A description of developer behaviour. e.g. "Uses camelCase for variable names"

Example:

Some cards give examples to help clarify the described behaviour. These are not intended to be comprehensive; base your response on ANY symptoms you can think of which manifest the behaviour

Gives each variable the smallest possible lifetime and scope

Example:

Declares a local object right before its first usage

Writes functions which retain state between calls

Example:

strtok (C library string tokeniser) uses static data to maintain the current position in the buffer

Seems to write a lot of lines of code to achieve a given outcome

Tends to "own" code

Example:

* Does not like others to work on code they wrote
* "Resident expert" who always wants to be the one to write the device drivers

Uses long multi-part conditions

Example:

Uses a complex conditional expression with multiple operators, rather than nested conditions or intermediate boolean variables

Is willing to discuss suggestions about their code

Example:

* Asks for more information about suggested approach
* Debates pros and cons
* Makes a change or explains a clear rationale for the existing approach

Makes data immutable whenever relevant

Example:

```
const float PI = 3.1415926535;

final String exitMessage = "Goodbye";
```

Prefers <a programming language> and uses its idioms when coding in other languages

Automates tasks

Example:

* Running of tests
* Building code
* Generating documentation
etc

Uses horizontal and vertical spacing to align and separate code

Example:

* Alignment of related structures
* An empty line between two sections

Includes accurate details of symptoms and how to reproduce the bug in their bug reports

Keeps the flow of control easy to follow

Example:

* Avoids GOTO
* Uses break judiciously
* Avoids deeply nested code

Follows encapsulation principles

Example:

Data structures and the processes that operate on them are in the same module, e.g. temperature data and the associated methods for conversion between units

Tries to provide early outline functionality that other team members can use

Example:

Defines APIs and implements some minimum functionality which will compile and run

Checks return values from function calls which may return error codes

Example:

Checks return value when opening a file for write

Follows formal methods to the letter

Is good at helping others

Example:

* Willing to answer questions
* Patient with less experienced developers
* Explains how to fix a problem rather than taking the keyboard and just fixing it

Is rigorous about deallocating allocated resources

Example:

* malloc - free
* file open - file close
* new - delete
etc

Lets existing code constrain future code

Example:

* Tries to make minimal changes to existing structure
* Is reluctant to refactor

Ignores build warnings

Accompanies each commit with a suitably informative message

Includes code features that are not currently needed

Example:

* Thinks it might be needed in future
* Finds it easier to implement it than check if required

Uses the idioms of the programming language

Example:

* Lambda expressions in functional languages
* List comprehensions in Python
* Ternary operators in Java
* Use of the STL in C++

Writes long if... else if... else if... blocks

Includes brackets which are not demanded by mathematical operator precedence (BODMAS)

Example:

```
fahrenheit = (celsius * 9.0 / 5.0) + 32.0;
```

Logs it in the issue-tracking system when knowingly making a sub-optimal change

Example:

Quick and dirty change to get customer's system working

Is always willing to consider that the bug may lie in their code

Commonly writes methods with 6 or more parameters

Catches exceptions at a level of the code where they cannot be resolved

Tries to leave a module a bit better than when they checked it out

Example:

* Splits up an over-long function
* Improves a variable name
* Boards up "broken windows" if there isn't time to fix them
  e.g. display a "not implemented" message

Rarely uses exceptions as part of a program's normal flow

Example:

Assuming no unexpected events, the program could still run correctly if all exception handlers were removed.

Is often the person who breaks the build

Example:

* Missing files
* Missing steps
* Broken code
etc

Puts project goals over individual goals

Example:

Willing to give priority to helping fix a problem which is holding others back but does not affect their own current task

Prioritises performance in the design of their code

Example:

Code is optimised for speed over readability without any benefit to overall system performance

Asks questions without giving context

Example:

I'm getting an exception. Do you know what the problem is?

Writes short, simple functions which perform a single task

Uses assignments within expressions

Example:

```
a = 3 / (b = c + 1) % d;
```

rather than

```
b = c + 1;
a = 3 / b % d;
```

Does not assume that a complex problem necessarily results in complex code

Fixes the symptoms without discovering the root cause of a bug

Finds out whether functionality is already available before writing their own implementation

Example:

* Uses project libraries
* Uses language libraries
* Uses frameworks

Includes useful logging messages in their code

Example:

* Errors, warnings and info are clearly distinguished
* Content of messages is informative and succinct
* Quantity of messages is sufficient but not verbose

Doesn't always update or add tests when changing code

Example:

* Does not create tests for new code they write
* Does not update tests for code they modify/extend
* Does not add a new test to trap the bug they are fixing

Espouses "one true way" of doing things

Includes brackets which are not demanded by the language's operator precedence

Example:

```
result = operand << (a + b);


if ((day == 31) && (month == 12) && (year == 1999))
```

Assumes that things which "can't happen" won't

Practises good housekeeping

Example:

* Removes temporary debug statements
* Does not leave TODOs for others to deal with

Is willing to ask questions

Example:

* Asks whether there is existing code to do X
* Asks for advice on programming issues
* Asks for explanation of domain-specific concepts
* Asks for clarification of a requirement rather than making assumptions about the correct interpretation

Codes using implementation terms rather than domain terms

Example:

```
if (list.contains(user.getId()))

   rather than

if (user.isAuthorised())
```

Follows the DRY (Don't Repeat Yourself) principle

Example:

* Avoids "copy-and-paste" coding
* Contributes code to project libraries when they notice common use of functionality

Tends to work in isolation

Example:

* Checks in their work infrequently
* Rarely integrates their work with others

Makes APIs easy to use correctly

Example:

* Documents the API
* Designs APIs which seem natural and obvious, not for the convenience of the underlying implementation

Tends to apply a favourite pattern regardless of context

Uses code comments in ways that aid understanding

Example:

* Explains the domain logic of the code
* Documents design decisions, e.g.
  assumptions; alternatives discarded; trade-offs; workarounds
* Explains the task done by the following section of code,
* Updates existing comments to match changes to the code
* Does not obscure the structure of the code
* Comments only what the code cannot say

Chooses identifiers which are not succinct, meaningful and distinct

Example:

* Spelling mistakes: String passwrod;
* Silent differences: oldPwd & oldpwd; userInput & user_input
* Meaningless: String s; float ex; int foo; doStuff();
* Misleading: getData() actually writes data to disk
* "Cute": int avadaKedavra; // exit code when system dies
* Superfluous: temperatureData vs temperature
* Verbose: performReconciliation() vs reconcile()
* Nouns for function names: conversion() vs convert()