

Triggered Placeholder

An Object Creational Pattern for Delaying Object Creation

Joe Hoffert

joeh@cs.wustl.edu

Distributed Programing Environments Group

Department of Computer Science,

Washington University, St. Louis, MO. 63130, U.S.A.

Abstract

There are times when delaying an object's creation is desirable to avoid untimely side effects or the use of resources before they are actually needed. It is non-trivial to receive a request to create an object and delay that object's creation until a later time. The Triggered Placeholder pattern decouples a request for an object's creation from its actual creation. An example usage is shown along with the benefits and liabilities of using the pattern. An implementation outline is also provided along with some sample code. Finally, patterns related to the Triggered Placeholder pattern are listed.

1.0 Intent

Delay the creation of an object until a particular method on the object has been invoked.

2.0 Also Known As

Temporal proxy *[maybe not applicable yet?]*

3.0 Classification

Object Creational

4.0 Motivation & Context

There are times when the creation of an object is delayed to avoid the untimely side effects of object creation or to avoid the use of resources before they are actually needed. A request for an object may be received but the object may not actually be needed at that moment. Creating the object at request time may use resources that could be used elsewhere when the object will not access or use those resources until a later time. The motivation for delayed creation in this instance is efficient resource management.

The creation of an object may also create undesirable side effects since its creation may affect other parts of the system. For instance, the creation of an element in a container object such as a list or an array has ramifications on the list or array. There may be times when a request is made to create an element object but the containing object should not yet be affected. The motivation for delayed creation in this instance is management of creation side effects.

5.0 Example Usage

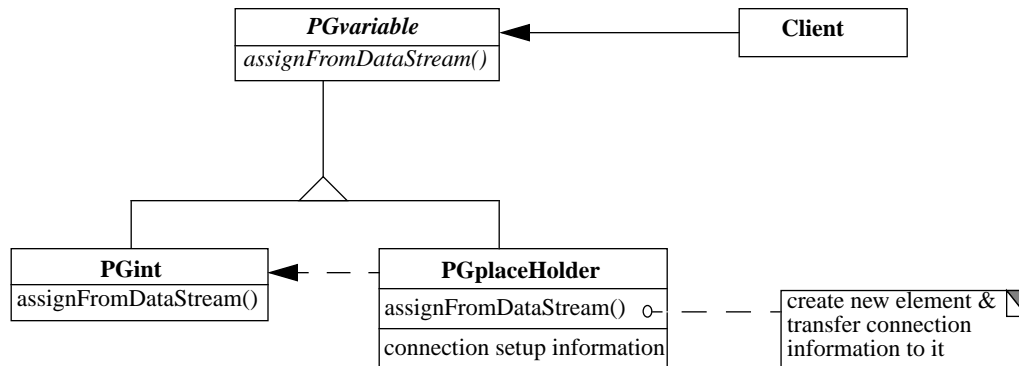


FIGURE 1. The Playground Example

In the Playground distributed programming environment [1], element-to-aggregate functionality allows a connection to be made from a variable of a certain type in one distributed component to an aggregate variable in another distributed component whose elements are of the same type. For instance, an element-to-aggregate connection can be made from a *PGint* variable to a *PGlist* variable composed of *PGint* elements. The data values actually flow between the *PGint* elements but this abstraction is useful for certain applications. By default, the Playground C++ class library will create an element in the aggregate for the actual connection if applicable. Additionally, send-on-connect functionality (which can be specified as part of a connection request) sends a value across once the connection is made.

If an element-to-aggregate connection is made with the send-on-connect property the element created in the aggregate should not actually be created until a value is received. However, connection information will be needed by the element when it is created. This information comes from the connection negotiation process that has already completed. This information needs to be stored somewhere between the time the negotiation process has completed and the time the element is created.

An example helps to illustrate the pattern. An element-to-aggregate connection is being made from a *PGint* variable in one distributed component to a *PGlist* of *PGints* variable in another distributed component with the send-on-connect property specified. During connection negotiation the *Triggered Placeholder* pattern proscribes the creation of a *PGplaceholder* to store the needed connection negotiation information and to create the new *PGint* element when the first data value is sent over.

The *PGplaceholder* triggers on the unmarshalling *assignFromDataStream* method (which indicates that an initial value has been received). It overrides this method to create the new *PGint* element, pass the needed connection negotiation information to the new element, register the new element with its associated aggregate, and send on the incoming data value to the newly created *PGint* element for processing. Finally, the *PGplaceholder* unregisters itself from intercepting the trigger method and then deletes itself.

The *Triggered Placeholder* pattern is similar to the *Proxy* pattern [2] and the *Ghost* pattern [3] but has a distinct use case. A triggered placeholder object is not intended as a permanent go-

between for another object. It is temporary and will go away once the triggered method has been called. This implies a lifetime typically much shorter than the actual object.

The *Triggered Placeholder* pattern differs from the *Proxy* pattern in that the placeholder and the actual object will not co-exist from the user's point of view. There is a brief time when the 2 objects physically co-exist but this is only when the placeholder is creating the actual object. This is transparent to the user.

The *Triggered Placeholder* pattern differs from the *Ghost* pattern in that the placeholder does not change itself into the actual object. As noted in the *Ghost* pattern, this is difficult to facilitate in a statically typed language such as C++. The placeholder object and the actual object are distinct objects. This is part of the pattern.

6.0 Applicability:

Use the Triggered Placeholder pattern when you want to:

- manage resources involved with object creation; *or*
- delay the side effects of object creation.

Do *not* use the Triggered Placeholder pattern if you:

- want a permanent go-between for the actual object. Use the *Proxy* pattern instead; *or*
- need to have object identity at the time the object is requested. The identity of the placeholder object will be returned instead. This will cause problems later when the placeholder is deleted in lieu of creating the actual object.

It will also cause problems if methods are invoked on the placeholder object which are not defined. This will occur if the placeholder does not define all the methods of the real subject; *or*

- want to manage resources with object creation but the place holder object uses just as many resources as the actual object. Using the pattern in this instance will only add complexity and will not reduce the resources needed. For instance, if memory is the only resource being managed and the place holder object takes as much memory as the actual object then there is nothing to be gained by using this pattern.

7.0 Structure

8.0 Participants

- Subject (PGvariable)
 - defines the interface for RealSubject and Placeholder. Specifically, it declares the method(s) the RealSubject will implement and on which the Placeholder will trigger.
- RealSubject (PGint)
 - defines the real object the Placeholder will create when it is triggered.
- Placeholder (PGplaceholder)

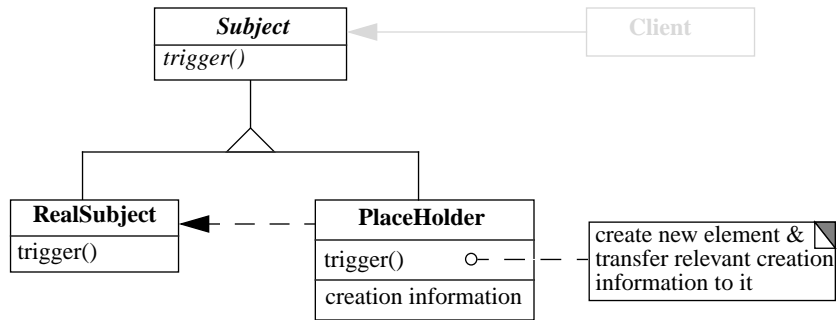


FIGURE 2. The Triggered Placeholder Pattern

- takes the place of the RealSubject until it is triggered by the trigger method. It then creates an appropriate RealSubject and deletes itself.

9.0 Collaborations

- A *Placeholder* creates a *RealSubject* when its trigger method is invoked. The *Placeholder* then deletes itself.

10.0 Consequences

10.1 Benefits

The Triggered Placeholder pattern offers the following benefits:

Fine grained control of object creation resources: The *Triggered Placeholder* pattern decouples object creation requests from actual object creation. This allows finer grained control over the creation of a desired object. This can be helpful in delaying resource allocation until absolutely needed. Resources used by an object can be used elsewhere after the request for an object has been made. The resources are only used when the object is created and this only happens when one of the object's methods (i.e., a trigger method) has been invoked.

Fine grained control of object creation side effects: The *Triggered Placeholder* pattern allows fine grained control over timing of side effects from object creation. The creation of certain objects implies certain ramifications to other objects. For instance, when an element in an array or list is created it has an impact upon the array or list. Using this pattern can bring fine grained control as to when creation occurs and accordingly when creation side effects occur.

10.2 Liabilities

The Triggered Placeholder pattern has the following liabilities:

Potential Interface Bloat: There needs to be a virtual method that the *Placeholder* can override as the trigger. This may not always be readily available. Therefore, a virtual method may need to be introduced to the *Subject* interface as the trigger method.

Additionally, a *RealSubject* may have several methods that could be triggers. It may be the case that the exact method that will be triggered is not known. It may be one of several possible methods depending on the execution of the application. In this case, all possible methods will need to be treated as trigger methods. If these methods are not virtual, they will need to be made virtual and moved into the *Subject* interface. This may increase the size of the interface for the *Subject* class.

Slower Execution: All potential trigger methods need to be virtual in the *Subject* class. This will add execution overhead if the trigger methods were not originally virtual.

Dependency Management: When the *Placeholder* creates the *RealSubject* any objects holding references to the *Placeholder* must be updated to refer to the *RealSubject*. It can be non-trivial to manage all the updates when objects hold references to the *Placeholder*. However, use of the *Publisher-Subscriber* pattern [4] can help with this management. The *Placeholder* could send out an *update* message to all its dependents and pass as a parameter the newly created *RealSubject*.

11.0 Implementation

This section describes how to implement the Triggered Placeholder pattern in C++. The implementation described below is influenced by the Playground distributed programming environment.

- **Determine the relevant classes/objects:** The developer needs to determine the objects whose creation resources or side effects need to be decoupled from the requests to create the objects. The class of these objects will become the *RealSubject* class in the *Triggered Placeholder* pattern.
- **Determine the trigger method(s):** The method on which the *Placeholder* will be triggered to create the *RealSubject* needs to be determined. This trigger method may only exist in the *RealSubject* interface. This method needs to be promoted to the *Subject* interface. All trigger methods must be declared in the *Subject* interface as virtual.

There may be several potential trigger methods. The developer needs to determine which methods might be the first invoked after the request for the *RealSubject*'s creation. All of these methods will need to be trigger methods. Otherwise, a method will be called on the *Placeholder* that is undefined. For this reason, it is best to make all the trigger methods declared in the *Subject* interface undefined (i.e., pure virtual) so that these calls can be caught at compile time.

- **Determine relevant creation information:** Some information may be needed to create the *RealSubject* properly. The developer needs to determine what information the *Placeholder* needs to have to create the *RealSubject* appropriately. The developer also needs to determine when this information will be available to store in the *Placeholder*. It may be that several different pieces of information will need to be stored at different times to have all the applicable information at the time of the *RealSubject* creation.

Use of the *Abstract Factory* pattern [2] may be helpful to reduce the amount of information the *Placeholder* needs to create the *RealSubject*. Some of the relevant information can be managed by the *Abstract Factory*.

- **Determine RealSubject memory management:** If the *Placeholder* creates the *RealSubject* from memory off the heap then this memory will need to be deleted when the *RealSubject* is no longer needed. In this case, some object will need to hold a pointer to the allocated *RealSubject* to avoid a memory leak. The *Placeholder* will be deleted by this time and is therefore not an appropriate candidate to manage the *RealSubject*'s memory.

Alternately, other memory management strategies may be applicable. The *Placeholder* may call some other object (such as a factory) to create the *RealSubject*. This other object could have the memory management responsibility and the *Placeholder* need do nothing special to avoid a memory leak.

Variations:

Cascading Place Holders: The developer may want to build up portions of a requested object incrementally to facilitate a finer-grained control of resources. A *Placeholder*'s trigger method may be invoked but the resources needed to service that particular request may only be a subset of the total resources included for the *RealSubject*. The *Placeholder* could create an intermediate object able to service the particular request.

The newly created intermediate object could itself act as a *Placeholder* and have trigger methods of its own. When its trigger method is invoked it could in turn create another "fuller" or "more nearly complete" *Placeholder* that would be able to handle both the original and latest trigger methods. This building up of the *RealSubject* could go on indefinitely with only those resources being allocated that are essential to handling the methods that have currently been invoked.

12.0 Sample Code

When a *Placeholder*'s trigger method is invoked, it creates the *RealSubject* and forwards on the method invocation to the *RealSubject*. The *Placeholder* implementation of *trigger* might look something like this:

```
void
Placeholder::trigger()
{
    // Create the RealSubject
    RealSubject* realSubject = new RealSubject();

    // Perform any registration or initialization that may be necessary for
    // the RealSubject and any other interested objects. This includes managing
    // the memory of the newly created RealSubject.
    ...

    // Delete the Placeholder since it is no longer needed
    delete this;

    // Forward the trigger method call to the RealSubject
    realSubject->trigger();
}
```

In the code example below, an abstract factory is used to create the *RealSubject*. This can reduce the size of the *Placeholder* since it will not need to keep as much creation information.

If the *RealSubject*'s trigger method returns a value the *Placeholder*'s trigger method should return the *RealSubject*'s trigger value.

```
int
Placeholder::trigger()
{
    // Create the RealSubject
    Subject* subject = factory->createSubject();

    // Perform any registration or initialization that may be necessary for
    // the RealSubject and any other interested objects. We will not need to worry
    // about memory management here if the factory will manage it for us.
    ...

    // Delete the Placeholder since it is no longer needed
    delete this;

    // Forward the trigger method call to the RealSubject
    return (subject->trigger());
}
```

The use of the factory to create the *RealSubject* in the code above also increases decoupling. The *Placeholder* no longer needs to know about the *RealSubject* type. It only needs to know about the *Subject* base type. The trigger method will be serviced by the *RealSubject* polymorphically.

13.0 Known Uses

The *Triggered Placeholder* Pattern is used in the Playground C++ class library. It is used to delay object creation for certain objects until a value has been received for that object from another component.

Do OSes that support paging do something similar for page faults? Probably not close enough to fit the pattern. There probably aren't trigger methods used, for example.

14.0 Related Patterns

The following patterns relate to the Triggered Placeholder Pattern:

- A *Placeholder* may use the *Abstract Factory* pattern [2] to create the appropriate *RealSubject* concrete object. Then the *Placeholder* does not need to keep extra creation information about the *RealSubject* which decreases the size of the *Placeholder*.
- The *Publisher-Subscriber* pattern [4] may be used to manage dependencies objects have with the *Placeholder*. When the *Placeholder* creates the *RealSubject* it can update all its dependents with the *RealSubject* as a parameter.
- The *Triggered Placeholder* pattern is similar in structure to the *Proxy* pattern [2]. However, it differs from the *Proxy* pattern in that the *Placeholder* is a temporary proxy. Once a trigger method is invoked, the *RealSubject* is created and the *Placeholder* goes away.

References

- [1] Kenneth J. Goldman, Bala Swaminathan, T. Paul McCartney, Michael D. Anderson, Ram Sethuraman, "The Programmers' Playground: I/O Abstraction for User-Configurable Distributed

- Applications”. *IEEE Transactions on Software Engineering*, 21(9):735-746, September 1995.
- [2] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Reading, MA: Addison-Wesley, 1995.
 - [3] K. Wolf and C. Liu, “New Clients with Old Servers: A Pattern Language for Client/Server Frameworks,” in *Pattern Languages of Program Design* (J. Coplien and D. Schmidt, eds.), Reading, MA: Addison-Wesley, 1995
 - [4] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*. West Sussex, England: John Wiley & Sons, 1996.