what is software design pattern or appplication in c++ QT? explain with examples and explain the examples in detail, each line

Software design patterns are reusable solutions to common design problems, and Qt (with C++) uses several patterns heavily, especially Observer (signals/slots) and Model–View (MVC/MVD). [1] [2]

Below are two Qt-focused patterns with detailed, line-by-line C++ examples.

Observer with signals/slots

Qt's signals and slots are an implementation of the Observer pattern: one object (subject) emits a signal, and any number of other objects (observers) receive it via slots. [3] [1]

Example: a simple counter that notifies a label whenever its value changes.

```
// counter.h
#include <QObject>

class Counter : public QObject {
    Q_OBJECT

public:
    explicit Counter(QObject *parent = nullptr);

    int value() const;

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};
```

Line by line:

- `#include <QObject>`: Includes Qt's base object class, needed for signals/slots. [2]
- `class Counter : public QObject {`: Declares a Counter class derived from QObject, which enables the meta-object features (signals, slots). [2]

- `Q_OBJECT`: Macro required in any class that defines signals or slots so Qt's meta-object compiler can generate code.[2]
- `public:`: Starts the public section, where the constructor and getter are accessible to all.
- `explicit Counter(QObject *parent = nullptr);`: Declares a constructor taking an optional parent QObject for Qt's object ownership system.[2]
- `int value() const;`: Declares a const member function that returns the current counter value without modifying the object.
- `public slots:`: Marks functions below as slots, which can be connected to signals and called automatically when signals are emitted.[1]
- `void setValue(int value);`: Slot that sets the counter's value; will be notified via signals.
- `signals:`: Marks functions below as signals, which can be emitted to notify observers.[1]
- `void valueChanged(int newValue);`: Signal emitted whenever the value changes, passing the new value to receivers.
- `private:`: Starts the private section, accessible only inside the class.
- `int m_value;`: Stores the internal counter value as an integer data member.
- `};`: Ends the class definition.

```
// counter.cpp
#include "counter.h"

Counter::Counter(QObject *parent)
    : QObject(parent),
      m_value(0)
{
}

int Counter::value() const
{
    return m_value;
}

void Counter::setValue(int value)
{
    if (m_value == value)
        return;

    m_value = value;
    emit valueChanged(m_value);
}
```

Line by line:

- `#include "counter.h"`: Includes the corresponding header so definitions match the declarations.
- `Counter::Counter(QObject *parent)`: Defines the constructor with the parent parameter.

- `: QObject(parent),`: Calls the base QObject constructor, registering this object in Qt's parent–child hierarchy.[2]
- `m_value(0)`: Initializes the internal counter value to 0 in the member initializer list.
- `{` and `}`: Empty constructor body; all initialization was done in the initializer list.
- `int Counter::value() const`: Defines the getter, marked const because it does not change the object.
- `{`: Begins function body.
- `return m_value;`: Returns the stored counter value.
- `}`: Ends the function.
- `void Counter::setValue(int value)`: Defines the setter slot for changing the value.
- `{`: Begins the function body.
- `if (m_value == value)`: Checks if the new value is the same as the current one.
- `return;`: Exits immediately if there is no change, avoiding unnecessary notifications.
- `m_value = value;`: Updates the internal stored value.
- `emit valueChanged(m_value);`: Emits the valueChanged signal so all connected observers are notified with the new value.[1]
- `}`: Ends the function.

Now connect this Counter to a QLabel in a small Qt Widgets app:

```
// main.cpp
#include <QApplication>
#include <QLabel>
#include "counter.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    Counter counter;
    QLabel label;

    label.setText("Initial: 0");
    label.show();

    QObject::connect(&counter, &Counter::valueChanged,
                     &label, [&label](int newValue) {
        label.setText(QString("Value: %1").arg(newValue));
    });

    counter.setValue(5);
    counter.setValue(10);

    return app.exec();
}
```

Line by line:

- `#include <QApplication>`: Brings in Qt's application class needed for any GUI program.[2]
- `#include <QLabel>`: Includes QLabel, a simple text-display widget.[2]
- `#include "counter.h"`: Includes the Counter class declaration to use it in main.
- `int main(int argc, char *argv[])`: Standard C++ entry point, with command-line arguments.
- `{`: Begins main function body.
- `QApplication app(argc, argv);`: Creates the Qt application object that manages the event loop and GUI resources.[2]
- `Counter counter;`: Instantiates a Counter object that will be the subject emitting signals.
- `QLabel label;`: Instantiates a QLabel that will act as an observer, updating when notified.
- `label.setText("Initial: 0");`: Sets the initial text shown on the label.
- `label.show();`: Makes the label visible on screen.[2]
- `QObject::connect(&counter, &Counter::valueChanged,`: Starts a connection between counter's signal and a slot (lambda).[1]
- `&label, [&label](int newValue) {`: Specifies the receiver (label) and defines a lambda function that takes the new value.
- `label.setText(QString("Value: %1").arg(newValue));`: Inside the lambda, updates the label text to show the latest value formatted into the string.
- `});`: Closes the connect call, finalizing the connection.
- `counter.setValue(5);`: Changes the counter's value to 5, which emits valueChanged(5) and updates the label.
- `counter.setValue(10);`: Changes the value to 10, emitting valueChanged(10) and updating the label again.
- `return app.exec();`: Starts Qt's event loop and returns its exit code when the app closes.[2]
- `}`: Ends main.

This pattern decouples the event source (Counter) from the UI component (QLabel), which improves modularity and reusability.[1]

Model–View (MVC/MVD) in Qt

Qt uses a Model–View architecture (and for QML, Model–View–Delegate) to separate data (model) from presentation (view). The model exposes data via a standard interface, and views like QListView or QTableView display it without knowing the data's internal structure.[4] [5] [2]

Example: show a list of strings with QListView and a QStringListModel.

```
// main.cpp
#include <QApplication>
#include <QListView>
```

```
#include <QStringListModel>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    QStringList data;
    data << "Apple" << "Banana" << "Cherry";

    QStringListModel model;
    model.setStringList(data);

    QListView view;
    view.setModel(&model);
    view.setWindowTitle("Fruits");
    view.show();

    return app.exec();
}
```

Line by line:

- `#include <QApplication>`: Includes the application class managing the event loop.[2]
- `#include <QListView>`: Brings in QListView, a standard item view widget implementing the view side of the model–view pattern.[2]
- `#include <QStringListModel>`: Includes QStringListModel, a simple model class that stores a list of strings.[2]
- `int main(int argc, char *argv[])`: Program entry point.
- `{`: Begins main function body.
- `QApplication app(argc, argv);`: Creates the GUI application instance.[2]
- `QStringList data;`: Declares a QStringList container to hold a list of strings.
- `data << "Apple" << "Banana" << "Cherry";`: Appends three fruit names to the list using the stream operator for convenience.
- `QStringListModel model;`: Instantiates a model object that will wrap the QStringList and expose it via the model interface.
- `model.setStringList(data);`: Sets the model's internal data to the QStringList so the view can access it.[2]
- `QListView view;`: Creates a QListView, which will display items provided by the model.
- `view.setModel(&model);`: Connects the view to the model; the view now queries the model for row count and data.[2]
- `view.setWindowTitle("Fruits");`: Sets the window title shown in the title bar.
- `view.show();`: Displays the view window so the user can see the list of items.[2]
- `return app.exec();`: Starts the Qt event loop and returns its exit code on application termination.
- `}`: Ends main.

In this pattern, the view does not own or manage the data; it simply asks the model what to display, which allows swapping models or views independently and keeps the design cleaner and more testable. [5] [2]

If you want, the next step can be a custom model (subclass QAbstractListModel) or an MVVM-style pattern with QML and C++ backend for a more advanced Qt design pattern example. [6] [7]

⁂

1. https://www.joulescope.com/blogs/blog/software-architecture-and-state
2. https://doc.qt.io/qt-6/model-view-programming.html
3. https://stackoverflow.com/questions/4625464/signals-and-slots-design-pattern-in-qt
4. https://doc.qt.io/qt-6/qtquick-modelviewsdata-modelview.html
5. https://radekp.github.io/qtmoko/api/model-view-introduction.html
6. https://forum.qt.io/topic/9157/design-patterns-qml-c
7. https://www.qt.io/product/qt6/qml-book/ch07-modelview-model-view
8. http://link.springer.com/10.1007/978-1-4842-3640-6
9. https://www.semanticscholar.org/paper/b516d9bde1885a335711b3013b31e174cd432ca8
10. https://www.semanticscholar.org/paper/5218a9ba8ef70cf2a4c0f0bfb72b3475bac2592e
11. https://services.igi-global.com/resolvedoi/resolve.aspx?doi=10.4018/IJITWE.370962
12. https://link.springer.com/10.1007/978-981-16-6269-0_21
13. https://www.mdpi.com/2076-3417/11/1/140
14. http://link.springer.com/10.1007/s10588-020-09308-7
15. https://www.semanticscholar.org/paper/fed853a5aa7392a0a8a645d31b5ca33fd52e702b
16. https://arxiv.org/pdf/2307.07069.pdf
17. https://arxiv.org/pdf/1805.05121.pdf
18. http://arxiv.org/pdf/1512.03487.pdf
19. http://arxiv.org/pdf/2303.07839.pdf
20. https://arxiv.org/html/2502.00202v1
21. https://www.scienceopen.com/document_file/39631d78-293f-4ccc-8204-e3d644625236/ScienceOpen/001_Lano.pdf
22. https://arxiv.org/pdf/1608.03355.pdf
23. https://arxiv.org/pdf/2202.11142.pdf
24. https://www.qt.io/product/qt6/qml-book/ch06-controls-common-patterns
25. https://www.youtube.com/watch?v=KaBCslzEUC0
26. https://www.geeksforgeeks.org/system-design/factory-method-pattern-c-design-patterns/
27. https://www.bogotobogo.com/DesignPatterns/introduction.php
28. https://www.ics.com/blog/exploring-model-view-design-qt-quick
29. https://forum.qt.io/topic/74425/signal-slot-mechanism-vs-observer-pattern-good-practice
30. https://www.youtube.com/watch?v=spbbVEAuguk

31. https://refactoring.guru/design-patterns/observer/cpp/example
32. https://www.qtcentre.org/threads/42649-Replace-the-typical-observer-pattern-to-the-Qt-s-signal-and-slot
33. https://ptgmedia.pearsoncmg.com/images/9780131879058/downloads/0131879057_Ezust_book.pdf
34. https://www.linkedin.com/pulse/qt-signals-slots-comprehensive-guide-vamsi-siddhani-mznfc
35. https://stackoverflow.com/questions/13422909/qt-model-view-controller-example
36. https://www.reddit.com/r/cpp/comments/5tp7om/whats_the_point_of_the_observer_patternsignals/
37. https://www.semanticscholar.org/paper/dc69c72877d9b6e11e80f6a1c6a0a1c2ff0ca4af
38. https://www.semanticscholar.org/paper/c4f5d27463b685535fbb25768b24f286de0472f3