Design Patterns in KDE

Marc Mutz

Bielefeld University
Overview

- What Design Patterns are
- What they are not
- Why would I want to use one?
- Case Study: The KDE DOM implementation.
- Other Examples of Design Pattern use in KDE and Qt
- Examples of Patterns not yet used in KDE and Qt
What Design Patterns are

Design Patterns are

- A higher-level language than the programming language to talk about software systems in,
- A set of general solutions to common problems in software engineering,

Definition of Gamma et al: “Design Patterns are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.”

E.g., in C, **Inheritance** might be a **Design Pattern**, whereas in C++, it’s built into the language.
Design Patterns are not

- Algorithms such as QuickSort or Boyer-Moore.
- Language features, such as polymorphism or exceptions.
- Implementation patterns, such as the d-pointer or the virtual_hook.
- Software components such as toolkits and frameworks
Why would I want to use one?

Short answer: Because it makes you a better programmer.

Long answer: On learning Design Patterns, you will find that you have known and implemented at least some of the patterns already. You may have copied code or ideas, or you might have solved the problem yourself.

What you more often than not have **not** done is think of the solution as a pattern to **re-apply** whenever a similar problem arises again.

And almost certainly, you haven’t **named** them and put them in a catalog to refer to for **inspiration** and discussions about design.
A Case Study: KDE’s DOM Implementation
DOM::Document implements the Factory Method pattern:

```cpp
class Document : public Node {
    // ...
    // All of these are Factory Methods:
    Element createElement(...);
    Attr createAttribute(...);
    Event createEvent(...);
    // ...
};
```

Factory Methods abstract away the creation of objects. This is essential if you want to hide the implementation and only want to export the interface.
Almost all DOM classes implement the Bridge pattern (a.k.a. Handle):

- Node (the Abstractor) contains a member of type NodeImpl (the Implementor)
- Element (the Refined Abstractor) inherits Node

This pattern is not a d-pointer:

- d-pointer involves a dumb Data Object with no behavior.
- Implementor is stand-alone (contains all data and behavior).
Case Study: DOM: Composite

The Node-derived DOM classes implement the Composite pattern:
For a basic interface (Node), there are two classes of implementations:

- **Leaves** (Text, Comment) that contain no further children.
- **Composites** (Element, Attr) that may have one or more children.

The important aspect of this pattern is that container classes here implement the interface of the objects they contain.
Event, together with NodeImpl::dispatchEvent() implements a variant of the Chain of Responsibility pattern:

- **Handler** (Node) objects hold references to other Handlers, thus forming an object chain.
- A `handleRequest()` method on Handler either handles the request or passes it on to the next Handler in the chain.

Typically, **Handler** will be an interface so that the class of each Handler in the chain can be different. This is not the Chain of Responsibility as the book says, but similar enough.
The EventListener class implements the Observer pattern:

- A **Subject** (Node) registers **Observers** (EventListener)
- On state change, Subject calls Observer’s single method, in this case **handleEvent (Event&)**

You’ll find few Observers in KDE, since Qt’s **signal/slot** mechanism reduces the need for this pattern. Subjects know their type, so Observers can also be **Visitors**.
Other Examples of Design Pattern Use in KDE/Qt
Both Abstract Factory and Factory Method are implemented in QTextCodec:

```cpp
class QTextCodec {
    // ...

    static QTextCodec * codecForName( const char * name, ... );
    // ...

    virtual QTextDecoder * makeDecoder() const;
    virtual QTextEncoder * makeEncoder() const;
    // ...
};
```
KSieve::Parser implements the Builder pattern: Instead of constructing an explicit parse tree, it calls methods in the KSieve::ScriptBuilder interface:

```cpp
class ScriptBuilder {
    // ...
    virtual void commandStart(const QString & identifier ) = 0;
    virtual void commandEnd() = 0;
    virtual void taggedArgument(const QString & tag ) = 0;
    virtual void stringArgument(const QString & str, ... ) = 0;
    virtual void numberArgument(unsigned long number ) = 0;
    // ...
};
```
Implementations of this interface can reuse the parser to do totally different things, such as **pretty-printing** the script, creating a DOM-like **object tree**, test the parser...

```cpp
class PrettyPrintingScriptBuilder {
    // ...
    void commandStart( const QString & id ) {
        mStream << QString().fill( ' ', mIndent ) << id;
    }
    void commandEnd() { mStream << endl; }
    void numberArgument( unsigned long num ) { mStream << num; }
    // ...
private:
    QTextStream mStream;
    int mIndent;
};
```
QFrame implements the **Decorator** pattern:

- **QFrame**: *Is a QWidget, decorates one with a border.*
- **QScrollView**: *Is a QWidget, decorates one with scrollbar(s).*

The pattern here is:

- **Basic interface** (**QWidget**)
- **Concrete implementations** (**QPushButton**, etc)
- **Decorator implementations** that contain a member of basic type, which they decorate with additional state or functionality.
Design Pattern Use: Strategy

There are two examples of Strategy implementations in KMail:

**AttachmentStrategy** Encapsulates an algorithm to decide which attachments to display.

**HeaderStrategy** Encapsulates an algorithm to decide which header fields to display.

The pattern here is:

- Base interface (AttachmentStrategy)
- Concrete implementations ({Iconic,Smart,...}AttachmentStrategy).
- Context class that refers to a member of the abstract type to perform tasks.
class AttachmentStrategy {
    virtual bool inlineNestedMessages() const = 0;
};

class IconicAttachmentStrategy : public AttachmentStrategy {
    bool inlineNestedMessages() const { return false; }
};

class SmartAttachmentStrategy : public AttachmentStrategy {
    bool inlineNestedMessages() const { return true; }
};

class ObjectTreeParser {
    bool processMessageRfc822Subtype(...) {
        // ...
        if (mAttachmentStrategy->inlineNestedMessages() ) {
            // expand the message
        } else {
            // make an icon instead
        }
    }
};
The Rest
I promised to explain the **Null Object** pattern. You all know code that looks like this:

```cpp
if ( mFoo )
    mFoo->doSomething();
else
    doSomethingWithoutFoo();
```

If your implementation is cluttered with this type of code, and `mFoo` is **State** or **Strategy**-like, you can **refactor** to make the code use the **Null Object** pattern:
Refactoring: Introduce Null Object

if ( mFoo )
    mFoo->doSomething();
else
    doSomethingWithoutFoo();

1. Implement mFoo’s interface in a NullFoo class, where each method’s implementation consists of the else leg of conditionals like the above.

2. Instead of setting mFoo to null to indicate its absence, put a reference to a NullFoo instance there.

3. Remove the conditionals.
You’ve just seen one application of a technique called *Refactoring*

This is supposed to be a slide on refactoring, but that is for another talk, at another conference. ;-)

(or get your hands on the Book(s))