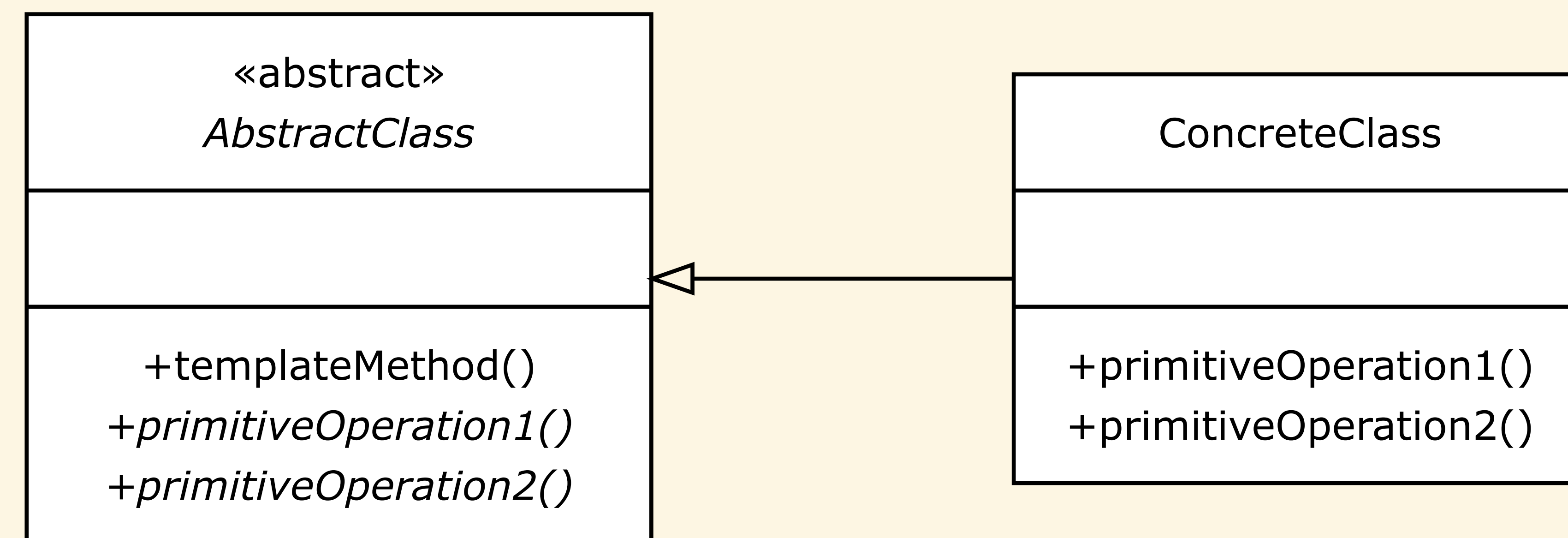
Object-Oriented Programming

Handlers

Michael L. Collard, Ph.D.

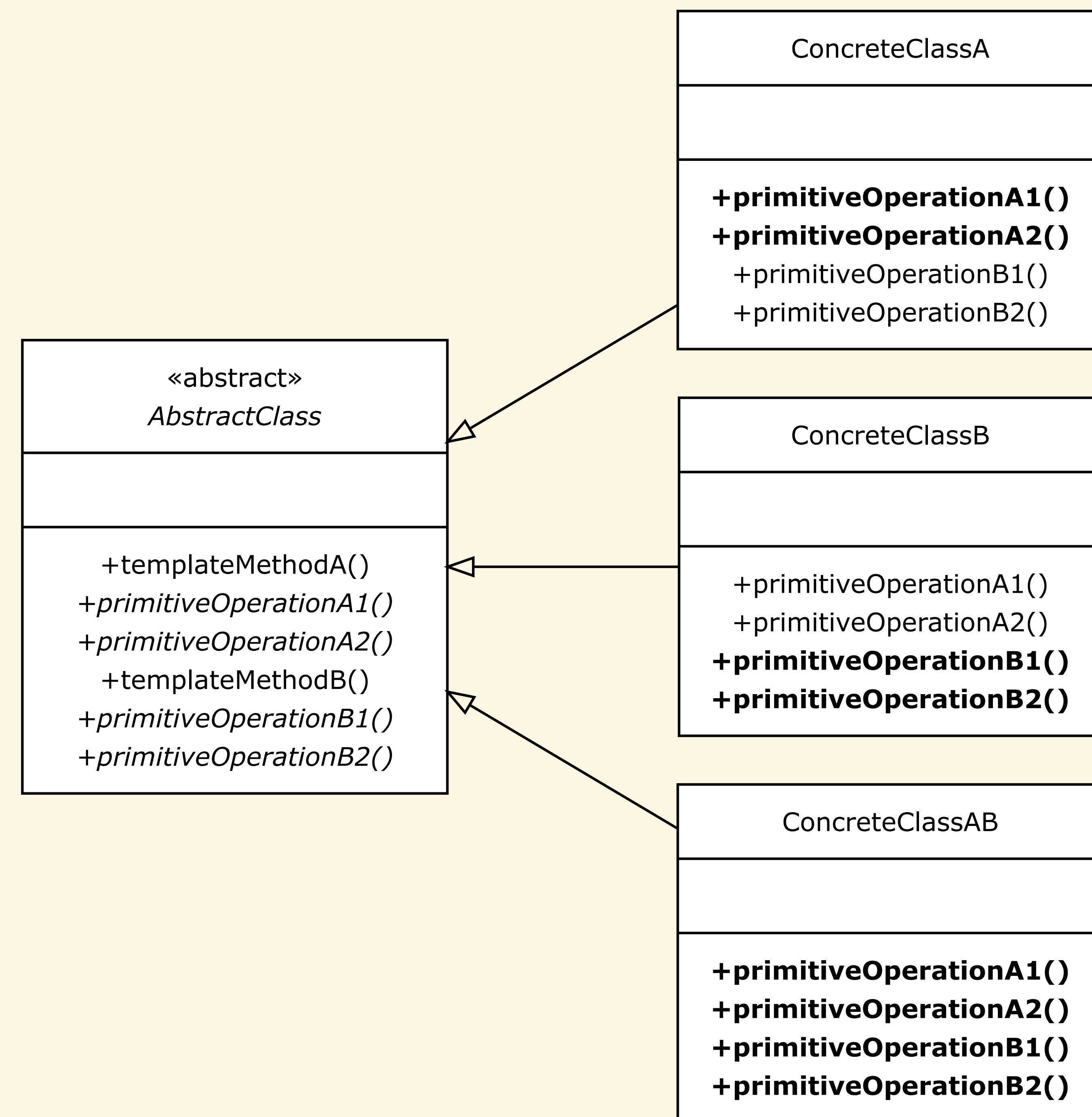
Department of Computer Science, The University of Akron

Single Template Method



- `ConcreteClass` has to subclass the entire `AbstractClass`
- The `ConcreteClass` has to include any `AbstractClass` dependencies
- For each abstract method, `ConcreteClass` must provide a definition

Multiple Template Methods



- What about if there is more than one *template method*?
- `ConcreteClassA` has to implement all `AbstractClass` abstract methods, i.e., `primitiveOperationA1()`, `primitiveOperationA2()`, `primitiveOperationB1()`, and `primitiveOperationB2()`, even if only interested in `templateMethodA()`
- Similarly, for `ConcreteClassB`
- The only class this is okay for is `ConcreteClassAB`

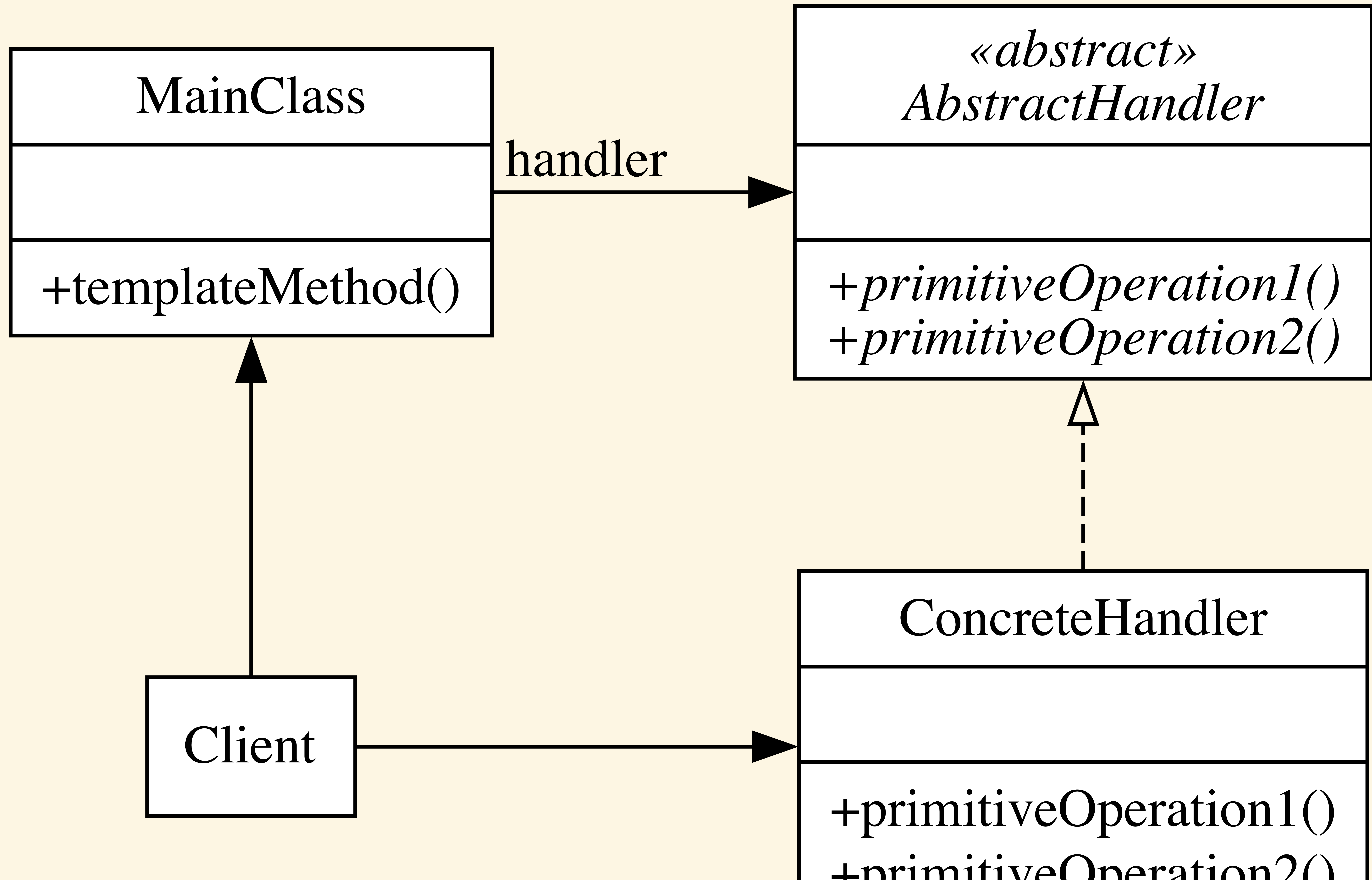
Goal

- Separate primitive operations for a particular purpose from the rest of the class interface allows for the decoupling of the rest of the class

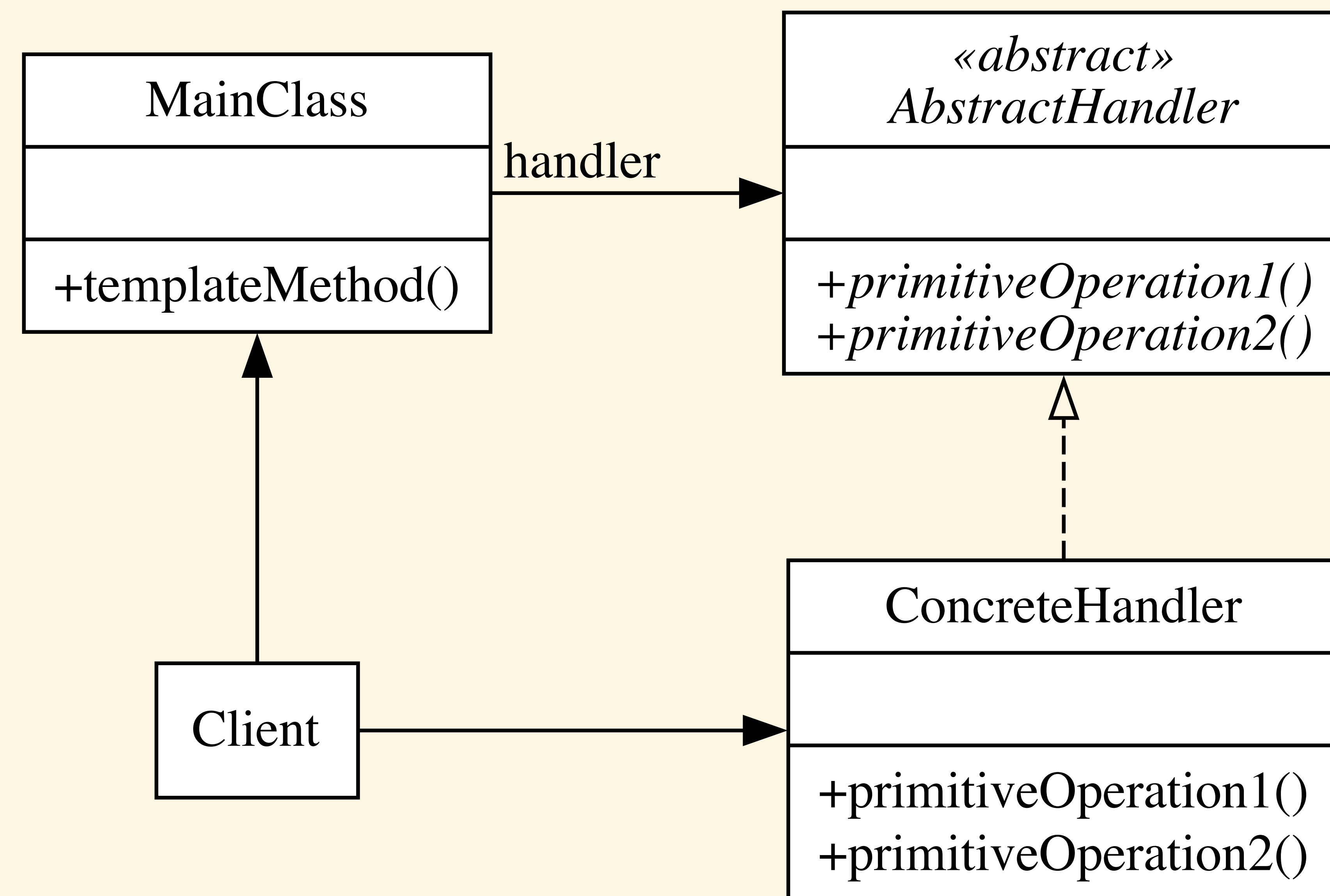
- Subclass the set of primitive operations instead of the entire class

- Reduces coupling and has multiple template methods

Solution



Solution



- `AbstractHandler` has only essential dependencies for the interface and does not have the dependencies of the `MainClass`

- `ConcreteHandler` is derived from `AbstractHandler`, so it does not share the dependencies of `MainClass`

Solution: Class Structure

```
class AbstractHandler {
public:
    virtual void primitiveOperation1() = 0;
    virtual void primitiveOperation2() = 0;
};

class MainClass {
public:
    // constructor with handler
    MainClass(AbstractHandler& handler)
        : handler(handler) {
    }

    void templateMethod();

private:
    AbstractHandler& handler;
};

class ConcreteHandler : public AbstractHandler {
public:
    void primitiveOperation1();
    void primitiveOperation2();
};
```

Solution: main()

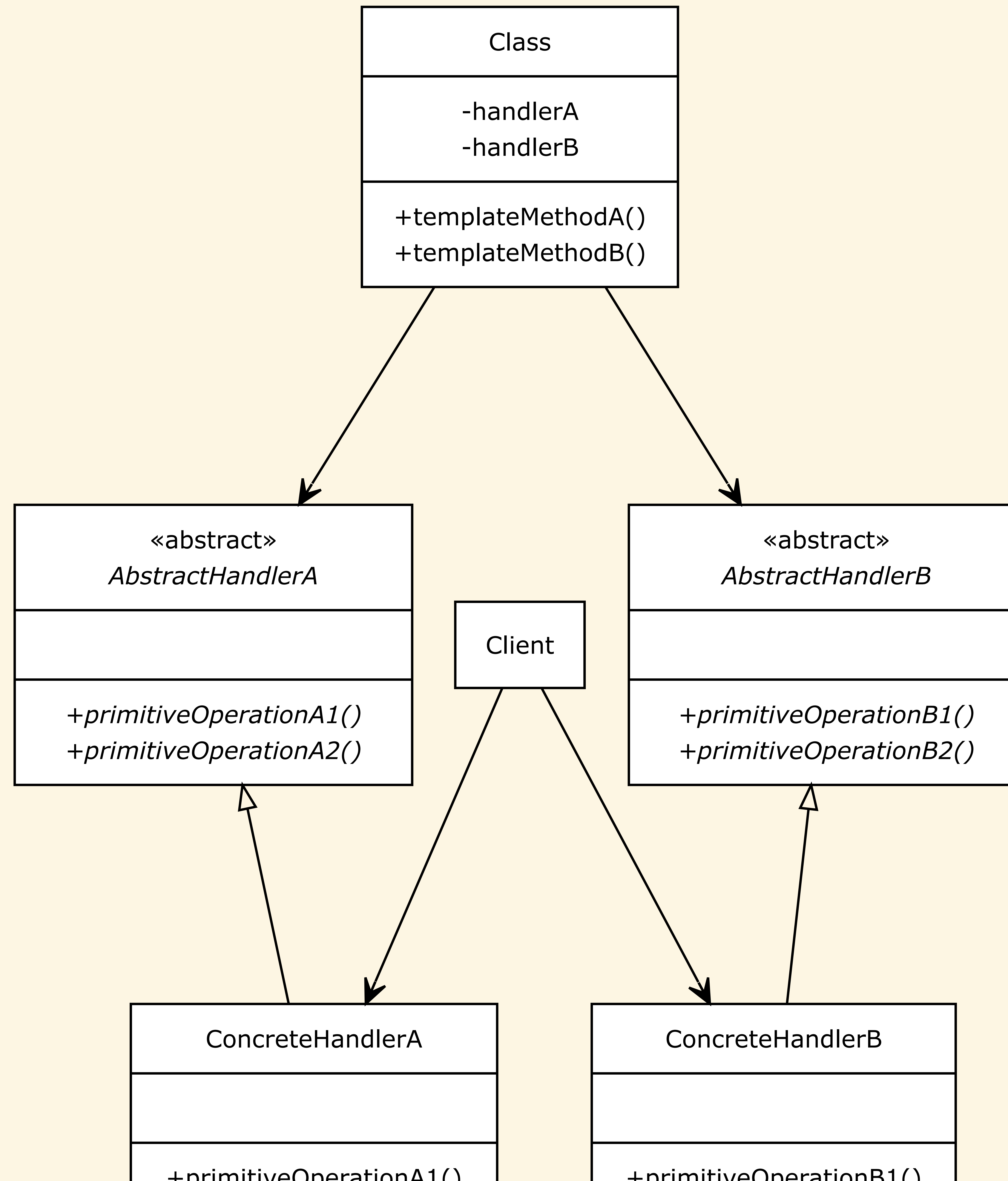
```
int main() {

    // setup our specific handler
    ConcreteHandler handler;

    // build the MainClass with our handler
    MainClass worker(handler);

    // run the template method
    worker.templateMethod();

    return 0;
}
```

Solution: Methods

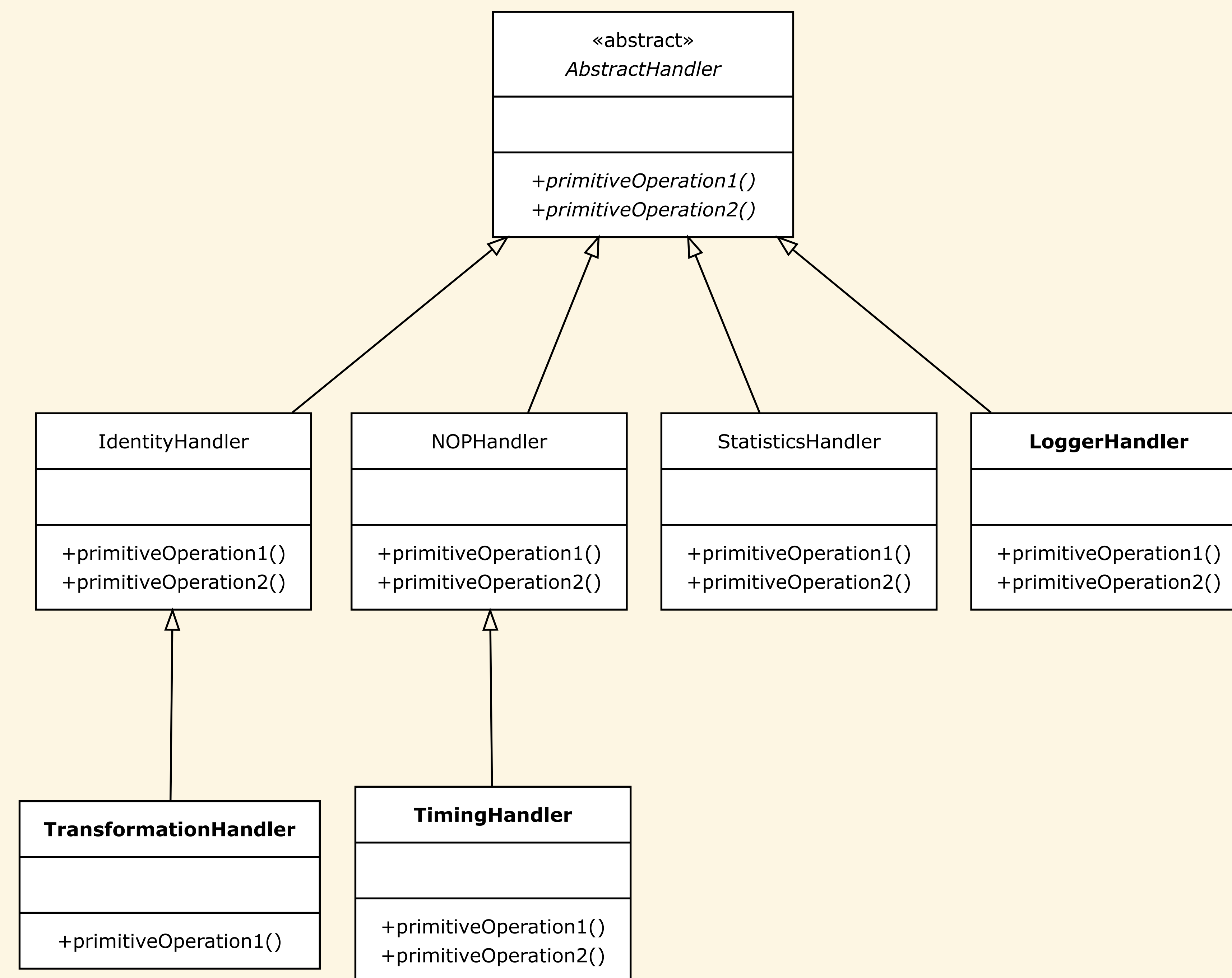
```
void MainClass::templateMethod() {
    std::clog << "MainClass::" << __FUNCTION__ << " BEGIN" << '\n';
    // ...
    handler.primitiveOperation1();
    // ...
    handler.primitiveOperation2();
    // ...
    std::clog << "MainClass::" << __FUNCTION__ << " END" << '\n';
}

void ConcreteHandler::primitiveOperation1() { std::clog << "ConcreteHandler::" << __FUNCTION__ << '\n'; }
void ConcreteHandler::primitiveOperation2() { std::clog << "ConcreteHandler::" << __FUNCTION__ << '\n'; }
```

```
MainClass::templateMethod BEGIN
ConcreteHandler::primitiveOperation1
ConcreteHandler::primitiveOperation2
MainClass::templateMethod END
```

Multiple Handlers



Standard Handlers

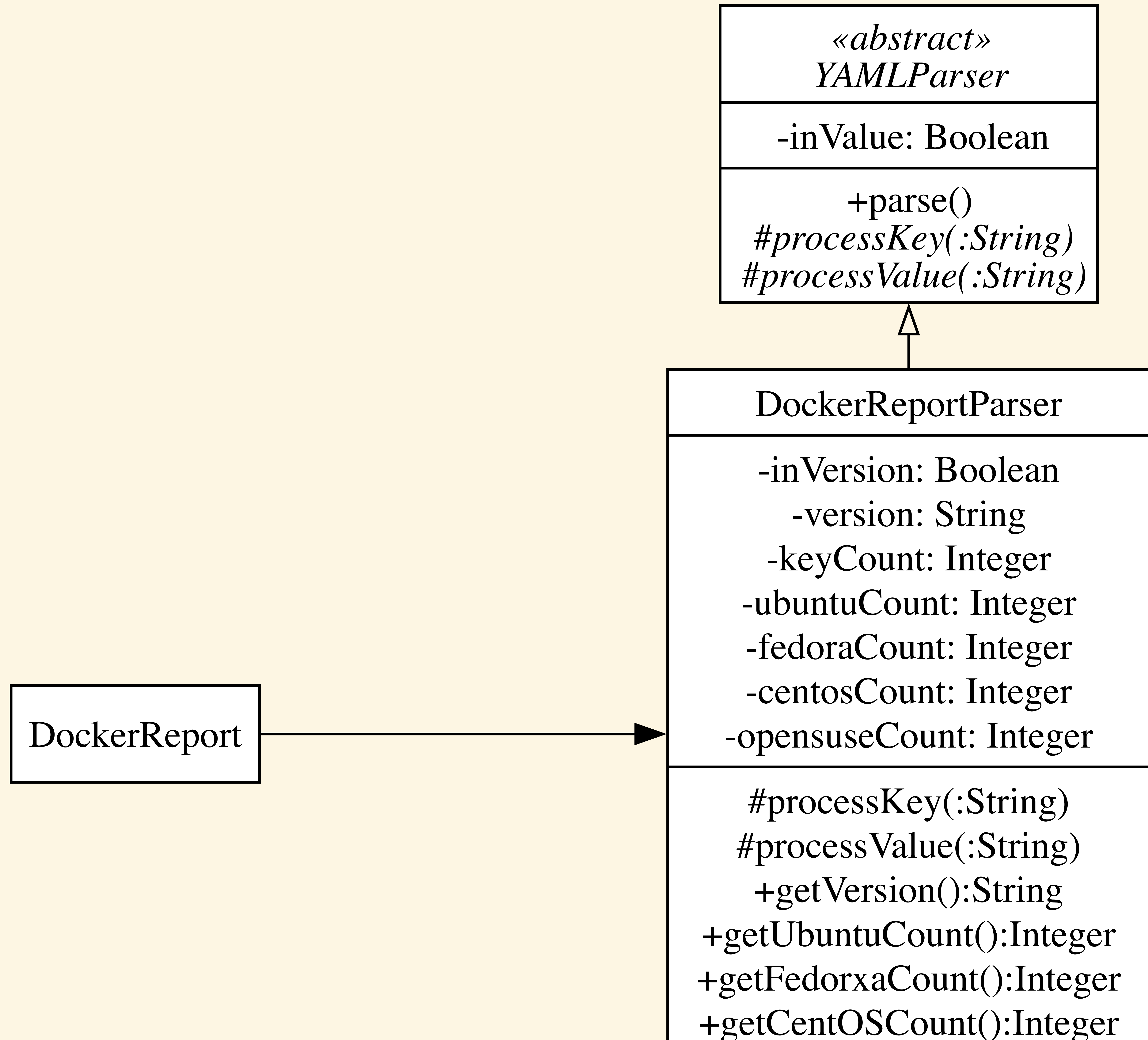


- Can provide standard, off-the-shelf handlers
- Clients can pick these or even further subclass them
- Users can even create their own sets of off-the-shelf handlers

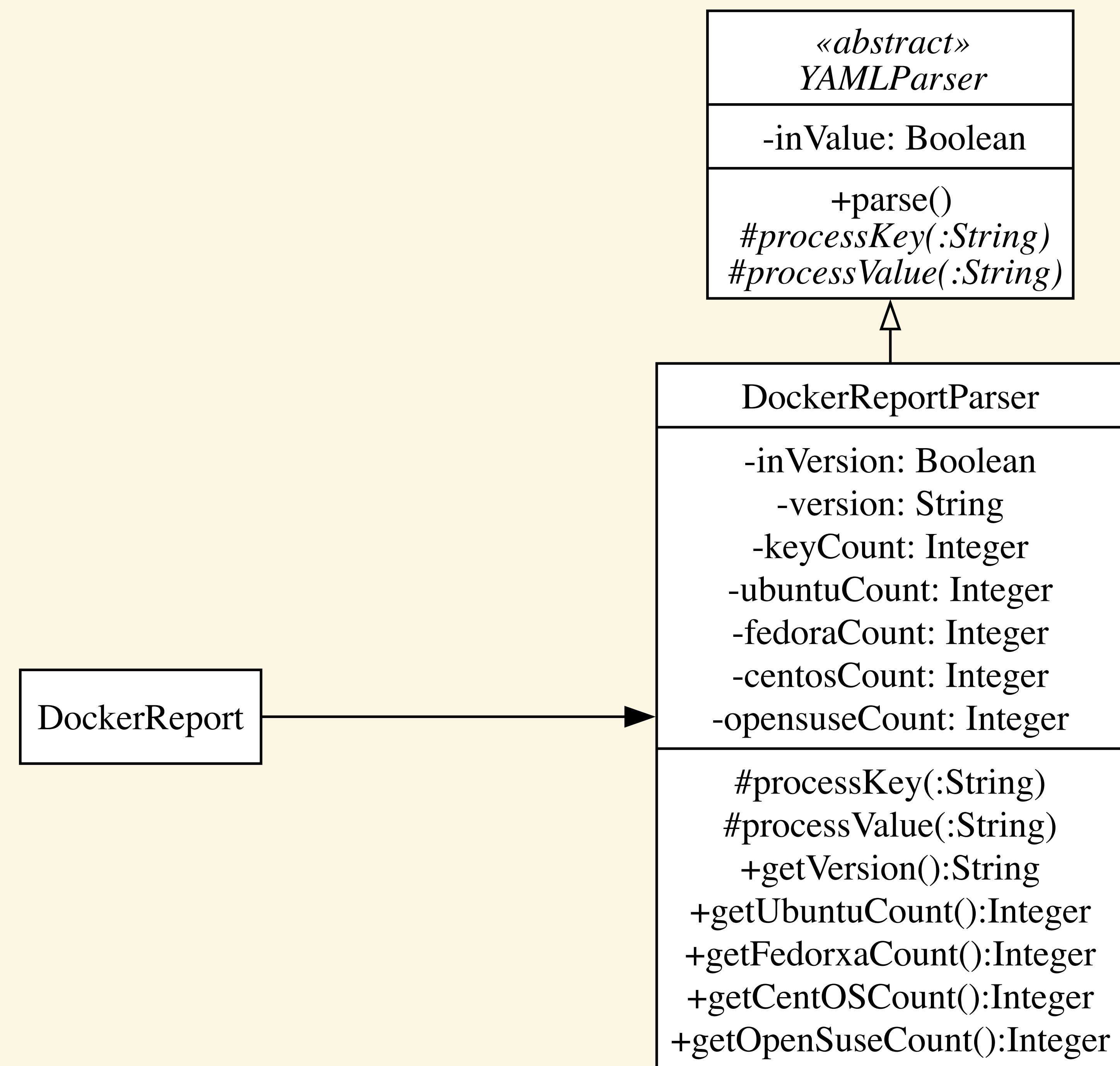
Advantages

- Replaces subclassing a large, complex class with subclassing a cohesive interface
- Handlers are almost always all pure virtual or empty methods
- The only dependencies that handlers have are for parameters and return types
- Allows multiple types of handlers, which the design can mix in new ways
- Allows standard handlers to be created and shared
- Easier to test, as can test individual handlers

DockerReport Template Method Design

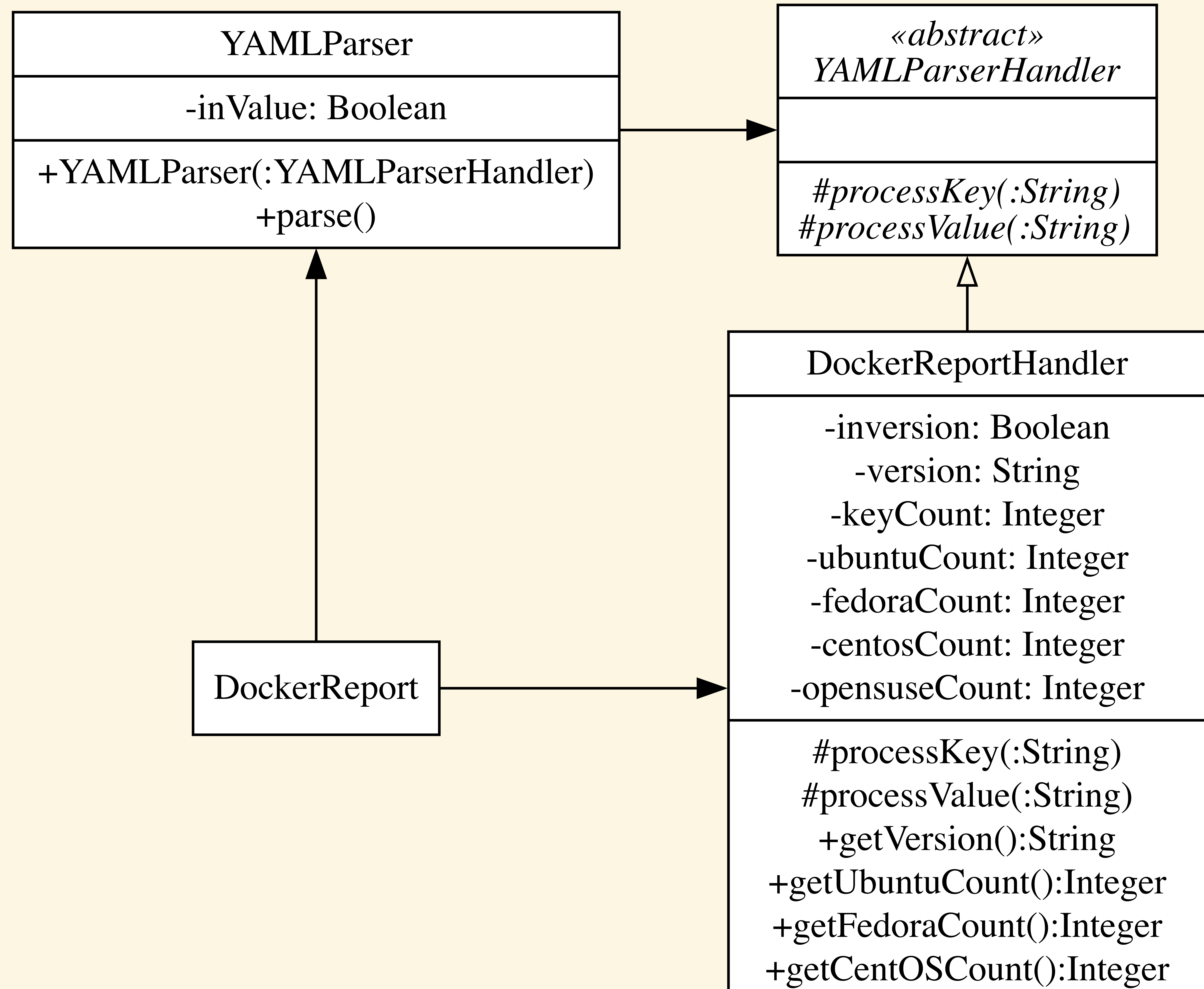


Template Method Includes

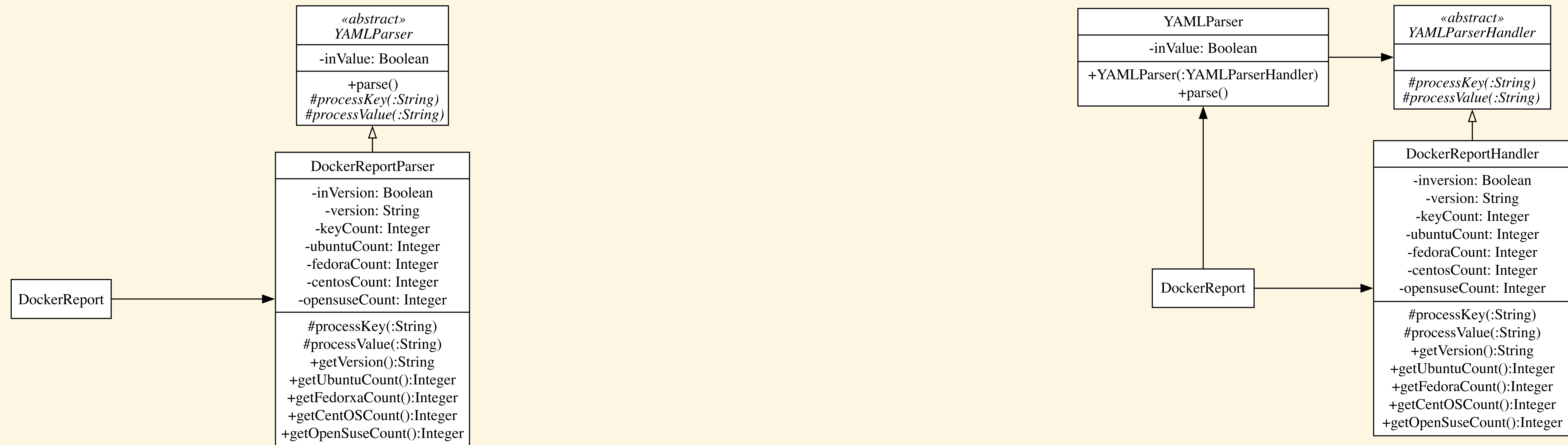



```
// DockerReport.cpp
#include "DockerReportParser.hpp"
    #include "YAMLParser.hpp"

// DockerReportParser.hpp
#include "YAMLParser.hpp"
```

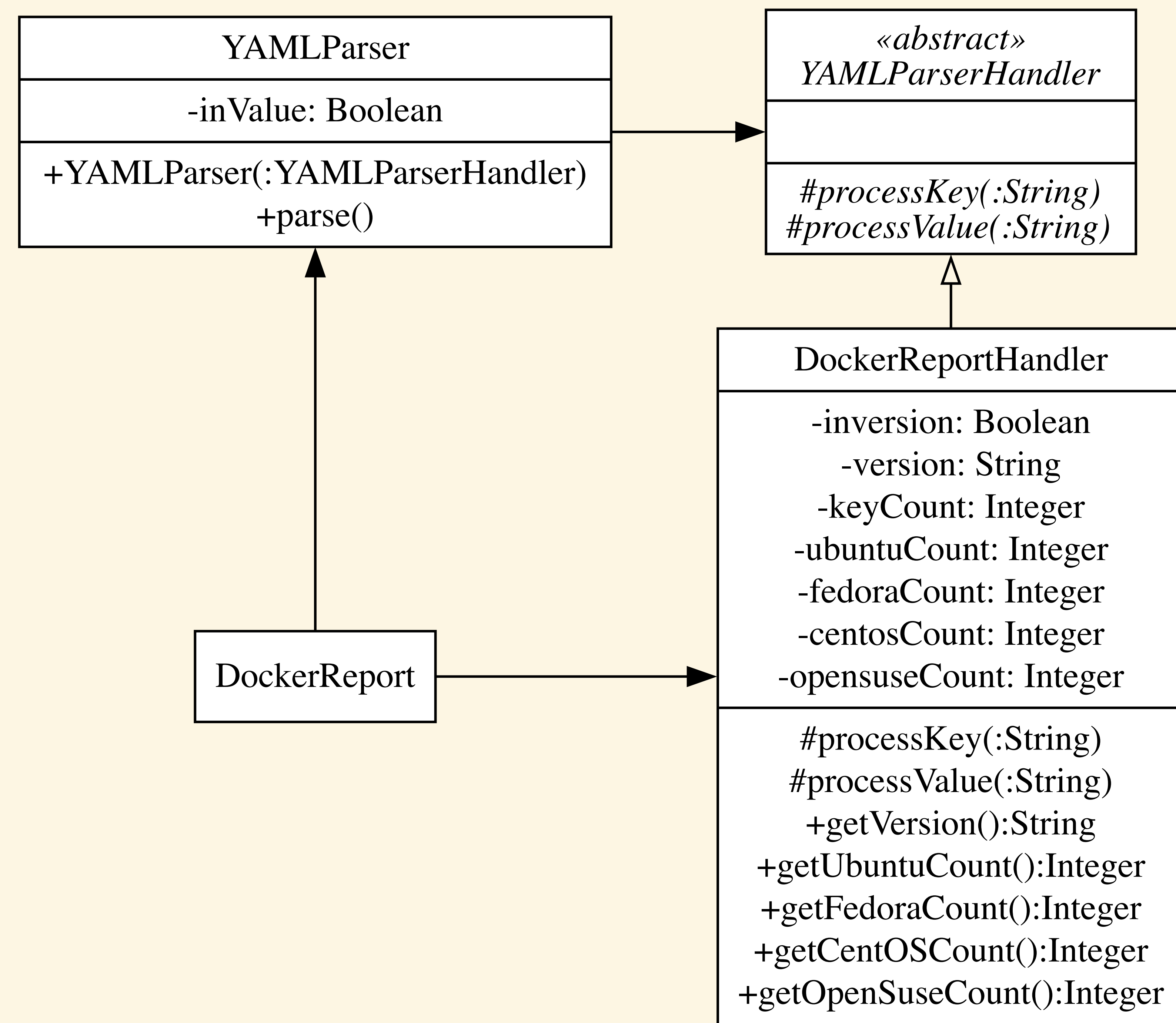
DockerReport: Handler Design



DockerReport: Template Method & Handler Comparison

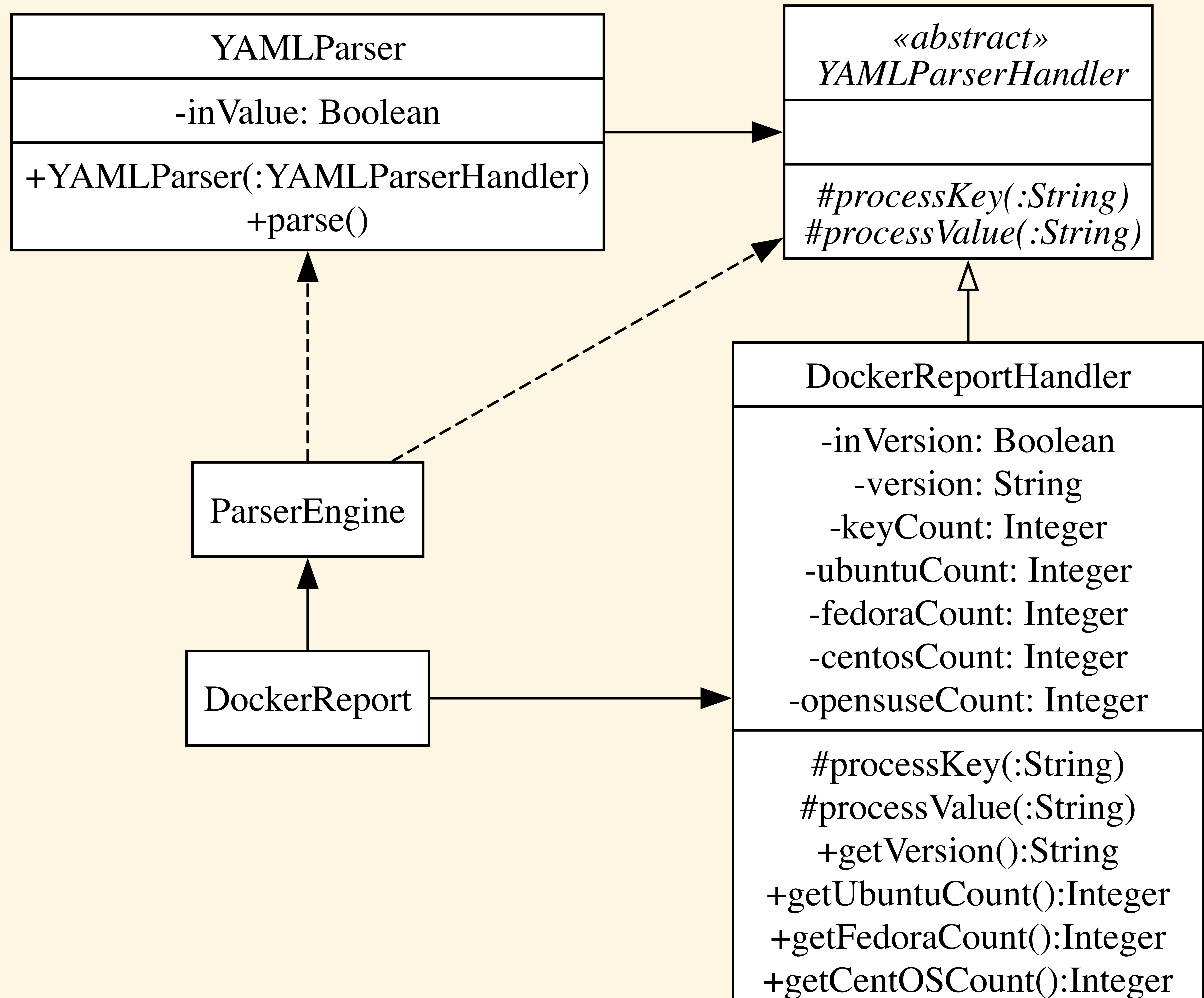


Handler Includes

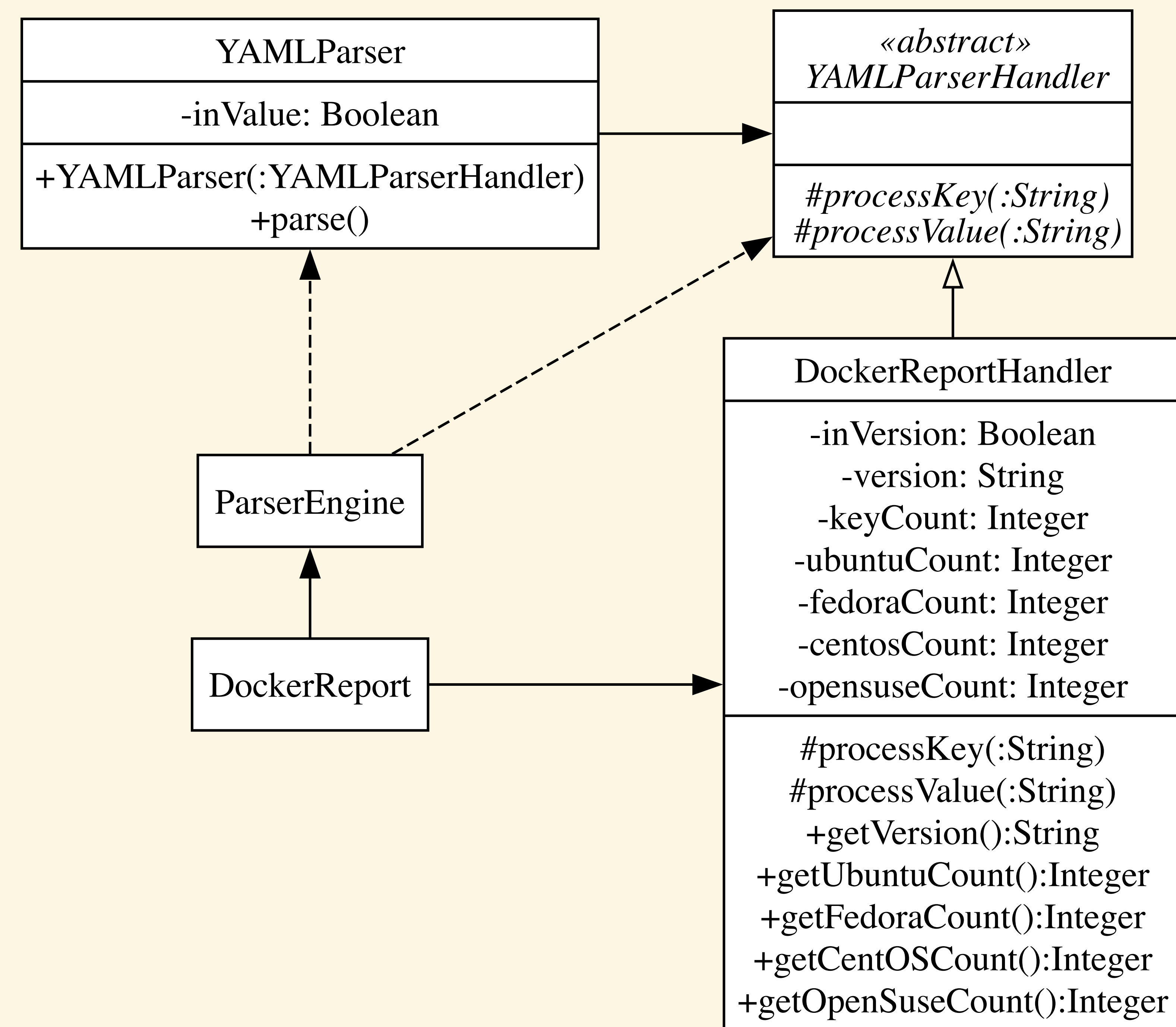



```cpp
// DockerReport.cpp
#include "DockerReportHandler.hpp"
    #include "YAMLParserHandler.hpp"
#include "YAMLParser.hpp"

// DockerReportHandler.hpp
#include "YAMLParserHandler.hpp"

// YAMLParser.hpp
#include "YAMLParserHandler.hpp"
```

Decoupling YAMLParser



Decoupling YAMLParser




```
// DockerReport.cpp
#include "DockerReportHandler.hpp"
    #include "YAMLParserHandler.hpp"
#include "ParserEngine.hpp"

// ParserEngine.hpp
#include "YAMLParserHandler.hpp"

// ParserEngine.cpp
#include "YAMLParser.hpp"

// DockerReportHandler.hpp
#include "YAMLParserHandler.hpp"

// YAMLParser.hpp
#include "YAMLParserHandler.hpp"
```

Decoupling YAMLParser

```
// DockerReport.cpp
#include "DockerReportHandler.hpp"
    #include "YAMLParserHandler.hpp"
#include "ParserEngine.hpp"

// ParserEngine.hpp
#include "YAMLParserHandler.hpp"

// ParserEngine.cpp
#include "YAMLParser.hpp"

// DockerReportHandler.hpp
#include "YAMLParserHandler.hpp"

// YAMLParser.hpp
#include "YAMLParserHandler.hpp"
```

- `ParserEngine` can be a free function, a static method of a class, a method of a class, or even part of a framework
- The implementation of `ParserEngine` (i.e., `.cpp` files) are the ones that use `YAMLParser`
- The interface of `ParserEngine` (i.e., `.hpp` files) does not use or include `YAMLParser` and is completely decoupled from `YAMParser`
- The interface of `ParserEngine` (i.e., `.hpp` files) does use `YAMLParserHandler` as a parameter