Assignment **A2: Bead**

Binding Instructions | Well-Meant Advice | Ideas for Thought | Common Mistakes

201. [**Decomposition into functions and classes**] It is expected that the whole code will be decomposed into appropriately designed global functions, classes and their member functions (methods), at different levels of abstraction from the user-friendly control of individual diodes to the actual logic of the running bead itself.

202. [**Class for diode representation**] Individual diodes will be represented as instances of a class that will encapsulate all the necessary data items and functionality. This means at least two functions, one to initialize a given diode and the other to change its lighting state. Using these functions, we will wrap the respective low-level Arduino system calls, giving users a more user-friendly means to control diodes at a higher level of abstraction.

203. [**Private data members**] A good practice when designing classes of our kind is to make all their data members private. Therefore, manipulating with them will not be possible directly from outside, only indirectly through public methods that we design for this purpose. Thanks to that, we will be in control of their content, otherwise they could be changed by anyone and from anywhere without our knowledge.

204. [**Logical diode numbers**] In order to really achieve higher abstraction, we will only use logical numbers `0` to `3` for diodes instead of low-level pins everywhere in the code. These numbers, in this order, will correspond to diodes labeled D1 to D4, and therefore pins `led1_pin` to `led4_pin`. The only exception where we can work directly with the pins will be the two functions mentioned above, being the only ones that will internally perform the necessary translation from logical numbers to the corresponding pins.

205. [**Use of pin number values**] By looking at the attached header file, we can easily find out that pin numbers of individual diodes have values `13` to `10`, but we certainly cannot work with them, regardless of directly or indirectly. We cannot even assume that they are potentially ordered or that they form a continuous interval.

206. [**Independence on constant values**] We can generalize this idea and extend it to the work with any other constants that we do not fully control. Unless their author has explicitly promised us some special guarantees, we simply cannot make any assumptions about their specific values. Thus we have to be completely independent on them.

207. [**Variable number of diodes**] The entire application must be implemented in a way that we do not enforce or assume any specific fixed number of diodes anywhere. In other words, everything must be universal and work even with a different number of diodes than just `4`. We will derive the specific number of available diodes from the size of the array we created for the purpose of translation of logical diode numbers to their pins.

208. [**Initial turning off of all diodes**] As a part of Arduino initialization, in addition to setting modes of pins, it is also necessary to switch off all the diodes, because we are generally not guaranteed in which current state they may occur.

209. [**Premature system calls**] Although we should actually not get into such a situation with resources we learned, let us emphasize that we must not call any Arduino system functions before the `setup` function, i.e., before its initialization. While such calls would usually pass without any effect on a real Arduino, the emulator in ReCodEx monitors such situations strictly.

210. [**Constructors of diode objects**] The complication described above could specifically occur in a situation where we would like to define explicit constructors for our diode objects, within which we would like to, for example, set modes of pins. Instances of objects in global variables are created at the beginning of the program, i.e., before the `setup` function. But since we can easily manage everything without constructors in our classes, we will not create them. If we do, we need to be aware of this behavior, though.

211. [**Internal state of diodes**] If we decide to remember the current internal state of our diodes, it is definitely necessary to represent it at the logical level, not using the low-level constants `LOW` and `HIGH`.

212. [**Named constants**] All constants having some logical meaning in the relation to our application must be declared and appropriately named. Moreover, if it is possible regarding to the other already defined constants, it is also necessary to calculate or otherwise derive their values with the help of such constants.

213. [**Constancy flag**] Let us also not forget to include the constancy flag as such for each data item that has such a character.

214. [**Global variables**] Considering the programming model offered by Arduino, it is necessary to remember certain information through global variables. Otherwise, we would not be able to transfer it between the `setup` function and the individual calls of the `loop` function. However, we must keep the number of such variables to a necessary minimum, we must especially not use global variables in situations where just ordinary local ones would suffice in fact.

215. [**Simple loop content**] However we decompose the logic of our running bead, it is necessary to ensure that the code put directly into the body of the `loop` function is not too large, complicated or technical. Ideally, we should only work here with some basic operations at the highest possible level of abstraction, similarly as was the case with the `main` function.

216. [**Efficient loop implementation**] Function `loop` forms the core of our programming model, therefore it has a key impact on the efficiency of our entire application. Obviously, it is not only about the function itself, but also all the other functions we will directly or indirectly call from within it. It is performed perpetually over and over, approximately $1000\times$ per second. Depending on the program complexity, this may be an order of magnitude higher as well as lower, though.

217. [**Class for timer representation**] It is expected we solve the functionality for event timing using a suitably designed separate class, too. It will become useful in the future, when we will need to use even more parallel and mutually independent timers at once.

218. [**Detection of elapsed time**] Checking whether a required time interval elapsed needs to be resolved carefully. As we have already discussed, one iteration of the `loop` function can take significantly less time than 1 ms, but it can also take longer. As a result, we are not guaranteed to be able to detect the intended moment of the next event with absolute accuracy, i.e., at the level of individual milliseconds.

219. [**Time of the previous event**] For the purpose of detecting the next event, we will obviously need to remember the time of the previous one, i.e., the last one performed. But if we are not able to detect time moments precisely, it would be a mistake to save the actual current time when this event occurred. If we did that, it could cause systematic and repeated delaying that would gradually accumulate and disrupt the required overall regularity of events.

220. [**Retrieving the current time**] Calling the `millis` function is not overly time-consuming, but it is also not fast. For this reason, it is advisable to ensure that within one iteration of the `loop` function, we call it to get the current time only once. This will also bring us another benefit. Although we do not need it yet, if we were to get the current time more than once within just a single iteration, it could happen that we get different values. Depending on our code, this could lead to very uncomfortable error situations caused by such an inconsistency.

221. [**Time value overflow**] For the system time representation, data type `unsigned long` is in particular used. Although it allows us to store time values in milliseconds corresponding up to almost 50 days, sooner or later its overflow will inevitably occur. Therefore, we must be able to treat such situations correctly.

222. [**Arithmetic over diode numbers**] We should always work with variables in accordance with their data type, but also with the expectations arising from their logical nature. For example, if we represent positions of diodes by their logical numbers, we cannot incorporate them in multiplication nor calculate their absolute value, although such operations would be allowed by integers as such from the technical point of view.

223. [**Prohibition of invalid values**] Analogously, we also cannot work with positions like `-1`, which are not even valid by themselves. We can then generalize this idea so that we can never reserve and use any otherwise normal values to represent erroneous, marginal or perhaps unexpected situations, values, etc. even in other situations.

224. [**Execution of initial actions**] Various initialization actions are expected within the `setup` function, hence it is not appropriate here to start executing operations that are part of the standard running logic of our application as such. In other words, we cannot yet turn on the diode corresponding to the initial position of the running bead.

225. [**Zombie actions in loop**] Following our efficiency requirement on the `loop` function, it is always necessary to think very carefully about what actions we want to carry out within it. It would be completely unacceptable, for example, if we put some conditional action in it while knowing in advance that it would actually be executed only once in the very first iteration and then never again.

226. [**Meaning of the current position**] In order to manage the logic of the running bead, we will probably need to deal with its current position somehow. We should therefore think about what exactly the word *current* actually means in this context. In other words, we should make sure that the current position actually does not reference, for example, the previous position or the following one, on the contrary.

227. [**Template of bead positions**] Although it would actually be a nice idea, we certainly do not want to solve our bead movement problem by creating kind of a template for the sequence of expected bead positions, which we would subsequently iterate through. It does not correspond to the logic of our assignment, which we must of course always respect.

228. [**Logical order of actions**] When realizing the actual bead movement, it is more than appropriate to respect the intuitive order of individual actions, i.e., first turning off the current diode, then moving to the next position, and only then turning on the new diode. Different orderings would be misleading.

229. [**Disallowed system functions**] Our entire application must be programmed in a way that we do not need to use `delay` or `delayMicroseconds` functions, just as we must not block the progress of the `loop` function in any other way. Simply because such attempts would go directly against the very logic of our entire programming model.