1. Одиночка

Одиночка - гарантирует, что у класса есть только один экземпляр, и предоставляет глобальную точку доступа к этому экземпляру.

```
class Singleton
{
public:
        static Singleton& getInstance()
        {
                static Singleton instance;
                return instance;
        }

private:
        Singleton() { } // конструктор недоступен
        ~Singleton() { } // и деструктор
        // необходимо также запретить копирование
        Singleton(Singleton const&); // реализация не нужна
        Singleton& operator= (Singleton const&); // и тут
};

int main(int argc, char** argv) {
        //new Singleton(); // ошибка
        Singleton& instance = Singleton::getInstance();
        return 0;
}
```

Объяснение: В этом примере Singleton гарантирует, что в системе существует только один экземпляр логгера.

Метод getInstance создает экземпляр, если он еще не создан, и возвращает его.

2. Шаблонный метод

Шаблонный метод - определяет скелет алгоритма в базовом классе, оставляя реализацию некоторых шагов подклассам.

```
class BeverageMaker {
public:
        void prepareRecipe() {
```

```
                boilWater();
                brew();
                pourInCup();
                addCondiments();
        }
protected:
        virtual void brew() = 0; // Сварить
        virtual void addCondiments() = 0; // Добавить пряности
        void boilWater() {
                cout << "Boiling water!" << endl;
        }

        void pourInCup() {
        cout << "Pouring into cup!" << endl;
        }

};

class TeaMaker : public BeverageMaker {
protected:
        void brew() override {
                cout << "Steepint the tea!" << endl;
        }
        void addCondiments() override {
                cout << "Adding lemon and honey!" << endl;
        }

};

class CoffeeMaker : public BeverageMaker {
protected:
        void brew() override {
                cout << "Dripping coffee trought filter!" << endl;
        }
        void addCondiments() override {
                cout << "Adding sugar and milk!" << endl;
        }
};
```

Объяснение: В этом примере BeverageMaker определяет скелет приготовления напитка, а подклассы TeaMaker и CoffeeMaker реализуют конкретные шаги приготовления чая и кофе соответственно.

3. Фабричный метод

Фабричный метод - определяет интерфейс для создания объектов, но позволяет подклассам решать, какой класс инстанцировать (чей экземпляр создавать)

```
class Button {
public:
        virtual void render() = 0;
};

class WindowsButton : public Button {
public:
        void render() override {
                cout << "Rendering a Windows button..." << endl;
        }
};

class WebButton : public Button {
public:
        void render() override {
                cout << "Rendering a Web button..." << endl;
        }
};

class Dialog {
public:
        void render() {
            Button* button = createButton();
            button->render();

            delete button;
        }

protected:
        virtual Button* createButton() = 0;
};

class WindowsDialog : public Dialog {
protected:
        Button* createButton() override {
                return new WindowsButton();
        }
};

class WebDialog : public Dialog {
protected:
        Button* createButton() override {
            return new WebButton();
```

```
        }
};
```

Объяснение: В этом примере Dialog определяет фабричный метод createButton, который подклассы WindowsDialog и WebDialog переопределяют для создания кнопок для разных платформ.

4. Абстрактная фабрика

Абстрактная фабрика - предоставляет интерфейс для создания семейств взаимосвязанных или взаимозависимых объектов без указания их конкретных классов.

Примечание: паттерны фабричный метод и абстрактная фабрика тесно связаны дург с другом, в нижеприведённом коде инициалы AF обозначают сокращение об Abstract Factory (Абстрактная фабрика).

```
class AFButton {
public:
  virtual void paint() = 0;
};

class AFCheckbox {
public:
  virtual void paint() = 0;
};

class AFWindowsButton : public AFButton {
public:
  void paint() override {
    cout << "Painting a Windows button..." << endl;
  }
};

class AFWindowsCheckbox : public AFCheckbox {
public:
  void paint() override {
    cout << "Painting a Windows checkbox..." << endl;
  }
};

class AFWebButton : public AFButton {
public:
  void paint() override {
    cout << "Painting a Web button..." << endl;
```

```
  }
};

class AFWebCheckbox : public AFCheckbox {
public:
  void paint() override {
    cout << "Painting a Web checkbox..." << endl;
  }
};

class AFGUIFactory {
public:
  virtual AFButton* createButton() = 0;
  virtual AFCheckbox* createCheckbox() = 0;
};

class AFWindowsFactory : public AFGUIFactory {
public:
  AFButton* createButton() override {
    return new AFWindowsButton();
  }

  AFCheckbox* createCheckbox() override {
    return new AFWindowsCheckbox();
  }
};

class AFWebFactory : public AFGUIFactory {
  AFButton* createButton() override {
    return new AFWebButton();
  }

  AFCheckbox* createCheckbox() override {
    return new AFWebCheckbox();
  }
};

void clientCode(AFGUIFactory& factory) {
  AFButton* button = factory.createButton();
  AFCheckbox* checkbox = factory.createCheckbox();

  button->paint();
  checkbox->paint();

  delete button;
```

```
  delete checkbox;
}
```

Объяснение: В этом примере AFGUIFactory определяет интерфейс для создания кнопок и чекбоксов.

Подклассы AFWindowsFactory и AFWebFactory реализуют этот интерфейс для создания компонентов для разных платформ.

5. Стратегия

Стратегия - определяет семейство алгоритмов, инкапсулирует (скрывает решение) каждый из них и делает их взаимозаменяемыми.

```
class SortStrategy {
public:
  virtual void sort(vector<int>& data) = 0;
};

class BubbleSort : public SortStrategy {
public:
  void sort(vector<int>& data) override {
    cout << "Sorting using Bubble sort..." << endl;
  }
};

class QuickSort : public SortStrategy {
public:
  void sort(vector<int>& data) override {
    cout << "Sorting using Quick sort..." << endl;
  }
};

class SortContext {
private:
  SortStrategy* m_strategy;

public:
  SortContext(SortStrategy* strategy) : m_strategy(strategy) {}

  void setStrategy(SortStrategy* strategy) {
    m_strategy = strategy;
  }
```

```
  void sort(vector<int>& data) {
    m_strategy->sort(data);
  }
};
```

Объяснение: В этом примере SortingContext использует различные стратегии сортировки, такие как BubbleSort и QuickSort.

Контекст может динамически менять стратегию сортировки, что делает алгоритмы взаимозаменяемыми.

Использование в main

```
int main() {
  setlocale(LC_ALL, "Rus");

  cout << "\t1. Одиночка:" << endl;
  Logger* logger = Logger::getInstance();
  logger->log("Logging a message");

  cout << "\n\t2. Шаблонный метод:" << endl;
  TeaMaker teaMaker;
  CoffeeMaker coffeeMaker;

  cout << "Making tea..." << endl;
  teaMaker.prepareRecipe();
  cout << endl;
  cout << "Making coffee..." << endl;
  coffeeMaker.prepareRecipe();

  cout << "\n\t3. Фабричный метод:" << endl;
  Dialog* dialog = new WindowsDialog();
  dialog->render();
  delete dialog;

  dialog = new WebDialog();
  dialog->render();
  delete dialog;

  cout << "\n\t4. Абстрактная фабрика:" << endl;
  AFGUIFactory* factory = new AFWindowsFactory();
  clientCode(*factory);
  delete factory;
```

```
    cout << endl;
    factory = new AFWebFactory();
    clientCode(*factory);
    delete factory;

    cout << "\n\t5. Стратегия:" << endl;
    vector<int> data{ 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5 };
    SortContext context(new BubbleSort());
    context.sort(data);

    context.setStrategy(new QuickSort());
    context.sort(data);

    return 0;
}
```