СЕМИНАР 10

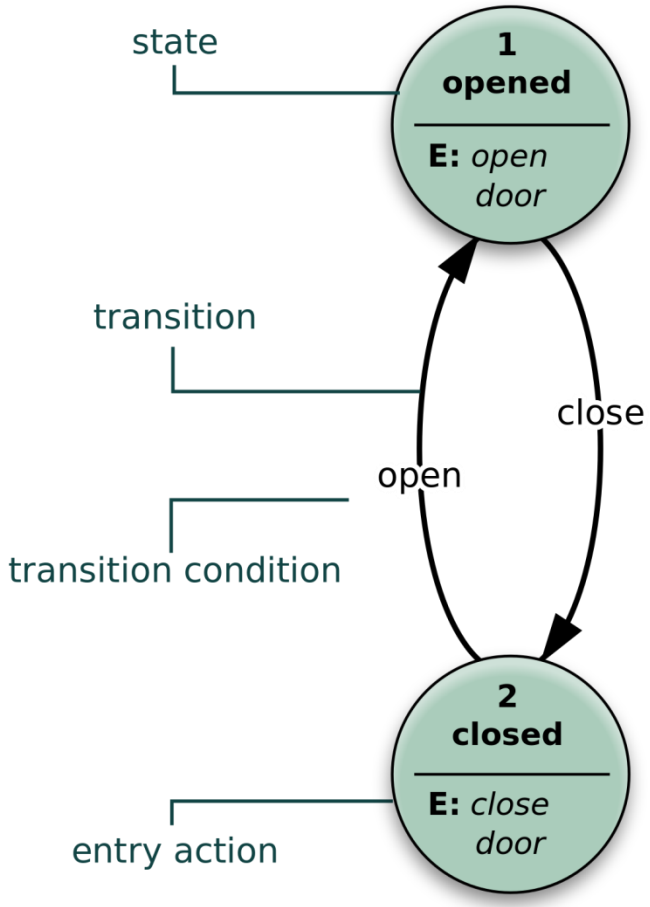
Конечный автомат

QStateMachine

Что такое конечный автомат?

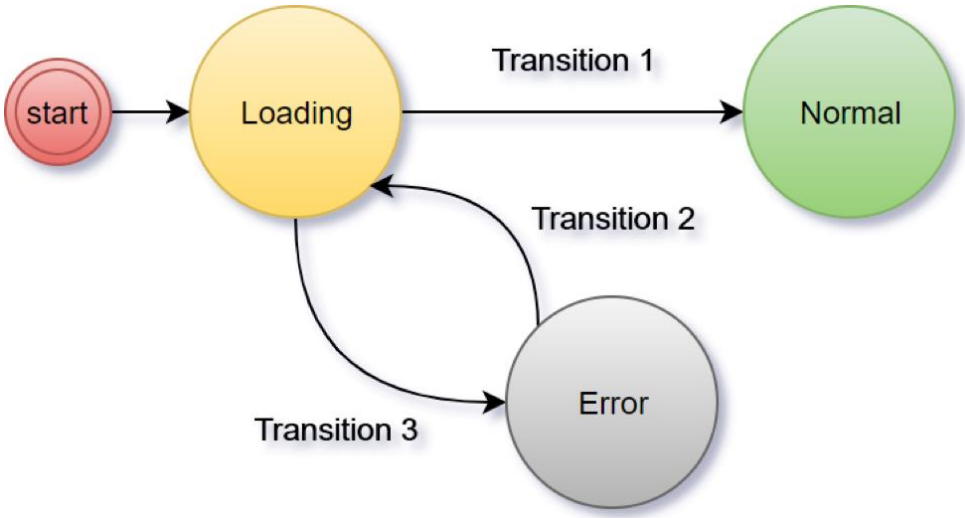
Конечный автомат – математическая абстракция, которая состоит из:

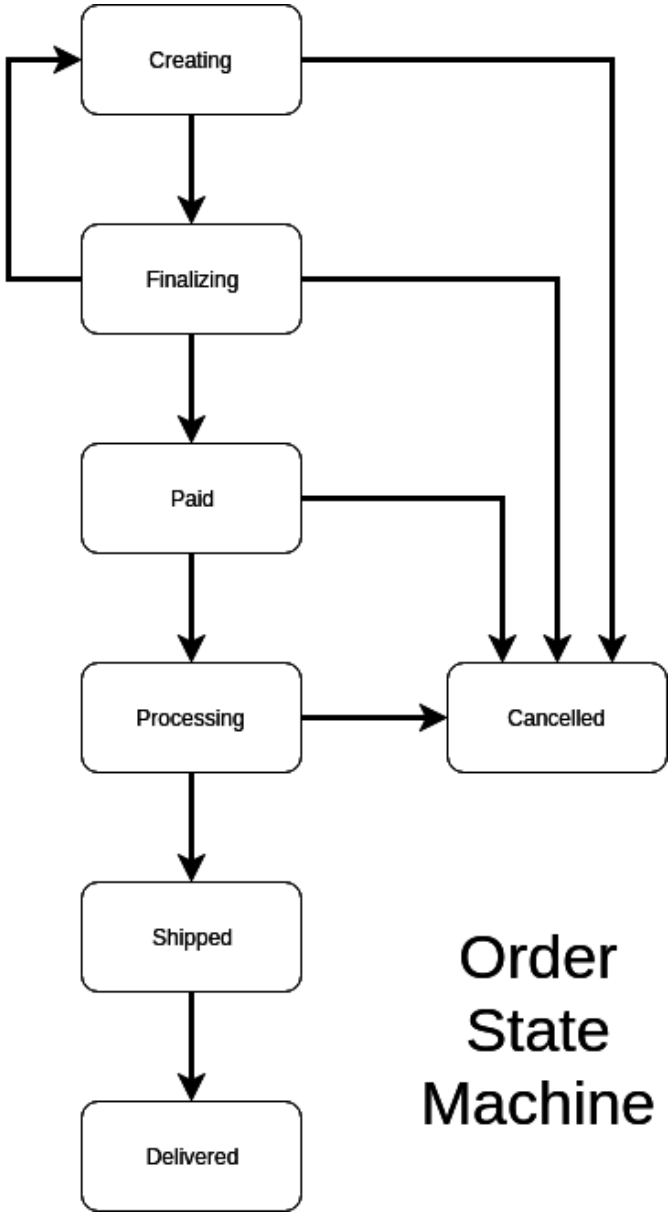
- Множества внутренних состояний;
- Множества входных сигналов, которые определяют переход из текущего состояния в следующее,
- Множества конечных состояний, при переходе в которые автомат завершает работу.



Конечный автомат

Представление того, как система может реагировать на внешние воздействия.





Зачем используют конечный автомат?

- Формализация: при анализе задачи продумываются все состояния приложения, переходы между ними, т.е. задача продумывается более детально, а также на выходе такого анализа будет документация на софт.
- Контроль ошибок;
- История операций;
- Ключевая особенность приложений, что их поведение зависит не только от текущего состояния, но и от предыдущих событий.

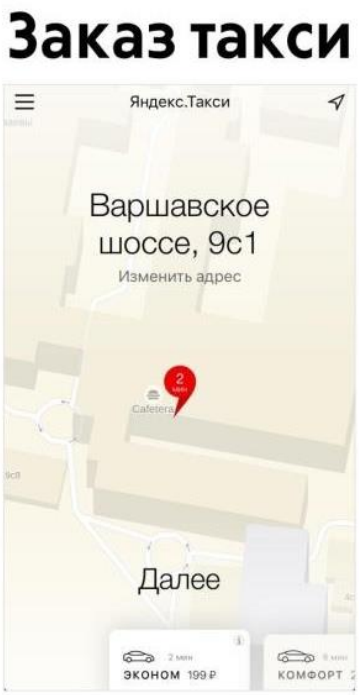
В каких случаях есть смысл использовать конечный автомат?

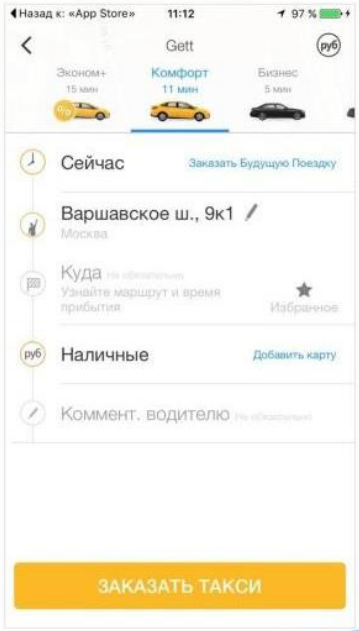
- Программа содержит ограниченное количество состояний
- В любой момент времени программа может быть в одном из этих состояний
- Программа может переключаться из состояния в состояние
- Когда поведение программы можно представить как «жизненный цикл» перехода из состояние в состояние

Примеры

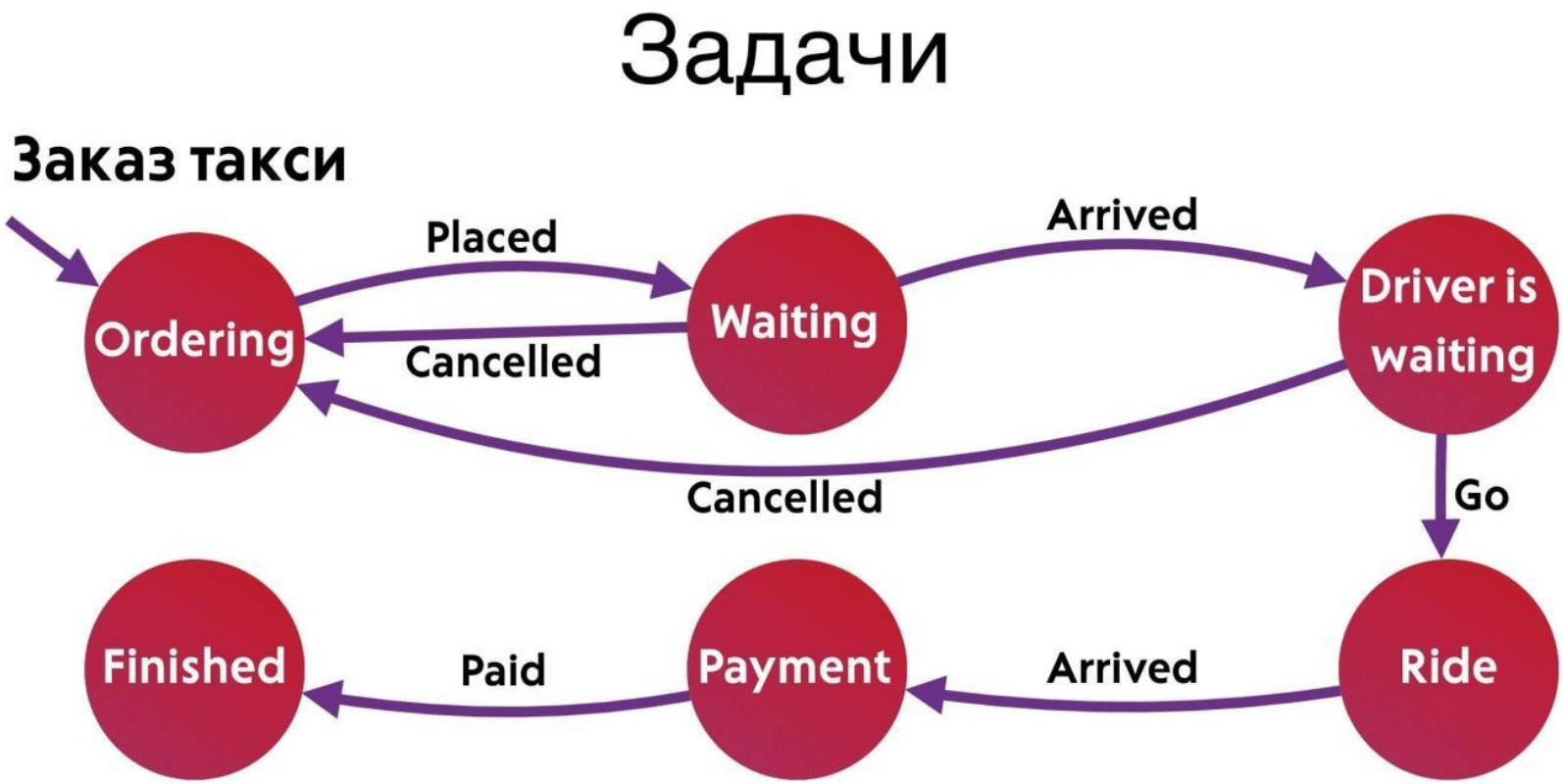
- Приложение для заказа такси
- Оформление заказа, авторизация
- Игровые приложения
- Анализ текстов
- Робототехника
- АНПА
- Приложения для планирования миссий
- И т.п.

Примеры. Заказ такси

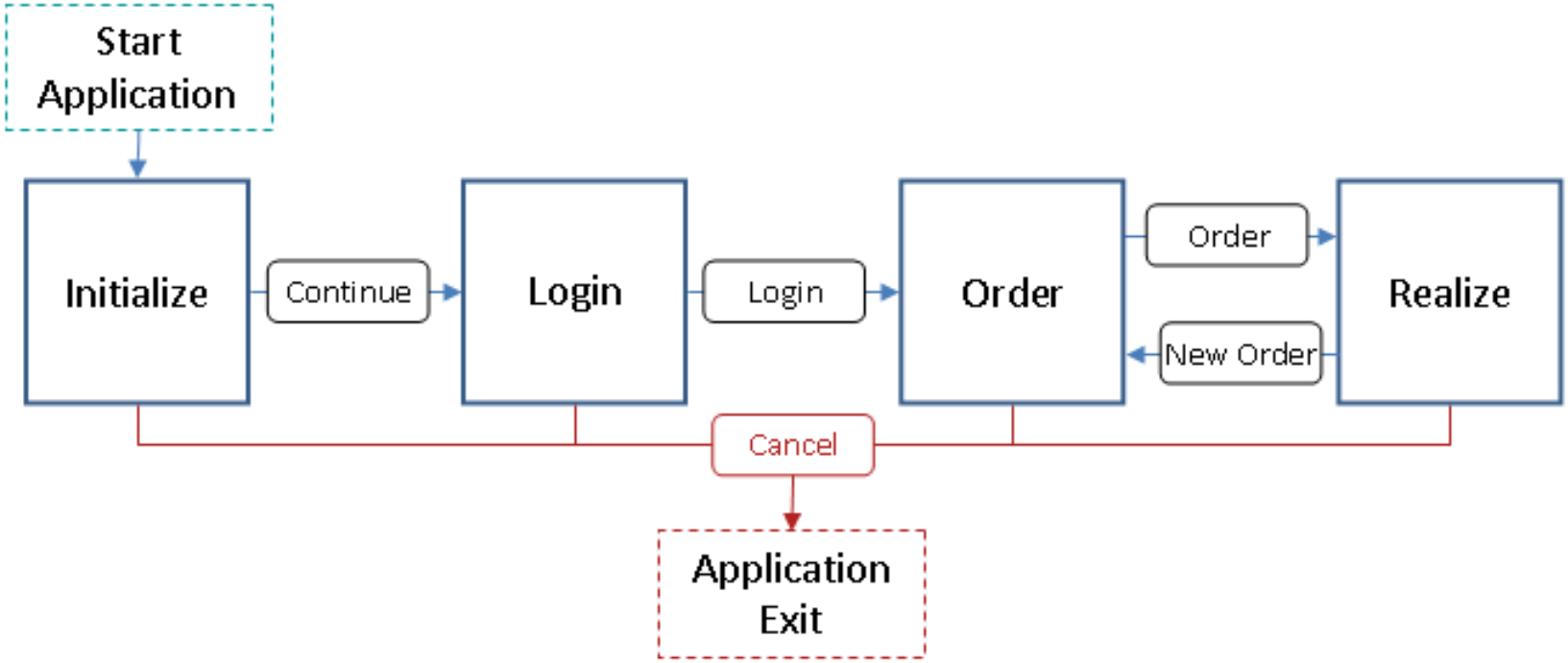




Примеры. Заказ такси



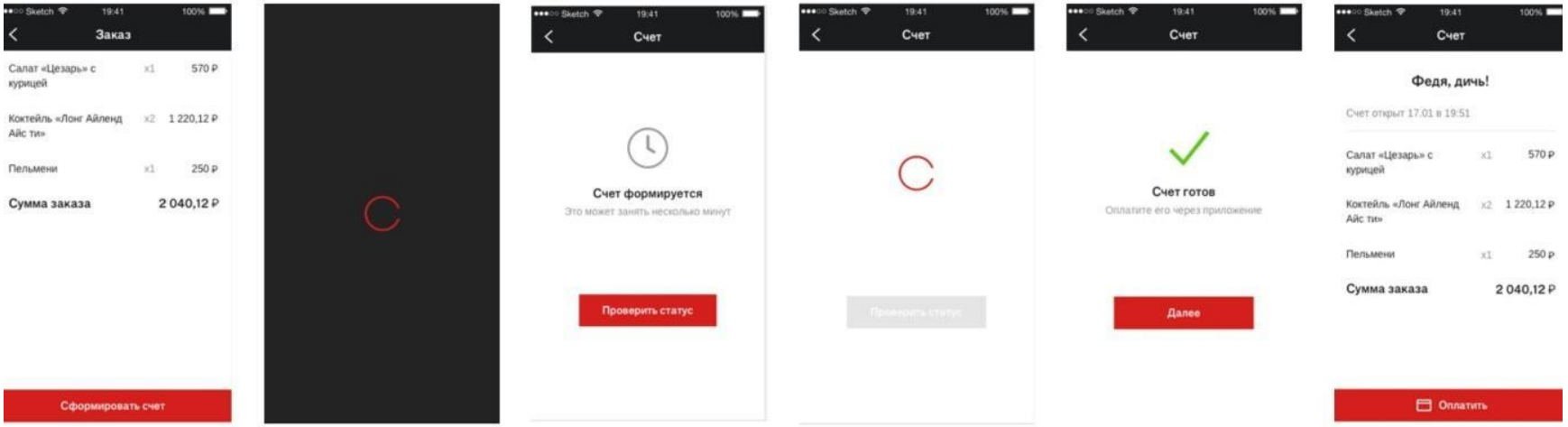
Примеры. Авторизация



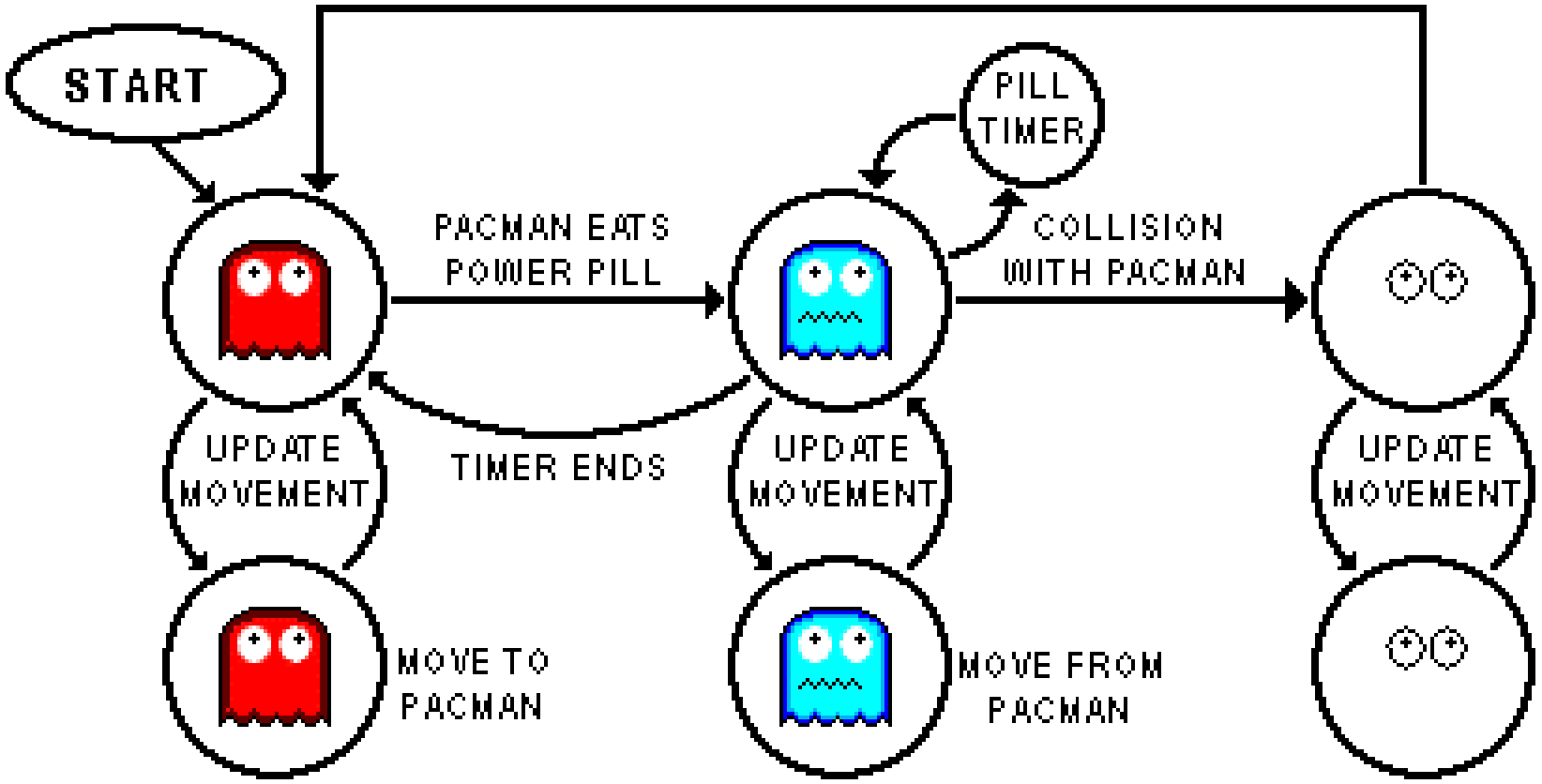
Примеры

Задачи

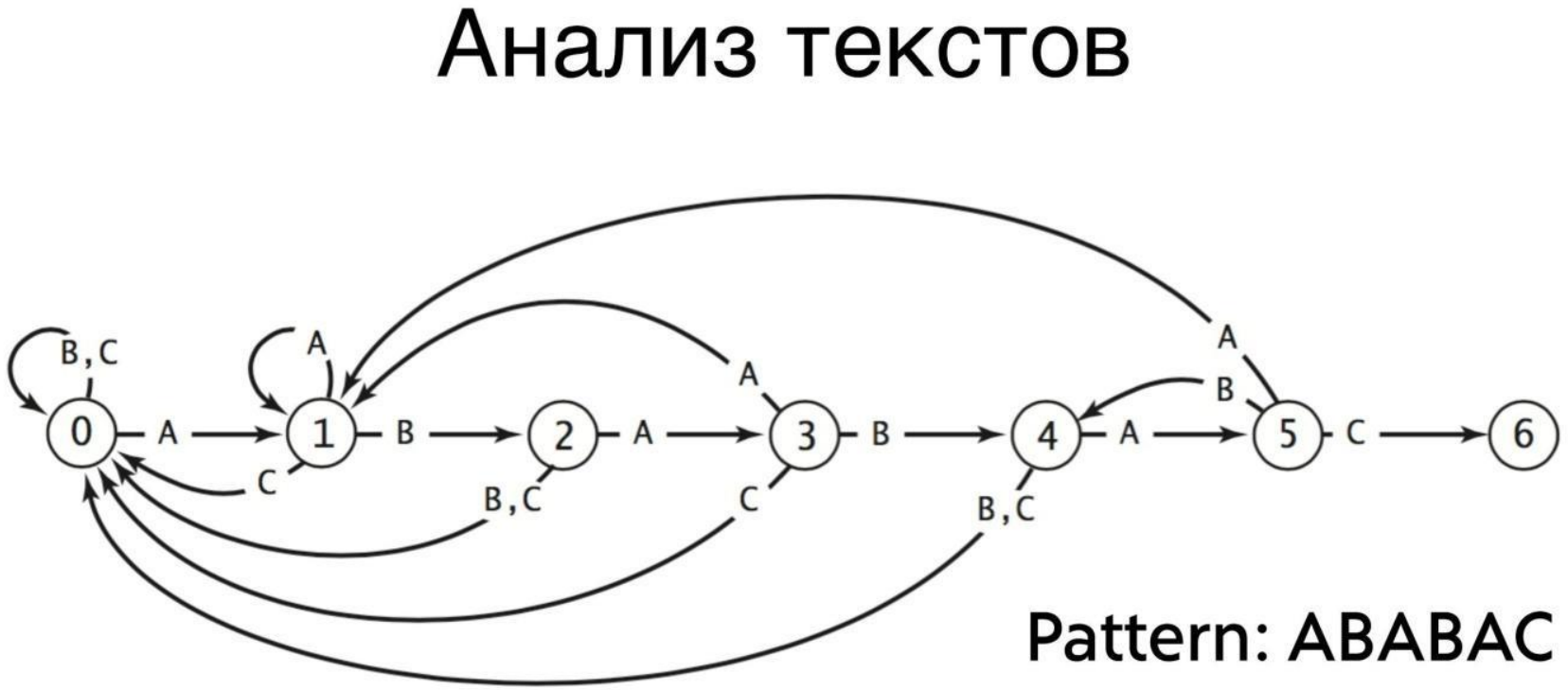
Оплата



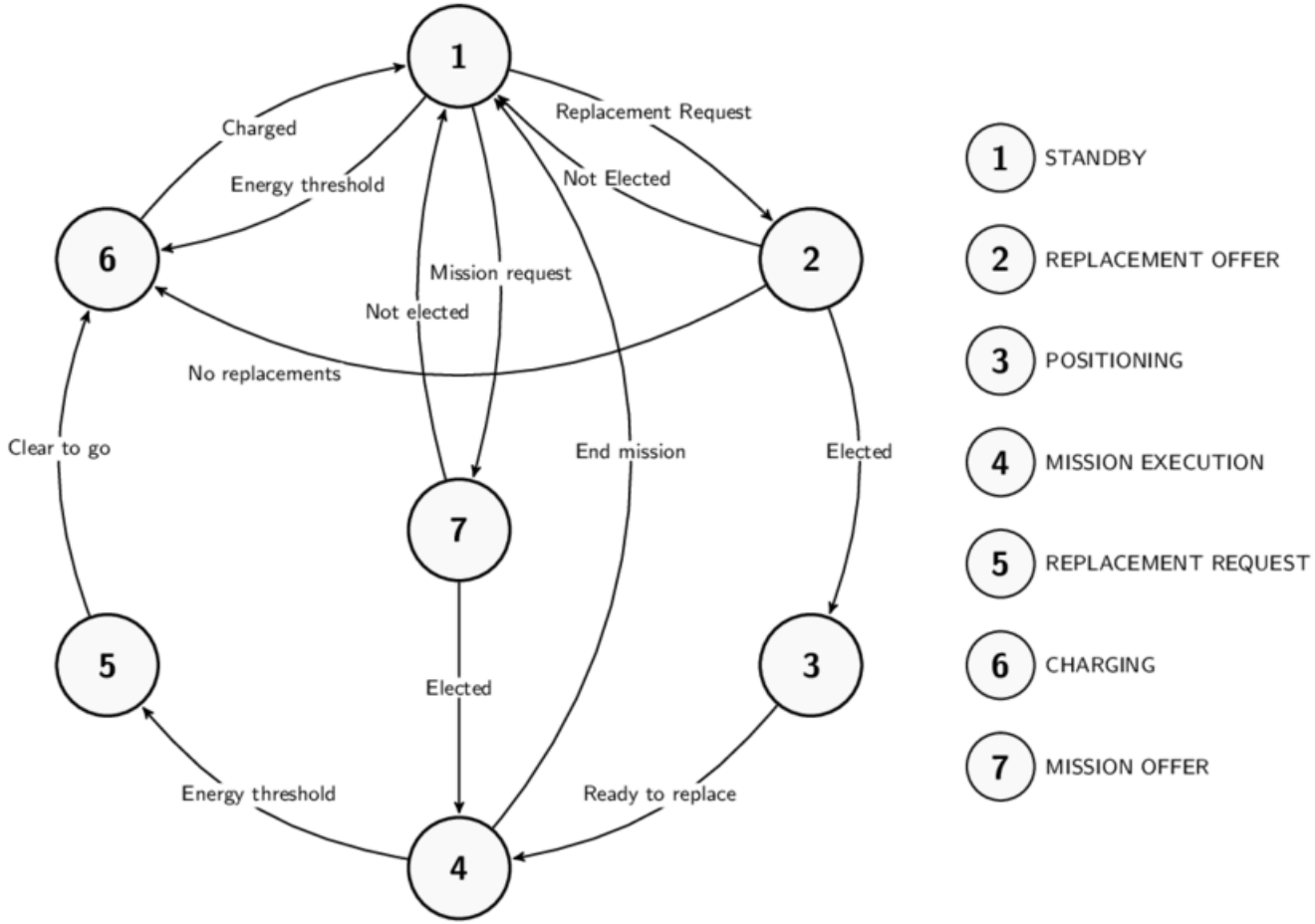
Пример. Игры


START
PILL TIMER
PACMAN EATS POWER PILL
COLLISION WITH PACMAN
UPDATE MOVEMENT
TIMER ENDS
UPDATE MOVEMENT
UPDATE MOVEMENT
MOVE TO PACMAN
MOVE FROM PACMAN

Пример. Анализ текстов



Пример. Робототехника



The State Machine Framework в Qt

Идеи конечного автомата реализованы с помощью мета-объектной системы фреймворка Qt.

У системы определяются:

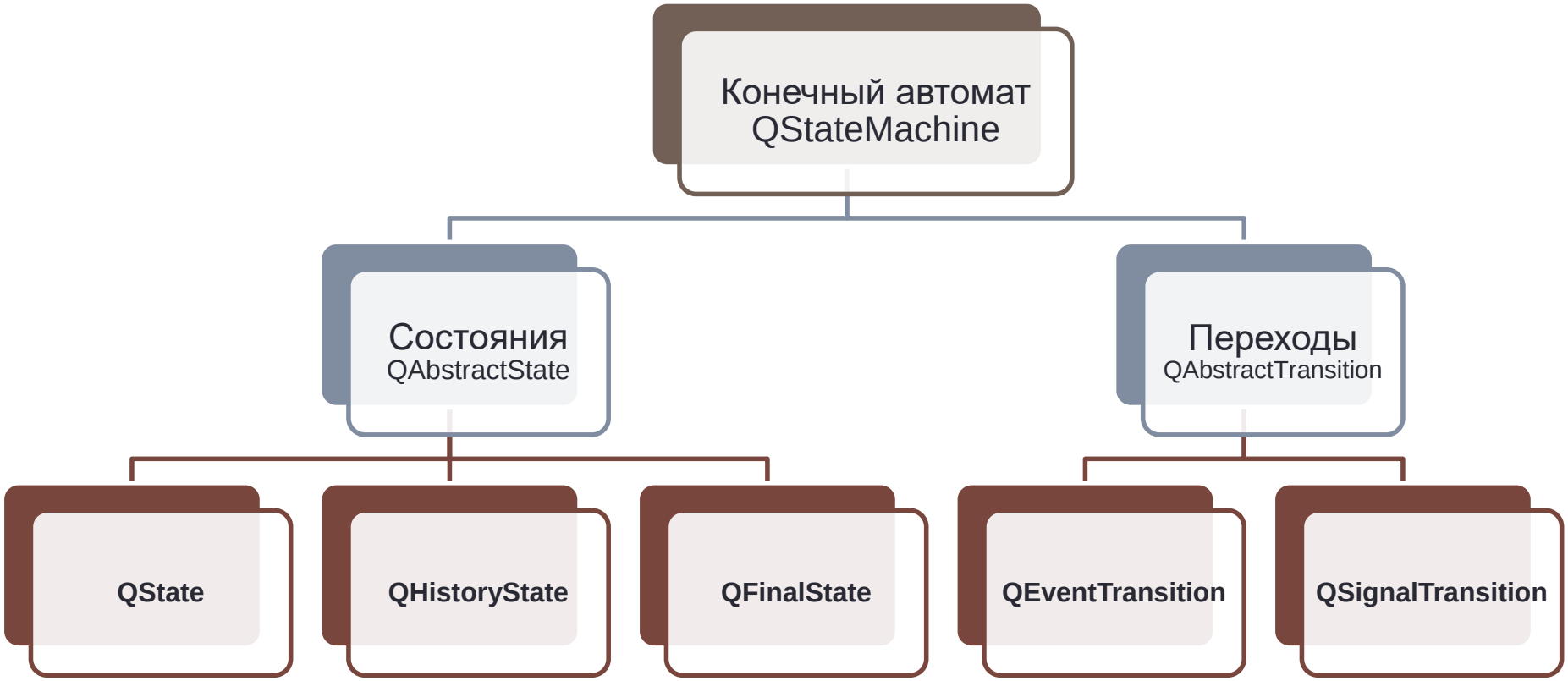
- возможные *состояния*, в которых она может быть;
- то как система может *переходить* из состояния в состояние (переходы реализованы за счет сигналов и слотов или механизма событий).

The State Machine Framework в Qt

Особенности:

- Qt моделирует работу иерархического конечного автомата;
- Состояния могут быть вложенными друг в друга;
- Конечный автомат в Qt работает в своём event loop-е

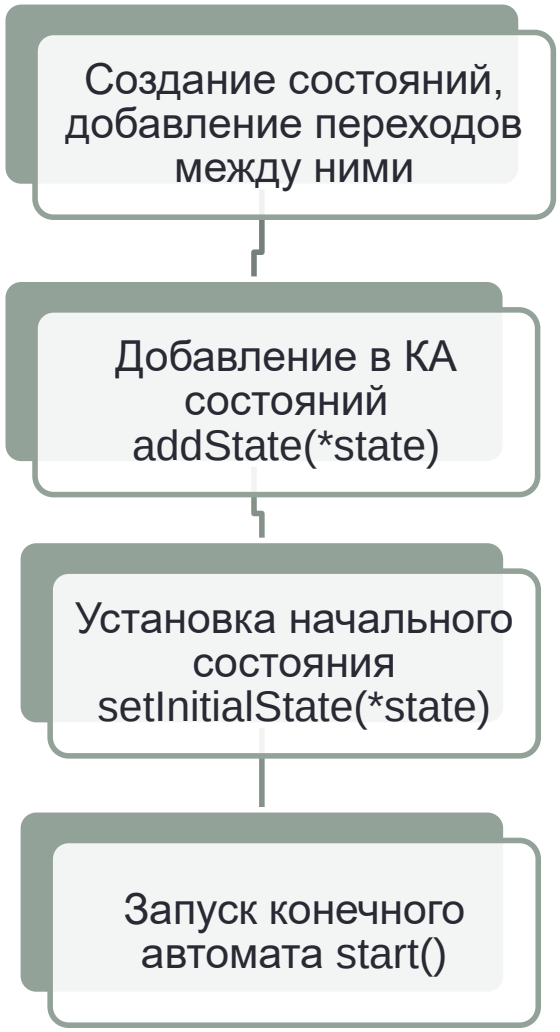
The State Machine Framework в Qt



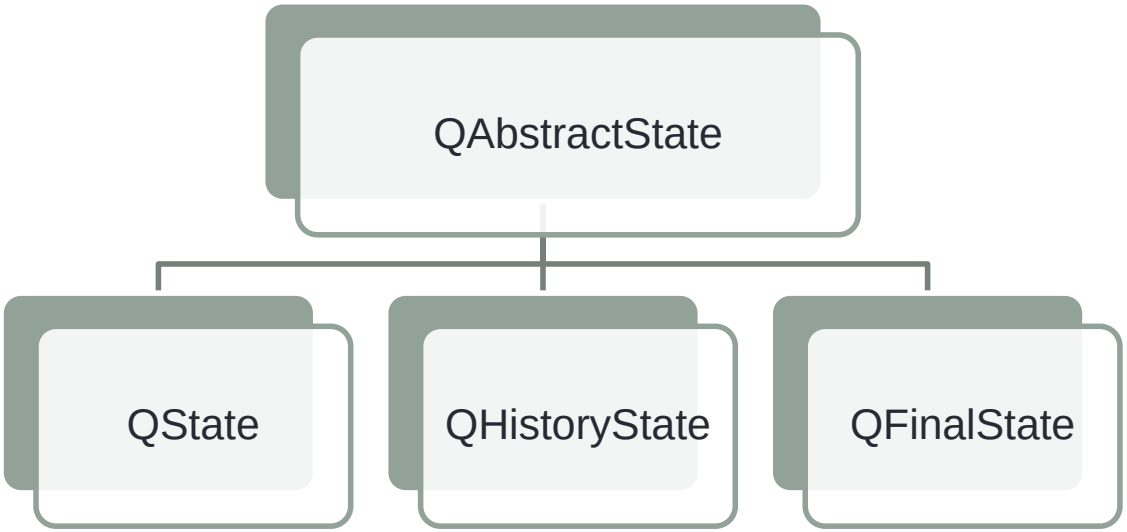
QStateMachine

- Конечный автомат работает асинхронно;
- Конечный автомат работает в своем event loop.
- QStateMachine – наследник QState, так что в состояние может быть «вложен» конечный автомат.

Алгоритм работы с конечным автоматом в Qt



Классы работы с состояниями



QState

- При группировании состояний необходимо указывать начальное состояние для группы;
- Чтобы сделать состояния вложенными друг в друга, при создании состояния ему в конструктор передается указатель на объект родителя.

QState – параллельные состояния

- При создании состояния указывается флаг (QState::ParallelStates)
- Когда параллельные события запускаются это происходит одновременно
- Переходы между состояния происходят в обычном порядке
- Любое из параллельных состояний может иметь переход, ведущий к завершению
- Все операции параллельных состояний происходят в общем шаге обработки событий

QHistoryState

QHistoryState - Псевдо состояние, которое запоминает в каком состоянии было родительское состояние до выхода из него.

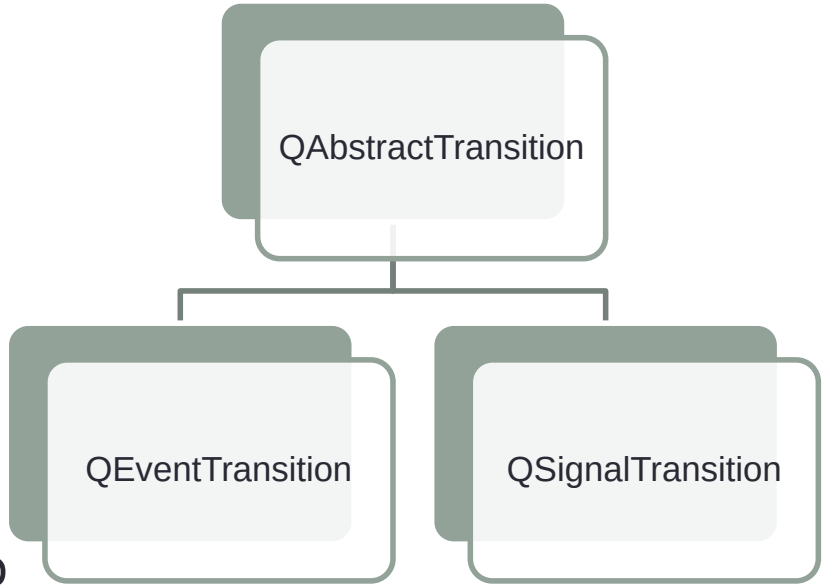
- При переходе в QHistoryState происходит автоматическое перенаправление в последнее состояние, в котором был родитель.

QFinalState

1. Если QFinalState – состояние верхнего уровня, то когда конечный автомат доходит до него, то он испускает сигнал finished() и завершает работу конечного автомата.
2. Если QFinalState – дочернее состояние, то оно завершает выполнение состояний родителя.

Переходы между состояниями

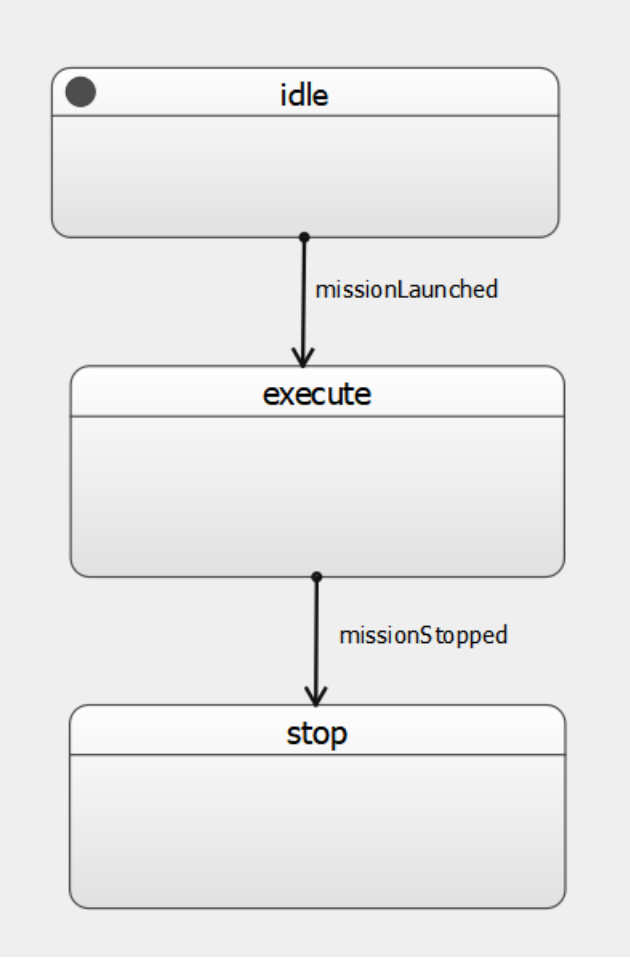
1. Переход из состояния в состояние не ограничен иерархией состояний;
2. Дочерние состояния наследуют переходы родительских состояний;
3. Дочернее состояние может перезаписать унаследованное состояние;
4. Для добавления перехода к состоянию используют метод addTransition(источник, сигнал, новое состояние)
5. Можно создавать переходы, которые не меняют текущее состояние системы.



Как связать всё это с кодом приложения?

- Соединение сигналов и слотов
- Изменение свойств объектов, при переходе из состояния в состояние

Практическая часть. Разработка конечного автомата для АНПА



Практическая часть 1

```
4   #include "ui_mainwindow.h"
5   #include <QObject>
6   #include <QStateMachine>
7   #include <QHistoryState>
8   #include <QFinalState>
9
10  class MainWindow : public QMainWindow, private Ui::MainWindow
11  {
12      Q_OBJECT
13      Q_PROPERTY(QString m_state READ state WRITE setState)
14  signals:
15      //сигнал о запуске миссии
16      void missionLaunched();
17      //сигнал об остановке миссии
18      void missionStopped();
19      //сигнал о продолжении выполнения миссии
20      void missionContinue();
21      //сигнал о том, что миссия выполнена
22      void missionDone();
```

Практическая часть 1

```
23  public:
24      explicit MainWindow(QWidget *parent = 0);
25      //метод, для чтения значения переменной состояния системы
26      QString state(){return m_state;}
27      //метод для записи нового значения перменной состояния системы
28      void setState(QString st){
29          m_state = st;
30          textBrowser->append(st);
31      }
32  private:
33      //переменная, в которой хранится текущее состояние системы
34      QString m_state;
35      //состояния конечного автомата
36      QState *idle, *execute, *stop;
37      //конечный автомат
38      QStateMachine stateMachine;
39
40  };
```

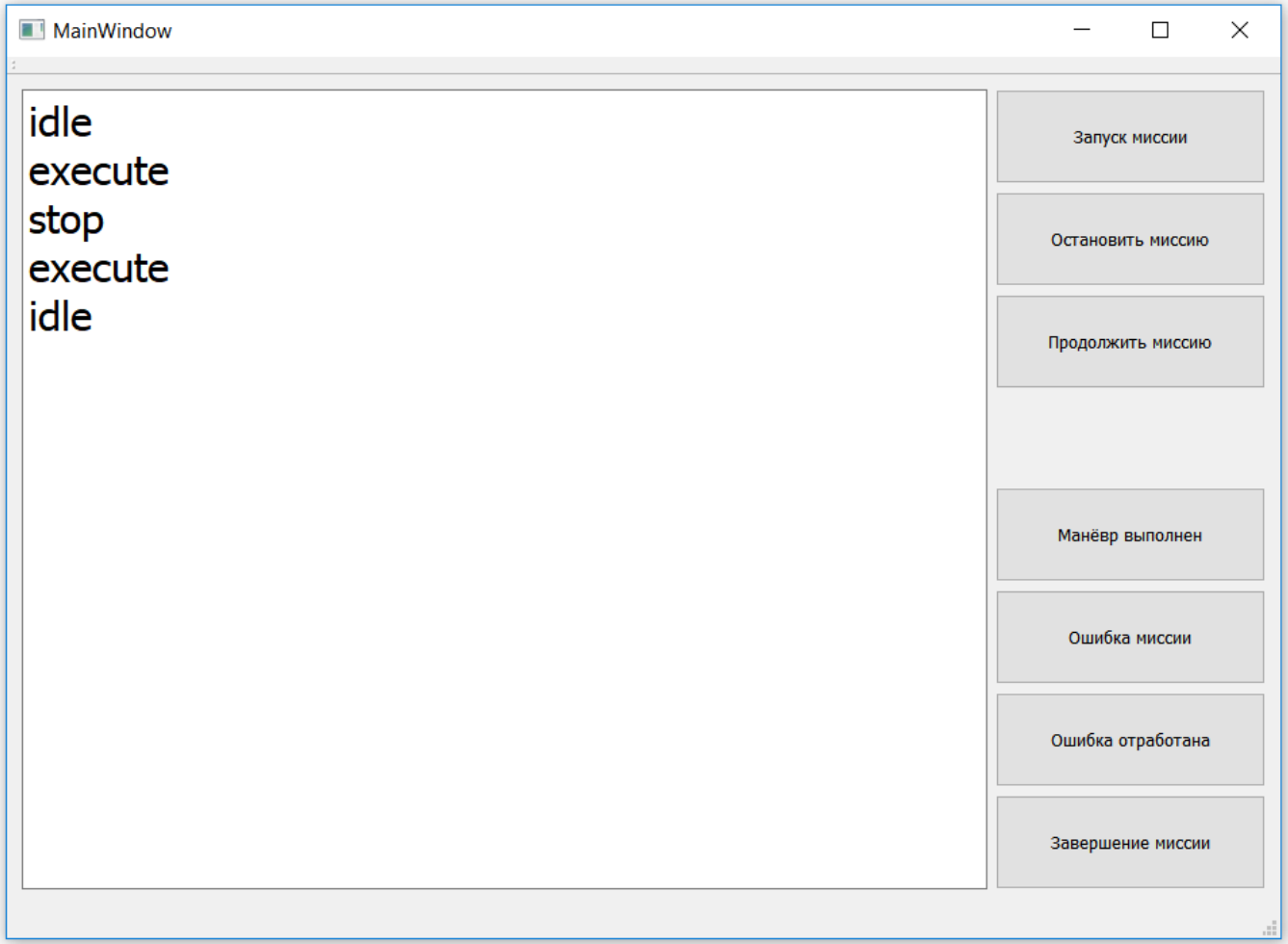
Практическая часть 1

```
#include "mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent)
{
    setupUi(this);

    connect(btnLaunch, SIGNAL(clicked()),SIGNAL(missionLaunched()));
    connect(btnStop, SIGNAL(clicked()),SIGNAL(missionStopped()));
    connect(btnContinue,SIGNAL(clicked()), SIGNAL(missionContinue()));
    connect(btnDone,SIGNAL(clicked()),SIGNAL(missionDone()));

    //1. создадим объекты состояний конечного автомата
    idle = new QState();
    execute = new QState();
    stop = new QState();

    //2. добавим переходы между состояниями с помощью метода
    //addTransition(ук-ль_на_источник_синала, сигнал, новое состояние)
    idle->addTransition(this,SIGNAL(missionLaunched()),execute);
    execute->addTransition(this,SIGNAL(missionStopped()),stop);
    execute->addTransition(this,SIGNAL(missionDone()),idle);
    stop->addTransition(this,SIGNAL(missionContinue()),execute);
```

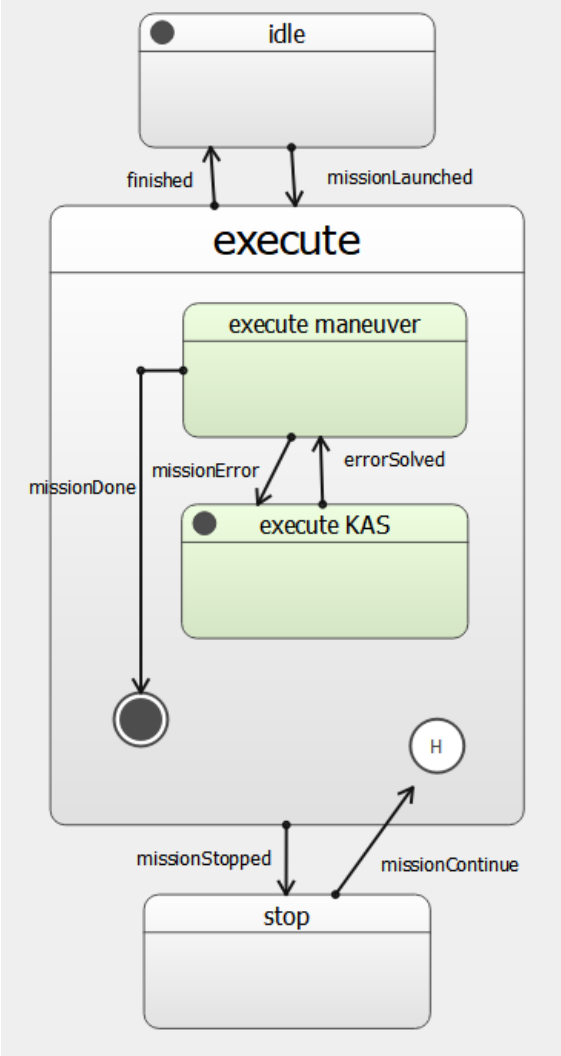
Практическая часть 1

```
24
25      //3. добавим изменение свойств объекта при изменении состояний
26      //assignProperty(1,2,3):
27          //1-я переменная: ук-ль_на_объект, свойства которого меняем
28          //2 - название перемнной, которую меняем
29          //3 - новое значение
30      idle->assignProperty(this,"m_state","idle");
31      execute->assignProperty(this,"m_state","execute");
32      stop->assignProperty(this,"m_state","stop");
33
34      //4. добавим состояния в конечный автомат
35      stateMachine.addState(idle);
36      stateMachine.addState(execute);
37      stateMachine.addState(stop);
38
39      //5.установим состояние по умолчанию
40      stateMachine.setInitialState(idle);
41
42      //6.запустим конечный автомат
43      stateMachine.start();
44  }
```

Практическая часть 1



Практическая часть 2. Дополнительный функционал



Практическая часть 2

```
9
10  class MainWindow : public QMainWindow, private Ui::MainWindow
11  {
12      Q_OBJECT
13      Q_PROPERTY(QString m_state READ state WRITE setState)
14  signals:
15      //сигнал о запуске миссии
16      void missionLaunched();
17      //сигнал об остановке миссии
18      void missionStopped();
19      //сигнал о продолжении выполнения миссии
20      void missionContinue();
21      //сигнал о том, что миссия выполнена
22      void missionDone();
23
24      //2. Добавим сигналы об ошибке вып-я миссии и исправлении ошибки
25      void missionError();
26      void missionErrorSolved();
```

Практическая часть 2

```
27  public:
28      explicit MainWindow(QWidget *parent = 0);
29      //метод, для чтения значения переменной состояния системы
30      QString state(){return m_state;}
31      //метод для записи нового значения перменной состояния системы
32      void setState(QString st){
33          m_state = st;
34          textBrowser->append(st);
35      }
36  private:
37      //переменная, в которой хранится текущее состояние системы
38      QString m_state;
39      //состояния конечного автомата
40      QState *idle, *execute, *stop;
41      //конечный автомат
42      QStateMachine stateMachine;
43      //2. вложенные состояния
44      QState *executeManeuver, *executeKAS;
45      //состояние, которое запоминает последнее состояние родителя
46      QHistoryState *hs;
47      //конечное состояние
48      QFinalState *missionDoneState;
49
50  };
```

Практическая часть 2

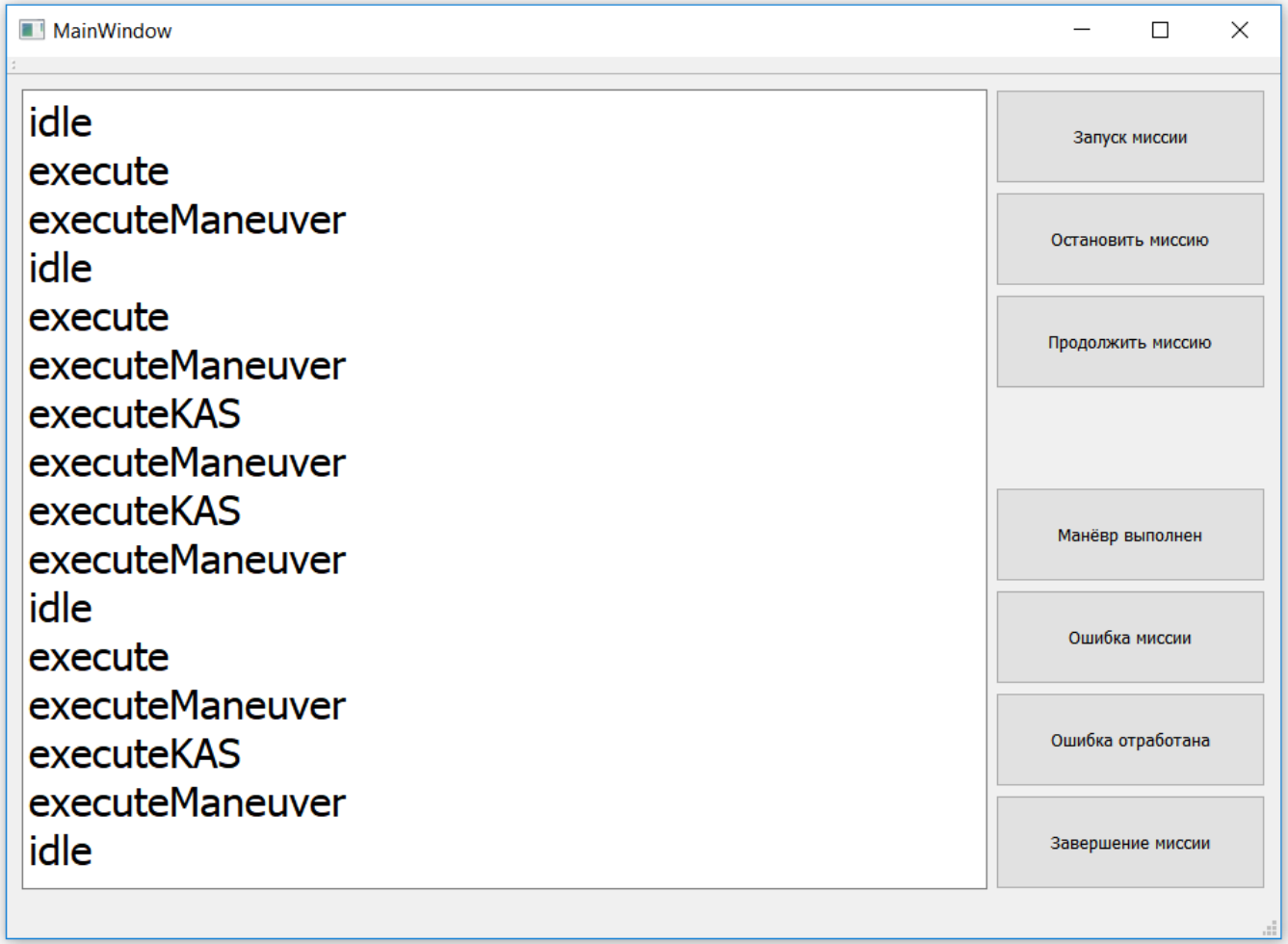
```
14
15      //1. создадим объекты состояний конечного автомата
16      idle = new QState();
17      execute = new QState();
18      stop = new QState();
19      //1.2 добавим дочерние состояния для execute
20      //для этого передадим в контсруктор указатель на родительское состояние:
21      executeManeuver = new QState(execute);
22      executeKAS = new QState (execute);
23      //добавим состояние, которое будет запоминать последне подсостояние для execute
24      hs = new QHistoryState (execute);
25      //для корректной настройки нужно указать, какое состояние будет для hs
26      //состоянием по умолчанию:
27      hs->setDefaultState(executeManeuver);
28      //добавим конечное состояние для execute
29      missionDoneState = new QFinalState (execute);
30      //для завершения настройки вложенных сосотяний необходимо какое состояние
31      //будет начальным при переходе в execute, выберем executeManeuver
32      execute->setInitialState(executeManeuver);
33
```

Практическая часть 2

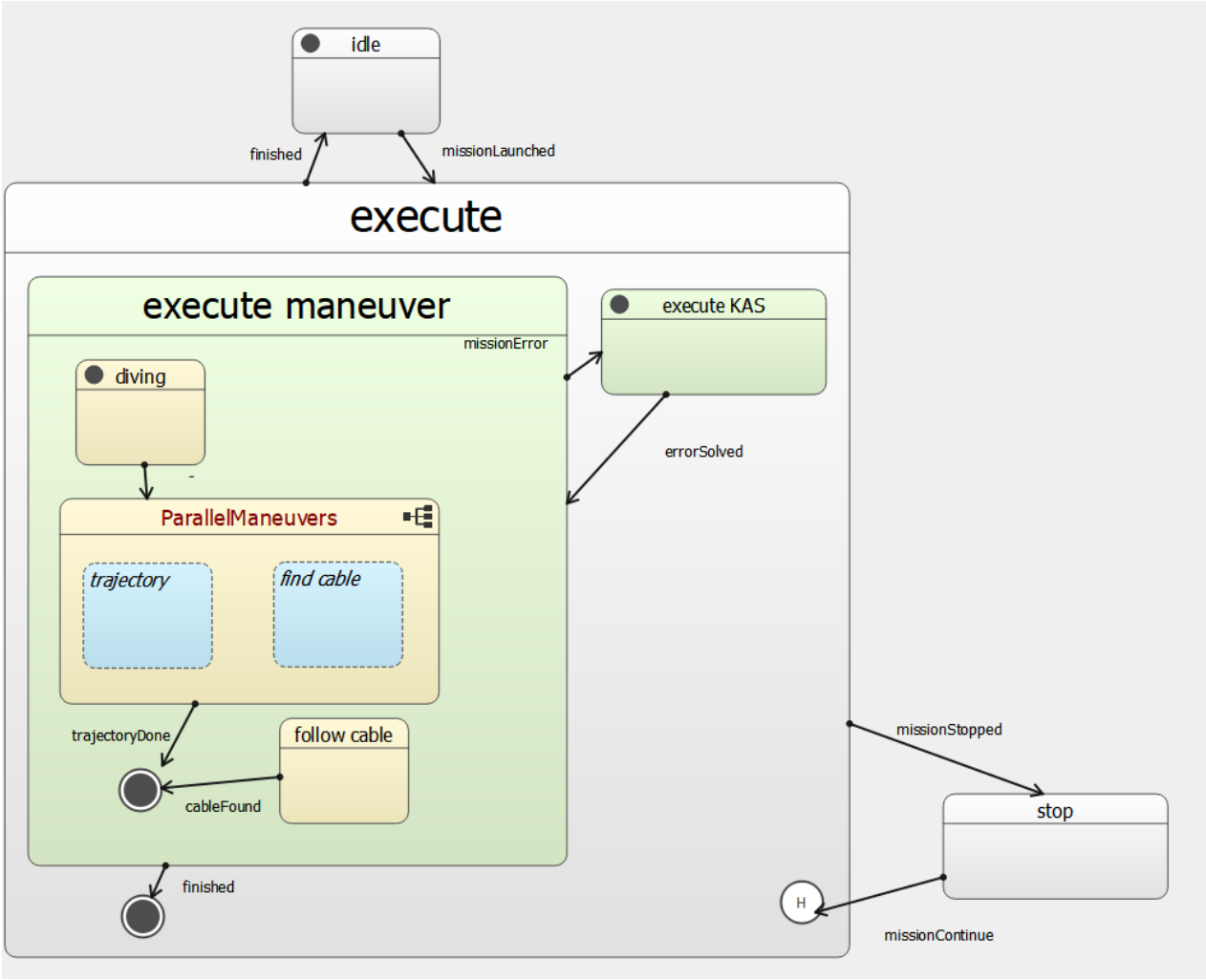
```
    //2. добавим переходы между состояниями с помощью метода
    //addTransition(ук-ль_на_источник_синала, сигнал, новое состояние)
    idle->addTransition(this,SIGNAL(missionLaunched()),execute);
    execute->addTransition(this,SIGNAL(missionStopped()),stop);
    //execute->addTransition(this,SIGNAL(missionDone()),idle);
    //2.2 теперь при переходе из стоп к продолжению миссии мы будем возвращаться к тому
    //состоянию, на котором были прерваны, для этого добавим переход к HistoryState:
    stop->addTransition(this,SIGNAL(missionContinue()),hs);
    executeManeuver->addTransition(this,SIGNAL(missionDone()),missionDoneState);
    executeManeuver->addTransition(this,SIGNAL(missionError()),executeKAS);
    executeKAS->addTransition(this,SIGNAL(missionErrorSolved()),executeManeuver);
    execute->addTransition(execute,SIGNAL(finished()),idle);
```

```
    //3.2
    executeManeuver->assignProperty(this,"m_state","executeManeuver");
    executeKAS->assignProperty(this,"m_state","executeKAS");
```

Практическая часть 2



Практическая часть 3.



Практическая часть 3

- Добавим новые сигналы

```
10  class MainWindow : public QMainWindow, private Ui::MainWindow
11  {
12      Q_OBJECT
13      Q_PROPERTY(QString m_state READ state WRITE setState)
14  signals:
15      //сигнал о запуске миссии
16      void missionLaunched();
17      //сигнал об остановке миссии
18      void missionStopped();
19      //сигнал о продолжении выполнения миссии
20      void missionContinue();
21      //сигнал о том, что миссия выполнена
22      void missionDone();
23
24      //2. Добавим сигналы об ошибке вып-я миссии и исправлении ошибки
25      void missionError();
26      void missionErrorSolved();
27
28      //3.
29      void cableFound();
30      void trajectoryDone();
31  public:
```

Практическая часть 3

- Добавим новые состояния миссии

```
40  private:
41      //переменная, в которой хранится текущее состояние системы
42      QString m_state;
43      //состояния конечного автомата
44      QState *idle, *execute, *stop;
45      //конечный автомат
46      QStateMachine stateMachine;
47      //2. вложенные состояния
48      QState *executeManeuver, *executeKAS;
49      //состояние, которое запоминает последнее состояние родителя
50      QHistoryState *hs;
51      //конечное состояние
52      QFinalState *missionDoneState;
53      //3. доп. состояния миссии
54      QState *diving, *trajectoryFollowing, *findCable, *followCable;
55      QState *parallelManeuvers;
56      QFinalState *doneExecute;
57
58  };
59
```

Практическая часть 3

```
37      //1.3 Выделим память под новые состояния
38      diving = new QState(executeManeuver);
39      //при создании параллельно-выполняемых состояний, в конструктор
40      //состоянию передается специальный флаг QState::ParallelStates
41      parallelManeuvers = new QState(QState::ParallelStates, executeManeuver);
42      trajectoryFollowing = new QState(parallelManeuvers);
43      findCable = new QState (parallelManeuvers);
44      followCable = new QState(executeManeuver);
45      doneExecute = new QFinalState(executeManeuver);
46      executeManeuver->setInitialState(diving);
```

```
62
63      //2.3 добавим переходы между новыми состояниями
64      //от погружения сразу переходим к составному манёвру
65      diving->addTransition(parallelManeuvers);
66      // если обнаруживаем кабель, то начинаем следовать вдоль него
67      findCable->addTransition(this,SIGNAL(cableFound()),followCable);
68      //если завершили задачу следования вдоль кабеля, то можем сказать, что миссия завершена
69      followCable->addTransition(this,SIGNAL(cableFound()),doneExecute);
70      //если кабель нам так и не встретился, то миссию можем считать законченной, если завершим
71      //следование траектории в заданном районе
72      trajectoryFollowing->addTransition(this,SIGNAL(trajectoryDone()),doneExecute);
73
```

Практическая часть 3

```
86      //3.3 добавим изменение свойств при переходе в новые добавленные состояния
87      diving->assignProperty(this,"m_state","diving");
88      parallelManeuvers->assignProperty(this,"m_state","parallelManeuvers");
89      trajectoryFollowing->assignProperty(this,"m_state","trajectoryFollowing");
90      findCable->assignProperty(this,"m_state","findCable");
91      followCable->assignProperty(this,"m_state","followCable");
```

```
  //если хотите, чтобы после остановки миссии конечный автомат возвращался к
    //манёвру, на котором был прерван, то для этого установите флаг QHistoryState::DeepHistory
    hs->setHistoryType(QHistoryState::DeepHistory);
```

Практическая часть 3