MEMO 02/2017: Michael Murtaugh:

DO REPEAT YOURSELF

Do (not) Repeat Yourself was written by Michael Murtaugh.

The text was first published 2014 in **Fun and Sofware** edited by Olga Goriunova.

http://c2.com/cgi/wiki?DontRepeatYourself

Dont Repeat Yourself

(Don't Repeat Yourself. Don't Repeat Yourself.)

Context:

Duplication (inadvertent or purposeful) can lead to maintenance nightmares, poor factoring, and logical contradictions.

Duplication, and the strong possibility of eventual contradiction, can arise anywhere: in architecture,requirements, code, or documentation. The effects can range from misimplemented code and developer confusion to complete system failure.

The Mars Climate Orbiter was lost due to a semantic contradiction: part of the system was working in Imperial units, another in Metric. There was a duplication of knowledge (implicit units), and the duplicates were out of step.
One could argue that the most of the difficulty in Y2K remediation is due to the lack of a single date abstraction within any given system; the knowledge of dates and date-handling is widely spread.

The Problem:

But what exactly counts as duplication? Copy-and-paste is generally cited as the chief culprit (see OnceAndOnlyOnce, etc.), but there is more to it than that. Whether in code, architecture, requirements documents, or user documentation, duplication of knowledge - not just text - is the real culprit.

Therefore:

The DRY (Don't Repeat Yourself) Principle states:

> *Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.*

It's a battle of two really strong urges -- OnceAndOnlyOnce vs. avoiding PrematureGeneralization. Do I duplicate for now and try to live with the duplication for a while, or violate YagNi and come up with some half-cocked generalized solution? It's a tough one, because almost all programmers hate duplication; it's a sort of primordial programming urge.

http://c2.com/cgi/wiki?DontRepeatYourself

Dont Repeat Yourself

(Don't Repeat Yourself. Don't Repeat Yourself.)

Context:

Duplication (inadvertent or purposeful) can lead to maintenance nightmares, poor factoring, and logical contradictions.

Duplication, and the strong possibility of eventual contradiction, can arise anywhere: in architecture,requirements, code, or documentation. The effects can range from misimplemented code and developer confusion to complete system failure.

The Mars Climate Orbiter was lost due to a semantic contradiction: part of the system was working in Imperial units, another in Metric. There was a duplication of knowledge (implicit units), and the duplicates were out of step.
One could argue that the most of the difficulty in Y2K remediation is due to the lack of a single date abstraction within any given system; the knowledge of dates and date-handling is widely spread.

The Problem:

But what exactly counts as duplication? Copy-and-paste is generally cited as the chief culprit (see OnceAndOnlyOnce, etc.), but there is more to it than that. Whether in code, architecture, requirements documents, or user documentation, duplication of knowledge - not just text - is the real culprit.

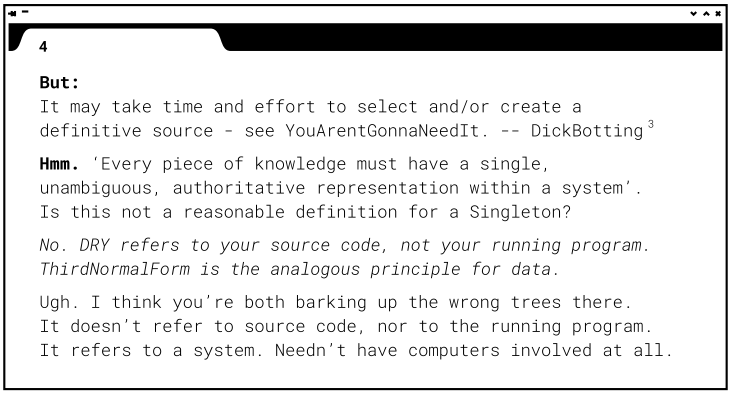
Therefore:

The DRY (Don't Repeat Yourself) Principle states:

> Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.

It's a battle of two really strong urges -- OnceAndOnlyOnce vs. avoiding PrematureGeneralization. Do I duplicate for now and try to live with the duplication for a while, or violate YAgNi and come up with some half-cocked generalized solution? It's a tough one, because almost all programmers hate duplication; it's a sort of primordial programming urge.

This long quote is the opening section of the page titled ‘Don’t Repeat Yourself’ (DRY) on the Portland Pattern Repository (PPR). The pages of the website, devoted in part to the practice of programming, are heavily cross-linked, forming a messy tangle of programming paradigms and self-help tips for improving code with `CamelCased` hyperlinks,[1] often in the form of an imperative such as `RefactorMercilessly`, `PutThingsWhereYouLook` and `SeparateTheWhatFromTheHow`. At the same time it contains seemingly contradictory meta-tips that warn against `SilverBullet` and `OneTrickPony` solutions. Despite the apparent clarity and appeal for *authoritativeness*, later in the same page, the discussion presented above takes on a slightly different tone:


4

But:
It may take time and effort to select and/or create a definitive source - see YouArentGonnaNeedIt. -- DickBotting[3]

Hmm. ‘Every piece of knowledge must have a single, unambiguous, authoritative representation within a system’. Is this not a reasonable definition for a Singleton?

No. DRY refers to your source code, not your running program. ThirdNormalForm is the analogous principle for data.

Ugh. I think you’re both barking up the wrong trees there. It doesn’t refer to source code, nor to the running program. It refers to a system. Needn’t have computers involved at all.

The PPR is the product of a particular community of programmers. It is in fact the ‘mother of all wikis’, and was created (and maintained) by Ward Cunningham, who coined the term ‘wiki’. The fact that the discourse of the PPR takes place on, and came about in conjunction with the development of, the wiki, is significant. While superficially a wiki page provides for a kind of ‘single source’, its great strength lies in the fact that by containing the entire history of edits, by permitting editing by anyone and by allowing differences of opinion to be made explicit, the wiki itself is far from being free of contradiction or duplication.

---

1 `CamelCased` hyperlinks are links in which capitalized words are merged into compound words, sometimes used in Wiki markup languages for key terms that become automatically linked to other Wiki pages. See the entry on Wikipedia: http://en.wikipedia.org/wiki/`CamelCase` (accessed 13.06.2013).

2 http://c2.com/cgi/wiki?`DickBotting`

3 See the page ‘Don’t Repeat Yourself’.

The language of seeking 'single, authoritative, unambiguous' knowledge by the architects of the software that would go on to inspire the more popular Wikipedia project seems to sadly devalue its own core strengths. While the prescriptive nature of the DRY discourse appeals to a programmer's sense of utility and efficiency, it also seems to be in denial of the very nature of the practice it aims to serve.

Programmers walk a fine line between seeking ecstatic singularities while at the same time enduring dutiful and crushing conformity to correctness and convention, performing a practice that is by nature highly repetitive. Programmers often have to search for a balance between considering the details of a particular situation and flying into euphoric quests for abstraction, desiring code that goes beyond the satisfaction of some particular need into the ecstatic realm of the unforeseen and unexpected. The quasi-ejaculatory nature of this process is evident in another of the PPR principles, that of avoiding 'premature generalization'.

Popular web programming frameworks, such as Ruby on Rails and Django, pride themselves on adhering to the 'principle of DRY'. The Rails framework initially promoted itself as a response to the drudgery of website programming and was unique in listing the 'joy' of using it, among its other technical features, as a way of winning over programmers: 'Rails is a full-stack, open-source web framework in Ruby for writing real-world applications with joy and less code than most frameworks spend doing XML sit-ups'.[4] Such 'joy' is contrasted with repetition, something much less desirable and seen as valueless, with its essential qualities remaining unrecognized.

However, it is often the programmers' recognition of a pattern already learned through repetition that is most compelling in the use of a particular framework. In the same way that one feels lost and isolated in the woods and then reassured when one comes across a well worn path, frameworks are condensations of practice and reinforce a sense of community among their practitioners. In contrast to the practice of belonging, the fun to be had in the pioneering discovery of the novel or unique is isolating. In addition to this, repetition is intrinsically intertwined with the development of the craft of programming.

---

[4] The 'Ruby on Rails' website soon after its launch in 2005 – archived link, http://web.archive.org/web/20050601015146/http://www.rubyonrails.org/ (accessed 13.06.2013).

> *[. . . Skill] development depends on how repetition is organised. This is why in music, as in sports, the length of a practice session must be carefully judged: the number of times one repeats a piece can be no more than an individual's attention span at a given stage. As skill expands, the capacity to sustain repetition increases. In music this is the so-called Isaac Stern rule, the great violinist declaring that the better your technique, the longer you can rehearse without becoming bored. There are 'Eureka moments' that turn the lock in a practice that has jammed, but they are embedded in routine.* [5]

There can be a tangible pleasure in quickly typing out the template of a familiar programming structure. Far from celebrating the birth of a unique new creation from scratch, it is rather a joyful expression of the pattern that increasingly becomes physically embodied in the programmer him/herself. Here, the material that one once struggled with, with time becomes something ingrained in 'one's fingers'.

On the surface, the black box of abstraction promises the programmer that if he/she can only get the abstraction right, he/she will never have to deal with a particular kind of problem again. In fact, working with abstraction is a gradual process, inclusive of struggling repeatedly with the material of a problem and, thus, acquiring the skill that would feel 'natural and easy' by becoming a part of the body of the programmer. Rather than removing the problem, repetition produces increased capacity to deal with the problem and, thus, the problem can be repeatedly successfully tackled. The formalization of abstraction in the form of code (the syntax and naming of a function object, for instance) can be seen as merely a culmination of the necessary prerequisite of practising a repetitive process that made its recognition and recall possible.

Working with recursion is a particular kind of programming skill that often takes a great deal of practice before it is fully mastered. Once it is learned, there is a self-effacing wonder in watching 20 lines of code dissolving down to ten, then five, as all the 'edge cases' that could possibly be imagined map onto the few folds of a particular recursive structure. Similar to the pleasure of kneading dough, working with recursion is about the almost miraculous transformation of code through repeatedly working it.

---

[5] Sennett, Richard: *The Craftsman*. London, UK 2009. 38.

Repetition is an essential part of the process of recognizing and constructing abstractions. The fact that experienced programmers might directly write code using concise and 'correct' abstractions is more a reflection of their experience than an absolute (and transferable) measure of quality.

Code smells

> *If you are aware of* `CodeSmells`*, and duplicate code is one of the strongest, and you react accordingly, your systems will get simpler. When I began working in this style, I had to give up the idea that I had the perfect vision of the system to which the system had to conform. Instead, I had to accept that I was only the vehicle for the system expressing its own desire for simplicity. My vision could shape initial direction, and my attention to the desires of the code could affect how quickly and how well the system found its desired shape, but the system is riding me much more than I am riding the system.* [6]

My great Aunt Margaret, a piano teacher and former Catholic school principal, would implore all overnighting nieces and nephews to take a shower in the morning, to counter the 'beddy smell'. Naive as I was, it was only years later that it occurred to me how the term 'beddy' was in fact a thinly veiled euphemism for the less comfortable subject of one's 'body'. Just as my Aunt displaced the body by the bed, the struggles of programming often get projected from programmers onto the code. Bad code has a smell, independent desires and an ability to ride or be ridden by the programmer. Repetition can be experienced as a tiring physical exercise.

In addition to devaluing repetition as something smelling 'bad', something to be either absolutely avoided or at best tolerated, the transference of a smell to code is also indicative of another set of displacements, such as the code being separated from the practice of programming, and the practice of programming being separated from the physical effort required of the body of the programmer him/herself. Coding can be, and often is, physically exhausting work, as illustrated by the following passage:

---

[6] See the page 'Once and Only Once', http://c2.com/cgi/wiki?OnceAndOnlyOnce, citation signed '`KentBeck`, feeling mystical, see `MysticalProgramming`'– another page at PPR (accessed 13.06.2013).

Just got off the phone with F. Feeling slightly remorseful at being kind of pissy and short. Maybe need to write an email: Sorry I was vague. Orality impaired – I could better write an email. I've been coding intensely over the past 4 days. I was only half-listening as F ran down the planning for the workshop – discounting that information that I already knew (annoyed at the redundancies). . . a Borg voice speaks to me from the collective: this information is not relevant; this conversation is inefficient. I'm having difficulty following what's being said.

Thinking of the 'code smells' – after these long stretches of coding the smells take a physical form – though it's not coming from the code . . . I'm unable to tell if the unpleasant odors I seem sporadically aware of are originating from rotting garbage in the bin or from me. Reminiscent of baby diapers. Probably the garbage bag . . . must be.

Dim the screen – too bright. On with coding. . .

I can no longer come up with meaningful names for things. Have started using names like aa, bb, and aaa. Switching between these abstract symbols seems easier; reduced semantic overload = less need to think.

Can no longer remember what task I am currently working on. I start writing down tasks not 'to do', but what I'm supposedly 'doing now', so that each time I slip, I can refer to the note. Need to update the model and regenerate the database before going to bed. Before I forget why it's important. [7]

For programmer Will Crowthers, commenting on his hobby below, rock climbing does not merely provide an escape from programming:

You would have to forget about everything. When you're rock climbing, you must not think about anything but the rock climbing or you're apt to get killed. And it just wipes everything out for a day or two or whatever it is. However long you're off climbing, which tended to be a weekend, you don't think of much else. [8]

---

[7] Personal notes written during programming work.

[8] INTERVIEW WITH WILL CROWTHER., 1994. p.4

Dim the screen – too bright. On with coding...

Crowthers' 'extreme hobby' parallels the intensity of his experience of programming itself. He needs an escape with a sufficiently matching intensity to give him a break from that of the engrossingly 'disembodied' practice of coding. Crowthers is best known for programming the early computer game classic 'Collossal Cave Adventure' (also known as 'Adventure') in his spare time while working at the company BBN Technologies in Massachusetts in the late 1970s. A pioneering example of an interactive program that simulates the experience of cave exploration, the writing of the game was, according to Crowthers, in part an attempt to reconnect to his children after an estrangement from his wife whom he had met while caving. [9]

'Extreme Programming', a concept whose origin and development can be traced to Kent Beck on the PPR, [10] has become an important movement concerning rethinking traditional approaches to software programming practice. Beck's choice of the term 'extreme programming' clearly invokes 'extreme sport', and indirectly references the often frustrated desires of a programmer to experience, in their practice of programming, the intensity of physical experience such as that described by Crowthers.

According to the principles of DRY, it would seem that the job of a programmer is to detect patterns and to fold these into redundancy-free perfection. This suggests an ideal, Plato-inspired practice of programming, wherein the programmer, after meditative moments of reflection, is able to effortlessly condense the chaotic cacophony of the reality around him/herself into a stream of precise expressions, gliding from unique formulation to unique formulation and never looking back.

People imagine that computer programming is logical, a process similar to the one of fixing a clock. Nothing could be further from the truth. Programming is more akin to an illness, a fever, an obsession. It is like riding a train and never being able to get off. [11]

Programmer and journalist Ellen Ullman compares software design to using methamphetamine, as the 'speed high is the only state that approximates the feel of a project at its inception. Yes, I understand. Yes, it can be done. Yes, how straightforward. Oh yes, I see'. The trip is, however, brought to an abrupt end when 'you write some code, and suddenly there are dark, unspecified areas'. [12]

---

[9] INTERVIEW WITH WILL CROWTHER., 1994.

[10] See http://c2.com/cgi/wiki?ExtremeProgramming (accessed 13.06.2013).

[11] Ullman, Ellen: "Out of Time: Reflections on the Programming Life". In: Brook, James/Boal, Iain A. (eds.): *Resisting the virtual life: the culture and politics of information*. 1995.

[12] Ullman, Ellen: *Close to the Machine: Technophilia and Its Discontents*. 1997. p.21

Ullman's description of how the transition from the plan to the writing of code drops from the luminous clarity of a pre-implementation specification into the dark areas of the unspecified seems to invoke something of the fear and pleasure of Crowther's cave explorations. The experience of working with code can be an exhilarating modulation between the light and the dark, between losing and regaining one's footing, between the logical and the absurd.

Do repeat yourself

> *There is always a reason for missing an easy toss. Repeat toss and you will find it. If you rap your knuckles against a window jamb or door, if you brush your leg against a desk or a bed, if you catch your feet in the curled-up corner of a rug, or strike a toe against a desk or chair go back and repeat the sequence. You will be surprised to find how far off course you were to hit that window jamb that door that chair. Get back on course and do it again. How can you pilot a spacecraft if you can't find your way around you own apartment? It's just like retaking a movie shot until you get it right. And you will begin to feel yourself in a film moving with ease and speed. But don't try for speed at first.* [13]

In the short story 'The Discipline of DE', William Burroughs, in the guise of a retired Colonel Sutton-Smith, describes to the reader the joys of living a life according to the prescripts of 'Do Easy'. The text is written primarily in the second person: a parodic fusion of self-help guide and military pep talk. While DE's 'do repeat yourself' paradigm would ostensibly seem to oppose the 'don't' of `DontRepeatYourself`, the two have a lot in common. When Burroughs writes: 'once you find the easy way you don't have to think about it, . . . it will almost do itself', [14] one can hear echoes of Kent Beck's 'mystical' musings of a systems' 'own desire for simplicity', cited above. DE's message of 'repeat until perfection' captures much of the reality of software design practice: a frequently obsessive attention to detail and process, a tendency towards excessive (self-) optimization and an aesthetization of efficiency. [15] The promises of ease and speed could be taken straight from the copy of a new programming framework.

---

[13] Burroughs, William S.: *Word Virus: The William S. Burroughs Reader*. 1999. p.386–92

[14] Burroughs, William S.: *Word Virus: The William S. Burroughs Reader*. 1999. p.390

[15] Fuller, Matthew: "Elegance". In: idem (ed.): *Software Studies – A Lexicon*. Cambridge, Massachusetts 2008. p.87–92

Everyday tasks become painful and boring because you think of them as WORK something solid and heavy to be fumbled and stumbled over. Overcome this block and you will find that DE can be applied to anything you do even to the final discipline of doing nothing. The easier you do it the less you have to do. He who has learned to do nothing with his whole mind and body will have everything done for him. [16]

Burroughs' reductio ad absurdum reveals a dark side to DRY in its relation to a larger software industry. In a system where code is a product to be protected and exploited commercially, the efficiency of the process tends to eliminate the usefulness of the programmer; truly efficient coding would lead to a point where the coder him/herself becomes 'redundant', expendable. Burroughs' final image of the mastery of doing nothing leading to a position of privilege and power concisely reveals the implicit motivations behind much of the venture capital interest in software development.

The GNU project was the response of one programmer, Richard Stallman, to what he felt were the injustices of a software industry that separated the programmer from the product of his/her labour through nondisclosure agreements and restrictive software licenses. The GNU project and the ensuing Free Software movement, encourage a practice of software development whereby code is released under a license that ensures that it remains not only freely usable, but also reworkable and redistributable by subsequent programmers. In 'freeing' the code, the General Public License (GPL) shifts value from the code to the surrounding practice. The value of free software is the community of developers, documenters, researchers, designers and users which is rather than the 'shrink-wrapped product' or 'killer app' *per se*.

Even with free software's fundamental shift in value from code to community, Stallman's early manifesto still includes a 'DRY' stance as one of the core 'benefits' of the project. '[The GNU project] means that much wasteful duplication of system programming effort will be avoided. This effort can go instead into advancing the state of the art.' [17] As with the PPR, arguments for efficiency seem to be inevitably made even when they contradict the realities of the practice.

---

[16] Burroughs, William S.: *Word Virus: The William S. Burroughs Reader*. 1999. p.391
[17] THE GNU MANIFESTO. Richard Stallman, 1985.

The free software community is a rich tapestry of duplication, forked projects and reinventions of the proverbial wheel. The term 'yet another' is common in the names of free software projects as a humorous way of acknowledging (and gently atoning for) the redundancy.[18] Recursion and contradiction play a substantial role in programmer humour. The GNU name itself (standing for 'GNU's not UNIX') is a kind of nerd joke, doubly contradictory both as a version of UNIX that is not UNIX, and inherently incomplete in its recursive definition. In a similar way the very formulation of 'Don't Repeat Yourself' as a kind of a programmer's mantra, and thus to be recursively repeated, is also absurd.

The negative implications of separating code from practice are many: formal software instruction is pervasively discouraging to beginners and the 'uninitiated'.[19] The labour of software design is easily exploitable and software professions are precarious, whereas the economic forces promote the fragile and decontextualized product of code and ignore its larger sustaining community.

As a programmer, I 'get' DRY and I value Beck and the PPR in their contributions to software and to the discussion of software practice. The problem is that maxims, such as **'Don't repeat yourself'** only work when they are not taken literally, and when their implicit values are questioned. There is a continual need to (re)value software practice and avoid reducing it to a kind of 'shortest path' problem.

Software practice includes logical contradiction, necessitates 'bad' code and requires repetition. When, in the definition of DRY, the '*duplication of knowledge, not just text*' is defined as a core part of '*the problem*' to be solved, it exposes an impoverished conception of knowledge isolated from practice. In software design, as in other forms of cultural discourse, redundancy and repetition are essential to the necessarily incomplete processes of knowledge production, practice, circulation and maintenance.

---

[18] YET ANOTHER — WIKIPEDIA, THE FREE ENCYCLOPEDIA. Wikipedia, 2013.

[19] See, for instance, Margolis, Jane/Fisher, Allan: *Unlocking the Clubhouse: Women in Computing*. 2003. A quote from Papert can also be useful here: "Children do not follow a learning path that goes from one *true position* to another, more advanced *true position*. Their natural learning paths include *false theories* that teach as much about theory building as true ones. But in school *false theories* are no longer tolerated. [. . .] Our educational system rejects the *false theories* of children, thereby rejecting the way children really learn".
Papert, Seymour: *Mindstorms: Children, Computers, and Powerful Ideas*. New York, NY, USA 1993.

Free Art License 1.3. (C) Copyleft Attitude, 2007. You can make reproductions and distribute this license verbatim (without any changes). Translation: Jonathan Clarke, Benjamin Jean, Griselda Jung, Fanny Mourguet, Antoine Pitrou. Thanks to framalang.org

PREAMBLE

The Free Art License grants the right to freely copy, distribute, and transform creative works without infringing the author's rights.

The Free Art License recognizes and protects these rights. Their implementation has been reformulated in order to allow everyone to use creations of the human mind in a creative manner, regardless of their types and ways of expression.

While the public's access to creations of the human mind usually is restricted by the implementation of copyright law, it is favoured by the Free Art License. This license intends to allow the use of a works resources; to establish new conditions for creating in order to increase creation opportunities. The Free Art License grants the right to use a work, and acknowledges the right holders and the users rights and responsibility.

The invention and development of digital technologies, Internet and Free Software have changed creation methods: creations of the human mind can obviously be distributed, exchanged, and transformed. They allow to produce common works to which everyone can contribute to the benefit of all.

The main rationale for this Free Art License is to promote and protect these creations of the human mind according to the principles of copyleft: freedom to use, copy, distribute, transform, and prohibition of exclusive appropriation.

DEFINITIONS

"work" either means the initial work, the subsequent works or the common work as defined hereafter:

"common work" means a work composed of the initial work and all subsequent contributions to it (originals and copies). The initial author is the one who, by choosing this license, defines the conditions under which contributions are made.

"Initial work" means the work created by the initiator of the common work (as defined above), the copies of which can be modified by whoever wants to

"Subsequent works" means the contributions made by authors who participate in the evolution of the common work by exercising the rights to reproduce, distribute, and modify that are granted by the license.

"Originals" (sources or resources of the work) means all copies of either the initial work or any subsequent work mentioning a date and used by their author(s) as references for any subsequent updates, interpretations, copies or reproductions.

"Copy" means any reproduction of an original as defined by this license.

OBJECT

The aim of this license is to define the conditions under which one can use this work freely.

SCOPE

This work is subject to copyright law. Through this license its author specifies the extent to which you can copy, distribute, and modify it.

FREEDOM TO COPY (OR TO MAKE REPRODUCTIONS)

You have the right to copy this work for yourself, your friends or any other person, whatever the technique used.

FREEDOM TO DISTRIBUTE, TO PERFORM IN PUBLIC

You have the right to distribute copies of this work; whether modified or not, whatever the medium and the place, with or without any charge, provided that you: attach this license without any modification to the copies of this work or indicate precisely where the license can be found, specify to the recipient the names of the author(s) of the originals, including yours if you have modified the work, specify to the recipient where to access the originals (either initial or subsequent). The authors of the originals may, if they wish to, give you the right to distribute the originals under the same conditions as the copies.

FREEDOM TO MODIFY

You have the right to modify copies of the originals (whether initial or subsequent) provided you comply with the following conditions: all conditions in article 2.2 above, if you distribute modified copies; indicate that the work has been modified and, if it is possible, what kind of modifications have been made; distribute the subsequent work under the same license or any compatible license. The author(s) of the original work may give you the right to modify it under the same conditions as the copies.

RELATED RIGHTS

Activities giving rise to authors rights and related rights shall not challenge the rights granted by this license. For example, this is the reason why performances must be subject to the same license or a compatible license. Similarly, integrating the work in a database, a compilation or an anthology shall not prevent anyone from using the work under the same conditions as those defined in this license.

INCORPORATION OF THE WORK

Incorporating this work into a larger work that is not subject to the Free Art License shall not challenge the rights granted by this license. If the work can no longer be accessed apart from the larger work in which it is incorporated, then incorporation shall only be allowed under the condition that the larger work is subject either to the Free Art License or a compatible license.

COMPATIBILITY

A license is compatible with the Free Art License provided: it gives the right to copy, distribute, and modify copies of the work including for commercial purposes and without any other restrictions than those required by the respect of the other compatibility criteria; it ensures proper attribution of the work to its authors and access to previous versions of the work when possible; it recognizes the Free Art License as compatible (reciprocity); it requires that changes made to the work be subject to the same license or to a license which also meets these compatibility criteria.

YOUR INTELLECTUAL RIGHTS

This license does not aim at denying your author's rights in your contribution or any related right. By choosing to contribute to the development of this common work, you only agree to grant others the same rights with regard to your contribution as those you were granted by this license. Conferring these rights does not mean you have to give up your intellectual rights.

YOUR RESPONSIBILITIES

The freedom to use the work as defined by the Free Art License (right to copy, distribute, modify) implies that everyone is responsible for their own actions.

DURATION OF THE LICENSE

This license takes effect as of your acceptance of its terms. The act of copying, distributing, or modifying the work constitutes a tacit agreement. This license will remain in effect for as long as the copyright which is attached to the work. If you do not respect the terms of this license, you automatically lose the rights that it confers. If the legal status or legislation to which you are subject makes it impossible for you to respect the terms of this license, you may not make use of the rights which it confers.

VARIOUS VERSIONS OF THE LICENSE

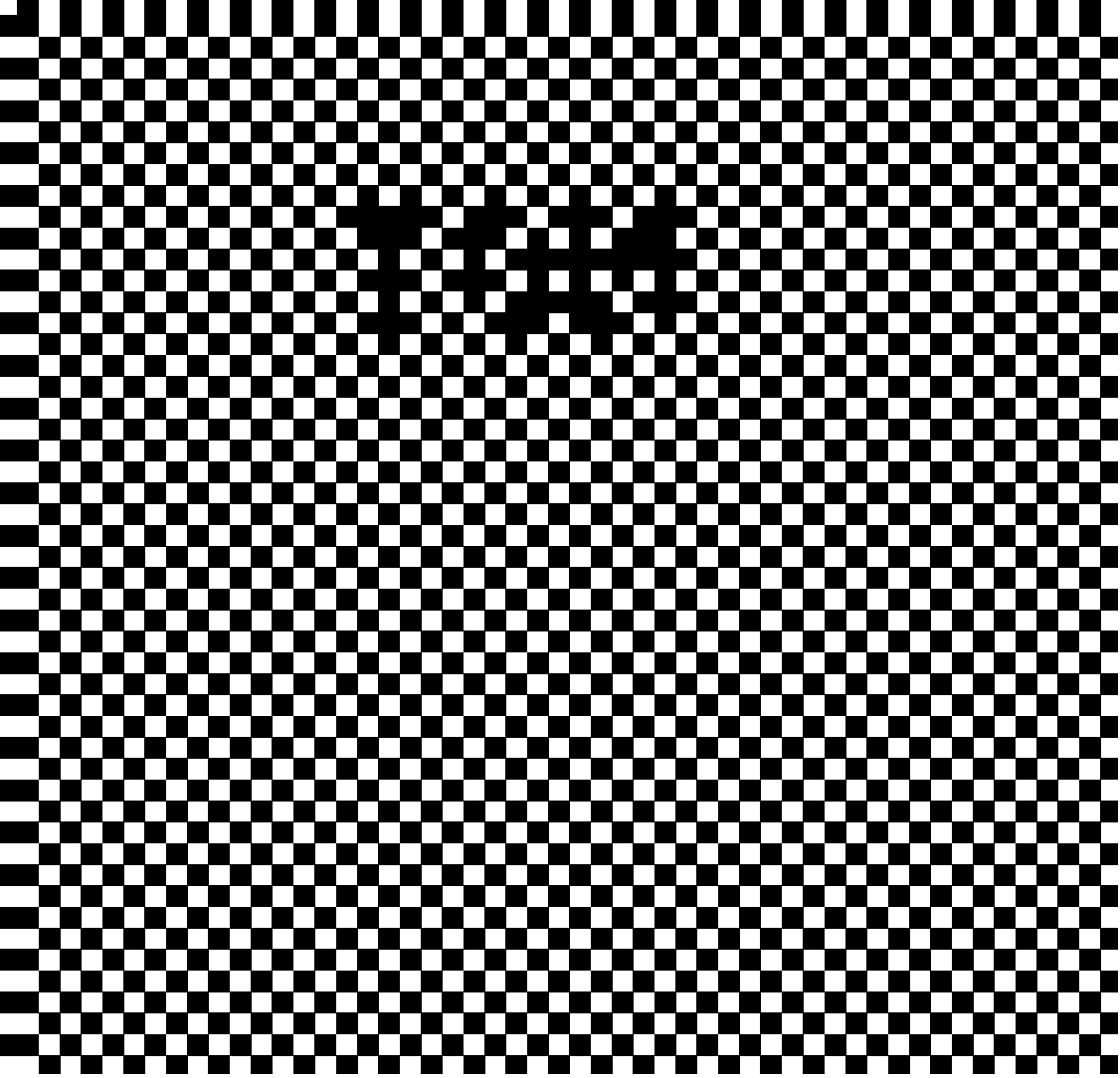
This license may undergo periodic modifications to incorporate improvements by its authors (instigators of the Copyleft Attitude movement) by way of new, numbered versions. You will always have the choice of accepting the terms contained in the version under which the copy of the work was distributed to you, or alternatively, to use the provisions of one of the subsequent versions.

SUB-LICENSING

Sub-licenses are not authorized by this license. Any person wishing to make use of the rights that it confers will be directly bound to the authors of the common work.

LEGAL FRAMEWORK

This license is written with respect to both French law and the Berne Convention for the Protection of Literary and Artistic Works.



FREEZE+PRESS freeze.sh/0007

```
:-------------------------------------------------------------:
|  the physical process of a liquid turning into a solid      |
|    the halting of all movement in a clever position         |
| a condition when computer software becomes unresponsive     |
:-------------------------------------------------------------:
```

PRINT DISTRIBUTION MADE POSSIBLE BY LAFKON PUBLISHING. THANK YOU!

freeze.sh/sponsorazine www.lafkon.net