7

Operationalising Digisprudence

> An insurmountable barrier between users and system programmers safeguards the computer's inalterable functions. Beyond this barrier, as in Kafka's story, a new barrier appears between the programmer and the programmer of the programming language who decides how the basic set of elements is to be designed, which rights and properties will be granted to whom, and which will be denied.[1]

The previous chapter set out the digisprudential affordances as an adapted representation of both the legal-theoretical principles of legality and legisprudence, and as normative criteria for code. This penultimate chapter discusses some practical ways to operationalise some elements of that framework. This is not an exhaustive survey of coding practices – that would require several volumes in its own right – but rather my goal is to draw attention to some points where operationalising the framework is especially important, as well as to existing approaches that could contribute to digisprudential legitimacy. There will without question be further avenues for exploration, some of which I highlight in the next and final chapter.

We have seen that the intent of the framework is to bind the design of code to underlying 'constitutional' principles, regardless of the artefact's ultimate commercial purpose. (A corollary being that those principles may logically prevent certain business models from being pursued.) We saw in Chapter 1 how this idea of the product designer being herself constrained by a prior set of 'constitutional' design choices can be conceptualised in the notion of the *programmer of the programmer* ('PoP').[2] This is an under-studied area in the legal literature, and part of the contribution here is to strengthen the practical connection between the PoP and its legal-theoretical analogues.

[1] C Vismann and M Krajewski, 'Computer juridisms' (2007) *Grey Room* 90, 101.
[2] Ibid. 100.

7.1 The Programmer of the Programmer

I referred in Chapters 1 and 2 to Vismann and Krajewski's discussion of the 'structural homologies' between computers and law. The vertical model of normative relationships (Figure 1.1 in Chapter 1[3]) hints at the analogy described there: the constitution binds the legislature, which promulgates norms that regulate the citizen, those norms being legitimated by the democratic process and the formal requirements of legality and legisprudential legitimation (including their contestability in court). This is the top-down aspect of the vertical model. By analogy, from a bottom up perspective there is the *programmer of the programmer* ('PoP'), which represents the software, tools, and development practices used by designers. These have the potential to impose 'constitutional' limits within the design environment, binding the product designer's coding activities ex ante. This possibility is particularly relevant to integrated development environments, which are the software applications that lie at the very heart of code production (I discuss these in more detail below). The 'legislative' work of the product designer can thus be constrained according to the (dis)affordances and inscriptions contained in that design environment, which, if defined according to the digisprudential perspective, can in turn mean that the normativities embodied in an artefact's code are legitimate from the outset.

The product designer is thus herself rendered a 'user', because despite the vast freedom she enjoys in defining her code's normativity, she is nevertheless constrained by prior design decisions made by notional PoPs and embodied in the tools of her trade: hardware, programming languages, and the software tools used to write new code. The parallel runs down to the fundamental level of the computer's architecture, where Vismann and Krajewski characterise the chip as a 'sovereign' and Intel (one of the world's largest processor manufacturers) as a 'legislator', by dint of the power they wield over the design of the internal rules, or 'instruction sets', of the processor.[4] This is the apotheosis of the PoP metaphor; the ultimate technical constitution lies in the low-level instructions defined physically in the very silicon of the chip. For Vismann and Krajewski, the PoP

> maintains the ultimate power because he or she, as the constructor of the programming language itself, defines what the 'normal' programmer, as a

[3] See 'Normative Relationships in Code and Law' in Section 1.4.

[4] Vismann and Krajewski (n 1) 96–7. See also FA Kittler, 'Protected mode' in J Johnston (ed.), *Literature, Media, Information Systems: Essays*, trans. S Harris (Psychology Press 1997) 162, suggesting that our conceptions of power should come not from analysing society but from examining chip architectures.

> user, will be able to do. Both types of programmers establish the conditions of using the computer, and, as such, they behave like lawmakers or, rather, *code*-makers.[5]

One can appreciate the implications of this 'meta-architecture', and how it mediates the work of the product designer. We can therefore conceive of the PoP not as an individual person or enterprise, but as the conditions of possibility that govern what the product designer can possibly do. The latter is situated within an assemblage of programming languages, a community with standardised practices and design patterns, and pre-existing libraries of code, all of which are to some extent constitutive of her work before she writes even a single line of her own code.

Viewed from this broader perspective, the PoP is a deeply normative force, operating at a 'constitutional' level within the design process. And, again, just as with production code itself, nothing is given; the conditions represented by the PoP are all to some extent contingent on design choices, which can themselves be guided through the support of certain values.

(a) From Primary and Secondary Rules to Primary and Secondary (Dis)affordances

Thinking normatively about the role of the PoP, we can consider how to leverage it to impose elements of the 'constitutional' framework of digisprudence on product designers working later in the production process. The idea is to push for 'legitimacy by design, by design', through the structuring, guiding, and restraining of product design practices according to the requirements and aims of digisprudence. This should be aimed for whatever the substantive purpose of the code being produced and whatever the underlying business model being pursued.

One might think of this in terms of Hart's primary and secondary rules. As we saw previously, primary rules are those that require a substantive behaviour (or forbearance) on the part of the addressee.[6] Secondary rules are those that define the conditions under which the primary rules can be created, changed, and adjudicated.[7] Secondary rules are thus ex ante and 'constitutional', defining how to create primary rules and the proper form that they should take.

The primary rules find their analogue in the (dis)affordances and inscriptions embodied in the design of the artefact, constraining and enabling the

[5] Vismann and Krajewski (n 1) 100 (emphasis supplied).
[6] HLA Hart, *The Concept of Law* (2nd edn, Clarendon Press 1994) 91–3.
[7] Ibid. 95–6.

Table 7.1 Hartian–legisprudential–digisprudential homologies

Hartian Norm	Legisprudential Locus	Digisprudential Actor
Secondary rule	Constitution binds the rule-maker	PoP implements primary digisprudential (dis)affordances/inscriptions in design environment
Primary rule	Legislature creates rules of conduct, subject to legisprudential legitimation	Product designer creates technological normativity, subject to constraining (legitimating) secondary (dis)affordances/ inscriptions in the design environment
–	Citizen is subject to legitimated text-based legal normativity	End-user is subject to legitimated technological normativity

behaviour of the end-user (this was the focus of Part I of the book). In the digisprudential context, we can envisage including in the design process secondary rules that constrain what primary (dis)affordances and inscriptions the designer may build into her product's code. This analysis suggests certain homologies between Hart's thesis, the legisprudential hierarchy, and digisprudence, which are set out in Table 7.1. The concept in the central column of a hierarchy of regulative force building up from a base 'constitutive' foundation maps onto the legisprudential model of legitimation discussed in Chapter 3.[8]

In terms of operationalisation, where the legislature is constrained by secondary rules and the legisprudential principles in traditional law-making, we can imagine in the design sphere the 'legislature' of the design process (including on a concrete technical level the integrated development environment, discussed below) being similarly constrained by secondary rules which guide what primary (dis)affordances and inscriptions can possibly be created there.

Assessing the embodiment of some of the secondary digisprudential affordances will involve qualitative judgements, for example whether a given delay is sufficient to enable comprehension, or the extent to which oversight is afforded. In addition to such qualitative questions, however, we can also envisage 'secondary' (dis)affordances/inscriptions, built into the very design environment itself, which guide the work of the 'designer-legislator' in her creation of primary (dis)affordances/inscriptions. The product designer thus becomes another regulatee, this time at the hand of the PoP. Substantive 'primary' (dis)affordances and inscriptions are aimed at the end-user, while the constitutional 'secondary'

[8] See 'Legalism According to Legisprudence' in Section 3.1.

(dis)affordances and inscriptions are aimed at the product designer. The latter operate to produce legitimate instances of the former.

In the following sections I discuss elements of the software development process that are appropriate targets for this kind of 'meta-normativity'. Development practice is a multifaceted and ever-evolving thing, so as suggested above the intention is not to set out a hard and fast roadmap for digisprudence. The topics selected here are, however, representative of the levels of the process at which the greatest impact is likely to be made in terms of legitimation, and are chosen in light of the need to take into account long-term trends in code production. I first consider the agile development process, then the software applications (integrated development environments) used to write code, followed by the interpretative affordances of code, facilitated by commentary, programming languages, and visual modelling.

7.2 Agile Development

A welcome shift in focus towards the production of code is beginning to emerge amongst legal scholars. An example is Gürses and van Hoboken's discussion of the 'agile' development methodology, which they describe as a 'paradigmatic transformation in the production of digital functionality'.[9] Although their primary focus is privacy and the production of platforms rather than individual artefacts, they acknowledge the 'wider societal implications of the agile turn',[10] and as we saw in Chapter 5 their concern about production is equally applicable to the more fundamental question of legitimacy.

According to the Agile Manifesto, agile development processes are characterised by a focus on end-users, continuous development and testing, collaboration, and response to change.[11] This approach contrasts with the 'waterfall' paradigm, dominant between the 1970s and 1990s,[12] which is built around discrete, sequential stages that have limited recursion and feedback between them.[13] The waterfall model is thus brittle: whereas the focus of the agile model is on producing modularised, working code as early as possible, with feedback being integrated as it is gathered throughout the

[9] S Gürses and J van Hoboken, 'Privacy after the agile turn' in E Selinger, J Polonetsky and O Tene (eds), *The Cambridge Handbook of Consumer Privacy* (Cambridge University Press 2018) 579.

[10] Ibid. 580.

[11] Beck K et al., 'Manifesto for agile software development' (2001) <https://agilemanifesto.org/> last accessed 4 March 2021.

[12] WW Royce, 'Managing the development of large software systems' in *Proceedings of IEEE WESCON* (Los Angeles 1970). See also Gürses and van Hoboken (n 9) 582.

[13] DA Norman, *The Design of Everyday Things* (MIT Press 2013) 234–5.

process,[14] the waterfall model relies on 'rigorously regimented practices, extensive documentation and detailed planning and management'.[15] Agile processes are cyclical and responsive, while waterfall processes move between predetermined phases that are less flexible vis-à-vis contingencies and feedback. By nature, agile processes also accelerate the code development process because kinks and problems tend to be identified and fixed 'on-the-fly', rather than waiting until later testing phases when more significant problems might be expensive and time-consuming to fix (and might therefore be quietly ignored).[16]

This idea of incremental cycles responsive to changing requirements fits well with a notion of a process involving continual assessment of legitimacy.[17] In that vein, the technology ethics thinktank doteveryone suggests augmenting agile cycles with anticipatory assessments of the potential consequences of design choices to enable the mitigation of problems during the design process.[18] This chimes with the idea of continually assessing code functionality according to whether and how it reflects the digisprudential affordances. As with Wintgens's plotting of proposed legislative norms that we saw in Chapter 6, an element of code functionality can be assessed according to its embodiment and balancing of the affordances. Taking a digisprudential 'stance' can continually adjust the design throughout cycles of agile development, refining it towards greater legitimacy. This is important, since design processes are often long and complex and cannot be neatly compartmentalised as in the waterfall model. The common enjoinder that 'by design' of whatever form (legitimacy, privacy, or legal compliance more generally) take place during the 'early stages of the process' is thus insufficient;[19] the proper

[14] T Hoeren and S Pinelli, 'Agile programming – introduction and current legal challenges' (2018) 34 *Computer Law & Security Review* 1131, 1132.

[15] Gürses and van Hoboken (n 9) 582.

[16] See for example 'The Lean Startup | Methodology' <http://theleanstartup.com/principles> last accessed 4 March 2021, promoting a cyclical 'build–measure–learn' approach to code development. See also E Luger and M Golembewski, 'Towards fostering compliance by design; drawing designers into the regulatory frame' in M Taddeo and L Floridi (eds), *The Responsibilities of Online Service Providers* (Springer 2017) 296.

[17] For a practical discussion making this point in relation to Privacy by Design, see AC García et al., 'PRIPARE privacy- and security-by-design methodology handbook' (EU FP7 2015) 103 *et seq*.

[18] S Brown, 'An agile approach to designing for the consequences of technology' *doteveryone* (13 February 2019) <https://medium.com/doteveryone/an-agile-approach-to-designing-for-the-consequences-of-technology-18a229de763b> last accessed 4 March 2021.

[19] L Diver and B Schafer, 'Opening the black box: Petri nets and privacy by design' (2017) 31 *International Review of Law, Computers & Technology* 68, 76; Gürses and van Hoboken (n 9) 592. On the inadequacy of a 'checklist' approach to privacy by design, see S Gürses,

embodiment of the value-based affordances I have described requires continual assessment and reassessment throughout the process, which the cyclical agile methodology can help facilitate.

7.3 Integrated Development Environments

Returning to the discussion above of primary and secondary (dis)affordances and inscriptions, one place in which this concept might be implemented in a technically robust way is the integrated development environment (IDE). In contemporary development the text of code is written in an IDE, which compiles it into object code executable by the machine.[20] This fundamental function has over time been augmented by further features designed 'to assist the software lifecycle process'.[21] They do this in myriad ways (too many to fully canvass here), but some important points to note are that applications for writing code vary in complexity and sophistication, from those that are simple text editors requiring additional software (a compiler) to produce executable code, to more powerful suites that include compilers, debuggers, build automation tools, version control, tools for highlighting syntax and auto-completing code statements, et cetera. Most IDEs can detect problems in source code, including syntax errors identified according to the requirements of the programming language being used, naming mistakes (incorrect variable or method names), logically impossible statements, and other incorrect programming 'grammar' that will cause errors fatal to execution. More sophisticated IDEs auto-complete formulaic expressions in the relevant programming language, and will keep track of a project's structure, suggesting relevant connections between code modules as the designer is working (this is termed 'intelligent code completion'). It is also possible to suggest points at which the code might be documented, or explanatory comments added (see the next section),[22] to enable other developers to interpret and understand what the code is designed to do. We saw in Chapter 3 the importance of being able to interpret a code-based rule; this is one means by which transparency could be implemented within the IDE, the software requiring the designer to include explanatory comments.

C Troncoso and C Diaz, 'Engineering privacy by design' (2011) 14 *Computers, Privacy & Data Protection*.

[20] In practice development tool chains often separate the IDE from the compiler.

[21] A Abran et al., *Guide to the Software Engineering Body of Knowledge (SWEBOK)* (IEEE Computer Society and Angela Burgess 2004) 10–11.

[22] In modern sophisticated IDEs this kind of functionality is included for documenting traditional code. See for example Microsoft, 'XML documentation (Visual C++)' (Microsoft 2016) <https://docs.microsoft.com/en-us/cpp/ide/xml-documentation-visual-cpp> last accessed 4 March 2021.

One can appreciate from this extremely brief survey of IDE functionality how the particular features of the software being used to produce code – its affordances – play a central role in structuring how the product designer does her job.[23] As an element of the PoP, the IDE structures and even dictates the scope of action of the product designer. The (secondary) affordances of the IDE might therefore be designed to discourage or prevent not only logical or aesthetic infelicities in the product designer's code – as the features just described already do – but also the writing of code that fails to meet digisprudential standards.

Some initiatives already exist that augment the purely practical or logical aspects of the programming assistance that IDEs provide. One example is the use of machine learning to provide code completion suggestions that are derived from the code of existing, third-party projects. Here the 'wisdom of the crowd', embodied in the code of those projects, constitutes a dataset from which the algorithm provides 'predictions' that go beyond the static syntax-derived suggestions provided by traditional intelligent code completion.[24] Microsoft's Intellicode system, for example, uses as training data the open source code from popular projects on the company's GitHub platform. When enabled, the code recommendations in Microsoft's Visual Studio IDE[25] are thus based in part on real-world projects, which will inevitably include aesthetic and value-based code design choices.

This phenomenon can be looked at as both a risk and an opportunity; the intention behind Intellicode is of course to encourage 'best practice' in the production of new code, the assumption being that the most popular projects on GitHub represent such practice by dint of their prominence. It is of course questionable whether popularity is the appropriate measure for the quality of code, assuming a quantitative measure is even possible where the medium has so much inherent normative power. The risk, then, is that systems like Intellicode in fact perpetuate bad practice, reflexively setting in train path-dependent 'habits of mind'[26] in the designers who rely on its computationally legalistic suggestions.

[23] RB Kline and A Seffah, 'Evaluation of integrated software development environments: Challenges and results from three empirical studies' (2005) 63 *International Journal of Human-Computer Studies* 607.

[24] M Bruch et al., 'IDE 2.0: Collective intelligence in software development' in *Proceedings of the FSE/SDP Workshop on Future of Software Engineering Research – FoSER '10* (ACM Press 2010).

[25] Microsoft, 'Visual Studio IntelliCode' (Microsoft 2018) <https://visualstudio.microsoft.com/services/intellicode/> last accessed 4 March 2021.

[26] P Graham, 'Beating the averages' (2003) <http://www.paulgraham.com/avg.html> last accessed 4 March 2021; J Weizenbaum, *Computer Power and Human Reason: From Judgment to Calculation* (Freeman 1976) 102–4.

A more desirable possibility is that digisprudentially legitimate code might have a positive effect on such data-driven code completion systems. Code that embodies the sorts of legitimacy-creating practices I am discussing might, if promulgated widely, come to be reflected in the suggestions provided by systems like Intellicode, in turn contributing to a greater standardisation of these concepts at the level of source code.

(a) Code Verification versus Legal Proof: Justice being Seen to be Done

The importance of transparency and contestability has led to increased interest in formal methods that not only guarantee a certain outcome in the code, but also allow relevant parties to see the 'why' of the code's behaviour.

From that perspective, the goal is to verify in advance that a system will operate according to a predefined set of characteristics (this is of course the core of code's ruleishness). Although this is an important development from the perspective of legal compliance, it is not necessarily sufficient to afford true contestability (recall the distinction made in Chapter 1 between compliance by design and legitimacy[27]). Legal proof and formal verification of code may have similarities, but there are also crucial differences. In practice, a legal proof contains not just the empirical evidence and legal sources necessary to construct a valid syllogism,[28] but also evidence of procedural propriety as a concern separate from the validity of the syllogism's conclusion. In other words, the code may be compliant, but in order for legitimate contestability properly to be afforded, it must provide evidence of 'due process' in order for the matter to be proven according to the relevant legal standard.[29] Merely 'doing justice' is insufficient; due process requires evidence of the procedure that was followed; justice must not simply be done, it must be seen to be done. Apprehending the right person in a criminal case is only one part of the equation; if their confession is obtained without legal representation then it is *de facto* illegitimate, regardless of any truth it might contain. Due process under the law thus takes what might have happened to be as important as what actually happened. Evidence of this proof must be communicable in a specific way; legitimacy of the legal process requires a form of evidential transparency that goes beyond merely telling the 'whole truth' in a given instance; it requires that this truth be demonstrated to external observers (including

[27] See 'Why Not "Compliance by Design"?' in Section 1.4.

[28] On which, see N MacCormick, *Rhetoric and the Rule of Law: A Theory of Legal Reasoning* (Oxford University Press 2005) chapter 3.

[29] For a salient analysis in the US context, see DK Citron, 'Technological due process' (2008) 85 *Washington University Law Review* 1249.

the courts) in a form that is intelligible to them. In institutional law this is of course achieved by means of a public trial.

Affording true contestability, then, requires enabling the end-user to detect code conditions that are susceptible to contest (transparency of operation, discussed in the previous chapter). Crucially, however, it can also be interpreted as requiring the demonstration of how the code was developed in the first place.[30] In a sense, documenting the design process (particularly the outcomes of agile cycles discussed above) will in itself provide this, but it is also feasible to integrate a measure of this kind of 'due process' into the IDE by bridging the gap between human comprehension and machinic execution, guided by the goal of ensuring legally relevant intelligibility.

7.4 Code and Natural Language

At the level of its source, code is a bi-directional text: it is both a set of instructions for the computer to perform, and a document for the human interpreter that explains what the machine will do upon execution. This central characteristic of code as simultaneously performative and documentary separates it from most other types of text, in degree if not category[31] (legal texts are of course also performative, albeit within the constraints of a text-based normative order). On the one hand, we can think of what code affords the end-user at the interaction level, described in Chapter 2.[32] But on the other, we can also think of what code affords, and to whom, when it is looked at as a text. The digisprudential affordances are not concerned only with this first level of communication, namely what will ultimately be afforded to the end-user in operation; they also, necessarily, require the code-as-text to afford certain things. The production of 'interactive' code, or 'architecture', is by definition dependent on the writing of code's text, and so some engagement with that text is necessary to gain a holistic sense of what the code does and does not do, and what it does and does not afford.

In code of any real complexity, the documentary function of the medium is crucial for understanding what the system does or is intended to do. This is particularly true where more than one designer is involved in its creation: the ability to understand what a programmer 'meant' by choosing one particular

[30] TJ Bench-Capon and FP Coenen, 'Isomorphism and legal knowledge based systems' (1992) 1 *Artificial Intelligence and Law* 65, 70–1.

[31] P Swartz, 'How do programs mean?' in *Division III: Essays in Programs as Literature* (Hampshire College 2007); I Arns, 'Code as performative speech act' (2005) 4 *Artnodes*.

[32] Lessig's 'code as law' analysis is focused primarily on this level. See L Lessig, *Code: Version 2.0* (Basic Books 2006) *passim*.

approach rather than a reasonable alternative can be crucial to a successful implementation, even within a small team (and all the more across continents and potentially significant stretches of time, where third-party open source code is being used).

The documentary function of code has two basic aspects. The first comes in the form of *ad hoc* comments included alongside the executable code. These are ignored by the machine, but can explain to the human reader what that part of the code is intended to do, why the designer chose this particular approach, or for example that some element of code is a temporary 'hack'[33] that works ostensibly but will need future revision. These comments are like notes in the margin of a book; they are intended for a human reader and can include however much detail the designer wishes, without this in any way affecting the nature or execution of the code statements they appear alongside. Providing commentary in this way is a powerful means of explicating what code does, although a corollary of its *ad hoc* nature is that the designer might fail to provide it, or might mischaracterise what the code in fact does because of misunderstanding or the desire to obfuscate.[34] It is possible to some extent to bridge the 'isomorphic gap' between natural language commentary and executable statement; I will discuss some potential approaches below but for now we turn to the second aspect of code's documentary function, which flows from the programming language itself.

(a) The Interpretative Affordances of Programming Languages

While *ad hoc* comments need not have any isomorphism with the code they accompany, it is also possible for the code to speak for itself, through the programming language in which it is written and the structures and functions that constitute its 'grammar'. Whether or not these will afford intelligibility will vary according to the language: on the one hand, some languages aim as far as possible to promote human understanding over computer 'cognition' (I discuss these below), while many so-called esoteric languages are explicitly designed to be as difficult to understand as possible, despite being, from the

[33] P Swartz, 'The hack as form' in *Division III: Essays in Programs as Literature* (Hampshire College 2007).

[34] The intelligibility of comments can facilitate differing ends: at the centre of the 'climategate' controversy were comments in climate modelling code that appeared to suggest to non-experts (specifically James Delingpole) that the code had been written intentionally to produce false data in support of the consensus on anthropogenic climate change. In this case, a little knowledge was a dangerous thing. For a full account, see MC Marino, *Critical Code Studies* (MIT Press 2020) chapter 4.

perspective of the machine, every bit as intelligible as their human-friendly counterparts.[35]

Programming languages that are designed for intelligibility will communicate their meaning more clearly and thus afford a wider notion of transparency at the level of the text,[36] which can have positive reflexive effects during production that would be a valuable addition to the forms of operational transparency previously described. By using a more intelligible programming language, the designer is in effect anticipating the affordance of contestation, and particularly its evidential dimension. In this way, the PoP plays a role in this aspect of digisprudential legitimacy through the design of the language.

Programming languages, as tools that are themselves designed by the PoP, have interesting properties as compared with human languages. Whereas the grammar of a natural language is an ever-changing crystallisation of its use in practice,[37] for programming languages this arrangement is almost entirely upended.[38] As participants in the hermeneutic development of a natural language we all to some degree contribute to the evolution of its grammar; the same cannot be said for the designer in respect of the language in which she is writing her code. There, the 'speaker' of the language is entirely constrained by the syntax imposed ex ante by the PoP; she has no input into the specification of its rules. Furthermore, a given statement in code is entirely ineffectual if it fails to meet the precise requirements of that predetermined grammar – there can be no ex post reinterpretation to make up for infelicities of expression.[39] In that case, the code will simply not execute. (Any reader who has programmed will know all too well the experience of staring hopelessly at lines of code that will simply not execute, unable to identify precisely where the error lies.)

[35] An example of such an esoteric language, whose name sums up this ethos perfectly, is Brainfuck. For an interesting discussion of the aesthetic qualities of such languages, see M Mateas and N Montfort, 'A box, darkly: Obfuscation, weird languages, and code aesthetics' in *Proceedings of the 6th Digital Arts and Culture Conference* (IT University of Copenhagen 2005).

[36] P Swartz, 'A tower of languages' in *Division III: Essays in Programs as Literature* (Hampshire College 2007) 118–19.

[37] P Ricoeur, *Interpretation Theory: Discourse and the Surplus of Meaning* (TCU Press 1976) chapter 1.

[38] WJ Ong, *Orality and Literacy: The Technologizing of the Word* (3rd edn, Routledge 2012) 7.

[39] Swartz, 'How do programs mean?' (n 31) 81–4.

(b) The Linguistic Relativity of Programming Languages

The 'grammar' and design conventions of a programming language are particularly responsible for framing the solution to a given problem.[40] Nearly all modern languages are Turing complete, meaning they can perform the same set of atomic calculations, regardless of the higher-level abstractions they include to make them easier for designers to use in practice. Despite this, some languages are designed for, or are especially appropriate for solving, particular kinds of problem.[41] This is down to the amount and forms of abstraction they include, for example predetermined functions that achieve particular computational goals in a single black-boxed step that does not require further bespoke coding by the designer. These 'off-the-shelf' functions are integrated into the core grammar of the language, somewhat akin to idioms in natural language that 'formalise' particular meanings in a single phrase that those familiar with the language can understand.

Setting aside for a moment the factors that make a particular programming language more fashionable at a given point in history, the choice to use one language over another can in principle be tied to the usefulness of the abstractions it provides, and how these 'fit' the designer's understanding of the problem at hand.[42] Of course, the question of what constitutes the 'best' solution will be contested, but my argument is that any answer should include the goal of mitigating computational legalism.[43] Programming languages are, as Graham puts it, 'not just technologies, but habits of mind'.[44] These habits

[40] Marino (n 34) 144.

[41] Ibid. 124–5. This often results in (heated) debates about the relative merits of languages, an example being the rivalry in the statistics/machine learning world between proponents of R and Python.

[42] See Graham's discussion of the 'power' of different programming languages (n 26). The W3C recommends the use of the 'least powerful language' suitable for a given task. See W3C, 'The rule of least power' (W3C 2006) <https://www.w3.org/2001/tag/doc/leastPower.html> last accessed 4 March 2021. This latter suggestion reflects the ethos behind designing for modularity along 'tussle lines', recommended in DD Clark et al., 'Tussle in cyberspace: Defining tomorrow's Internet' (2005) 13 *IEEE/ACM Transactions on Networking (ToN)* 462 and discussed in detail in the previous chapter.

[43] This is far from the norm. Amongst programmers, the 'best' solution (assuming it works, *ceteris paribus*) is usually the 'most efficient'. I argued against this framing in the previous chapter's discussion of the affordance of delay. For a discussion relating to legal technologies, see L Diver, 'Computational legalism and the affordance of delay in law' (2021) 1 *Journal of Cross-disciplinary Research in Computational Law* <https://journalcrcl.org/crcl/article/view/3> last accessed 19 April 2021.

[44] Graham (n 26). See also Swartz, 'A tower of languages' (n 36).

develop over time, based in part on the kinds of problem being solved. Given that a designer will usually become 'fluent' in only a handful of languages, she will begin to see the problems she is charged with solving through the lens of those languages, with the affordances of their syntax, functions, and data structures coming to frame the solutions she conceives.[45]

This representational idea relates to the notion of linguistic relativity, the theory that the language we use mediates our situatedness within empirical reality.[46] Much like a native tongue, the language we are most familiar with frames our experience; the available vocabulary and grammatical structures form a lens through which our worlds are constructed, culture shaping language and vice versa.[47]

Whatever the merits of the theory of linguistic relativity in the context of linguistics, from a pragmatic perspective it is no stretch to claim that programming languages structure how designers approach the problems they are tasked with solving.[48] The ultimate design of a given code's ontology will be affected to whatever degree by the data structures the programming language 'naturally' accommodates (for example arrays, data frames, matrices), while the rules that process the system's inputs and outputs will be affected by the grammar and functions that the language provides in the first instance. An inexperienced or unconfident programmer will avoid tackling complex functionality that the language does not accommodate by default, which in turn will affect her approach to building her application.

Chen's discussion of linguistic relativity in programming languages is instructive here. For him, once the practice of writing code has begun, the 'boilerplate and design patterns' of a given programming language are internalised as 'unconscious and automatic idioms', ready to be 'regurgitated on demand'.[49] In a study of the standard 'split, apply, combine' task in data science, Chen illuminates linguistic relativity as between the R, MATLAB, APL, and Julia languages (the latter of which he helped design). Depending on the language, the higher-order functions that were immediately available to the programmer varied, such that in some languages the tools for solving the problem were very much more 'ready-to-hand' than in others in which the

[45] For an early study confirming this tendency, see RL Wexelblat, 'The consequences of one's first programming language' (1981) 11 *Software: Practice and Experience* 733.

[46] BL Whorf, 'Science and linguistics' in JB Carroll (ed.), *Language, Thought, and Reality: Selected Writings of Benjamin Lee Whorf* (28th edn, MIT Press 2007).

[47] E Sapir, 'The status of linguistics as a science' (1929) *Language* 207.

[48] Swartz, 'A tower of languages' (n 36) 105 *et seq*.

[49] J Chen, 'Linguistic relativity and programming languages' (2018) arXiv:1808.03916 [cs, stat], 2 <http://arxiv.org/abs/1808.03916> last accessed 4 March 2021.

problem required sustained attention and the coding of a bespoke solution. Thus, where some languages have relatively constrained data structures and prefigured 'habits', others are more flexible in their generative possibilities. The designer is necessarily situated within a set of habitual and community practices that surround the language and its associated libraries and tools, and thus will to a greater or lesser degree be guided in her understanding of the solution to the programming challenge she is faced with.[50]

In the end, then, the programming language wields significant power over the product designer, framing her actions from the outset.[51] From the perspective of digisprudence, the design of the language ought to reflect values of legitimacy, facilitating their embodiment in the code written using them. Proposals already exist for such explicitly value-driven programming languages, in the sense that the values are clearly and intentionally embodied in their design, rather than passively or unthinkingly represented. These proposals have been aimed for example at representing feminist perspectives[52] or the particularities of non-Western cultures, as in 'ethnoprogramming'.[53] An example of the latter is قلب ('heart'), a language that poses a vivid challenge to the Anglo-centricity of contemporary programming languages, the vast majority of which use English verbs and nouns.[54]

(c) Describing Code Isomorphically

Returning to code's bi-directionality, approaches exist that intertwine the documentary and performative roles of the text. A prominent, early example

[50] Ibid. 8. See also C Thompson, *Coders: Who They Are, What They Think and How They Are Changing Our World* (Pan Macmillan 2019) chapter 1.

[51] Weizenbaum (n 26) 102–3.

[52] Cf. Schlesinger's proposals for a feminist programming language. See A Schlesinger, 'Feminism and programming languages' *HASTAC* (26 November 2013) <https://www.hastac.org/blogs/ari-schlesinger/2013/11/26/feminism-and-programming-languages> last accessed 4 March 2021. For a perspective on how masculinity has structured the practices of science and software production, see T Estrin, 'Women's studies and computer science: Their intersection' (1996) 18 *IEEE Annals of the History of Computing* 43; P Swartz, 'White boys' code' in *Division III: Essays in Programs as Literature* (Hampshire College 2007) 32 *et seq*.

[53] O Laiti, 'The ethnoprogramming model' in *Proceedings of the 16th Koli Calling International Conference on Computing Education Research* (Association for Computing Machinery 2016). Laiti's work stems from the broader concept of *ethnocomputing*, which aims to challenge the positivism dominant in (Western) computer science. See M Tedre et al., 'Ethnocomputing: ICT in cultural and social context' (2006) 49 *Communications of the ACM* 126.

[54] R Nasser, *Nasser/---* (Github 2020) <https://github.com/nasser/---> last accessed 4 March 2021. On English as the *lingua franca* of programming, see NK Hayles, 'Print is flat, code is deep: The importance of media-specific analysis' (2004) 25 *Poetics Today* 67, 79; Marino (n 34) 151 *et seq*.

is Knuth's Literate Programming paradigm and its WEB language, which tightly weave together executable code and commentary in a single file, from which both isomorphic documentation and the executable code can be generated.[55] A more recent incarnation is the Jupyter Notebook, a self-contained 'document' that allows the designer to combine executable code with 'interactive widgets, plots, narrative text, equations, images, and video'.[56] These approaches are motivated by the notion that programming ought to serve human ways of thinking, rather than imposing computational structures upon the latter. This means, at least notionally, that culture can shape the code (mediated by the language) rather than the converse, which in turn shines a normative light on the linguistic relativity of those languages. We see this aim reflected throughout the history of programming in the design of business-oriented languages such as COBOL and its ancestor FLOW-MATIC,[57] as well as languages like LOGO, which aims to reflect the programmer's embodied perception,[58] and Inform 7, which uses entirely natural language sentences for its executable expressions.[59] The ends of a given language are expressed in its vocabulary and grammar; these might conceivably reflect the aim of legitimacy and legitimation. And even if the underlying language still facilitates the building of legalistic structures (as invariably it will, if it is Turing complete), the IDE might provide hints or even mandates that encourage new forms of 'best practice' that can avoid them.

Behaviour-Driven Development

Beyond the programming language itself, Behaviour-Driven Development (BDD) is an approach that focuses on the point of practical implementation, bridging the isomorphism gap between *ad hoc* comments and what the code in fact will do. Like the Petri net visual model discussed below, BDD facilitates isomorphism between code and a representation that is intelligible to non-technologists, in this case a natural-language textual description. BDD's originator North describes it as an 'outside-in' methodology, starting from a set of desired outcomes and evolving towards the code features that

[55] DE Knuth, 'Literate programming' (1984) 27 *The Computer Journal* 97.

[56] 'What is the Jupyter Notebook?' (27 March 2019) <https://jupyter-notebook.readthedocs.io/en/latest/examples/Notebook/What%20is%20the%20Jupyter%20Notebook.html> last accessed 4 March 2021.

[57] Marino (n 34) chapter 5.

[58] See for example S Papert, 'Different visions of LOGO' (1985) 2 *Computers in the Schools* 3. See also Estrin (n 52) 45. The notion of the PoP shaping the designer's perception shaping the end-user's perception is clear.

[59] 'About' *Inform 7* <http://inform7.com/about> last accessed 4 March 2021.

implement them.[60] Although usually aimed at bridging the domains of business requirements and code development, the quasi-isomorphism of BDD means it can operate as a post hoc evidentiary mechanism as much as a means of developing ex ante design specifications.

BDD uses natural language[61] templates for defining features the code is required to implement. These are combined with IDE tools that generate both the framework of code statements from those specifications and the 'unit tests', or granular checks of the output of discrete sections of code, that verify that they behave as expected. Code features are defined using natural language, making them intelligible for evidentiary purposes (and indeed feasibly for end-users). Here is an example specification of a shopping basket in an online application:

```
Feature: Online shop basket
  In order to buy products
  As a customer
  I need to be able to put interesting products into a
    basket
Rules:
  Delivery for basket under £10 is £3
Scenario: Buying a single product under £10
  Given there is a "Product X", which costs £5
  When I add the "Product X" to the basket
  Then I should have 1 product in the basket
  And the overall basket price should be £8
```
[62]

The IDE parses the keywords in the template (`feature`, `in order to`, `as a`, `I need to`, `rules`, `scenario`, `given`, `when`, and `then`) and generates the necessary code functions. These have a specificity that encourages the

[60] Dan North & Associates, 'What's in a story?' (Dan North & Associates, 11 February 2007) <https://dannorth.net/whats-in-a-story/> last accessed 4 March 2021. This mirrors the concept of 'bottom-up' programming, which, like agile development cycles, is about 'following the path of the program [system] as it develops'. See Swartz, 'White boys' code' (n 52) 36.

[61] Known as a 'ubiquitous language', or a 'business readable domain specific language'. See M Fowler, 'Business readable domain specific language' *martinfowler.com* (15 December 2008) <https://martinfowler.com/bliki/BusinessReadableDSL.html> last accessed 4 March 2021.

[62] This and the next example are adapted from 'Behat documentation' *Behat* <http://docs.behat.org/en/latest/quick_start.html> last accessed 4 March 2021. Behat is a set of tools for implementing BDD in the PHP programming language.

designer to write modular code, which in turn facilitates more cyclical testing and verification. For example:

```
/**
* @Given there is a(n) :arg1, which costs £:arg2
*/
public function thereIsAWhichCostsPounds($arg1, $arg2) {
  [the implementing code goes here]
}
```

One can see how the function name ('`thereIsAWhichCostsPounds`') has been automatically generated from the natural language '`Given`' line in the feature description. Within the curly braces – `{` and `}` – the designer writes the code that corresponds to that precise element of functionality, and nothing more (the automated inclusion of only two *arguments* `$arg1` and `$arg2` – values passed into the function for processing – limits the scope of what this function can feasibly be written to perform). This discreteness of functionality is ideal for granular testing and for achieving the aim of modularity discussed in the previous chapter. The goal of BDD, then, is to achieve 'living documentation', both intelligible to non-developers and simultaneously isomorphic with the underlying instrumentality of the code.

Interpreting Code as a Visual Model

If BDD is concerned with textual isomorphic descriptions of code, the Petri net is one approach to visual description. Dating from 1962,[63] the Petri net is a standardised formal modelling approach for representing arbitrary processes in terms of 'states' and the 'transitions' between them. Petri nets have been applied in many domains, not least in the modelling of legal provisions and processes.[64] The nets are commonly used in the early stages of the design of code to visually map the changing states of the system over time. Despite their graphical appearance and apparent simplicity, the temporal flow of a Petri

[63] CA Petri, *Kommunikation Mit Automaten* (PhD thesis, University of Bonn 1962) <http://epub.sub.uni-hamburg.de/informatik/volltexte/2011/160/> last accessed 4 March 2021.

[64] J Freiheit et al., 'Lexecute: Visualisation and representation of legal procedures' (2006) 3 *Digital Evidence & Electronic Signature Law Review* 19; JA Meldman, 'A Petri-net representation of civil procedure' (1977) 19 *Idea* 123; JA Meldman and AW Holt, 'Petri nets and legal systems' (1971) 12 *Jurimetrics Journal* 65. More recent work has used Petri nets to model normative relationships in law, with a view to aligning norms with technical implementation in the way outlined below. See for example G Sileno, A Boer and T van Engers, 'Towards a representational model of social affordances from an institutional perspective' in *Proceedings of the Workshop Computational Social Science and Social Computer Science: Two Sides of the Same Coin* (Institute of Advanced Studies, University of Surrey 2014).

net can be both easily simulated and formally verified. This means it can be mathematically proven whether or not the system enters a given state, and the conditions under which this takes place.[65] Petri nets in a sense facilitate the 'live documentation' of the system, describing the functionality of the code in a way that is both intelligible to non-technologists but that is also isomorphic with the concrete behaviour of the code.[66] Existing research has demonstrated the automated generation of Petri nets from object-oriented source code[67] as well as (contrariwise) the automated generation of code from Petri models of intended functionality.[68] The validation and certification affordances of Petri nets through formal proofs and reachability analysis[69] mean we can be sure of isomorphism between the code and the net, thus making the graphical representation a potentially valuable evidential tool for making intelligible the concrete behaviour of the code.[70]

The states and transitions in the model are represented by circles and rectangles, respectively. These are connected with arcs (arrows) that represent the flow of the process, which at any given moment is represented by the distribution of 'tokens' across the model's states. These four basic elements (states, transitions, arcs, and tokens) are the essence of all Petri nets (Figure 7.1).

A state containing a token (a dot) currently 'holds'. Multiple states can lead to, or from, a given transition, and they can hold simultaneously. When a transition fires, all the states leading to it will lose x tokens, and all the states leading from it will gain y tokens, where x and y correspond to the numerical weightings alongside each of the relevant arcs (the default being one). A transition can only fire – and will always fire – where the number of tokens in its preceding state(s) is greater than or equal to the weighting of the relevant arc. This is demonstrated in Figure 7.2, where the transitions T_1 and T_2 are

[65] Both viewed of course from within the limits of the system's ontology – but that indeed is precisely the point; to be able to contest that ontology and to argue why it is limited in ways that are unlawful.

[66] B Lin, 'Software synthesis of process-based concurrent programs' in *Proceedings of the 35th Annual Design Automation Conference* (ACM 1998); SM Shatz and WK Cheng, 'A Petri net framework for automated static analysis of Ada tasking behavior' (1988) 8 *Journal of Systems and Software* 343.

[67] Lin (n 66); Shatz and Cheng (n 66).

[68] KH Mortensen, 'Automatic code generation method based on coloured Petri net models applied on an access control system' in M Nielsen and D Simpson (eds), *Application and Theory of Petri Nets* (Springer 2000).

[69] Diver and Schafer (n 19) 82–3.

[70] K Salimifard and M Wright, 'Petri net-based modelling of workflow systems: An overview' (2001) 134 *European Journal of Operational Research* 664, 667. This can be contrasted with other software modelling tools that are not necessarily isomorphic, for example entity-relationship diagrams.

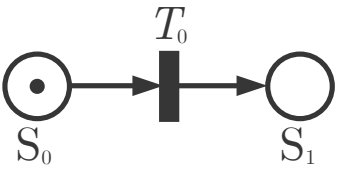

Figure 7.1 A basic Petri net

competing, with T_2 'winning' because the two states leading to it have the requisite number of tokens to trigger that transition.

This allows for control over the flow of the net, as tokens are distributed across the net according to the outcomes of prior transitions. This limited semantics enables complex processes to be simplified into graphical representations without losing formal validity.[71]

Mechanistic elements of complex processes can be abstracted into 'subnets' and then subsequently into transitions, thus mirroring the fundamental concept of abstraction in object-oriented programming.[72] Recursive abstraction of this kind allows for the modelling of even very complex systems, whilst simultaneously enabling the interpreter to drill down into the particulars of the code's logic as required. It can be appreciated how these representations might be useful from an evidential perspective, should the code ultimately be contested in court.

I have demonstrated the normative complexity that can be represented by Petri nets in prior work with Burkhard Schafer.[73] For example, the net in Figure 7.3 shows a model of Article 8 of the Data Protection Directive, precursor to the GDPR, which concerns the processing of special categories of data.

In principle, such a model of a legal provision could be interfaced with an abstracted model of a code system, providing a way of communicating between the states generated, and required, by each, as illustrated in Figure 7.4.

As one can appreciate from this simplified example, in order for the code to traverse between states S_0 and S_4, it must first pass the test in the legal

[71] For a more detailed discussion of Petri nets' application in the legal domain, see Diver and Schafer (n 19). For a theoretical background, see either Petri's doctoral thesis (Petri (n 63)) or T Murata, 'Petri nets: Properties, analysis and applications' (1989) 77 *Proceedings of the IEEE* 541.

[72] Bench-Capon and Coenen (n 30) 72.

[73] Diver and Schafer (n 19) 77 *et seq*.

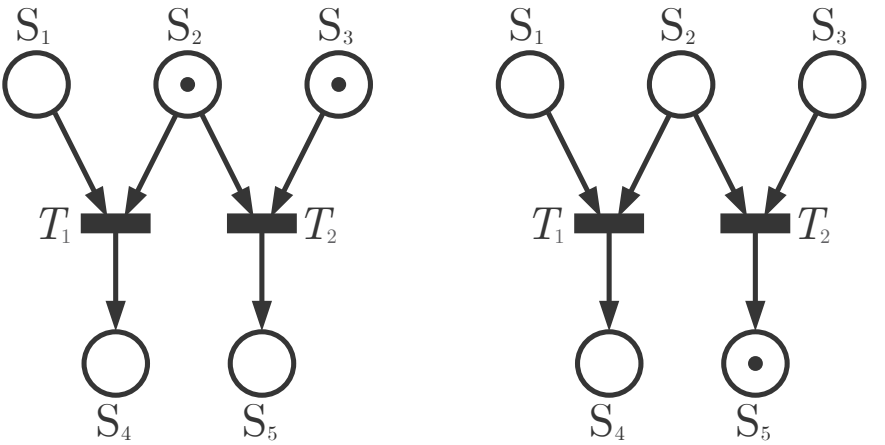

Figure 7.2 Competing transitions

net (left), which is itself contingent on an input from a sub-element of the software net. The idea is that the legal net will permit the code to 'continue' (that is, reach state S_4) only if there is some other condition in place that demonstrates the existence of a legally required state. The discussion of the model above in the original article envisages a registration form that collects sensitive personal data (in that case, the end-user's ethnic origin). This fact was represented by S_2, which when set thus communicates to the legal net that a special category of data was being processed, which in turn means one of the Article 8(2) exceptions must apply for processing to be lawful.[74]

Of course, these latter examples concern compliance with the substantive law; in other words, they are about 'compliance by design'. Nevertheless, the very existence of the model as a form of documentation demonstrates the second aspect of contestability I have been discussing. Whether or not the formal verification of the model results in compliance by design (indeed, legal compliance may not be the aim), the model itself affords contestability by providing intelligible evidence of how the code operates. One can appreciate the flexibility and abstraction of this kind of approach, and its ability to model and communicate in a single representation more than one aspect of the code.

[74] For a more in-depth discussion, see ibid. 79 *et seq*. For a more sophisticated example of this kind of approach, see Sileno et al. (n 64).

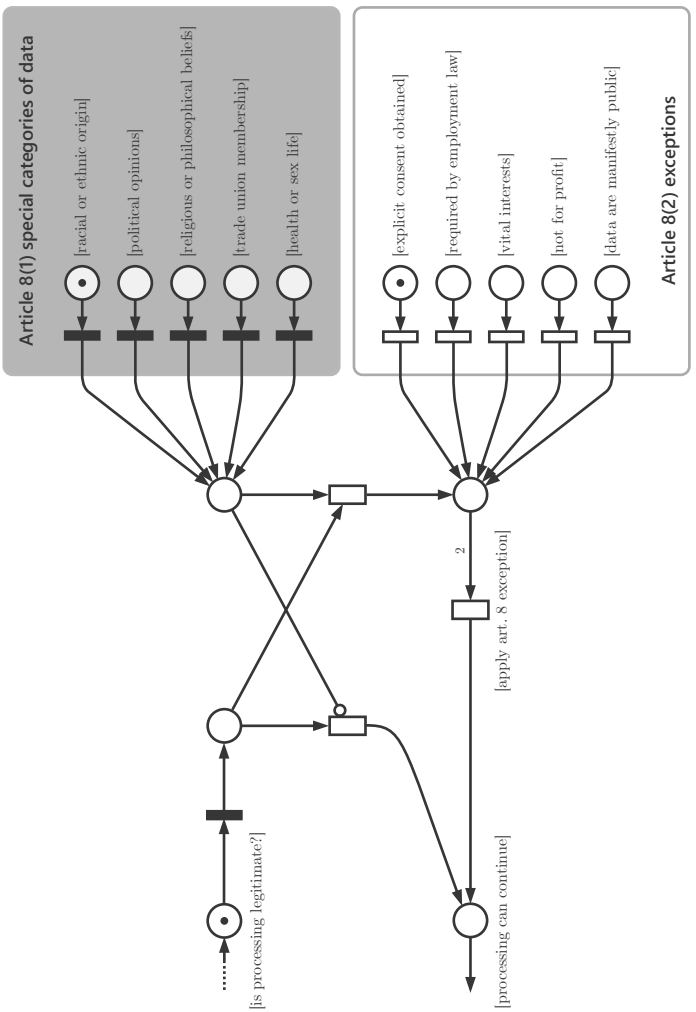

Figure 7.3 A Petri net model of Article 8 of the Data Protection Directive

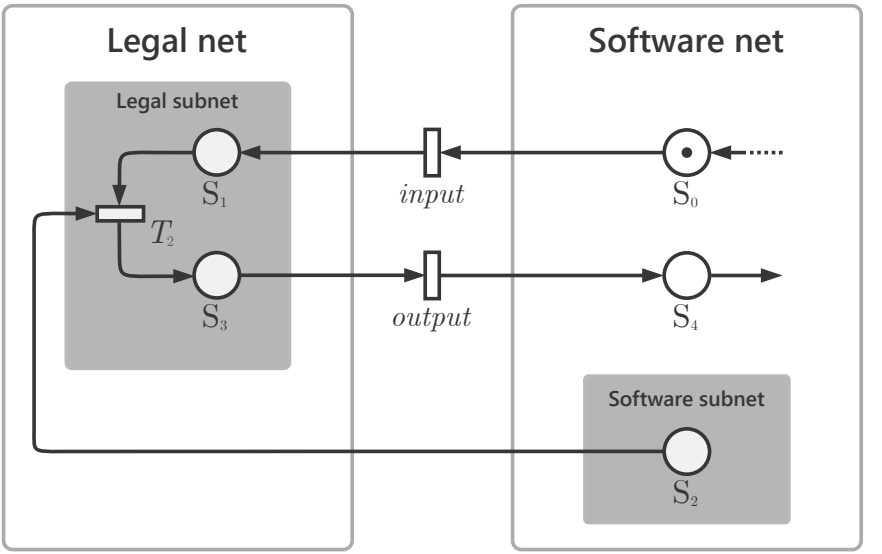

Figure 7.4 Communication between legal and software models

7.5 Conclusion

The goal of this chapter has been to highlight various aspects of the practical implementation of digisprudence, rather than to canvass all possible ways of achieving legitimacy. Especially relevant is the 'constitutional' role of the PoP, as embodied in code development paradigms, programming languages themselves, and the software environments in which code is invariably produced, known as integrated development environments (IDEs). Neither programming languages nor IDEs are in any sense 'found'; they are themselves the product of many design decisions that have normative weight which, to a greater or lesser degree, filter through to the code that is produced with(in) them. There is thus a parallel between constitutional and parliamentary law-making on the one hand, and the programmer of the programmer – represented *inter alia* in the affordances of the integrated development environment – and the product designer on the other. By considering what the (secondary) affordances of the design environment are and ought to be, we can imagine binding the creator of the (primary) technological normativity embodied in the finished product.

The approaches described above (and others that achieve similar ends) can only ever be a part of a broader commitment to legitimate design. At best they address only some aspects of the digisprudential scheme, concerned as they are primarily with formal contestability rather than the qualitative

aspects of design described in Chapter 2. It would be dangerous to assume that the approaches of the kind described in this chapter would on their own exhaust the duty to legitimise code as a normative order. Fashions and technologies change of course; the discussion above has pointed to some classes of approach that can have a bearing on the operationalisation of digisprudence, without becoming so granular as to be left behind as coding practices evolve.