It Depends: How to Develop Judgement in Programming

John Wilson

Edinburgh Medical School, University of Edinburgh
jwilso3@ed.ac.uk

Brittany Blankinship

Edinburgh Medical School, University of Edinburgh
b.blankinship@ed.ac.uk

Introduction

Programming courses often focus on what to write... syntax, functions, packages, patterns, pipelines. What gets less attention, even in "intermediate" or "advanced" courses, is how context influences the coding approach and how to dynamically adjust approach to fit the context. Programming courses and tutorials usually give advice on writing DRY, clean, expressive code. We believe there is a gap in how such advice is presented that can be filled by considering three tactics that in combination create a programming "mindset space". Having an awareness of these tactics allows a learner to consider which tactics are most useful at a given time, for a given problem, and to understand they can change in response to the different pressures their current situation might be exerting. As a way of making this explicit and teachable, [13] the three tactics we suggest are being Independent, Conscious, and Social (Figure 7.1). And further, we refer to these tactics as "forces" because they can be used to push against, or accommodate, the pressures felt as appropriate to the immediate context.

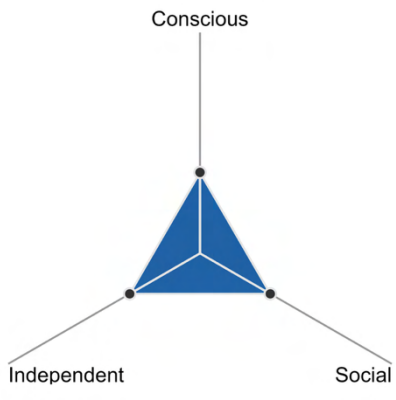

Figure 7.1: Radar plot of coding forces visualized with the 3 forces being equally balanced

The key idea is simple: good programming is not a single fixed standard, regardless of what language you might be programming in. Instead, different situations call for different combinations of skills, habits, approaches, and considerations. The classic "well, it depends..." applies to coding as well! This is to say, that in each project or task, the appropriate intensity of each coding force may vary.

The 3 Coding Forces

Independent coding focuses on problem-solving autonomy and self-sufficiency. This coding force is all about developing coding approaches without hand holding. This often presents in debugging, especially knowing when and how to navigate documentation or help files, search web resources (e.g., StackOverflow, books, blogs, etc.) effectively to find workable solutions and crucially adapt them to your context. Independent coding is about daring to try things out or experiment with ideas, adapt existing approaches, or develop your own new approaches without step-by-step guidance from a blog, colleague, or teacher.

Much of personal learning lives here. Think "I can do it all by myself, and how to implement it (all by myself), and know where to look for help if not". When primarily leaning into or applying the independent coding force you might produce exploratory scripts, small experiments, and the kind of trial-and-error work that helps to build competence and intuition.

Strong independent coding develops with time and once solid, presents as the confidence to be able to independently learn new packages or

[13] Here we have decided on a radar plot to represent the coding forces. It is not absolutely perfect, but this is a limitation we are accepting for now for clarity of communication. A Venn diagram seems like a good way to present these the forces, however, this gives an impression of three discrete "modes" of programming. The radar plot is more accurate in that it can show varying degrees of agency along each axis. A ternary plot we feel would be even more precise because it also acknowledges that there is a trade-off inherent in moving along any single axis. But ternary plot can be difficult to interpret, so we have spared you this.

workflows. Independence is what allows programmers to keep moving forward when they encounter unfamiliar problems. This is one of the hallmarks of "strong coders". It is not about knowing everything but having the confidence and skillset to learn and apply (nearly) anything, or to proactively decide to seek help when necessary. Programming skillsets like problem decomposition and debugging come into play here, which are built and strengthened through practice and exposure.

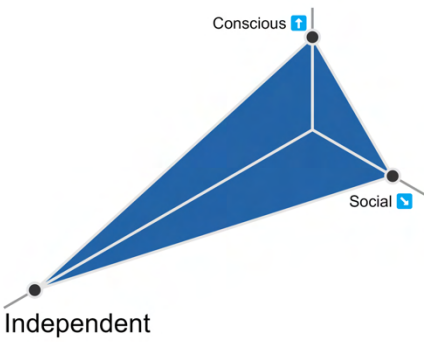

Figure 7.2: A dominant Independent force. Being close to the centre of the radar plot suggests zero independence, waiting to be given answers. The far end of the axis suggests taking the initiative to make progress, and judging when, how, and where to find necessary help.

Conscious Coding focuses on awareness and intention. This coding force is all about being mindful and making deliberate (and informed) choices about the structure, workflow, tools you choose to use in your code base. Conscious coding requires thinking ahead about structure, assumptions, and trade-offs rather than just making code "work". When coding consciously, you need to consider the bigger picture of how and where your code will be implemented, how it can be maintained, and why you are writing a given bit of code, sometimes even at the level of individual lines. When this force is the dominant force in your mindset, code comments should document why certain decisions were made (e.g., using one suite of packages over another due to downstream integration, etc.).

This is where questions about workflow, reproducibility, and tool choice live, including if or how to use AI tools! [14] Always bear in mind what AI produces, when it is helpful, and when reliance on it may hinder learning, flexibility, or long-term understanding. This is especially important when you are developing your skills. Learners may well encounter experienced coders that routinely use AI, but they should consider that perhaps those coders have already mastered the skills they are trying to learn! One good piece of advice is to type every single line of code AI produces with your own fingertips - avoid the temptation of mindless copy-pasting! This force is not about speed in making code that "works" or simply executes, it's about making conscious decisions using learned judgement.

Conscious coding asks you to slow down and reflect:

- Do I understand what this code is doing?
- Do I understand how to use this solution in another situation?
- Is this approach appropriate for this task?

If the answer is no to any of the above, take the time to experiment and learn (moving into the independent coding force) until the answers are yes. Otherwise, if time does not permit exploratory learning, do not use an approach or solution you do not understand.

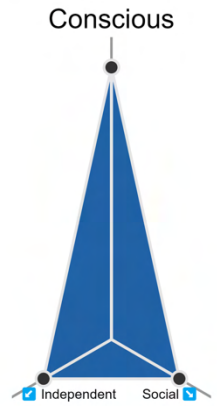

Figure 7.3: A dominant conscious force. Being close to the centre represents a lack of consideration about the code being written, such as the blind/mindless copy pasting of AI outputs. The far end of the axis represents making decisions based on long term maintenance of the code, only implementing solutions you understand and can debug.

Social Coding focuses on communication. Code is written to be read by other humans and executed by the machine. Social coding recognizes that most code is read (by other humans) more often than it is written. In this way, code itself is a form of communication, which you should want to be as clear as possible. We do not code in binary (can you imagine how gross this would be) for a reason!

This coding force is all about writing code that other human-beings can read and understand quickly and easily as well as reuse and trust. This often presents itself in robust documentation and ensuring consistency in procedures and approaches in a given script. Doing this well means writing code with clarity of meaning - i.e., consistent code style, descriptive naming conventions, and clear documentation through judicious use of comments, and properly documenting your functions.

When coding with a focus on the social force, your code may become longer, taking up more lines - which is not a bad thing! It does not mean your code is not DRY (the principle of Do not Repeat Yourself) but rather balances such principles with readability and

[14] Here we intentionally use AI generally not just Generative AI or LLMs, but AI as an umbrella term including developing tools like agents.

communication. Collaborative projects, research codebases, (and... assessed work in courses...), where maintainability and clarity are as important as correctness, depend heavily on a strong application of the social force. A useful realization for a learner to have is that collaborators may be people that you never have an actual face to face conversation with. Socially aware code, then, has to be friendly and approachable and the best ways to do this are to provide clear, accessible documentation, and code that is also self-documenting.

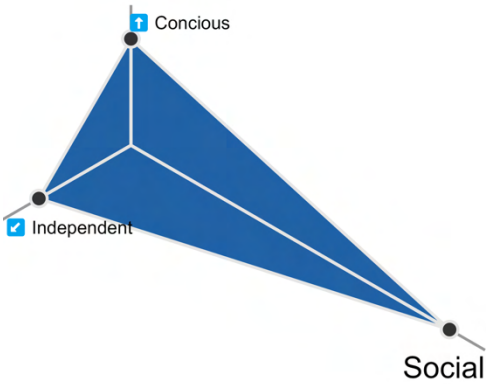

Figure 7.4: A dominant Social force. Being close to the centre represents, possibly rapidly developed, code that is ugly or without structure. The far end of the axis represents code nicely refactored for maximal readability, with the intention of sharing with collaborators or colleagues.

Different tasks call for the forces to be applied with different intensities, to actively balance them in the way that best fits the situation. What that balance might look like, is a deliberate choice of the coder:

- **quick exploratory work** may mean a dominance of independence because this allows faster iteration and experimentation.
- **short-lived scripts** might justifiably ignore social expectations. If that exploratory work transitions into being **integrated into a proper project with collaborators**, the influence of the conscious and social forces will become more important, likely requiring a tidying up and reconsideration of the code.
- **collaborative projects** need more from conscious and social forces
- **troubleshooting** may require a shared dominance of the conscious and independent coding forces
- **research and work projects** typically sit near the centre, requiring independence to solve problems, consciousness to make good design decisions, and social awareness so the rest of your team can understand, reuse, and replicate the work.

As a Tool for Reflection – Questions To Ask Yourself in Navigating These Coding Forces

We have intentionally chosen the term "force" to emphasise that these are not discrete modes you switch between, but influences that are always present... and which you have agency over. Each force can be applied with varying intensity depending on the situation, which is where your agency comes in.

This framing makes these forces something you can actively work with. They become levers you can pull to adjust each force, to manipulate how strongly each one shows up in your coding workflow. In practice, this isn't about turning one force "on" and another "off", but more like pulling levers that allow you to fiddle with the relative intensities of each force. This leads to a continuous tuning or rebalancing of forces as required by the current context.

Of course, those adjustments do not happen in a vacuum. They are often shaped by "pressures" - internal or external signals such as deadlines, collaborator expectations, or even just that weird feeling that there's something not quite right in your code. The radar plot reflects this idea. Rather than separating forces into neat regions it shows them all present at once, just at different intensities. The resulting shape is simply a snapshot of how things are balanced at a given moment (Figure 7.5).

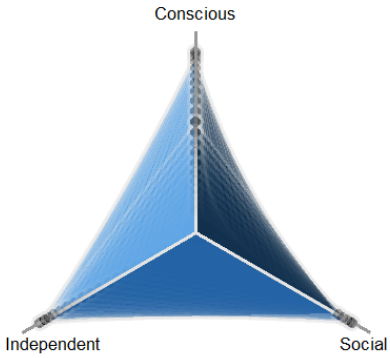

Figure 7.5: Ever changing balance of independent, conscious, and social code forces. In the online version, this is a dynamic gif but presented here as a static snapshot.

These three forces define a "mindset space", different regions of which are better suited to different applications. Moving about this mindset space requires reflection and awareness. This can be regularly asking yourself, "What am I doing right now and what mindset do I need to adopt?".

Importantly, it should be understood that using these coding forces is not a matter of switching from one to another. It's not as if you turn off "social coding" and suddenly become a purely independent coding machine. You do not take a metaphorical turn off Conscious Drive in order to exclusively be on Independence Avenue. Rather, you deliberately and dynamically alter the relative intensities of each force adapting to the context or circumstance.

You can think of the radar plot like a 3-way seesaw with each coding force functioning as a lever arm of the seesaw. All three forces are always in play just as all lever arms are in play on a seesaw though with different weights. As a visual, this analogy is useful because it highlights that raising one lever arm affects at least one of the others. In practice, this feels less like stepping between modes and more like

constantly rebalancing, or repositioning. You might lean heavily into independence when poking at a new problem, then gradually bring in more conscious structure as you converge to something worth keeping and then add social clarity as things settle down or need to be shared.

Speaking of our own tendencies, it is often that we (and possibly also the case for you) naturally fall into fast code that is not social or conscious. Code is never perfect the first time it is written, which is to say, we never get it perfect right out of the gates. Outputting final "socially acceptable" code is iterative, where each iteration changes the intensity of each force. When designing course content, however, the social coding force strongly dominates in the first instance.

Finding the Right Balance

Some questions you can ask yourself when searching for the right balance between these coding forces:

To increase the influence of the Independent coding force:

• I am stuck, how can I create a mini-reprex[15] for this issue?

To increase the influence of the Social coding force

• Will a collaborator (or marker) understand this code?

To increase the influence of the Conscious coding force

• Where or how will this code base be used? Does it need to be over-engineered and consider all edge cases, or no?

• Oh, I have a lot of functions, should I maybe separate them out into a separate script and source them in for debugging and maintainability?

More generally, when building up the awareness of the relative dominance of the coding forces and developing the reflexivity skill, it can be useful to take a step back and reflect on the following:

- What mindset and coding force am I operating in right now?
- What mindset should I be in right now?
- How do I even know what mindset I am/should be in right now? What is the primary purpose of the task I am working on?

Pressures and Rebalancing

Importantly, these coding forces are not separate "modes" that you switch between like changing tools in a toolbar or dramatically entering a new mental state while cinematic music plays in the background. All three forces are always present to some extent. What changes is the relative intensity of each force and how heavily you choose to lean into them depending on the situation.

This is why we use the term "force". A force is something that can be applied with varying strength. You can increase the intensity of independent coding when experimenting with a new idea or debugging an unfamiliar issue. You can increase the conscious force when making decisions about structure, reproducibility, or long-term maintainability. You can increase the social force when preparing code for collaborators, assessment, or future-you three weeks from now wondering what on earth past-you was thinking.

The important point is that you have agency in how these forces are balanced. Different situations create different pressures, and they may suggest a certain balance over others. So, rephrased, the context may influence the intensity of each force, but it is you who actually decides how heavily to lean into each force.

Some pressures are **external**. Deadlines may push you (temporarily) away from careful, consciously structured code, toward rapid exploratory work. Collaborator expectations may increase the need for clarity or documentation or consistency. Assessment tasks often increase the importance of social coding because your marker is, in a maybe slightly awkward sense, one of your collaborators.

Other pressures are **internal**. You may notice that your code is becoming difficult to debug, difficult to explain, or increasingly fragile as your work becomes more complex. You may realise you are over-engineering a tiny script that somehow now contains five layers of wrappers and several coding strategies named after medieval architecture. These are all signals that your current balance of coding forces may no longer fit the task particularly well.

With experience, programmers become better at recognising these pressures and adjusting accordingly. This is less about following fixed rules encountered on the world wild web, and more about developing judgement about when to increase or relax the emphasis on a particular force. You might begin with very independent exploratory work, gradually introduce more conscious structure as ideas settle, and later increase the social force once the code needs to be shared, reused, or maintained.

The radar plot reflects this idea. It is not intended to divide coding into neat regions or categories, but to show a changing balance of forces at a given moment. The resulting shape is simply a snapshot of which forces are currently being emphasized and how strongly they are being applied. Thinking back to our seesaw metaphor, the radar plot is a snapshot of the seesaw at a given moment. The goal is not to maximise every individual axis independently, but to arrive at a balance appropriate for

[15] What is a reprex? It's a reproducible example, as coined by Romain Francois. The goal of a reprex is to package your problematic code in such a way that other people can run it and feel your pain. Then, hopefully, they can provide a solution and put you out of your misery. The vast majority of the time creating an excellent reprex reveals the source of your problem. It is amazing how often the process of writing up a self-contained and minimal example allows you to answer your own question.

the task at hand. The important part is not any single axis alone, but the overall shape created by how the forces interact. As such, the plot is best understood as a snapshot of the relationship between the forces, not simply three separate values.

Moving Into the Next Dimension

Thus far, the framework as we have presented it can safely and easily live in a 2D mental space, represented by the radar plot and seesaw metaphor.

If you will permit us to further extend this metaphor, imagine that as you gain more awareness and experience on the seesaw of coding forces that the lever arms no longer move up and down but become a 3D puzzle pieces. As a learner acquiring a new programming language, you can think of your progression through the learning journey and the coding forces as a process of acquiring the puzzle pieces. Once an understanding of the independent, conscious, and social pieces are acquired, you realize they are complimentary elements that fit together and form a globe-like mindset-space. If you think of each force as a direction of travel within that globe, a new action is unlocked: spinning the globe and navigating the full 3D space!

Figure 7.6: Extending the 2D radar plot into a third dimension lets us imagine a continuous volume, through which we can navigate by changing the influence of the coding forces, until we find the region that corresponds to the best balance between the forces.

In this metaphor, meta-awareness and reflection is the glue that fits the puzzle pieces together forming the globe and allowing you to take on the hands of God role and move the globe. This meta-awareness is only possible to reach once you have knowledge of all 3 forces. It is required to then apply a fitting intensity and direct forces that feels appropriate for the problem at hand.

Using the radar plot imagery when applied to a specific problem it is unrealistic (and difficult to visualize) maxed out intensity applied to all 3 forces. After all, as human attention is a finite resource. This is where the more 3D globe visual takes over. Your knowledge of the force can reach a maximum - this is where meta-awareness comes in to know how to best apply intensity to each force for the specific problem at hand. This is to say, critical thinking becomes your meta-awareness.

And once you realise that you are applying your critical thinking skills in the form of meta-awareness of the coding forces, you can measure your own progress and development!

As a Pedagogical Framework

One of the many benefits of this framework, in our very unbiased opinion, is that it is language agnostic and can be adapted with specific examples for whatever language(s) are being taught or used.

As a reflection point, you may consider how code written for teaching and learning purposes is very different from code written for an open shared project or your own projects. This framework provides a model for considering why this is the case and vocabulary for communicating to students, other teaching staff, and even yourself.

We have presented the framework forces in the order in which they are often (unintentionally) taught and experienced by learners. Independent (write or edit code on your own to make it execute without error) > Conscious (understand the bigger picture of your code base and where and how it might be used and make decisions accordingly) > Social (write code for others to read, use, or maintain) reflects the process as a learner and the forces that they may become more aware of as they progress from novice to competent and even intermediate or advanced.

This framework helps students understand why expectations change across tasks. It “allows” exploratory, messy code in learning contexts, while still setting clear standards for shared or assessed work. As an educator, learning how to think about these forces can help you identify where students are struggling and push against forces they may be struggling with. This framework provides a vocabulary to communicate that opens the door for insights and empathy while giving students the same tool.

Rather than teaching a single notion of “good code,” this framework teaches students to employ critical thinking skills and choose how to code based on context. It can be used a means of explicitly communicating to students why the code they find in different places (e.g., GitHub, StackOverflow, etc.) or see in the course may look or feel different and how to make informed decisions on how to model their own code depending on its use case. Shifting from rules to judgment is what shows a transition to more mature, adaptable programmers.

The framework can be used to hang “good coding practices” on, both language specific and agnostic, and highlight **why** they are useful for considered good practice. For example, naming conventions can be good practice aligning with both conscious and social forces. File management allows for independent, conscious, and social forces to all shine. Code comments are an example of how the forces of coding that you are working in can change to make a “good” or “useful” comment. A code comment to yourself is very different from one intended for a colleague to maintain the code base and understand your decision making process.

While we will not go into depth here (to save us all the headache for now), it also provides a language for discussing AI use not as something to accept or reject wholesale, but as a tool whose value depends on whether it supports independence, consciousness, and social responsibility. Using LLMs as a collaborator necessarily means that you must not take what is generated blindly and ensure it is social, which requires conscious and independent coding forces to be in full effect and awareness of the social force. Reading and working with AI generated code also requires some intensity in the social force. Guiding questions if using tools such as LLMs when coding are:

- Do you understand the code that has been generated? Do you understand it well enough to debug it, not just what it is doing?
- Would your colleagues understand this code?
- If the LLM is a collaborator, how do you weigh its suggestions vs other (likely human) collaborator inputs?

Equally, we would suggest that this framework provides a structured answer to ridiculous questions such as *"Why should I learn to code if AI can seemingly generate code for me?"*

Our Answer Would Be Something Along the Lines Of:

That is fine if you want to do that, but you still must use the tool (AI) in a conscious way. You need to know how to prompt, which demonstrates a certain level of independence as you have to know what the ask for in a prompt. This does not demonstrate conscious thought on its own.

In this framework it is your job to change the level of intensity of conscious thought. Similarly, you must intentionally decide what advice to listen to - human collaborator or AI? Should you be "guided" by the AI tool or should you be "directing" the tool? This hits all 3 forces.

But don't take our word for it. Try coming up with your own answer by asking which of the forces the AI is applying. Is it truly applying independent judgement? Is it conscious of the wider needs or implications of your project? Is its code actually as socially aware as it looks at first?

If you would like to hear more about our thoughts in this, get in touch or stay tuned for another chapter in Version 2 of the book.

The Need For This Framework

Yes, none of these forces are a new or unique idea. Many of us are familiar with them already. If you've spent any time around programming education (or lurking on StackOverflow at 2am), you've seen them all before. People talk a lot about things like reading the manual (rtfm)... debugging skills... DRY code (but not too DRY!)... clean code (but not too clean!). And entire books exist for each of these, often with very strong opinions. And yet, many learners struggle to apply them appropriately. The part that's a bit rarer is seeing them all in the same room, talking to each other - or if you will permit, all together on the same seesaw.

More often, they're presented like separate "best practices", each living in its own neat little box. We might have read something about clean code... that leans heavily on social practices, presented as "writing for others", "name things well", or "write clear documentation". But without much mention of the messy, trial-and-error habits that actually allow us to make progress in the first place. Or a debugging guide might encourage us to poke, prod, and experiment freely, but give us little guidance about when it's time to slow down and be more deliberate.

So the advice isn't wrong, but tends to come across as ideas that are isolated from each other, instead of as a set of forces that can be deliberately balanced, where we, ourselves, can decide, rather than guess, how much to lean into a given force at a given time. And that leaves learners to stitch it all together themselves, which is a bit like being given three excellent maps without being told they're of the same place. That's where a lot of the tension creeps in. Beginners, especially, can end up feeling like they're being asked to do contradictory things. "Write clean, self-documenting code" sits right next to "just try stuff and see what breaks," with no explanation of how those ideas fit together. It's not obvious that these

aren't competing rules, but different forces that make sense in different situations.

The Challenge For Educators

Most educators already *feel* this, even if they don't always spell it out. You know that a rough, experimental script written while figuring something out is perfectly fine - even desirable. You also know that the same code would raise eyebrows (or worse) if it showed up in an assessment or shared project. That sense of "what's appropriate here?" develops over time. Learners, however, haven't had time to build those instincts yet. And importantly, they may not even realise there *is* a shift to notice.

A learner who has spent time writing code that is exploratory, maybe slightly chaotic or full of half-formed ideas, comments to self, and creative shortcuts may not understand why those same habits suddenly become a problem elsewhere. It's not that they're being careless, it's just that they're applying the advice they've encountered in the best way they know how... just without a clear sense of how or when to shift their approach as the context changes.

This often shows up in predictable ways. Bits of exploratory code sneak into submissions or shared projects, bringing their weirdness and chaos with them. The result? Code that's harder to read, harder to trust, and harder to build on. On the flip side, some students go the other way by trying to write perfectly polished, fully documented code at all times. Which can be unnecessarily slow and frustrating.

There's also a slightly awkward truth worth saying out loud: **assessment is collaborative.** i.e., the assessor *is* your collaborator. If your code is easy to read, understand, and trust, it's not just "good practice", it will also assess well. For many students, this is their first real experience of writing code for someone else, and without a clear way to frame that shift, assessment criteria can feel a bit arbitrary.

What This Framework Brings

This framework is an attempt to bring some clarity to a muddy reality, not by adding more rules, but by giving people better language.

Instead of treating good coding as a single fixed standard (which we either meet or don't), it frames things as a space we can move around in. Independent, Conscious, and Social aren't boxes to tick; they're forces we can lean into more or less, depending on what we're doing. The more we apply one force to a problem, the more we move our mentality into our desired mode. Deliberately! And conversely, the more we feel ourselves being forced into a corner of a space that isn't working for our problem, the more we can resist that pressure, and move ourselves into a different region of the force-space.

Sometimes we need to experiment quickly and figure things out (Independent). Sometimes we need to slow down and think carefully about structure and decisions (Conscious). Sometimes we need to prioritise clarity for others (Social). Most of the time, we're somewhere in between. And importantly, we can apply conscious thought to independently position ourselves where we need to be.

The aim is to make that movement visible and intentional. Rather than relying on intuition that develops slowly (and unevenly), learners can start to recognise where they are, where they might need to be, and how to adjust.

The radar shows that all three forces are always there, just pulling with different strengths. We don't have to turn one off to use another - we **rebalance** the forces.

This all works regardless of whether we are using R, Python, C, or something more obscure (中文[16] anyone?). The specifics change, but the forces don't, which makes this something we can carry with us across courses, projects, and (eventually) jobs.

16 https://esolangs.org/wiki/%E4%B8%AD%E6%96%87