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 to provide a finer grain control of resource management or to delay side effects of object creation until absolutely needed.

2.0 Also Known As

Stub

3.0 Classification

Object Creational

4.0 Motivation

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 in an aggregate but the element needs to be created with some initial value. This initial value may only be known after the request for the element has been made and therefore the creation of the element needs to be delayed until the initial value is available. The motivation for delayed creation in this instance is management of creation side effects.

5.0 Example

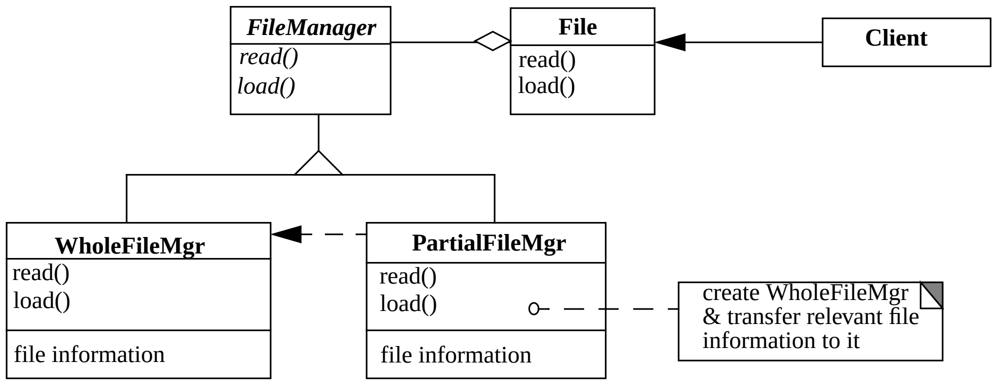

FIGURE 1. A File Example

The example application deals with manipulating and analyzing files. The motivation for using the Triggered Placeholder pattern in this instance is to manage file descriptors and memory efficiently both potentially scarce resources. Initially, small portions of a file are retrieved (e.g., looking at the first paragraph for a summary or abstract). Eventually, some analysis and comparison will be done on the entire file.

In this application, there are *File* objects that represent the files. The interface supported by the *File*s would have methods like *read()* and *load()*. *Read()* is used to bring in only a portion of the file. *Load*() is used to bring in the entire file for analysis such as word counting or spell checking. Initially, a *PartialFileMgr* is created. It hold an open file descriptor and small buffers to store the small portions of the file that are needed at first. When the *load* method is invoked, the *PartialFileMgr* creates a *WholeFileMgr* to store the entire contents of each file in memory for faster analysis. The *PartialFileMgr* triggers on the *load* method, creates the *WholeFileMgr*, and then deletes itself. The *WholeFileMgr* reads in the entire file and releases the file descriptor previously opened by the *PartialFileMgr* since it would not be needed any more. This frees up file descriptors which are typically limited by an operating system.

Since objects are being deleted and created some dependency management may be needed. A *PartialFileMgr* would need to know who was dependent upon it or using it in order to update them with the *WholeFileMgr* when the trigger method was called. In this specific example, the *File* may have a method to notify its *FileManager* reference. In general, the *Publisher-Subscriber* pattern [1] would be applicable. Any objects that use or are dependent upon the *FileManager* would register themselves with it. When the trigger method is called, the *PartialFileMgr* would call the *notify* method on all its dependents passing as a parameter the *WholeFileMgr*.

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 place holder will be returned instead. This will cause problems later when the place holder is deleted in lieu of creating the actual object.

  It will also cause problems if methods are invoked on the place holder which are not defined. This will occur if the place holder does not define all the methods of the real subject; *or*
- want to manage resources with object creation but the place holder uses just as many resources as the real subject. 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 takes as much memory as the real subject then there is nothing to be gained by using this pattern.

7.0 Structure

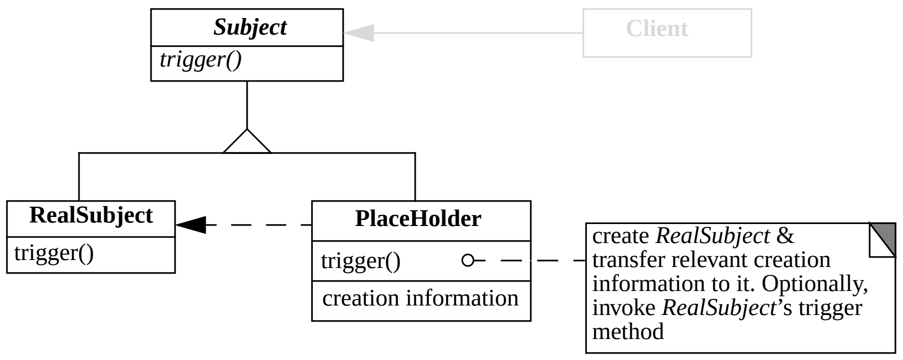
Subject
trigger()
Client
RealSubject
trigger()
PlaceHolder
trigger()
creation information
create RealSubject & transfer relevant creation information to it. Optionally, invoke RealSubject's trigger method

FIGURE 2. The Triggered Placeholder Pattern

8.0 Participants

- Subject (FileManager)
  - 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 (WholeFileMgr)
  - defines the real object the *PlaceHolder* will create when it is triggered.
- PlaceHolder (PartialFileMgr)

- 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 the 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 modifies 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 method that the *PlaceHolder* can override as the trigger. This may not always be readily available. Therefore, a method may need to be introduced to the *Subject* interface as the trigger method. *(In C++, the trigger method will need to be virtual.)*

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 and will need to appear in the *Subject* interface. This may increase the size of the interface for the *Subject* class.

An existing method can be made into a trigger method to avoid interface bloat. However, the object is made more complicated since users of the trigger method will potentially need to be aware of its extra functionality.

Slower Execution *(C++)* : All potential trigger methods need to be virtual in the *Subject* class. This will add execution overhead in C++ 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 [1] can help with this management. The *PlaceHolder* could send out a *notify* message to all its dependents and pass as a parameter the newly created *RealSubject.*

In general, it can be tricky to manipulate references to *PlaceHolder* objects since these references will become invalid after the trigger method has been invoked. It is usually helpful to have some class to encapsulate this behavior and shield a user from stale references such as the *File* class in the example above.

11.0 Implementation

This section describes how to implement the Triggered Placeholder pattern in C++. The implementation described below is influenced by Playground [2], a distributed programming environment, which uses the Triggered Placeholder pattern to delay creation of aggregate elements until an initial value has been provided.

- **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(s) 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. *(C++) 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 *(e.g., pure virtual in C++)* so that these calls can be caught.
- **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 [3] 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:** *(C++)* 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.

(Smalltalk, Java) In garbage-collected languages references to the *PlaceHolder* will need to be updated to references to the *RealSubject* in order to have the *PlaceHolder* garbage collected.

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 most recent 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 in C++:

```
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 decreases coupling. 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

13.1 Playground Distributed Programming Environment [2]

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.

In the Playground distributed programming environment, element-to-aggregate functionality

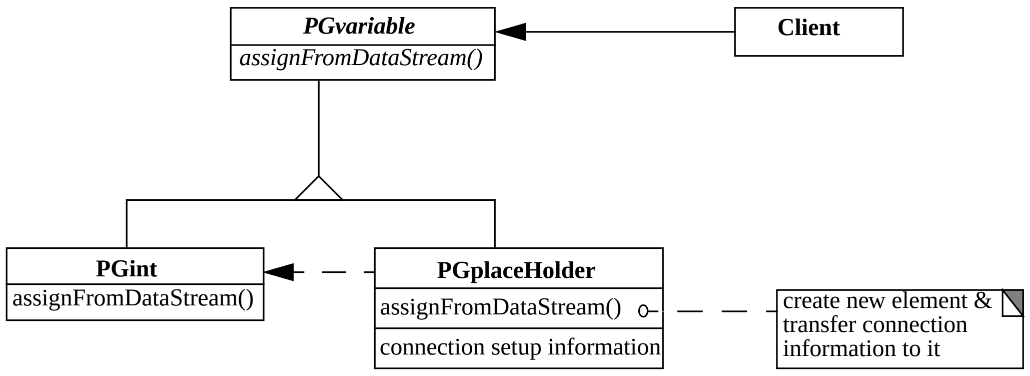

FIGURE 3. The Playground Example

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 *PGint*s 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 demarshalling *assignFromDataStream* method (which indicates that an initial value has been received). It overrides this method to create the new *PGint* element in its containing aggregate, 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.

13.2 Dynamically Linked Libraries

Operating systems that support dynamically linked libraries use the *Triggered Placeholder* pattern in a non-object-oriented way. The operating system will insert a stub in the executable in place of the library routine requested. When the library routine is actually invoked the stub replaces itself with the actual code.

14.0 Related Patterns

The following patterns relate to the Triggered PlaceHolder Pattern:

- A *PlaceHolder* may use the *Abstract Factory* pattern [3] 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*. Also, the client may use an *Abstract Factory* initially to create the appropriate *Subject* subclass (i.e., *PlaceHolder*). The client then only holds a reference to a *Subject* which can refer to a *PlaceHolder* or a *RealSubject*.
- The *Publisher-Subscriber* pattern [1] 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. This can be done by adding a parameter to the *Subscriber*s' update method which is a pointer to a *Subject*. The *Subscriber* can check this pointer against its previous pointer to see if its *Publisher* has changed.

- The *Triggered Placeholder* pattern is similar to the *Proxy* pattern [3] and the *Ghost* pattern [4] but has a distinct use case. A *PlaceHolder* 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 *RealSubject*.

  The *Triggered Placeholder* pattern differs from the *Proxy* pattern in that the *PlaceHolder* and the *RealSubject* 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 *RealSubject*. 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 *RealSubject*. As noted in the *Ghost* pattern, this is difficult to facilitate in a statically typed language such as C++. The *PlaceHolder* and the *RealSubject* are distinct from each other.

References

[1] 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.

[2] 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.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software.* Reading, MA: Addison-Wesley, 1995.

[4] 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