TU Delft

Delft University of Technology

# Code as Art - Art as Code: On the Use of Poetry and Paintings in Programming Education

Hermans, Felienne

**Publication date**
2017

**Document Version**
Final published version

**Published in**
Proceedings of the 28th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2017, Delft, The Netherlands, July 1-3, 2017

**Citation (APA)**
Hermans, F. (2017). Code as Art - Art as Code: On the Use of Poetry and Paintings in Programming Education. In *Proceedings of the 28th Annual Workshop of the Psychology of Programming Interest Group, PPIG 2017, Delft, The Netherlands, July 1-3, 2017* http://ppig.org/library/paper/code-art-art-code-use-poetry-and-paintings-programming-education

**Important note**
To cite this publication, please use the final published version (if applicable).
Please check the document version above.

**Copyright**
Other than for strictly personal use, it is not permitted to download, forward or distribute the text or part of it, without the consent of the author(s) and/or copyright holder(s), unless the work is under an open content license such as Creative Commons.

**Takedown policy**
Please contact us and provide details if you believe this document breaches copyrights.
We will remove access to the work immediately and investigate your claim.

*This work is downloaded from Delft University of Technology.*
*For technical reasons the number of authors shown on this cover page is limited to a maximum of 10.*

# Code as Art - Art as Code: On the Use of Poetry and Paintings in Programming Education

**Felienne Hermans**
Delft University of Technology
Software Engineering Research Group
f.f.j.hermans@tudelft.nl

## Abstract

Programming is often taught by having students do practical programming exercises. From simple string reversal to search tree, the examples and methods of teaching mimic the life of a professional programmer in a sense. This leads to young children developing an idea of what programming is. We found that children under 12 already have clear preconceptions of what programming is for.

Can we design educational materials to battle this notion? Can we teach programming by using less traditional forms or viewing programming? In this paper we describe a four part course called Code as Art - Art as Code. It uses poems and paintings to teach novices and experienced programmers to see source code differently. In the first two lessons, participants practice viewing source code as a poem or as a painting (Code as Art). In the other two, they use source code to generate poems and paintings (Art as Code). We describe the scientific and creative rationale behind both and describe our experiences in teaching each of the four parts.

## 1. Introduction

Programming education is in fashion: in many countries around the world, programming is mandatory now in the UK, Australia and the US. This is of course, wonderful news for the lovers of programming! In the future, more children will be able to enjoy our field of profession. However, while teaching children programming, we also teach them what programming is used for and useful for. For example, it seems that the choice of example to use for programming lessons is often an afterthought, rather than a deliberate choice. Code.org for example, while a lovely platform used by many children (they report over 280 million children served) mainly consists of lessons where children create games of control robots. You could argue that is a relatively narrow view of what programming can do.

This narrow view is not without direct consequences for children learning to program. On two separate occasions we gave a lecture for about 70 children, aged 8 to 12. In the first lecture, we just asked the children to name one thing you can create with software. The answers were quite shocking! About 80% of children named robots, web sites or games. Only a few more creative answers were given, like 'everything' or 'cake'. In the second lecture, the results were similar, but we added a second question in which we asked children to be as creative as possible. Some did come up with many great things in the second question, like pets, an operating system, or a new nature to replace the existing nature when we as humanity have destroyed it. However, somewhat more realistic answers like art or music were not given. This made us wonder if it is possible to teach programming in different ways? Can we help children and adults think about what source code is in a different way? And can we design educational materials to diminish the impact of the traditional examples of programming like robots and games?

We have therefor developed a four part course called Code as Art - Art as Code, in which we help novices and experienced programmers to integrate their views of programming and art in order to broaden existing views. The course consists of four different lessons, which can be taught separately, or as a whole. In the course poems and visual artworks are used as art forms in two different roles. In Code as Art, art is the lens through which participants view code, while in Art as Code art is the subject to create with code.

The course has two parts, each with two lessons, resulting in these four lessons:

1. Code as Art: Poems
2. Code as Art: Paintings
3. Art as Code: Poems
4. Art as Code: Paintings

In the lessons 1 and 2, together forming the Code as Art part, participants are encouraged to view source code not as code, but as an artwork. This means its features can be studies, like we study art. How does this line of code sound? Do these lines rhyme? These are questions answered in lesson 1. In lesson 2 participants focus on the metaphor of visual art rather than poetry, and think about questions like How does the code look, and How does it make me feel?.

Lessons 3 and 4 as in a sense the opposite of lessons 1 and 2; we use code as a means of creating art. This is often called 'computational creativity' (Newell, Shaw, & Simon, 1959). Computational creativity and is a vibrant field where mainly artists and scientists participate. While many developers pride themselves on having 'pet projects' these usually do not include creating artworks with code. We would love for people and especially children to learn about the role of the computer as a creative partner, who can generate ideas, and with which one can even exchange ideas. Therefore both these lessons are aimed at novice programmers, but they could also very well be ran with professional developers.

Both the Code as Art and the Art as Code part start with poems[1], since their textual form is somewhat are similar to source code, resulting in the fact that some concepts like metrum and rhyme are easily transferred. Visual art is a nice subsequently deepening step, since it is more traditionally seen as 'art' and different art styles like surrealism or cubism are relatively well known among developers and novice programmers.

All four parts could be taught separately, as a whole of four, or in combinations on topic, such as only lessons 1 and 3, forming Code as Art - Art as Code: Poems. Over the past year, we have experimented with different formats, combinations and audiences, which we will describe in this paper. Overall we have found that professional programmers enjoy viewing their work in a different way, and that novices are surprised and excited by the idea of creating art.

## 2. Lesson 1: Code as Art: Poetry

### 2.1. Setup

In the first lesson, participants focus on elements of poetry that can be recognized in source code. The lesson consists of two parts that are traditionally seen as the building blocks of poetry: rhyme and rhythm.

#### 2.1.1. Rhyme

The rhyme exercise is concerned with the sounds of code. We ask participants to vocalise what a line of code sounds like? For some lines of code, determining the ending sound might be quite simple, like this one:

*Listing 1 – Sounds like: middle is length of m divided by two*

```
middle = len(m) // 2
```

But how does the line below sound? You could say it is 'm from middle', but you might as well say it sounds like 'm from the middle to the end', as that is the meaning of this line of code in Python. Or even 'take from m the items from middle to the end'.

[1] In earlier versions of the course, we only used paintings, but this proved to be quite challenging for some participants

*Listing 2 – Sounds like: ???*

```
middle = m[middle:]
```

*Listing 3 – Sounds like: zero point five times length of em*

```
middle = int(0.5 * len(m))
```

After just listing the sounds, participants are encouraged to also play with them. For example, the line from Figure 1 could also be written like in Listing 3, making it rhyme with other lines ending in 'em'.

#### 2.1.2. Rhythm

After the rhyme part, we move on to rhyme. In the first rhythm exercise, participants simply count the number of syllables of a given line of code. We asked them to add dots in between the syllables, and add the number as a comment so we could easily see and discuss the choices they made, as shown in the screen shot below.

```
def bub·ble·Sort·(a·list): #6
```

*Figure 1 – The six syllables of a line of code added as comment*

After simply observing the rhythm, participants also play with the metrum of a collection of lines. What can be changed to make a whole method have a certain rhythm? A simple exercise is to manipulate two lines of code so that they have the same number of syllables.

As an example, consider the two lines below:

*Listing 4 – Two lines with a different number of syllables*

```
alist[i] = min(alist[i], alist[i+1]) #13
alist[i+1] = max(alist[i], alist[i+1]) #15
```

By introducing an additional variable and initializing this variable to 0, the lines will consist of the same number of syllables:

*Listing 5 – An addition line followed by two lines with a the same number of syllables*

```
x = 0 #4
alist[i+x] = min(alist[i], alist[i+1]) #15
alist[i+1] = max(alist[i], alist[i+1]) #15
```

Both exercises together lead to some methods or algorithms with a nice metrum and rhyme.

### 2.2. Experiences

#### 2.2.1. Rhyme

This first lesson of the Code as Art, Art as Code course was ran at a conference for professional software developers in Norway, with about 20 attendees. In the first exercise, concerning rhyme, participants made interesting choices here and there. Some people annotated the sounds of lines of code quite literally, like this:

*Listing 6 – Literal sounds for lines of code*

```
def bubbleSort(alist): #list
    for passnum in range(len(alist)-1,0,-1): #one
        for i in range(passnum): #num
            if alist[i]>alist[i+1]: #one
                some = alist[i] #i
```

```
            alist[i] = alist[i+1] #one
            alist[i+1] = some #some
```

Others however pronounced list operations with the list name at the end, as seen in Listing 7. Appending an item *x* to a list *less* could be pronounced like 'add x to less' putting the list name at the end.

*Listing 7 – Comments rhymes with code*

```
if x < piv:            #piv
  less.append(x)       #less
if x == piv:           #piv
  equal.append(x)      #equal
if x > piv:            #piv
  greater.append(x)  #greater
```

When manipulating rhyme, some added comments to rhyme with lines of code, like this:

*Listing 8 – Comments adding to rhyme with lines of code*

```
if(you <= 2) { // you
  return 1;     // one
} //done?;      // one
```

In this case, the comment even has meaning in the algorithm, in addition to the right sound, as these lines of code check whether the recursion is finished, i.e. 'done'. Some people let themselves be inspired by keywords, for example naming a variable 'strength' to rhyme with length:

*Listing 9 – Variable names selected to rhyme with code*

```
 int len = data.length; #length
 int loop, i = 0;       #zero
 key = data[strength];  #strength
```

#### 2.2.2. Rhythm

In the second exercise, participants focus on the rhythm of code. In the first exercise, they just observed the metrum of a line of code. This seems like a futile exercise, but it revealed some interesting truths about source code. For example, how many syllables are in this line?

*Listing 10 – How many syllables are in this line of code?*

```
int x = 5
```

*int*, *x* and 5 clearly all contain just one syllable. But what about =? It that one, since it is *is*\\? Or two, since it is not really an *is*\\ but more of an *e·quals*\\? We found a number of interesting choices. Some preferred the simple *is* \\, others choose to vocalise this as *be·comes*\\, as the variable takes on the value of the following expression, while a third group thought *stores*\\ is more correctly representing what it means. In any case, this lead to some interesting discussions among the participants.

Another interesting fact was revealed when participants started to play with the numbers of syllables. That made them reflect on where variables were used, since a change of a variable name would impact the number of syllables of only the lines in which they were used.

All in all this course was received as very interesting and insightful by participants, since developers usually do not view source code in a vocal and poetic way.

## 3. Lesson 2: Code as Art: Paintings

The second art form explored in the Code as Art course is visual art. Programming as it is now is, in a sense, like realistic art, it tries to represent the world. This holds especially for programming education, in which often the real world is taught through modeling real world objects. What happens if we let this go? The metaphor of code as a painting is a rich one. Paintings can be beautiful or intentionally ugly. Modern art forms especially played with the notion of art itself, for example with Warhol creating Brillo boxes, an everyday utensils as an artwork. Could we do the same for code? Can we create code that is not representing the world in a realistic way? Can we envision cubistic or expressionistic source code? Can source code be liberated from the tyranny of usefulness or even executability?

### 3.1. Setup

In this second lesson of Code as Art, the goal is to view code as an artistic expression. Participants received a random painting or artwork from a preselection we made: cubism, surrealism, graffiti, De Stijl, art nouveau, expressionism, dadaism and rococo. Participants were then asked to work in pairs, and to select an sorting algorithm and recreate that in the style of their painting. The selection as well as the implementation of the algorithm could be inspired by the art style. For example insertion sort might fit a more realistic style painting as it is how people usually sort cards. Inspired by a certain brand of realism, this algorithm could be further adapted.

### 3.2. Experiences

This lesson was ran twice, each time with around 10 computer scientists and programmers. It was interesting to see what participants came up with. Some participants really were inspired by the philosophy of the art style. For example, one group received Dada, and created a sorting algorithm from random lines of code, since in Dada, there were no rules. Another group, receiving Rococo, placed it in the history of art after Barok, attempting to be simpler than what was before, but also still more involved than art that came later. They therefore selected shell sort, since it is more efficient than insertion sort, but not by a lot. Another Rococo group observed that Rococo is not efficient at all and thus devised a very inefficient algorithm based on bin tree sorting.

However, not all participants had a deep knowledge of art history, and this resulted in some groups to focus on the artwork we used as illustration of the concept, rather than on the idea of the art style itself. For example, the de Stijl painting we used was Victory Boogie Woogie by Piet Mondriaan (Figure 2) in which the canvas is divided in blocks. This lead participants to use Radex sort, in which the input list is divided into groups as well. While this of course is a fine interpretation, we would have prefer a more high level interpretation of the art works like the Dada and the Rococo groups desired above did, or maybe even more abstract.

For example the idea that surrealism aimed to free art from the dogma of the bourgeoisie and to allow more people to be artists. It pushed the boundaries of what art is and is not, and we are interested in hearing what sorting (or programming) is or is not.

A lesson we took from this for subsequent lessons is that programmers or computer scientists may have too little knowledge of art forms, and that courses in the future should be preceded with an introduction to art. A more viable approach might be to select one art style, explain that in more detail including its history, and then having all participants create an algorithm in that style[2].

[2] Yes, we would love to do this at PPIG 2017!

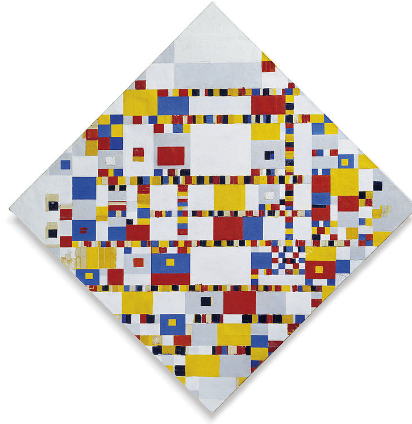

*Figure 2 – Victory Boogie Woogie - Piet Mondriaan*

Some participants however did use the idea of the art style as an inspiration, for example, one group that received rococo made the observation that rococo is the opposite of efficient, it uses a lot of fluff to deliver a message. They therefore created a sorting algorithm as inefficient as possible.

## 4. Lesson 3: Art as Code: Poetry

### 4.1. Setup

In the poetry course, we start with the generation of simple sentences. For this, only basic programming concepts such as lists and random numbers are needed. Figure 12 for example generates a noun and a random verb. These together can form a sentence by using two nouns with a verb in between, for example 'Polly desires cracker'. Note that there are words that will not lead to correct sentences, like 'must', given we create noun-verb-noun sentences.

*Listing 11 – Some random words to form sentences with*

```
function getRandomNoun(){
   var nouns = [ 'apple' ,'tree', 'laptop', 'Carl', 'Frances', 'house'];
   var index = Math.floor(Math.random() * nouns.length)
   return nouns[index];
}

function getRandomVerb(){
   var verbs = [ 'eats', 'wants', 'loves', 'desires', 'must', 'kills'];
   var index = Math.floor(Math.random() * verbs.length)
   return verbs[index];
}
```

Participants are then encouraged to change or add words and to play with the order to create better (or worse!) sentences. In a second exercise, conjugations are added to make more complex sentences by adding other word types like conjugations.

*Listing 12 – Adding conjugations to make more complex sentences*

```
function getRandomConjugation(){
   var conjugations = ['because' , 'unless', 'as', 'and', 'but', 'since'];
   var index = Math.floor(Math.random() * conjugations.length)
   return conjugations[index];
}
```

This resulted in funny, crazy or weird sentences, like "Carl must house since house eats tree".

### 4.2. Experiences

We ran this course two times. One with about 40 children aged 8 to 10. These children had been following our programming lessons for 4 weeks prior, so they had some experience with programming in Scratch. For that group we used Scratch. Secondly we ran it with about 100 teenagers in groups of 20, aged twelve to seventeen at a science festival for schools. Most of these teenagers had no prior programming experience, although some of them had used Scratch or Lego Mindstorms, or named Minecraft as programming experience.

Overall, we observed that children of both groups were very engaged with this exercise. Over 80% of the children in our Scratch named this lesson as their favorite of all Scratch lessons. The other lessons being more traditional Scratch lessons in which they created games and animations. The teenagers too loved the lesson, it seemed especially those that previously thought that they would not like programming.

There were interesting differences in approaches among both groups. Some children of both ages took the first sentence they got and started creating a poem of story with that. Other wanted to 'tweak' the programming, and added more words until it gave them a sentence they liked, without playing with the structure. A third group wanted to add more word types and sentences, learning quite some things about grammar too in the mean time.

We believe that the fact that this course supports many different paths to a success experience makes it an interesting first introduction to programming.

## 5. Lesson 4: Art as Code: Paintings

In the fourth lesson of the course, code is used to create Mondriaan artworks. This is another way of demonstrating to novice programmers the wide range of applications for programming. Mondriaan art is a nice topic, because it is relatively easy to create with source code but is still recognizable for children as 'real art'.

### 5.1. Setup

For this course the setting is a big lecture hall, where an instructor creates one program with input of the participants. We gave all children a green and red card which they could use to vote on questions like 'is this program correct?' or 'do you think this is the best way to program this'. With these questions as input, the presenter created one program while explaining the steps taken, in a form of *observational learning*, where a teacher demonstrates a task before learners attempt it (?, ?).

The course starts by asking children to draw a Mondriaan painting. Most children in the Netherlands are familiar with the work op Mondriaan and can create this with relative ease. In many schools around the country this is a common drawing exercise, especially in 2017, the Mondriaan year[3].

When they have drawn their Mondriaan, the next step is to compare their drawing to those of a neighbor and ask for one commonality and one difference. These factors will be input into our program. For example, if the two paintings differ in color, that should be a setting on the resulting program.

When the participants have created and compared their paintings, the programming starts. We first simply create a program that draws a line, and then a colored area too, as shown in Figure 3.

[3]http://www.destijlutrechtamersfoort.nl/en

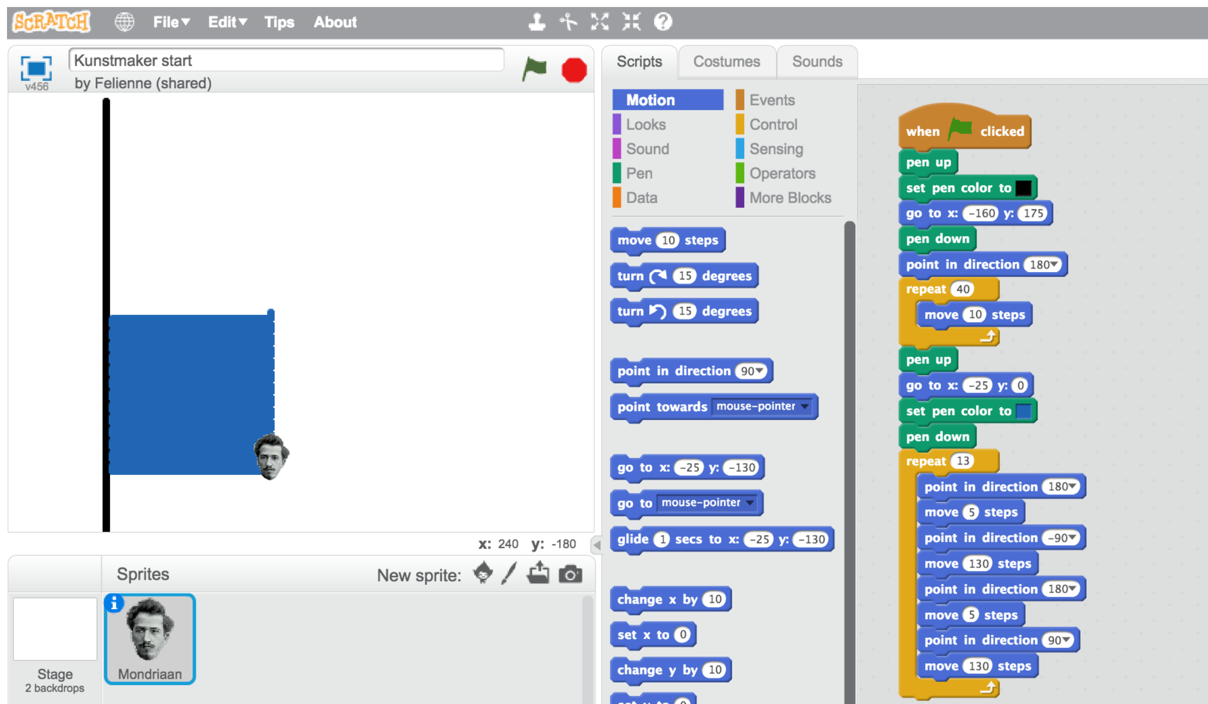


*Figure 3 – Creating a very simple Mondriaan*

This code is already a bit 'smelly', and the course leader can ask the audience if they agree, which they do in our experience. The code can then be put into functions for lines and areas, making it easier to generalize. A nice technique for this is to first make the code a bit worse, by creating a second line, as follows:

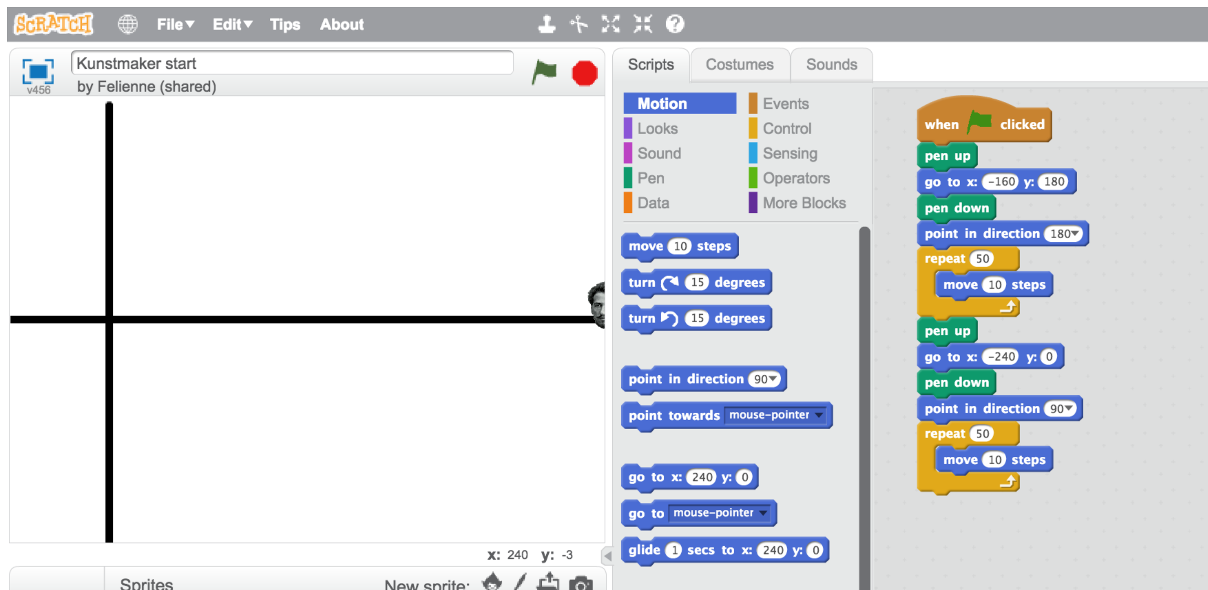


*Figure 4 – Creating a very simple Mondriaan, now with two lines*

The two pieces of code can now be placed side by side, as an illustration of their similarities and differences:

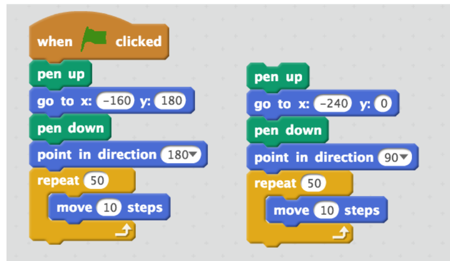


*Figure 5 – What is the same and what is different?*

It is fun to have the participants perform the same exercise on code as they did on their paintings, further stressing the fact that indeed code is like art and art is like code. After putting the line and the area into blocks, we brainstorm with the room on what variability is needed to create as many Mondriaans as possible. For example, what colors are needed, how many lines do the paintings have, and what should their direction be. In one of the courses a parent drew Victory Boogie Boogie(Figure 2), seriously complicating things.

While this course could also be done by participants individually, or in pairs, we opted for a plenary form, since it introduces high level concepts like functions which children aged 10 to 12 are usually not familiar with.

### 5.2. Experiences

This lesson two was ran two times, both times with about 70 children, once aged 11 to 12 with parents present and once with children aged 8 to 11 without parents. The lecture took a little but over an hour in both cases. We used the plenary method of teaching, starting with drawing their one Mondriaan and comparing them to the shared solution.

One observation is that some children were disengaged and even annoyed initially by the fact that they had to start by creating a drawing. “Why do we have to do this?” “This is supposed to be about programming!”. We promised them this was useful practice for programming skills, and they sort of went with it although some kept compacting a little bit. This seems proof that programming and making things on the computer are already very much tied to each other in young children’s brains. Interestingly enough, this was more common in the younger group without the parents.

When we pointed out later in the exercise that the drawing and the comparison with the drawing of neighboring children had been useful practice for creating the blocks, it made sense to many children.

At the end of the lecture, we succeeded in creating a Mondriaan painting, and, at the suggestion of one of the children, we even added some randomness to it to create new paintings.

After the lecture, we asked children after the lecture what stuck with them most, and the answers generally fell into two categories. One group of children stayed that the biggest take away was the idea that you van create something like any with programming. Other children were impressed by the notion of abstraction and the ability to create custom blocks.

## 6. Related work

Papert, one of the founders of programming education for children already argued programming is a creative endeavor (Papert, 1980). Papert states that the unique educational power of a computer is to support children in creating, exploring and experimenting, aspects of learning that are ignored in traditional education. While this work is almost forty years old, it seems to still be representative of modern day teaching, where most focus is on learning facts and techniques and not on learning though creating. Kafai and Peppler (Kafai & Peppler, 2012) described that games are used in education, but mainly to support traditional learning while they argue that game design can also contribute to creative and critical thinking in children.

More and more art shows and exhibitions are showing digital art, from computer generated paintings, to digital installations using projections. Peppler and Kafai note that professional artist are using technology increasingly, while children in art class rarely explore programming as a means of creating art (Kafai & Peppler, 2009). Romeike performed a literature review of introductory programming and found that creativity is rarely used in programming education, despite some authors describing promising results (Romeike, 2007).

Barnard argues that teaching programming and art together matters, since the process of creating an artwork is so similar to creating a program. Like the programming, an artist has a dialog with their work, which can only start when the real work has started (Barnard, 2015). In her thesis, van Groenestijn (Groenestijn, 2016) describes a method to integrate teaching programming and creativity, for example by having children create a digital tour of their school.

## 7. Concluding remarks

The world of programming is filled with programmers bringing their own ideas of what programming is. The ruling opinion seems to be that programs should have a purpose, such as apps, games and websites. We found that children as young as 10 already clearly have this bias, but many professional developers have too since they developer useful software all the time. To that end we have developed a course series Code as Art - Art as Code consisting of four lessons in two parts. In Code as Art, participants view source code as poems and paintings, observing the rhythm, rhyme and structure of lines of code. This part of the series was ran with professional developers and computer scientist, and we observed that they enjoyed a fresh way of looking at source code, and making decisions on he sounds of operators was even insightful. In Art as Code, participants use source code to create poems and paintings, so they learn to see programming as a broad tool, but we found that they also learned about language and abstraction in the lessons. In this paper we described sessions of Code as Art with professionals and Art as Code with children, however we think that the reverse should also be possible. Code as Art with professionals is no problem at all, but for 'Code as Art' some knowledge of programming is needed. One future research direction we envision is to explore how much programming experience is needed for this. Maybe starting with reading source code aloud is a nice way to start programming education, even if the source code makes no sense yet? After all, children also start to draw some letters before they can read.

In any case, we plan to continue this way of teaching to children as young as possible, so we can educate a new generation of creative programmers.

## 8. Acknowledgements

Thanks to the hundreds of participants that were willing to try out crazy things with me, including the Rainbow Group in Cambridge. A special shout out to the organizers of Boosterconf in Norway, who were the first ones to let me run Code as Art - Art as Code at a developers event.

## 9. References

Barnard, B. (2015). *De programmeur als kunstenaar.* Retrieved 04-30-2017, from `http://mandarin.nl/presentaties/nioc_lang.pdf` (in Dutch)

Groenestijn, S. v. (2016). *Van scratch tot kunst: van mediaconsument tot mediaproducent* (Unpublished master's thesis). Piet Zwart Institute, the Netherlands.

Kafai, Y. B., & Peppler, K. A. (2009). Creative coding: Programming for personal expression. In *The 8th international conference on computer supported collaborative learning (cscl)* (p. 76-78).

Kafai, Y. B., & Peppler, K. A. (2012). Developing gaming fluencies with scratch. In C. Steinkuehler, K. Squire, & S. Barab (Eds.), *Games, learning, and society: Learning and meaning in the digital age* (p. 355-380). Cambridge University Press. doi: 10.1017/CBO9781139031127.026

Newell, A., Shaw, J. C., & Simon, H. A. (1959). *The processes of creative thinking*. Rand Corporation Santa Monica, CA.

Papert, S. (1980). *Mindstorms: Children, computers, and powerful ideas*. New York, NY, USA: Basic

Books, Inc.

Romeike, R. (2007). Applying creativity in cs high school education: Criteria, teaching example and evaluation. In *Proceedings of the seventh baltic sea conference on computing education research - volume 88* (pp. 87–96). Darlinghurst, Australia, Australia: Australian Computer Society, Inc. Retrieved from http://dl.acm.org/citation.cfm?id=2449323.2449333