Object-Oriented Programming

Design Pattern Template Method

Michael L. Collard, Ph.D.

Department of Computer Science, The University of Akron

Template Method

A class pattern that defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior

- Behavioral Pattern
- Class pattern uses generalization/inheritance

Template Method

A class pattern that defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior

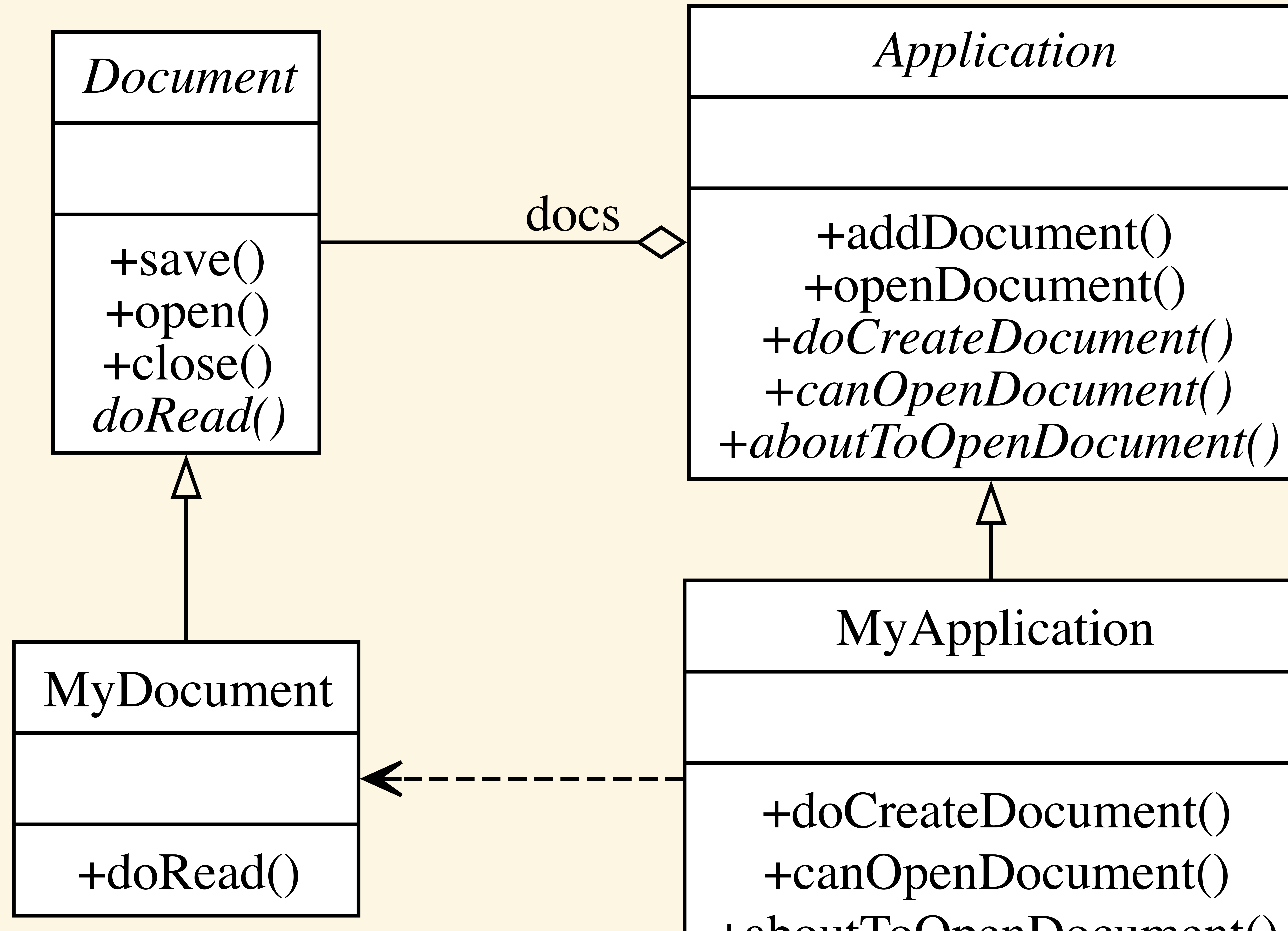
- A fundamental technique for code reuse
- It is particularly important in class libraries to factor out common behavior
- Template methods lead to Inversion of Control (IoC), which is sometimes referred to as "the Hollywood principle," that is, "Don't call us, we'll call you" [Swe85]
- The Template Design Pattern is **not** related to C++ templates and uses inheritance, not C++ templates

Template Method: Applicability

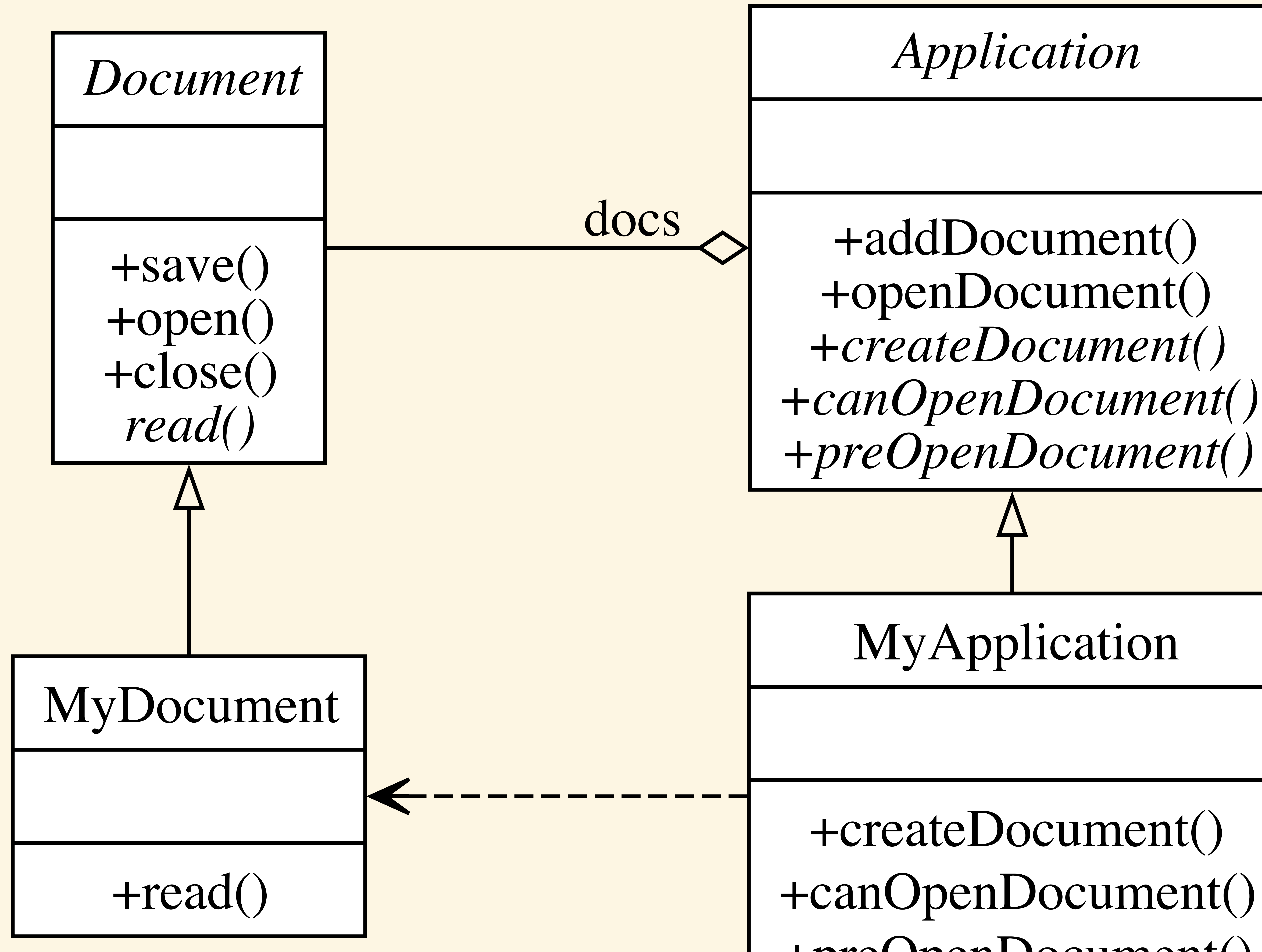
```
/* invariant */ void parse() {

/* invariant */     // determine encoding
/* invariant */     std::string encoding = getEncoding();

/* invariant */     // create buffer with the proper encoding
/*   variant */     auto buffer = createBuffer(encoding);

/* invariant */     // parse with buffer
/* invariant */     while (!isDone()) {
/* invariant */         // ..

/* invariant */         // allow subclasses to update progress
/*   variant */         updateProgress(bytesRead);
/* invariant */     }
/* invariant */ }
```

- Implement *invariant* parts of an algorithm and leave it up to subclasses to implement behavior that can vary
- Factor and localize common behavior in subclasses to avoid code duplication
- Control subclass extensions with "hook" operations

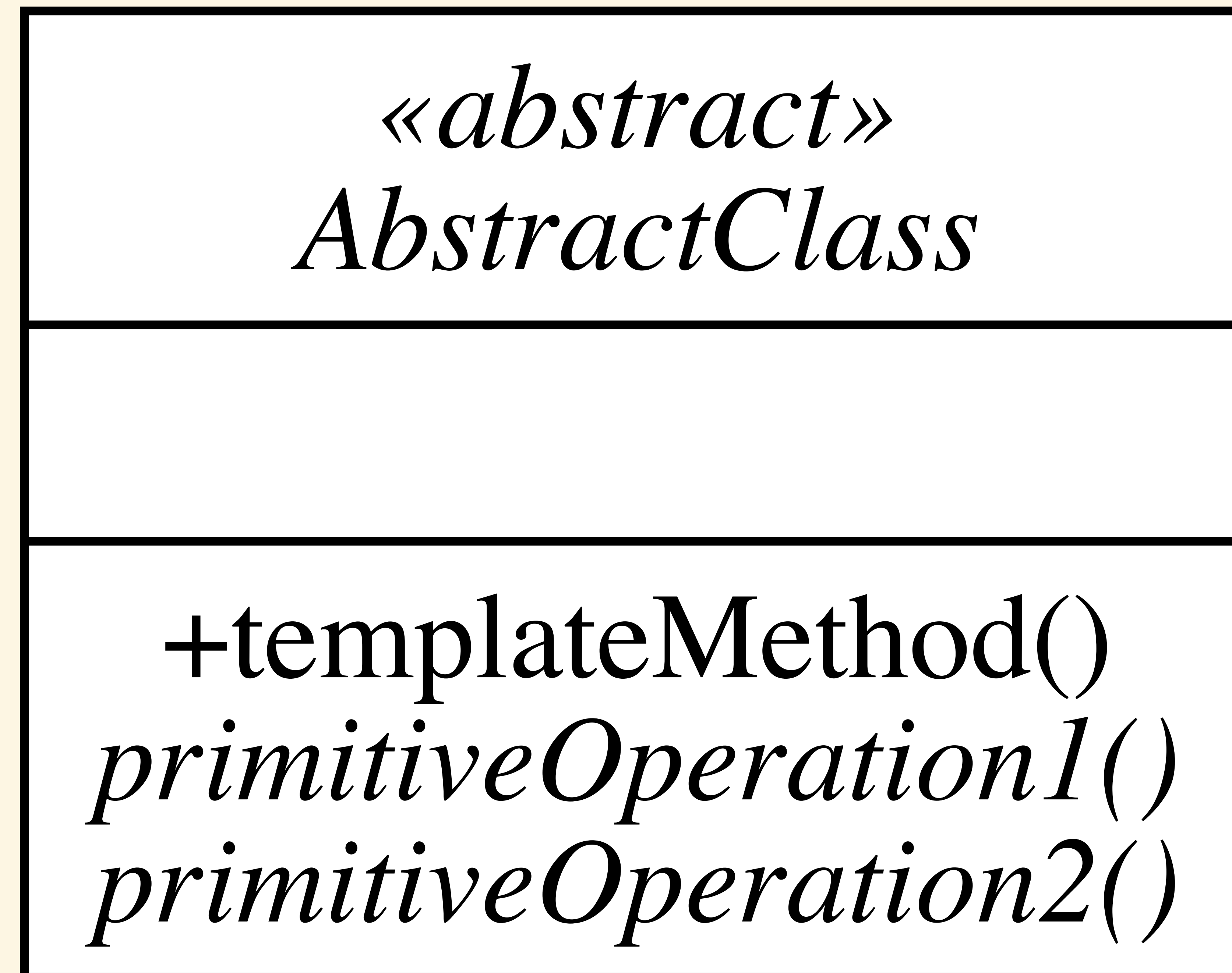
Template Method: Structure (GOF Example)

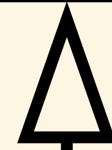


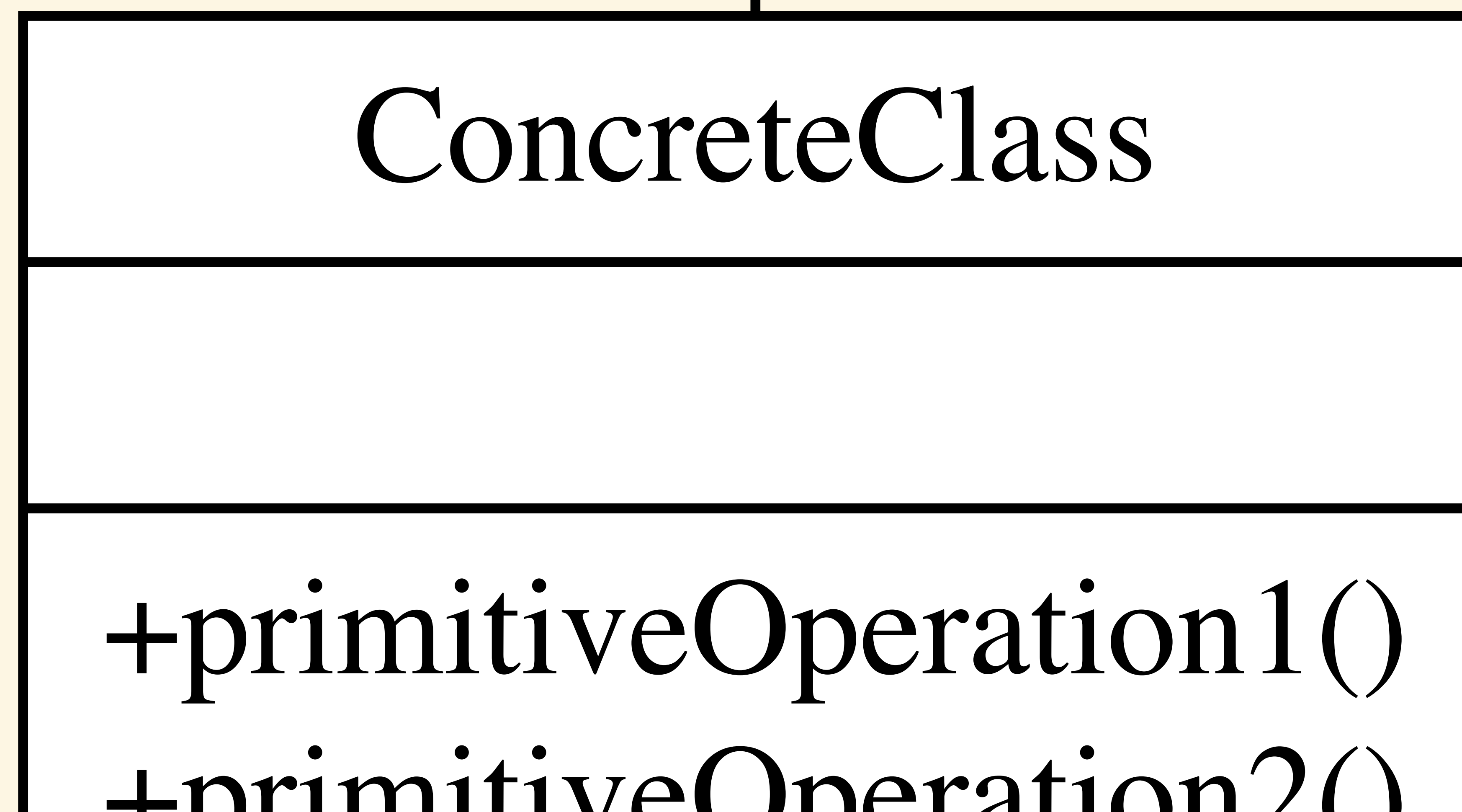
Template Method: Structure (GOF Example Update)



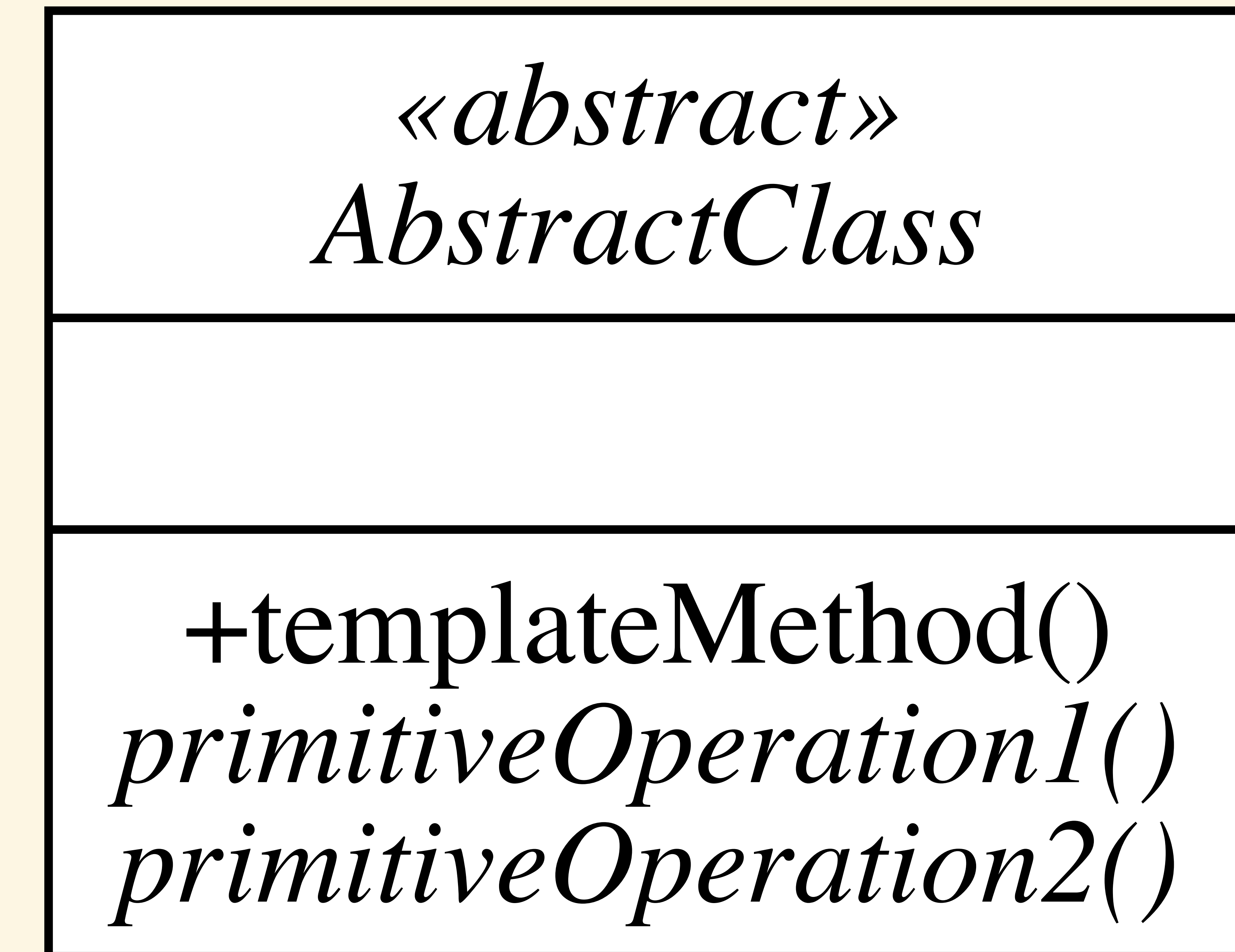
Template Method: Structure







Template Method: Structure

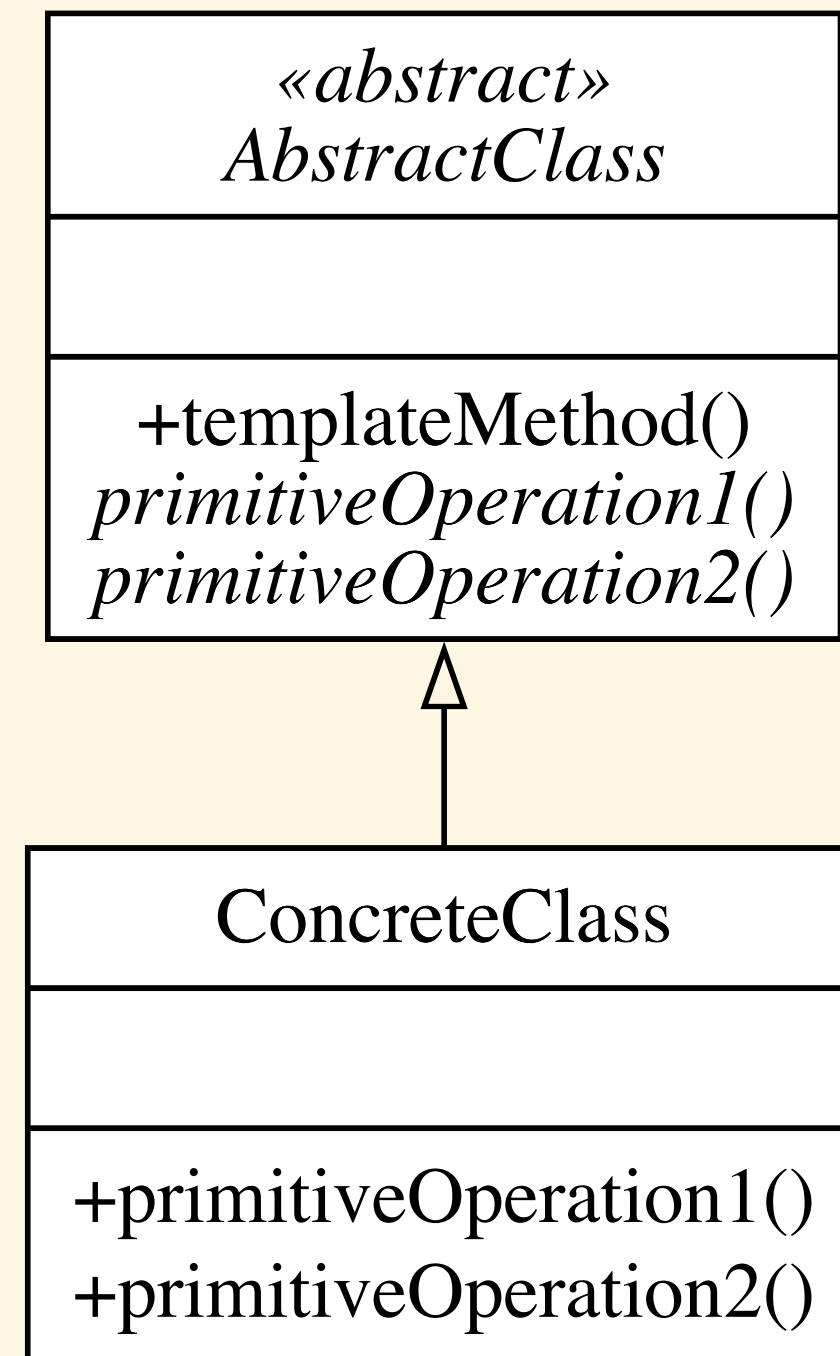



```
templateMethod() {
    //...
    primitiveOperation1();
    //...
    primitiveOperation2();
    //...
}
```

ConcreteClass

+primitiveOperation1()
+primitiveOperation2()

Template Method: Participants



- **AbstractClass** (e.g., `Application`)

  Defines *primitive operations* that concrete subclasses define to implement steps of an algorithm

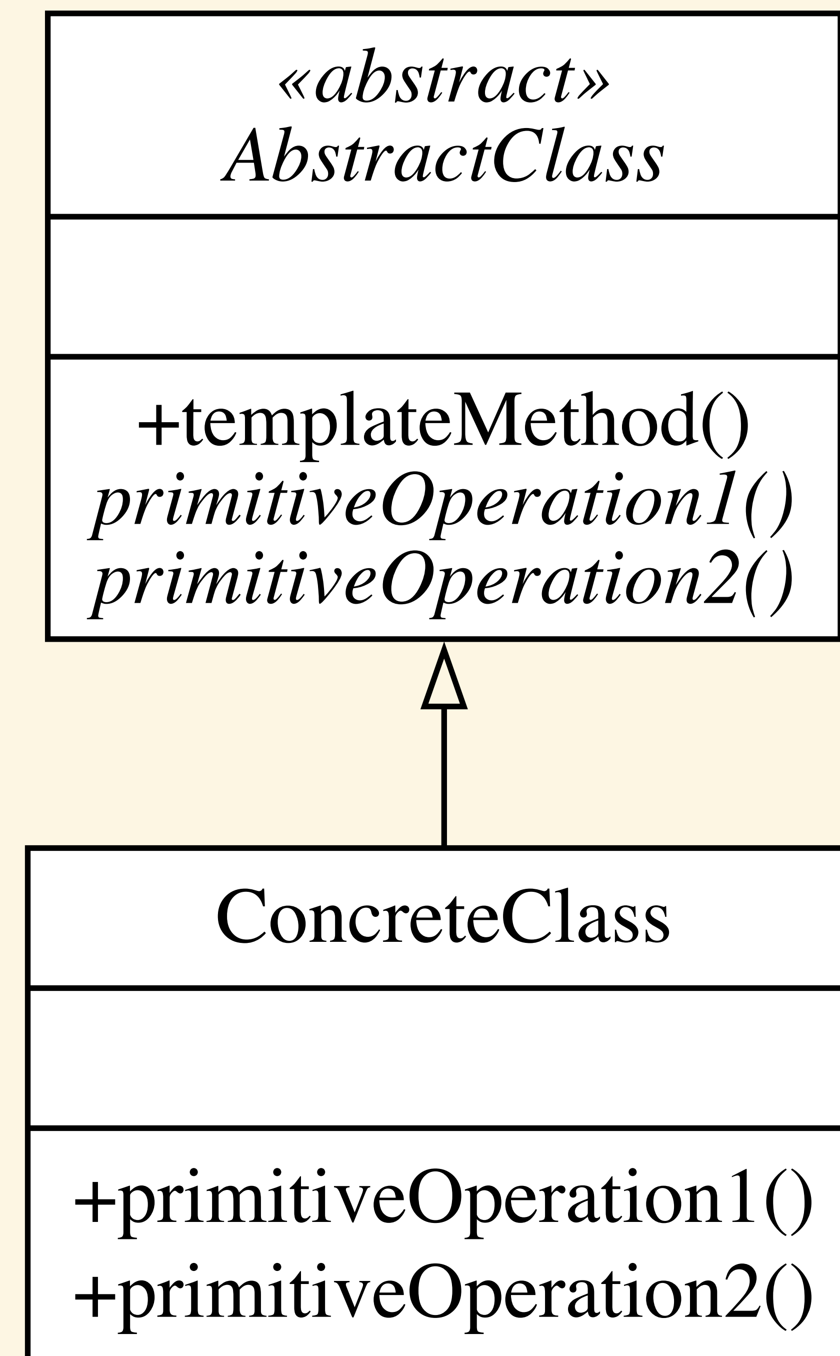
  Implements a *template method* defining the skeleton of an algorithm, which calls primitive operations (in `ConcreteClass`), operations in `AbstractClass`, or anything else

-

- **ConcreteClass** (e.g., `MyApplication`)

  Implements the *primitive operations* to carry out subclass-specific steps of the algorithm

Template Method: Collaborations



- ConcreteClass relies on AbstractClass to implement invariant steps of the algorithm

Template Method: Consequences - Kinds of Operations

- *primitive operations* (i.e., abstract operations) ***must* override**
- *hook operations* - default behavior that subclasses can extend if necessary. ***may* override** and often does nothing
- concrete operations (ConcreteClass or on client classes)
- concrete AbstractClass operations (i.e., operations that are generally useful to subclasses)
- factory methods (later)

Extension via Direct Overriding

```
void ConcreteClass::Operation() {
        AbstractClass::Operation();

        // Derived class extended behavior
}
```

- Useful when no common parent (class) behavior

Extension via Hook Operation

```
void AbstractClass::Operation() {
    // AbstractClass behavior

    hookOperation();
}

void AbstractClass::hookOperation() {}

void ConcreteClass::hookOperation() {
    // derived class extension
}
```

- Useful when extension may not be needed

- Useful when the extension is a *side-effect* of standard processing

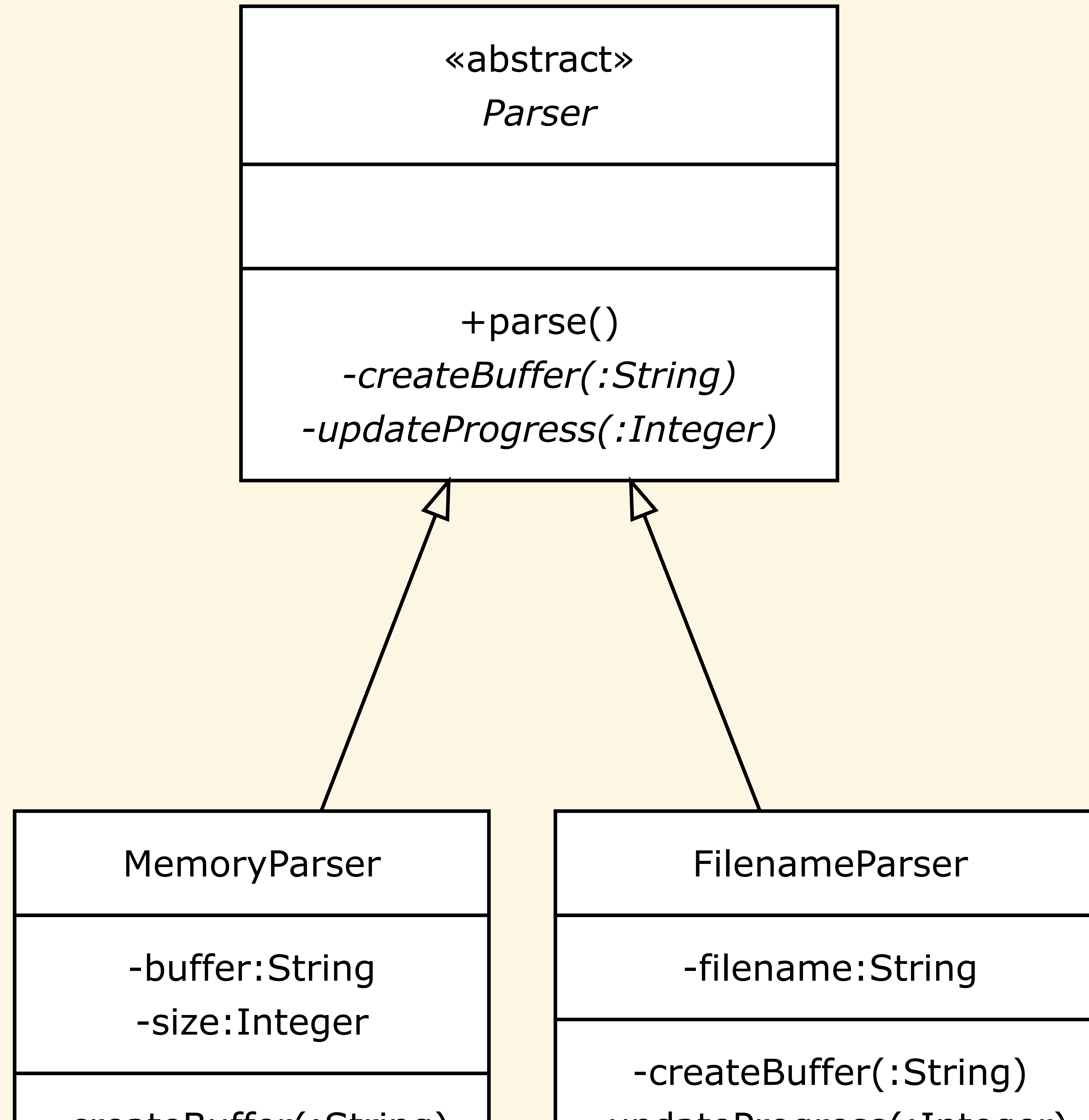
Parser.hpp

```cpp
class Parser {
public:
    // parses the given buffer using a calculated encoding
    void parse() {

        // determine encoding
        std::string encoding = getEncoding();

        // create buffer with the proper encoding
        auto buffer = createBuffer(encoding);

        // parse with buffer
        while (!isDone()) {
            // ..

            // allow subclasses to update progress
            updateProgress(bytesRead);
        }
    }

private:
    // creates a buffer of the proper encoding
    virtual std::string createBuffer(std::string_view encoding) = 0;

    // hook during parsing for progress updates
    virtual void updateProgress(int bytes) {}

    // parsing is done
    bool isDone() const;

    // get the current encoding to use
    std::string getEncoding() const;
};
```

FilenameParser.hpp

```
class FilenameParser : public Parser {
public:
    // constructor from filename
    FilenameParser(std::string_view filename);

private:
    // creates a encoded buffer from a filename
    virtual std::string createBuffer(std::string_view encoding) {
        // create buffer using filename
    }

    // hook during parsing to display number of bytes parsed
    virtual void updateProgress(int bytes) {
        // Display progress in the terminal
        totalBytes += bytes;
        std::clog << "Total: \r" << totalBytes;
    }

    std::string filename;
    int totalBytes = 0;
};
```

Parser UML



MemoryParser.hpp

```
class MemoryParser : public Parser {
public:
    // constructor with memory buffer
    MemoryParser(void* buffer, int size);

private:
    // creates a encoded buffer from memory
    virtual std::string createBuffer(std::string_view encoding) {
        // create buffer using buffer and size
    }

    void* buffer;
    int size;
};
```

Implementation

- The template method is the only method that calls the primitive operations; therefore, the primitive operations do not have to be public

- A template method can be a private, non-virtual member function

- Primitive operations that require overriding are declared pure virtual

- Minimize the number of primitive operations a subclass is required to implement

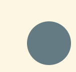
Known Uses

- Found in almost every abstract class

Related Patterns

- *Factory Methods*

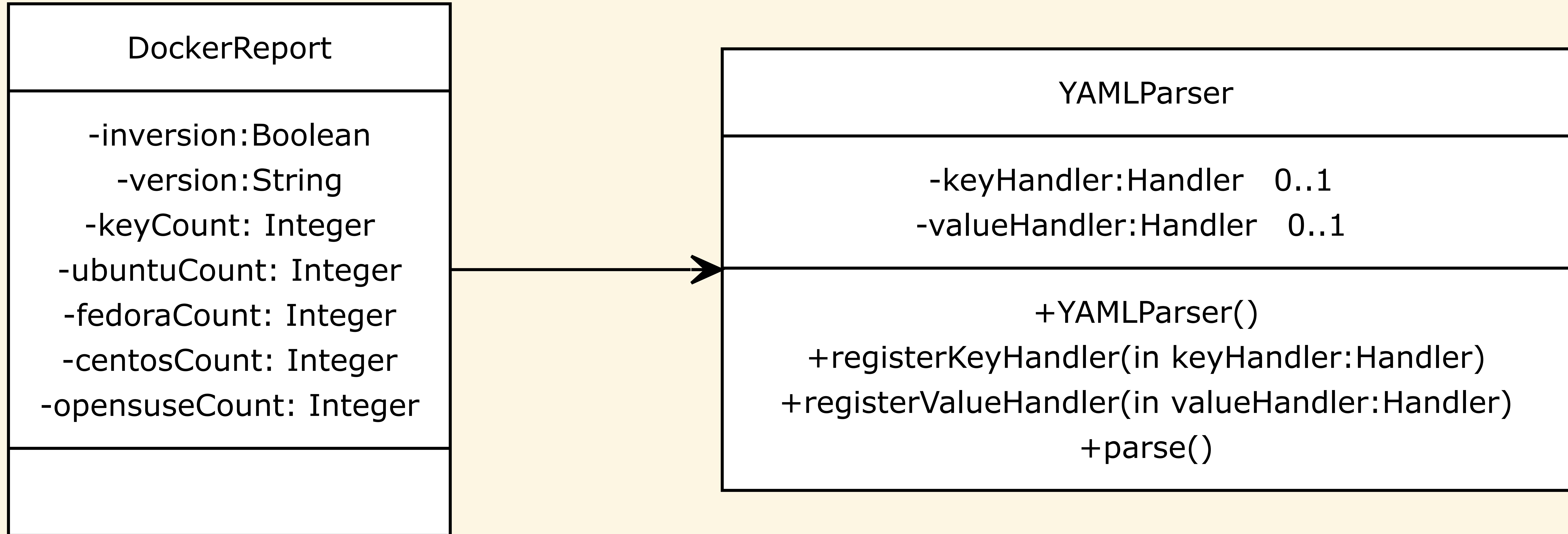
  Are often called by template methods

- 

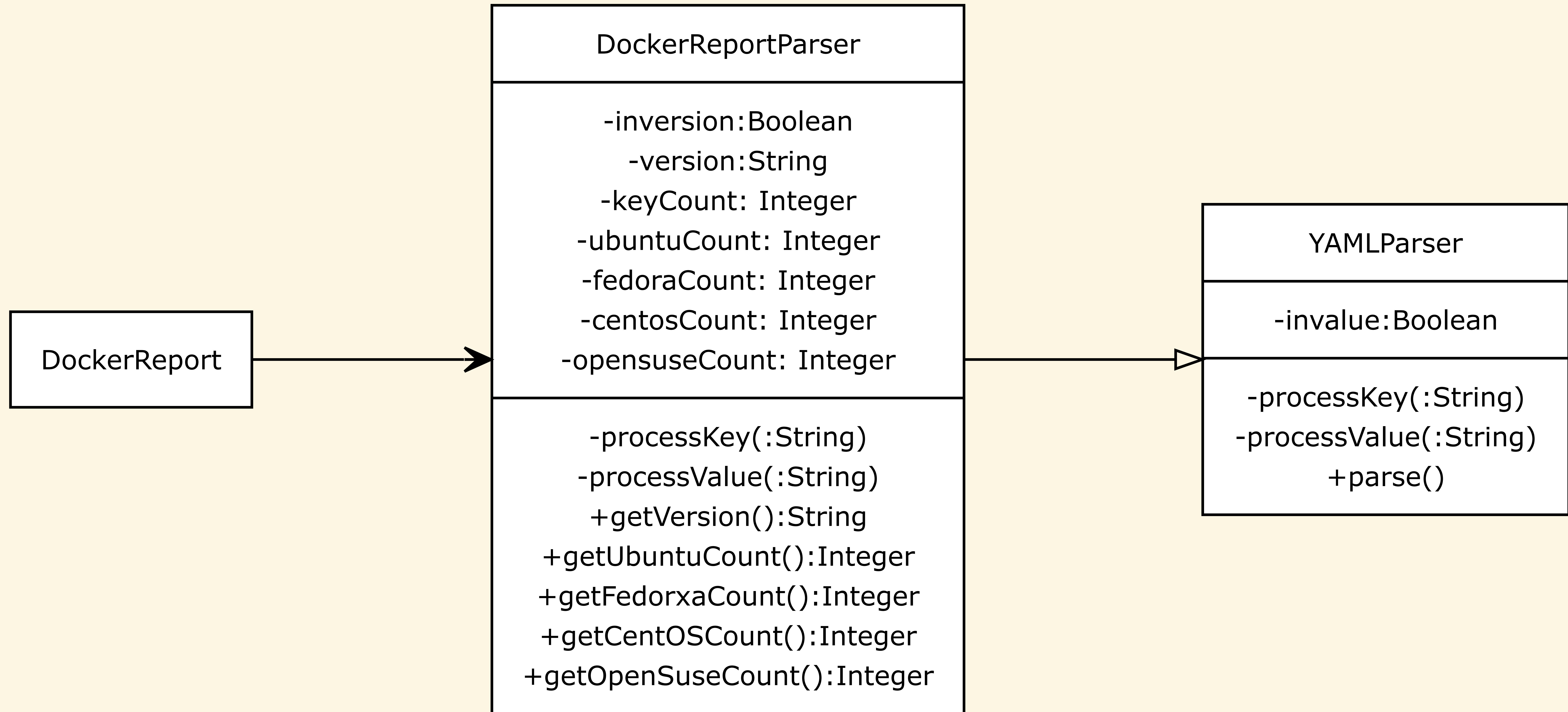
- *Strategy Pattern*

  Template Method pattern uses *inheritance* to vary parts of an algorithm. Strategy patterns use *delegation* to vary the entire algorithm.

DockerReport: Lambda Design



DockerReport: Template Method Design



DockerReport: Lambda & Template Method Comparison