

# From Kandinsky to Java

## (The Use of 20<sup>th</sup> Century Abstract Art in Learning Programming)

Colin B. Price

Computing Section, Business School, University of Worcester,  
Worcester, WR2 6AJ, UK. c.price@worc.ac.uk

**Abstract:** At the University of Worcester we are continually striving to find new approaches to the learning and teaching of programming, to improve the quality of learning and the student experience. Over the past three years we have used the contexts of robotics, computer games, and most recently a study of Abstract Art to this end. This paper discusses our motivation for using Abstract Art as a context, details our principles and methodology, and reports on an evaluation of the student experience. Our basic tenet is that one can view the works of artists such as Kandinsky, Klee and Malevich as Object-Oriented (OO) constructions. Discussion of these works can therefore be used to introduce OO principles, to explore the meaning of classes, methods and attributes and finally to synthesize new works of art through Java code. This research has been conducted during delivery of an “Advanced OOP (Java)” programming module at final-year Undergraduate level, and during a Masters’ OO-Programming (Java) module. This allows a comparative evaluation of novice and experienced programmers’ learning. In this paper, we identify several instructional factors which emerge from our approach, and reflect upon the associated pedagogy. A Catalogue of ArtApplets is provided at the associated web-site.

**Keywords:** abstract art; Java; object oriented programming; teaching programming

## 1. Introduction

There has been much discussion on how to teach programming; typical issues concern which *paradigm* to use (e.g., declarative, functional, object-oriented), which *language* to use (e.g., C, Scheme, Java) and of course the instructional methodology. See for example the report of the Joint Task Force on Computing Curricula (JTFCC 2001) and Proulx et al. (2002). Irrespective of the chosen path within this “Garden of Forking Paths”, we must surely accept that software development is essentially a *creative* process, constructing of an artefact which does not yet exist. Many commercial software applications are highly *visual*, so it seems also appropriate to encourage students to produce visual programs. But where can we find a theoretical and practical basis to inform a creative and visual approach? We suggest that such a basis can be found in a study of Art, Architecture and Design, domains which are fundamentally *creative* and *visual*.

The subject of this paper is to explore how a study of Art, in particular the 20<sup>th</sup>-Century abstract artists such as Kandinsky, Klee, Mondriaan, Rothko and Newman, may be used to produce effective learning and efficient teaching materials to be used in programming classes. We suggest that there are useful synergies between Art and Programming which may be used to this end, and that there is a clear “arrow of information” constructively pointing from Art to Programming. It is interesting to note that that Richard Gabriel, Distinguished Engineer at Sun Microsystems, is also an advocate of the Fine Arts informing software development; he has proposed the development of a “Masters of Fine Arts in Software”, (Gabriel 2002). The work at MIT’s Media Lab is also relevant to this discussion, especially the “processing” project (Processing, 2007) and also the work of John Maeda (2004). Research undertaken during the academic year 2006-7 has involved working with two groups of students. The first, a final year BSc. Computing class, had prior exposure to OOP (Java). The second, an MSc. Computing class met OOP and Java for the first time. The use of (almost) identical materials with both classes allows a comparative evaluation of the approach.

A critical reflection on working with these students yields the following factors which are discussed in this paper: (i) The process of viewing and analyzing an artwork into its components, such as simple abstract shapes of tone and color, establishes a classification of components and the corresponding OOP Java classes, (ii) Task-based learning, where students work autonomously through online tutorials with tutor support, is an efficient and effective pedagogical strategy, (iii) An integration of learning about objects and about classical program constructs avoids the “objects first?” question, (iv) The exclusive use of visual and interactive tasks provides a rapid learning experience, (v) “Reading Code” activities help students create a code *Gestalt*, an understanding of how a whole program works, (vi) An “Object Oriented” (OO) approach to instruction (discussed below) leads to efficient teaching and efficacious learning.

This paper is structured as follows. In section 2 we shall tease out some useful correspondences between Art and Programming. Section 3 presents details of a task-based learning approach. Section 4 considers pedagogical issues and in Section 5 we develop the notion of a correspondence between Art and Programming. Section 6 provides an evaluation of the approach. We shall strive to show that there are synergies between Art and Computing, and that these synergies are *constructive*, and that an “arrow of information” from Art and Design into Software Development should be acknowledged as being useful in an educational context. Web-pages containing some ArtApplets demonstrating ideas discussed in this paper are available at the author’s website (Price 2007).

## **2. THE NATURE OF PROGRAMMING, THE NATURE OF ART**

When learning how to program, students should be confronted by just that, and no more. The complex task of programming should be situated in a simple context. There should be no distractions, students should learn in a “minimalist cognitive zone”. For example, while learning a programming *language* students should not also be asked to learn an *algorithm*, nor should they be expected to write some code for a highly *abstract* application, such as a database. It is surely more useful to place learning in a simple, uncluttered, visual, non-abstract context where there are no distractions from the learning of the language (and its principles). The study of abstract art provides an accessible visual context to establish such a minimalist cognitive zone.

Consider the painting shown in Fig.1. Despite its simplicity, Malevich’s *Self-Portrait* is a complex canvas of interacting objects, but the painting as a whole can only be understood as a *Gestalt*. This is also true of a computer program, which consists of a complex interacting system of objects and their methods. Ultimately, students must learn how to understand the whole program, to see the *Gestalt*. We have often found, that while students show their understanding of OO principles and program constructs (demonstrated by their ability to write them), understanding the “whole” is another matter. This is especially true of Java syntax. The directly perceived *Gestalt* of the artwork under discussion may well help the construction of the corresponding code *Gestalt* in the mind of the student.

A second correspondence can be distilled from a consideration of Art Education. A crucial part of Art education is the study of great Masters, through a study of their paintings, and through an appreciation of their lives and social context. The “programming” parallel is easy to see; we ask our students to “read code”. They are taken to a room without computers and provided with code written by the tutor (“Master”) on sheets of A3 paper. The class begins with a discussion of objects’

attributes and methods and inheritance, and then moves on to mapping out the method calls (messages) and, in particular how the Java syntax supports this. Through these “reading code” activities we have found that our students, as well as clearing-up issues of syntax and developing confidence in OO principles, develop a *Gestalt*; they move towards understanding the code in its entirety. They are effectively building up mental models of the entire code. The use of a paper-based activity rather than a computer-based IDE seems to be crucial to the success of this activity.

Why do we create programs? What are programs? How should we best create programs? These are in effect *metaphysical* questions, however reality returns when the “subject benchmark statement” is embraced (QAA, 2006). This highlights programming as a core element of the computing curriculum, as does Denning in his report on the “discipline” of computing (Denning 2000). But what about OOP, is this a language or a paradigm? We view OOP as primarily a methodology of thinking and not a paradigm of programming; we emphasise the cognitive aspect (Soloway 1986). This methodology *induces* from the special to the general. In our context, a discussion of a number of Artists’ paintings leads to the classification of objects within that painting. When these classes are established, instances may be created in code, and placed on the Canvas; this is a process of *deduction*. These processes of *induction* and *deduction*, the reasoning processes of ordinary life, provide a powerful methodology for understanding programming.

The learning tasks involve writing graphical programs; the “console” is only used for student-elicited debugging messages. The rationale for using an interactive graphical approach to learning and teaching programming is not difficult to support: A graphical visualisation of the students’ modification of code is rapid and information-rich. Students obtain immediate feedback of the consequences of their coding. But also, in this context of an inductive-deductive relationship with Art, students are motivated to be creative. This approach is in line with Stein’s “Interactive Programming” project where computing is seen as interaction and not calculation (Stein 2003). In a typical task (discussed below) students may be asked to make an interesting composition of abstract shapes. They may experiment with methods, parameters and attributes and obtain rapid feedback of the effects of their coding. Their code becomes alive and plays a constructivist role. A traditional programming course may output quantities as a series of numbers on the console. These numbers require a further level of interpretation by the students who must think about the code, the numbers and the algorithm. As a consequence they do not obtain immediate intuitive feedback of their coding; they must work simultaneously with two mental models, the program and the algorithm, and of course interpret the “experimental data”. This does not create a minimalist cognitive zone.

The question remains how to implement an interactive graphical learning environment? Some educators support the use of specially crafted graphical libraries to hide away the “complexities” of Java AWT and Swing, such as the “Nice Graphics Package” developed at Brown University. We choose not to do this, but to present the AWT and Swing packages as a learning experience for our students. They “drill-down” into the Java API class tree, discovering the location of attributes and methods and the structures of inheritance. This process is akin to “studying the Masters” and surely aids the learning of OO principles?

Ours is certainly not the first project to argue a connection between Art and programming. But previous work has emphasised the use of programs to generate Art, e.g., the *Processing* project at MIT and the work of John Maeda (2004). We refer to these approaches, which in effect produce visualizations of algorithms, as belonging to a “School of Algorithmic Art”. Algorithmic Art is like an arrow flying from programming to art, producing art by programming (algorithms), such as those classic visualizations of Mandelbrot and Julia sets, or Reaction-Diffusion textures used by Witkin and Kass (1991). Recent more subtle attempts by Maeda have not really shifted the paradigm. A good analysis of the methodologies associated with the construction of Algorithmic Art is provided by the artist Roman Verostko (1990). We suggest that this arrow may be reversed, and propose that Art may act as a constructive *metaphor* for OOP: Looking at an Artwork involves categorization, discovering base classes and dependencies, inheritance. It expresses relationships and suggests methods and parameters.

### **3. TASK-BASED LEARNING**

Our students work autonomously through a series of tutor-produced tasks, with the tutor providing short inputs to keep the class synchronized, and to answer particular problems. The tutor is freed-up to provide appropriate scaffolding assistance to individual students. This is important, since our students have diverse programming backgrounds. The first session consists of an introduction to Object-Oriented Analysis (OOA) and OOP with discussion centred on several artworks. It soon becomes clear that there is more than one way to “classify” a picture, so the whole class negotiates a set of classes which provides enough expressive power. These classes are then coded by the tutor and act as base classes which all students will then use in their tasks and projects. In their tasks, students extend these base classes, add attributes, add or overload methods. No student will ever produce a complete program from scratch. While some may object to this, we argue it is standard professional practice. In the educational context, it leads to large efficiencies; the tutor has a common code framework in his mind to help each student. Students can read each other’s code and provide peer reviews. We refer to this as “Object Oriented

Instruction” where knowledge is systematically extended. This has pedagogical implications (see section 4 below).

The 12 module sessions are divided into 5 “Phase 1” activities where students worked exclusively on several on-line tasks with tutor input. The remaining “Phase 2” sessions are devoted to individual projects, which again used the negotiated base classes; here students develop creative code at their individual levels of attainment. In the first session of instruction, we looked at the work of Malevich, (e.g., Fig.1). A class discussion of several of his works soon led to the identification of similar, common elements. A classification of these elements was made, and so we converged to the notion of “Class” (elements which shared common “attributes” - such as form) and then the notion of “objects” as “instances” of classes, actually painted on the canvas was discussed. Suddenly, the need for a “Canvas” class emerged! We also discussed the behaviour of visitors to an Art gallery and their relationship to the Artworks. From this emerged the need for a “GUI” class which captures the interaction of the viewer with the canvas. The final base structure comprises (i) “Art Objects” drawn on the (ii) “Canvas”, which itself was viewed by (iii) the “visitor” via a GUI.



Figure 1. Malevich *Self Portrait*

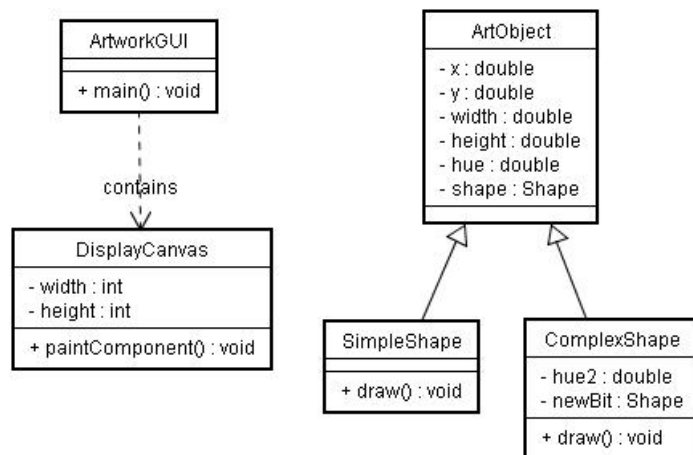


Figure 2. Classes and Dependencies

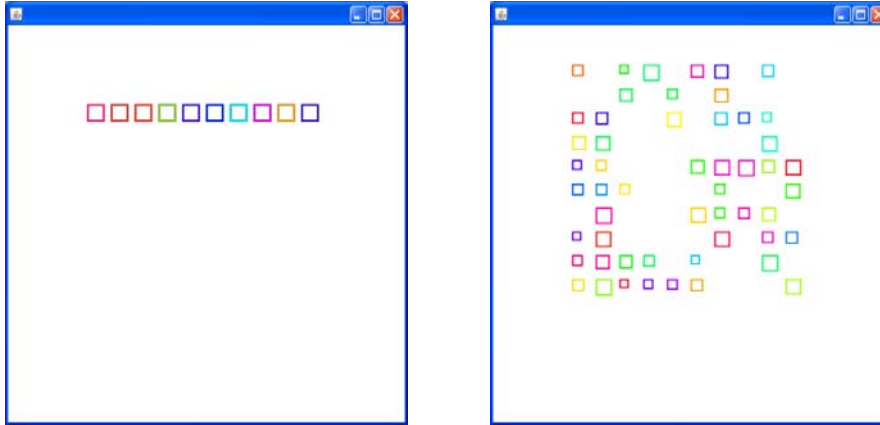
This generates three “base” classes: *ArtObject*, *DisplayCanvas*, and *ArtworkGUI*. The discussion then moved on to identify the “intrinsic qualities” of these classes (e.g., *width* and *height* of the *DisplayCanvas*) and so to the concept of attributes emerged. There followed a discussion of methods e.g., where the should the “draw()” method be located? An important conclusion was drawn, the draw() method should reside within the *ArtObject* class; each element should be responsible for drawing itself. Revisiting the classification of *ArtObjects* soon led to the notion of a

hierarchy, i.e., sub-classes and inheritance. In initial tasks students instantiated several *ArtObjects* explicitly in the *DisplayCanvas*, (later this would be done using arrays or Collections). Using the constructors, they explored the meaning of *parameters*, setting different locations, sizes and colors, whether the object was filled or outlined. Then they were invited to extend a *SimpleShape* into a *ComplexShape* discovering the meaning of extension, inheritance and overloading (via, e.g. the *draw()* method).

At this point the array data structure was introduced, as a means of collecting multiple canvas objects into a coherent whole. A 1D array produces a line of *ArtObjects* (see Fig.3(a)). Of course the loop program construct needs to be introduced simultaneously. Then the selection “if {} else{}” construct was introduced, to vary the color or the *ArtObjects* produced, or their sizes. The natural progression to a 2D array followed, with students experimenting with mathematical functions and conditionals to produce pleasing 2D canvases (Fig.3(b)). Finally we discussed the array, its limitations (fixed type, and fixed size), and the need for the *ArrayList* to hold various numbers of objects from different classes.

There has been much work to investigate the efficacy of an “objects-first” approach to Java programming (JTFCC 2001; Cooper et al. 1993). Our approach sidesteps this issue. We introduce OOP principles and programming constructs simultaneously, in the very first session. This is facilitated by the use of “base classes” which contain elements of both constructs and OOP principles.

Further activities within “Phase 1” introduce (i) the construction of GUIs using Swing, (ii) dynamic interacting *ArtObject* (“Interactive *ArtObjects*” *IAOs*), using Threads, (iii) applets to publish cool constructions on the Web. One session is devoted to the study of the game “Ping-Pong” where the concept of events, and the corresponding Java implementation is introduced. While other courses use “simplified” Java programming environments, where the complexity of event-handling is hidden under wrappers (Bruce et al. 2001), we intentionally use “raw” Java packages and classes *ab initio*. There is minimalism here, but not in the language; the locus of minimalism is shifted to the structure of the learning context which comprises “tasks” designed to both guide students and help them to become autonomous learners. We suggest that it is most efficient and honest to deal with the “real” Java programming environment at any stage in CS education, and not to hide the complexities of the API behind a curtain which necessitates an addition level of interpretation by the student. Again, this is the notion of “reading the Masters!”



**Figure 3.** (a) Left shows the depiction of a 1D array with random association of hue, (b) Right shows depiction of 2D array with selection structure driven by random numbers

“Phase 2” activities are designed first to review those technicalities of Java used in Phase 1 (such as Swing events, loading of Files or Images, etc.). Second to introduce new Java technology, such as RMI and JMF. This material is presented “formally”, through short tutor inputs supported by on-line activities which students may choose to explore if they feel it is appropriate to their projects. Phase 2 is very much directed by the students’ needs in developing their final projects. The pedagogy is simple: The aim is to provide a diversity of materials and approaches to support individual learners, but also through class discussion, to identify those technical areas of common usefulness. Here, there is a combination of autonomous, peer-interactive and tutor-informed learning.

#### 4. PEDAGOGY

Details of our general pedagogical approach have been given elsewhere (Price 2006). Our fundamental idea is to provide a *rich* learning environment which may support many different learning styles, especially *active learning* where each student constructs their own knowledge and understanding. This process is *iterative*, as noted by Soloway who suggests that an educational environment should (i) propose several solutions to a problem, (ii) provide several alternative decompositions of a problem through exploration and experimentation, (iii) produce a synthesis of a solution through analysis of these alternatives, (iv) allow an implementation of the solution in software, and (v) encourage reflection on this process (Soloway 1986; Soloway et al 1998). Such a rich environment should to support students with *diverse backgrounds* of programming competence. The mechanics of the approach embodies Vygotsky’s notion of a “zone of proximal development” where each student may advance in learning with appropriate “scaffolding” support from the tutor (Vygotsky 1978). We refer to our approach as “Educational Darwinism”; within a rich learning environment, students may find their individual “niches” and so flourish.

To model the actual iterative dynamic process of learning, we were influenced by a number of approaches, especially Kolb's "learning cycle" (Kolb 1983) which models learning as an interactive loop from (i) experimentation (ii) to obtain a concrete experience, (iii) and reflection on that experience and finally (iv) to abstraction. We implemented this cycle by requiring the students to maintain a learning journal, where at the end of each set of tasks ("activity") they were invited to reflect upon the results of their investigative work. Kolb's abstraction stage was obtained via class discussion. Our assessment methodology was designed to support this pedagogy. Students were invited to produce an assessed "portfolio" of work, including a learning journal. This facilitated the meta-cognitive dimension, and was much appreciated especially by the MSc. students. The portfolio was structured to include hand-drawn sketches, screen-shots of students' program output, reflections on the relationship with artworks, annotated code, discussion of problems and solutions, and evaluation of individual student targets and their success.

We are also appreciative of the psychological theory of Ausubel who emphasises that for students to integrate new knowledge into existing knowledge structures (i.e., to learn), then those structures must exist beforehand (Ausubel et a. 1978). Ausubel stresses the importance of prior knowledge. We suggest the use of Art as a metaphor may help to "bootstrap" the learning process.

We also reflected upon the debate over the location of *convergent* and *divergent* thinking. Some research aligns divergent thinking with creativity and convergent thinking with intelligence, though there is still much debate here (Fryer 1996). A combination of OOA, through analysis of art (convergent) and OOP-based creative production of novel electronic art (divergent) caters for both models of the student mind. During the MSc. module, it also became clear that some students were engaging at a meta-cognitive level; see Flavell (1976) for a definition. The idea is that to be an effective learner one must be aware of one's own progress in the learning process. This meta-cognitive level emerged spontaneously through a student-initiated discussion of the suitability of the Art-based approach to learn how to program in Java.

A consideration of art education also informed our pedagogy. Consider the Art and Design schools of the early 20<sup>th</sup> century, especially the Bauhaus and the Russian Constructivists. They strove with common tensions between the need for a skills-based education and an academic discipline, a product of the socio-political climate of that epoch. Computing education in HE finds itself today torn by similar tensions, where is a need to ensure employability of graduates, and a need to be academic. To be academic is to encourage and to be informed by research, leading to the creation of new knowledge.

## 5. A CONFLUENCE OF UNDERSTANDING

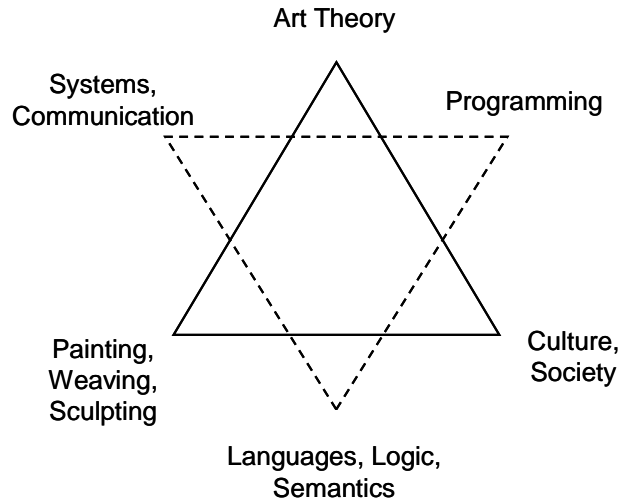
As artists, we paint, as programmers, we code. While these are two different media of expression, they share one goal, to produce new artefacts. Consider the Genetic Algorithm (GA) paradigm, a well known technique in Artificial Intelligence. Applied in our context, GAs step outside the action of “painting” and raise the question “how to paint”. GAs produce a “family” of Artworks, each unique but sharing familiar aspects, *concepts* of Art. What are these “concepts” which are embodied in the GA and other code? These are defined by the established artist, following years of experimentation and discovery. Their realization as *software Art* is nothing more than a realization of the artists’ own *concepts*. For example, Klee has taught us to pay attention between the *controlled* and the *uncontrolled* (Klee 1992), a dialectic between the specification of the geometrical and the provisional. In one software ArtApplet a juxtaposition between programmed composition and user interaction is explored: Rectangular *fields* define localized dynamics of the IAOs. The programmer-artist instantiates IAOs onto the canvas which then interact with these fields and a dynamic *emerges*. The computer “controlled” (the code) is juxtaposed with the artist’s “uncontrolled” (the composition). The concept of *fields* of activity on the canvas was motivated by Piet Mondriaan’s concept of “dynamic equilibrium”, which is based on dialectic between opposites and equilibrium. Mondriaan, Klee, Malevich and Kandinsky provided theory and examples to inform the production of several “ArtApplets” to explore this concept of visual dialectic.

The Artist’s expression of free aesthetic conception is realized through brush strokes and color mix on a physical canvas. This is constrained by the physical reality (the physical texture of the brush, the mixing of colors and generation of the optical percept), but also by the physics of the real world. Consider the work of Jackson Pollock where gravity colludes with aesthetics to create an Artwork. Can this be programmed, parameterized? This is a simple interactive simulation task, using the laws of physics to simulate Pollock’s dripping paint. The notion of “forces” between elements of a painting is well known in art theory (Arnheim 1954). In addition to the laws of *Gestalt* we have also introduced laws of conservation, familiar in the physical sciences. Some ArtApplets conserve total tone or hue across all objects in the canvas, or a sum of orientation, or the degree of order and disorder. The underlying Java code solves a system of N second-order differential equations, simulating the physics of the real world. This has been informed by the work of Schmidt, Klee and Itten, (see Itten 1975), and is implemented by solving the appropriate Laplace or Poisson partial differential equations, see the on-line catalogue (Price 2007).

Since we are proposing a cross-disciplinary approach, it important to establish a working theory of how these disciplines may interact. Our current thinking is summarized in the diagram Fig.5 which presents three fundamental dimensions of computing; (i) Programming (“Software

Development”), (ii) Systems and communication, (iii) Languages, logic and semantics (“Computer Science”). They are juxtaposed with corresponding dimensions of art : (i) Painting, weaving, sculpting, (ii) Culture and society, (iii) Art theory. The north-south axis corresponds to “thinking”, the north-east south-west axis to “creating” and the north-west south-east to “discussing”. Let us explain: Contemporary wisdom surely places the human individual and societies of individuals central to the definition of the “Computing Project”. Advances in internet communication, information and e-commerce technologies are profound. The impact on our culture and society is reflected by technological developments in systems and communication. This “NW-SE” axis reflects human activities of on-line shopping, chat and information gathering. The evolution of computer languages has grappled with linguistic notions of syntax and semantics. This is reflected in the development of Art Theory which establishes the “N-S” thinking axis. Both computer language and art theory underpin the creative realization of new products. The practice of programming is creative; software engineering is based upon something called “Computer Science”. This is juxtaposed with the Creative Arts, painting, weaving, sculpting, based upon Natural Sciences, and so the “NE-SW” axis is established.

Contemporary computer software applications present the user with a visual *representation* of numerical or abstract information to support interpretation and interaction. This is not unlike the viewing of a painting, textile or sculpture. Just as the artist moves from abstract ideas and concepts to a real physical work of art, so the programmer is challenged to effect a representation of scientific or business information in a semantically meaningful rich visual medium. More recently, distributed computing has taken inspiration from sociology (Watts 1999). Advances in computer processing power have led to developments in computer-generated art (Sims 1994). While such art clearly uses computing as a *tool*, there have also been attempts to forge a closer brotherhood between art and computing. The Aesthetic Computing project of Paul Fishwick renders mathematical equations into art objects situated within colourful, dynamic virtual worlds which represent the solution of the equations in an appealing visual experience (Fishwick 2002). At MIT John Maeda revisits traditional visual communication design where e.g. computing can provide self-modifying typography (Maeda 2007). Both of these projects start from a well-defined mathematical or scientific principle, and construct a visual *metaphor* of that principle. The MIT “Processing” environment explicitly encourages artists to write programs! In all of these examples, the intellectual arrow points clearly from computing to art. As mentioned above, this arrow may be usefully inverted.



**Figure 4.** Working model of the relationship between Art and Computing, juxtaposing similar concepts or processes.

## 6. EVALUATION AND CONCLUSIONS

Here we provide but a short overview of our evaluation methodologies and the results. Since the BSc. students had already been exposed to OOP and Java, a pre and post module Likert attitude test was used to determine the effectiveness of their learning. Various aspects of OOP were tested via questions, such as “How well do you understand *encapsulation*”, presented at the start and the end of the module. While this test indicated progression, one clear area of concern emerged; should one sub-class or use an additional parameter in an existing class? The MSc. students were evaluated via a short paper where they were invited to discuss the usefulness of OOP concepts. These papers were ranked by the tutor, and a set of criteria developed for assessing understanding of particular OOP concepts and. Re-reading the papers and applying these criteria led to grading which hopefully reflected understanding of OOP concepts. Here one clear issue emerged, concerning encapsulation. Students suggested that the encapsulation of attributes seemed draconian; direct access to an attribute (when made “public”) often made more sense than the need for a “get()” method when that attribute was made “private”.

It is also interesting to consider anecdotal evaluations of our approach. Following an end-of-module evaluation, (conducted via a paper-based questionnaire), the following criticisms and praises were consistently noted by students. (i) Some students had difficulty in understanding the mechanics of the *ArrayList* collection, in particular the *iterator*. (ii) While “message-passing”, through method calls, apparently was understood, a complex sequence of calls was not appreciated; students could not easily synthesise their own sequence of calls. This problem

centred on the need for passing a reference to calling and called objects. As mentioned above, a “reading code” activity significantly helped here. (iii) Students were unanimous in their fear of Java error and warning messages, and also of the API which required a significant investment of time, despite tutor support.

Points of praise fortunately outweighed these criticisms: (i) Students unanimously reported that their understanding of arrays and loops, (especially nested loops) was facilitated by the visual approach. (ii) The use of parameters to change the attributes of objects was appreciated. Given a task of producing an “appealing” Artwork, students experimented (and produced good code) using loops, conditionals and parameters. (iii) The concept of dynamics as implemented through *Threads* was appreciated, students reporting that the ease of coding an immediate visualization of moving objects was beneficial to understanding Thread functionality.

When specifically asked to comment on the use of Art as a substrate for learning OOP and Java, students’ comments were interesting. (i) Several had delved into the details of Java2D to produce interesting dynamical graphical works. (ii) Others reported that analysis of abstract paintings, and the resulting synthesis of their own OOP-Java realizations provided an interesting and worthwhile experience. (iii) One student reported a real appreciation of the connection between programming and Fine Art, and a desire to take this further into research.

Concerning the assessment approach, all students appreciated the “portfolio” method which allowed them to produce work and to have it reviewed at regular intervals. This seemed to resolve issues of “time-management” which causes problems even for level-3 and MSc. Students! All were unanimous that learning programming was not easy, but very satisfying. They suggested that learning programming is a *complex* activity; so much must be learned at once; concepts, syntax, development tools (the IDE), modelling tools and engagement with the Java API documentation! Our MSc. students appreciated discussing these issues, especially the need for a concept of *Gestalt*, and found the approached help support the development of this concept.

The negative response of our MSc. students in researching UML as a useful design tool was disappointing. They seemed to have become immersed in the technical details of creating working code in Java, focussing on the language rather than on the paradigm. Clearly students have a need for a focus in a programming module, or as suggested above, a minimalist zone.

Concerning the autonomous learning of our students, we report positive results. Some students engaged with the Art context and immersed themselves in the study of 20<sup>th</sup>-Century art. One MSc. student has opted to research this approach as a topic of his dissertation. Another student

investigated collision detection algorithms, to produce a realistic visualisation of fluid flow. Two students became engaged in the details of Java2D, one chose to investigate imaging, (transformations, rendering), the other the use of primitive elements to produce an Applet to investigate mathematical functions! All students seemed to have flourished in their Darwinian niche!

One mature student, a professional Visual Basic programmer contributed at a meta-cognitive level to an evaluation of the learning experience. His evaluation endorses the use of a minimalist cognitive zone, the use of JCreator rather than “drag and drop” IDE, e.g., JBuilder. This student, who in his professional activity used a VB drag-and-drop approach, embraced the minimalist Java learning structure with enthusiasm. He recounted that the approach, while initially unfamiliar and taxing, was extremely constructive. He suggested that this minimalist approach gave him deeper insight into the use of these IDEs. He later constructed an industrial Java application, involving the use of RFID (“radio frequency identification”) technology and Java-coded databases. His evaluation was simple: Use of drag-and-drop Java IDE should follow the minimalist coding approach.

In conclusion, this study has been a motivated attempt to cross two disciplines. The results are encouraging. Each student has been able to identify a particular area of interest, and have learned in their zone. On exit they have learned both a lot of Java and the principles of OOP: In their projects all students used encapsulation, inheritance, overloading; they coded with Threads and made use of Swing and its associated event structure. The numbers involved were small; we expect around 20 students on the BSc. module and 6 or 7 on the MSc. All students produced graphic and dynamic pieces of software of high quality, and asserted their individual preferences. This is laudable. Yet, unfortunately, no single student of programming has become an artist! Thomas Aquinas defined “beauty” as “harmony, unity and radiance” (Aquinas 1981). We hope that in asking our students to create beautiful ArtApplets, we also encourage them to write beautiful code. Aquinas’ definition is particularly appropriate to help us reflect on how to approach learning and teaching of computer programming, one of our most fundamental and creative contemporary human activities.

## 7. REFERENCES

- Aquinas, T., (1981), *Summa Theologica*, (transl.) Christian Classics
- Arnheim, R., (1954), *Art and Visual Perception, a psychology of the creative eye*. Faber and Faber, London
- Ausubel, D., Novak, J., and Hanesian, (1978), *Educational Psychology: A Cognitive View*. Holt Rinehart and Winston, New York.
- Bruce, K.B., Danlijik, A.P., Murtagh, T.P., (2001), Event-driven Programming can be simple enough for CS1. *SIGCSE Bulletin* 33 (3)
- Cooper, S., Dann, W., Pausch, R., (2003), Teaching Objects-first in Introductory Computer Science. *Proceedings of the 34<sup>th</sup> SIGCSE technical symposium on computer science education (Reno, Nevada)*, pp.191-195
- Denning, P.J. (2000) *Computer Science: The Discipline* in Encyclopaedia of Computer Science, Ralston, A., and Hemmendinger. D (eds.) George Mason University, Fairfax VA.
- Fishwick, P., (2002) Aesthetic Programming: Crafting Personalized Software, *Leonardo*, 35 (4) August. pp. 383-390
- Flavell, J.H., (1976), Metacognitive aspects of problem solving. In Resnick, L.B., (Ed.), *The Nature of Intelligence*, Lawrence Erlbaum, Hillsdale NJ.
- Fryer, M., (1996) *Creative teaching and Learning*, Paul Chapman, London.
- Gabriel, R., (2002), *Dreamsongs*. <http://www.dreamsongs.com/Projects.html> (accessed May 2007)
- Green, T.R.G. & Petre, M (1996) Usability analysis of visual programming environments: a 'cognitive dimensions' framework. *Journal of Visual Languages and Computing*, 7 (3) pp. 131-174.
- Itten, J., (1975) *Design and Form. The Basic Course at the Bauhaus and Later*. Revised Edition, John Wiley & Sons Inc. and Thames and Hudson Ltd., London.
- JTFCC (2001), Joint Task Force on Computing Curricula. "Computing Curricula 2001, Computer Science". *Journal of Educational Resources in Computing* 1(3) .
- Klee, P., Spiller, J (ed.) (1992) *Paul Klee Notebooks Vol. 1: The Thinking Eye* William Stout, San Francisco CA.
- Kolb, D.A., (1983) *Experiential Learning. Experiences as the source of Learning and Development*. Prentice-Hall.
- Maeda, J., (2004), *Creative Code*. Thames & Hudson, London.
- Maeda, J., (2007) Aesthetics and Computation Group. <http://acg.media.mit.edu> (accessed May 2007)
- Processing* (2007) <http://processing.org/> (accessed May 2007).
- Proulx, V., Raab, T., & Rasala, R., (2002), *Objects from the beginning – with GUIs*. Proceedings of the 7<sup>th</sup> Annual Conference on Innovation and Technology (Arhus, Denmark), pp. 65-69.

Price, C.B., (2006) *Learning Java, Learning Game Programming*. Proceeding of JICC10, London Metropolitan University, Jan 2006.

Price., C.B (2007) Supporting Web-Pages available at [http://www.worc.ac.uk/departs/bm\\_it/colin/Resources/ITALICS/Catalogue.htm](http://www.worc.ac.uk/departs/bm_it/colin/Resources/ITALICS/Catalogue.htm) (accessed May 2007)

QAA, (2007) *Subject Benchmark Statements Computing* (QAA 170 03/07) <http://www.qaa.ac.uk/academicinfrastructure/benchmark/statements/computing07.asp> (accessed May 2007)

Sims, K., (1994), *Evolving Virtual Creatures*. Computer Graphics, *Siggraph '94* Proceedings July 1994.

Soloway, E., (1986), "*Learning to Program Learning to Construct Mechanisms and Explanations*," Communications of the ACM, 29 (9), pp. 850-858.

Soloway, E., Spohrer, J., Littman, D., (1988), "*E unum pluribus: Generating alternative designs*", in Teaching and Learning Computer Programming: Multiple Research Perspectives, R. E. Mayer (ed.), Lawrence Erlbaum Associates.

Stein, L.A., (2003), Interactive Programming in Java. Available online at <http://www.cs101.org/ipij> (accessed May 2007)

Verostko, M., (1988) Epigenetic Painting. Software As Genotype, A New Dimension of Art. 1<sup>st</sup> Symposium on Electronic Art (FISEA) Utrecht. Published in *Leonardo* 23 (1)

Vygotsky, L.S., (1978), *Mind and Society*, Harvard University Press, Cambridge, MA.

Watts, D. J., (1999), *Networks, Dynamics and the Small-World Phenomenon*. American Journal of Sociology, 105 (2).

Witkin, A., Kaas, M., (1991), *Reaction-Diffusion Textures*, Computer Graphics 25(3).