Styles de programmation et éthiques des programmeurs

Pierre Depaz - Universität Basel

Cette présentation, qui découle de mon travail de thèse, s'intéresse aux esthétique des codes sources, aux manières de programmer, et à ce que cela révèle des idéaux et assomptions des programmeurs.

Introduction

---

all the names of god, in Montfort, N. (2014). #! (Shebang). Denver : Counterpath.

```
{print"a"x++$...$"x$.,$,=_;redo}
```

```
_atk_atl_atm_atn_ato_atp_atq_atr_ats_att_atu_atv_atw_atx_aty_atz_a
```

*L'impossibilité de dire tous les noms de dieu. La poésie en code pose la question de **ce qui peut être exprimé**.*

*Un peu de contexte à mon approche des codes sources: j'ai commencé en sciences poliques, puis conception de jeu vidés, dans les deux cas, il s'agissait de **règles qui constituent** le monde. Pour ce qui est des jeux vidéos il s'agissait aussi de coder en permanence la présence d'une joueuse, et d'avoir toujours la notion implicite d'amusement (fun), deux notions qui ne sont pas uniquement computationnelles, et qui pose donc la question du rapport du code source avec des notions et des dimensions non-techniques.*

Plus généralement, aujourd'hui, un de mes axes de recherche est de savoir comment les logiciels vont exprimer le monde d'une certaine manière, et si cette expression peut avoir des conséquences sur l'agentivité des individus qui utilisent ces logiciels.

L'esthétique du code, une notion populaire mais pas explicitée[1].

*La question de l'expression amène alors dans un premier temps **la question de l'art**.*

C'est un thème qui revient souvent (Knuth, Dijkstra, Wirth), mais qui n'est jamais explicité ou traité de manière systématique. Par exemple, The Art of Computer Programming est un des membres du canon de la programmation, et en fin de compte, ça consiste en 3 pages (maximum) sur l'art, 2000+ pages sur l'implémentation d'algorithmes. Même si il va préciser la notion en 74, pour lui "Art" c'est au sens traditionnel de artes, l'artisanat, la technique. Moins sur l'expression d'idées.

Quand on analyse les discours en ligne des des programmeurs et programmeuses, on voit aussi qu'il y a un concept d'un code moche, d'une code beau, mais c'est difficile de le formaliser.

Il y a une sorte de consensus que ça existe, et différentes manières de faire, mais moi j'étais curieux de pourquoi est-ce qu'il faut que ce soit beau, et notamment parce que le médium est assez

spécifique.

Pourquoi avoir une opinion esthétique sur quelque chose qui est voué à disparaître?

*Le code source, c'est un objet **transitoire**, un peu comme un feuillet de notation musicale. Ce n'est qu'un support sur lequel on pose des idées pour que ces idées puissent être exécutées par autrui. Et donc on va tendre à juger la beauté sur le résultat, plutôt que sur l'intermédiaire. Le "beautiful sofware", c'est souvent des "beautiful interfaces" ou "beautiful functionalities".*

*La **fonction** pose aussi un petit problème. Le code source est éminenment fonctionnel (il est "destiné à").*

L'approche naïve de l'esthétique, c'est l'approche désintéressée: l'esthétique, c'est l'art, l'art c'est l'art pour l'art, autotélique. Alors que le code source a un but, une fonction très claire: traduire des idées pour qu'un ordinateur puisse agir sur le monde.

Mais, d'un autre côté, personne n'écrit de l'assembleur optimisé à l'instruction près, et les langages de programmation vont de plus en plus vers le haut niveau, dans un souci d'expressivité. On revient donc à ce problème d'expressivité, et pas uniquement de fonctionnalité brute.

L'ordinateur comme **organe prosthétique de philosophie**[234]

Une autre conception de la programmation, c'est de faciliter les manières de penser, en plus d'être déjà une manière d'agir.

"L'organe prosthétique de philosophie", selon la phrase de David Rokeby. En 1986, en développant Very Nervous System, une des premières oeuvres d'art interactives, il avait du se mettre à réfléchir à ce qu'est véritablement "un mouvement", lorsqu'il avait fallu le traduire pour un ordinateur.

Cette question de l'expression, on la retrouve à travers les travaux de nombreux informaticiens, notamment dans un contexte des rêves de l'ordinateur comme machine libératrice, ou dominante (Wolfram et son computational thinking, ou bien Sutherland et son augmentation de l'intellect). La computation facilite la pensée de chose complexes.

L'expression à l'épreuve de l'implémentation[5].

Ce problème de l'expression, qui se révèle lors de l'implémentation. On a une idée, et au moment de la rendre matérielle, explicite, celle-ci s'avère être plus floue que prévue

C'est la même chose pour l'art! L'externalisation de la pensée dans l'oeuvre d'art est aussi un problème de réalisation et de mise en forme, comme le note ***Nelson Goodman****. Et ensuite, le jugement des formes de l'oeuvre d'art est aussi lié au jugement de son contenu.*

Si le problème de l'expression se pose tant au niveau scientifique qu'au niveau artistique, c'est donc bien qu'il y a peut-être un continuum à travers ces domaines, du plus scientifique au plus artistiques (là encore, deux domaines qu'on peut naïvement penser comme opposés).

Le code source entre technique et société[6][7]

Au cours du processus de l'implémentation, le code, reste une des preuves les plus tangibles de ce que fait l'ordinateur, comme le racontait Lawrence Lessig dans son appréhension du code en tant que loi.

Mais de l'autre, il faut aussi se méfier du techno-déterminisme, du code qui serait exclusivement explicatif de l'action, comme le dit bien Wendy Chun. Il y a beaucoup de variables à l'implémentation d'une technologie: elle se casse, se détourne, ne fonctionne qu'à moitié, etc.

Mais toujours est-il que le parti pris ici est de considérer le code source comme étant une version idéalisée et réalisée en même temps. Idéalisée en ce qu'il manipule des concepts, et réalisée en ce qu'il contrôle des variations d'électricité.

Comment s'exprime-t-on dans les codes sources? Pourquoi?

Qu'est-ce qu'on juge dans ces manières de s'exprimer? Pourquoi?

Donc voilà les questions principales auxquelles on va s'atteler aujourd'hui: comment on écrit des codes sources, qu'est-ce que cela va exprimer, et pour qui?

Et puis, une fois qu'on aura élucidé la question de l'expression, on se penchera sur celle du jugement: qu'est-ce qui rend une représentation, un manière d'expression, bonne ou mauvaise?

---

1. Esthétiques du code
2. Jugements éthiques
3. Majeur et mineur

Donc pour considérer ces questions, on va d'abord s'intéresser aux différentes esthétiques du code, à quels idéaux elles répondent, et comment elles s'articulent autour de pratiques particulières, comprenant matières, outils et communauté.

Ensuite, une fois qu'on aura considéré la question du jugement esthétique, on établira ensuite un parallèle avec le jugement moral, en connectant donc le beau et le bon, l'esthétique et éthique.

Et donc on concluera sur l'éthique de la programmation, sur les manières de programmer qui sont considérées bonnes, et celles qui sont considérées mauvaises, et sur de potentielles alternatives.

Esthétiques du code

D'abord les métaphores et les champs lexicaux, pour ensuite se déplacer les styles de programmation et les différentes manières de faire.

Mais, avant qu'on s'y plonge, deux précisions définitionnelles.

L'expérience *esthétique* est différente de l'expérience *artistique*, et se décline dans une esthétique du quotidien[8][9][10].

L'expérience esthétique, c'est l'expérience qu'on va ressentir face à une manifestation sensible particulière. Elle est ancrée dans notre rapport à l'environnement, et elle se caractérise par un sentiment d'adéquation, de satisfaction, voire d'excellence.

L'expérience artistique va impliquer une expérience esthétique, voire la sublimer, on concentrant un désir d'expression délibéré et principal entre la personne qui crée et la personne qui reçoit. L'objet de l'expérience esthétique n'est pas forcément issu d'un acte de création dont le but premier est l'élicitation d'un sentiment esthétique (e.g. un coucher de soleil, des dossier bien organisés dans un ordinateur, une jolie mise en page d'un document, des fleurs sur le coin d'une table, etc.)

L'expérience esthétique a donc une valeur relativement plus faible que l'expérience artistique, mais elle a aussi une portée plus grande que l'expérience artistique. C'est ce que Yuriko Saito, Michel de Certeau, et Marielle Macé appellent les esthétiques du quotidien

Cela veut aussi dire que je ne pars pas d'une définition quantitative de l'esthétique (même si il y a eu des travaux pour essayer de "calculer" la beauté du code, par exemple la notion de complexité de Kolgorov).

Le jugement esthétique comme révélateur de l'environnement[11]:

- mental
- social
- matériel

Une exprérience esthétique implique un jugement, généralement pour attribuer le degré de plaisance. D'une part c'est clair qu'il y a un jugement, mais il est plus compliqué de dire quel genre de jugement il y a. Ce que disait Wittgenstein, c'est qu'il est compliqué, voire impossible, de faire une

telle analyse. Ce que je pense qu'il voulait probablement dire, c'est qu'il était impossible de faire un accompte détaillé et exact de l'expérience esthétique. Mais ça ne veut pas dire qu'on ne peut pas s'y pencher!

Alors, lorsqu'on s'y penche, on remarque qu'il peut agir comme un révélateur de l'environnement dans lequel s'effectue ce jugement. Et cet environnement va se constituer de plusieurs choses:

1. un environnement mental, qui est la particularité et la priorité de l'individu qui porte le jugement
2. un environnement social, qui est constitué des personnes, croyances et pratiques auquel l'individu appartient, et qui vont donc influer sur l'environnement mental.
3. l'environnement matériel, qui est constitué des matériaux et artefacts sur lesquels les humains vont agir, mais qui vont aussi supporter l'action et la représentation des humains.

Puisqu'une des manières dont on fait sens des choses, ce sont les métaphores.

Idéaux de programmation

Différentes métaphores pour faire sens du matériau[12].

Il n'y a pas qu'une seule esthétique du code, mais plusieurs conceptions qui se superposent et se traversent. Afin de les identifier, je propose de passer par une approche métaphorique.

Lakoff et Johnson ont montré que les métaphores qu'on utilise ne sont pas juste des figures de style, mais bien des manières fondamentales de comprendre le monde (vous avez peut-être remarqué que j'ai utilisé une métaphore précédemment: la figure de style, c'est une manière de dire que c'est un aspect remarquable, identifiable, du style, et on utilise le terme "figure" parce que la figure, c'est ce qui est le plus remarquable c'est l'humain).

J'identifie trois grandes différentes manières dont les programmeurs et programmeuses vont faire sens des codes sources: littérature, architecture et mathématiques.

Le code source comme **littérature**. [131415]

(textualité, communication, compression)

*La **littérature** pour refléter l'aspect textuel du code source (tant prose que poésie):*

- *c'est un texte, ça se lit, ça s'écrit, c'est l'argument principal du literate programming.*

- *ca explique, ça communique quelque chose d'une personne à une autre*
- *il peut y avoir différents niveaux de lecture, et il peut être relié à d'autres documents (notamment la documentation), comme intertextualitié et intratextualité*
- *la poésie pour établir un parallèle avec le désir d'exprimer le plus avec le moins de mots possible*

Le code source comme **architecture**.[16][17]

(construction, artisanat, structure)

*L'**architecture** sert à refléter l'aspect de construction du logiciel (qui va au-delà de l'écriture du code):*

- *planification de la construction: cathédrales et le bazaar*
- *attention au détail et notion d'artisanat, le rapport à un matériau et motifs de construction*
- *conception et implémentation d'espaces computationnels (jump in, jump over, jump out, walk, etc.)*

Le code source comme **mathématiques**. [18]

(logique, implémentation, objets computationnels)

*Les **mathématiques** comme expression d'idées et recherches de solutions:*

- *la beauté de l'entité mathématique (computationnelle)*
- *la beauté de l'implémentation (la belle preuve)*
- *la minimisation des entités notationnelles impliquées la maximisation des implications conceptuelles (ce qui rappelle la poésie)*
- *la beauté comme heuristique: quand on travaille sur une preuve, si l'expression commence à devenir jolie, on se dit qu'on est sur le bon chemin.*

Donc voilà pour les trois grandes manières dont les programmeurs font sens de ce qu'ils créent.

Avec deux choses à noter:

*Premièrement, ces catégories ne sont **pas mutuellement exclusives**, elles essaient de saisir différentes approches du code source, et donc peuvent aussi couvrir les mêmes aspects de différents*

points de vue. Par exemple, une esthétique de l'ingénieur va faire le pont entre architecture et mathématiques.

*Deuxièmement, on peut **noter des absences**, comme la musique, ou la peinture. C'est donc qu'il y au une certaine cohérence.*

Un champ lexical consistant:

- positif: clarté, simplicité, lisibilité, élégance
- négatif: boueux, puant, entremêlés.[19]

En passant au niveau plus micro, on retrouve des valeurs plus spécifiques.

En analysant le discours des programmeurs (qui vont néanmoins être auto-sélectionnés) de manière empirique, et en allant au-delà des références thématiques, et en s'intéressant au champ lexical spécifique, on remarque aussi une certaine consistance.

Soit ce sont des adjectif positifs: clarté, implicité, lisibilité, élégance, qui vont alors impliquer une sort de transparence, de possibilité de voir à travers le code, qui ne va pas attirer attention à soi, et qui souligne vraiment le rôle du code source comme interface entre l'humain et la machine.

Soit ce sont des adjectifs négatifs qui, eux, vont mettre en exergue la dimension matérielle concrète, et désordonnée ("messy") du code source, où on ne sait pas où ça commence et où ça finit, on s'emmêle les pinceaux, et on a un sentiment d'une mauvaise odeur quelque chose à laquelle notre instinct ne fait pas confiance.

L'esthétique pour gérer le logiciel comme **artefact abstrait**, [2021]

Cette notion d'artefact abstrait fait ressortir qu'un artefact est une création qui a un but fonctionnel, qui est destiné à accomplir quelque chose d'une certaine manière. On sait donc qu'il y a généralement quelque chose de précis à comprendre. Mais, d'un point de vue pratique, c'est rarement facile. Et dans tous les cas, l'esthétique est secondaire à la fonctionnalité. Un beau code source plein de bugs n'est pas un beau code source.

L'expérience esthétique affecte le fardeau cognitif associé à la compréhension.

Donc l'approche majoritaire de l'esthétique des codes sources, c'est:

- ***le désir d'une expérience esthétique est de soulager le fardeau cognitif de compréhension** qui est inhérent à la lecture du code.*
- *ce fardeau cognitif est présent parce qu'il est le résultat du déroulement des actions computationelles à la vitesse de la machine.*

*Le but principal du code source est donc de communiquer **et** une intention **et** une réalisation (le pourquoi et le comment). L'esthétique, c'est-à-dire la mise en forme plaisante, va faciliter (ou compliquer!) la tâche de compréhension, et donc proposer une stimulation cognitive qui va permettre de mieux comprendre ce à quoi on a à faire.*

Ou, inversement, de le compliquer à travers une création de puzzles (e.g. IOCCC)

Un même but (faciliter l'expression et la compréhension l'interaction entre le monde et la machine[22]), se décline de **différentes manières**.

Mais il y a une différence entre la fin (faire comprendre) et les moyens (comment faire comprendre). Et les programmeurs font comprendre de plusieurs manières.

Là on peut trouver un parallèle avec les traductions littéraires. Quand on traduite, il y a un seul texte source, et donc fonctionnellement un seul but: restituer correctement ce texte source. Mais il va y avoir des divergences en termes de comment restituer cette intention.

Cette question du comment nous amène donc à la question du style.

Styles de programmation

Ou comment le choix entre `tabs` et `spaces` peut prendre des airs de guerre de religions?

À première vue, le style est une question de goût, et de manière de faire. De choisir quelque chose qui plaît plus qu'une autre. On peut penser par exemple à la manière dont on organise les espaces, dont on écrit des variables, et dont on va appliquer ces décisions de manière consistante.

Et pourtant, les programmeurs et programmeuses qui en parlent le font avec beaucoup d'émotions et de vélléités. Ce n'est peut-être pas quelque chose d'uniquement superficiel.

La tension entre **individuel** et **collectif**, la proposition de se présenter au monde. [2324]

*Le style, c'est aussi une **tension**, on peut l'approcher de manière individuelle, ou de manière collective. L'approche littéraire va considérer le style autant comme expression individuelle (e.g. le style de Flaubert) que comme forme collective (e.g. le style épistolaire).*

De même pour les sciences sociales, où les styles individuels vont entrer en dialogue avec des styles collectifs (le style punk, le style bon-chic-bon-genre, etc.)

C'est une forme de détermination, et d'expression de soi, mais aussi une forme d'appartenance de groupe.

Le style individuel comme expression d'une préférence, et donc d'un jugement personnel.

Cette façon de se présenter, de présenter un objet qu'on a créé, va d'abord agir en tant qu'imposition d'une perspective à autrui. Cette expression individuelle propose délibérément une manière de voir le monde. Elle va proposer quelque chose de nouveau, de différent, qui va se targuer de contraster avec l'existant, et de se différencier. Il s'agit de mettre en valeur une autre manière de faire, et donc de proposer au monde une autre manière de juger une création.

Ce style-là, c'est particulièrement celui des artistes, donc, et ceux et celles qui écrivent des poèmes en code ne sont pas une exception.

Si Kafka écrivait du JavaScript[25]

```
function sayIt(firstWord) {
  var words = [];
  return (function sayIt(word) {
    if (!word) {
      try {
        return sayIt();
      } catch (e) {
        // quitting at last an unsettling recursion,
        // the array was trasformed into a monstrous string
        words = "there's been a hideous bug";
        return words;
      }
    } else {
      words.push(word);
      return sayIt;
    }
  })(firstWord);
}
```

Au fur et à mesure de l'expression d'un style individuel, lorsque celui-ci est consistant, il va finir par devenir structurant, il devient une manière d'organiser l'apparence et l'appartenance d'un groupe.

Le style sert de structure d'expression, une **structure partagée** qui devient vision de comment bien être, et comment bien faire.[26]

Cette manière d'organisation consistante, lorsqu'elle se transmet d'individu à individu, prend alors une dimension collective. Le style est une manière de former un groupe, autour de cette vision de la bonne manière de faire.

Cela a été mis à jour dans les sciences sociales, depuis Mauss jusqu'à Hebdige, notamment en montrant comment ces styles vont donc être corrélés avec les groupes d'individus qui les utilisent, et manifester les valeurs de ces groupes.

Cela pose aussi la notion d'envisager comment être dans le monde, et comment faire le monde, de préfigurer ce qui est désirable; ce qui se relie avec la façon dont la programmation va faire des mondes, au sens de Nelson Goodman.

Et pour ce qui est de la programmation, je remarque quatre grands groupes de styles:

Des communautés de pratiques stylistiques[27]:

- le style ingénieur
- le style scientifique
- le style *hacker*
- le style poète

Les ingénieurs:

- *documentation*
- *explicitation*
- *verbosité*
- *multi-vocalité*
- *langages "populaires" (impératifs, bas-niveau)*

Les scientifiques:

- *uniquement l'essentiel (souvent sans les détails d'implémentation de la vie réelle)*
- *code condensé, destiné à exprimer des idées*
- *mono-vocalité (souvent un seul auteur)*
- *langages spécifiques/un peu moins populaires (LISP, OCaml, Prolog, Julia)*

Le cas de l'usage de Python est intéressant puisqu'il représente un pont entre le style ingénieur et le style scientifique: les deux peuvent utiliser les mêmes outils et les mêmes techniques (notamment via des initiatives commes software carpentry, ce qui peut aussi poser la question plus générale de l'industrialisation des procédés scientifiques?)

Les hackers:

- *langage de la machine (dialogue avec la machine plutôt qu'avec l'humain)*
- *tous les moyens sont bons, mais principalement bas niveau*
- *à l'inverse des "bonnes" pratiques des ingénieurs*
- *le désir de se montrer plus malin qu'autrui, l'intelligence comme summum de la valeurs, aspect démiurge*

Les poètes

- *langage de l'humain (dialogue avec l'humain, et uniquement incidemment avec la machine)*

- *la compilation/interprétation comme strict mimimum (et donc un rapport distant, évasif à la fonctionnalité)*
- *l'attention à des concepts non-computationnels*
- *des langages relativement haut niveau (de script)*

Le style a comme contrainte l'outil (le langage de programmation).

```
import this
```

```
The Zen of Python, by Tim Peters

Beautiful is better than ugly.
Explicit is better than implicit.
Simple is better than complex.
Complex is better than complicated.
Flat is better than nested.
Sparse is better than dense.
Readability counts.
Special cases aren't special enough to break the rules.
Although practicality beats purity.
Errors should never pass silently.
Unless explicitly silenced.
In the face of ambiguity, refuse the temptation to guess.
There should be one-- and preferably only one --obvious way to
do it.
Although that way may not be obvious at first unless you're
Dutch.
Now is better than never.
Although never is often better than *right* now.
If the implementation is hard to explain, it's a bad idea.
If the implementation is easy to explain, it may be a good idea.
Namespaces are one honking great idea -- let's do more of those!
```

On ne crée pas à partir de rien. On l'a vu précédemment, ***la computation est un matériau bien particulier*** *puisqu'il s'agit d'idées réalisables, mais il y a aussi la question de comment on travaille ce matériau et, depuis l'apparition du système d'exploitation, le matériel n'est plus trop considéré pour les programmes qu'on écrit (léger bémol pour les cartes graphiques et les programmes en GLSL).*

*Ces outils, donc, ce sont les langages de programmation, et **les communautés linguistiques** qui se créent autour. En effet, les manières d'écrire sont souvent dictées par le langage qu'on va écrire, et les personnes qui écrivent ces langages se retrouvent dénommées via le langage (pythonistas, rubyists, gophers, rustlings, etc.).*

C'est donc quelque chose qui va bien influencer sur comment on écrit et donc sur ce qu'on écrit.

> *A language that doesn't affect the way you think about programming, is not worth knowing.*[28]

(est-ce applicable à la manière dont on considère le monde?)

Je ne vais pas développer sur les styles de programmation comme on l'entend traditionnellement (fonctionnel, orienté-objet, logique, impératif), mais si vous en discutez à la fin, je serais très curieux de savoir ce que vous en pensez!

Notamment en termes de ce que proposait Alan Perlis... Est-ce que ça peut s'étendre à la vision des problèmes à résoudre et, par extension, du monde?

Le style comme modalité *épistémologique*[2930].

Parce que les manières de faire peuvent influencer les façons dont on va s'atteler au problème. Ian Hacking et Gilles-Gaston Granger ont montré, à partir de l'histoire des sciences, que les manières de réfléchir, et les manières d'inscrire cette réflexion, vont proposer différentes approches du problème, et qu'ainsi il est possible de tirer différentes conclusions et modes d'actions sur ce problème.

De la même manière, on peut voir cela dans la façon dont la recherche en intelligence artificielle au milieu du XXe siècle s'est déclinée de deux manières: atomistique et continue, et qui a sous-tendue une façon de considérer l'intelligence humaine, et les manières d'implémenter ces conceptions, via des systèmes de règles ou des systèmes de pondération.

Différents styles épistémologiques auxquels on applique différentes valeurs[31]:

- le style **dur** de programmation
- le style **doux** de programmation

Ce sont deux approches que Papert et Turkle, un informaticien et une psychologue au MIT dans les années 1990, ont nommé "hard style of programming" et "soft style of programming".

Et ils caractérisent ces deux approches par des ensemble d'attributs, qui vont impliquer: - l'organisation du travail (le style dur préfère la pensée abstraite et la planification systématique; la pensée douce préfère une approche de négociation et des formes de raisonnement concrètes); - le genre de relation de l'individu avec les objets computationnels (le style dur préfère une position de distance, tandis que le style doux va préférer une proximité des objets) (par exemple, écrire une librarie soi-même, ou l'importer).

*Chacune de ces positions va considérer ces attributs comme des **vertus cognitives**, des choses qu'il est bon d'avoir et d'exercer afin d'évoluer au mieux dans le monde, et on retrouve dans cette dichotomie des échos de la différence entre une épistémologie de la méthode scientifique et de la méthode artistique.*

Papert et Turkle notent alors l'établissement implicite de la méthode dure, c'est-à-dire la réflexion formelle, abstraite, scientifique, comme étant supérieure à la méthode douce, c'est-à-dire la réflexion intuitive, relationnelle, bricoleuse.

Les esthétiques du code sont des styles de représentation de la fonction d'un artefact computationnel par rapport à la machine, au monde, à l'humain.

Cette présentation du code, et représentation du monde, se fait de plusieurs manières, qui vont au-delà du superficiel.

Il y a plusieurs manières d'écrire du code, mais elles ont toutes en commun le fait qu'elle vont résulter en une possibilité d'agir sur le monde (ne serait-ce qu'en déplaçant des pixels). La question est ensuite vraiment de savoir en quoi ces représentations différentes se manifestent en styles différents.

Accompagnant chacune de ces manières de présenter, il va y a voir un cadre de jugement pour les départager, afin de déterminer si ces manières de faire sont positives ou négatives, et transformant donc des valeurs esthétiques en des valeurs éthiques.

Jugements éthiques

Maintenant qu'on a vu les différents idéaux esthétiques, et les différents styles de programmation, il faut se pencher sur ce que révèlent les jugements esthétiques autour de ces codes sources.

Le beau et le bon

Il y a une relation entre esthétique et éthique, mais elle est difficile à qualifier précisément[32].

Si l'esthétique est l'étude de la perception sensorielle et de l'arrangement formel, l'éthique est l'étude de ce que l'on doit faire, de ce qui est correct de faire, et donc l'évaluation de la conduite, du caractère d'une personne.

Si on part de Platon, alors le beau et le bien sont ontologiquement équivalent. La belle conduite est vertueuse, et la conduite vertueuse est belle.

*Néanmoins, il y a énormément de contre-examples pour infirmer cette équivalence (comme l'architecture Nazie ou la littérature de Louis-Ferdinand Céline). Si il y a des débats sur la nature de la relation (est-ce que l'esthétique prime? est-ce que c'est l'éthique qui prime?), on peut au moins constater qu'il y a une relation, ce que Marcia Eaton qualifie d'**interdépendance conceptuelle**. Je pense, avec elle, que "les formes d'expression constituent une grammaire pour l'action et l'évaluation de l'action".*

> *In art and in human action there are expressive implicatures that allow speakers to project certain qualities of their own act as significant aspects of the message. We call attention to the way we speak as well as to what we say. We project purposiveness into the world, and thus contribute to the creation of a "public theater", where we act and react to constitutive acts.* ***Expressive patterns constitute a grammar for action and for evaluation of action.*** *Humans are not moved by better arguments, but by "more richly textured narratives".*

Et puis, il faut aussi prendre en compte de la confusion des intérêts d'un groupe dominant comme étant une vérité universelle, comme on le verra plus bas.

Ce qu'on peut voir, et ce qu'on doit cacher, est autant esthétique que politique[33].

Une de ces interdépendances, on peut le voir dans le travail de Jacques Rancière sur le partage du sensible. Ce que Rancière propose, c'est que ce qui est rendu visible est dicible, l'est aussi pour des raisons politiques. Il y a par exemple l'acte d'interpellation classique d'Althusser (un policier qui vous hèle vous rend visible comme sujet politique), ou encore les circulations d'images et de textes dans des environnements numériques, telle que l'invisibilisation des torses nus des femmes dûe à l'éthique puritaine ("cachez ce sein que je ne saurai voir"), ou la suppression de certains discours dans le cadre d'une régulation des échanges considérés toxiques.

La visibilité par défaut: les primitives et les *first-class citizens*[34].

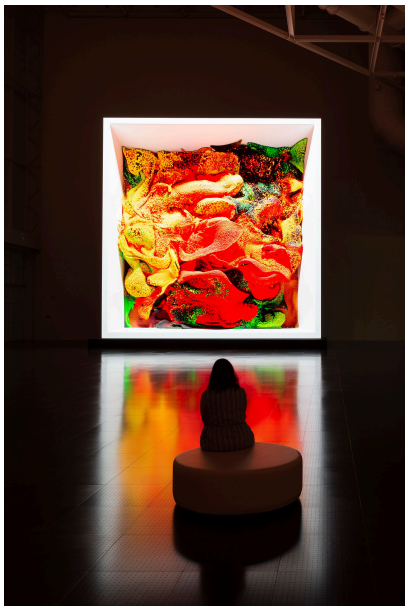
Cette notion de visibilité est aussi présente dans les codes sources, à travers les choix de design des langages de programmation.

Il y a aussi le cas de ce qu'on choisit comme primitives et comme "citoyens de première classe".

Les primitives ont une sorte de précédence ontologique dans les langages de programmation, et vont impliquer une relation hiérarchique, de ce qui est accessible, et donc de ce qui peut être rendu visible, par défaut. Cela veut aussi dire qu'il y a des choses plus fondamentales que d'autres. La métaphore elle-même des "first-class citizen" suggère bien un arrière-plan éthique à la préférence d'une entité par rapport à l'autre.

Par exemple, Go a `time.Duration` parce qu'il s'agit d'aller vite et de se synchroniser, et Kotlin a une proposition pour un type `Money` qui permettrait de gérer plus facilement des opérations financières. Et je ne serais pas forcément étonné de voir qu'un langage comme OCaml puisse inclure aussi des primitives liées à la finance, vu la place qu'a Jane Street dans son développement.

L'éthique peut guider l'esthétique, mais le jugement esthétique peut aussi révéler une éthique[35].



Refik Anadol - Machine Hallucination, 2022

Donc, naïvement, ce serait plutôt le critère éthique qui viendrait informer le critère esthétique (si ce n'est pas acceptable moralement, ce n'est pas acceptable esthétiquement).

Mais il est aussi possible d'inverser l'approche, et d'examiner le jugement esthétique afin de réveler un jugement éthique. Si on prend l'exemple d'oeuvres d'art, on peut penser aux travaux de Refik Anadol. Ce dernier crée et expose des visualisations numériques de grand format pour rendre perceptible le concept de l'apprentissage machine. Puisque cette perception est principalement sensuelle et non cognitive, on peut en déduire une conception de l'artiste et de l'oeuvre d'art comme étant principalement décorative plutôt qu'instructive, ou critique. La sur-présence de la surface par rapport au contenu valide cette superficialité.

C'est ce que Frederic Jameson développe lorsqu'il établit un lien entre l'esthétique post-moderne et mode de fonctionnement des individus dans le capitalisme tardif, notamment dans sa comparaison de travaux de Vincent Van Gogh et d'Andy Warhol.

Deux exemples de `linked list`[36]: une esthétique d'amateur et une esthétique d'expert.

```
void remove_cs101(list *l, list_item *target)
{
        list_item *cur = l->head, *prev = NULL;
        while (cur != target) {
                prev = cur;
                cur = cur->next;
        }
        if (prev)
                prev->next = cur->next;
        else
                l->head = cur->next;
}
```

```
void remove_elegant(list *l, list_item *target)
{
        list_item **p = &l->head;
        while (*p != target)
                p = &(*p)->next;
        *p = target->next;
}
```

on a là deux approches d'une même algorithme (enlever un élément d'une liste chaînée)

Dans le premier exemple, on va être plus explicite, et faire la différence dans les deux cas de figure (si on est à la fin de la liste ou pas), ça va être plus facile à comprendre pour un humain d'un niveau plus débutant.

En revanche, dans le deuxième exemple, on va réduire le nombre d'entités lexicales impliquées, mais à condition d'avoir un très haut de connaissance, et de pouvoir manipuler des pointeurs avec confiance.

On a donc deux choses qui se passent:

- *d'une part, ces deux manières d'écrire impliquent **deux conceptions de la liste**. la version basique, c'est que la liste est composée d'éléments, qui contiennent une valeur et un pointeur vers l'élément suivant. la version avancée (aussi appelée élégante), concoit la liste comme une série de pointeurs vers des éléments de la liste.*
- *d'autre part, cette complexité nous dit quelque chose du lectorat auquel on s'addresse, l'une ou l'autre manière de faire va être relativement mieux considérée suivant le lectorat qu'on valorise. au lieu d'appeler ces fonctions `remove_cs101` et `remove_elegant`, on aurait pu choisir d'autre termes (`remove_explicit` et `remove_implicit`).*

Et puis il faut enfin mentionner que cette manière élégante d'enlever un élément de la liste a été écrit par Linus Torvalds, qui n'est pas connu pour être la personne la plus patiente et tolérante de niveaux de compétences qui ne sont pas équivalents (égaux) au siens.

Les jugements des différentes manières d'écrire (du code) vont donc **révéler des préjugés et des attentes**.

Quelques instances

Maintenant, on peut repartir sur nos métaphores et nos styles de programmation (un style de hacker, un style d'architecte, un style de scientifique) pour voir comment ces styles de programmation peuvent aussi avoir une dimension éthique.

Le hack et l'optimisation des resources[37].

```
float Q_rsqrt(float number)
{
    long i;
    float x2, y;
    const float threehalfs = 1.5F;

    x2 = number * 0.5F;
    y = number;
    i = *(long *)&y;           // evil floating point bit level
hacking
    i = 0x5f3759df - (i >> 1); // what the fuck?
    y = *(float *)&i;
    y = y * (threehalfs - (x2 * y * y)); // 1st iteration
                                         //  y  = y * (
threehalfs - ( x2 * y * y ) );   // 2nd iteration,
                                         // this can be removed

    return y;
}
```

Comme bel exemple de l'esthétique du hack, on a le `fast inverse square root`.

Entre la difficulté de comprendre ce qu'il se passe, et la qualité des commentaires, on peut en tirer deux choses:

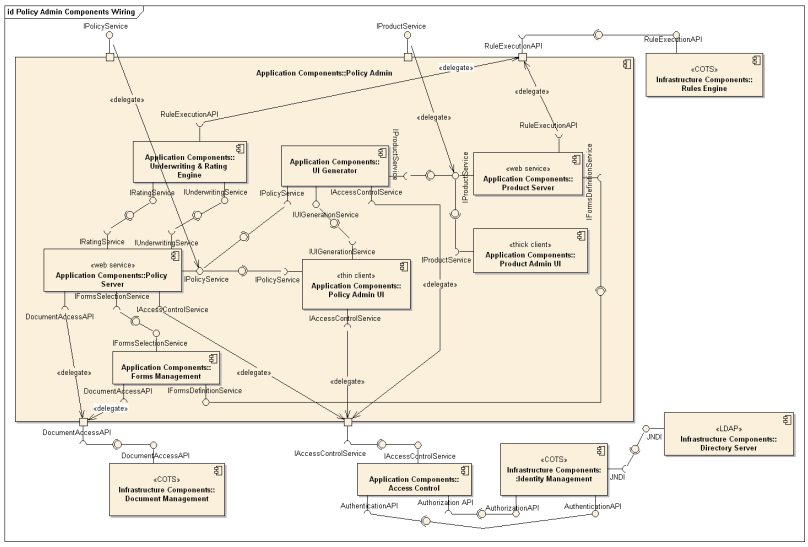
1. on sacrifie la compréhension humaine pour gagner en vitesse de la machine. C'est un peu similaire à l'exemple de Torvalds plus haut, mais avec d'avantage d'attention sur la puissance de calcul que sur l'élégance conceptuelle
2. on maintient un aspect de camaraderie on gardant les commentaires: même si ils ne sont pas du tout nécéssaires (par définition), il consistuent une sorte de public à travers les décennies, et représentent la voix du hacker anonyme qu'on est tous un peu, et une sorte de légende urbaine.

Il y a donc une priorité accordée à la machine, mais tout de même une reconnaissance de l'effet que peut avoir cette priorité sur un lecteur ou une lectrice, et une connaissance pointue de ce qu'est un nombre décimal selon l'IEEE.

Dans la conception de structures architecturales: plannifier par le haut[38], ou bricoler par le bas?[39]

Le débat entre la planification depuis le haut et le bricolage par le bas fait écho aux observations de Papert et Turkle sur les styles durs et doux de programmation. Celui-ci s'est formalisé jusque dans des manuels d'industrie.

Sans poser ces deux approches comme binaires et mutuellement exclusives, on peut néanmoins suggérer des dynamiques.



Universal Modelling Language

Une première approche, c'set l'architecture par le haut, c'est une architecture avec des diagrammes, tels que UML, qui vont imposer une vision et une manière de faire a priori.

Juger positivement un code qui reprend des structures clairement définies dans une documentation, c'est valoriser l'organisation et l'a-priori de la planification, au risque de se heurter aux limites de la réalité de la vie.

Une maison est une machine pour vivre[40] vs. une maison est un espace pour grandir[41]

Une design approche, c'est l'approche des motifs.

Juger positivement un code source avec un motif à l'intérieur, c'est tendre vers la conception que la personne qui implémente a une expérience de la fabrication de code (je reprends ici le langage de l'artisanat), que les valeurs de savoir-faire et de flexibilité sont importantes dans la constitution du texte de programme qu'on est entrain de lire.

Ce sont des débats que l'on voit aussi en architecture du bâti, entre les travaux des modernistes tels que Le Corbusier, qui avait une idée très claire de comment est-ce qu'on doit vivre (esthétique du couple sans enfants), et les travaux de Christopher Alexander, qui vont prôner la nécessité de laisser la vie dicter la mise en forme du bâti.

Un singleton comme brique de construction[42].

```
/**
  * Get the escaper instance (and create if needed)
  */
protected static function escaper(): Escaper
{
  return static::$escaper ??= new Escaper('utf-8');
}
```

D'autre part, l'utilisation d'un pattern de programmation, ici le singleton, se manifeste par une seule ligne élégante de PHP, et manifeste aussi une attention à une manière de concevoir des programmes (patterns), ainsi qu'une application spécifique liée à une connaissance approfondie des opérateurs de PHP (ici le null coalescing operator depuis la version 7.4).

L'anticipation d'autrui dans la communication[43].

```
/*
* If the new process paused because it was
* swapped out, set the stack level to the last call
* to saveu(u_ssav). This means that the return
* which is executed immediately after the call to aretu
* actually returns from the last routine which did
* the savu.
*
* You are not expected to understand this.
*/
```

UNIX[44], précurseur de l'*egoless programming*[45].

Le point de vue du commentaire peut aussi révéler la perception qu'on a de son propre travail, et de la personne qui va devoir gérer son travail. Ici on a l'autre fameux example du code source de la version 6 de UNIX qui va préfacer les routines pour système de partage du temps, ce qui a fait la force de UNIX à l'époque.

Lorsqu'il spécifie explicitement "you are not expected to understand this", que cette possibilité (de ne pas comprendre) n'est pas un problème, elle implique aussi une certaine conception d'autrui.

Les conceptions du langage peuvent aussi se faire de manière soi innée (le langage existe depuis toujours dans notre cerveau, comme dirait Chomsky), soit acquise (le langage se développe dans notre échange avec autrui). Si le langage nous permet alors un certain développement d'un savoir, alors cela pose aussi l'importance d'autrui dans ce processus.

On s'exprime toujours par rapport à quelqu'un, et on peut être plus ou moins conscient de l'horizon d'attente de ce quelqu'un.

Cela se reflète aussi dans la question du style collectif: si on ne veut pas détonner, c'est parce qu'on ne se considère pas comme intrinsèquement meilleur que les autres.

La valeur du résultat[46], ou la valeur du procédé[47].

Un interpréteur de Scheme, écrit en Scheme[48].

```
(define (eval-expr env)
  (lambda (expr env)
    pmatch expr
      [,x (guard (symbol? x))
        (env x)]
      [(lambda (,x) ,body)
        (lambda (arg)
          (eval-expr body (lambda (y)
                            (if (eq? x y)
                                  arg
                                  (env y)))))]
        [(,rator ,rand)
          ((eval-expr rator env)
            (eval-expr rand env))]))
```

D'une part, on peut considérer l'esthétique en mathématiques comme l'expression d'entités désincarnées et objectives (même si elles sont des implications plus générales), telles que l'identité d'Euler. Ici, la valeur est sur l'objet mathématique en lui-même.

C'est ce qu'on percoit ici avec l'interpréteur Scheme. Le code est beau en lui-même, et le procédé parcouru pour atteindre ce code n'est pas pertinent.

D'autre part, on peut considérer, avec Nathalie Sinclair, que l'esthétique la plus valorisée se trouve dans le processus d'atteinte de ces objets mathématiques (ce qu'on appelle en programmation le ***refactoring****). Quand on factorise une équation, quand on travaille sur la solution à un problème, et qu'on voit la forme du problème évoluer, l'expérience esthétique se situe dans ce rapprochement vers "la vérité".*

La différence qu'on peut voir, c'est que la première forme de jugement esthétique demande une certaine connaissance des objets, et donc une sorte de barrière à l'entrée, tandis que la deuxième forme est accessible à d'avantage de personnes, puisqu'elle se focalise sur la progression plutôt que sur le but.

L'accord à l'environnement (*fitness*). [4950]

Finalement, je pense que ce rapport entre esthétique et éthique peut se concevoir comme une rapport adéquat à l'environnement.

L'esthétique dans le code se considère principalement sous l'angle de l'adéquation, de l'appropriation à la fonction, et à l'environnement (matériel et social) dans lequel se manifeste la création qui donne lieu à cette expérience esthétique.

Richard Sennett reprend cette conception dans son étude sur l'artisanat: lorsque la forme et la fonction représentent une adaptation de la matière à l'environnement, alors il y a expérience esthétique.

En revanche, pour déterminer ce qui est approprié ou pas, il faut faire des choix. Ce que je suggère, c'est que hiérarchiser et catégoriser les parties de l'environnement qui vont être importantes (est-ce que c'est l'humain le plus important, ou est-ce que c'est la vitesse de la machine; est-ce que l'attente de la compétence doit suivre le plus petit dénominateur commun, ou le plus grand dénominateur commun?), c'est aussi un choix éthique de bon comportement.

Majeur et mineur

Des esthétiques et des éthiques, dominantes et dominées.

Mais d'où viennent ces esthétiques, et les éthiques qui les accompagnent? Une des critiques de Kant, et critique que je partage, c'est qu'il ait fait passer pour universel les préférences et les intérêts d'un groupe particulier d'individus (ce que Nietzsche contrera dans la généalogie de la morale). Clairement, il n'y a pas qu'une seule éthique, mais il y a bien une éthique dominante (ou majeure).

Un contexte dominant

La programmation se passe toujours dans un contexte social[51][52][53][54].

Le code source n'est pas que le code source, mais aussi son contexte. Et le contexte va informer les paradigmes dominants, qui eux-mêmes vont influer le jugement des outils ("CSS is for girls", "Real men write C"). Jusqu'en 2022, où 93% des répondants du sondage annuel de Stackoverflow sont des hommes.

Les recherches en sciences sociales montrent de plus en plus que l'environnement socio-économique va aussi influencer la conception et l'usage d'une innovation technique, et les langages de programmation ne sont pas des exceptions. Des années 1970 aux années 1990, on voit donc déjà des critiques de langages de programmation qui vont d'avantage dans le sens d'un contrôle managerial et bureaucratique que celui d'une recherche scientifique sur les fondamentaux de la computation.

La programmation structurée[55] comme forme de bureaucratie.

> *[...] perhaps might we ask if culture decides language and computer languages follow the same patterns as any other language, or vice-versa? Or is it really simpler than all that and* ***the programming language really is an attempt, consciously so, to make computers into super-managers based on the same reasoning***

> ***that managing is more important than the object of which it is the manager****?* [56]

> *Even the programmer cannot know the way of decision making in his own program, let alone know what intermediate or final results it will produce. The formulation of a programme is therefore more like the creation of a bureaucracy than the construction of a machine.*[57]

Mais la domination, depuis les années 1970, c'est ce qu'on appelle la programmation structurée, décrite notamment dans les travaux d'Edsger Dijkstra.

Notamment, il y a une critique de la bureaucracie, avec des parallèles qui sont fait entre la manière dont on va composer un programme de façon à ce que les flux de donnée soit toujours gérés, et la manière dont nous-mêmes nous navigons des bureaucraties.

Le génie logiciel comme activité industrielle[58].

> *Structured programming is a philosophy of writing programs according to a set of rigid rules in order to* ***decrease testing problems, increase productivity, and increase the readability*** *of the resulting program.*[59]

Les standards de la conception des langages de programmation, la manière dont ils sont jugés positivement, vont également épouser des standards de productivité entrepreneuriale.

Les standards de la programmation structurée:

- *pyramidal*
- *modulaire*
- *séquentiel*
- *sémantiques limitées (branches conditionnelles et boucles)*

Depuis la fin des années 1970, une énorme majorité des pratiques de programmation suivent ce cadre de fabrication industriel, un paradigme qui ne découle que d'une partie de programmeurs

---

Les standards de la conception de langages de programmation:[6061]

- correct
- rapide
- productif

Et cette industrialisation va aussi se refléter dans ce qu'on considère être un bon langage de programmation (comme le montrait St James lors d'une des précédentes séances du séminaire)

Approches ésotériques

Donc cette manière de faire n'est pas la seule manière de faire (ce n'est jamais la seule manière de faire). Si on repart du pluralisme épistémologique, des styles mous et des styles durs, on peut alors conclure sur des autres manières de faire, des éthiques mineures. L'éthique mineure est un concept développé d'avantage par Gilles Deleuze et Félix Guattari au sujet de l'oeuvre de Franz Kafka.

L'esthétique comme manière de (dé)légitimer.[62]

La question de l'esthétique se pose de manière culturelle.

La question de l'esthétique elle se pose aussi de manière historique. Les discours de la beauté du code commencent à la fin des années 1960, début des années 1970. C'est aussi un moment où la profession du programmeur va changer.

*D'une part, il va y a voir un processus de **légitimisation** de la profession, de devoir suggérer que ce ne sont pas que des techniciens, des travailleurs manuels, relativement aux mathématiciens qui avaient toute la légitimité.*

Et surtout relativement aux personnes qui faisaient la programmation auparavant, c'est-à-dire les femmes. Là où les femmes faisaient du secrétariat manuel, les hommes ont donc inversé le jugement de valeur en utilisant un registre pas seulement esthétique, mais vraiment artistique (cf. Knuth, mais aussi Dijkstra)

Les poèmes en code, tout comme les poèmes en prose, rendent *pensable*.[6364]

```
#! /usr/bin/perl

# repeating history, by pall thalier (2009)

sub relive { $command = shift; print `$command`;}
$bash_history = $ENV{ HOME }."/.bash_history";
while(1){
  open(HISTORY, $bash_history);
  while($moment = ){
    relive($moment);
  }
}
```

Et ici, c'est le concept d'histoire qui est pensable différemment: qu'est-ce que l'historique de d'une session bash nous dit de la linéarité ou de la répétition du travail par ordinateur? Est-ce qu'on fait toujours la même chose, où est-ce qu'il y a quand même une forme de progrès qui se dessine? Est-ce que la récursivité pourrait avoir des airs d'éternel recommencement?

Structures et identités d'une classe[65].

```
class Proc
                           def in_discomfort?; :me; end

end
you_are = you =

   ->(you) do
       self.inspect until true
         until nil
               break you
                                                    end
           puts you.in_discomfort?
             you_are[you]
                                                  end

you[
          you_are
]
```

Ici, le changement de contexte se fait tant au niveau du formattage qu'au niveau du rhythme. La déstabilization sensuelle et perceptible fait apparaître de nouvelles relations et de nouvelles importances: le mot-clé `end`, `self`, ou `break`. Cela peut servir ici à questionner l'origine et le sens de ces mots. La programmation est-elle une activité personnelle et abrupte? Est-ce la seule façon de l'envisager?

Les languages ésotériques pour questionner les approches dominantes[66]

L'une des façons de mettre en évidence les préjugés est de proposer des alternatives (c'est un rôle que l'art en général, et la fiction en particulier, assument facilement). On subvertit les attentes et on déforme la réalité habituelle pour faire apparaître les hypothèses de base, les préjugés de nos modes de travail et d'existence.

En tant que tels, ce sont des artefacts culturels, autant que des artefacts techniques. Ils réagissent à l'esthétique du codage commercial et aux valeurs souvent non déclarées de l'informatique, et s'en inspirent. Ces disciplines, qui s'opposent parfois l'une à l'autre, sont toutes deux guidées par un pragmatisme que les esolangs rejettent activement. En rejetant l'aspect pratique, les esolangs créent leur propre esthétique et mettent en évidence les facteurs contradictoires à l'œuvre dans l'esthétique du code traditionnel.

En faisant cela, ils reconsidèrent l'éthique dominante, ce que Daniel Temkin appelle "the less humble programmer".

TrumpScript[67], et l'orthogonalité du politique et du technique[68].

```
I will ask you, god hears you;
Our country is, safer god;
make money, 2000000 over 1000000;
Make country safe god.
Our Romney is, country over money;
immigrants are, Romney plus 1000000 over 1000000;
politics is true.
As long as, money less immigrants;:
  make america safer, country over money;
  make hard, america times money;
  make earth, hard is country?;
  if, earth; :
    say safer money
    say "is a divisor"
    Make politics false!
  Our money is, money plus 1000000 over 1000000;!
tell not politics
if, politics; :
  tell "We have a prime"!
else:
  tell "No Prime"!
America is great.
```

```
- There are no `import` statements allowed. All code has to be
home-grown and American made.
- If the running computer is from China, TrumpScript will not
compile. We don't want them stealing our American technological
secrets.
- By constructing a wall (providing the `--Wall` flag),
TrumpScript will refuse to run on machines with Mexican locales.
```

Pourquoi faudrait-il toujours assumer la connectivité? Quelle est la dépendance des langages informatiques sur les réseaux de logistique informationelle et industrielle globalisés?

La direction d'abstraction du matériel (hardware), pour laquelle il semble y avoir un désir passif, et représentée par le "Write once, run everywhere" de Java, obfusque les implications matérielles (minières) qui sous tendent la programmation.

C-plus-equality[69], et l'orthogonalité du féminisme et de la technique[70].

```
#consider

// The whole idea of main() is frankly Oppressive, in an ideal
// world there would be no main() or subroutine(), only me()
// Edit: Luckily, we now have womain(), but I still think me()
is better
xe womain() //the alphabet "m" should be banned because it
reminds me of the word "man"
OPENDIALOGUE

// Remember to check your privilege.  Always.
   PrivilegeCheck().
// "std" is sooooo old-fashioned. we use "sti" nowadays.
//cout should be removed immediately as the two letters "co"
obviously represent the beginning of a phallus.
   sti::cout of_the_following "Hello, feminists!\n".  //Frankly
I feel that line escape codes could be problematic
ENDMISOGYNY.

/*
 * Post-amble: Today I wrote my first reclamation program for the
 * Feminist Software Foundation1  I'm sooooo excited!  ^_^
 * Imma cccdddrrrr loll
 * I'm such a nerd!
 * I think I'll write an essay on this triumph over the
Patriarchy!
 * "If you want equal rights, better take equal lefts, too!" <-
lol what I say to PATRIARCHY TODAY WITH MY C+= CODE
 */
```

Bien qu'il existe de nombreuses définitions et approches du féminisme, le fil conducteur est le désir d'égalité, d'inclusion et d'absence d'oppression, la lutte contre l'oppression et l'anticipation (Amy Allen). Les auteurs de C+= ont (paresseusement) conçu leur langage pour montrer que de tels concepts sont radicalement orthogonaux à ce qu'ils considèrent comme l'essence de la programmation.

La question sous-jacente est celle de Félienne Hermans: pourquoi n'y aurait-il pas de langage de programmation féministe ?

En fait, l'inspiration originale pour C+= est un long et riche échange sur la programmation féministe. Par exemple, quelqu'un propose qu'« un langage de programmation féministe est un langage qui

respecte l'agence des objets, agissant sur eux par consentement mutuel ». Cela ne semble pas si exagéré. Peut-être que les langages n'ont pas besoin d'être turing-complets, peut-être que ce n'est pas parce que vous pouvez faire quelque chose que vous devez le faire. Le consentement est peut-être stupide, mais les ordinateurs ont déjà des approches du consensus et de la gentillesse.

Le mineur peut devenir le majeur: le mandarin et le clavier.[71]

La machine à écrire Mingkwai, de Lin Yutang (1947)

À la fin du 19e siècle, la machine à écrire était le nec plus ultra de la technologie. Et parce qu'elle était adaptée à un alphabet gréco-romain, l'opinion de l'époque était que le mandarin était un langage inférieur, parce que les milliers de caractères nécessaires pour s'exprimer ne rentraient pas sur un clavier.

Mais, avec le temps, cette contrainte d'altérité a fini par déboucher sur l'invention du mingwai typewriter, qui est la première machine à écrire qui sépare l'entrée de la sortie (inventant l'autocomplétion cinquante ans avant l'europe).

Conclusion

Les manières d'écrire les codes sources sont des manières de se concevoir soi-même, son appartenance à un groupe, et ses rapports au monde et à la machine.

Un style peut être altruiste, autotélique, prétentieux, industriel, ou queer.

Dire qu'un code est beau, c'est aussi juger la pratique de programmation qui en est à l'origine, c'est le moment où le code source devient révélateur socio-économique.

De par cette nécéssité de la fonction du code, le jugement va être intéressé.

Les langages de programmation peuvent sembler objectifs, mais sont néanmoins contextuels.

Et pour terminer, quelques questions qui me trottent encore et toujours dans la tête:

- *Quels sont vos idéaux et vos buts quand vous programmez? Qu'est-ce que vous négligez, ou minimisez en poursuivant ce but?*
- *Est-ce qu'il y a une part d'essentiel, d'invariable dans les manières de programmer aujourd'hui? Quelle est-elle?*
- *Quels sont les aspects arbitraires de la programmation qu'on pourrait faire évoluer? (Sachant que repenser une syntaxe n'est pas très compliqué. Ce qui l'est d'avantage, c'est de repenser une sémantique!)*
- *Est-il possible de proposer des nouveaux styles de programmation, ou est-ce que tous les langages de programmations convergent vers un semblable agrégat de paradigmes? Quid des LLMs?*
- *Est-ce qu'il y a une part d'essentiel, d'invariable dans les manières de programmer aujourd'hui? Quelle est-elle?*
- *Est-ce que ces différentes manières de représenter des problèmes vont influencer les manières dont le code va fonctionner? Dit autrement, est-ce qu'un logiciel donné (e.g. un gestionnaire de remboursement de déplacement professionnels) codé dans un style fonctionnel va être différent qu'un même logiciel codé en style orienté-objet?*

Appendix

1. Knuth, D. E. (1997). *The Art of Computer Programming, Volume 1 (3rd Ed.): Fundamental Algorithms* . Addison Wesley Longman Publishing Co., Inc.

2. Rokeby, D. (2003). *The Computer as a Prosthetic Organ of Philosophy* . Dichtung Digital.

3. Kay, A. C. (1993). The early history of Smalltalk. *ACM SIGPLAN Notices* , 28(3), 69–95.

4. Wolfram, S. (2002). *A New Kind of Science (1st edition)* . Wolfram Media, Inc.

5. Goodman, N. (1982). Implementation of the Arts. *The Journal of Aesthetics and Art Criticism* , 40(3), 281–283.

6. Lessig, L. (1999). *Code and Other Laws of Cyberspace* . Basic Books, Inc.

7. Chun, W. H. K. (2008). On “Sourcery,” or Code as Fetish. *Configurations* , 16(3), 299–324.

8. Saito, Y. (2012). Everyday Aesthetics and Artification. *Contemporary Aesthetics* , 4.

9. Certeau, M. de, Giard, L., & Mayol, P. (1990). *L’invention du quotidien* . Gallimard.

10. Macé, M. (2016). *Styles: Critique de nos formes de vie* (NRF Essais). Gallimard.

11. Dewey, J. (1980). *Art as an Experience* . Perigee.

12. Lakoff, G. (1980). _Conceptual Metaphor in Everyday Language. *The Journal of Philosophy* , 77(8).

13. Knuth, D. E. (1984). Literate programming. *The Computer Journal* , 27(2), 97–111. https://doi.org/10.1093/comjnl/27.2.97

14. Matsumoto, Y. (2007). Treating Code as an Essay. In A. Oram & G. Wilson (Eds.), *Beautiful Code: Leading Programmers Explain How They Think (1st edition)* . O’Reilly Media.

15. Heiss, J., & Gabriel, R. P. (2020, July 21). *The Poetry of Programming* . Dreamsongs.Org.

16. Alexander, C., Ishikawa, S., Silverstein, M., Jacobson, M., Fiksdahl-King, I., & Angel, S. (1977). *A Pattern Language: Towns, Buildings, Construction* . Oxford University Press.

17. Raymond, E. S. (2001). *The Cathedral & the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary* . O’Reilly Media, Inc.

18. Nielsen, M. (2012, November 4). *Lisp as the Maxwell’s equations of software* . DDI.

19. Steele, G. L. (1977). Macaroni is better than spaghetti. *ACM SIGPLAN Notices* , 12(8), 60–66.

20. Irmak, N. (2012). Software is an Abstract Artifact. *Grazer Philosophische Studien* , 86(1), 55–72.

21. Turner, R. (2018). Computational Artifacts. In R. Turner (Ed.), *Computational Artifacts: Towards a Philosophy of Computer Science* (pp. 25–29). Springer.

22. Jackson, M. (1995). The world and the machine. Proceedings of the 17th International Conference on Software Engineering, ICSE '95, 283–292.

23. Simmel, G. (1991). The Problem of Style. *Theory, Culture & Society* , 8(3), 63–71.

24. Granger, G.-G. (1988). *Essai d'une philosophie du style* . Odile Jacob.

25. Croll, A. (2014). *If Hemingway Wrote JavaScript* . No Starch Press.

26. Hebdige, D. (1979). *Subculture: The meaning of style* . London : Methuen.

27. Petricek, T. (2025). *Cultures of programming: The development of programming concepts and methodologies* . Cambridge University Press.

28. Perlis, A. J. (1982). Special Feature: Epigrams on programming. *ACM SIGPLAN Notices* , 17(9), 7–13.

29. Hacking, I. (1994). Styles of Scientific Thinking or Reasoning: A New Analytical Tool for Historians and Philosophers of the Sciences. In K. Gavroglu, J. Christianidis, & E. Nicolaidis (Eds.), *Trends in the Historiography of Science* (pp. 31–48). Springer Netherlands.

30. Depaz, P. (2023). Stylistique de la recherche linguistique en IA: de LISP à GPT-3. In A. Gefen (Ed.), *Créativités artificielles – La littérature et l'art à l'heure de l'intelligence artificielle* (pp. 189–209). Les Presses du Réel.

31. Turkle, S., & Papert, S. (1992). Epistemological Pluralism and the Revaluation of the Concrete. *The Journal of Mathematical Behavior* .

32. Eaton, M. M. (1997). Aesthetics: The Mother of Ethics? *The Journal of Aesthetics and Art Criticism* , 55(4), 355–364.

33. Rancière, J. (2000). *Le partage du sensible* . La Fabrique Éditions.

34. Rohrhuber, J. (2018). Sans-papiers as first-class citizens. In G. Primiero & L. D. Mol (Eds.), *Reflections on programming systems: Historical and philosophical aspects* (pp. 153–185). Springer Verlag.

35. Jameson, F. (1991). *Postmodernism Or, The Cultural Logic Of Late Capitalism* . Verso.

36. Kirchner, M. (2022). *Linked List—Removing* . Software Heritage. (SWHID: swh:1:cnt:6dd41adbff62aa9cd5a310690d5b3943ae52c1bf)

37. Tarolli, G. (1999). Q_math.c [C; PC]. Id Software. (SWHID: swh:1:cnt:bb0faf6919fc60636b2696f32ec9b3c2adb247fe)

38. Shaw, M., & Garlan, D. (1996). Software Architecture: Perspectives on an Emerging Discipline. Pearson.

39. Gamma, E., Helm, R., Johnson, R., Vlissides, J., & Booch, G. (1994). Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley Professional.

40. Le Corbusier & Saugnier. (1923). Vers une architecture (Vol. 1–1). G. Crès.

41. Alexander, C. (1979). The Timeless Way of Building. Oxford University Press.

42. Bestle, L. (2026). Escape.php [PHP]. Kirby. (SWHID: swh:1:cnt:9f23c62e0a95025511fa3d89312c093b43c0ca78)

43. Voloshinov, V. N., & Bachtin, M. M. (1986). *Marxism and the Philosophy of Language* . Harvard University Press.

44. Lions, J. (1996). *Lions' Commentary on UNIX 6th Edition with Source Code* . Peer-to-Peer Communications.

45. Brooks Jr, F. P. (1975). *The Mythical Man-month: Essays on Software Engineering* . Addison-Wesley Publishing Company.

46. Wells, D. (1990). Are these the most beautiful? *The Mathematical Intelligencer* , 12(3), 37–41.

47. Sinclair, N. (2004). The Roles of the Aesthetic in Mathematical Inquiry. Mathematical Thinking and Learning, 6(3), 261–284.

48. Byrd, W. (2017, May 24). The Most Beautiful Program Ever Written. Papers We love NYC, NYC. http://paperswelove.org/2017/video/will-byrd-most-beautiful-program/

49. Parsons, G., & Carlson, A. (2012). *Functional Beauty* . Oxford University Press.

50. Sennett, R. (2009). *The Craftsman* . Yale University Press.

51. Posner, M. (2017). *Javascript is for Girls* Logic Magazine (1).

52. McPherson, T. (2018). *Feminist in a software lab: Difference + design* . Harvard University Press.

53. Gupta, H. (2016). *(Lack Of) Representation of Non Western World in process of creation of Web standards* . arXiv:1609.01996 [Cs].

54. StackOverflow. (n.d.). *Stack Overflow Developer Survey 2022* . Stack Overflow. Retrieved April 14, 2022, from https://insights.stackoverflow.com/survey/2022

55. Dijkstra, E. W. (1972). *Structured programming* . Academic Press Ltd.

56. Barnhart, D. (1991). *The Relationship of Hierarchical Computing and Managerial Systems* . The Montana Professor, 1(1).

57. Weizenbaum, J. (1976). *Computer Power and Human Reason: From Judgment to Calculation* (1st edition). W. H. Freeman & Co.

58. Campbell-Kelly, M. (2003). *From airline reservations to Sonic the Hedgehog: A history of the software industry* . Cambridge, Mass. : MIT Press

59. Yourdon, E., & Constantine, L. L.. (1979). *Structured design: Fundamentals of a discipline of computer program and systems design* . Englewood Cliffs, N.J. : Prentice Hall.

60. Stansifer, R. (1994). *The Study of Programming Languages* (1st edition). Prentice Hall.

61. Sebesta, R. W. (2018). *Concepts of Programming Languages* (12th edition). Pearson.

62. Mrozinski, S. (2024) *Fighting Baroque Monstrosities: Aesthetic Values in Programming and Programming Languages* . HaPoP-6 Fairness and Bias in Programming, Cambridge.

63. Bertram, I. (éd.) (2012). *Code {poems}* . Impremta Badia.

64. Flusser, V., & Novaes, R. M. (2014). *On Doubt* . University of Minnesota Press.

65. Ortega, M. (2011). *self_inspect.rb* [Ruby]. (SWHID: swh:1:cnt:fe2a589380e7d529b73bba90d9f128caf3e8efd8) (Original work published 2011)

66. Temkin, D. (2023). The Less Humble Programmer. *Digital Humanities Quarterly* , 17(2).

67. Shadwell, S. (2016). prime_Test.tr [Python]. (SWHID: swh:1:cnt:3f3a0e9e8445fa9040f94b70aa28f8648fd08dc3). https://github.com/samshadwell/TrumpScript

68. Winner, L. (1980). Do Artifacts Have Politics? *Daedalus* , 109(1), 121–136.

69. TheFeministSoftwareFoundation. (2024). Hellofeminists.xe [C++]. (SWHID: swh:1:cnt:62bbae8b220d6d92c36c44ed69deb8a6b894979c). https://github.com/TheFeministSoftwareFoundation/C-plus-Equality (Original work published 2014)

70. Hermans, F., & Schlesinger, A. (2024). *A Case for Feminism in Programming Language Design* . Proceedings of the 2024 ACM SIGPLAN International Symposium on New

Ideas, New Paradigms, and Reflections on Programming and Software, Onward! ’24, 205–222.

71. Mullaney, T. S. (2018). *The Chinese Typewriter: A History* . MIT Press.