Kernel Implementations IV

Thanks to Dr. Bruce W. Weide
of Ohio State University for these slides

Recording Design Decisions

- It is also important to *record (document)* the key design decisions illustrated in the implementation of an layered piece of software, or new component

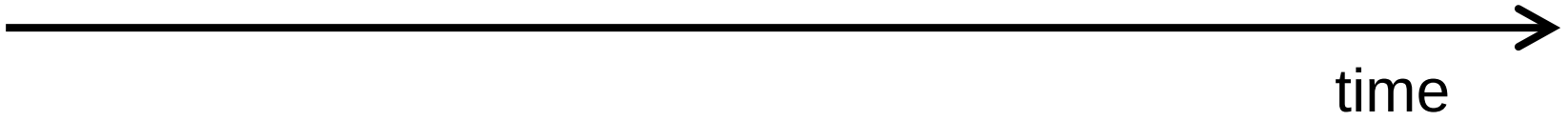
Two Key Design Decisions

- Perhaps surprisingly, there are really only two key design decisions that need to be recorded in comments of your component:
  - The ***representation invariant***: Which "configurations" of values of the instance variables can ever arise?
  - The ***abstraction function***: How are the values of the instance variables to be interpreted to get an abstract value?

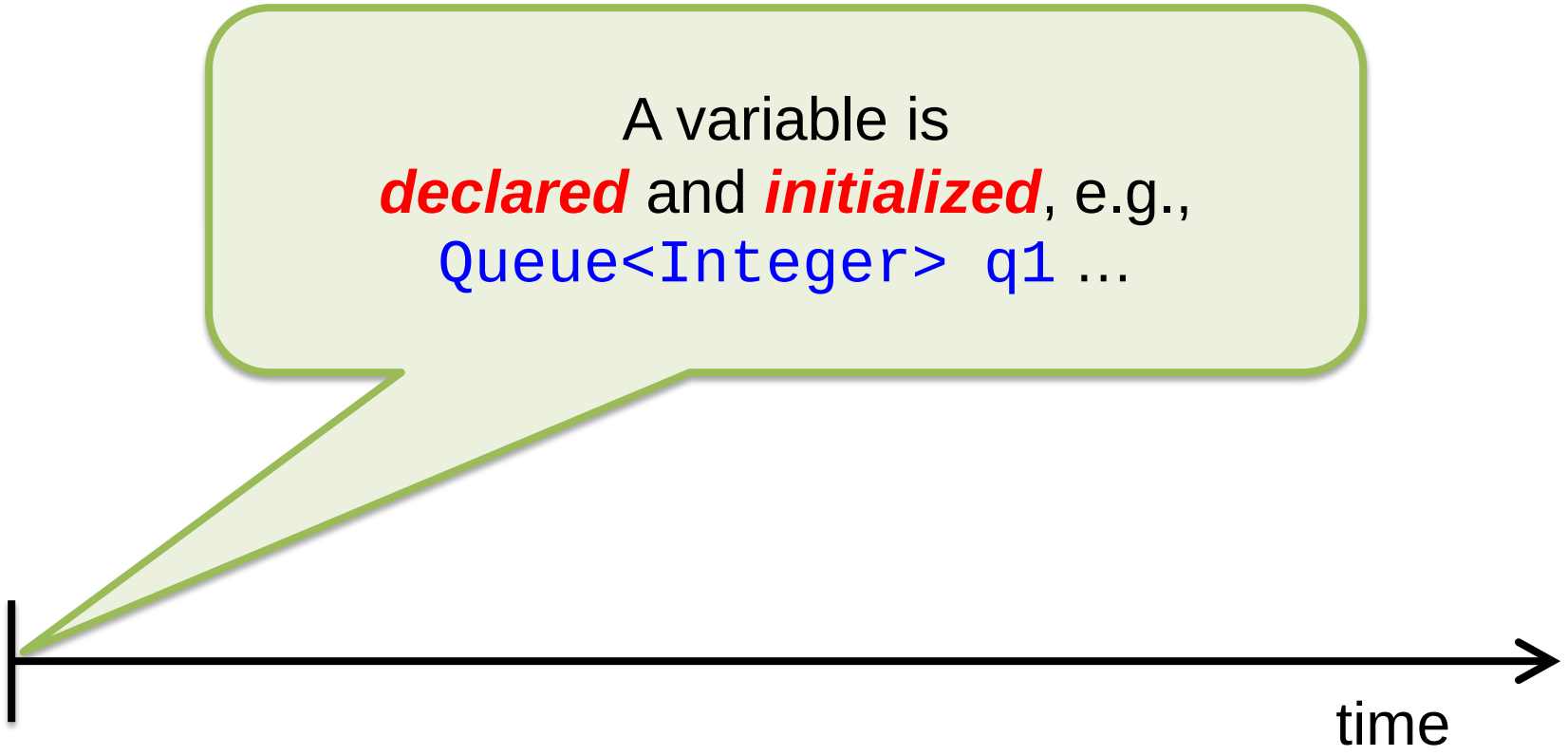
The Representation Invariant

- The ***representation invariant*** characterizes the values that the data representation (instance variables) might have at the *end* of each kernel method body, including the constructor(s)
- The representation invariant is *made to hold* by the method bodies' code, and it is *recorded* in the ***convention*** clause in a comment for the kernel class

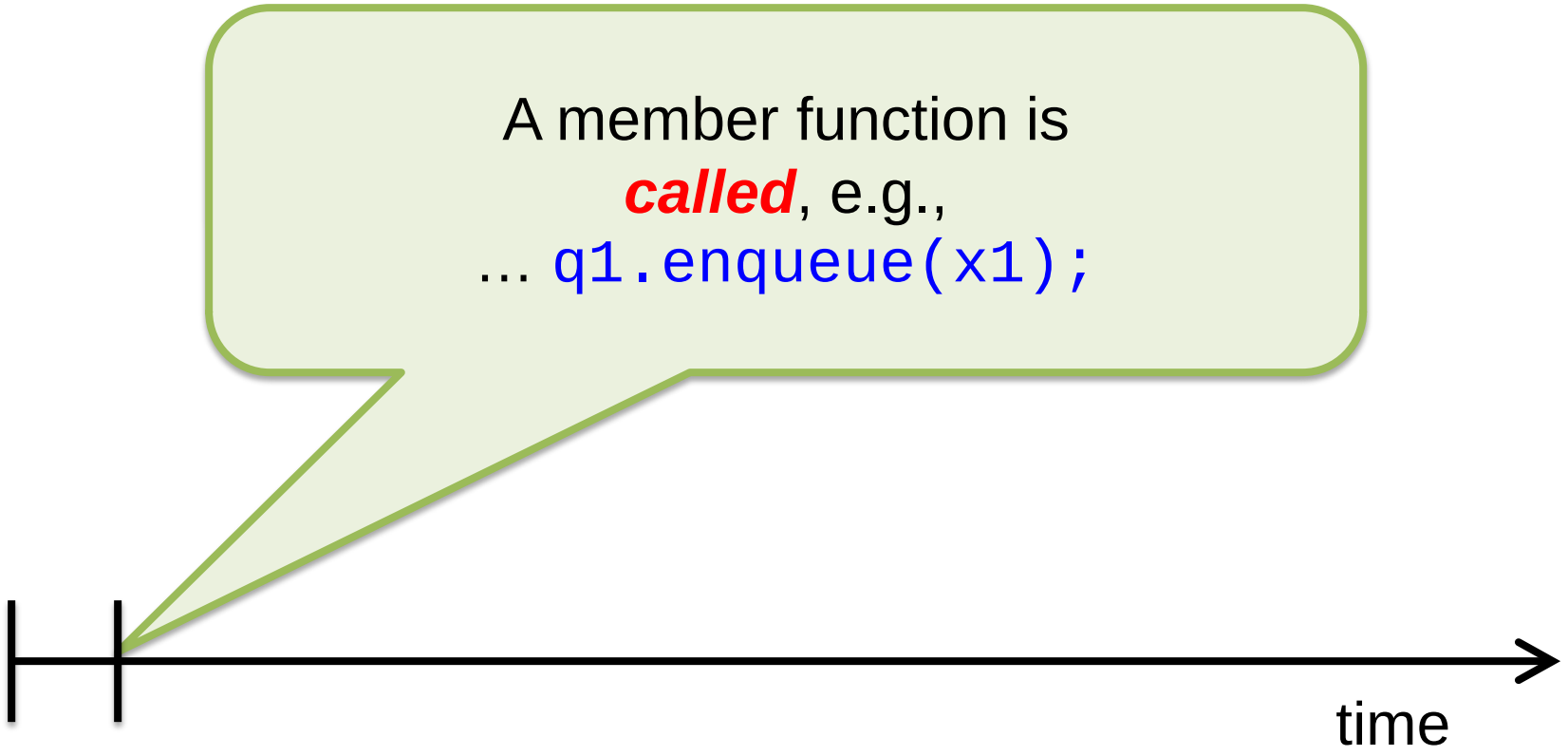
Variable Life-Cycle: Client



Variable Life-Cycle: Client

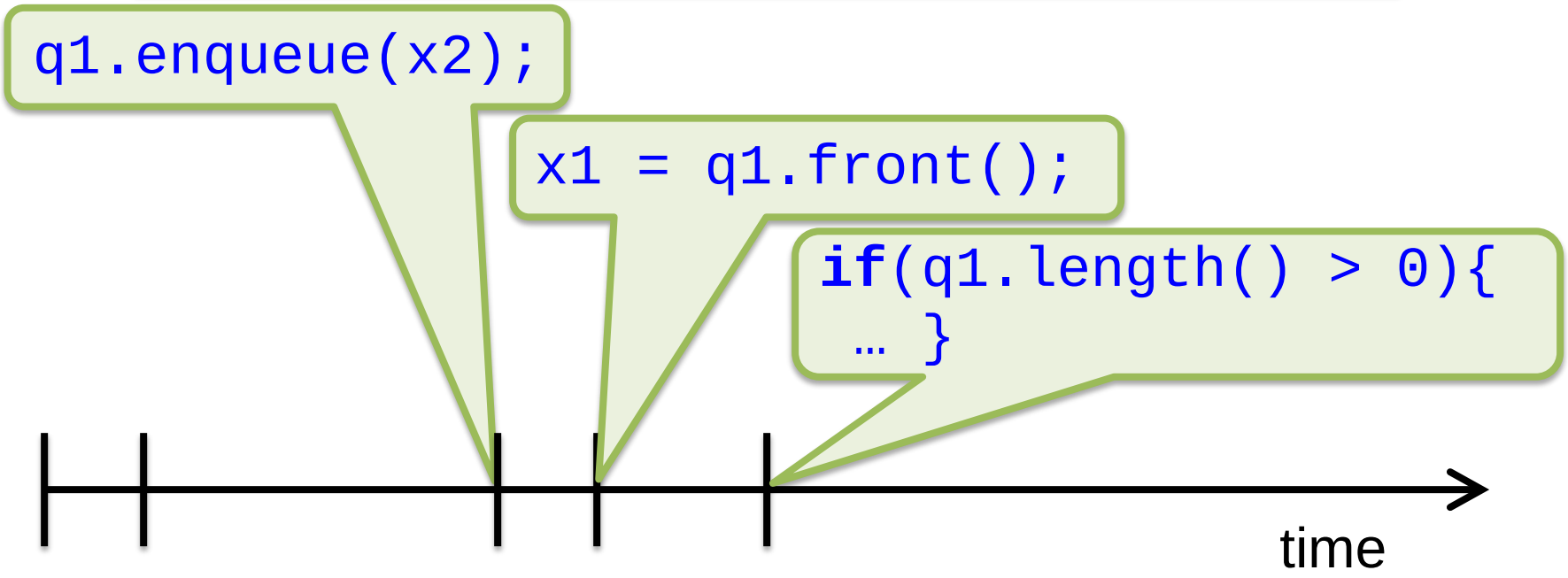


Variable Life-Cycle: Client

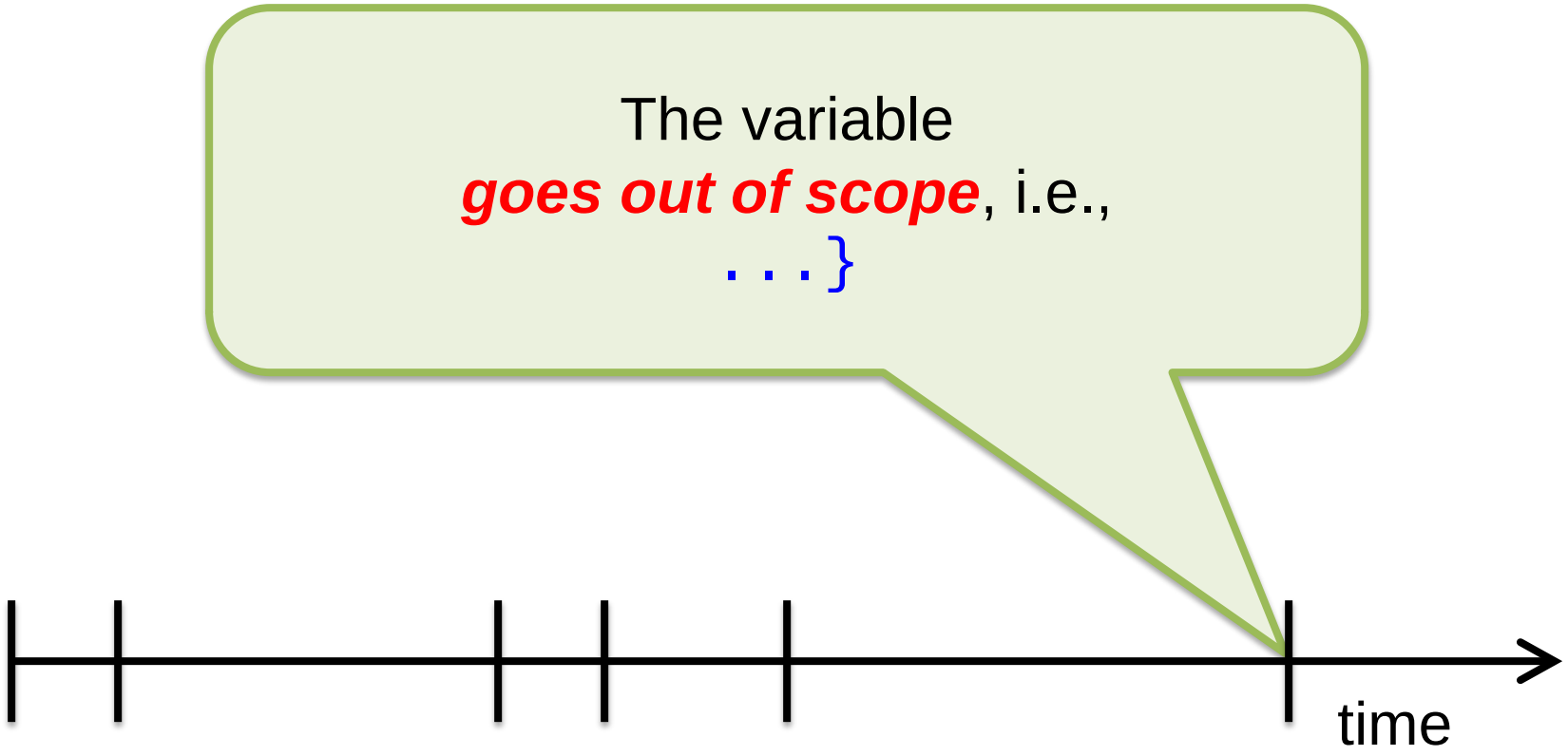


Variable Life-Cycle: Client

More member functions are called, for example:

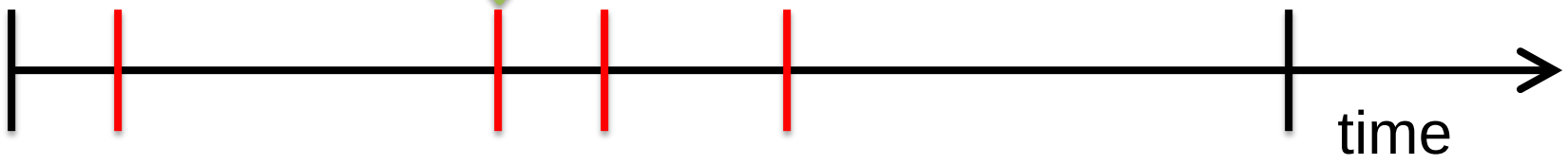


Variable Life-Cycle: Client



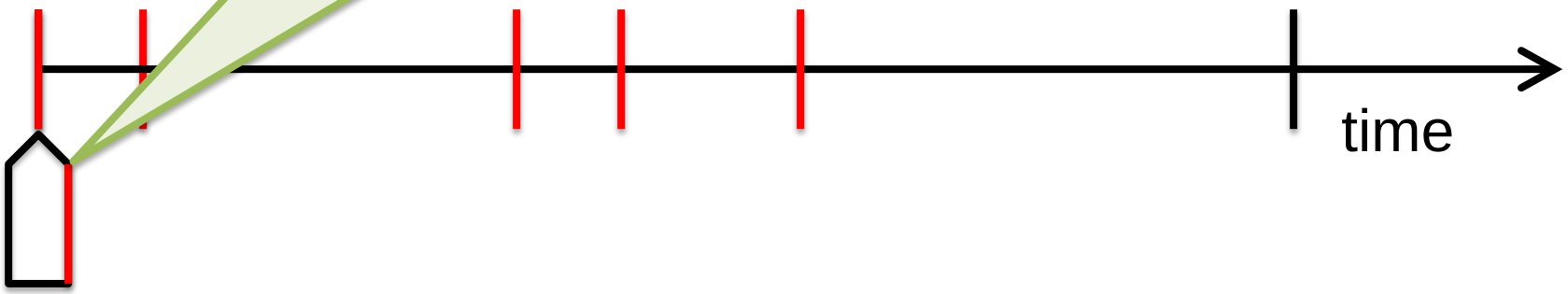
Variable Life-Cycle: Client

The claim of the kernel class implementer is that the representation invariant holds at the *end* of the constructor call and each subsequent member function call.

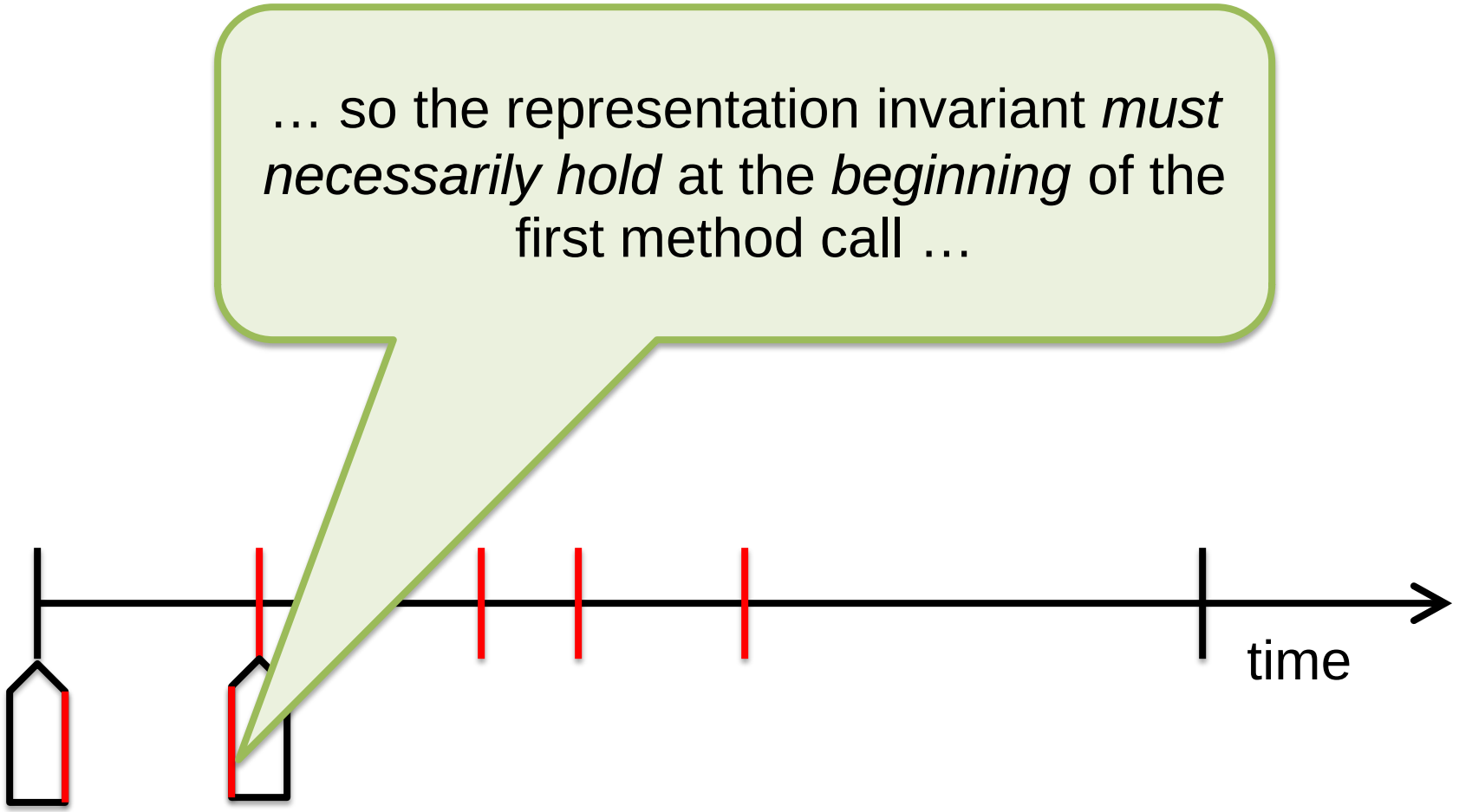


Variable Life-Cycle: Implementer

Now look *inside each call*.
Note that the constructor body must *make* the representation invariant hold at the end of the constructor …

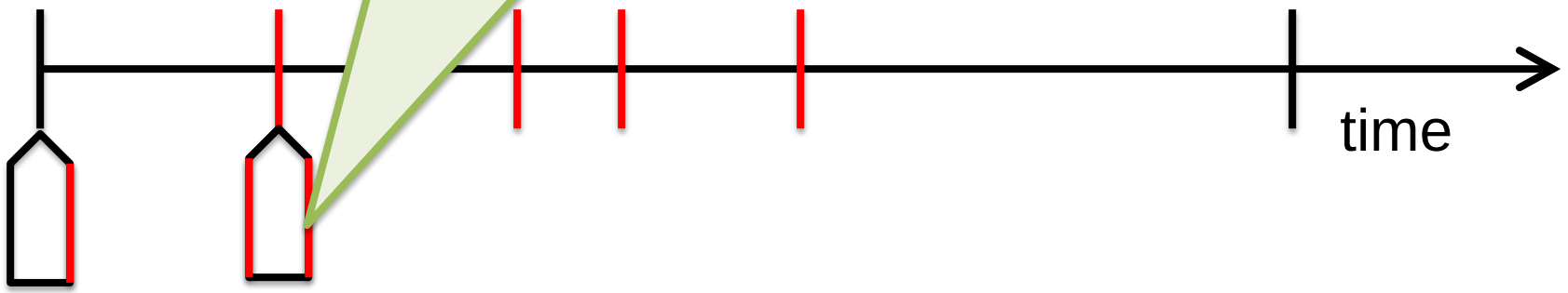


Variable Life-Cycle: Implementer



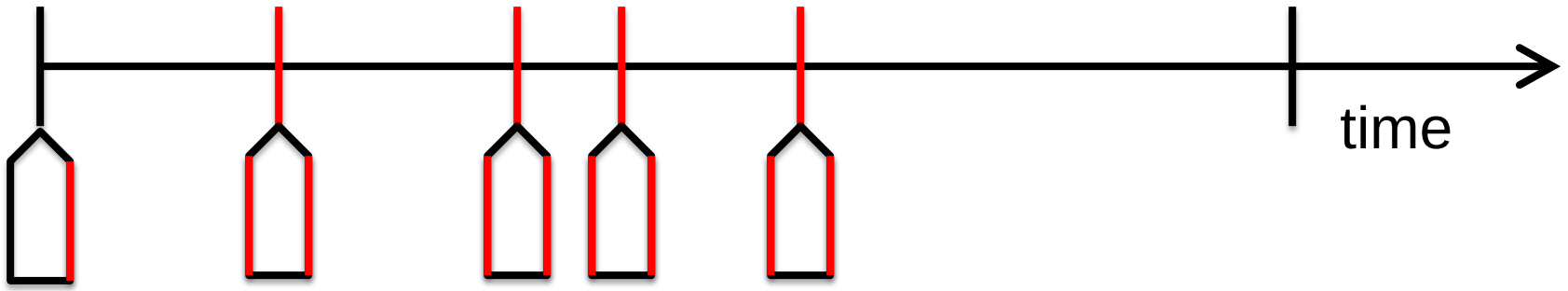
Variable Life-Cycle: Implementer

… and the code in the body for that method must *make* the representation invariant hold at the *end* of the first method call …



Variable Life-Cycle: Implementer

… and so on for each method call. The representation invariant therefore may be *assumed* to hold at the *beginning* of each method body, if the code makes it hold at the *end* of each method body!



The Representation Invariant

- To summarize, for a kernel class:
  - The constructor(s) must *make* the representation invariant true
  - The representation invariant may be *assumed* to be true at the *beginning* of each method body
  - Each method body (except the destructor) must *make* the representation invariant true (again) at the time it returns

What’s Left to Write Down?

The Abstraction Function

- The ***abstraction function*** describes how to *interpret* any concrete value (that satisfies the representation invariant) as an abstract value
- The abstraction function is not computed by any code, but is merely *recorded* in the ***correspondence*** clause in a comment for the kernel class

Consequences

- If the representation invariant and abstraction function are documented as suggested, then the work of implementing each constructor and each member function in a kernel class can be done independently, and all the code will still "work together"
  - The code for each constructor and each member function can be written by a different person!

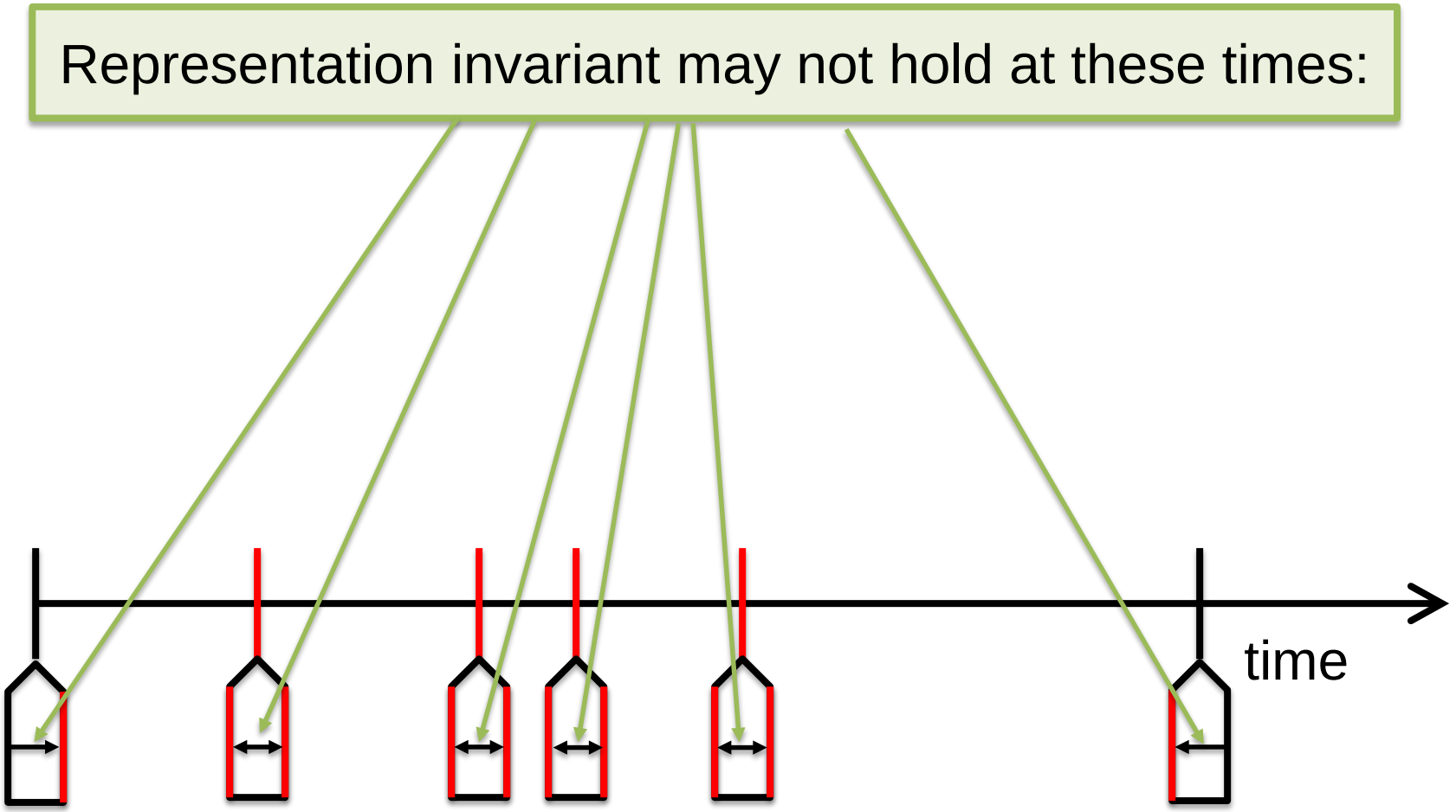
Kernel Purity Rule

- ***Kernel Purity Rule*** — No member function body in the kernel class should call any public method from the same component family
  - Every public member function in the component family relies (for its correctness) on the representation invariant being satisfied when it is called, and this might not be true when a call is made from inside a public member function of the kernel class

Kernel Purity Rule

- ***Kernel Purity Rule*** — Why do we need this rule?
  - Every public member function in the component family relies (for its correctness) on the representation invariant being satisfied when it is called
  - This might not be true when a call is made from inside a public member function of the kernel class

Variable Life-Cycle: Implementer



Implications Part 1

- Implications of the kernel purity rule:
  - No public kernel member function should call any other public member function method from the same class
  - No public kernel member function should call itself recursively
  - No member function (public or private) in the kernel class should call any layered or secondary member function from the same component family

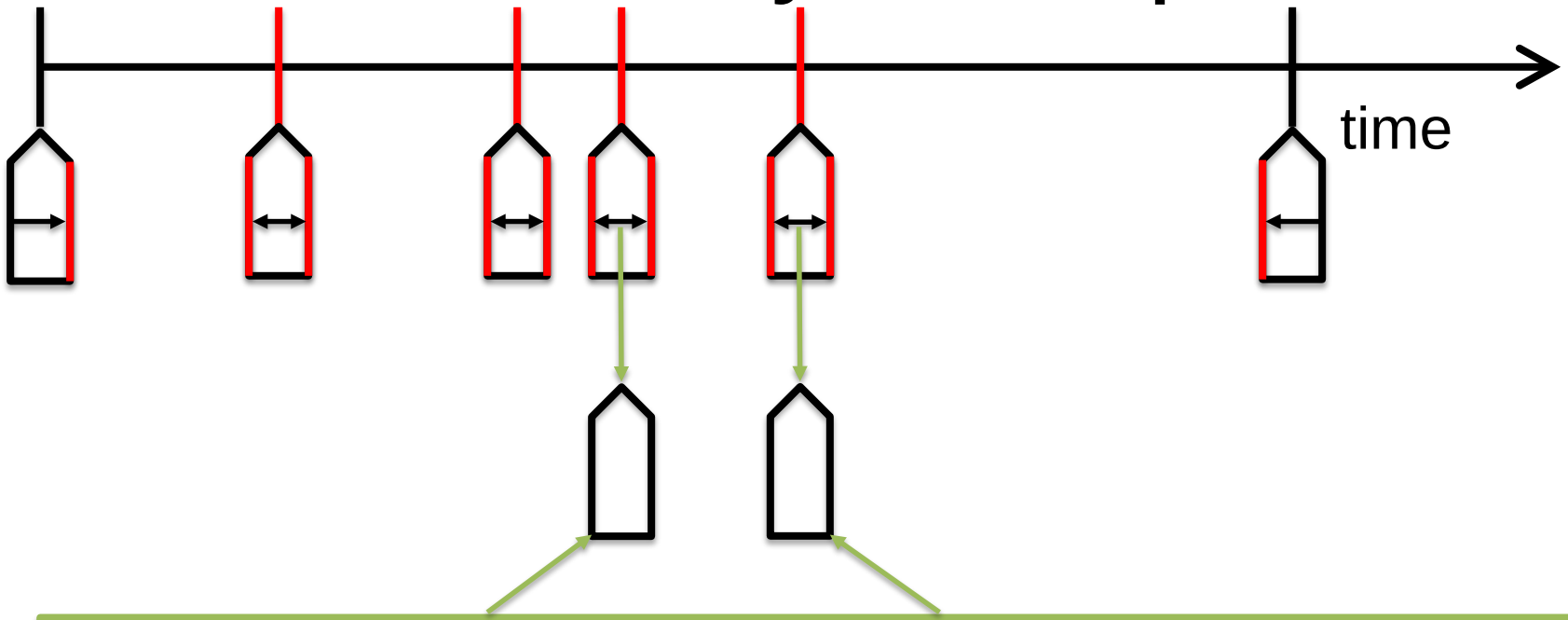
Implications Part 2

- Implications of the kernel purity rule:
  - Could lead to duplicated code in kernel member functions, except …
  - Duplicated code can be placed in private operations that follow the following rules:
    1. The operation has its own *requires* and *ensures* clauses
    2. If the operation depends on the representation invariant holding, then that must be stated in the operation's *requires* clause

Implications Part 3

- Implications of the kernel purity rule:
  - Could lead to no recursive implementations of operations, except …
  - Recursive operations are implemented as private operations that follow:
    1. The same rules for all private operations
    2. And the rules for correctly implementing recursive operations

Variable Life-Cycle: Implementer



Black sides of private operation indicates:

- It does not necessarily expect the representation invariant to hold
- It does not necessarily reestablish the representation invariant