

---

**Steffen Brandorff,\* Morten Lindholm,† and Henrik Bærbak Christensen†**

\*Institute of Information and Media Studies  
Aarhus University,  
Helsingforsgade 14, DK-8200 Aarhus N, Denmark

†Department of Computer Science  
Aarhus University  
Aabogade 34, DK-8200 Aarhus N, Denmark

# A Tutorial on Design Patterns for Music Notation Software

Design patterns are becoming standard in the computer software community, as they convey “best practice” solutions to standard software-design problems. A new music-notation editing application serves as an interesting “problem generator,” and the complex notational conventions exemplify the ways design patterns can be applied to such non-computer logic. This article is intended as a tutorial. It introduces some well-established patterns to the novice, discussing the reasoning behind each pattern.

## OMIS: Optical Music Interpretation System

Around 1998, a music-notation project was initiated at the University of Aalborg, Denmark. The vision was to be able to place old, printed sheet music (instrumental parts) onto a scanner, scan them, optically recognize and interpret the notational units involved, and ultimately print them out following modern music notation conventions.

The authors have participated in the second part of the project, which consists of reading a textual representation of the music, assembling the various parts into a score, spacing the columns, and printing the result. As might well be expected, this work is not yet fully finished, but a great deal of experience has been gathered in the process.

Several others have participated in the group, each with special skills, particularly Henrik Lynbech and Jesper D. Thomsen, both computer science students. Algorithm examples are quoted in C++, but they are kept quite simple, which we hope will improve legibility.

## Music Notation

From the neumes of early music onward, an ever-increasing precision has been required from music notation. Symbols have been added or altered, and new musical ideas have demanded extended notational solutions. In short, the requirements have expanded to include expressions of new musical epochs.

Standard music notation outlines a time/frequency grid that must coerce all other kinds of information into its form. Fingerings and accents have a “point form” and fit into this grid fairly easily, but other elements, like dynamic markings, are dealt with in ambiguous ways. Some have a point representation (e.g., *forte*), whereas others, like “hairpins” (crescendo and decrescendo marks), cover a time interval. We, the music readers, are accustomed to them, but the system does not unfold in a logical fashion to a machine.

Musicians and conductors read music by interpreting its symbols into sounding reality. This “decoder” will often reside implicitly in the interpreter, who will usually know the background of the piece and thus be able to provide an emotionally and musically adequate articulation.

We see a globally accepted format for notation emerging, as described in Ross (1970) or Gerou and Lusk (1996). National variants were previously quite common; for example, the notation of “flagged” notes in Germany and France. Even now, some variations survive, and any notation package striving for completeness must be able to adapt to such variations. New notational practices add to this need.

For most of its history, notation has been a hand-written skill, creating jobs for educated, professional copyists. With the introduction of mechanical printing, music notation had to standardize its symbols, but even with printed music, engravers used

---

hand tools and their professionally trained eyes when spacing and laying out the page. Each publishing house could have its own standards, and each music engraver could have a personal aesthetic fingerprint. Much music never appeared in print, and we are left with manuscripts with all the individuality of handwriting.

In short, the tradition of notating music is quite diffuse and varied, and this is one reason that it is difficult to reduce its aesthetics to a few principles for a machine to follow. How, then, do we make a logical system out of the varying traditions and schools, some of which are influenced by the needs of an individual composer in a specific piece? And how and where should we implant a logic for positioning staccato dots when the position of those depends on so many different factors, such as note position, note size, stem direction, beaming, etc.? How do we grasp the styles of using courtesy accidentals when some use them and others do not?

The key must be flexibility. It must be easy to create variants and alternatives, which is in fact one of the goals of the object-oriented paradigm. But even here, basic differences can be seen. An important phase of object-oriented software design is to establish a hierarchy of the participating objects. Objects include containers that can consist of objects or other containers. Said another way, we can nest objects inside other objects. This way of thinking differs from standard notation conventions, as can be seen in the treatment of horizontal groups (slurs, ties, beams, hairpins, etc.). Standard object-oriented container paradigms prefer to nest groups (e.g., a beam started “inside” a slur must end before the slur can end). In music, such limitations make no sense; groupings are not usually “nestable.”

All such conventions from various parts of music history would ideally have to be incorporated in the design of a notation program. In the present implementation, however, emphasis initially has been on working with 19th-century Western classical music, securing slots for other eras and styles.

In transferring standard notation guides to a computer program, it would be desirable to have some analytically derived, clear precedent to lean on, instead of being forced to regard every detail as new or unique. Perhaps experience from other fields could

be invoked, thereby making the present task easier. The following documents our findings in the realms of architecture and software design.

## Design Patterns

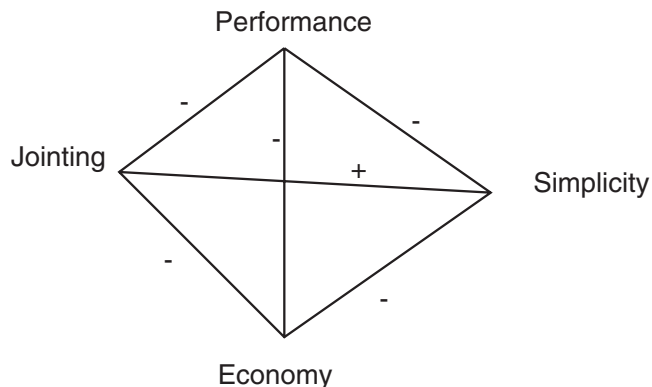
The origin of the term “design pattern” can be traced back to the book *Notes on the Synthesis of Form*, written by American architect Christopher Alexander (1964). Mr. Alexander examined what he thought to be a new foundation for architectural design, namely that of achieving a “good fit” between form and context (see Figure 1). To illustrate these concepts, he considered the task of designing a kettle. On one hand, there are constraints regarding the physical shape (the “form”) the designer must keep in mind. The kettle must hold water, it must be possible to drain it without scalding oneself, it should be convenient to fill, etc. On the other hand, there are equally important constraints regarding the context of use; should it be designed for gas, electricity, or an open fire? And what about manufacturing? It should be inexpensive to produce and easy to assemble. A design that achieves a “good fit” is a design in which all the constraints are taken into consideration and have been balanced against each other.

Mr. Alexander originally strove to create a “science of design” by using mathematics to analyze the various elements, or “forces,” that influence the solution to the design problem. With this approach, he hoped to overcome the growing complexity of design problems. His analysis of forces later evolved into what is now called “design patterns.”

During the 1970s, Mr. Alexander and colleagues published three books in which they applied the pattern concept to the design of houses and urban plans (see Alexander et al. 1975; Alexander, Ishikawa, and Silverstein 1977; and Alexander 1979). In his books, Mr. Alexander orders the patterns by size, beginning with the largest down to the smallest. The largest of the architectural patterns applies to a region, and the smallest applies to some construction detail. Each pattern is connected to the other so that no pattern becomes an isolated entity. The total of all the patterns grows into a “pattern catalog” to be

Figure 1. Christopher Alexander's original explanation of forces. His example concerned the production of a kettle, and hence the need for "joint-

ing." Plus or minus symbols indicate whether the constraints have a positive agreement or are in conflict.



used as a design tool. This term involves the interrelation between different patterns, where the larger-scale patterns are constructed from smaller-scale patterns. A house is built using small patterns, houses are then clustered following larger patterns, and finally, the relation between towns is laid out according to the largest patterns.

Even though Mr. Alexander writes his patterns in a specific way, there are no established rules for writing down patterns. The generally accepted format follows his form verbatim and is therefore known as the Alexandrian Form of pattern writing.

### The Purpose of Patterns

The first question is what problem patterns attempt to solve. The answer is complexity. Mr. Alexander noted in 1964 that the modern world has become increasingly more complex and that no modern individual, no matter how capable, can hold all current knowledge. The consequence is the proliferation of specialists, each knowledgeable in a certain area. His observation was that the world, and thus also design activities, have accelerated. Consequently, there is a shorter and shorter time span to complete designs. He then used these two observations to comment on the (to him) apparent lack of quality in modern architecture. To do that, he first had to define what is quality in architecture, or more generally in design. It proved more difficult than he anticipated, because quality in anything is difficult to express verbally. It is not easy to say what charac-

teristic of an object or house that makes it possess quality; it is easier to tell when something does not. It might have a "wrong" shape or lack harmony in some way. Mr. Alexander's definition of quality thus became a negating definition, and he named the concept a "quality without a name."

This brings us to his second observation, that of speed. Mr. Alexander found that some houses of age had this "quality without a name"—the same quality he also found in palm-leaf huts and Polynesian architecture. He found the reason to be rooted in the cultures that built these houses: they were pre-modern, and their techniques evolved slowly over time. Therefore, when a house was built and there was something about it that did not look "right," the wrong feature was not included in the next house to be built. If a scarf was woven and the pattern did not please, the pattern was not used again. It follows that quality is the result of a historic process in which the odd designs were culled and the pleasing designs were elaborated. Quality is viewed as an emerging concept. It was refined to a degree where even a small alteration disrupted the balance between the various elements affecting the design. The design converged on perfection—a perfect match between form and context.

The central question for Mr. Alexander became how to achieve this perfect match, the good fit, when the designer was constrained by lack of knowledge and a shortage of time. Initially, he tried to solve this problem by using mathematics and logic, but later on, his development of the pattern concept to describe interrelating parts of the design proved more fruitful.

The fact that patterns describe interrelating parts of the design leads to a recursive use of patterns (i.e., using a pattern to describe a part of another pattern). Mr. Alexander saw this as one of the great advantages of his approach, and he called this use a "pattern language"—a language for speaking about design in terms of patterns consisting of other patterns. This also reflects the structure of the books he wrote, in which large-scale patterns describing the layout and placement of towns are themselves divided into smaller-scale patterns describing neighborhoods and still smaller-scale patterns dealing with the construction of houses and rooms.

---

Following Mr. Alexander's development of the ideas behind patterns, it is helpful to see a pattern as a communicative device used to convey knowledge among designers about problem solving and design. The pattern itself is an open-ended structure that, by itself, does not force a specific design into the debate, but rather hints at relevant possibilities, taking surrounding structures into consideration.

### Patterns as a Communication Device

Another aspect of using patterns in interpersonal communication is the implicit transfer of knowledge that takes place when a person is introduced to patterns. Because the pattern itself encompasses knowledge that former designers have found to be well-founded, the act of learning a pattern presents a special case of learning from the "masters." New ideas are not presented as monolithic solutions but rather in the context of the elements (forces) that may influence how the resulting design may deviate from the actual described solution. Combined with the notion of using pattern languages, this constitutes knowledge transfer, in a way more systematic and encompassing than the presentation of singular, optimal solutions.

The idea of using patterns in architecture did not catch on, and, during the 1980s, the idea apparently was disavowed by architects. In another community, however, the patterns concept started to generate interest, namely in the computer-science/software-engineering camp. In the mid to late 1980s, rumors about "patterns" were heard at conferences on software design.

In 1987, Kent Beck and Ward Cunningham presented the first paper on patterns at the 1987 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'87). Beck and Cunningham (1987) describes the results of developing a small pattern language with the goal of guiding novice Smalltalk programmers in creating applications. Over the next few years, interest in patterns was growing, and in 1995, the seminal book *Design Patterns: Elements of Reusable Object-Oriented Software* by Eric Gamma et al. was published. This book brought the term "Design Pattern" into main-

stream use in software development, and it is still considered an essential book on patterns.

Many other books and articles describing additional patterns have been published, but Gamma et al. (1995) remains one of the most influential books on software design of that period. Gamma et al. stated that "patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context" (p. 3). This definition still stands.

The idea was contagious and spread to other fields. In software design, it has been more successful than in any other field we know. In music, however, we could easily find examples similar to applying pattern thinking; in some ways, this is the very basis of the "genre" concept. Form conventions in any style could be regarded as examples of the same kind of thinking.

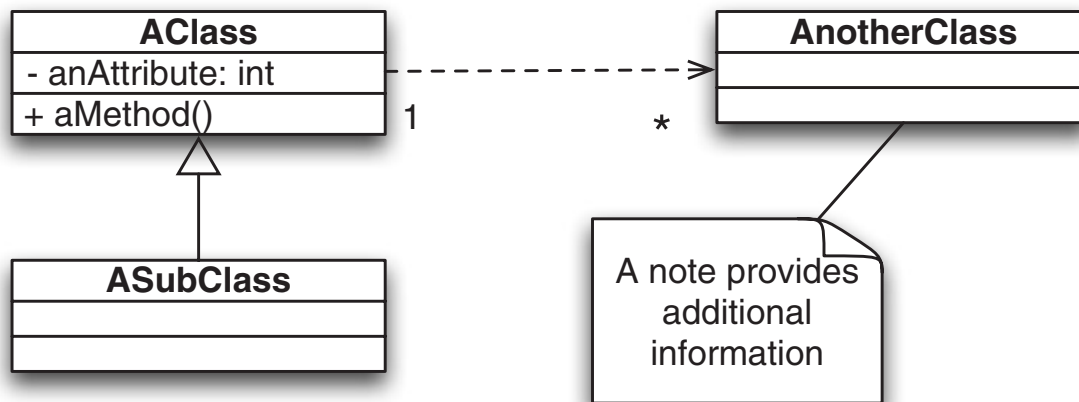
In music, one could interpret the Classical sonata form, for example, as such a design pattern. It prescribes a layout for a whole movement, the character of the themes, the tonal master plan, and so on. Needless to say, using a sonata form as a template will not ensure that the result will be good music. It may be quite unoriginal and boring, but it will be easy to follow and understand.

Good software design has a different goal than does music. In most cases, a program is vastly preferable when it solves problems in an "unoriginal" and indeed common way—and therefore is easy to understand. An imperative quality of a program is that it is free from errors. A program structure that is easy to understand is much less prone to having errors introduced by programmers.

A defining characteristic of patterns is that one does not invent them; one finds them. A pattern must have proven its worth in real life, and it must have done so more than once. As a rule of thumb, a pattern is not really considered a pattern until it has been verified as a recurring principle, a solution that is general and powerful enough to have been used several times and solved problems in existing systems. A so-called "rule of three" is used in selecting patterns. A solution must be observed at least three times in different systems before it should be considered a pattern.

Some applications use patterns extensively to

Figure 2. A very simple class diagram (UML notation).



demonstrate their use. An example is the graphical framework JHotDraw, which was originally developed as a study in design-pattern usage. From the outset of our project, JHotDraw was a major inspiration.

### Notation for Design: Unified Modeling Language

Musical notation is used to convey musical information among musicians. Software developers and programmers have likewise developed notations to convey program design. After a couple of years of debate, the Unified Modeling Language (UML) has emerged as a more or less commonly accepted standard for describing software design using a visual language. UML describes software at a higher level than the executable level of program languages, but it is still tightly coupled to the programming-language level in its basic form.

Design patterns are usually described using various forms of UML diagrams. We will use the class diagram notation from UML, and, as a service to readers not familiar with this notation, we will briefly explain the basic UML class diagram constructs.

Figure 2 shows a very simple class diagram. A rectangular box is used to denote classes. A class defines both a specification of operations (methods) as well as their concrete implementations. The class box is subdivided. The upper box contains the name of the class (e.g., AClass). Class attributes or vari-

ables are shown in the middle box, and methods are shown in the lower box. Thus, in the example above, the AClass class defines the method `aMethod()`. The minus (-) and plus (+) denote whether a variable or method is accessible from other classes. This is a particularity of object-oriented thinking, but it is of less relevance for the present discussion.

In Figure 2, lines between classes represent relations. Lines with triangular arrowheads represent inheritance. Thus, the relationship between AClass and ASubClass shows that AClass is a generalization of ASubClass, which is derived from the former. The dotted line with an arrow is a relation denoting association. Association is a weak relationship between two classes telling us that these classes work together by invoking methods on each other. The arrow's direction in this example shows that AClass can call methods in AnotherClass but not the opposite: AClass knows AnotherClass, but not vice versa. Finally, *multiplicity* is shown by numbers (an asterisk denotes "zero or more") next to the relations. This means that objects of class AClass can be associated with multiple objects of class AnotherClass, but these are in return only associated with a single AClass object.

Finally, it is often advantageous to add informal information to a class diagram. This is accomplished in Figure 2 with a UML *note*, a document symbol with a flipped corner. We have used notes as in Gamma et al. (1995) to show parts of our class implementations.

## Examples of Design Patterns in a Graphical Music Editor

Having several hundred design patterns (see Rising 2000) at our disposal, one main concern was to select the patterns relevant to our application. In the following sections, we present three well-known patterns that have served us well in the process. These are the *Observer Pattern*, the *Abstract Factory Pattern*, and the *Strategy Pattern*. These all have musical pendants. Other cardinal patterns work on the internal structure of an application and are of less musical interest.

### The Observer Pattern: A Behavioral Pattern

Notes, of course, are of primary concern in a notation program. The representation of a note must consider all the contexts of notes in practical uses. For one, the choice of an internal representation of a note will depend on what kind of editing will actually take place.

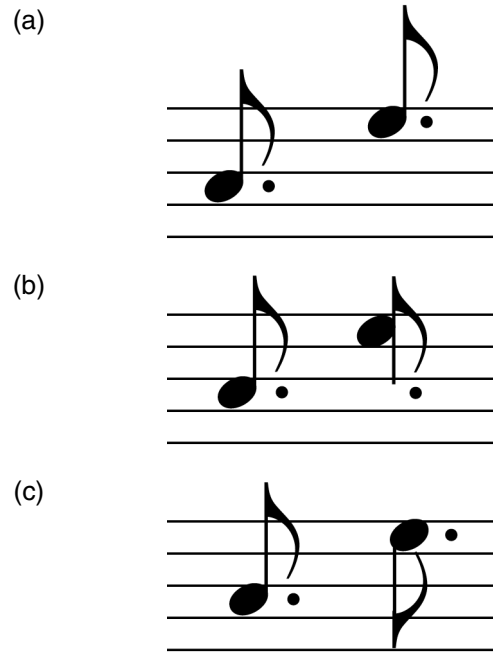
The first thought of a programmer might be to represent notes the way that, for example, the Adobe Sonata font seems to imply: use a combined symbol for flagged notes. This is possible, but how should we move notes about during the editing process? Here, we recapitulate some alternatives of the very simple process of shifting a flagged-and-dotted note upward, as outlined in Figure 3. Three suggestions are shown in this figure, where the first note in each “bar” is shifted to produce the second note.

A first option is to move the note as a whole. This, however, does not adhere to conventions about how stem directions must behave on a single-voice staff. The next design could be to construct notes out of independent atoms, like the note head, stem, flags, dots, etc. This design would lead to an interim scenario in which every atom must be edited separately. This is clearly too tedious for the editor.

The last example lets the editor move the note head, and other parts of the note decide for themselves what graphical action they must take to keep the note as a whole visually correct. The dot will follow the note head directly; the stem will change its point of attachment to the note head;

Figure 3. Three possibilities for transposing a note: (a) moving everything as one entity (a glyph);

(b) moving atoms one at a time; (c) letting each part know what to do when changes occur.



and the flag must change completely, either by some graphical transformation or by changing its symbol of representation. The point here is that the various parts of a note must respond in appropriate but different ways when the position of the note head is changed.

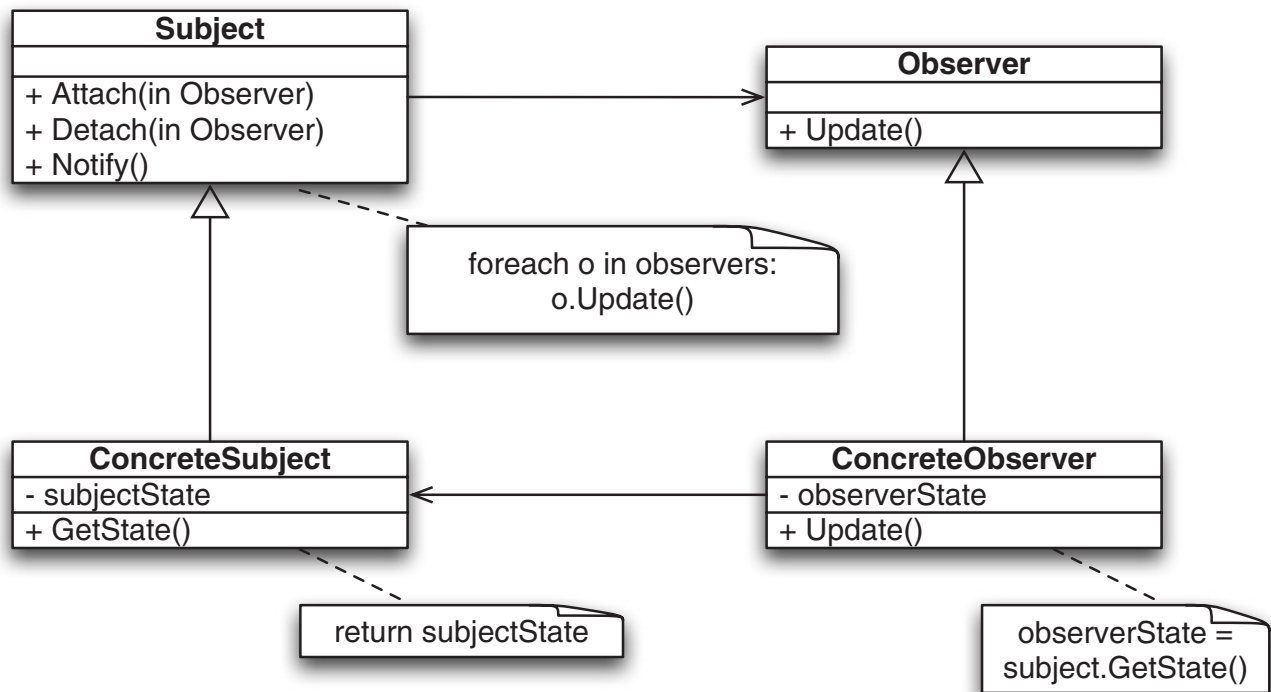
The *Observer Pattern* is a pattern that provides a solution to problems of keeping a number of objects consistent. It is a behavioral pattern, signifying that it is concerned with algorithms and the assignment of responsibilities among objects. This class of patterns therefore decrees the communication among different objects, and it is used to abstract communicative behavior from the program's structure.

The pattern consists of two elements, the *observer*, and the object being observed, also referred to as the *subject*. In our example above, the note head would be the subject, and the stem, dot, and flag are observers. The Observer Pattern defines a one-to-many relationship between a subject and a number of observers. It handles the situation where a change in the subject can trigger changes in a number of other objects. If we chose to program this relation by using absolute references among all the objects, we might end up with a very tightly coupled

Figure 4. Diagram of the principle of an Observer Pattern. In a musical context, the ConcreteSubject could be a note head. All accompanying parts (stem,

flags, etc.) are ConcreteObservers that attach themselves to the note head. Whenever the note head is changed in some way, it will notify all attached ob-

jects by sending an Update() message to every one of them. This will prompt them to find out what has changed and react accordingly.



system in which even small internal changes in some of the objects would lead to mysterious failures in other parts of the program. Instead, observers can subscribe to a list that we make inside the subject. Whenever something new happens, they will be notified.

The Observer Pattern abstracts all knowledge of dependent objects behind a simple interface: the subject/object distinction.

A diagram of the pattern is shown in Figure 4. The subject class defines three methods: an Attach(Observer) method, with which the other objects can register their interest in the subject; a Detach(Observer) method, which they can use to remove themselves from the list of observers; and a Notify() method. The first two methods usually include some kind of container that can hold references to all the observers, and the Notify() method can iterate over this container to notify each observer.

The Observer class defines only a single method, Update(), which is the method called by the Subject when executing the Notify() method. As Figure 4

shows, the Subject does not know anything about the observers, except that they will implement the Update() method. By contrast, each Observer must know enough about the Subject to retrieve the relevant information from it when needed. It is therefore the responsibility of the Observer to inspect the Subject when notified and to react to the changes in the Subject.

Thus, in the situation outlined in Figure 3c, the observers of the note (flag, stem, and dot) have attached themselves using the Attach method to the note-head object. Every time the note head is moved by the editor, it calls the Update method on each attached Observer object: the flag object, the dot object, etc. They can then inspect the new coordinates of the note head and draw themselves appropriately at the new location. (For other reasons, we later chose to have the note head draw the stem in our design.)

The observer approach is ideal for many small-scale, localized, graphical movement operations, like the example of shifting notes. However, we

---

have found that it is not feasible for large-scale operations in the music-editing process such as repagination, merging voices, or moving bars from one page to another. The problem with such operations is that they require “global” information to compute the proper graphical layout. Acquiring this by using the observer protocol would lead to an extremely complex web of communicating objects. Here, we found a better approach in encapsulating such complex operations in a more traditional procedure that traverses the object graphs and forces new positions of objects. Because this is a single process, it can maintain the global state necessary during the operation—something one would not want to build into every note object.

### Abstract Factory: A Creational Pattern

Another obvious requirement for music-notation software is the ability to insert notation into an existing score. A typical example is inserting a note into the score. The need to insert a note is present in several places in the OMIS software: the editor must be able to do so, and so must the part of OMIS that interprets the textual output from the scanning phase. As software designers, we want to encapsulate this note-creation process into one method. The underlying code must do something along like the following:

```
void insertNote(Staff *staff, int atStaffDegree) {
    Note *newNote = new Note();
    staff->insertNote(newNote, atStaffDegree);
}
```

This code snippet inserts a graphical note object with a given pitch (staff degree) into the graphical staff object. For the purpose of this discussion, we have ignored attributes like duration here.

This appears very general and thus rather simple. However, this small code fragment touches upon a problem that computer science calls “coupling.” Coupling is a qualitative measure of how strongly two pieces of software depend upon each other. The `insertNote` method is strongly coupled to the `Note` class—the “new” statement mentions the class name explicitly.

A problem arises when the editor wants to insert percussion (“drum”) notes. Percussion notes look different graphically, and we have defined a subclass of `Note`, `DrumNote`, that embodies such changed appearance, as this allows the drum notes to inherit most of the behavior from ordinary notes. But how do we create drum notes?

One proposal is to make another method, `insertDrumNote`, with identical code except for the new statement. This leads to code duplication that must be avoided because duplicated code is a maintenance nightmare. Another slightly better suggestion is to define a global Boolean flag that tells if we are editing drum notation and then switches on this flag in the following way:

```
void insertNote(Staff *staff, int atStaffDegree) {
    Note *newNote;
    if ( isInsertingDrumNotation ) {
        newNote = new DrumNote();
    } else {
        newNote = new Note();
    }
    staff->insertNote( newNote, atStaffDegree );
}
```

But what about inserting grace notes? And what happens if we need to create notes in other places in the code besides in this method? In this case, all the places where “new Note( )” appear must be rewritten with an “if” statement. This proposal is not very elegant or scalable.

This is a recurring problem in software design: creating objects inherently defines a strong coupling between the creator object and the created object. A set of solutions to this problem is expressed in what is known as *creational patterns* that describe ways to create objects at the same time, as the coupling is kept low. The *Abstract Factory* is one of these patterns, and it provides a very high degree of flexibility and control over the object-creation process.

The basic idea in Abstract Factory is to delegate the responsibility of creation to a special object with the sole responsibility of creating objects. Such an object is called a *factory*. (This is the motto of most design patterns: delegate the responsibility to another object.) To use an Abstract Factory, our code from before is changed to the following:

```
void insertNote(Staff *staff, int atStaffDegree) {
    Note *newNote = factory->createNote();
    staff->insertNote( newNote, atStaffDegree );
}
```

The factory object is an instance variable in the surrounding object's scope. The factory is defined by an abstract class:

```
class Factory {
public:
    Note *createNote(void) = 0;
}
```

We can now create any number of classes that implement the creational method: one for ordinary notes, one for drum notes, grace notes, and any future symbols that we may invent:

```
Note *DrumNoteFactory::createNote(void) {
    return new DrumNote();
}
```

What are the benefits and liabilities of this approach? The primary benefit is that the implementation of `insertNote` has been coded and tested once and for all. No matter what symbols we may define in the future, we do this by defining new factories—not by changing existing code. Adding code is much more manageable and less error-prone than changing existing code. For instance, if notes were created in 150 different places in the source, adding a new note type would require that all 150 places where the “new” operator is written be revisited, correctly understood, and changed.

A second important benefit is that the factory object becomes the central hub for creating notes, and thus global changes can be made simply by changing the concrete factory. Thus, to make drum notes, we just call a method that changes which factory to use:

```
void changeToDrumNotation() {
    factory = drumNoteFactory;
}

void changeToNormalNotation() {
    factory = ordinaryNoteFactory;
}
```

Thus, when any of these methods are invoked, any future notes created are guaranteed to be of the proper type.

What are the liabilities? The complexity of the design is greater than the first proposals, because we have added the factory classes to the design. Additionally, the programmers must be familiar with the abstract factory design pattern to keep their programming in line with the intentions of the pattern. For instance, if the programmers had created notes using the “new” operator directly in the previously written code, then the program would have a defect that might be difficult to locate. But this is a general liability of all design patterns: programmers must know and understand them in order to use them.

One may also argue that performance suffers, as we have to execute one extra method every time we create a note. The answer, however, is a bit more complex than that, because we also eliminate the need for the conditional in the code. If we really had 150 places in the code where we could avoid introducing a conditional, then the resulting code size might in fact benefit from using factories for note creation instead.

## The Strategy Pattern

Duration is a fundamental aspect of musical graphics, but is handled in several different ways in standard notational practice. Consider the variety of graphical representations to denote duration: note heads, dots, flags, beams, etc. Short-duration notes use flags attached to the note head using a stem. A derived problem here is that stems can be attached both upward and downward from the note head. On closer inspection, the rules for determining correct stem direction are quite complex. For a note on a single-voice staff, the rule is often that the stem should point upward if the note head is placed above the middle line and downward otherwise. If the note is part of a group, then the stem should point upward if the majority of notes have their stems up, and down otherwise. But again, other rules apply if the stem is attached to a beam or if the note is part of a polyphonic staff. And, as the fi-

---

nal complication, the editor may decide to break all the usual rules.

The question is then, “Who should decide in which direction to draw the stem?” Or, in more object-oriented terminology, we might ask, “Which object has the responsibility of knowing the stem’s direction?” In our editor software, the note head object is responsible for drawing the stem. However, the decision about which direction to choose depends on many other objects in the system: the chord to which it belongs, whether a beam or a flag is attached to the stem, whether the staff is a single-voice staff—not to mention the whims of the editor.

This complexity has two problematic consequences. First, a complex algorithm will have many branches (e.g., “if part of a chord then . . . else if stem attached to beam . . . else. . .,” etc.). Second, the note-head object must know all the relevant objects: chord object, beam object, staff object, editor preferences, etc. This would mean yet another example of high coupling: every object depends on all other objects in the system, and such a mess is known to be very difficult to maintain.

At this point, the strategy pattern comes to the rescue. The solution is that the note object simply delegates to a special strategy object, a “stem-direction-deciding object,” to choose which way to draw the stem. This frees the note object from deciding how to draw the stem. The note only maintains a pointer to the strategy object. It reflects neither the complex rules nor the references to chords, staff, beams, etc. This approach greatly simplifies the code of the note head.

The draw method now looks like the following:

```
void Note head::draw() {
    // draw the note head itself
    if ( stemDirectionStrategy->direction() == stemUp) {
        // draw the stem upward
    } else {
        // draw the stem downward
    }
}
```

The `stemDirectionStrategy` object implements a simple interface with only one, pure virtual method:

```
enum StemDirection { stemUp, stemDown };
class DirectionStrategy {
public:
    virtual StemDirection direction() == 0;
}
```

For each of the general rules, there is a subclass to implement this method. For instance, there is a subclass called “`SingleVoiceSingleNoteDirectionStrategy`” that calculates the direction for a single note (not part of a chord) in a single-voice staff. It embodies the algorithm that returns “`stemUp`” or “`stemDown`” depending on whether the note head is above the middle line or not:

```
StemDirection
SingleVoiceSingleNoteDirectionStrategy::direction() {
    if ( [associated note head is below middle line
in staff] ) {
        return stemUp;
    } else {
        return stemDown;
    }
}
```

Note that this approach provides a number of specific strategy objects, each of which is relatively simple and embodies just one of the rules. Thus, the complex code is broken into natural chunks that are highly focused and easy to understand. They are also easy to change without inducing major side effects in the surrounding system.

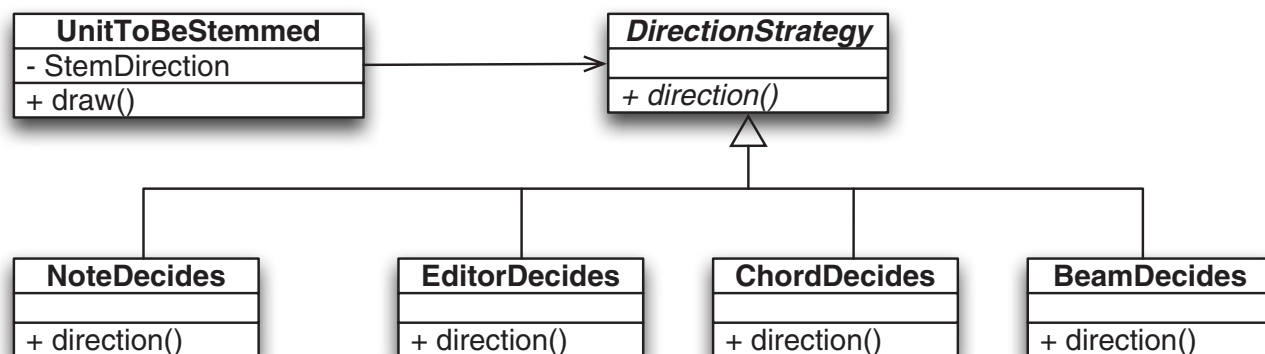
Of course, we do not avoid the decision of which strategy should be assigned for each note head. But this is very simple and localized wherever there is a need to change the strategy, for instance, in the command code associated with a menu allowing the editor to overrule the present strategy:

```
// editor has overruled
note head->SetNewDirectionStrategy
( editorDecidesStemStrategy );
```

or when a note is added to a chord:

```
// added to a chord
note head->SetNewDirectionStrategy
( chordDecidesStemStrategy );
```

Figure 5. Four alternatives to the question of who decides the stem direction of a note.



A design using the strategy pattern is flexible, as it is easy to add new strategies later or even at run-time: one simply assigns a new strategy object to the note head, and there is neither the need to re-inspect note-head code nor make any changes that are liable to introduce new defects.

The liability of the pattern is increased complexity in the object model: there are more classes to overview in the software, and at run-time, there is an increased number of objects to manage. The class diagram for strategy could look like the following for a single-voice staff, as shown in Figure 5. The general diagram of a Strategy Pattern is shown in Figure 6.

The Strategy Pattern allows us to think similarly in cases not concerned with stems. The general idea is to encapsulate the facts that we need different variants of an algorithm and that we do not want every object to “know” about the complexities involved. The Strategy Pattern thus hides the complexity of deciding the correct stem direction from the user/programmer, and everything can be decided when a note is created. The responsibility of decision is delegated to the part of the program that actually knows about the context.

### Evaluating the Use of Patterns in the OMIS Editor

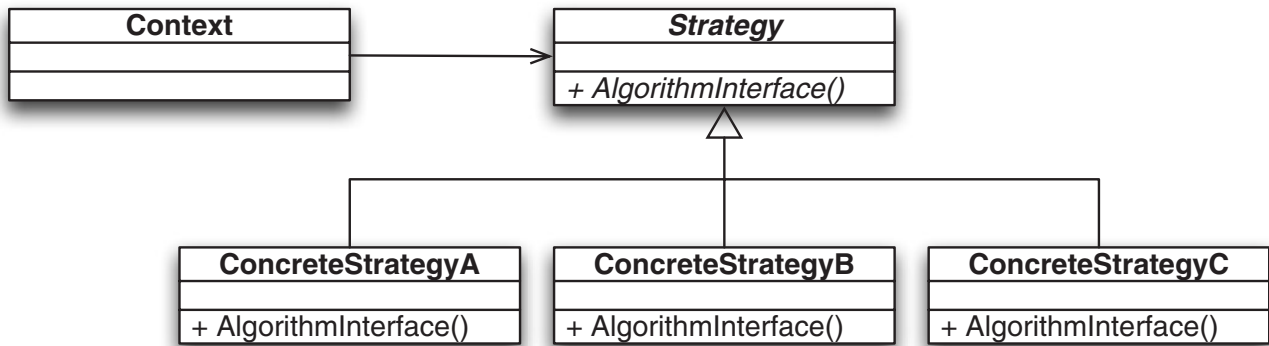
After examining the different ways we have used patterns in the design and implementation of OMIS, we will now reflect on the notion of quality again. The benefits of using design patterns do not originate from some intrinsic connection to music nota-

tion; they come from a more general perspective. The software simply becomes more maintainable, more structured, better working, but also somewhat more difficult to read for programmers with a background in procedural programming.

One of the explicit criteria for good design found in the design patterns philosophy is the ability to “Design for Change” (see chapter 1 of Gamma et al. 1995), which roughly translates to designing programs in which the structure is flexible and where the system can evolve gracefully over time. Software engineering is a discipline with much emphasis on cost, and experience has shown the phase after deployment to be very costly in terms of correcting errors, adding new requirements, and porting to ever-changing hardware platforms. Design patterns are a means to make software more easily adaptable and extensible in the face of these challenges—and thus minimize maintenance costs. Although OMIS has not yet officially shipped as of this writing, there have nevertheless been many requests for changes and additions from its users throughout its history. We have observed how the use of patterns in OMIS often facilitated addition of these new features with relative ease, as has also been reported by many other authors. The strategy pattern allows us to define new stem-strategy rules without changing existing code; the factory of note heads allows new notation styles to be introduced, and so on. The result is more robust code that is easier to read and maintain, for example, as we encounter additional notational needs.

Our use of patterns also underlines another cru-

Figure 6. General disposition of the Strategy Pattern.



cial aspect of patterns: the use of patterns as means of communication among developers. By learning names and contents of the patterns, developers essentially expand their design vocabulary. We were able to discuss various design strategies without spending time on clarifying the details of our ideas. In short, design patterns define a common vocabulary of well-tested design proposals at a much more powerful level of abstraction than individual program-code statements. This has greatly sped up our discussions, fuelled our considerations of alternatives, helped us in seeing the “big picture,” and provided a strong basis for evaluating and choosing what turned out to be superior designs.

Beck and Johnson (1994) report that “patterns generate architecture.” We have observed this quality of patterns in two ways. First, as mentioned above, patterns have improved communication among programmers. Second, design patterns used properly often weave themselves into complex and flexible architectures. One challenge was dealing with the complex semantics of moving musical notation as outlined previously in the observer pattern section. However, the real complexity is much higher. What if several windows show the same section of the score when the note is moved? How do we choose which aspects of the score to manipulate: notes as a whole, or “micro-adjustments” of parts of a note, lyrics, etc.?

Here, we took inspiration from work on two-dimensional semantic editors, HotDraw and later JHotDraw, that were originally developed as studies in design-pattern usage. The pattern density of this

architecture is high but has served both as inspiration for how to make patterns work together as well as a concrete starting point for the underlying architecture in OMIS. In both aspects, this has been advantageous in making the underlying architecture stable and has served us well through all the iterations and additions to OMIS.

Design patterns have not provided solutions to every problem we have faced in the OMIS design. Patterns are additions to a software engineer’s design toolbox; they do not replace existing techniques. One of the major challenges has been the complex semantics of moving musical notation, as has been touched upon several times in this article. The Observer Pattern handles localized changes well (e.g., moving a single note on the staff). However, it does not provide an adequate solution for large-scale changes, like moving whole sections from one page to the next. It is simply not feasible that every part of the score “listens” to changes from every other part and adjusts their position accordingly. As outlined in the observer pattern section, we here found a traditional procedural-processing scheme to be superior.

## Acknowledgments

We thank professor Finn Egeland Hansen, Aalborg University, Denmark, for being the patient end-user representative. We also thank students Henrik Lynbech and Jesper D. Thomsen, both from the Computer Science Department at Aarhus Univer-

---

sity, for skillful implementations and delightful design sessions.

## References

- Alexander, C. 1964. *Notes on the Synthesis of Form*. Cambridge, Massachusetts: Harvard University Press.
- Alexander, C. 1979. *The Timeless Way of Building*. Oxford: Oxford University Press.
- Alexander, C., et al. 1975. *The Oregon Experiment*. Oxford: Oxford University Press.
- Alexander, C., S. Ishikawa, and M. Silverstein. 1977. *A Pattern Language: Towns, Buildings, Construction*. Oxford: Oxford University Press.
- Beck, K., and W. Cunningham. 1987. "Using Patterns Languages for Object-Oriented Programs." In *Proceedings of the 1987 Conference on Object-Oriented Programming, Systems, Languages, and Applications*. New York: Association for Computing Machinery.
- Beck, K., and R. Johnson. 1994. "Patterns Generate Architecture." In *Proceedings of the 1994 European Conference on Object-Oriented Programming*. Kaiserslautern, Germany: Association Internationale pour les Technologies Objets.
- Gamma, E., et al. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston: Addison-Wesley.
- Gerou, T., and L. Lusk. 1996. *Essential Dictionary of Music Notation*. Los Angeles: Alfred Publishing.
- Rising, L. 2001. *The Pattern Almanac 2000*. Boston: Addison-Wesley.
- Ross, T. 1970. *The Art of Music Engraving and Processing*. Miami Beach, Florida: Hansen Books.