

Template Method [GoF]

Intent

Provide a skeleton of an algorithm in a method, deferring some steps to sub-classes.

Motivation

Suppose you have to write several `Buffer` classes all supporting the methods

- `public void put(Object o) throws InterruptedException`
- `public Object get() throws InterruptedException`

One or more producer threads put objects into an instance of `Buffer` provided that it is not full; one or more consumer threads get objects from the same instance of `Buffer` provided that it is not empty.

Writing to a full `Buffer` instance blocks the producer; reading from an empty `Buffer` instance blocks the consumer.

The sketch of a variant of a `Buffer` class might look like:

```
// Buffer variant.
public class BoundedBuffer {
    private Object[] buf; int putPtr = 0, getPtr = 0;
    private int usedSlots = 0;

    public BoundedBuffer (int capacity) throws
        IllegalArgumentException
    {
        if (capacity <= 0) throw new IllegalArgumentException();
        buf = new Object[capacity];
    }

    public synchronized void put(Object o) throws InterruptedException.{
        /*1*/ while (_usedSlots == _buf.length) {
            wait();
        }
        /*2*/ buf[putPtr] = obj; putPtr = (putPtr + 1)%buf.length;
        /*3*/ if (usedSlots++ == 0) {
            notifyAll();
        }
    }
    ... }
```

The `get` method of above class `BoundedBuffer` is analogous:

```
// In class BoundedBuffer:
public synchronized Object get() throws InterruptedException. {
    /*1*/ while (usedSlots == 0) { wait(); }
    /*2*/ Object x = buf[outPtr]; buf[outPtr] = null;
        outPtr = (outPtr + 1) % buf.length;
    /*3*/ if (usedSlots-- == capacity) {
        notifyAll();
    }
    return x;
}
```

Analyzing the code of the methods `put` and `get` shows that their actions depend on the state of the buffer. The buffer is in one of the following states:

<i>State</i>	<i>Condition</i>	<i>put</i>	<i>take</i>
full	<code>usedSlots == capacity</code>	no	yes
partial	<code>0 < usedSlots < capacity</code>	yes	yes
empty	<code>usedSlots == 0</code>	yes	no

So, for method `put` for example, the following *state-dependent actions* are carried out:

```
// pseudo code:
public synchronized void put(Object o) throws InterruptedException. {
    /*1*/ wait while full;
    /*2*/ insert object into buf;
    /*3*/ notifyAll if buf was empty;
}
```

This kind of sequence of applies for many kind of implementations of buffer variants. Consider for example a one place buffer:

```
// pseudo code for a one place buffer:
public synchronized void put(Object o) throws InterruptedException. {
    /*1*/ wait while full;
    /*2*/ insert object into one place buf;
    /*3*/ notifyAll if buf was empty
}
```

The comparison of the pseudo-code section above yields:

- */*1*/* `part wait while full` seems to be equal
- */*2*/* insertion part is subject to the underlying data structure
- */*3*/* notification part depends on the state structure of the buffer variant

The similarities for method `put` can be described as follows:

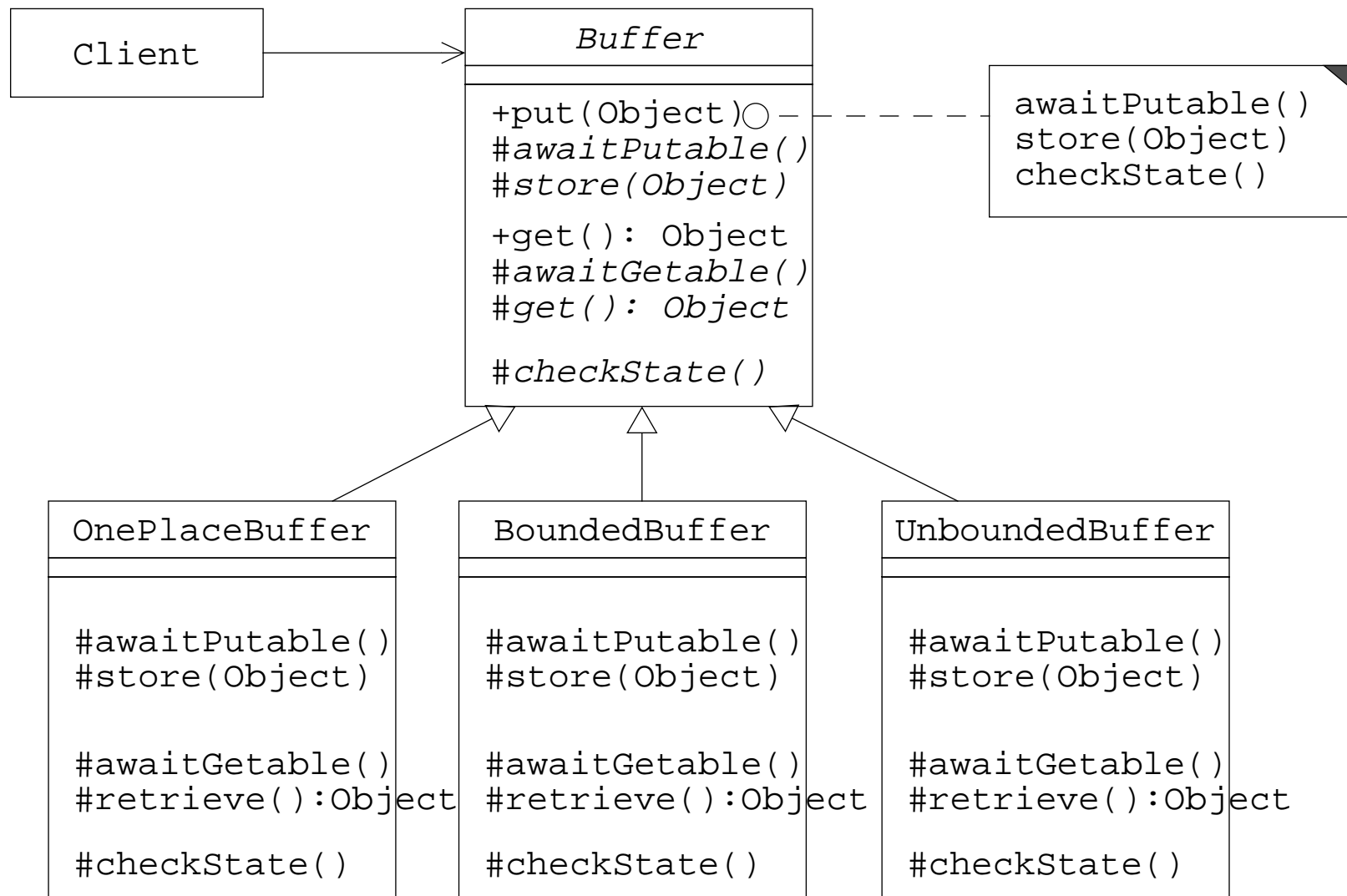
```
// pseudo code:
public synchronized void put(Object o) throws InterruptedException.{
    common_wait_while_full();
    specialized_insertion(o);
    specialized_notification();
}
protected abstract void specialized_insertion(Object o);
protected abstract void specialized_notification();
```

Similar things can be said for the `get` method. It seems that each of the methods can be divided into a sequence of more *primitive* methods. One of them seems to be implementable for all kind of buffer (`common_wait_while_full`), while others need to be specialized.

The following class diagram can now be derived from the above discussion. The diagram shows that public method `put` is composed of more primitive methods.

Method `awaitPutable` is private and fully implemented, while methods `store` and `checkState` are protected and abstract.

Methods `put` and `get` are called *template methods* since they define an algorithm in terms of more primitive, abstract and/or concrete methods. Subclasses then override the abstract primitive methods.

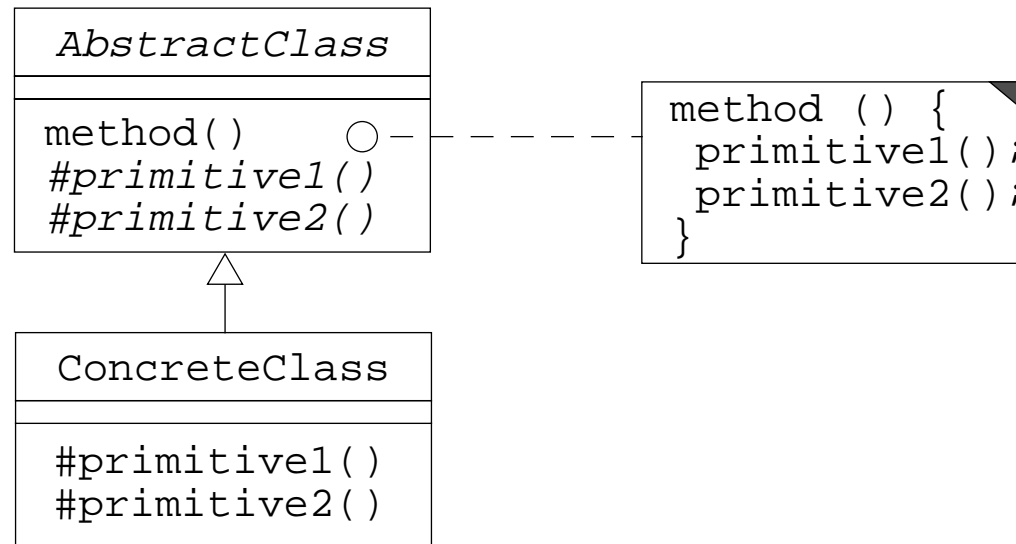


Applicability

The Template Method pattern should be used:

- to implement invariant parts of an algorithm once, and to leave it to subclasses to implement parts of the algorithm that can vary
- when common behavior among subclasses should be factored and localized in a common class to avoid code duplication
- to control subclass extensions

Structure



Participants

- *AbstractClass*:
 - defines abstract/concrete primitive methods that subclasses override
 - implements a template method defining the skeleton of an algorithm
- *Concrete Class*:
 - implements the primitive methods

Collaborations

- `ConcreteClass` relies on `AbstractClass` to implement primitive methods
- Clients preferably bind to the `AbstractClass` type which they may receive via factory method from, say, a factory.

Consequences

- Template methods are a fundamental technique for code reuse, for example, in class libraries.
- Template methods lead to an inverted control structure that's sometimes referred to as the "Hollywood principle," that is, "Don't call us, we call you".
- Template methods are equally well applicable in frameworks: They invoke so-called *hook methods*, which provide default behavior which subclasses can extend if necessary.
- It's important for template methods to specify which methods are hooks (*may* be overridden) and which are abstract operations (*must* be overridden).

Implementation

- **Access control:** Primitive methods should be declared as protected. This ensures that they are only called by the template method.
- **Minimizing primitive methods:** Minimize the number of primitive methods that a subclass must override to implement the algorithm.
- **Naming convention:** You can identify the methods that should be overridden by adding a prefix to their names.

Sample Code

The following code example shows a class hierarchy of `Buffer` classes. We start with the abstract class `Buffer`:

```
public abstract class Buffer {  
    protected static final int EMPTY = 0, FULL = 1;  
    protected int state = EMPTY;  
  
    public void init(int size)// used for buffer size initial.  
        throws IllegalArgumentException {}  
  
    public final synchronized void put(Object x) throws Int.Ex.  
    {  
        awaitPutable();  
        store(x);  
        checkState();  
    }  
  
    public final synchronized Object get() throws Interr.Ex. {  
        awaitGetable();  
        Object x = retrieve();  
        checkState();  
        return x;  
    }  
    protected abstract void store(Object x) throws Interr.Ex.;  
    protected abstract Object retrieve() throws Interr.Ex.;
```

```
// Buffer class continued...
private final synchronized void awaitPutable()
    throws InterruptedException
{
    while (state == FULL) {
        try {
            wait();
        } catch (InterruptedException ex) { throw ex; }
    }
}
private final synchronized void awaitGetable()
    throws InterruptedException
{
    while (state == EMPTY) {
        try {
            wait();
        } catch (InterruptedException ex) { throw ex; }
    }
}
protected abstract void checkState();
}
```

Concrete buffer classes subclass class `Buffer`. For example, class `BoundedBuffer`:

```
public class BoundedBuffer extends Buffer
{
    private static final int MIDDLE = 2;
    protected Object[] buffer;
    protected int putPtr = 0;
    protected int getPtr = 0;
    protected int usedSlots = 0;

    public void init(int capacity)
        throws IllegalArgumentException {
        if (capacity <= 0)
            throw new IllegalArgumentException();
        buffer = new Object[capacity];
    }

    protected synchronized void store(Object x) {
        buffer[putPtr] = x;
        putPtr = (putPtr + 1) % buffer.length;
        usedSlots++;
    }
}
```

```
protected synchronized Object retrieve() {
    Object x = buffer[getPtr];
    getPtr = (getPtr + 1) % buffer.length;
    usedSlots--;
    return x;
}

protected synchronized void checkState() {
    int oldState = state;

    if      (usedSlots == 0)           state = EMPTY;
    else if (usedSlots == buffer.length) state = FULL;
    else                                     state = MIDDLE;

    // Notify when leaving BOTTOM or TOP states.
    if (state != oldState &&
        (oldState == EMPTY || oldState == FULL))
        notifyAll();
}
}
```


Related Patterns

Factory Methods are often called by template methods.

Strategy: Template methods use inheritance to vary part of an algorithm. Strategies use delegation to vary the entire algorithm.