other two observations focus on the idea that design patterns describes best practices for important aspects of software design.

In [Prechelt et al., 1998] two controlled experiments using deign patterns for maintenance exercises are presented. For one experiment students were used to compare the speed and correctness maintenance work with and without design patterns used for the documentation of the original program. The result of this experiment was that using patterns in the documentation increases either the speed or decreases the number of errors for the maintenance task and thus seems to improve communication between the original developer and the maintainer via the documentation.

Another quantitative experiment is presented in [Hahsler, 2004]. They analyzed historic data describing the software development process of over 1000 open source projects in Java. They found out that only a very small fraction of projects used design patterns for documenting changes in the source code. Though the study had many limitations, e.g., the information on the quality of the produced code is not included. the results show a correlation between use of patterns and project activity, and that design patterns are adopted for documenting changes

and thus for communicating in practice by many of the most active open source developers.

## 2.3 Graphical Models of Computation

*Models of computation* are abstract representations of a family of related systems. Thus, they are not simple models (that represent concrete systems) but models of a family of models; that is, metamodels. For sound and music computing, the most useful Models of Computation are those that belong to the category of *Graphical Models of Computation* that are metamodels expressed using (mathematical) graphs. Many Graphical Models of Computation are characterized by assigning a concrete semantic to arcs and nodes and by restricting the general structure of the graph.

Many Graphical Models of Computation exist for different purposes and with different features. For example: Queuing Models, Finite State Machines, State Charts, Petri Nets, Processing Networks and Data-flow Networks. Each of these can have many variants that can be found in the context of the Ptolemy project [Hylands et al., 2003] [1].

Using the proper Graphical Model of Computation improves de development process and yields a better analysis of the properties of the system under design. Selecting the appropriate Graphical Model of Computation depends on the purpose and requirements of the system to develop but the choice is also generally conditioned by the application domain. Flexible sound and music systems, for instance, will generally benefit from Data-flow Networks — while, say, control-intensive applications will benefit from Finite State Machine.

We will now give a brief description of those metamodels that are more suited for sound and music:

- *State Charts* consist on *states*, *events*, *conditions* and *actions*. Events and conditions cause transitions and there are AND and OR compositions of

[1] Ptolemy project include the following Graphical Models of Computations: Component Interaction, Communicating Sequential Processes, Continuous Time, Discrete Events, Distributed Discrete Events, Discrete Time, Synchronous Reactive, and Timed Multitasking can be found in the context of the Ptolemy project (see [Hylands et al., 2003]).

states.

- *Petri Nets* are suited for systems with concurrency, asynchronous messages, distribution, non-deterministic, and parallelism. It consists of *places*, *transitions* and *arcs* that connect them. A Petri net is executed by the firing rules that transmit the tokens from one place to another. Such firing rules gets enabled when each input place has a token inside.
- *Process Networks* (or Kahn Process Networks) is a concurrent model of computation that is a super set of data-flow models. Its graph is directed and each arc represents a queue for communicating tokens, and each node represents an independent, concurrent process.
- *Data-flow Networks* is a special case of Process Networks where nodes are actors that respond to *firing rules*

We are now going to describe the Data-flow Network metamodel since this is the one more related to our pattern language and implemented systems.

### 2.3.1 Data-flow Networks

Data-flow Networks is a Graphical Model of Computation very closely related to Process Networks. In this model arcs also represent queues. But now the nodes of the graph, instead of representing processes, represent *actors*. Instead of responding to the simple the blocking-read semantics of Process Networks, actors use *firing rules* that specify how many tokens must be available on every input for the actor to fire (see figure 2.1). When an actor fires, it consumes a finite number of tokens and produces also a finite number of output tokens. A process can be formed by repeated firings of a data-flow actor.

An actor may have more than one firing rule. The evaluation of the firing rules is sequential in the sense that rules are sequentially evaluated until at least one of them is satisfied. Thus an actor can only fire if one or more than one of its firing rules are satisfied. In general, though, synchronous data-flow actors have a single firing rule of the same kind: a number of tokens that must be available at each of the inputs. For example, an adder with two inputs has a single firing rule saying that each input must at least have one token.

As pointed out by [Parks, 1995] breaking down processes into smaller units such as data-flow actors firings, makes efficient implementations possible. Restricting the type of data-flow actors to those that have a predictable consumption and production pattern makes it possible to perform static, off-line analysis to bound the memory.

In Data-flow Networks instead of suspending a process on blocking read or non-blocking write, processes are freely interleaved by a scheduler that determines the sequence of actor firings. The biggest advantage is that the cost of process suspension and resumption is avoided[Lee and Park, 1995].

In many signal processing applications the firing sequence can be determined statically at compile time. The class of data-flow process networks where this is possible are called"synchronous data-flow networks" and will be commented in next section.

Data-flow graphs have data-driven semantics. The availability of operands enables the operator and hence sequencing constraints follow only from data availability. This feature has its limitations. The principal strength of data-flow networks is that they do not over-specify an algorithm by imposing unnecessary sequencing constraints between operators [Buck and Lee, 1994].

### 2.3.2 Synchronous Data-flow Networks

Synchronous Data-flow Networks (SDF) is a special case of Data-flow Networks in which the number of tokens consumed and produced by an actor is known before the execution begins. The same behavior repeats in a particular actor every time it is fired. Arcs can have initial tokens. Every initial token represents an offset between the token produced and the token consumed at the other end. It is a unit delay and is represented by a diamond in the middle of the arc. Figure 2.2 illustrates a Synchronous Data-flow Network.

Schedule can be performed statically. As the execution of the graph is going to be repeated the compiler should just construct one "complete cycle" of the periodic schedule. A "complete cycle" is defined as the sequence of actor firings that returns the graph to its original state. From the static information of the network we can construct a "topology matrix" that contains relations between

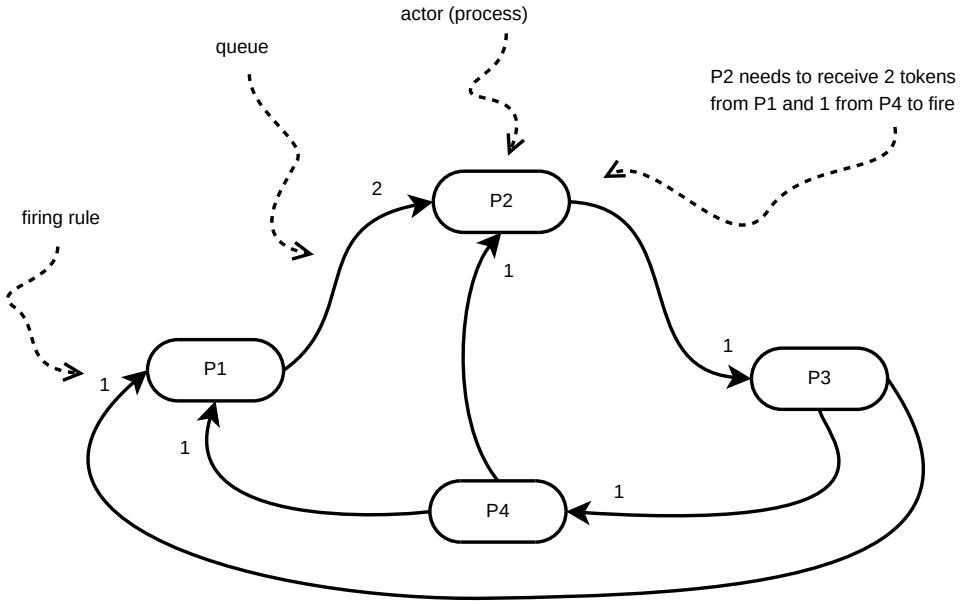


Figure 2.1: Data-flow Process Network

produced/consumed tokens in every arc. The element ij is defined as the number of tokens produced on the ith arc by the jth actor. Although it is only a partial info because there is no information on the number of initial tokens on each arc we can use the matrix to build the static schedule. For doing so we must find the smallest integer vector that satisfies the equation matrix*vector=0. It must be noted though that in complex networks these equations may not have a solution.

### 2.3.3 Boolean Data-flow Networks

Although SDF is adequate for representing large parts of systems it is rarely enough for representing an entire program. A more general model is needed to represent data-dependent iteration, conditionals and recursion. We can generalize synchronous data-flow to allow conditional, data-dependent execution and still use the balance equations. Boolean Data-flow Networks (BDF) is an extension of Synchronous Data-flow that allows conditional token consumption and production.

By adding two simple control actors like switch and select we can build conditional constructs like if-then-else and do-while loops. The switch actor gets a

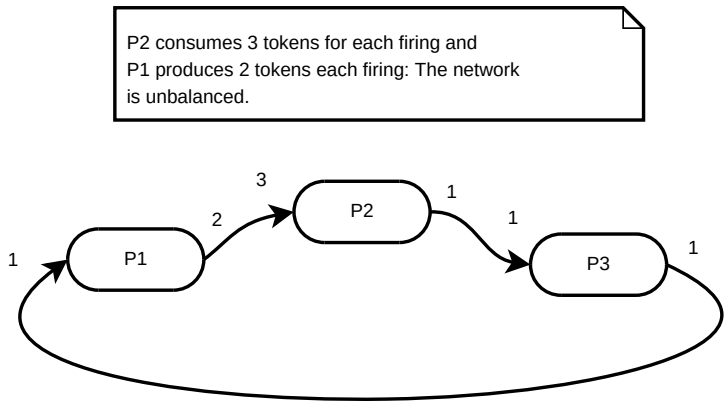


Figure 2.2: Synchronous Data-flow Process Network

control token and then copies a token from the input to the appropriate output, determined by the boolean value of the control token. The select actor gets a control token and then copies a token from the appropriate input, determined by the boolean value of the control token, to the output. These actors are not SDF because the number of produced/consumed tokens is not fixed and depends on an input boolean control [Buck and Lee, 1994].

### 2.3.4 Dynamic Data-flow Networks

Dynamic Data-flow Networks are a Boolean Data-flow Networks with one additional variation: the control actors mentioned in the BDF model can now read multiple token values and the data actors can be fired conditionally based on the control actors read. Although dynamic scheduling might also be used for any of the previous models, it is a must for this model as production/consumption rates may vary during execution.

Dynamic scheduling can be classified as *data-driven* (*eager* execution), *demand-driven* (*lazy* execution) or a combination of the two. In eager execution a process is activated as soon as it has enough data as required by any of its firing rules. In lazy execution a process is activated only if the consumer process does not have enough data tokens. When using bounded scheduling (see [Parks, 1995]) three rules must be applied: (a) a process is suspended when trying to read from an empty input, (b) a process is suspended when trying to write onto a full queue

and (c) on artificial deadlock, increase the capacity of the smallest full queue until its producer can fire.

### 2.3.5 Computation Graphs

Computation graphs are a model of parallel computation similar to Process Networks. It is represented by a finite graph with a set of nodes, each associated with a function, a set of arcs , where a branch is a queue of data directed from one node to another. Four non-negative integers (A, U, W and T) are associated with each arc. A is the number of tokens initially present in the arc, U is the number of tokens produced by the function associated with the node, W is the number of tokens consumed by the function associated with the node, T is a threshold that specifies the number of tokens that must be present in the arc before the function can be fired (obviously, $T \geq W$).

Questions of termination and boundness are solvable for Computation Graphs, which turn out to be a restricted version of PN. It is interesting to note that Synchronous Data-flow Networks is a special case of Computation Graphs where T=W for all arcs.

### 2.3.6 Context-Aware Process Networks

A special kind of Process Network introduced as an extension to the basic model but that is interesting for our purposes is that of Context-aware Process Networks [van Dijk et al., 2002]. This new model emerges from the addition of asynchronous coordination to basic Kahn Process Networks so process can immediately respond to changes in their context. This situation is very common in embedded systems.

In Context-aware Process Networks, stream oriented communication of data is done through regular channels but context information is sent through unidirectional register links (REG). These links have destructive and replicative behavior: writing to a full register overwrites the previous value and reading from a register returns the last value regardless if it has been read before or not. Thus, register links are an event-driven asynchronous mechanism. As a consequence, the behavior of a CAPN depends on the applied schedule or context.

A simple example of a system that can be effectively modeled by a context-aware network is a transmitter/receiver scheme in which the receiver needs to send information about its consumption rate to the transmitter so transmission speed can be optimized. The basic transmitter/receiver scheme can be implemented with a Kahn Process Network but in order to implement feedback coordination we need to use the register link provided by context-aware process networks.

Context-aware systems are indeterminate by nature. Unless the indeterminate behavior can be isolated, a composition of indeterminate components becomes a non-deterministic system, which is possible but not practical. Nevertheless as mentioned in [van Dijk et al., 2002] some techniques can be used in order to limit indetermination.

# Chapter 3

# Related Work

This chapter discusses the Metamodel for Multimedia Systems, a customization of a data-flow network for multimedia systems. The metamodel is readily-usable for object-oriented analysis in the sound and music domain. However, the translation from models to concrete implementations is still a big gap. This gap should be covered by design patterns. This chapter also discusses a generic Data-flow Pattern Language. Though it is very related to our work, it does not cover many aspects that are specific of sound and music.

## 3.1 Sound and Music Models

### 3.1.1 Metamodel for Multi Media Systems

4MS [Amatriain, 2004] stands for MetaModel for Multi-Media Systems, and was initially known as DSPOOM. [1] 4MS can be instantiated to describe any multimedia processing design, and that combines the advantages of the object-oriented paradigm with system engineering techniques and Graphical Models of Computation.

The 4MS metamodel is based on a classification of signal processing objects into two primary categories: *Processing* objects that operate on data and control, and *Processing Data* objects that passively hold media content. Data input to and

[1] DSPOOM stands for Digital Signal Processing Object-Oriented Metamodel

output from Processing objects is done through *Ports*, and control data is handled through the *Control* mechanism.

In 4MS, applications consist of networks of processing objects interchanging signal flow (via ports) and events (via controls).

Processing objects are the object-oriented encapsulation of a process or algorithm. They include support for synchronous data processing and asynchronous event-driven control processing as well as a configuration mechanism and an explicit life-cycle.

Processing objects can only process Processing Data objects. Processing Data classes should offer a number of services like: introspection (accept queries about which are its attributes); homogeneous interface, to operate independently of the concrete subclass; encapsulation of its internal data representation; persistence, automatically built-in; and Display facilities that allow for debugging and visualizing its content at any time.

The metamodel defines several class hierarchies (processing, port, control...) and a processing network composite model.

### 3.1.2 4MS as a Graphical Model of Computation

In his thesis, Amatriain classifies the 4MS metamodel as a *Context-aware Data-flow Network*. [Amatriain, 2004] First, it can be assimilated to *Data-flow Networks* because Processing objects can produce and consume different quantities of data tokens. The "firing rules" of Data-flow Networks are translated into region sizes in the 4MS. Second, the control mechanism introduces an extension the basic Process Network and Data-flow language. This extension is almost identical to that in *Context-aware Process Networks*. The coordination introduced by the context information in Context-aware Process Networks has the same requirements and features than 4MS control mechanism.

## 3.2 A Data-flow Pattern Language

Many pattern catalogs have been written on different domains. Some of them relates in more or less degree to the Graphical Models of Computation, and the

data-flow paradigm [Buschman et al., 1996a, Shaw, 1996]

others covers specific aspects of data-flow [Meunier, 1995, Edwards, 1995]; and some of them are specialized patterns for a particular domain [Posnak et al., 1996].

But the most complete catalog is given by Dragos-Anton Manolescu who gives a complete overview of software patterns applied to the data-flow model [Manolescu, 1997]. Based on the previous studies and on a different set of system examples, Manolescu organizes the existing patterns, redefines its granularity and identify 3 other patterns.

These patterns are not a theoretical approach to data-flow models but rather the result of an exhaustive analysis of existing software solutions. Therefore, they represent a key element to translate the model requirements into the software domain.

Data-flow is a very broad area, thus is not strange that we find different systems with conflicting quality-of-service requirements. The applicability of Manolescu's pattern language to the sound and music domain varies depending on the pattern. The most architectonic or high-level ones apply very well, but others have forces that are in conflict with those in the sound and music domain. Moreover, sound and music models have many specific requirements that are not covered at all by the general patterns.

The pattern language is composed by the following patterns that will be summarised in the following sections: Data-flow architecture, Payloads, Module data protocol, and Out-of-band/in-band partitions.

### 3.2.1 Data flow architecture

A variety of applications apply a series of transformations to a data stream. The architectures emphasize data flow and control flow is not represented explicitly. They consist of a set of *modules* that interconnect forming a new module or *network*. The modules are self-contained entities that perform generic operations that can be used in a variety of contexts. A module is a computational unit while a network is an operational unit. The application functionality is determined by: types of modules and interconnections between modules. The application could also be required to adapt dynamically to new requirements.

In this context, sometimes a high-performance toolkit applicable to a wide range of problems is required. The application may need to adapt dynamically or at run-time. In complex applications it is not possible to construct a set of components that cover all potential combinations. The loose coupling associated with the black-box paradigm usually has performance penalties: generic context-free efficient algorithms are difficult to obtain. Software modules could have different incompatible interfaces, share state, or need global variables.

The Solution is to highlight the data flow such that the application's architecture can be seen as a network of modules. Inter-module communication is done by passing messages (sometimes called tokens) through unidirectional input and output ports (replacing direct calls). Depending on the number and types of ports, modules can be classified into *sources* (only have output ports and interface with an input device), *sinks* (only have input ports and interface with output devices), and *filters* (have both input and output ports).

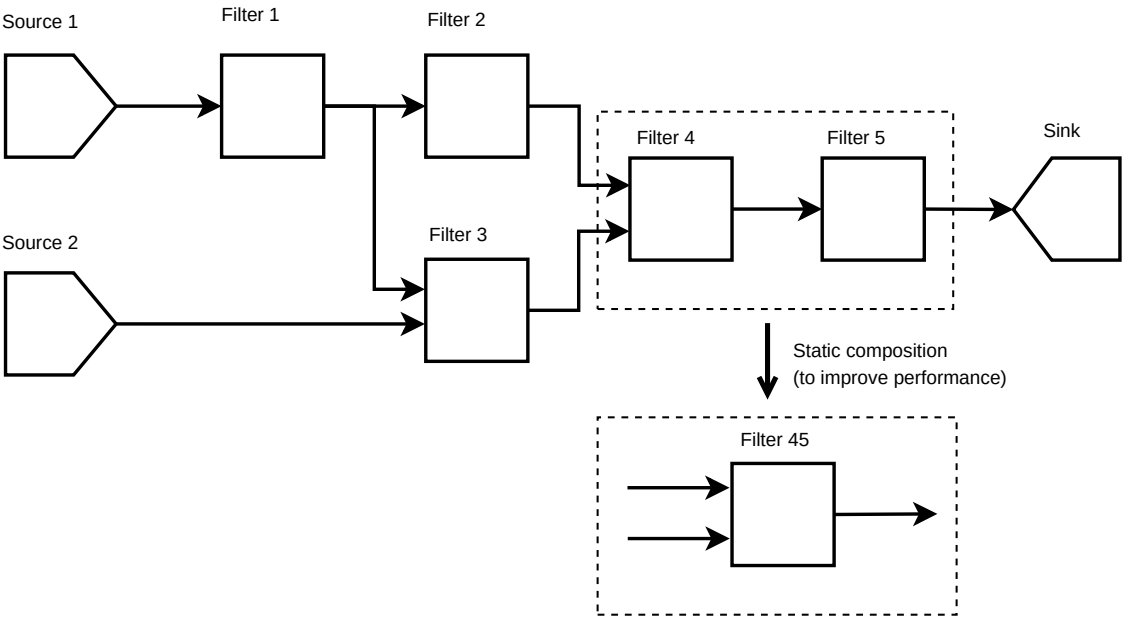


Figure 3.1: Data-flow architecture

Because any of the component depends only on the upstream modules it is possible to change output connections at run-time. For two modules to be connected the output port of the upstream module and the input port of the downstream

module must be plug-compatible. Having more than one data type means that some modules perform specialized processing. Filters that do not have internal state could be replaced while the system is running. The network usually triggers re-computations whenever a filter output changes.

In a network, adjacent performance-critical modules could be regarded as a larger filter and replaced with an optimized version, using the *Adaptive Pipeline* pattern [Posnak et al., 1996] which trades flexibility for performance. Modules that use static composition cannot be dynamically configured.

### 3.2.2 Payloads

In data-flow-oriented software systems separate components need to exchange information either by sending messages (payloads) through a communication channel or with direct calls. If it is restricted to message passing, payloads will encapsulate all kinds of information but components need a way to distinguish the type as well as other message attributes such as asynchronousity, priority... Some overhead is associated with every message transfer. Depending on the kind of communication, the mechanism must be optimized.

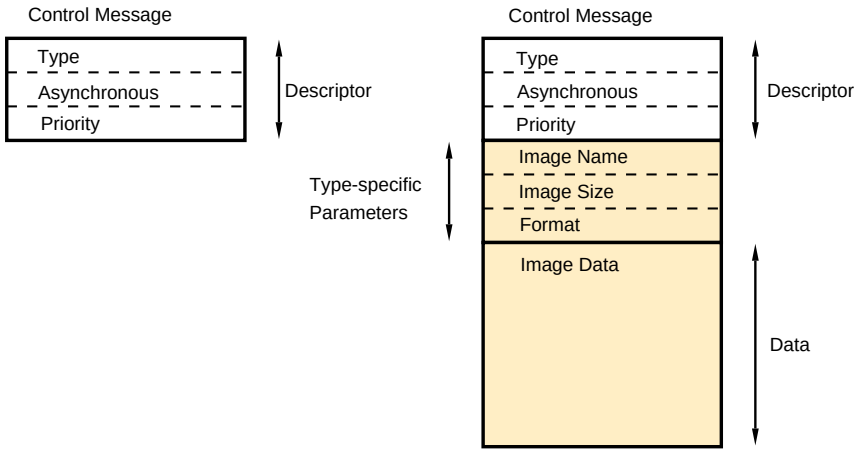


Figure 3.2: Different payloads and their components

Payloads give a solution to this problem. Payloads are self-identifying, dynamically typed objects such that the type of information can be easily identified.

Payloads have two components: a descriptor component and a data component. In the case where different components are on different machines, payloads need to offer serialization in order to be transmitted over the channel.

Payload copying should be avoided as much as possible using references whenever possible. If the fan out is larger than one, the payload has to be cloned. In order to reduce copies even in that case, the cloned copies can be references of the same entities and only perform the actual copy if a downstream receiver has to modify its input. If it is not possible to avoid copying there are two possibilities: shallow copy (copy just the descriptor and share the data component) and deep copy (copy the data component as well maybe implementing copy-on-write).

The greatest disadvantage of the payload pattern compared to direct call is its inefficiency, associated with the message passing mechanism. One way to minimize it is by grouping different messages and sending them in a single package.

A consequence of this pattern is that new message types can be added without having to modify existing entities. If a component receives an unknown token, it just passes it downstream.

### 3.2.3 Module data protocol

Collaborating modules pass data-blocks (payloads) but depending on the application, the requirements for these payloads could be very different: some may need asynchronous user events, some may have different priority levels, some may contain large amounts of data. On the other hand, sometimes the receiving module operates at a slower rate than the transmitter, to avoid data loss the receiver must be able to determine the flow control.

Besides, we must take into account a number of possible problems. Large payloads make buffering very difficult. Payloads with time-sensitive data have to be transferred in such a way that no deadlines are violated. Asynchronous or priorized events are sent from one module to another- Shared resources for inter-module communication might not be available or the synchronization overhead not acceptable. And flow control has to be determined by receiving module.

There are three basic ways to assign flow control among modules that exchange Payloads:

- Pull (functional): The downstream module requests information from the upstream module with a method call that returns the values as result. This mechanism can be implemented via a sequential protocol, may be multi-threaded and may process in-place. The receiving module determines flow control. It is applicable in systems where the sender operates faster than the receiver. This mechanism cannot deal with asynchronous or high-priority events.

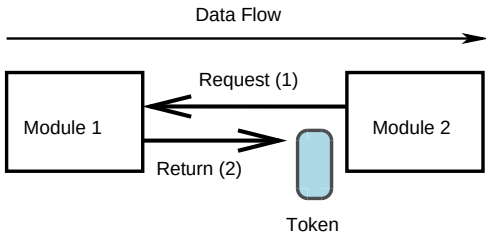


Figure 3.3: The pull model for inter-module communication.

- *Push* (event driven): The upstream module issues a message whenever new values are available. The mechanism can be implemented: as procedure calls containing new data as arguments; as non-returning point to point messages or broadcast; as high-priority interrupts; or as continuation-style program jumps. Usually the sending module does not know whether the receiver is ready or not. To prevent data loss the receiver can have a queue. If there are asynchronous or high-priority events, the queue must let them pass, else a simple queue can do.

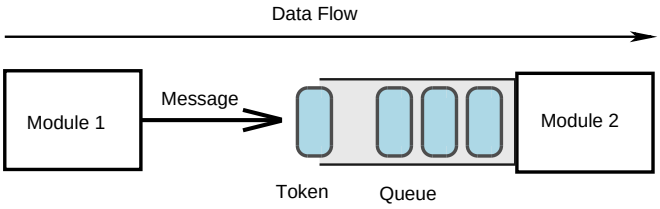


Figure 3.4: The push model for inter-module communication.

- *Indirect* (shared resources): Requires a shared repository accessible to both modules. When the sender is ready to pass a payload to the receiver, it writes in the shared repository. When ready to process, the receiver takes a payload from the repository. The sender and the receiver can process at different rates. If not all the payloads are required by the receiver, the upstream module can overwrite data.

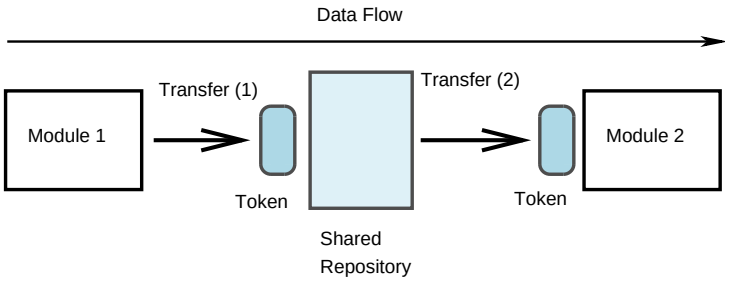


Figure 3.5: The indirect model for inter-module communication.

It must be noted though that having more than one input port complicates flow control and requires additional policies.

### 3.2.4 Out-of-band and in-band partitions

An interactive application has a dual functionality: first it interfaces with the user handling event-driven programming associated with the user interface and the response times have to be in the order of hundreds of milliseconds; second it handles the data processing according to the domain requirements

User actions are non-deterministic so user interface code has to cover many possibilities. Data processing has strict requirements and the sequence of operations (algorithm) is known before hand. Human users require response in the order of hundreds of milliseconds but applications emphasize performance that is irrelevant for the user interface. Generally, a large fraction of the running time is spent waiting for user input. The user interface code and data processing code are part of the same application and they collaborate with each other.

The solution is to organize the application into two different partitions:

- Out-of-band partition: typically responsible for user interaction.
- In-band partition: it contains the code that performs data processing. This partition does not take into account any aspects of user interaction

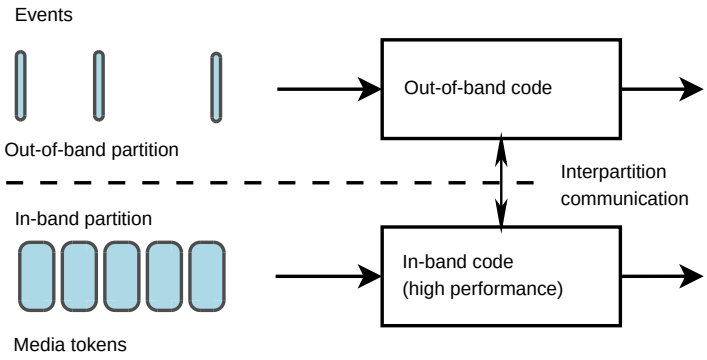


Figure 3.6: Out-of-band and in-band partitions within an application.

# Part II

# Contributions

# Chapter 4

# Introduction to Audio Data-flow Patterns

We have shown that exist previous efforts in building pattern languages for the data-flow paradigm. The following chapters offer an initial pattern catalog for data-flow systems in sound and music computing. All patterns presented in this catalog fits within the generic Data-flow Flow Architecture pattern.

The Data Flow Architecture pattern solves the problem by designing a system which performs some number of sorted operations on similar data elements (that we will call *tokens* ) in a flexible way so they can dynamically change the overall functionality without compromising performance. The pattern solution is an architecture that can be seen as a *network* of *modules* with strict interfaces at the module boundary, allowing a large number of possible combinations.

Modules read incoming tokens through their in-ports and writes them through their out-ports. Module connections are done by connecting out-ports to in-ports, forming a network.

In sound and music computing, tokens flow through modules in two different fashions: at regular (or almost regular) rate, which is known as a *stream* flow, and when they flow without any regularity, which is known as *event* flow. For example, the flow of data coming from an audio card is a stream flow, while the flow of note-on and note-off messages from a MIDI keyboard is an event flow.

Each module in a network is periodically *executed*, which means a call to the

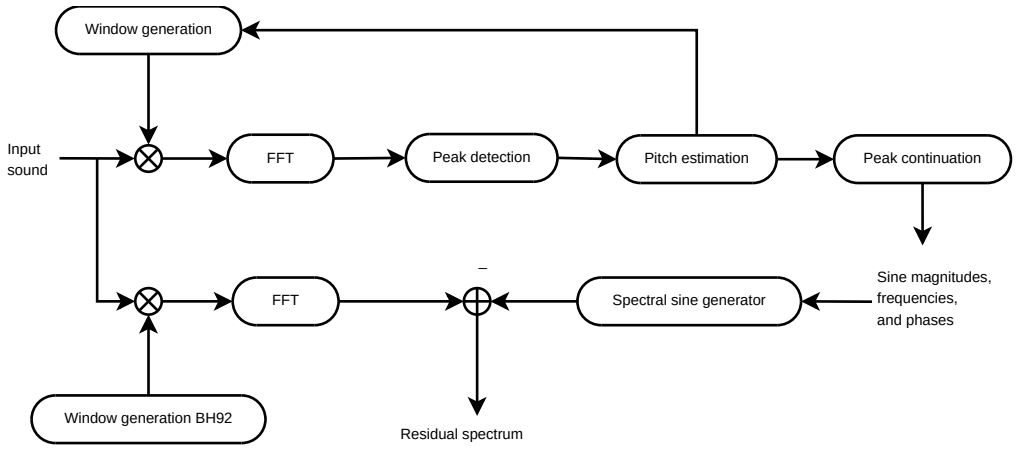


Figure 4.1: A use case for audio data-flow: the Spectral Modeling Synthesis.

module's *execution method* (also known as *module's algorithm*).

It is important to note that the Data Flow Architecture pattern does not impose any restrictions on issues like message passing protocol, module execution scheduling, or data token implementation. All these aspects imply different problems that can be addressed in other fine-grained patterns, like the ones in the present pattern language. This pattern granularity [Vlissides, 1998] proved very useful because we have been able to incorporate orthogonal patterns that work synergistically among them and with the existing ones from Manolescu.

The proposed patterns are inspired by our experience in the audio domain. And some patterns are clearly motivated by the requirements of the spectral processing. A use case that exemplifies its complexity is the analysis-synthesis using sinusoids plus residual (see figure 4.1), where different Fast Fourier Transforms are done consuming different number of tokens (audio samples) in parallel.

The following pattern structure has been chosen for all our patterns. Adherence to a structure facilitates browsing the catalog and comparing patterns.

**Context and Problem Statement** Sets the solution space. Defines what is an "admissible" solution and what is not. Problem statement is just the core statement of the problem. Should be punchy and easy to remember. But it is not different from context in essence.

**Forces** Do not define the solution space but gives criteria on what is a good solution and what is a bad one. In other words, the quality-of-services that we want to optimize.

**Solution** The architecture/design/implementation that solves the problem, without giving many justifications. The given solution should make clear that it belongs to the solution space.

**Consequences** They justifies why the solution is a good one in terms of the stated forces. That is, why all the forces are optimized (or resolved) or how the forces are balanced in case they are conflicting.

**Related Patterns** References higher-level patterns describing the context in which this pattern can be applied, and lower-level patterns that could be used to further refine the solution. As well as other used or similar patterns.

**Examples** Gives a list of real-life systems where the pattern can be found implemented.

Taking into account the previously introduced background, the patterns contributed in this thesis are presented in the next 3 chapters, organized in three categories:

- **General Data-flow Patterns:** Address problems about how to organize high-level aspects of the data-flow architecture, by having different types of modules connections.
- **Flow Implementation Patterns:** Address how to physically transfer tokens from one module to another, according to the types of flow defined by the *general data-flow patterns*. Tokens life-cycle, ownership and memory management are recurrent issues in those patterns.
- **Network Usability Patterns:** Address how humans can interact with data-flow networks without compromising the network processing efficiency.

# Chapter 5

# General Data-flow Patterns

## 5.1 Semantic Ports

### Context

Applications with a dataflow architecture consist on a directed graph of modules, like shown in figure 5.1. Is a very common case that a module receives tokens with different semantics. For example, a module that mixes $n$ audio channels will receive tokens of audio data corresponding to each channel. Identifying which token corresponds to each channel —the token semantics— is fundamental to produce output tokens containing the audio mix. The Payloads pattern described by Manolescu provides a solution to this problem consisting on adding a descriptor component into each token which provides the semantic information about the token, as well as type-specific parameters. The implication of applying Payloads is that incoming tokens needs to be dispatched according to its descriptor component, before doing any processing.

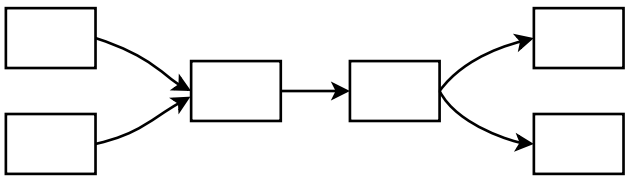

Figure 5.1: A directed graph of components forming a network

Tokens produced by a module may also have different semantics. One might want to send to a connected module only tokens with a given semantics and not all the produced tokens.

## Problem

How can a module manage tokens according to their semantics in order to deal with the incoming ones in different ways and send the produced ones to different destinations?

## Forces

- Module implementation should be as simple as possible, because modules are developed by different authors while general infrastructure is just implemented once by experienced programmers.
- Dispatching tokens adds complexity to module programming
- Module execution should be efficient in time, often real-time constrains are imposed.
- Dispatching tokens adds a run-time overhead.
- Token semantics fields on tokens add overhead
- Token semantics should be given by the module and they should not be restricted.
- Incoming might also have different priorities, and modules should consume the tokens with greatest priority first.

## Solution

Use different ports for every different token semantics in each module. So that modules have as many in-ports and out-ports as different input and output semantics are needed. Instead of connecting modules directly, connect modules by pairing out-ports with in-ports, as shown in figure 5.2. Module's execution method knows the semantics associated to each port, thus, it can obtain tokens of specific semantics just by picking the proper in-port. Because connections are done among

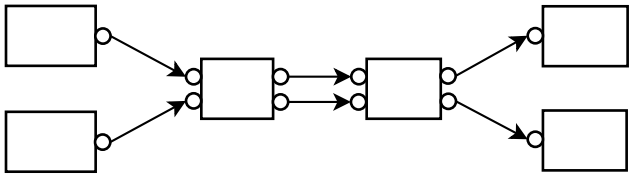

Figure 5.2: A network of components with multiple ports

ports instead of modules, a processed tokens will target the proper destination just by sending tokens through the proper out-port.

## Consequences

Tokens does not need to incorporate a description component. Module implementation is simplified because programming a token dispatcher regarding its semantics is not needed. Also, run-time penalty associated to the dispatching is avoided. The pattern solution implies that token semantics is not defined inside the tokens with a description component, but semantics is something intrinsic of the ports.

Retaking the audio mixer example; instead of having a "channel" field on each token arriving to the mixer, using the Semantic Ports pattern, we would have a mixer with $n$ different in-ports, each one receiving tokens of a single channel.

Tokens with different priorities should be routed to different in-ports. The module knows the priority of each in-port and so is able —in its execution method— to consume tokens in the right order.

## Related Patterns

Most patterns in this collection build on Semantic Ports: Driver Ports, Stream and Event Ports, and Cascading Event Ports are clear examples of separation of ports regarding its semantics.

Semantic Ports also relates to Payloads in the sense that the problems they solve are similar but, since they have different forces, they end up with different solutions.

Semantic Ports can handle different token types by using the Typed Connection pattern.

### Examples

CLAM uses Semantic Ports to separate different flows. Visual environments like Pure-Data (PD) [Puckette, 1997] or MAX/MSP [Puckette, 1991] also do. They ports separate both audio ("tilde") streams lower rate streams on their semantic. We find another good examples in Open Sound World (OSW) [Chaudhary et al., 1999] and the JACK sound server [Davis et al., 2004].

Anti-examples —systems that do not use *Semantic Ports* because they use other approaches— are also interesting to see for this pattern: Marsyas [Tzanetakis and Cook, 2002] and SndObj [Lazzarini, 2001], they do not use separated ports for its network connections but they do it at module level. SndObj modules, for instance, keeps a pointer to their connected producers and reads its output signal doing a direct call.

## 5.2 Driver Ports

### Context

Module execution on data-flow system is driven by the availability of flowing tokens. But not all token flows drive the execution.

Imagine a module which receives an audio signal and performs a low-pass filter with a given cutoff frequency. The audio signal is fed into the module with a constant rate but the cutoff values are fed seldom into the module. These cutoff values typically come from a sequencer module or a knob in the user interface. Each execution of the module must wait for the availability of new audio signal data. But there is not such dependency on the seldom received cutoff values, it just uses the last value. To summarize, whereas audio stream tokens drive the modules execution, the frequency event tokens does not.

## Problem

How can we make module execution depend on the availability of tokens on certain in-ports and not on others?

## Forces

- Concrete modules implementation should be simple.
- Visual programming tools should be able to distinguish the flow that drives the module execution from the one that does not.

## Solution

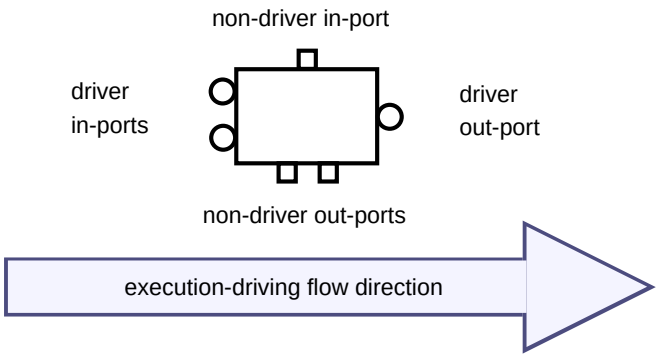


Figure 5.3: A representation of a module with different types of in-ports and out-ports

Allow the concrete module developer to define which are the driver in-ports and which are not. Give the modules a common interface from which external entities can know which are the drivers and which are not. The module execution will be enabled by the availability of enough tokens on the driver in-ports. Note that enabling is not the same as triggering. The network scheduling policy determines if a module will be executed as soon as it is able —in a *pull* strategy— or if it will be postponed until other module executions end.

**One implementation strategy** :

Figure 5.4 shows a class collaboration with two class hierarchies: *Module* and *Port*. This structure allows separate general infrastructure in base classes making the concrete classes simpler to implement —this is actually an example of *white box reuse* in frameworks. Some modules services are implemented in the base class, usually delegating to its ports, and thus freeing the concrete module writer form this responsibility. An important detail here, is that concrete modules own its ports, declared as normal member attributes. However, at construction time, they get "registered" to the module base class so that it can implement generic operations, independently of the concrete module. Examples of such operations are *ableToExecute* which can be useful to a firing manager; and *driverPorts*/*nonDriverPorts* which give the lists of driver and non-driver ports to, say, a GUI client.

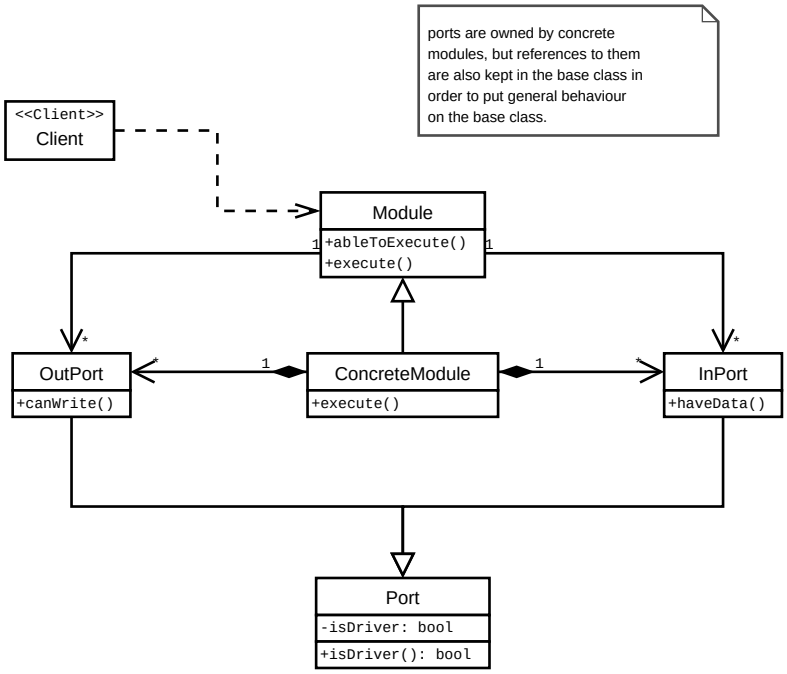


Figure 5.4: Separated Module and ConcreteModule classes, to reuse behaviour among modules

Other patterns like Stream and Event Ports and Typed Connections also benefit from using this class structure. However, each pattern enriches the *Port* and

*Module* base class interfaces to fit its needs.

## Consequences

Whether a module is ready to be executed or not can be checked without relying on the concrete module implementation because the visibility of the separation between driver and non-driver ports. This is safer and simplifies concrete module implementation.

Visual builder tools can distinguish driver and non-driver flows by identifying driver and non-driver ports and displaying them differently.

As mentioned in [Foote, 1988] module networks are often built with visual programming tools. Such tools should give the user a clear separation between stream ports and event ports, else, event connections might hide the main data-flow —the stream flow that drives the modules execution.

For example, CLAM's visual builder called Network Editor (see figure 5.5) uses horizontal connections (left to right) for driver flow, and vertical (top-down) connections for the non-driver flow.

Other visual builders takes different approaches. Open Sound World (OSW), for instance, paints the driver ports in green while the non-driver ports are gray. This can be appreciated —though if the copy is not colored it can be hard— in figure 5.6.

## Related Patterns

Driver Ports is strongly related with Stream and Event Ports. Actually, in most of the examined applications those ports that drive the execution coincide with the stream ports. However, they are better off being separate patterns because they solve orthogonal problems. Moreover, examples exist where driver ports and stream ports are totally independent.

Systems whose driver ports are stream ports that can produce and consume different number of tokens needs dynamic scheduling. Its design is explained in the Multi-rate Stream Ports pattern.

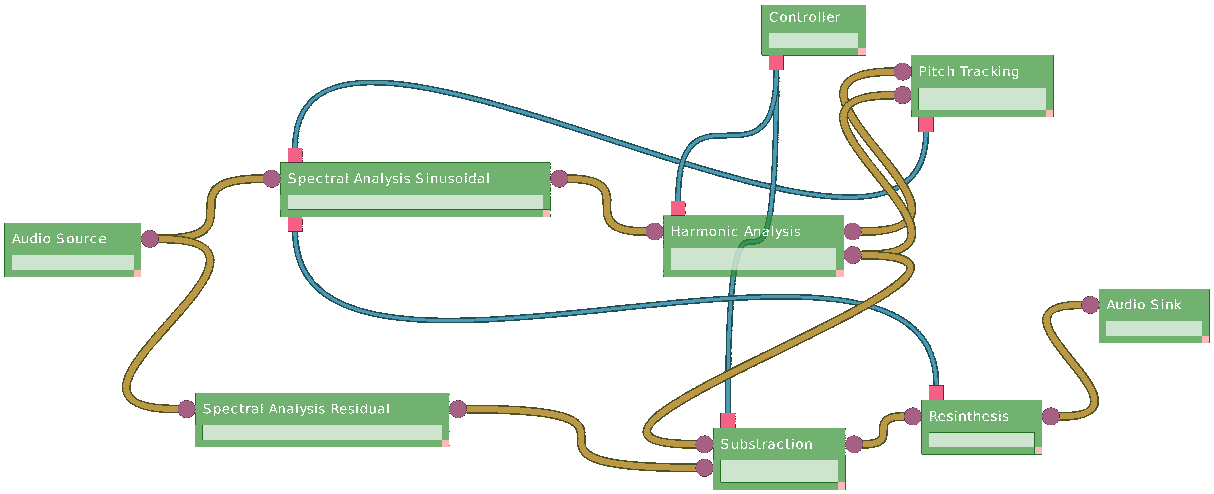


Figure 5.5: Screenshot of CLAM visual builder (NetworkEditor) doing SMS analysis-synthesis

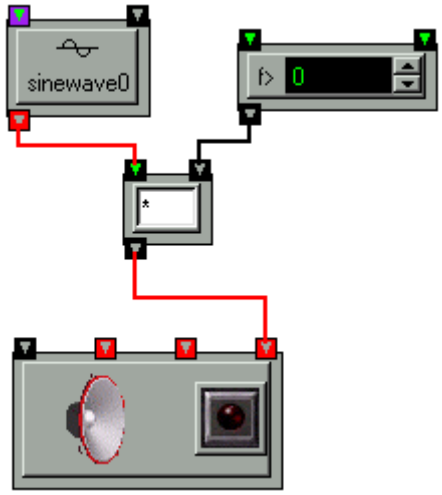


Figure 5.6: Screenshot of Open Sound World visual builder

## Examples

Pure Data (PD) [Puckette, 1997] and MAX/MSP [Puckette, 1991] are graphical programming environment for real-time musical applications with a widespread use among composers. Its ports are called *inlets* and *outlets* and they are visually arranged horizontally. With few exceptions (notably the "timer"), objects treat their leftmost inlet as "hot" in the sense that messages to left inlets can result in output messages. The rest of inlets are "cold" in that they only store the received message and do not trigger any execution. Thus, the "hot" or leftmost inlets are the driver ports. However, since modules have only one driver port and modules are executed at the time a token arrives at the driver port, the following problematic situation may occur when two modules are connected by more than one connection: the module might be triggered before receiving all its data because the "hot" inlet was not the last to receive the data. In order to avoid this output messages are —by convention— written from right to left and modules connections should be (visually) done without any crossing lines. Finally, PD and MAX/MSP is important example where driver ports do not coincide with stream ports.

Open Sound World (OSW) [Chaudhary et al., 1999] have a similar approach to PD and MAX/MSP but it does not limit the number of driver ports. In the JACK audio server [Davis et al., 2004] all ports are drivers. CLAM also uses

*Driver Ports* and restricts its drivers to be constant-rate stream ports.

## 5.3 Stream and Event Ports

### Context

In audio systems, two kind of flows exist: stream flow, when tokens flows at continuous (or almost continuous) rate; and event flow, when tokens flow with an unpredictable rate.

A module may receive tokens of both kinds —stream and event— coming from different sources. Moreover, streams may arrive at different rates. For example, a module may receive two audio samples streams one at 44100 Hz and the other at 22050 Hz. Figure 5.7 shows another example: a module is receiving two streams at different (though constant) rates and an irregularly distributed flow of events. Its output stream have the same rate as the second input stream.

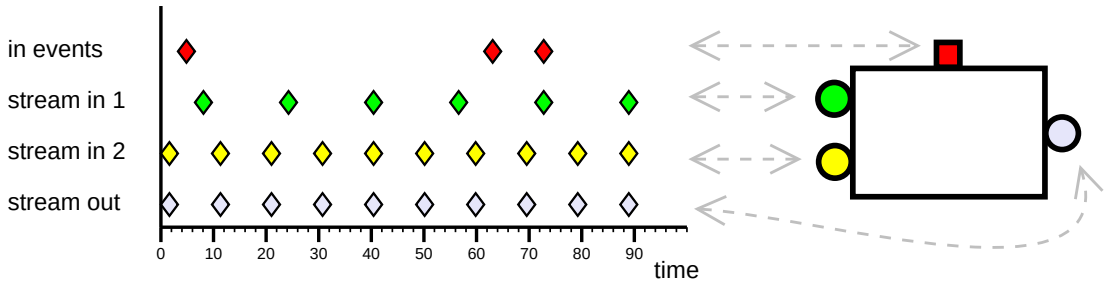


Figure 5.7: Chronogram of the arrival (and departure) time of stream and event tokens

Such a module consume its incoming tokens and then calls its execution method that will take the consumed tokens as its input. When receiving tokens at different rates, the module needs to synchronize all the incoming tokens prior to its processing. This synchronization can also be seen as a time-alignment of incoming tokens, and it implies knowing the time associated to each token. Here, is important to differentiate the time associated to the tokens, with the "real" time where the module is executed. The two kind of times might be totally different. Figure 5.8 illustrates the alignment of tokes of different nature.

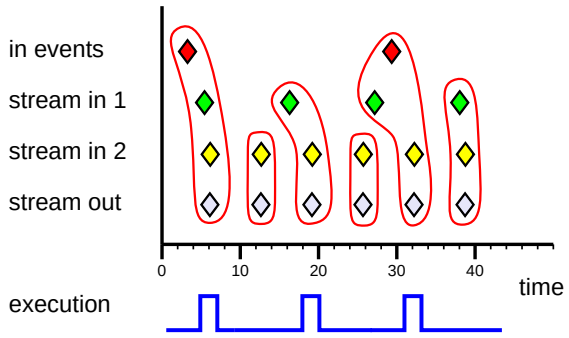


Figure 5.8: Alignment of incoming tokens in each execution. Note that time corresponds to token's time-information and does not relate to the module execution time (though they are equally spaced).

While incoming stream tokens always needs to be accurately aligned, this is not always true for incoming event tokens. Some applications requires a precise alignment of event tokens, while others admit a loose time alignment.

An obvious approach is to use the Payloads pattern, adding a precise time information —*time-stamp*— to each token. In real-time systems, this time-stamp relates to the time when the token is introduced into the system. In non real-time systems it relates to a virtual time. Transformations on a token should preserve the original time-stamp. However, this Payloads approach can be overkilling in some cases. For example, when stream tokens flow at a high rate, as it happens with audio samples.

## Problem

In order to synchronize incoming tokens we need its time information. How can we get the time information of incoming tokens when they arrive both as streams in different rates and as events?

## Forces

- Time-stamp is a big overhead when the data token is relatively small.
- Propagating timestamps from the consumed tokens to the produced tokens is a run-time overhead and makes concrete module implementation more

complex.

- Concrete module implementation should be simple.
- All stream sources must share the same hardware clock, so that their (token) rates can not vary among different streams.

## Solution

Calculate time information of incoming stream tokens instead of using time-stamps. If the application needs accurate timing for events use time-stamps —only for event tokens—, else do not.

Separate the stream and event flow in different kinds of ports: stream and event ports. Place the stream timing responsibility into the stream in-port class. Stream in-ports are initially configured with a "token-rate" and "first-token-time" values, and they also keep the sequence —with a counter, for example— of consumed tokens. When a module execution method asks the stream in-port for new tokens to consume, the in-port provides the time information along with the tokens itself.

Ports parameters ("token-rate" and "first-token-time") configuration is a key issue to solve. Two main approaches exist: ports handshaking and centralized management.

Ports handshaking consists in propagating parameters down-stream. In-ports receives parameters from their connected out-ports, and modules propagate them from in-ports to out-ports. In most cases, modules only need to copy them from the in-ports to out-ports. However, in some cases, the module processing may introduce delay and may change the token-rate; thus, this must be reflected in the out-port settings. In consequence, modules do not impose port parameters, they receives it and propagate them. Of course, source modules [1] are the exception to that rule. They must set the out-port parameters, because they are the *source* of the stream.

The second approach —centralized management— consist in incorporating an entity that orchestrates the configuration —and maybe the modules execution— of the whole network. This configuration manager is responsible for configuring

[1] Source modules are that ones that do have stream out-ports but do not have any stream in-ports

all the stream ports in the network.

Alignment of event tokens with stream tokens is done in slightly different ways depending on whether the application needs accurate event timing or not —that is, whether they incorporate time-stamps or not. Note that on each execution, the module may consume not only one stream token but a bunch of them. In some cases, like with audio samples, even a large number of them like, say, 1000. If incoming events are time-stamped, the module knows the time information for all the incoming tokens, thus the module can align each event token with the stream tokens precisely. If events are not time-stamped, the module should align all consumed events with the first consumed stream token of each in-port.

## Consequences

Making the stream tokens time implicit instead of explicit, the space overhead is avoided. For event tokens, time-stamps is allowed, though not imposed by the pattern solution. In case of having timestamps in event tokens the overhead is not as problematic as with stream tokens, since they flow non continuously, in much lower frequency than streams.

**Events Jitter:** Having a big number of stream tokens to be consumed on each execution and having non time-stamped event tokens at the same time is a common cause for jitter. That is, unsteadiness or irregular variation on the time the system respond to incoming events. The amount of jitter is bounded by the time interval between executions, which is proportional to the number of stream tokens consumed on each execution. Thus, making modules consume fewer stream tokens each time, reduces jitter. Of course, when events comes in with time-stamps, jitter can be eliminated completely. The consequences of having jitter varies enormously depending on the concrete application. In most cases jitter can be neglected, in other cases, however reducing it is paramount.

**Ports connectivity:** The solution forbids time-stamps in the stream tokens and this restricts how stream ports can be connected. Because time must be inferred from the incoming stream sequence, in-ports must receive well formed

sequences —without gaps, etc.— of an individual stream. Thereof, in general, N-to-1 connections of stream ports are to be forbidden by the system. However, an exception to this rule exists when the in-port is able to implicitly perform a combining operation prior to keeping track of the incoming sequence. For example, imagine an in-port of audio samples fed by multiple different streams. Parallel samples are added, forming an audio mix. Then the in-port counts the incoming tokens the same way as if the source were unique. Apart from addition, packing —that is, create a new composite token— is another common combining operation.

In general, multiple stream combinations is better handled explicitly in specific modules. It gives the system designer flexibility to choose his or her combining operation. Moreover, a system can have token types without any valid combining operation, thus making implicit combinations impossible.

We have seen that, in general, N-to-1 connections of stream ports are to be forbidden by the system. On the other hand, N-to-1 connections of event ports are perfectly fine, since there is no need to infer the token time information from the order of arrival. Time information is either read from the time-stamp or simply ignored.

Splitting one stream to multiple streams is a different story: 1-to-N connections of both stream and event ports are allowed. The consideration that has to be done here is how to duplicate outgoing tokens flowing to multiple destinations. Two strategies exist: one is making a copy of each outgoing token to every in-port, and the other is passing a managed reference to each in-port. In the later case, the in-ports will have to enforce read-only semantics.

## Related Patterns

Stream and Event Ports uses to go together with Driver Ports and, in most of the cases, stream ports are also the driver ones.

Systems that use Stream and Event Ports may also use Multi-rate Stream Ports for designing the stream ports —allowing stream ports to consume and produce at different cadences—, and may use Cascading Event Ports for its event ports —allowing event ports to propagate events immediately.

Stream ports designed with the Multi-rate Stream Ports pattern defines the

number of released tokens for each stream port on each execution. This numbers influence how the ports "token-rate" settings are propagated —in this case, being re-calculated— from in-ports to out-ports.

Stream and Event Ports can handle different token types by using the Typed Connection pattern.

### Examples

SuperCollider3 [McCartney, 2002] and CSL [Pope and Ramakrishnan, 2003] use the pattern but events can not arrive at any time: they have a dual rate system, control rate and audio rate, being control rate a divisor of the audio block rate.

Marsyas, [Tzanetakis and Cook, 2002] uses this pattern though its event ports does not follow the data-flow architecture because connections are not done explicitly. Interestingly, it implements the token packing technique (from multiple sources) as mentioned in the Ports Connectivity section.

CLAM and OSW [Chaudhary et al., 1999] clearly use this pattern, separating ports for streams and for events.

## 5.4 Typed Connections

### Context

Most simple audio applications have a single type of token: the sample or the sample buffer. But more elaborated processing applications must manage some other kinds of tokens such as spectra, spectral peaks, MFCC's, MIDI... You may not even want to limit the supported types. The same applies to events channels, we could limit them to floating point types but we may use structured events controls like the ones OSC [Wright, 1998] allows.

Heterogeneous data could be handled in a generic way (common abstract class, void pointers...) but this adds a dynamic type handling overhead to modules. Module programmers should have to deal with this complexity and this is not desirable. It is better to directly provide them the proper token type. Besides that, coupling the communication channel between modules with the actual token

type is good because this eases the channel internal buffers management.

But using typed connections may imply that the entity that handles the connections should deal with all the possible types. This could imply, at least, that the connection entity would have a maintainability problem. And it could even be unfeasible to manage when the set of those token types is not known at compilation time, but at run-time, for example, when we use plugins.

## Problem

Connectable entities communicate typed tokens but token types are not limited. Thus, how can a connection maker do typed connections without knowing the types?

## Forces

- Process needs to be very efficient and avoid dynamic type checking and handling.
- Connections are done in run-time by the user, so they can mismatch the token type.
- Dynamic type handling is a complex and error prone programming task, thus, placing it on the connection infrastructure is preferable than placing it on concrete modules implementation.
- Token buffering among modules can be implemented in a wiser way by knowing the concrete token type rather than just knowing an abstract base class.
- The set of token types evolves and grows.
- A connection maker coupled to the evolving set of types is a maintenance workhorse.
- A type could be added in run time.

## Solution

Split complementary ports interfaces into an abstract level, which is independent of the token-type, and a derived level that is coupled to the token-type. The class

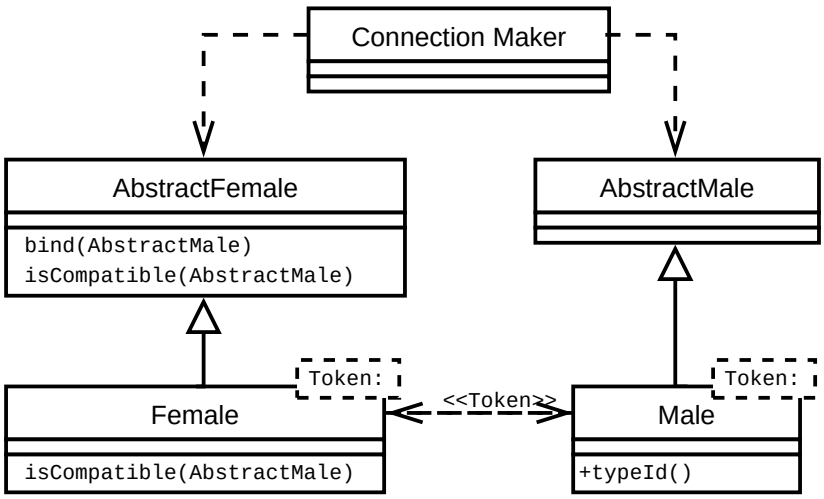


Figure 5.9: Class diagram of a cannonical solution of Typed Connections

diagram of this solution is shown in figure 5.9.

Let the connection maker set the connections throught the generic interface, while the connected entities use the token-type coupled interface to communicate each other. Access to typed tokens from the concrete module implementations using the typed interface.

Use run-time type checks when modules get connected (*binding time*) to get sure that connected ports types are compatible, and, once they are correctly connected (*processing time*), rely just on compile-time type checks.

To do that, the generic connection method on the abstract interface (`bind`) should delegate the dynamic type checking to abstract methods (`isCompatible`, `typeId`) implemented on token-type coupled classes.

## Consequences

By applying the solution, the connection maker is not coupled to token types. Just concrete modules are coupled to the token types they use.

Type safety is assured by checking the dynamic type on binding time and relying on compile time type checks during processing time. So this is both efficient and safe.

Because both sides on the connection know the token type, buffering structures

can deal with tokens in a wiser way when doing allocations, initializations, copies, etc.

Concrete modules just access to the static typed tokens. So, no dynamic type handling is needed.

Besides the static type, connection checking gives the ability to do extra checks on the connecting entities such as semantic type information. For example, implementations of the bind method could check that the size and scale of audio spectra match.

## Related Patterns

This pattern enriches Multi-rate Stream Ports and Event Ports, and can be also useful for the binding of the visualization and the Port Monitor.

The proposed implementation of Typed Connections uses the Template Method [Gamma et al., 1995] to call the concrete binding method from the generic interface.

## Examples

OSW [Chaudhary et al., 1999] uses Typed Connections to allow incorporating custom data types.

The CLAM framework uses this pattern notably on several pluggable pairs such as in and out ports and in and out controls, which are, in addition, examples of the Multi-rate Stream Ports and Event Ports patterns.

But the Typed connection pattern in CLAM is not limited to port like pairs. For example, CLAM implements sound descriptors extractor modules which have ports directly connected to a descriptor container which stores them. The extractor and the container are type coupled but the connections are done as described in a configuration file, so handling generic typed connections is needed.

The Music Annotator [Amatriain et al., 2005] is a recent application which provides another example of non-port-like use of Typed Connections. Most of its views are type coupled and they are mostly plugins. Data to be visualized is read from an storage like the one before. A design based on the Typed Connection pattern is used in order to know which data on the schema is available to be

viewed with each vista so that users can attach any view to any type compatible attribute on the storage.

# Chapter 6

# Flow Implementation Patterns

## 6.1 Cascading Event Ports

### Context

It is common that the arrival of an event token into a module implies propagating events to other modules. For instance, a sequencer controlled instrument has to send control events to several sound synthesis modules in response to an incoming note-on event. If the propagation chain has several levels and the event propagation through a module is not performed until it is executed, some of the receiver modules may execute several times before receiving the event tokens. So we need the event tokens to be propagated before any other module is executed.

### Problem

How can we send event tokens and run associated actions on the receiver, including propagation, so that they get to the destination before other modules are executed?

### Forces

- Module execution should be able to send events.
- Event token reception may imply change the module state.

- Event token reception may imply the sending of new event tokens to other modules.
- Event token propagation should not be costly in respect to module execution in order not to break execution cadence and real-time restrictions.
- Coarse event tokens are hard to propagate by copy.
- Feedback loops on event ports should be allowed.
- 1 to N event ports connections should be allowed.
- N to 1 event ports connections should be allowed.

## Solution

Provide the concrete module implementers a way to bind a event in-port with a callback method to be called on event reception. This callback might change the module state, or propagate other events in cascade through the module event out-ports. Make the sending of an event through an event out-port imply the immediate execution of callback methods associated with every connected event in-port.

Propagate coarse event tokens using references instead copies and make them read-only for the receiving modules. Limit the life of event tokens sent by reference to the cascade propagation and, forbid the receiving modules to keep references further than the callback execution.

## Consequences

Event in-port callback implementers should be careful not to do too much things on them. Events may be sent in bursts; thus, expensive callbacks could break the real-time restrictions.

Propagation of coarse events is something that could add penalty to in-port callbacks, but, by using references, this is avoided. Sending references could be dangerous when considering 1 to N connections, as one of the receiving modules may modify the event token. This is solved by making them read-only for the receiving modules.

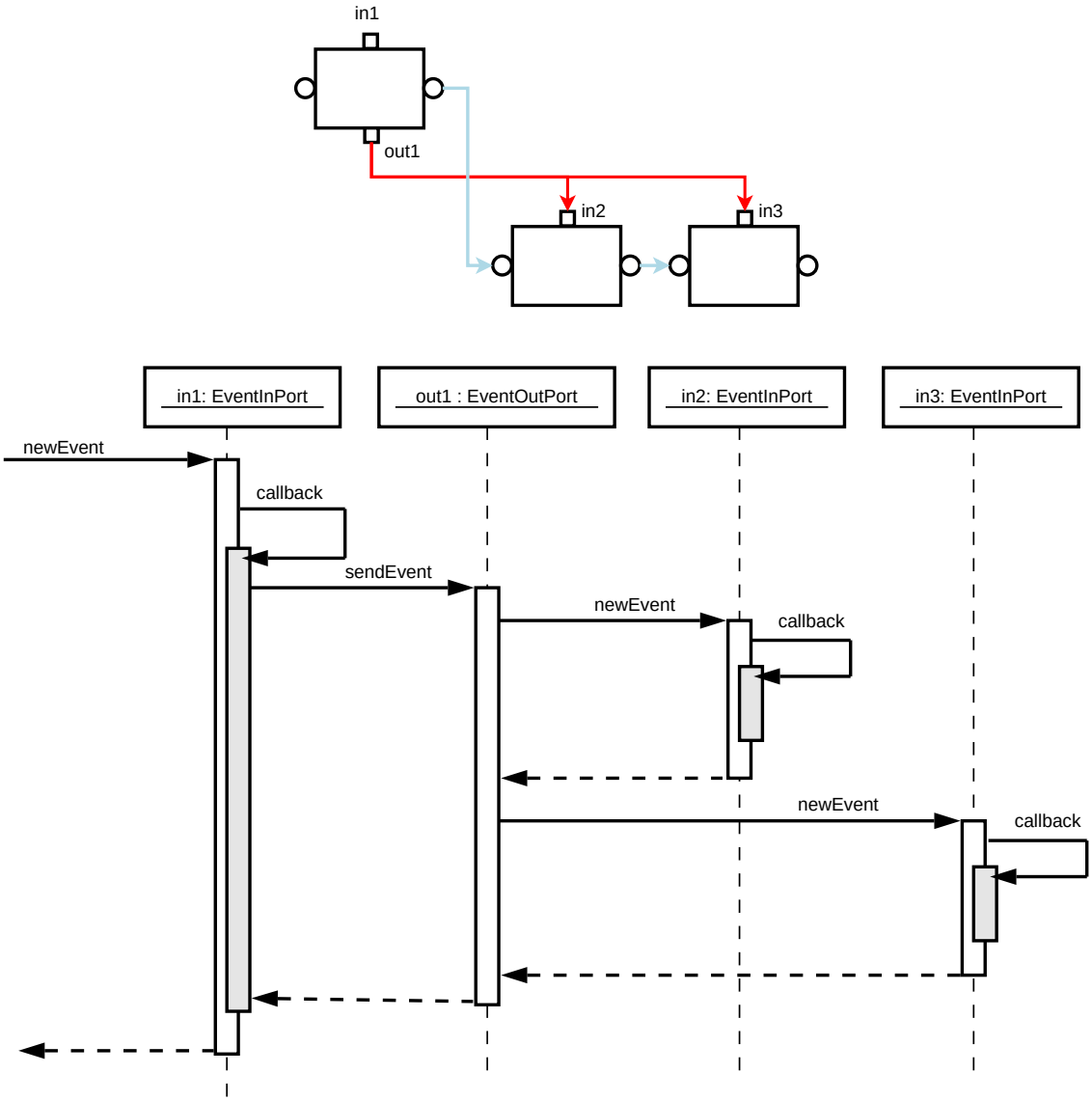


Figure 6.1: A scenario with cascading event ports and its sequence diagram.

Another danger associated to sending references is that modules might keep references to such tokens. Because of this, keeping references is forbidden, but we could loose this restriction by using reference counting on event tokens. The use of garbage collectors is not a good solution due to real-time restrictions.

The solution allows setting up loops on the event ports connection graph. Those loops might be harmless but they might be pernicious because the cascade callback calling enters in a non-ending loop. Harmful loops happen whenever the call sequence reaches a port that was already involved on the cascade.

Static analysis of the network topology to warn the user about harmful loops is useless: Not every event reception implies propagation on the event out-ports of the module; it depends on the callbacks methods. Because the sending of events is a synchronous call, one simple solution is to block sending tokens through a port which is already sending one. This is implemented just by adding a "sending-on-progress" flag in each port.

## Related Patterns

Event tokens could be restricted to a given set of types, but we could also use the Typed Connections pattern to a more flexible solution.

Cascading Event Ports provides a flexible way for communicating the two partitions of the Out-of-band and in-band-partitions pattern [Manolescu, 1997]. The user interface partition communicates with the processing partition via connected event ports. Since both partitions are in different threads, a safe thread boundary must be established. Using, for example, the Message Queuing pattern [Douglass, 2003].

This pattern could be seen as a concrete adaptation of Observer [Gamma et al., 1995] to a data-flow domain where modules can act both as *observers* and *subjects*, in a way that they can be chained.

## Examples

CLAM implements controls as cascading event ports. Multiple control inputs and outputs are supported. By default, events are copied as part of the module state but you can add a callback method to process each control in a special way. In its current version (0.91) event tokens are limited to floating point numbers.

PD [Puckette, 1997], MAX/MSP [Puckette, 1991] they all use cascading event ports for their non-audio-related modules "hot inlets".

## 6.2 Multi-rate Stream Ports

### Context

Many applications in the audio and music domain need to process chunks of consecutive audio samples in a single step. A common example is an FFT transformation which consumes $N$ audio sample tokens and produces a single spectrum token. Thereof, the flowing rate of spectrum tokens is $N$ times lower than the samples flowing rate. The FFT transformation may also need to process overlapping sample windows. That is, the FFT module reads $N$ samples through an in-port and, after the execution, the window slides a step of $M$ samples, where $M$ and $N$ are different.

In such applications the stream may undergo a type change (i.e. from samples to spectrum) after a filter module.[1] Moreover, filter modules may have different number of in-ports and out-ports.

This example shows two different —though related— problems:

- Streams can flow at different rates. Like, for example, sample and spectrum streams do.
- Modules may need to process different numbers of tokens on each execution, regardless of the rate of its incoming streams. For example, an FFT module may require 512 samples while another FFT module may require 1024.

That means that the number of tokens a module consume and produce should be flexible, allowing modules to operate with different consuming and producing rates.

How to approach this problem is not obvious. Some real-life systems[2] perform multi-rate processing inside their modules while restricting inter-module communication to a single rate. Since the number of tokens that a module's algorithm

[1] Filter modules are those that have both stream in-ports and stream out-ports

[2] One example is the JACK [Davis et al., 2004] audio server, with the Jamin mastering tool.

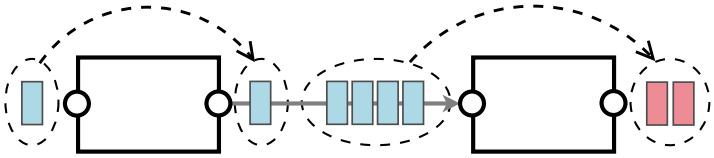

Figure 6.2: Two modules consuming and producing different numbers of tokens

needs is not the same as the number of tokens consumed on each execution, input and output buffering is needed inside the module. A weird effect of this approach is that the module execution not always implies its algorithm execution; when not enough tokens are ready for the algorithm, the module execution just adds incoming tokens to the internal buffers. Of course, this solution yields complex code in every concrete module.

## Problem

How to allow modules accessing a different number of consecutive tokens on every stream port?

## Forces

- The number of accessed tokens and the number of released tokens is particular for each port.
- For a given port, the number of accessed tokens and the number of released tokens are unrelated.
- Modules have to process a sorted sequence of stream data tokens (usually a time sorted sequence)
- All the stream tokens have the same priority.
- An out-port could be connected to multiple in-ports so that the tokens produced might be consumed by different modules.
- A module may need access to a number of consecutive tokens for each incoming stream to be able to execute.
- A module execution may produce a number of consecutive tokens for each

output stream.

- Arbitrary consuming and producing rates in a network renders static scheduling of executions impossible.
- Feedback loops should be allowed
- Copy of coarse tokens may be an important overhead to avoid.
- Concrete module implementation should be simple.

## Solution

Design stream ports so that they support consuming and producing at different rates. This way they can adapt to the rate the module's algorithm needs, while keeping the buffering details outside the module. The in-port and out-ports should give access to $N$ tokens from a queue[3] and should release $M$ tokens on every module execution. Let the module developer define $N$ and $M$ for each port.

Give the ports and interface for accessing tokens —but only a window of $N$ tokens at the head of the queue— and for releasing them. Releasing tokens results in that $M$ tokens will be dequeued and put away from the module reach. "Access" and "release" are operations implemented as port methods and they are to be called consecutively by the module execution method. Seen as a single operation, "access-and-release" is equivalent to "consume", when called in an in port, and "produce", when called in an out-port.

Make the ports own the tokens flowing between two modules, and make them responsible for all necessary buffering between out-ports and in-ports.

Buffers can be either associated to in-ports or to out-ports. In 1-to-N connections —that is, a single out-port connected to N in-ports— this decision makes the difference between heaving N different buffers (at the in-ports) or having a single buffer (at the out-port).

In the following paragraphs we discuss the implications of having buffers at the in-ports and at the our-ports.

[3]By queue we mean the abstract data type with its generic operations without making assumptions on it implementation.

**Buffers at the in-ports:** This is the simplest solution to implement. Give to each in-port an associated buffer (figure 6.3). Tokens being produced from an out-port are then passed to the connected in-port buffers. Tokens can be either passed by reference or by copy. Passing tokens by copy is easy to implement since each in-port is the owner of its tokens. Passing references, on the other hand, is more efficient because copies are avoided. This efficiency gain can be very important when tokens are to be passed to many in-ports or when tokens are coarse objects.

Of course, the efficiency gain associated with passing references instead of copying comes with a price: it is more difficult to implement. Given that multiple modules receive a reference to the same token, aliasing problems have to be avoided. In-ports have to be designed in a way that guarantees read-only semantics —or copy-on-write semantics— on the incoming tokens. The other aspect to be addressed here is the tokens life-cycle. Since token memory can not be freed —or recycled— while references to it exist, we need a reference-counting mechanism.[4]

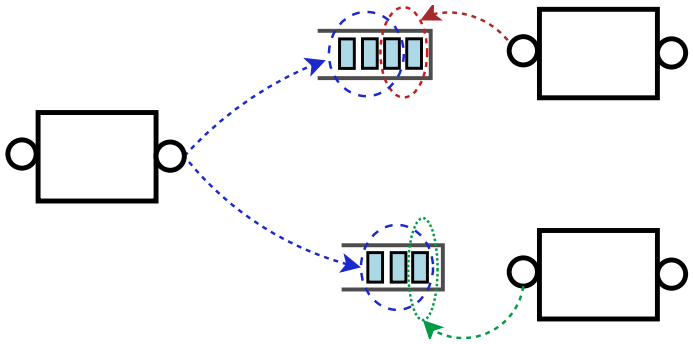

Figure 6.3: Each in-port having its own buffer.

However, passing references to the in-ports is not always feasible. Some applications may require its modules to operate on tokens placed on contiguous memory —examples of this are very common in the audio domain— as a consequence, such modules need the actual tokens data (not references) placed together in circular buffers at the in-ports.

This shortcoming can be overcome placing the buffers at the out-ports, which allows having both reference passing and contiguity. Again, we will see that effi-

[4]Like C++ smart pointers

ciency comes with a price.

**Buffers at the out-ports:** Having a single buffer for a 1-to-N connection — thus, associated to the out-port— allows benefiting from passing tokens by reference while achieving data contiguity (figure 6.4).

For that, the buffer must be implemented with a circular buffer. Not only the out-port is accessing the buffer but also the $N$ connected in-ports.

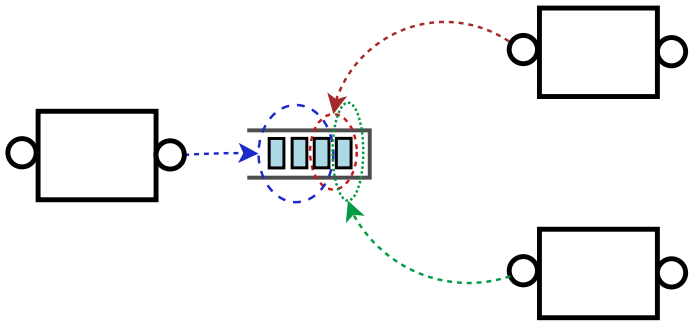

Figure 6.4: A buffer at the out-port is shared with two in-ports.

Allowing a buffer to be written by a producer and read by $N$ different consumers, while allowing each one to produce or consume at a different cadence needs to done carefully. The following two basic restrictions must be enforced by design:

- The out-port can not over-write tokens that still have to be red/consumed by some in-port.
- The in-ports can not read/consume tokens that still have to be written/produced by the out-port.

Though complex to implement, this approach avoids the need for unnecessary copies of tokens and allows contiguous memory access to all involved modules.

**Summing up:** We have seen different strategies for implementing ports buffering that present a trade-off between simplicity and efficiency. Placing buffers at the in-ports and passing tokens by copy is the most simple approach. If copies

are to be avoided, token references can be passed, but they have to be managed. Sometimes this is not enough. Apart from avoiding copies we need memory contiguity. Then, a circular buffer must be placed at the out-port and some restrictions must be enforced.

## Consequences

Since all buffers adaptation is done at the ports level by the general infrastructure, concrete module implementors do not have to deal with buffers adaptation. That results in simpler, less error-prone code in every module.

Modules with different production and consumption rates can be connected together, as drawn in figure 6.2. As a result, this increases the number of the possible networks that can be built out of a set of modules.

The number of tokens stored in each port connection depends on two factors: In one hand, the requirements given by each port regarding the number of accessed and released tokens and, on the other hand, the scheduling policy in use.

This solution implies that, in general, the network will need a dynamic scheduler of module executions. To facilitate the task of such scheduler, modules may provide an interface to inform whether they are ready to execute or not. Such module method could be easily implemented in the module base class by delegating the question into every driver port, and returning the *and* combination if its responses.

## Related Patterns

Multi-rate Stream Ports is applied in the context of systems that uses Stream and Event Ports, it addresses how stream ports can be designed so that they offer a flexible behavior.

Multiple Window Circular Buffer pattern addresses the low level implementation of the more complex variant of Multi-rate Stream Ports, that is *buffers at the out-ports*.

The design and implementation of *buffer at the out-ports* is a clear example of the Multiple Window Circular Buffer pattern.

### Examples

In most of the systems reviewed, ports of the same type have all the same window size, and thus do not need to use this pattern. This is, for example, the case of the CSL [Pope and Ramakrishnan, 2003] and OSW [Chaudhary et al., 1999] frameworks and the visual programming tool MAX [Puckette, 2002].

On the other hand the Marsyas [Tzanetakis and Cook, 2002], SuperCollider3 [McCartney, 2002] and CLAM frameworks allow different window sizes, but they follow different approaches.

SuperCollider3 [McCartney, 2002] features variable block calculation and single sample calculation. For example, modules corresponding to different voices of a synthesizer may consume and produce different block sizes. The SuperCollider3 framework permits embedded graphs that have a block size witch is an integer multiple or division of the parent. This allows parts of a graph which may require large or single sample buffer sizes to be segregated allowing the rest of the graph to be performed more efficiently.

Marsyas allows buffer size adaptation using special modules. CLAM — probably for its bias towards the spectral domain— is the most flexible, allowing any port connection regardless of its window size. CLAM sets up a buffer at each out-port.

## 6.3 Multiple Window Circular Buffer

### Context

As a result of incorporating the Multi-rate Stream Ports pattern, the ports-connection queues needs a complex behaviour, in order to access and release different number of tokens.

If it was not enough, such systems often have real-time requirements and some optimization factors must be taken into account: avoiding unnecessary copies, totally avoiding allocations and being able to work with contiguous tokens. Take for example (again) modules that performs the FFT transformation —delegating to some external library— upon a chunk of audio sample tokens; input samples must be provided to the library as an array. In the audio domain, not only FFTs

need to operate with arrays, temporal domain processing is typically done that way too.

A simple implementation of Multi-rate Stream Ports consists in having a buffer associated to each in-port. But, unfortunately, this means copying tokens. The copy-saving implementation of Multi-rate Stream Ports requires a single buffer to be shared by an out-port and many in-ports. A design for this is not obvious at all. So, this is what this pattern addresses.

Finally, note that though a normal circular buffer is not suited for accommodating the given requirements, what we are seeking may be seen as a "generalized" circular buffer. Moreover, this can be useful in scenarios other than data-flow architectures.

## Problem

What design supports a single source of tokens with one writer and multiple readers, giving each one access to a subsequence of tokens?

## Forces

- Each port must give access to a subsequence of $N$ tokens (the window).
- The subsequence of tokens should be in contiguous memory, since many algorithms or domain tool-kits and libraries works on contiguous memory.
- Windows sizes and steps should all be independent.
- Reading windows can only map tokens that have been already produced through the writing window.
- Allocation during processing time should be avoided, since (normal) dynamic memory allocation breaks the real-time requirements.
- All client executes in the same thread.

## Solution

Have a contiguous circular buffer with windows that maps (contiguous) portions of the buffer. There will be as many reading windows as needed but only one

writing window. Associate the reading windows with the writing window because, as we will see they will need to calculate their relative distances. Also, provide them means for sliding along the circular buffer.

The modules (the buffer clients) executions must be done in the same thread. Its scheduling can be done either statically —fixed from the beginning— if all ports consuming and producing rates are known; or dynamically, which is much simpler to implement.

Windows clients need to follow the following protocol in order to avoid data inconsistencies:

- The access to windows mapped elements and the subsequent slide of the window must be done atomically in respect of other window operations. So, these operations might be regarded as a single read-and-slice (or write-and-slice) operation. Only when a window has finished the sliding, other clients can access their own window.
- A reading window can only start a read-and-slice (also known as *consume*) operation when it is not overlapping the writing window (overlapping other reading windows is perfectly fine). This reader-overlapping-writer problem indicates that the client is reading too fast. This problem should be detected and, as a response, the reading module should not be executed till more data has been written into the buffer.
- The writing window can only start a write-and-slice (or *produce*) operation when it is not overlapping the furthest reading region. Such overlapping is possible since regions are circulating over the underlying circular buffer. This writer-overlapping-reading problem indicate either that a client is reading too slow or that the buffer size is not large enough. When this is detected, the writing module should not be executed and should wait for the readers to advance.

The solution design uses the Layered pattern [Douglass, 2003] for arranging different semantic concepts at different layers. Concretely, we distinguish three levels of abstraction (figure 6.5). Starting from the layer that gives direct service to the clients:

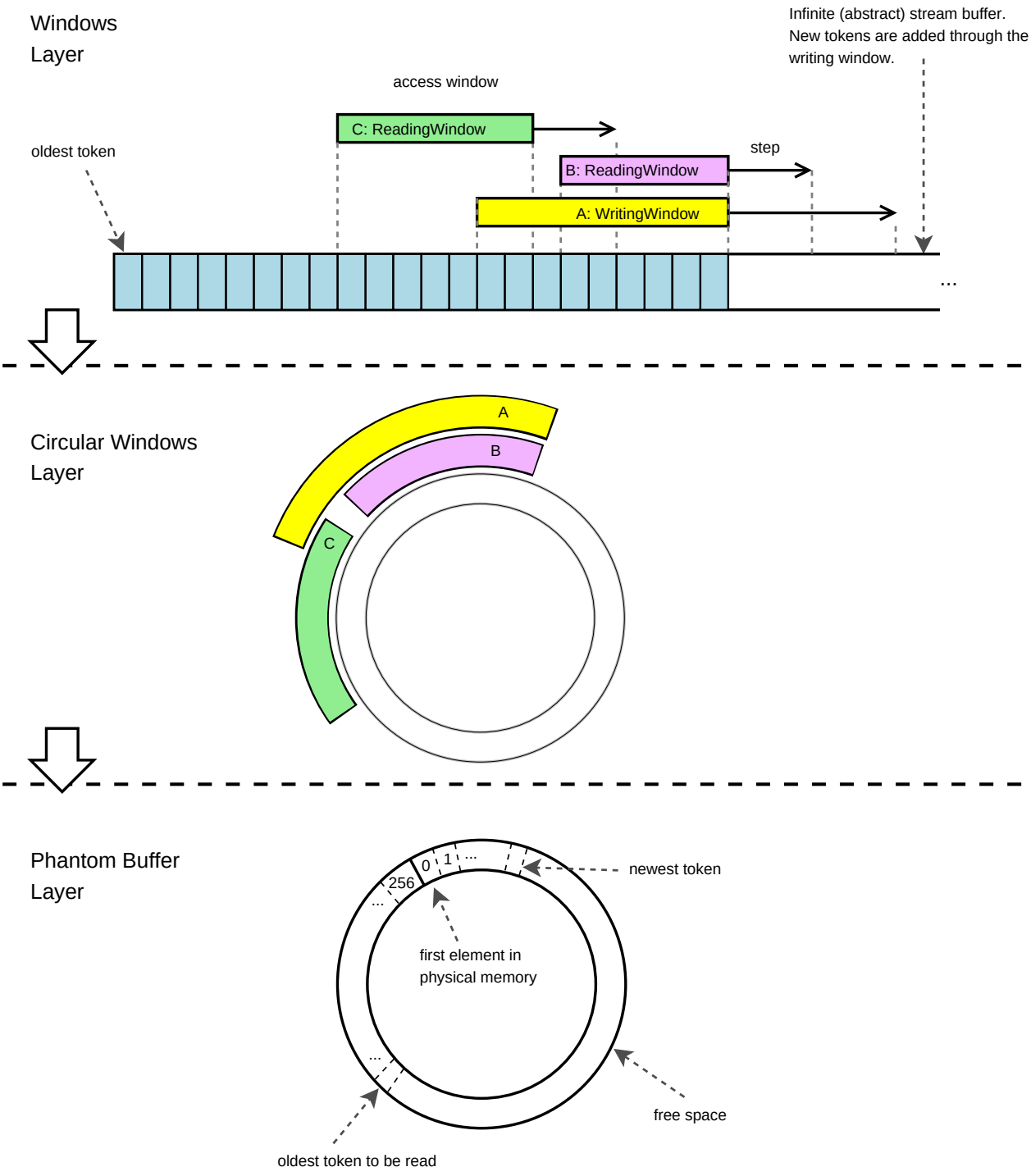


Figure 6.5: Layered design of port windows.

**Windows Layer** This is the upper or more abstract layer, which gives the clients a view of the windows advancing on an infinite buffer. It offers, at least, the following interface :

- Accessing the $N$ contiguous elements mapped by the window.
- Advancing a window its slicing step (not necessarily $N$) as if the windows where on an infinite buffer.
- Checking if a window is ready to be accessed-and-slided.

The state of this layer keeps the relative distances between each reading and the writing window. This layer is in charge of detecting when a reading window overlaps with the writing window, and delegates other checks to its underlying layer.

**Circular Windows Layer** The state of this layer keeps the physical pointers (to a circular buffer) for each window and also provides physical pointers to the upper layer. This layer is in charge of detecting —and preventing— circular overlapping with a reading window. That is, the case when the writer is about to write on a still not read element.

**Phantom Buffer Layer** This is the lower layer, which knows nothing about writing and reading and writing windows and is solely dedicated to provide chunks of contiguous elements for each window. Thereof, this layer's goal is to to provide contiguous elements subsequences of size $N$ or smaller. Where $N$ is the size of the biggest window.

The main problem this layer has to solve is the discontinuity problem associated to circular buffers —the next element in a logical sequence of the last physical element is the first physical element. To idea behind the solution is to replicate the first $N$ elements at the end of the buffer. This can be implemented using a data structure that we call "Phantom Buffer" and is presented in this catalog as the Phantom Buffer pattern.

## Consequences

The non-overlapping restrictions might suggest that there always exists a distance between writing and reading windows and, thus, causing the introduction of certain latency. But this is not the case, because the non-overlapping restrictions only apply, at the time of an access-and-slide operation. After a window have been slided, it is perfectly legal to be in an overlapping state. This allows the reading windows to consume the same tokens that the writing window has just produced.

The reader-too-slow and writer-too-slow problems can be handled in the context of a dynamic scheduler. Before doing any access-and-slide operation, a *canProduce()* or *canConsume()* check is done, so that the operation can be safely aborted.

The consequence of the layered approach is a flexible design that allows changing the underlying data structure easily, without affecting the windows layer and its client. It also eases the implementation task since the overall complexity is split in well balanced layers which can be implemented and tested separately. On the other hand, those many levels of indirections might carry a performance penalty. However, it should be note that the implementation does not require polymorphism at all. Thus, when implemented in C++, with a modern compiler, most of the indirections should be converted to in-line code by the compiler, reducing the function-calls overhead.

In general, setting the window parameters can be done at configuration time; that is, before the processing or module executions starts.

## Related Patterns

This pattern solves the *buffer at the out-port* approach of the Multi-rate Stream Ports pattern, which was the optimal one. Multiple Window Circular Buffer uses the Layered pattern [Douglass, 2003].

### Examples

This pattern is maybe a *proto-pattern* [www-PatternsEssential, ] as the authors only know their own implementation in the CLAM framework [www-CLAM, ]. Nevertheless, CLAM is a general purpose framework and several applications with different requirements have proven the value of the pattern.

## 6.4 Phantom Buffer

### Context

The goal of Multiple Window Circular Buffer is to design a generalized circular buffer where, instead of having a writing and a reading pointer, we deal with a writing window and multiple reading windows. The difference between a window and a plain pointer is that a window gives access to multiple elements which, moreover, need to be arranged in contiguous memory. Multiple Window Circular Buffer relies, for its elements storage, upon some data structure with the following functionalities: One, to be able to store a sequence of any size ranging from 0 to $MAX$ elements; and two, to be able to store each subsequence up to $N$ elements in contiguous memory.

A normal circular buffer efficiently implements a queue within a fixed block of memory. But in a normal circular buffer the contiguity guarantee does not hold: given an arbitrary element in the buffer, chances are that its next element in the (logical) sequence will be physically stored on the other extreme of the buffer.

Note that, here, window management is not relevant at all because it is a responsibility of upper layers. Thus, the only concern of this pattern is how the low-level memory storage is organized.

### Problem

Which data structure holds the benefits of circular buffer while guarantees that each subsequence of $N$ elements sits in contiguous memory?

## Forces

- Element copies is an overhead to avoid.
- Buffer reallocations are to be avoided.
- It should be possible for clients to read and write a subsequence of elements using a pointer to the first element. The rational is that modules might want to use existing libraries that, typically, use pointers as its input and output data interface.
- Multi-threading is not a requirement.

## Solution

The *buffer with phantom zone —phantom buffer* for short— is a simple data structure built on top of an array of $MAX + N$ elements. Its main particularity is that the last $N$ elements are a replication of the first $N$ elements. This guarantees that starting at any physical position from 0 to $MAX - 1$, exists a contiguous subsequence of size up to $N$ elements. In effect, this is clear considering the worst case scenario: take the element at position $MAX - 1$; let it be the first one in a subsequence; since it is a circular buffer of $MAX$ elements, the next element is in the position 0, but positions from 0 to $N - 1$ are also replicated at the end (starting at position $MAX$); thus the contiguity condition is guaranteed.

Interface of a PhantomBuffer class should include two methods: one for accessing a given window of elements, and the other, for synchronizing a given window of elements. For example, in C++:

A client that wants to read a window, should call the *access* method, and read elements starting from the returned pointer. If the client wants to write, the sequence is a little different; first if should call *access*, write elements and, finally, call *synchronize*. This method synchronizes, when needed, a portion of the phantom zone with its counterpart in the buffer beginning. To be accurate, a copy of elements will only be necessary when the window passed as argument to *synchronize* have intersection with the phantom or the initial zone.

Summing up, a phantom buffer offers a contiguous array where the last $N$ elements are a replication of the first $N$. Each write on the first or last $N$ element

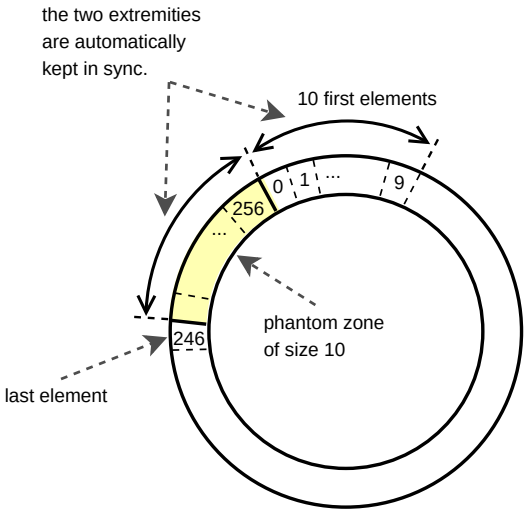


Figure 6.6: A phantom buffer of (logical) size 246, with 256 allocated elements and phantom zone of size 10.

```
template<class T> class PhantomBuffer
{
public:
    T* access(unsigned pos, unsigned size);
    void synchronize(unsigned pos, unsigned size);
    ...
};
```

Figure 6.7: PhantomBuffer class definition in C++

is automatically synchronized in its dual zone. Thus, the client of a phantom buffer will always have access to chunks of up to $N$ contiguous elements,

## Consequences

As a result of this design, clients must be well behaved. This includes two aspects: The first is that clients that receive a pointer for a given window should not access elements beyond that window; the second is that, after a write, a client must call the *synchronize* method. Failing to do any of this might result in a serious run-time failure.

Certainly, this results in a lack of robustness. But this is the price to pay for the requirement of providing plain pointers to the window, and avoiding unnecessary copies and reallocations. However, the phantom buffer interface should not be directly exposed to the concrete module implementation. The port classes presents a higher level interface to the module while hiding details such as window parameters and synchronizations.

The circular buffer allocation should be done at configuration time and the phantom size depends (must be greater) on the maximum window size.

## Related Patterns

This pattern can be regarded as a part of a more extensive pattern that provides a generalized circular buffer with many readers. In this context, the windows management issues are addressed in the more general Multiple Window Circular Buffer pattern. Phantom Buffer provides a refinement of the lower-level layer drawn in the general pattern. Thereof, these two patterns collaborate together to give a complete solution for a generalized circular buffer.

## Examples

This pattern can be found implemented in the CLAM framework. Specifically in the *PhantomBuffer* class.

# Chapter 7

# Network Usability Patterns

## 7.1 Recursive networks

### Context

The potential of the interconnected modules model is virtually infinite. You can connect more and more modules to get larger and more complex systems. But module networks are normally defined by humans and humans have limitations on the complexity they can deal. So, big networks with a lot of connections are difficult to handle by the user, and this fact limits, actually, the potential of the model.

One of the reasons that makes audio systems to become larger is duplication. Duplication happens, for example, whenever two audio channels have to be processed the same way, This duplication is very hard to maintain, because it implies having to apply repeated changes, and this is a very tedious and error prone process.

Duplication may happen also outside the system boundaries. The same set of interconnected modules may be present on several systems. Fixes on one of those systems do not apply to the other one so we have to apply it repeatedly and this is even more tedious and error prone.

## Problem

How to reduce the complexity the user has to handle in order to define large and complex networks of interconnected modules?

## Forces

- User defining big networks maybe too complex
- Human complexity handling is limited on the number of elements and relations
- Divide and conquer techniques help humans to handle complexity by focussing on smaller problems instead of the whole problem
- Duplications of sets of modules and connections is hard to maintain
- Reuse of previously designed networks helps on productivity
- Encapsulation hides details that can be useful on tracing the behavior of the system

## Solution

By applying the 'divide & conquer' idea, we allow the user to define an abstraction of a set of interconnected modules as a single module that can be used in any other network. Some of the stream and event ports of the internal modules may be externalized as the stream and event ports of the container module.

Several internal *stream in-ports* may be merged as a single external one. So that, incoming stream tokens are read by all the internal stream in-ports. The same happens with in and out *event ports*. But it doesn't happen with the *stream out-ports*. The same reasons that forbid to *stream out-ports* feed a single *stream in-port* apply here.

If the system forbids merging ports on externalization, the externalized ports may be the internal ones. But when port merging is permitted, the user needs an abstraction on connecting a single port. This abstraction is given by a Proxy [Gamma et al., 1995] port.

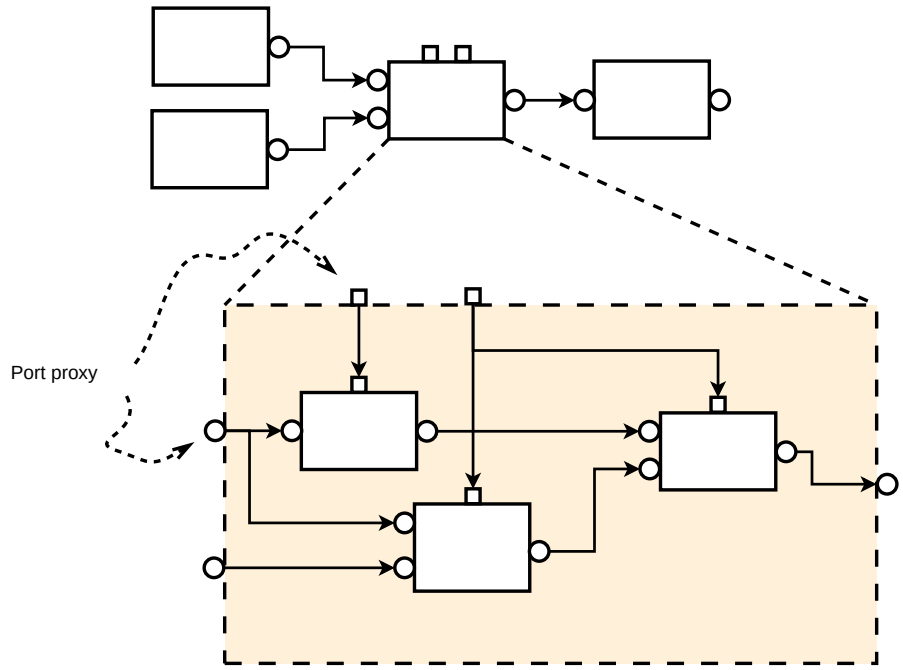


Figure 7.1: A network acting as a module.

Depending on the implementation, the *proxy port* may act as a proxy on connection time or additionally on process time.

A *connect time port proxy* is a proxy port that delegates binding calls to the proxied ports. This way, during processing time the communication is done directly at non-proxy port level.

A *processing time port proxy* is a proxy port that acts as a the complementary in/out port for the internal ports. For example, an in proxy port is seen as out port for the internal ports connected to it. This is similar to have an identity module that just pipes tokens. The *processing time port proxy* adds overhead but it is useful when we need a clear boundary between inwards and outwards.

Also several approaches can be used for the flow control to handle *recursive networks*. One approach is to make the inner modules visible to the outer flow control, so that once all the modules are accessible by the flow control, all happens the same way it would happen if the recursive network was not there.

A second approach is to hide the inner modules to the flow control. This can be done by providing an inner flow control to the subnetwork. The subnetwork

execution as module triggers the inner flow control. This approach is useful when a special flow control is needed. Also when we want to keep control on the proxied modules while processing.

### Related Patterns

This pattern is a direct use of the Composite and Proxy [Gamma et al., 1995].

The flow control approach that hides inner modules to the outer flow control by providing a inner one, is a Hierarchical Control [Douglass, 2003].

Adjacent performance critical modules can be replaced by an optimized version as an static composition, trading flexibility by performance, using Adaptive Pipeline [Posnak and M., 1996].

### Examples

Most audio domain frameworks implement Recursive Networks. For example MAX/MSP [Puckette, 1991], CSL [Pope and Ramakrishnan, 2003], OSW [Chaudhary et al., 1999], Aura [Dannenberg and Brandt, 1996b], Marsyas [Tzanetakis and Cook, 2002] and CLAM.

CLAM provides examples of most of the variants explained before. CLAM *Processing Composites* are compiled networks that provide their own flow control and they are seen for the flow control as a single module. *Processing Composites*'s ports are connection proxies so, external modules are actually connected on processing time to the inner ports. On the other side, CLAM also provides dynamic assembled networks, In this case, dummy modules which pipes directly event and stream tokens, are used as process time port proxies.

## 7.2 Port Monitors

### Context

Some audio applications need to show a graphical representation of tokens that are being produced by some module out-port. While the visualization needs just to be fluid, the processing has real-time requirements. This normally requires splitting

visualization and processing into different threads, where the processing thread has real-time requirements and is a high priority scheduled thread. But because the non real-time monitoring should access to the processing thread tokens some concurrency handling is needed and this often implies locking.

## Problem

We need to graphically monitor tokens being processed. How to do it without locking the real-time processing while keeping the visualization fluid?

## Forces

- The processing has real-time requirements (ie. audio)
- Visualizations must be fluid; that means that it should visualize on time and often but it may skip tokens
- Just the processing is not filling all the computation time

## Solution

The solution is to encapsulate concurrency in a special kind of process module, the *Port monitor*, that is connected to the monitored out-port. *Port monitors* offers the visualization thread an special interface to access tokens in a thread safe way.

In order to manage the concurrency avoiding the processing to stall, the *Port monitor* uses two alternated buffers to copy tokens. In a given time, one of them is the writing one and the other is the reading one. The *Port monitor* state includes a flag that indicates which buffer is the writing one. The *Port monitor* execution starts by switching the writing buffer and copying the current token there. Any access from the visualization thread locks the buffer switching flag. Port execution uses a *try lock* to switch the buffer, so, the process thread is not being blocked, it is just writing on the same buffer while the visualization holds the lock.

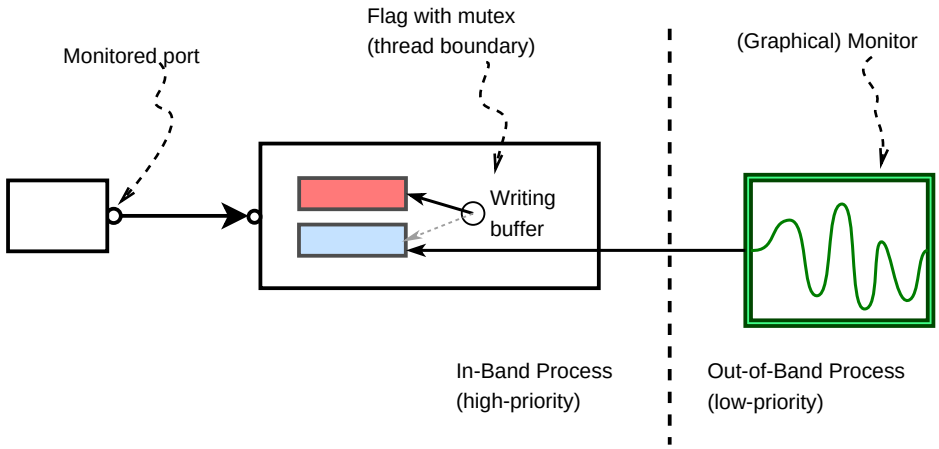


Figure 7.2: A port monitor with its switching two buffers

## Consequences

Applying this pattern we minimize the blocking effect of concurrent access on two fronts. On one side, the processing thread never blocks. On the other, the blocking time of the visualization thread is very reduced, due that it only lasts a single flag switching.

Any way, the visualization thread may suffer starvation risk. Not because the visualization thread will be blocked but because it may be reading always the same buffer. That may happen when every time the processing thread tries to switch the buffers, the visualization is blocking. This effect is not that critical and can be avoided by minimizing the time the visualization thread is accessing tokens, for example, by copying them and release.

When this effect is too notorious, a solution might be to use three buffers. This way, even when the visualization is blocking the buffer, the processing thread may alternate on the other buffers. The constraints that should apply are:

- Two buffer marks are always kept the *reading* buffer and the *last written* buffer.
- Three mutually exclusive operations may happen:
  - The processing thread should choose to write on any buffer that has none of those marks.

- When the processing thread ends writing it updates the *last written* buffer.
- When the visualization thread access, it moves the *reading* mark to the current *last written* mark.

This solution is not that good for real-time as the one based on just two buffers. The former may block the processing thread, while the latter never blocks it.

Another issue with this pattern is how to monitor not a single token but a window of tokens. For example, if we want to visualize a sonogram (a color map representing spectra along the time) where each token is a single spectrum. The simplest solution, without any modification on the previous monitor is to do the buffering on the visualizer and pick samples at monitoring time. This implies that some tokens will be skipped on the visualization, but, for some uses, this is a valid solution.

The number of skipped tokens is not fixed, thus, this solution may show time stretching like artifacts that may not be acceptable for some application. Double/triple buffering on the port monitor the full window of tokens solves that. It is reliable but it affects the performance of the processing thread.

## Related Patterns

Port Monitor is a refinement of Out-of-band and In-band Partition pattern [Manolescu, 1997]. Data flowing out of a port belongs to the In-band partition, while the monitoring entity (for example a graphical widget) is located in the out-of-band partition.

It is very similar to the Ordered Locking real-time pattern [Douglass, 2003]. Ordered Locking ensures that deadlock can not occur, preventing circular waiting. The main difference is in their purpose: *Port Monitor* allows communicate two band partitions with different requirements.

## Examples

The CLAM Network Editor [Amatriain and Arumí, 2005] is a visual builder for CLAM that uses Port Monitor to visualize stream data in patch boxes. The same

approach is used for the companion utility, the Prototyper, which dynamically binds defined networks with a QT designer interface.

The Music Annotator also uses the concurrency handling aspect of Port Monitor although it is not based on modules and ports but in sliding window storage.

# Chapter 8

# Refining The Catalog

## 8.1 Organizing the Catalog

The 10 patterns presented in this catalog have different scope. Some are very high-level, like Semantic Ports and Driver Ports, while other are much focused on implementation issues, like Phantom Buffer). However, they act as a *pattern language* in the sense that each pattern references higher-level patterns describing the context in which it can be applied, and lower-level patterns that could be used after the current one, to further refine the solution. These relations form a hierarchical structure drawn in figure 8.1. The arcs between patterns mean "enables" relations: introducing a pattern in the system enables other patterns to be used.

This pattern catalog shows how to approach the development of a complete data-flow system for sound and music computing, in an evolutionary fashion without needing a *big up-front design*. The patterns at the top of the hierarchy suggest that you start with high level decisions driven by questions like: "do all ports drive the module execution or not?" and "do you have to deal only with stream flow or also with event flow?" It might also happen that at some point you will need different token types. Then you'll have to decide "does ports need to be strongly typed while connectable by the user?". Each of these questions is addressed by one *General Data-flow Pattern*.

Having reached this point, it is feasible to get a relatively simple but useful

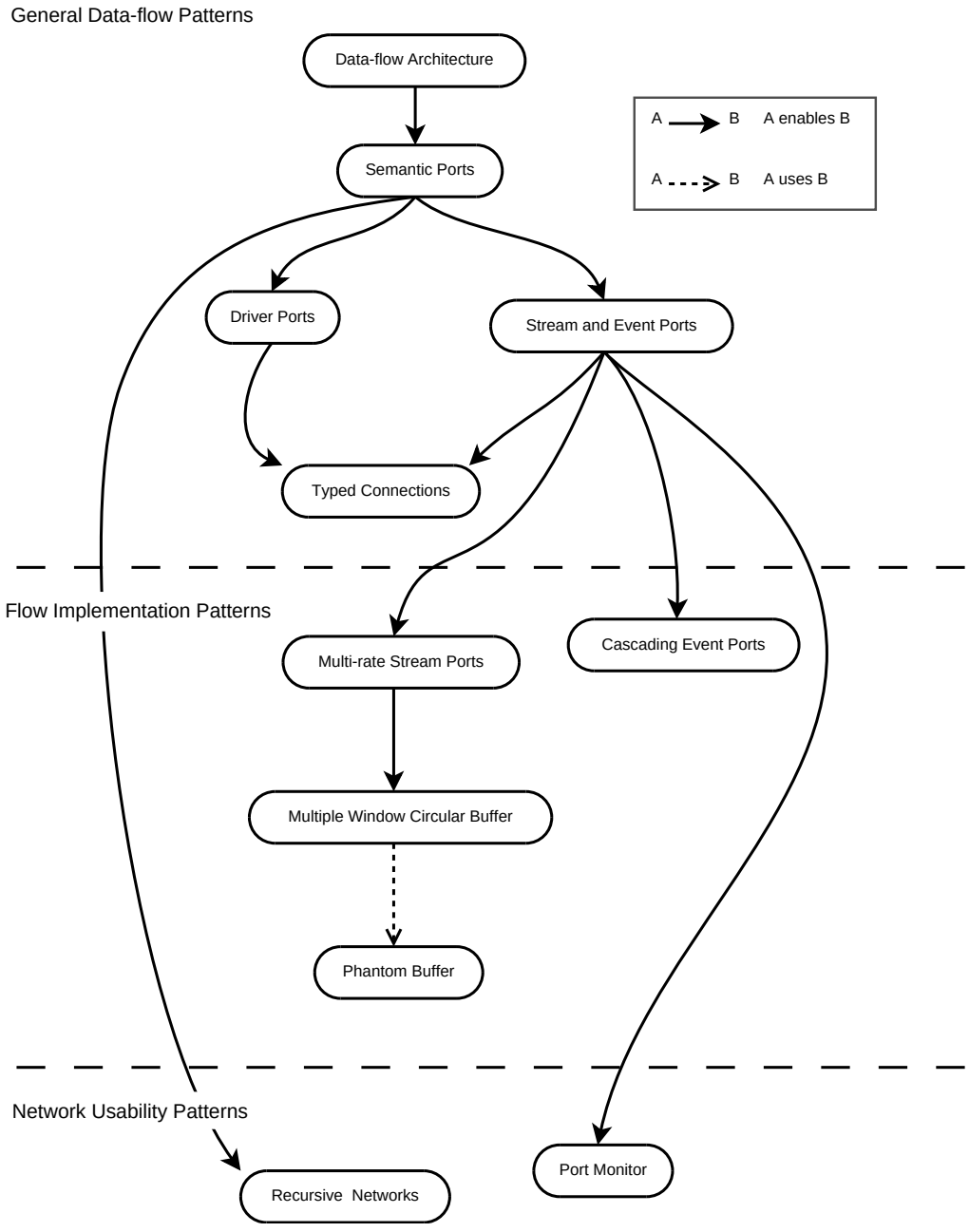


Figure 8.1: Introducing some patterns enables other patterns to be used.

data-flow system allowing many module connections. However, it will have some limitations. For example, limited ports connectivity and fixed block size. You might find yourself in the need of implementing more flexible flows for stream and events. Then you'll want to look at the *Flow Implementation Patterns*. They show how to implement efficient stream flows with flexible connectivity (i.e. consuming/producing different number of tokens) and event controls that propagates immediately.

Humans might needs to interact with your system. Possible interaction includes building (complex) networks and monitoring the flowing data. This is what the *Human Usability Patterns* do. They can be introduced in the first stages of the system evolution or later on.

## 8.2 Sketched Patterns

The following is a brief description of possible future patterns that would fit in the pattern language. All of them have to be worked out and assess that they hold the needed pattern qualities. Nevertheless, this list gives an idea of directions where the catalog could grow.

**Controller Module** Addresses how user events from the out-of-band partition can enter into the in-band partition and propagated through the event ports connections.

**Dynamic repository** Addresses how applications can incorporate new modules without needing a rebuild, while keeping all modules organized in hierarchical structure, using meta-data.

**Configuration Time** Addresses how expensive changes, like memory allocation, in modules can be done, without effecting the network execution, while providing error handling.

**Configuration Object** Addresses how modules can be configured by the user without having to define user interfaces for each module, while allowing configurations persistence.

**Enhanced Types** Addresses how to separate the type parameters from the data itself for an efficient processing, while allowing automatic type compatibility checking and rich data representation.

**Partitioned streaming** Addresses how the flow in a network can be partitioned in high-level orchestrated tasks, when big buffers are not viable. For example, a process needs a large number of consecutive tokens in its input to start producing its stream. User intervention might be needed to define the partition points. Outputs of every phase are stored in data pools. (Examples can be found in Unix pipes, i.e. *cat* | *sed* | *sort* | *sert*, and the D2K framework.)

**Stream time propagation** Addresses how stream-tokens associated time can be propagated though the network given that ports (and possibly module) buffers causes latency. It also allows working with varying-rate streams.

# Chapter 9

# Evaluation of the Patterns

## 9.1 Known Uses Recap

This section collect all the examples found for each pattern, in order to give a general picture.

The source for the examples are the following 10 systems: Pure-Data (PD) [Puckette, 1997], MAX/MSP [Puckette, 1991], Open Sound World (OSW) [Chaudhary et al., 1999], JACK [Davis et al., 2004], Super-Collider3 [McCartney, 2002], CSL [Pope and Ramakrishnan, 2003], Marsyas [Tzanetakis and Cook, 2002], Aura [Dannenberg and Brandt, 1996a], CLAM (framework) [Amatriain and Arumí, 2005, www-CLAM, ], CLAM Music Annotator [Amatriain et al., 2005]

- General Data-flow Patterns:
  - Semantic Ports address distinct management of tokens by semantic.
    PD, MAX/MSP, OSW, JACK, CLAM
  - Driver Ports address how to make modules executions independent of the availability of certain kind of tokens.
    PD, MAX/MSP, OSW, JACK, CLAM
  - Stream and Event Ports address how to synchronize different streams and events arriving to a module.
    SuperCollider3, CSL, Marsyas, OSW, CLAM

  - Typed Connections address how to deal with typed tokens while allowing the network connection maker to ignore the concrete types.
    OSW, Music Annotator, CLAM

- Flow Implementation Patterns:

  - Cascading Event Ports address the problem of having a high-priority event-driven flow able to propagate through the network.
    PD, MAX/MSP, CLAM

  - Multi-rate Stream Ports address how stream ports can consume and produce at different rates;
    Marsyas, SuperCollider3, CLAM

  - Multiple Window Circular Buffer address how a writer and multiple readers can share the same tokens buffer.
    CLAM

  - Phantom Buffer address how to design a data structure both with the benefits of a circular buffer and the guarantee of window contiguity.
    CLAM

- Network Usability Patterns:

  - Recursive Networks makes feasible for humans to deal with the definition of big complex networks;
    PD, MAX/MSP, CSL, OSW, Aura, Marsyas, CLAM

  - Port Monitor address how to monitor a flow from a different thread, without compromising the network processing efficiency.
    CLAM, Music Annotator.

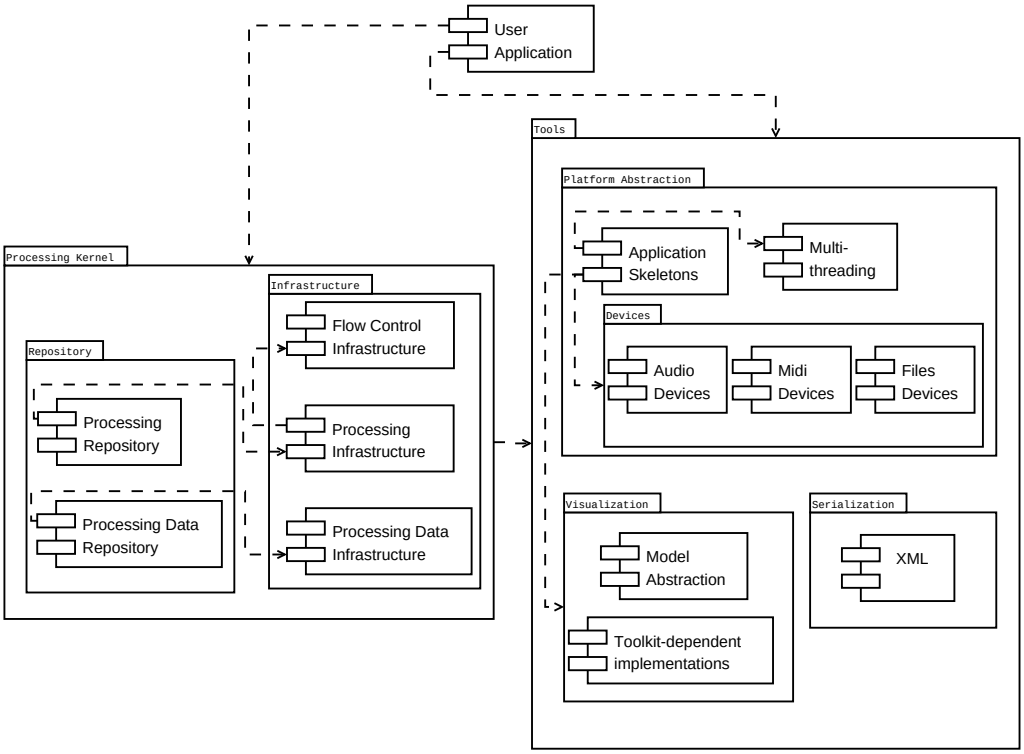


Figure 9.1: The CLAM framework components

## 9.2 Implemented Applications with the CLAM Framework

### 9.2.1 Introduction

CLAM stands for C++ Library for Audio and Music and it is a full-fledged software framework for research and application development in the audio and music domain. It offers a conceptual model; algorithms for analyzing, synthesizing and transforming audio signals; and tools for handling audio and music streams and creating cross-platform applications.

The CLAM framework is cross-platform. All the code is ANSI C++ and it is regularly compiled under GNU/Linux, Windows and Mac OSX.

CLAM offers a processing kernel that includes an *infrastructure* and processing and data *repositories*. In that sense, CLAM is both a *black-box* and a *white-box*

framework as described i [Roberts and Johnson, 1996]. It is black-box because already built-in components included in the repositories can be connected with minimum programmer effort in order to build new applications. And it is *white-box* because the abstract classes that make up the infrastructure can be easily derived to extend the framework components with new processes or data classes.

Apart from the kernel, CLAM includes a number of tools for services such as audio input/output or XML serialization and a number of applications that have served as a testbed and validation of the framework.

The CLAM infrastructure is a direct implementation of the 4MS metamodel, which will be explained in the following section. In the next sections we will also review the CLAM repositories, its tools and applications. Please refer to CLAM's website (www.clam.iua.upf.edu) for further information, documentation, and downloads.

### 9.2.2 CLAM's Metamodel

The Object-Oriented Metamodel for Multimedia Processing, 4MS for short, provides the conceptual framework (metamodel) for a hierarchy of models of media data processing system architectures in an effective and general way. The metamodel is an abstraction of many ideas found in the CLAM framework but also of an extensive review of similar frameworks and collaborations with their authors. Although derived and based in particular for audio and music frameworks, it presents a comprehensive conceptual framework for media signal processing applications. For a more detailed description of the metamodel and how it relates to different frameworks see Xavier Amatriain's PhD [Amatriain, 2004] .

The 4MS metamodel is based on a classification of signal processing objects into two categories: *Processing* objects that operate on data and control, and *Processing Data* objects that passively hold media content. Processing objects encapsulate a process or algorithm; they include support for synchronous data processing and asynchronous event-driven control as well as a configuration mechanism and an explicit life cycle state model. On the other hand, Processing Data objects offer a homogeneous interface to media data, and support for meta object facilities such as reflection and serialization.

Although the metamodel clearly distinguishes between two different kinds of objects the managing of Processing Data constructs can be almost transparent for the user. Therefore, we can view a 4MS system as a set of Processing objects connected in a graph called *Network*.

Processing objects are connected through intermediate channels. These channels are the only mechanism for communicating between Processing objects and with the outside world. Messages are queued (produced) and dequeued (consumed) in these channels, which act as FIFO queues.

The metamodel offers two kinds of connection mechanisms: *ports* and *controls*. Ports transmit data and have a synchronous data flow nature while controls transmit events and have an asynchronous nature. By synchronous, we mean that messages get produced and consumed at a predictable —if not fixed— rate. And by asynchronous we mean that such a rate does not exist and the communication follows an event-driven schema.

But apart from the incoming and outcoming data, some other entity — probably the user through a GUI slider— might want to change some parameters of the algorithm. This control events will arrive, unlike the audio stream, sparsely or in bursts. In this case the processing object will receive these events through various (input) control channels: one for the gain amount, another for the frequency, etc.

The data flows through the ports when a processing is fired (by receiving a *Do()* message).

Processing objects can consume and produce at different rates and consume an arbitrary number of tokens at each firing. Connecting these processing objects is not a problem as long as the ports are of the same data type. The data flow is handled by a *FlowControl* entity that figures out how to schedule the firings in a way that avoids firing a processing with not enough data in its input ports or not enough space into its output ports.

### 9.2.3 Repositories

The *Processing Repository* contains a large set of ready-to-use processing algorithms, and the *Processing Data Repository* contains all the classes that act as

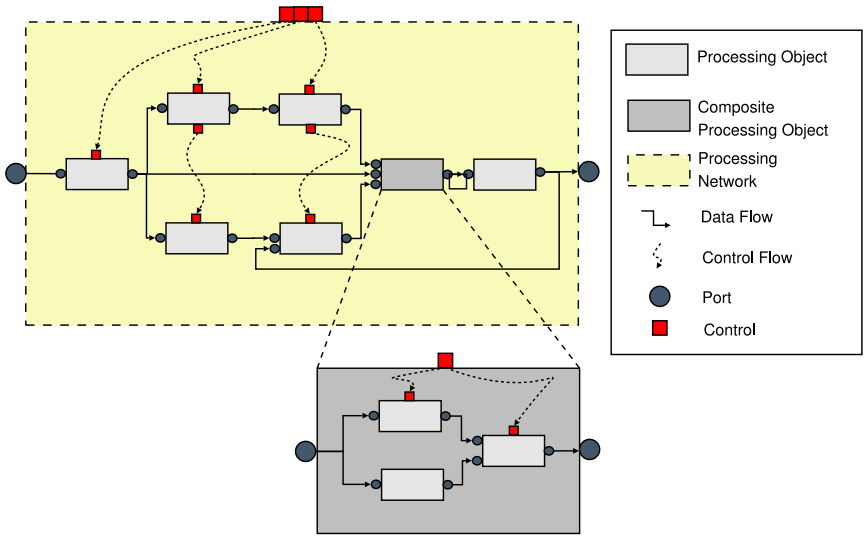


Figure 9.2: a 4MS processing network

data containers to be input or output to the processing algorithms.

The Processing Repository includes around 150 different Processing classes, classified in the following categories: Analysis, ArithmeticOperators, AudioFileIO, AudioIO, Controls, Generators, MIDIIO, Plugins, SDIFIO, Synthesis, and Transformations. Although the repository has a strong bias toward spectral-domain processing because of our group's background and interests, there are enough encapsulated algorithms and tools so as to cover a broad range of possible applications.

On the other hand, in the Processing Data Repository we offer the encapsulated versions of the most commonly used data types such as Audio, Spectrum, SpectralPeaks, Envelope or Segment. It is interesting to note that all of these classes make use of the data infrastructure and are therefore able to offer services such as a homogeneous interface or built-in automatic XML persistence.

### 9.2.4 Tools

Apart from the infrastructure and the repositories, which together make up the CLAM *processing kernel* CLAM also includes a number of tools that can be necessary to build an audio application.

### 9.2.5 XML

Any CLAM *Component* can be stored to XML. Furthermore, Processing Data and Processing Configurations make use of a macro-derived mechanism that provides automatic XML support without having to add a single line of code [Garcia and Amatrian, 2001].

### 9.2.6 GUI

When designing CLAM we had to think about ways of integrating the core of the framework tools with a graphical user interface that may be used as a front-end to the framework functionalities. CLAM offers a toolkit-independent support through the CLAM Visualization Module. This general Visualization infrastructure is completed by some already implemented presentations and widgets. These are offered both for the FLTK toolkit and the Trolltech's Qt framework . An example of such utilities are convenient debugging tools called Plots. Plots offer ready-to-use independent widgets that include the presentation of the main Processing Data in the CLAM framework such as audio, spectrum, spectral peaks...

### 9.2.7 Platform Abstraction

Under this category we include all those CLAM tools that encapsulate system-level functionalities and allow a CLAM user to access them transparently from the operating system or platform.

Using these tools a number of services –such as Audio input/output, MIDI input/output or SDIF file support– can be added to an application and then used on different operating systems without changing a single line of code.

### 9.2.8 Applications

The framework has been tested on —but also has been driven by— a number of applications. Most of them will be introduced in the following paragraphs. The last subsection shows the CLAM visual building tools. Though initially considered separate applications, they now allow visually building applications without writing any line of code thus becoming part of the framework.

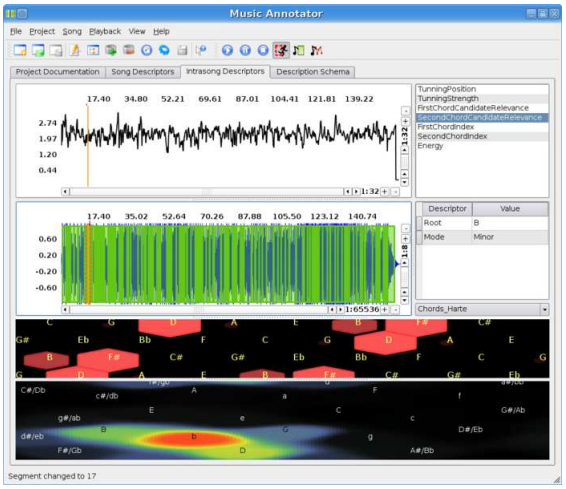


Figure 9.3: Editing low-level descriptors and segments with the CLAM Music Annotator

### 9.2.9 SMS Analysis/Synthesis

The main goal of the application is to analyze, transform and synthesize back a given sound using the Sinusoidal plus Residual model [Serra, 1996]. In order to do so the application reads an XML configuration file, and an audio file or a previously analyzed sdif file. The input sound is analyzed, transformed in the spectral domain according to a transformation score and then synthesized back.

### 9.2.10 The Music Annotator

The CLAM Music Annotator [Amatriain et al., 2005] is a tool for editing audio descriptors. The application can be used as a platform for launching extraction algorithms that analyze the signal and produce different kinds of descriptors. These processes can be either local or web services. But most importantly, the Annotator includes a powerful GUI to manually edit the result of these algorithms from the audio sample level to the song level.

### 9.2.11 SALTO

SALTO is a software based synthesizer that is also based on the Sinusoidal plus Residual technique. It implements a general architecture for these synthesizers but it is currently only prepared to produce high quality sax and trumpet synthesis. Pre-analyzed data are loaded upon initialization. The synthesizer responds to incoming MIDI data or to musical data stored in an XML file. SALTO can be used as a regular synthesizer on real-time as it accepts messages coming from a regular MIDI keyboard or a MIDI breath controller.

### 9.2.12 Spectral Delay

SpectralDelay is also known as CLAM's Dummy Test. In this application it was not important to actually implement an impressive application but rather to show what can be accomplished using the CLAM framework. The SpectralDelay implements a delay in the spectral domain, dividing the audio signal into three bands and allowing for each band to be delayed separately.

### 9.2.13 Others

Apart from the main sample applications CLAM has been used in many different projects that are not included in the public version either because the projects themselves have not reached a stable stage or because their results are protected by non-disclosure agreements with third parties. In the following paragraphs we will outline these other users of CLAM.

Rappid was a testing workbench for the CLAM framework in high demanding situations. Rappid was tested in a live-concert situation. Rappid was used as an essential part of a composition for harp, viola and tape, presented at the Multiphonies 2002 cycle of concerts in Paris.

The Time Machine project implemented a high quality time stretching algorithm that was later integrated and included in a commercial product. The algorithm uses multi-band processing and works in real-time. It is a clear example of how the core of CLAM processing can be used in isolation as it lacks of any GUI or audio input/output infrastructure.

The Vocal Processor is VST plug-in for singing voice transformations. This prototype was a chance to test CLAM integration into VST API and also to check the efficiency of the framework in highly demanding situations. Most transformations are implemented in the frequency domain and the plug-in must work in real-time, consuming as few resources as possible.

The CUIDADO IST European project was completely developed with CLAM. The focus of the project was on automatic analysis of audio files. In particular rhythmic and melodic descriptions were implemented. The Open Drama project was another IST European project that used CLAM extensively. The project focus was on finding new interactive ways to present opera. In particular, a prototype application was built to create an MPEG-7 compliant description of a complete opera play.

### 9.2.14 CLAM as a Rapid Prototyping environment

The latest developments in CLAM have brought *visual building* capabilities into the framework. These allow the user to concentrate on the research algorithms and not on application development. Visual patching is also valuable for rapid application prototyping of applications and audio-plug-ins.

CLAM's visual builder is known as the NetworkEditor (see Figure 9.4). It allows to generate an application—or its processing engine—by graphically connecting objects in a patch. Another application called Prototyper acts as the glue between a graphical GUI designing tool (such as qt Designer) and the processing engine defined with the NetworkEditor.

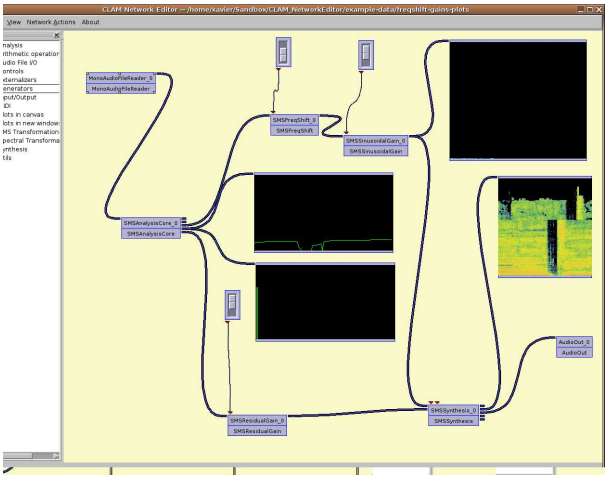


Figure 9.4: NetworkEditor, the CLAM visual builder

# Chapter 10

# Conclusions

## 10.1 Summary of Contributions

This thesis makes a number of contributions:

**Propose a pattern language for the sound and music domain.** We have presented 10 interrelated pattern addressing the following aspects of sound and music data-flow architectures:

*General Data-flow Patterns:* Address problems about how to organize high-level aspects of the data-flow architecture, by having different types of modules connections. Belonging to this category:

- Semantic Ports address distinct management of tokens by semantic.
- Driver Ports address how to make modules executions independent of the availability of certain kind of tokens.
- Stream and Event Ports address how to synchronize different streams and events arriving to a module.
- Typed Connections address how to deal with typed tokens while allowing the network connection maker to ignore the concrete types.

*Flow Implementation Patterns:* Address how to physically transfer tokens from one module to another, according to the types of flow defined by the

*general data-flow patterns.* Tokens life-cycle, ownership and memory management are recurrent issues in those patterns.

- Cascading Event Ports address the problem of having a high-priority event-driven flow able to propagate through the network.
- Multi-rate Stream Ports address how stream ports can consume and produce at different rates;
- Multiple Window Circular Buffer address how a writer and multiple readers can share the same tokens buffer.
- Phantom Buffer address how to design a data structure both with the benefits of a circular buffer and the guarantee of window contiguity.

*Network Usability Patterns:* Address how humans can interact with data-flow networks.

- Recursive Networks makes feasible for humans to deal with the definition of big complex networks;
- Port Monitor address how to monitor a flow from a different thread, without compromising the network processing efficiency.

**Demonstrate that design patterns provide useful design reuse in the domain of sound and music computing.**

On one hand, our patterns covers many (if not most) features of data-flow systems needed in our domain. On the other hand, their solutions are general enough to be used in many different contexts. Finally, all of them provide a teaching component (mostly found in the forces and consequences sections) which is the fundamental key that enables those solutions to be reused effectively.

**Show that all the patterns can be found in different applications and contexts.**

Every pattern have its examples section presenting known uses in real-life systems. Whenever possible we have presented more than three examples

on tools aiming at different purposes. There are few patterns that we have have not been able to find elsewhere than the CLAM framework. However, a framework can be instantiated in many applications. We have developed and shown here a number of CLAM applications with diverse purposes on different fields.

**Show how design patterns are useful to communicate, document and compare the designs of audio systems.**

As an example, consider the documenting value of the following sentence (but take a deep breath first) : "CLAM have a Data-flow Architecture where each stream and events channels are separated using Semantic Ports. It uses Stream and Event Ports and Driver Ports, where the stream ports coincide with the driving ones. While stream ports are Typed Connections with concrete types such as Audio, Spectrum, Note or Melody, event ports are restricted to floats. Event ports are implemented with Cascading Event Ports while audio stream ports uses Multi-rate Stream Ports implemented with Multiple Window Circular Buffer and a Phantom Buffer."

This paragraph concisely conveys an immense amount of design information. Though it might be a bit too dense it shows how using our patterns it is possible communicate and document the overall design of a system efficiently. Of course, it is required that the receptors are familiar with the patterns or, at least, links to the pattern catalog.

As noted Christopher Alexander, patterns are both a "thing" (the design) and "instructions on how to produce the thing" [Alexander, 1977]. When documenting a system design we are using the first meaning.

These patterns are also efficient tools for comparing different systems. Again an example: "MAX/MSP does use Driver Ports but unlike CLAM, its event ports can also be drivers." or "While CSL uses Stream and Event Ports, JACK does not because its network only transports audio samples. And neither one or the other use Typed Connections"

## 10.2 Open Issues and Future Work

### Empirical Quantitative Evaluation

We have not done empirical evaluation of the teaching value of our patterns. This kind of studies take an approach similar to those in social-science. Involves evaluating how individuals perform on technical problems when they know the patterns and when they not. Due to the specialized nature of our patterns, we believe that finding a significant sample of individuals would be very expensive.

Another kind of empirical study suitable for patterns, takes a more Darwinian approach. Consists in collecting data from many development projects: comments in the code, CVS logs, mailing-lists and the like, and then automatically analyzing the data searching for traces of pattern usage. Then, some metrics that capture the activity and "health" of the project are collected and used compare projects that use the patterns with projects that do not. As a side note we want to note that this approach have been used in several studies on *general* patterns and open-source projects [Hahsler, 2004]. Interestingly enough, they detect a clear correlation between the adoption of patterns and the amount of development activity.

This kind of quantitative study for the presented patterns would be very interesting. However, it requires many projects adopting those patterns, and this takes time.

### Growing the Pattern Catalog

The pattern language would be much more complete with patterns that covers aspects like: time propagation in streams, latency management or firing scheduling strategies. Some of these aspects should be covered in new patterns that have been sketched in section 8.2: For example: Stream Time Propagation, Partitioned Streaming and Dynamic Repository.

The presented work focuses on data-flow infrastructure for audio applications. However, there are many sub-domains in sound and music computing where design patterns should be mined: Real-time, Computer-Human-Interaction with real-time synthesis, multi-threading processing, distributed processing or Music

Information Retrieval[1], etc.

Therefore, we expect to do more research on patterns covering these areas before completing the PhD. Sound and music computing is a rich and multidisciplinary area. Collecting a complete pattern catalog is a cumbersome work far beyond of a single dissertation. Nevertheless, it is feasible as a collective effort.

## 10.3 Additional Insights

### Openness

We have found that patterns and open-source works synergystically. Open-source boost patterns on three fronts: One, open-source projects give material to pattern writers for new patterns to mine; two, they are a source of concrete code examples to people learning and using patterns; and three, they provide the necessary data for a quantitative evaluation of its effectiveness in real-life projects.

On the other hand, patterns also boost open-source. As [Seen et al., 2000] observes the design pattern adoption score very high in open-source developers. Specifically, patterns require no infrastructure investment, they can be adopted bottom-up and visible pattern adoption advertises competence. All three properties are certainly more important in an open-source environment than in a traditional company where the necessary infrastructure is provided and the management controls the development process. Another important aspect is about communication efficiency. The channels where communication takes place in open-source development (mailing lists, chats, forums...) are not appropriate for verbose design descriptions and they motivate the use of a concise vocabulary for communicate design ideas.

Finally, our experience is that code documented with design patterns is many times easier to understand. Patterns are not mere design solution but they also carry a deeper knowledge in form of "consequences" or "forces resolution".

[1]Recent contribution on MIR patterns can be found in Aucouturier's thesis [Aucouturier, 2006]

### Agility

Practices of Agile Methodologies like Test Driven Development [Beck, 2002] and Refactoring [Fowler et al., 1999] where key practices for experimenting with design choices. They relay on tools that support automatic tests. Since none of the existing solutions fitted our needs, we developed two lightweight tools for automatic testing: MiniCppUnit [www-MiniCppUnit, ], for C++ unit-testing; and Testfarm [Arumí et al., 2006], for automatic builds and tests in multiple clients and platforms; both released as Free Software. This research would have taken much longer without them.

### Creativity

Our experience with patterns gave us some insights about patterns and creativity. Design is a creative act: so is the creation or application of a design pattern. "If design is codified in patterns, does the need for creativity go away? The answer is that creativity is still needed to shape the patterns to a given context. (...) Patterns channel creativity; they neither replace nor constrain it." [Coplien, 1998]

## 10.4 Closing Statement

This thesis proposes design patterns as a better way to address the problem of software complexity in sound and music computing. We believe that formulation of design experience, methods and insight in the form of pattern languages should be done systematically in the field of sound and music computing. A uniform representation and computer support would help the search of the appropriate pattern.

Many developers in sound and music computing agree that it is difficult to manage the sheer complexity of their systems. We believe that new design patterns will allow software to "grow up" out of the craftsman state and into a more mature state.

# Bibliography

[Alexander, 1977] Alexander, C. (1977). *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, USA.

[Amatriain, 2004] Amatriain, X. (2004). *An Object-Oriented Metamodel for Digital Signal Processing*. PhD thesis, Universitat Pompeu Fabra.

[Amatriain and Arumí, 2005] Amatriain, X. and Arumí, P. (2005). Developing cross-platform audio and music applications with the clam framework. In *Proceedings of the 2005 International Computer Music Conferenc (ICMC'05)*. in press.

[Amatriain et al., 2005] Amatriain, X., Massaguer, J., Garcia, D., and Mosquera, I. (2005). The clam annotator: A cross-platform audio descriptors editing tool. In *Proceedings of 6th International Conference on Music Information Retrieval*, London, UK.

[Appleton, 1997] Appleton, B. (1997). Patterns and software: Essential concepts and terminology.

[Arumí et al., 2006] Arumí, P., Garcia, D., and Amatriain, X. (2006). Testfarm, una eina per millorar el desenvolupament del programari lliure. In *Proceedings of V Jornades de Programari Lliure*, Barcelona.

[Aucouturier, 2006] Aucouturier, J. (2006). *Ten Experiments on the Modelling of Polyphonic Timbre*. PhD thesis, University of Paris 6/Sony CSL Paris.

[Beck, 1988] Beck, K. (1988). Using pattern languages for object-oriented programs. In *ACM SIGPLAN*.

[Beck, 1999] Beck, K. (1999). *Extreme Programming Explained*. Addison Wesley.

[Beck, 2002] Beck, K. (2002). *Test-Driven Development*. Addison-Wesley Professional.

[Beck et al., 1996] Beck, K., Coplien, J. O., Crocker, R., Dominick, L., Meszaros, G., Paulisch, F., and Vlissides, J. (1996). Industrial experience with design patterns. In *Proceedings of the 18th International Conference on Software Engineering*, pages 103–114. IEEE Computer Society Press.

[Beck and Johnson, 1994] Beck, K. and Johnson, R. (1994). Patterns generate architectures. *Lecture Notes in Computer Science*, 821:139–??

[Booch, 1994] Booch, G. (1994). *Object-Oriented Analysis and Design with Applications*. Benjamin/Cummings, second edition edition.

[Borchers, 2000] Borchers, J. O. (2000). A pattern approach to interaction design. In *Symposium on Designing Interactive Systems*, pages 369–378.

[Buck and Lee, 1994] Buck, J. and Lee, E. A. (1994). *Advanced Topics in Dataflow Computing and Multithreading*, chapter The Token Flow Model. IEEE Computer Society Press.

[Buschman et al., 1996a] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996a). *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons.

[Buschman et al., 1996b] Buschman, F., Meunier, R., Rohnert, H., Sommerlad, P., and Stal, M. (1996b). *Pattern-Oriented Software Architecture - A System of Patterns*. John Wiley & Sons.

[Chaudhary et al., 1999] Chaudhary, A., Freed, A., and Wright, M. (1999). An Open Architecture for Real-Time Audio Processing Software. In *Proceedings of the Audio Engineering Society 107th Convention*.

[Cook and Scavone, 1999] Cook, P. and Scavone, G. (1999). The Synthesis Toolkik (STK). In *Proceedings of the 1999 International Computer Music Conference (ICMC99)*, Beijing, China. Computer Music Association.

[Coplien, 1998] Coplien, J. (1998). Software Design Patterns: Common Questions and Answers. *The Patterns Handbook: Techniques, Strategies, and Applications. Cambridge University Press, NY, January*, pages 311–320.

[Dannenberg and Brandt, 1996a] Dannenberg, R. B. and Brandt, E. (1996a). A Flexible Real-Time Software Synthesis System. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273.

[Dannenberg and Brandt, 1996b] Dannenberg, R. B. and Brandt, E. (1996b). A Portable, High-Performance System for Interactive Audio Processing. In *Proceedings of the 1996 International Computer Music Conference (ICMC96)*, pages 270–273. International Computer Music Association.

[Davis et al., 2004] Davis, P., Letz, S., D., F., and Orlarey, Y. (2004). Jack Audio Server: MacOSX port and multi-processor version. In *Proceedings of the first Sound and Music Computing conference - SMC04*, pages 177–183.

[Douglass, 2003] Douglass, B. P. (2003). *Real-Time Design Patterns*. Addison-Wesley.

[Edwards, 1995] Edwards, S. (1995). Streams: a Pattern for "Pull-Driven. In Coplien, J. O. and Schmidt, D. C., editors, *Pattern Languages of Program Design*, volume vol.1, chapter 21. Addison-Wesley.

[Foote, 1988] Foote, B. (1988). Designing to Facilitate Change With Object Oriented Frameworks. Master's thesis, University of Illinois at Urbana Champaign.

[Fowler et al., 1999] Fowler, M., Beck, K., Brant, J., Opdyke, W., and Roberts, D. (1999). *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

[Gamma et al., 1995] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1995). *Design Patterns - Elements of Reusable Object-Oriented Software*. Addison-Wesley.

[Garcia and Amatrian, 2001] Garcia, D. and Amatrian, X. (2001). XML as a means of control for audio processing, synthesis and analysis. In *Proceedings of the MOSART Workshop on Current Research Directions in Computer Music*, Barcelona, Spain.

[Graham, 1991] Graham, I. (1991). *Object Oriented Methods*. Addison-Wesley.

[Hahsler, 2004] Hahsler, M. (2004). A quantitative study of the application of design patterns in java.

[Hylands et al., 2003] Hylands, C. et al. (2003). Overview of the Ptolemy Project. Technical report, Department of Electrical Engineering and Computer Science, University of California, Berklee, California.

[Judkins and Gill, 2000] Judkins, T. and Gill, C. (2000). A Pattern Language for Designing Digital Modular Synthesis Software.

[Kay, 1993] Kay, A. (1993). The early history of Smalltalk. *History of Programming Languages*, pages 69–95.

[Lazzarini, 2001] Lazzarini, V. (2001). Sound Processing with the SndObj Library: An Overview. In *Proceedings of the 4th International Conference on Digital Audio Effects (DAFX '01)*.

[Lee and Park, 1995] Lee, E. and Park, T. (1995). Dataflow Process Networks. In *Proceedings of the IEEE*, volume 83, pages 773–799.

[Manolescu, 1997] Manolescu, D. A. (1997). A Dataflow Pattern Language. In *Proceedings of the 4th Pattern Languages of Programming Conference*.

[McCartney, 2002] McCartney, J. (2002). Rethinking the Computer Music Language: SuperCollider. *Computer Music Journal*, 26(4):61–68.

[Meunier, 1995] Meunier, R. (1995). The Pipes and Filter Architecture. In Coplien, J. O. and Schmidt, D. C., editors, *Pattern Languages of Program Design*, volume vol.1, chapter 22. Addison-Wesley.

[Parks, 1995] Parks, T. M. (1995). *Bounded Schedule of Process Networks*. PhD thesis, University of California at Berkeley.

[Pope and Ramakrishnan, 2003] Pope, S. T. and Ramakrishnan, C. (2003). The Create Signal Library ("Sizzle"): Design, Issues and Applications. In *Proceedings of the 2003 International Computer Music Conference (ICMC '03)*.

[Posnak et al., 1996] Posnak, E. J., Lavender, R. G., and Vin, H. M. (1996). Adaptive pipeline: an object structural pattern for adaptive applications. In *Proceedings of the 3rd Pattern Languages of Programming Conference*, Monticello, Illinois.

[Posnak and M., 1996] Posnak, E. J. Lavander, R. G. and M., H. (1996). Adaptive pipeline: an object structural pattern for adaptive applications. In *The 3rd Pattern Languages of Programming conference*, Monticello, IL, USA.

[Prechelt et al., 1998] Prechelt, L., Unger, B., Philippsen, M., and Tichy, W. (1998). Two controlled experiments assessing the usefulness of design pattern information during program maintenance.

[Puckette, 1991] Puckette, M. (1991). Combining Event and Signal Processing in the MAX Graphical Programming Environment. *Computer Music Journal*.

[Puckette, 1997] Puckette, M. (1997). Pure Data. In *Proceedings of the 1997 International Music Conference (ICMC '97)*, pages 224–227. Computer Music Association.

[Puckette, 2002] Puckette, M. (2002). Max at Seventeen. *Computer Music Journal*, 26(4):31–43.

[Roberts and Johnson, 1996] Roberts, D. and Johnson, R. (1996). Evolve Frameworks into Domain-Specific Languages. In *Procedings of the 3rd International Conference on Pattern Languages for Programming*, Monticelli, IL, USA.

[Seen et al., 2000] Seen, M., Taylor, P., and Dick, M. (2000). Applying a crystal ball to design pattern adoption. *tools*, 00:443.

[Serra, 1990] Serra, X. (1990). Spectral Modeling Synthesis: A Sound Analysis/Synthesis System based on a Deterministic plus Stochastic Decomposition. *Computer Music Journal*, 14(4):12–24.

[Serra, 1996] Serra, X. (1996). *Musical Signal Processing*, chapter Musical Sound Modeling with Sinusoids plus Noise. Swets Zeitlinger Publishers.

[Shaw, 1996] Shaw, M. (1996). Some Patterns for Software Architecture. In Vlissides, J. M., Coplien, J. O., and Kerth, N. L., editors, *Pattern Languages of Program Design*, volume vol.2, chapter 16. Addison-Wesley.

[Szyperski, 1998] Szyperski, C. (1998). *Component Software: Beyond Object-Oriented Software*. ACM/Addison-Wesley.

[Tzanetakis and Cook, 2002] Tzanetakis, G. and Cook, P. (2002). *Audio Information Retrieval using Marsyas*. Kluewe Academic Publisher.

[van Dijk et al., 2002] van Dijk, H. W., Sips, H. J., and Deprettere, E. F. (2002). On Context-aware Process Networks. In *Proceedings of the International Symposium on Mobile Multimedia & Applications (MMSA 2002))*.

[Vlissides, 1998] Vlissides, J. (1998). *Pattern Hatching, Desing Patterns Applied*. Addison-Wesley.

[Weinand et al., 1989] Weinand, A., Gamma, E., and Marty, R. (1989). Design and Implementation of ET++, a Seamless Object-Oriented Application Framework. *Structured Programming*, 10(2).

[Wright, 1998] Wright, M. (1998). Implementation and Performance Issues with Open Sound Control. In *Proceedings of the 1998 International Computer Music Conference (ICMC '98)*. Computer Music Association.

[www-CLAM, ] www-CLAM. CLAM website: http://www.iua.upf.es/mtg/clam.

[www-MiniCppUnit, ] www-MiniCppUnit. MiniCppUnit (a C++ xUnit port with a minimalistic aproach) homepage, http://www.iua.upf.es/ parumi/MiniCppUnit/.

[www-PatternsEssential, ] www-PatternsEssential. Pattern and Software: Essential Concepts and Terminology, http://www.cmcrossroads.com/bradapp/docs/patterns-intro.html.