ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ ПРОГРАММИРОВАНИЕ

Лекция № 2 / **9**
29.10.2019 г.

ПОВЕДЕНЧЕСКИЕ ПАТТЕРНЫ

Singleton — не только паттерн

ITERATOR

- *Итератор* используется для обхода контейнера и доступа к его элементам.
- Позволяет отвязать алгоритмы от конкретных контейнеров.

```
struct is_positive_number {
    bool operator()(int x) { return 0 < x; }
};

int main()
{
    int nums[] = { 0, -1, 4, -3, 5, 8, -2 };
    const int N = sizeof(nums) / sizeof(nums[0]);

    int *numbers = nums;

    typedef boost::filter_iterator<is_positive_number, int *> FilterIter;

    is_positive_number predicate;
    FilterIter filter_iter_first(predicate, numbers, numbers + N);
    FilterIter filter_iter_last(predicate, numbers + N, numbers + N);

    std::copy(filter_iter_first, filter_iter_last,
              std::ostream_iterator<int>(std::cout, " "));
    std::cout << std::endl;
}
```

Ленивые вычисления с помощью итераторов

РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ ITERATOR

Достоинства:

- Упрощает классы хранения данных.
- Позволяет реализовать различные способы обхода структуры данных.
- Позволяет одновременно перемещаться по структуре данных в разные стороны.

Недостатки:

- Не оправдан, если можно обойтись простым циклом.

COMMAND

- *Команда* инкапсулирует запрос как самостоятельный объект.

- Выполнение запроса «отрывается» от его создания.

- Запросы могут помещаться в очередь, протоколироваться, *отменяться*.

```
class TextBuffer {
    // ...
};

class EditorCommand {
public:
    EditorCommand(TextBuffer *);
    virtual void redo() = 0;
    virtual void undo() = 0;

private:
    TextBuffer *textbuf;
};
```

```
class DeleteTextCommand : public EditorCommand {
    // ...
};

class SearchAndReplaceCommand : public EditorCommand {
    // ...
};

class InsertBlockCommand : public EditorCommand {
    // ...
};

// .... другие команды ....
```

```
class Editor {
public:
    // ....
    void onCtrlVPressed() {
        addAndExecuteCommand(new InsertBlockCommand(clipboardContents()));
    }

    void addAndExecuteCommand(EditorCommand *cmd) {
        commands.push_back(cmd);
        cmd->redo();
    }

    void undo() {
        if (commands.empty()) return;
        auto cmd = commands.back();
        cmd->undo();
        delete cmd;
        commands.pop_back();
    }
private:
    std::vector<EditorCommand *> commands;
};
```

РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ COMMAND

Достоинства:

- Убирает прямую зависимость между объектам, вызывающими операции, и объектами, которые их непосредственно выполняют.
- Позволяет реализовать простую отмену и повтор операций.
- Позволяет реализовать отложенный запуск операций.
- Позволяет собирать сложные команды из простых.

Недостатки:

- Усложняет код программы из-за введения множества дополнительных классов.

TEMPLATE METHOD

- *Шаблонный метод* определяет основу алгоритма и позволяет подклассам переопределить некоторые его шаги.

- Базовый класс определяет шаги алгоритма с помощью абстрактных операций, а производные классы их реализуют.

- Концепция хуков *(hooks)*.

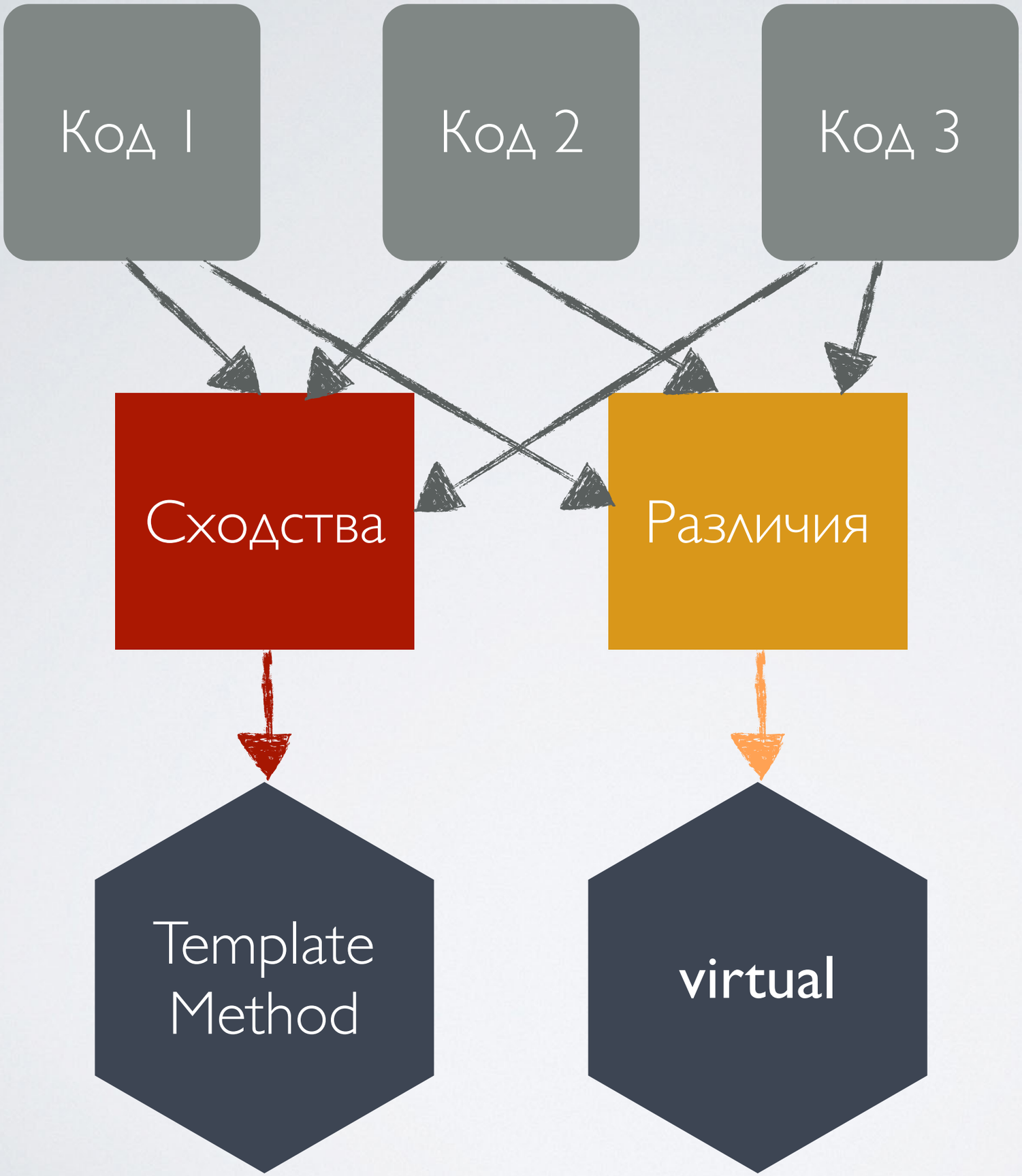
```cpp
class Account {
public:
    virtual void start() = 0;
    virtual void allow() = 0;
    virtual void end() = 0;
    virtual int maxLimit() = 0;

    // Template Method
    void withdraw(int amount) {
        start();
        if (amount < maxLimit())
            allow();
        else
            cout << "Not allowed"
                 << endl;
        end();
    }
};
```

```cpp
int main() {
    AccountPower power;
    power.withdraw(1500);

    AccountNormal normal;
    normal.withdraw(1500);
}
```

```cpp
class AccountNormal : public Account {
public:
    void start() {
        cout << "Start ..." << endl;
    }
    void allow() {
        cout << "Allow ..." << endl;
    }
    void end() {
        cout << "End ..."   << endl;
    }
    int maxLimit() { return 1000; }
};

class AccountPower : public Account {
public:
    void start() {
        cout << "Start ..." << endl;
    }
    void allow() {
        cout << "Allow ..." << endl;
    }
    void end() {
        cout << "End ..."   << endl;
    }
    int maxLimit() { return 5000; }
};
```

STL же!

```
void print_number(int i) {
    cout << i << ' ';
}

for_each(numbers.begin(), numbers.end(),
         print_number);

// ------------------------------------------

bool is_odd(int i) { return i % 2 == 1; }

std::vector<int>::iterator bound;
bound = std::partition(numbers.begin(),
                       numbers.end(),
                       is_odd);
```



Лечение копипаста

- Инвертированная структура кода: родительский класс вызывает методы подкласса, а не наоборот. Принцип Голливуда: **«Не звоните нам, мы сами вам позвоним»**.

Принцип инверсии зависимостей

- *Hooks (callbacks, крючки, зацепки)* — определяются по желанию. Как правило, в базовом классе это функции с пустыми телами.

- **virtual foo() = 0; заставляет** определять функции в подклассе.

РЕЗУЛЬТАТЫ ПРИМЕНЕНИЯ TEMPLATE METHOD

Достоинства:

- Облегчает повторное использование кода.

Недостатки:

- Вы жёстко ограничены скелетом существующего алгоритма.
- Вы можете нарушить принцип подстановки Барбары Лисков, изменяя базовое поведение одного из шагов алгоритма через подкласс.
- С ростом количества шагов шаблонный метод становится слишком сложно поддерживать.

STATE

- *Состояние* позволяет объекту варьировать свое поведение в зависимости от внутреннего состояния.
- Как будто бы меняется класс объекта.

```
class Machine;
class OffState;

class BaseState {
    Machine *machine;
public:
    BaseState(Machine *m) : machine(m) {}
    virtual void on()  = 0;
    virtual void off() = 0;
};

class OnState : public BaseState {
public:
    void on()  { cout << "Machine Already ON!" << endl; }
    void off() {
        machine->setState(new OffState(machine));
        cout << "Machine Turned OFF!" << endl;
    }
};

class OffState : public BaseState {
public:
    void off() { cout << "Machine Already OFF!" << endl; }
    void on()  {
        machine->setState(new OnState(machine));
        cout << "Machine Turned ON!" << endl;
    }
};
```

```
class Machine {
    BaseState *state;
public:
    Machine() {
        state = new OffState(this);
    }
    ~Machine() { delete state; }

    void on()  { state->on(); }
    void off() { state->off(); }

    void setState(BaseState *new_state) {
        delete state;
        state = new_state;
    }
};
```

TCPConnection

Состояния:

Established

Listening

Closed

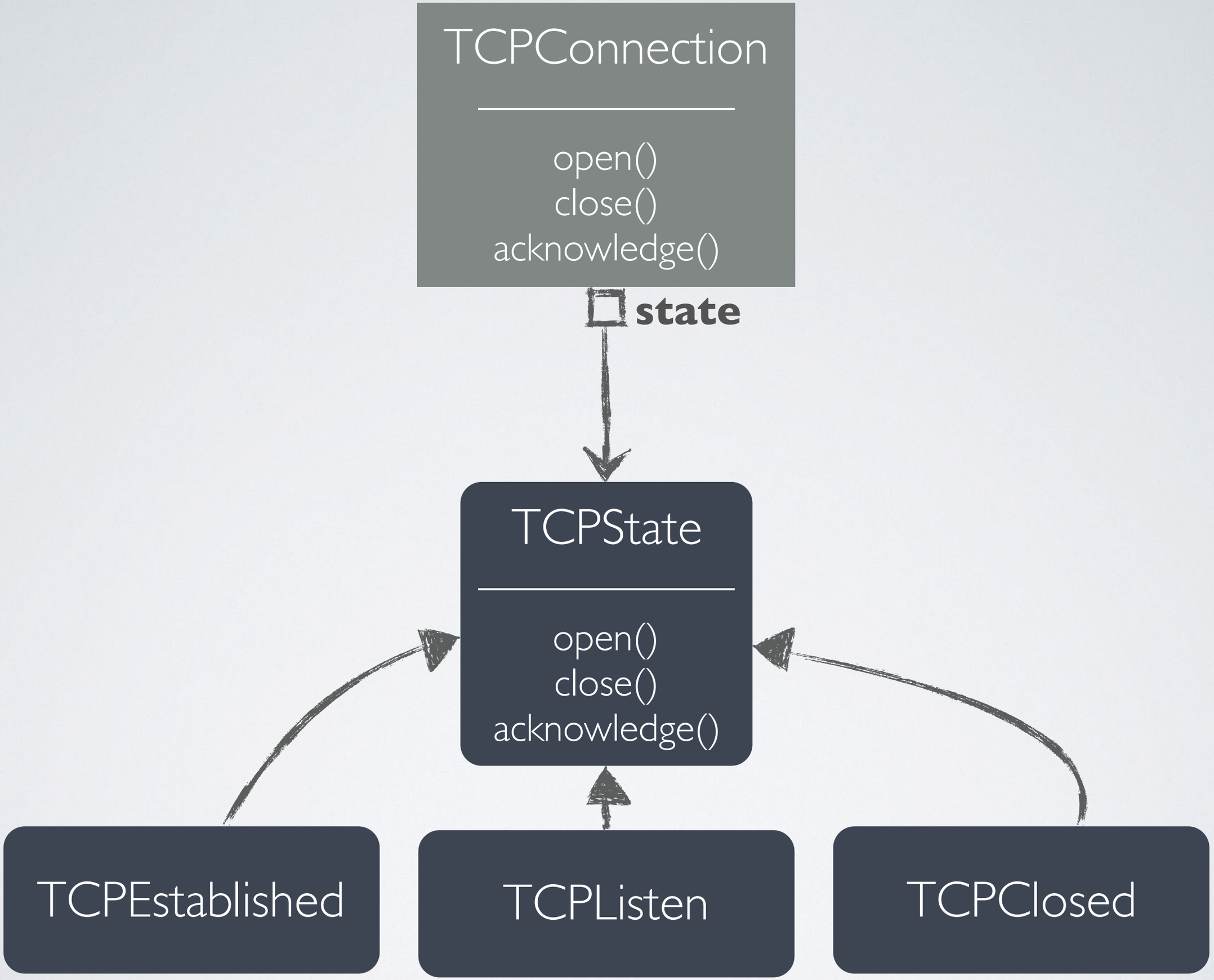
... 3x3 варианта ...

Методы:

open()

close()

acknowledge()


TCPConnection
open()
close()
acknowledge()
state
TCPState
open()
close()
acknowledge()
TCPEstablished
TCPListen
TCPClosed

- Избавляемся от кучи **if**-ов.
- Легко добавить новое состояние.
- Переходы между состояниями может делать как основной класс, так и классы состояний.
- Классы состояний можно *наследовать* друг от друга (если поведение несильно отличается).
- Можно сделать классы состояний *синглтонами* (если все данные лежат в основном классе).

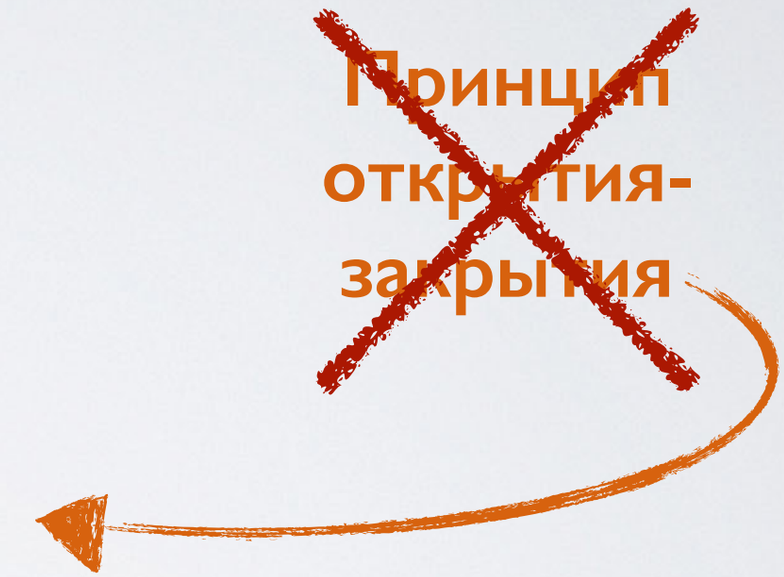
Принцип открытия-закрытия

STRATEGY

- *Стратегия* определяет семейство алгоритмов, инкапсулирует каждый из них и делает взаимозаменяемыми.

- Алгоритмы изменяются независимо от клиентов, которые ими пользуются.

```
enum FormatStyle {
    LEFT_JUSTIFY, RIGHT_JUSTIFY, CENTER
};

std::string formatText(const std::string &text, FormatStyle style) {

    // ...

    if (style == LEFT_JUSTIFY) {
        // ...
    } else if (style == CENTER) {
        // ...
    }

    // ...

    switch (style) {
        case LEFT_JUSTIFY:
            // ...
        case RIGHT_JUSTIFY:
            // ...
        case CENTER:
            // ...
    }

    // ...

    if (style != CENTER) {
        // ....
    }

    // ...
}
```



```
class FormattingStrategy {
public:
      FormattingStrategy(int width) { /* ... */ }

      virtual std::string justify(std::string text) = 0;
};

class LeftJustifyStrategy : public FormattingStrategy {
public:
      // ...
};

class RightJustifyStrategy : public FormattingStrategy {
public:
      // ...
};

class CenterStrategy : public FormattingStrategy {
public:
      // ...
};

class TextFormatter {
public:
    enum {
          LEFT_JUSTIFY,
          RIGHT_JUSTIFY,
          CENTER,
          // ....
    };

    void setStrategy(int type, int width);
    std::string justify(std::string text) = 0;
};
```

- Можно было бы сделать
  - LeftJustifyTextFormatter
  - RightJustifyTextFormatter
  - CenterTextFormatter
  - … унаследованные от базового класса TextFormatter,
- **… но … ?**

- … Тогда:
  - смешивается алгоритм и контекст;
  - нельзя динамически поменять алгоритм;
  - если используется несколько не связанных алгоритмов (*напр.: форматирование по ширине и расстановка переносов*), возникает взрыв классов-наследников.

Шаблонный метод

- Алгоритм фиксирован, отдельные шаги в подклассах могут меняться.

Стратегия

- Алгоритм не фиксирован, в рамках интерфейса меняется что угодно.

```
template <typename T>
struct OpNewCreator {
    static T *create() { return new T; }
};

template <typename T>
struct MallocCreator {
    static T *create() {
        void *buf = std::malloc(sizeof(T));
        if (!buf) return nullptr;
        return new(buf) T;
    }
};
```

Стратегии на темплейтах

Шаблон — аргумент шаблона

```
template <template <typename Created> CreationPolicy>
class WidgetManager : public CreationPolicy {
    // ....
};

typedef WidgetManager<OpNewCreator> MyWidgetMgr;
```

```
template <typename T,
          template <typename> class CheckingPolicy,
          template <typename> class ThreadingModel>
class SmartPtr;

typedef SmartPtr<Widget, NoChecking, SingleThreaded>      WidgetPtr;
typedef SmartPtr<Widget, EnforceNotNull, SingleThreaded> SafeWidgetPtr;
```

Две ортогональных стратегии для SmartPtr

```
template <typename T>
struct NoChecking {
    static void check(T *) {}
};


template <typename T>
struct EnforceNotNull {
    class NullPointerException : public std::exception { /*...*/ };

    static void check(T *ptr) {
        if (!ptr)
            throw NullPointerException();
    }
};


template <typename T>
struct EnsureNotNull {
    static void check(T *&ptr) {
        if (!ptr)
            ptr = getDefaultValue();
    }
};
```

```
template <typename T,
          template <typename> class CheckingPolicy,
          template <typename> class ThreadingModel>
class SmartPtr
    : public CheckingPolicy<T>,
      public ThreadingModel<SmartPtr>
{
    // ...
    T *operator->() {
        typename ThreadingModel<SmartPtr>::Lock guard(*this);
        CheckingPolicy<T>::check(pointer);
        return pointer;
    }

private:
    T *pointer;
};
```

- Необходимо раскладывать классы на *ортогональные* стратегии, не нуждающиеся в информации друг о друге.
- Повышается эффективность программы, ибо связывание делается на этапе компиляции, механизм виртуальных функций не нужен.
- Теряется возможность динамического изменения стратегий во время выполнения.