The Sequential Initializer Tree Pattern

Martin Eisemann
eisemann@cg.cs.tu-bs.de
TU Braunschweig
Braunschweig, Germany

Fabian Friederichs
friederichs@cg.cs.tu-bs.de
TU Braunschweig
Braunschweig, Germany

Sascha Fricke
fricke@cg.cs.tu-bs.de
TU Braunschweig
Braunschweig, Germany

Marcus Magnor
magnor@cg.cs.tu-bs.de
TU Braunschweig
Braunschweig, Germany

ABSTRACT

We present the *Sequential Initializer Tree Pattern*, a creational design pattern for flexible and safe initialization of complex objects in programming. While the Sequential Initializer Pattern [7] provides only one yet safe way to initialize a complex object, the Sequential Initializer Tree Pattern provides an equally safe but more flexible way to initialize a complex object, yet inherits all the benefits of the original pattern, including readability, writeability, enforced correct usage and reduced cognitive load for the user. We show that applying this pattern is as simple as the original pattern, yet provides a more flexible way to safely initialize objects and therefore prevents cumbersome data transformations at code level.

CCS CONCEPTS

• **Software and its engineering** → *Software design techniques*; *Object oriented development*; **Software design engineering**.

KEYWORDS

pattern, design pattern, creational design pattern, builder pattern

ACM Reference Format:
Martin Eisemann, Fabian Friederichs, Sascha Fricke, and Marcus Magnor. 2023. The Sequential Initializer Tree Pattern. In *28th European Conference on Pattern Languages of Programs (EuroPLoP 2023), July 05–09, 2023, Irsee, Germany*. ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3628034.3628041

1 INTRODUCTION

Creational programming design patterns, a subset of the classical design pattern of the *Gang of Four* (GoF) [9] deal with the process of initializing complex objects. Several patterns exist that make this process more flexible and less error-prone. In programming the concept of RAII (Resource Acquisition is Initialization) became quite popular in recent years to avoid partly or non-initialized objects, especially in the Perl and C++ community [15], emphasizing the need for techniques for safe object initialization. *Named parameters* [13] provide an expressive way to explain the usage of each parameter of a constructor at the caller side without the requirement to switch to the manual or declaration of the object. The *Builder Pattern* [9] is one classical way to achieve this. It makes use of a director that orchestrates the builders to construct the complex object. Using different builders allows for different initialization processes, yet the director needs to be adjusted manually by the programmer, even though automatic generators exist [11], which exemplifies the importance of having well usable ways to initialize complex objects. Recently, the *Sequential Initializer Pattern* (SIP) [7] was introduced, a variant of the Builder Pattern, specialized on providing a safe and easy to use way that enforces a certain way of initialization, making it very difficult for a user to initialize an object wrongly. In this paper we present the *Sequential Initializer Tree Pattern* (SITP) which *generalizes* the SIP to make it more flexible yet equally safe. The SITP is, therefore, a creational design pattern, that provides specific initialization paths for a complex object but with all the benefits from the original SIP including correct usage enforcement for object initialization, automatic support by modern IDEs, the compiler throws an error if the code for object initialization is not correct, and many more benefits.

EuroPLoP 2023, July 05–09, 2023, Irsee, Germany

ACM ISBN 979-8-4007-0040-8/23/07.
https://doi.org/10.1145/3628034.3628041

2 INITIALIZING OBJECTS

In the following we will look at different ways to initialize an object and their drawbacks before introducing our solution in Sec. 3. For explanatory reasons we will use the running example of a simple rectangle in pseudo-code. A complete C++ code example is provided in the appendix A. Other examples are provided in the examples section 3.9.

The classical way to initialize an object in object-oriented programming is to use a constructor. A typical call for a rectangle class is provided in listing 1.

```
1 // Object initialization using a simple constructor
2 Rectangle r(10,20,300,400);
```

Listing 1: Initialization with a simple constructor

While this is probably the most common way to initialize such an object, it comes with various drawbacks in both readability and writeability. What do the parameters stand for? Are these the lower-left and upper right corner? Or are these one corner (which?) and the width and height? The user does not know until they read the manual which is cumbersome and not very time efficient. This is also one of the reasons why programmers of all skills try to avoid these kind of constructors [16]. Even though modern IDEs, such as IntelliJ or Visual Studio Code, provide in-place documentation it still

requires one to place the cursor or pointer on the respective code and can become problematic for more complex objects requiring more parameters for initialization. This is especially true, if the naming of the constructor's parameters is not expressive. Default values can often be only provided for all the parameters or only for the last parameters, but not arbitrarily. In addition, there is no feedback to the programmer if the last parameters have been forgotten or whether this was on purpose.

Named parameters [13] are therefore common in many modern programming languages, such as Python, C++, Swift, Kotlin or Dart. With these the constructor call can be made more expressive, as shown in listing 2

```
// Object initialization using named parameters
Rectangle r(xmin: 10, ymin: 20, width: 300, height: 400);
```

Listing 2: Initialization with named parameters

Now the meaning of the parameters becomes obvious. While this works well for simple objects, as our running example here, it becomes tedious for higher numbers of passed parameters, requires looking up the documentation frequently, and the danger of forgetting a certain parameter increases.

Especially the last aspect is also a problem for the *Builder Pattern* [9]. Listing 3 shows how the code for our running example could look like using the Builder pattern.

```
// Object initialization using a builder
// with named parameters
Rectangle r = RectangleBuilder
    .lowerLeftCorner(xmin: 10, ymin: 20)
    .width(300)
    .height(400)
    .build();
```

Listing 3: Initialization with the Builder pattern

The code is easy to understand at a glance, especially when combined with named parameters. The RectangleBuilder class provides several convenience functions to partially initialize the Rectangle object and a build-function to return the Rectangle in the end. Unfortunately, calling all these functions is not enforced in the Builder Pattern and the user can only hope that reasonable default values are used if some functions are not called, which can easily happen if the process and object become more complex as there is no feedback from the IDE or compiler.

To make the initialization process less error-prone the *Sequential Initializer pattern* [7] **enforces** the order in which the functions need to be called. The code looks exactly the same as in listing 3 but the user **cannot** forget to call any of these functions as the compiler will complain otherwise as each intermediate step only returns an intermediate object until after the last call the initialized complex object is returned. In addition, if provided, default values can be set for any of the parameters explicitly, see listing 4.

```
// Object initialization using the sequential initializer
// pattern incl. default values
Rectangle r = RectangleBuilder
    .lowerLeftCorner(xmin: 10, ymin: 20)
    .defaultwidth()
    .height(400);
```

Listing 4: Initialization with Sequential Initializer Pattern using default values for some parameters

The process of initializing a complex object is expressive and safe with the Sequential Initializer pattern. A problem that still persists is a loss of flexibility which occurs if one wants to provide several ways to initialize a complex object. One might object here, that it was the whole purpose of the Sequential Initializer pattern to do exactly this: Provide one and only one way to initialize an object. While this is true, there are situations in which one wants to have more than one way to initialize an object, especially when different internal representations are possible for the object.

Take our running example of the Rectangle class. Providing the lower left corner and width and height is probably a useful way to initialize it. But what if your data is in another format, e.g., it might be provided as the lower left and the upper right corner. Initializing the rectangle with the Sequential Initializer pattern would require a mapping of the original data onto the representation required by the implementation of the pattern. This is again an error-prone process. But couldn't we provide several implementations of the Sequential Builder pattern for the Rectangle class? Yes, but then the question arises, which one would be the correct one given a certain data representation? This is not simply decidable, even in our simple example, as the first function call could be exactly the same: to set the lower left corner. So the user would have to dig through the manual again and even a modern IDE would probably not help you much with in-place documentation.

Could the other initializing patterns help here? Unfortunately, not. The Builder pattern suffers from the same problems as the Sequential Initializer pattern in this case. Even worse, if not implemented properly the user might be able to mix the representations and come out with a wrongly initialized object. Another solution could be constructor overloading. In most modern programming languages one can have several constructors for the same class. While this may help in many cases, it does not in this case, as constructor overloading in many programming languages is only possible if the signature of the constructor changes, as otherwise the compiler can not decide which constructor should be called. Modern programming languages, such as Dart, provide the functionality of named constructors to circumvent this problem [6], where each constructor is given a unique name. This certainly solves the problem of same types and numbers of parameters but comes again with the same problems of standard constructors mentioned above and in [7]. What is needed is a safe, yet flexible way to initialize complex objects. To solve this problem we introduce the *Sequential Initializer Tree Pattern* in the following.

3 SEQUENTIAL INITIALIZER TREE PATTERN

The *Sequential Initializer Tree Pattern* is a software design pattern that provides several ways to initialize a complex object while at the same time enforces a complete and correct initialization. In its most simple case, where there is exactly one way to initialize an object, it *uses* the Sequential Initializer pattern [7] to initialize an object, following the classification scheme of [12]. At the same time it *generalizes* the SIP as it offers the flexibility to provide multiple ways to initialize an object by extending the idea of the SIP's *Step objects*, to partly initialize an object, such that one Step object can have multiple successors.

3.1 Context

Objects in object-oriented programming require concrete and correctly initialized instantiations of classes with sensible parameter values. And these values should be provided with an interface that is easy to understand, read, and write. Sometimes more than one option to initialize an object is required or useful, e.g., when different representations for an object exist. This is often tackled by providing several constructors for the object.

3.2 Problem

Complex objects have to be initialized properly, *but* sometimes one needs more than one way to do so which quickly becomes an error-prone process from a writing perspective that might require referring to the documentation frequently and an unnecessary complex process from a reading perspective. The user needs to choose one of the provided ways to initialize a complex object *but* with an increasing amount of ways to do so it becomes more and more cumbersome for the user to inform themselves about the different ways and to compare them to the data that is available in order to check if a proper way exists. Modern IDEs fail to help properly if the process is too complex and too many options exist. This complexity may also lead to partially initialized objects if important parameters are not set properly but the compiler generally has no way to enforce correct initialization. The problem becomes particularly clear for less experienced users, who might have problems to follow the tedious process of initialization of a complex object. One example could be to set up a window in a graphics library, that requires to set many parameters correctly, otherwise the window will not appear or stay black.

3.3 Forces

In the following we describe the problems we want to tackle in object initialization using the principle of contradicting forces [17].

- **Multiple ways of initialization:** One wants to initialize a complex object, *but* one may need specific ways to do so, in order to prevent unnecessary data mappings in the caller-code.
- **Complexity of usage:** The programmer needs to choose one way to initialize an object *but* there could be multiple choices which are not always clear and concise.
- **Correct usage:** All provided options to initialize an object should lead to a correctly initialized object, *but* most often the user can introduce errors in all of them.
- **Correct initialization:** It should not lie in the hand of the user to check whether everything was correctly initialized, *but* correct initialization cannot be checked properly by the compiler, in the general case. Maybe to some extend if the language provides sound null safety.
- **Cognitive load:** A programmer wants intuitive ways to initialize a complex object *but* the often complicated process *and* the potential multiple ways to do so require frequent referrals to the documentation or definition of the complex object.
- **Modern IDE support:** Modern IDEs should help the programmer to initialize an object properly *but* code completion and in-place documentation can get confusing and hard to read, depending on the complexity of the process.

3.4 Solution

Our solution is the introduction of the *Sequential Initializer Tree*, a data structure where each path from the root node to a leaf node represents one valid way to initialize a complex object. The proposed solution is based on the observation that the initialization of a complex object can be partitioned into several smaller steps, this is called partial initializations. These steps follow a semi-strict ordering, i.e., some steps need to be conducted before others, while for some it does not matter when they are conducted as long as it is guaranteed that they are conducted at some point. In addition, some steps might become obsolete after others have been conducted before.

As there are always two views on programming patterns, we will use the term *developer* for the person implementing the proposed pattern for a complex object and we will use the term *user* for the person who wants to initialize the complex object using the proposed pattern. Our goal is to represent the plethora of possibilities mentioned above in a way that the developer can handle this complexity and that a user of our pattern can decide on the ordering of initialization steps on a step-by-step basis but with the restriction that only sensible combinations are available to them. Finally, the user only gets access to the complex object once it has been completely initialized. In the following we first describe the theoretical concept of our solution (Sec. 3.5), then the impact on the user's view (Sec. 3.6) and finally how a developer can implement our solution (Sec. 3.7).

3.5 Concept

Let us call the different ways to completely initialize an object *paths*. A path provides all the required steps to initialize an object. In our running example one path could consist of the step to provide the lower left corner of the rectangle together with the steps to provide the width and height, another path would consist of the step to provide the lower left corner and then the step providing the upper right corner. Within these paths the ordering of the steps is often less important and could be permutated, but one has to make sure that no step within a valid path is conducted twice or left out.

To describe all paths desired by the developer, i.e., all sensible and valid combinations of successive steps to create an object, we introduce the *Sequential Initializer Tree* a tree-data structure, where each path from the root node towards a leaf node represents a possible path to initialize the complex object. Each node is either a step in the initialization process or a special purpose node for a more compact representation, which we will describe in the following. A *X(or)-Node* (exclusive or) forces to choose a single child during the traversal from the root to the leaf node. A *P(ermutation)-Node* allows any permutation of child nodes as next steps in the initialization process, similar to P-Nodes in a PQ tree [2]. The Figure 1 shows the Sequential Initializer Tree for our running example of initializing a rectangle.

To map this representation to an actual implementation provided by a developer to the user, we make use of an intermediate representation, the expanded Sequential Initializer Tree, shown in

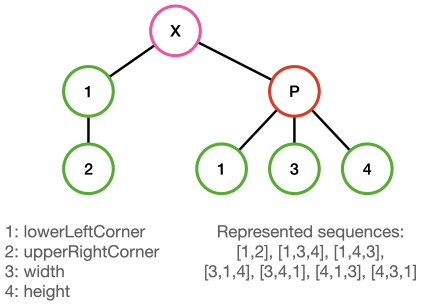

Figure 1: The Sequential Initializer Tree. Each path from the root to a leaf represents a single way to initialize the complex object. The X(or)-Node enforces to choose a single child in the traversal, the P(ermutation)-Node allows any permutation of the child nodes which then have to be executed in succession. The step nodes (green) represent different steps in the initialization process. Here our running example of initializing a rectangle is used.

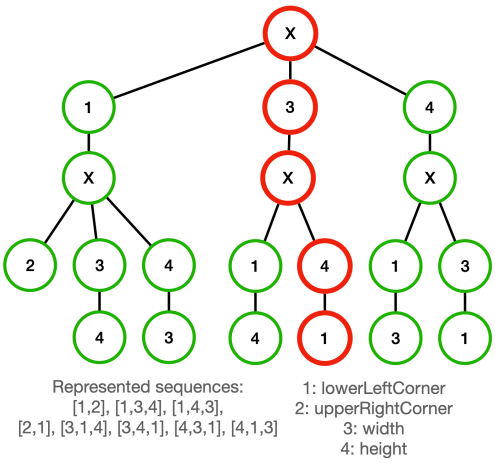

Figure 2: The expanded Sequential Initializer Tree shown here is the explicit representation of the Sequential Initializer Tree in Figure 1. Here any path from the root to the leaf node represents a possible initialization scheme of the complex object (one such path is highlighted in red). This tree can be effectively mapped to actual code.

Figure 2. The expanded Sequential Initializer Tree is an explicit representation of the Sequential Initializer Tree that can be efficienctly mapped to actual code.

In this representation we only have two types of nodes. An X(or)-node, whose purpose is simply to expose its child nodes to the user, so that they can choose which one to call next and pass the respective parameters to it. And a step node, which are responsible to initialize the respective part of the complex object and to return the next node in the sequence, equivalent to the Step objects in [7]. For example choosing the red path shown in Figure 2 would result in the user-side code shown in listing 5 (though the passed parameters can differ).

```
Rectangle r = RectangleBuilder
  .width(300)
  .height(400)
  .lowerLeftCorner(xmin: 10, ymin: 20);
```

Listing 5: Initialization with Sequential Initializer Pattern following the red path in Figure 3

3.6 User's view

Even though the Sequential Initializer Tree provides a vast number of options to initialize a complex object, the user only has to call one function after the other, choosing from a small subset and can never call the same initializing function twice or non-related initializing functions. I.e. every choice of path results in a valid initialization of the complex object.

At any time before a leaf node is reached the user has only access to the current node and its initializer function. The initializer function conducts the respective partial initalization of the complex object and returns the next node along the path until the leaf node returns the correctly initialized object.

The user-side code for all possible initializations using the Sequential Initializer Tree from Figure 2 is shown in listing 6. It should be noted that in practice it might be beneficial to provide less paths to the user and the developer should think about valid paths carefully. Optionally, one could add a finish-/create-node to each leaf which only provides a finish()- or create()-function to return the object, to make it obvious that the initialization is finished.

```
// Different variant to initialize a rectangle based on
// the Sequential Initializer Tree pattern
Rectangle r1 = RectangleBuilder
  .lowerLeftCorner(xmin:10, ymin:20)
  .upperRightCorner(xmax: 310, ymax: 420);

Rectangle r2 = RectangleBuilder
  .lowerLeftCorner(xmin:10, ymin:20)
  .width(300).height(400);

Rectangle r3 = RectangleBuilder
  .lowerLeftCorner(xmin:10, ymin:20)
  .height(400)
  .width(300);

Rectangle r4 = RectangleBuilder
  .width(300)
  .lowerLeftCorner(xmin:10, ymin:20)
  .height(400);

Rectangle r5 = RectangleBuilder
  .width(300)
  .height(400)
  .lowerLeftCorner(xmin:10, ymin: 20);

Rectangle r6 = RectangleBuilder
  .height(400)
  .lowerLeftCorner(xmin:10, ymin: 20)
  .width(300);

Rectangle r7 = RectangleBuilder
  .height(400)
  .width(300)
  .lowerLeftCorner(xmin:10, ymin: 20);
```

Listing 6: Potential usage of the Sequential Initializer Tree Pattern to create a rectangle object

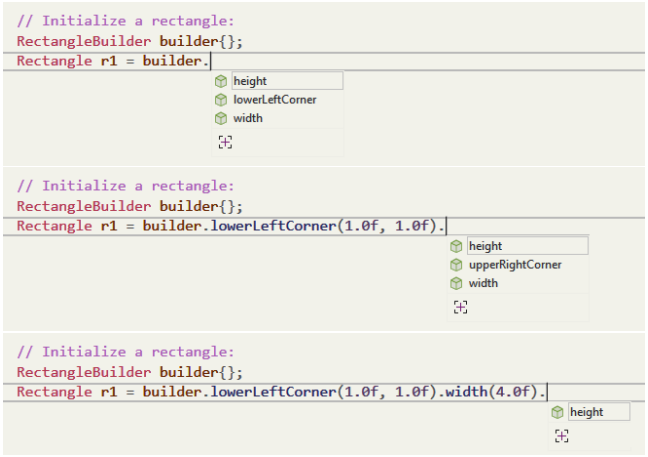

Figure 3: Typing experience in modern IDEs when using the SITP. The IDE proposes only those functions that are relevant at the current point in the initialization process.

Please remember that the user has only the choice to choose reasonable function calls depending on their previous function calls. This makes the pattern very easy to use especially with modern IDEs. An example of the typing experience is shown in Figure 3 and a live programming video example can be downloaded here: https://youtu.be/wFLdJu57G9U.

3.7 Developer's View

In the following we provide a prototype implementation of the SITP using generalization, variadic templates and template metaprogramming [1, 5, 10]. Listing 7 shows the basic structure of the step node for our running example, which simply contains the reference to the complex object to be initialized and a next function returning the next node along the path of the Sequential Initializer Tree, which in the case of a step node is unique.

```
class Step<Next>{
  // reference to the complex object
  Rectangle rect;
  // reference is passed on through the constructor
  Step(Rectangle r) : rect(r){};
  // return next step in the initialization tree
  // for step nodes there is only one next step
  Next next(){ return Next(rect); }};
```

Listing 7: Template of the step node in pseudo-code

The type of `rect` could be templated as well, but we kept it simple for our running example, see appendix A for a general implementation. A concrete implementation of the step node for the lower left corner of our running example is shown in listing 8 which inherits from the `Step` object, sets the respective values and returns the next node along the path in the tree.

```
class RlowerLeftCorner implements public Step<Next>{
    // reference to the complex object is passed on
    // to the base class constructor
    RlowerLeftCorner(Rectangle r) : Step<Next>(r){}
    // implementation of the provided
    // initializing function
    Next lowerLeftCorner(float x, float y){
        this->rect.minx = x;
        this->rect.miny = y;
        return this->next();
}};
```

Listing 8: Concrete step node for setting the lower left corner in pseudo-code

In languages supporting multiple inheritance the Xor-Node simply inherits from all child nodes in the tree as this exposes the interfaces of the child nodes to the user, see listing 9. Note the "..." in the code which stands for the variadic templates and therefore an arbitrary number of `Step` objects the Xor-Node inherits from.

```
class Xor implements public Next ...{
  Xor(Rectangle r) : Next(r)... {}
};
```

Listing 9: Xor-Node in pseudo-code

In languages not supporting multiple inheritance one needs to create a concrete class that implements all the interfaces provided by the child nodes for every Xor-node in the Sequential Initializer Tree. A complete implementation of our running example in C++-Code is given in the appendix.

3.8 Consequences

Benefits:

- **Multiple ways of initialization:** With the Sequential Initializer Tree one can represent multiple ways to safely initialize a complex object, in order to prevent unnecessary data mappings in the caller-code.
- **Complexity of usage:** The presented pattern allows for various ways to initialize an object, but all paths reflect a valid initialization path in the Sequential Initializer Tree and the complexity is constant and low as the user only needs to choose one function from a small set of functions in each initialization step.
- **Correct usage:** All of the paths within the Sequential Initializer Tree are strongly compiler enforced (for strongly typed languages). There is no chance to choose a wrong path that leads to a wrongly initialized object.
- **Correct initialization:** Each path in the Sequential Initializer Tree correctly and fully initializes the complex object. There is no way to access the complex object without following one of these paths. This can even be guaranteed for circular dependencies [14].
- **Cognitive load:** The proposed pattern will lead the user through the initialization steps. As each step is expressive in itself, the need to refer to the manual or definition of a class is diminished.
- **Modern IDE support:** Modern IDEs can use code completion and in-place documentation to provide the user with helpful information to initialize an object. This becomes even more helpful as the number of possible functions to call in each step of the Sequential Initializer Tree is very limited.

Liabilities:

- **Correct usage:** All information about the object to create should be available at the time it is instantiated. Otherwise the user would have to handle the step-objects hiding the

partially initialized object from the user until the remaining information becomes available.

- **Partial or wrong initialization:** Partial or wrong initialization could potentially be made possible by the developer, but that would simply be a bad implementation of the pattern. Partial initialization is forbidden by design or only possible through passing uninitialized references/pointers as parameters, which could be handled by the step-objects by throwing an exception.
- **Implementation Complexity:** The implementation of the pattern is quite involved but can be elegantly solved with variadic templates. The usage on the other hand is very simple, as the pattern guides you through the initialization process.
- **Completeness:** It is of utmost importance that when implementing the SITP for others to use, that all possible paths for a given set of steps are implemented. As otherwise initializing objects could become very tedious as the user needs to figure out which paths through the SIT actually exist.
- **Cost of Change:** Whenever a class using the SITP changes, the SITP implementation needs to be adjusted as well. Due to the implementation using multiple inheritance, this usually requires only to adjust the tree and implement a new step class if required.

3.9 Examples

The concept of semi-strict ordering to achieve a certain goal in the SITP can be found in the real-world without actually noticing it. E.g. many *cooking recipes* follow the SITP when writing "pour ingredients X, Y, and Z into the bowl, then stir", it actually does not matter, whether X is poured before Y or Z if not explicitly stated, as long as the stirring begins only after. Another example is *cleaning a child's room*. One certainly has to tidy up the toys from the ground before vacuum cleaning the floor, but it doesn't matter in which order the toys are tidied up. And as a third example, *writing* could be seen as an example for the SITP. Take the sentence "The dog wags its tail, because it is happy". The ordering of the words is important, otherwise the sentence might not make sense anymore, such as "Because dog happy is it its tail the wags" and someone would complain about this (in the software implementation of the SITP this is the compiler). On the other hand the ordering is semi-strict as the sentence "Because the dog is happy, it wags its tail." makes perfect sense.

In the following we give some more coding related examples where the usage of the Sequential Initializer Tree Pattern proves useful. Generally the Sequential Initializer Tree Pattern also strongly improves readability of code, as each step in the initialization process is neatly described.

Ordering: In many areas the semantic mapping is different from the mapping of data in memory. A typical example for this are matrices, which are a basic building block to implement complex algorithms in math, graphics, computer vision, or artificial intelligence. A common problem is transforming the matrices from one library matrix representation to another. Let's assume we want to initialize a 2×2 matrix. The code could look like in listing 10.

```
Matrix2D m(1.1, 2.2, 3.3, 4.4);
```

Listing 10: Bad example for a matrix initialization

Why is it bad? Because it is not clear if the Matrix class uses row or column-major format internally. And even if the user looks it up in the documentation, they need to remember what row or column-major actually means as these terms are often confusing and hard to remember.

Using the Sequential Initializer Tree pattern the code could look like listing 11. Two options are given using rows or columns.

```
// row-based initialization
Matrix2D m = Matrix2DBuilder
  .row1(1.1, 2.2)
  .row2(3.3, 4.4);

// column-based initialization
Matrix2D m = Matrix2DBuilder
  .col1(1.1, 3.3)
  .col2(2.2, 4.4);
```

Listing 11: Expressive example for a matrix initialization using the Sequential Initializer Tree

It immediately becomes clear to the reader whether the first two values describe the first row or column of the matrix because it is written directly in the code, making the code much more expressive. Neither can the user forget to set a row or column nor can they mix both representations as none such path exists in the Sequential Initializer Tree.

Conversion: In many cases one data value can have different representations, e.g. a color can be represented using RGB, YUV, Lab or other representations. Using the Sequential Initializer Tree pattern all possible initializations could be easily provided as in listing 12.

```
// Bad, what do the parameters mean?
Color c(0.5, 0.3, 0.1);

// Good, intention is clear
Color c1 = ColorBuilder.R(0.5).G(0.3).B(0.1);
Color c2 = ColorBuilder.RGB(0.5, 0.3, 0.1);
Color c3 = ColorBuilder.YUV(0.3, 0.25, 0.7);
```

Listing 12: Comparison of a simple constructor and initialization using the Sequential Initializer Tree for colors

Another common example in computer graphics would be 3D positions which can be expressed in terms of xyz-coordinates or polar coordinates where one defines two angles θ and ϕ and a radius r, see listing 13.

```
// Bad, what do the parameters mean?
Pos3D p(0.5, 0.3, 0.1);

// Good, intention is clear
Pos3D p1 = Pos3DBuilder.X(0.5).Y(0.3).Z(0.1);
Pos3D p2 = Pos3DBuilder.XYZ(0.5, 0.3, 0.1);
Pos3D p3 = Pos3DBuilder
            .RadiusThetaPhi(0.59, 80.27, 30.96);
Pos3D p4 = Pos3DBuilder
            .Theta(80.27)
            .Phi(30.96)
            .Radius(0.59);
```

Listing 13: Comparison of a simple constructor and four variants of initialization using the Sequential Initializer Tree for 3D positions

Settings: There are many cases which bear some resemblence to the SITP but would benefit from it. E.g., whenever certain settings are required the SITP could help to ensure all required settings have been set. This is a typical scenario when writing graphics applications and instantiating a window handler. An example is given in listing 14 for the well-known OpenGL Utility Toolkit.

```
// Bad, some important settings could be forgotten
glutInit(&argc, argv);
glutInitDisplayMode(GLUT_DEPTH| GLUT_DOUBLE| GLUT_RGBA);
glutInitWindowPosition(100,100);
glutInitWindowSize(512,512);
glutCreateWindow("My Window");

// Good, all important information is queried
// automatically from the user
GlutAppBuilder.
  .withDoubleBuffer()
  .withRGBA()
  .withDoublePrecision()
  .withDepth()
  .atPosition(100, 100)
  .withResolution(512, 512)
  .title("My Window")
  .create();
```

Listing 14: Comparison of initializing a window using GLUT and the fluent interface using the Sequential Initializer Tree

3.10 Related patterns

- **Named Parameters** [13]: Programming languages that incorporate Named Parameters enable the assignment of parameter names to the variables passed during a function call, often imposing it as a requisite. This facilitates the allocation of parameters in arbitrary order and enables default values to be set for each parameter, not just the last ones. Nevertheless, it conceals the usage of default values for each parameter, a limitation that is overcome in the Sequential Initializer Tree pattern. Furthermore, the Sequential Initializer Tree pattern offers the added benefit of conveying the intent of each parameter in a more explicit manner.
- **Currying** [4]: The process of converting a multi-argument function into a series of single-argument functions is referred to as currying. For instance, given $x = f(a, b)$, currying can transform it into $h = g(a)$ and $x = h(b)$, or more concisely, $x = g(a)(b)$. Currying itself is primarily used in functional programming and theoretical proofs, such as in Haskell. It does not modify internal states, a requirement that is vital in the context of object instantiation. Furthermore, currying is mainly used for partial function initialization, i.e., binding parameters to functions, in a sense the SITP can be seen as a kind of currying for object initialization.
- **Method Chaining/Cascading** [8]: Modifier methods return the host object so that additional modifiers can be applied in a single expression. Code-wise this looks similar to the SITP but the method chaining/cascading does not enforce a particular order or prevent misusage. Fluent interfaces use method chaining/cascading for object initialization.
- **Pipeline pattern** [3]: The Pipeline pattern establishes a robust pipeline that can be decomposed into a sequence of steps, each executing a computation that operates on the output of the preceding step. In contrast to our pattern, the Pipeline pattern disallows any additional parameters from being passed between stages, and is usually fixed without any possibility of configuration, rendering it unsuitable for most object initialization tasks. Typically, the Pipeline pattern is used to process data in a sequential, multi-stage manner, rather than for initializing objects.
- **Builder pattern** [9]: The Builder pattern is a powerful solution to the challenges of generating multiple representations of a complex object within a single construction process and streamlining the creation of a class that encompasses complex object creation. Typically, a director is used to oversee the creation process and call upon various builders. Although the Builder pattern has a wide range of applications, our paper focuses on its primary objective, which is object initialization. In contrast to the Sequential Initializer pattern [7], which enforces strict usage, the desired flexibility of the Builder pattern can lead to usage errors. Our pattern combines the best of both worlds, flexibility of initialization with a strict enforced usage.
- **Expression Builder** [8]: The Expression Builder provides a so-called fluent interface as a separate layer on top of a regular API to improve readability. It is a specialization of the Builder pattern for the context of object initialization and is related to Method Chaining. While the goal is similar to SITP the Expression Builder follows the concept of the standard builder pattern in that it generally does not enforce an ordering or ensures correct initialization, therefore function calls could be applied more than once or be left out.
- **The Billion Dollar Fix** [14]: The Billion Dollar Fix provides a safe modular circular initialisation with placeholders and placeholder types for circular structures (structures where one element needs a reference to another so that they form a circle in the end). This avoids the danger of temporarily uninitialized pointers, which are unavoidable when initializing these circular structures directly. The SITP could be used to hide the dangers of initialization also for specific circular structures from the user, however, inside the implementation of the SITP one would still encounter the same dangers or would have to use the Billion Dollar Fix internally.
- **Abstract Factory** [9]: The Abstract Factory pattern offers a means of generating collections of related objects while abstracting their concrete classes behind an interface. The primary objective of this pattern is to use polymorphism in order to hide the internal representation of the objects created. In contrast, our aim is to explicitly instantiate a specific object and to provide transparency to the user regarding the creation process.
- **Factory Method** [9]: The Factory Method pattern abstracts the complexities of object instantiation by offering a designated method for creation. The Sequential Initializer Tree pattern, on the other hand, is an extended iteration of the Factory Method pattern, that enhances readability, expressiveness, flexibility and reduces the workload of the user by breaking down the process into multiple function calls that modern IDEs can suggest to the user in a step-by-step manner.

4 CONCLUSION

We presented the Sequential Initializer Tree pattern, a creational design pattern for programming to safely initialize complex objects in various ways. The pattern allows to design several feasible paths to initialize a complex object and enforces the user to follow one of these paths based on their previous choices. This becomes especially useful if different representations for a complex object exist and one wants to avoid explicit conversion. The benefits of using the pattern are especially: Enforced correct usage and modern IDE support which guides through the different initialization steps leveraging cognitive load for the user, therefore, providing a safe yet flexible way to initialize complex objects. We believe the SITP pattern could become even more powerful as soon as programming languages support it natively, e.g., using new language constructs and keywords to implement this pattern instead of constructors. Till then it should be possible to write automated generators to lift the burden of writing all the required step classes manually.

ACKNOWLEDGMENTS

We would like to thank our shepherd Stefan Sobernig for his tremendous effort to improve the paper and everyone from the writers workshop for the helpful feedback.

REFERENCES

[1] Andrei Alexandrescu. 2001. *Modern C++ Design, Generic Programming and Design Patterns Applied.* Pearson Education, London, UK.
[2] Kellogg S. Booth and George S. Lueker. 1976. Testing for the consecutive ones property, interval graphs, and graph planarity using PQ-tree algorithms. *J. Comput. System Sci.* 13, 3 (1976), 335–379.
[3] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. 2007. *Pattern-Oriented Software Architecture.* Wiley, United States.
[4] Haskell Curry and Robert Feys. 1958. *Combinatory logic. Vol. I (2 ed.).* North-Holland Publishing Company, Amsterdam, Netherlands.
[5] Krzysztof Czarnecki and Ulrich Eisenecker. 2007. *Generative Programming: Methods, Tools, and Applications.* Addison Wesley, United States.
[6] Dart Docs. 2023. https://dart.dev/language/constructors#named-constructors Named Constructors.
[7] Martin Eisemann. 2022. The Sequential Initializer Pattern. In *Proc. European Conference on Pattern Languages of Programs (EuroPLoP).* Featured in ACM Showcase / Kudos.
[8] Martin Fowler and Rebecca Parsons. 2011. *Domain-specific Languages.* Addison-Wesley.
[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. 1995. *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison Wesley, United States.
[10] Douglas Gregor and Jaakko Järvi. 2008. Variadic Templates for C++0x. *Journal of Object Technology* (2008), 31—-51.
[11] Tom Misawa. 2019. https://riversun.github.io/java-builder/ Java Builder Pattern Generator Online.
[12] J. Noble. 1998. Classifying relationships between object-oriented design patterns. In *Proceedings 1998 Australian Software Engineering Conference (Cat. No.98EX233).* 98–107.
[13] United States Department of Defense. 1983. *Reference Manual for the Ada Programming Language.* United States Department of Defense, United States.
[14] Marco Servetto, Julian Mackay, Alex Potanin, and James Noble. 2013. The Billion-Dollar Fix: Safe Modular Circular Initialisation. In *Proceedings of the 27th European Conference on Object-Oriented Programming* (Montpellier, France) *(ECOOP'13).* Springer-Verlag, Berlin, Heidelberg, 205–229.
[15] Bjarne Stroustrup. 1994. *The Design and Evolution of C++.* Addison Wesley, United States.
[16] Jeffrey Stylos and Steven Clarke. 2007. Usability Implications of Requiring Parameters in Objects' Constructors. In *Proceedings of the 29th International Conference on Software Engineering (ICSE '07).* IEEE Computer Society, USA, 529–539.
[17] Valentino Vranić and Aleksandra Vranić. 2019. Drama Patterns: Extracting and Reusing the Essence of Drama. In *Proceedings of the 24th European Conference on Pattern Languages of Programs* (Irsee, Germany) *(EuroPLop '19).* Association for Computing Machinery, New York, NY, USA, Article 4, 9 pages.

A COMPLETE SOURCE CODE

```
// our "complex" object
struct Rectangle
{
    float minx{};
    float miny{};
    float maxx{};
    float maxy{};
};


// Builder for the Sequential Initializer Tree
template<typename ComplexObject>
struct SeqTreeBuilder{
private:
    ComplexObject object;
public:
    SeqTreeBuilder() = default;

    // Template of the Xor-Node
    // using multiple inheritance
    template<typename ... Next>
    struct Xor : public Next ...{
        explicit Xor(ComplexObject& o) : Next(o)... {}
    };

    // Template of the step node
    template<typename Next>
    struct Step{
        ComplexObject& obj;
        explicit Step(ComplexObject& o) : obj(o){};
    protected:
        Next next(){ return Next(obj); }
    };

    // Concrete implementations of the step nodes
    template<typename Next>
    struct RLowerLeftCorner : public Step<Next>{
        explicit RLowerLeftCorner(ComplexObject& o)
            : Step<Next>(o){}
        Next lowerLeftCorner(float x, float y){
            this->obj.minx = x;
            this->obj.miny = y;
            return this->next();
        }
    };

    template<typename Next>
    struct RUpperRightCorner : public Step<Next>{
        explicit RUpperRightCorner(ComplexObject& o)
            : Step<Next>(o){}
        Next upperRightCorner(float x, float y){
            this->obj.maxx = x;
            this->obj.maxy = y;
            return this->next();
        }
    };

    template<typename Next>
    struct RWidth : public Step<Next>{
        explicit RWidth(ComplexObject& o)
            : Step<Next>(o){}
        Next width(float w){
            this->obj.maxx = this->obj.minx + w;
            return this->next();
        }
    };

```

```
    template<typename Next>
    struct RHeight : public Step<Next>{
        explicit RHeight(ComplexObject& o)
            : Step<Next>(o){}
        Next height(float w){
            this->obj.maxy = this->obj.miny + w;
            return this->next();
        }
    };

    // optional: create node, whose only purpose is to
    // give feedback to the user,
    // that the initialization is finished
    template<typename Next>
    struct RCreate : public Step<Next>{
        explicit RCreate(ComplexObject& o)
            : Step<Next>(o){}
        Next create(){
            return this->next();
        }
    };


    // definition of the actual
    // Sequential Initializer Tree
    using SequentialInitializerTree =
    Xor<
      RLowerLeftCorner<
        Xor<
          RUpperRightCorner<RCreate<Rectangle>>,
          RWidth<RHeight<RCreate<Rectangle>>>,
          RHeight<RWidth<RCreate<Rectangle>>>
        >
      >,
      RWidth<
        Xor<
          RLowerLeftCorner<RHeight<RCreate<Rectangle>>>,
          RHeight<RLowerLeftCorner<RCreate<Rectangle>>>
        >
      >,
      RHeight<
        Xor<
          RLowerLeftCorner<RWidth<RCreate<Rectangle>>>,
          RWidth<RLowerLeftCorner<RCreate<Rectangle>>>
        >
      >
    >;
    SequentialInitializerTree build(){
        return SequentialInitializerTree(object); }
};
```

Listing 15: Full C++ example code: Rectangle and Sequential Initializer Tree

```
int main(){
    SeqTreeBuilder<Rectangle> builder;
    // all three variants are valid ways
    // to initialize a rectangle
    Rectangle r1 = builder.build()
        .lowerLeftCorner(1.0f, 1.0f)
        .width(4.0f)
        .height(3.0f)
        .create();

    Rectangle r2 = builder.build()
        .width(5.0f)
        .height(2.0f)
        .lowerLeftCorner(1.0f, 1.0f)
        .create();

    Rectangle r3 = builder.build()
        .height(3.0f)
        .lowerLeftCorner(1.0f, 1.0f)
        .width(4.0f)
        .create();
}
```

Listing 16: Full C++ example code: main function

Received 20 June 2023; revised 19 September 2023; accepted 30 September 2023