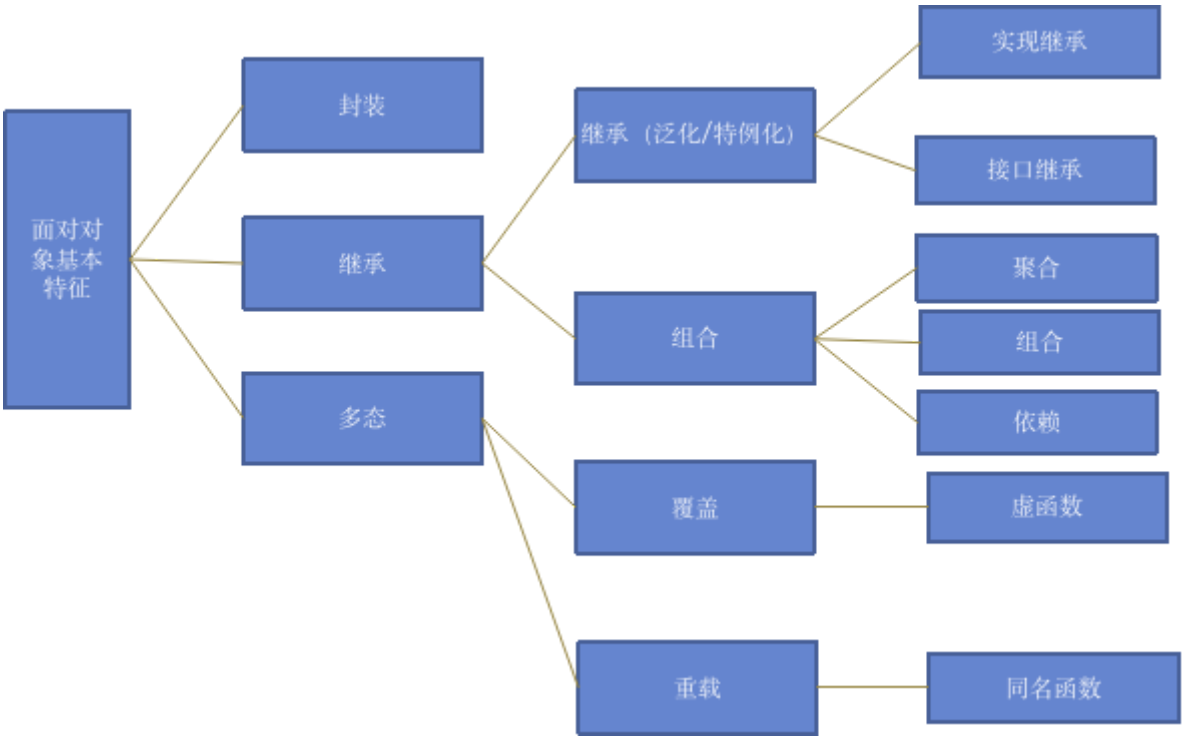
How to design the program in OOP

1. what 's the conception about OOP - 什么是面对对象设计

1.1. what's the Class and Object - 什么是类和对象

> 类和对象（class）是两种以计算机为载体的计算机语言的合称。对象是对客观事物的抽象，类是对对象的抽象。类是一种抽象的数据类型。它们的关系是，对象是类的实例，类是对象的模板。对象是通过new className产生的，用来调用类的方法。

1.2. what's there factors in OOP - 面对对象设计的基本特点

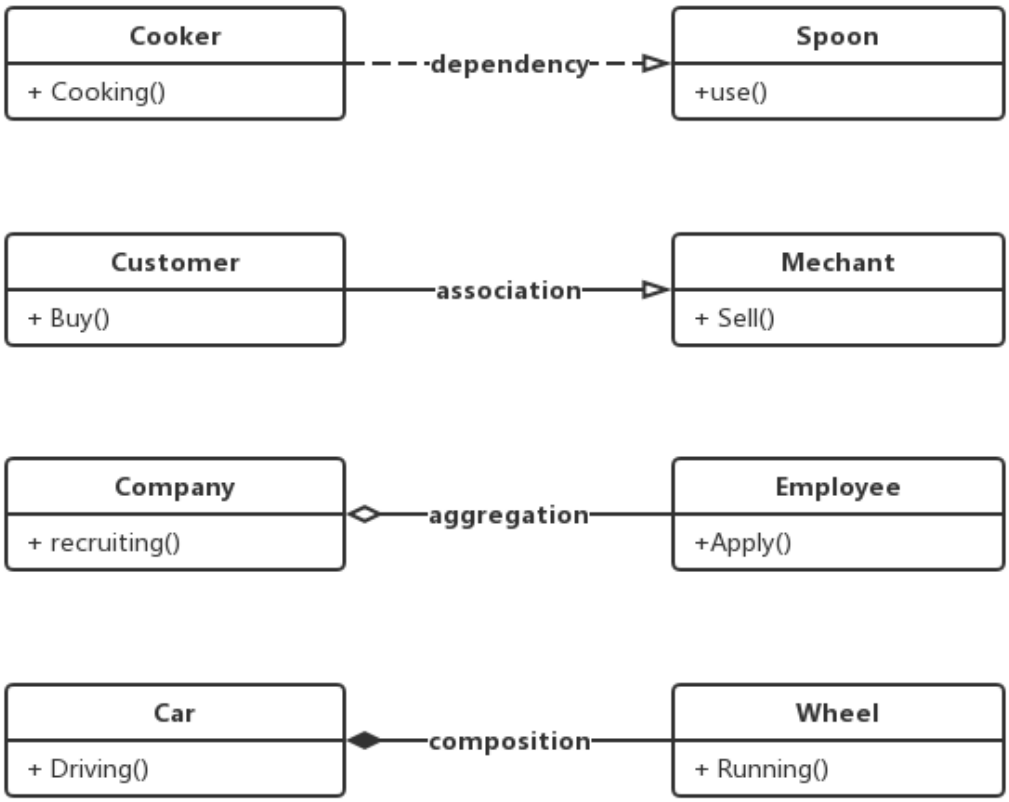


- 类之间关系: 依赖-关联-聚合-组合

  这四种关系，都表示类与类之间的关联关系，说明两个类之间有某种联系。而依据与联系的紧密程度，由弱到强，依次成为 依赖<关联<聚合<组合。

  - 依赖(dependency) 如果一个类，偶尔用到某个类。例如,厨师需要Cooking()，那么他需要依赖Spoon这个类，调用它的 use().
  - 关联(association) 比依赖关系稍微强些。例如，Customer需要与Merchant进行沟通，此时Customer与Merchant就是关联关系。通常关联关系中的两类，相对平等，都可以独立存在维护。
  - 聚合(aggregation) 比关联更强些，通常是整体与部分之间的关系，例如 公司(Company)和员工(Employee)。此时的整体和部分，都可以独立存在。

- 组合(composition)又叫合成 比聚合更强的整体与部分的关系，通常整体和部分都不能独立存在，必须要组合到一起存在，例如车(Car)和轮子(wheel)。图例中使用实心菱形+实线+箭头（箭头可选）表示.



- 覆盖-重载
  - 重写(覆盖)：是指派生类中存在重新定义的函数。其函数名，参数列表，返回值类型，所有都必须同基类中被重写的函数一致。只有函数体不同（花括号内），派生类调用时会调用派生类的重写函数，不会调用被重写函数。重写的基类中被重写的函数必须有virtual修饰。（C++）

```
    #include<iostream>

    using namespace std;

    class Base
    {
    public:
        virtual void fun(int i){ cout << "Base::fun(int) : " << i
<< endl;}
    };

    class Derived : public Base
    {
    public:
```

```
        virtual void fun(int i){ cout << "Derived::fun(int) : "
<< i << endl;}
    };
    int main()
    {
        Base b;
        Base * pb = new Derived();
        pb->fun(3);//Derived::fun(int)

        system("pause");
        return 0;
    }
```

- 重载：是指同一可访问区内被声明的几个具有不同参数列（参数的类型，个数，顺序不同）的同名函数，根据参数列表确定调用哪个函数，重载不关心函数返回类型。

```
    class A{
    public:
    void test(int i);
    void test(double i);//overload
    void test(int i, double j);//overload
    void test(double i, int j);//overload
    int test(int i);         //错误，非重载。注意重载不关心函数返回类
型。
    };
```

1.3. introduce the principles for software design - 简述软件

- 设计上的基本原则
    - a. 单一职责原则(Single Responsibility Principle - SRP)
    - b. 里氏替换原则(Liskov Substitution Priciple - LSP)
    - c. 依赖倒置原则(Dependence Inversion Principle, DIP)
    - d. 接口隔离原则(interface Segregation Principle,ISP)
    - e. 迪米特法则(Law of Demeter, LoD)
    - f. 开放闭合原则(Open Close Principle, OCP)

a. 单一职责原则(Single Responsibility Principle - SRP)

- 核心思想：应该有且仅有一个原因引起类的变更；
- 问题描述：假如有类CLASS A负责职责T1,T2,当职责T1或T2有变更需要修改时,有可能影响到该类的另外一个职责正常工作;
- 好处: 类的复杂程度降低、可读性提高、可维护性提高、降低了变更引发的风险；
- 注意：单一职责原则提出了一个编写程序的标准，用“职责”或“变更原因”来衡量接口或类设计得是否优良，但是“职责”和“变化原因”都是不可度量的，因项目和环境而异。

b. 里氏替换原则(Liskov Substitution Priciple - LSP)

- 核心思想：在使用积累的地方可以任意使用其子类，能保证子类完美替换基类。
- 问题描述：只要父类能出现的地方，子类就能出现。反之，父类则未必能胜任;
- 好处：增强程序的健壮性，即使增加子类，原有的子类还可以继续运行;
- 注意：如果子类不能完整地实现父类方法，或者父类的某些方法在子类中已经发生“变化”，则建议断开父子继承关系采用依赖、聚合、组合等关系代替继承。

c. 依赖倒置原则(Dependence Inversion Principle, DIP)

- 核心思想：高层模块不应该依赖底层模块，二者都该依赖其抽象；抽象不应该依赖细节；细节应该依赖抽象；
    - 说明：高层模块就是调用端，底层模块就是具体实现类。抽象就是指接口或抽象类。细节就是现实类。依赖倒置原则的本质就是通过抽象(接口或抽象类)使各个类或模块的实现彼此独立，互不影响，实现模块间的松耦合。
- 问题描述：类A直接依赖类B，加入要将类 A改为依赖类C，则必须通过修改类A 的代码来达成。这种场景下，类A一般是高层模块，负责复杂的业务逻辑；类B和类C是低层模块，负责基本的原子操作；所以如果修改类A，会给程序带来不必要的风险。
- 解决方案: 将类A修改为依赖接口(interface),类B和类C各自实现接口(Interface)，类A通过接口(interface) 间接与类B或者类C发生联系，则会大大降低修改类A的几率。
- 好处: 依赖倒置的好处在小型项目中很难体现出来，但是在大中型项目中可以减少需求变化引起的工作量，使并行开发更友好。

d. 接口隔离原则(interface Segregation Principle,ISP)

- 核心思想：类间依赖关系应该建立在最小的接口上 通俗来讲，建立单一接口，不要建立庞大臃肿的接口。尽量细化接口，接口中的方法尽量少。也就是说，我们要为各个类建立专用的接口，而不要试图去建立一个很庞大的接口供所有依赖它的类去调用。
- 问题描述：class A通过接口(interface)依赖B，class C 通过接口(interface)依赖class D,如果接口(interface)对于class A和 class B来说不是最小接口，则class B和class D 必须去实现它们不需要的方法。

e. 迪米特法则(Law of Demeter, LoD)

- 核心思想：类间解耦。通俗来说，一个类对自己依赖的类知道的越少越好-实现低耦合的关键。

f. 开放闭合原则(Open Close Principle, OCP)

- 核心思想：尽量通过扩展软件实体来解决需求变化，而不是通过修改已有的代码来完成变化。 软件实体应该是可扩展，而不可修改的。也就是说，对扩展是开放的，而对修改是封闭的。 因此，开放封闭原则主要体现在两个方面： 对扩展开放，意味着有新的需求或变化时，可以对现有代码进行扩展，以适应新的情况。 对修改封闭，意味着类一旦设计完成，就可以独立完成其工作，而不要对类进行任何修改。
- 解决方案:

1. 开放封闭原则，是最为重要的设计原则，Liskov替换原则和合成/聚合复用原则为开放封闭原则的实现提供保证。
2. 可以通过Template Method模式和Strategy模式进行重构，实现对修改封闭、对扩展开放的设计思路。

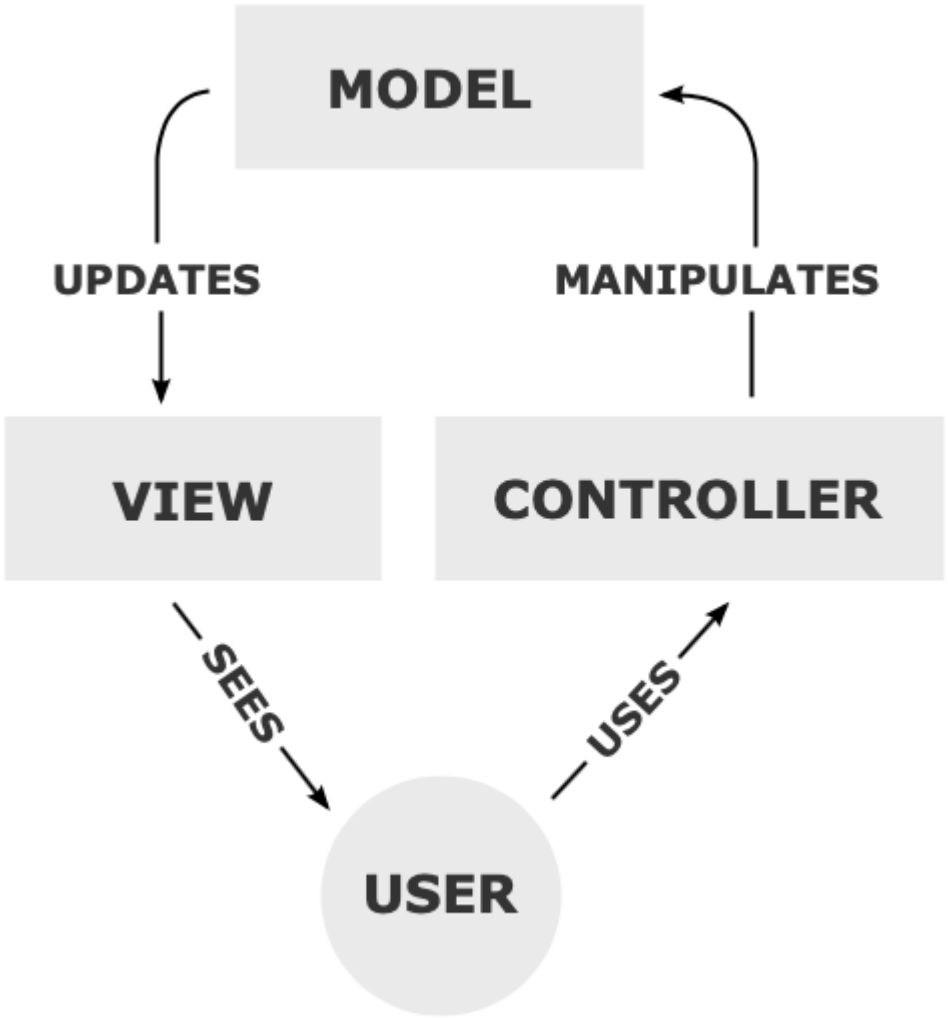
---

2. introduction the common design patterns and its pratices - 简介常见的软件设计模型及其实现方法

2.1 the software architecture Pattern-软件架构模型

- 软件架构模型，是一种结构型的编程模板。用来描述软件模块间的作用关系以及层级关系等。常见的架构模型除了MVC,MVP,MVVM,还有:publish-describe,client-server等。

2.1.1. Model-View-Controller:MVC



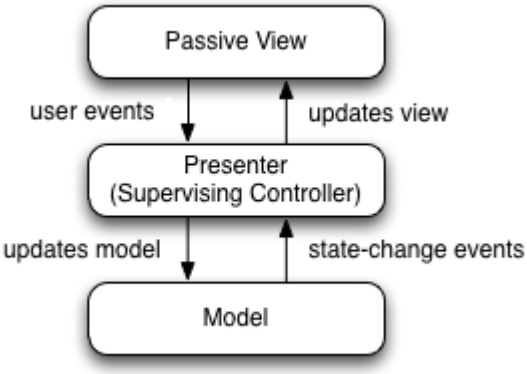
- 将应用程序划分为三种组件，模型 - 视图 - 控制器（MVC）设计定义它们之间的相互作用。
    - 模型（Model） 用于封装与应用程序的业务逻辑相关的数据以及对数据的处理方法。“ Model ”有对数据直接访问的权力，例如对数据库的访问。“Model”不依赖“View”和“Controller”，也就是说，Model 不关心它会被如何显示或是如何被操作。但是 Model 中数据的变化一般会通过一种刷新机

制被公布。为了实现这种机制，那些用于监视此 Model 的 View 必须事先在此 Model 上注册，从而，View 可以了解在数据 Model 上发生的改变。（比如：观察者模式（软件设计模式））

- 视图（View）能够实现数据有目的的显示（理论上，这不是必需的）。在 View 中一般没有程序上的逻辑。为了实现 View 上的刷新功能，View 需要访问它监视的数据模型（Model），因此应该事先在被它监视的数据那里注册。
- 控制器（Controller）起到不同层面间的组织作用，用于控制应用程序的流程。它处理事件并作出响应。“事件”包括用户的行为和数据 Model 上的改变。

- Qt中的MVC: TODO

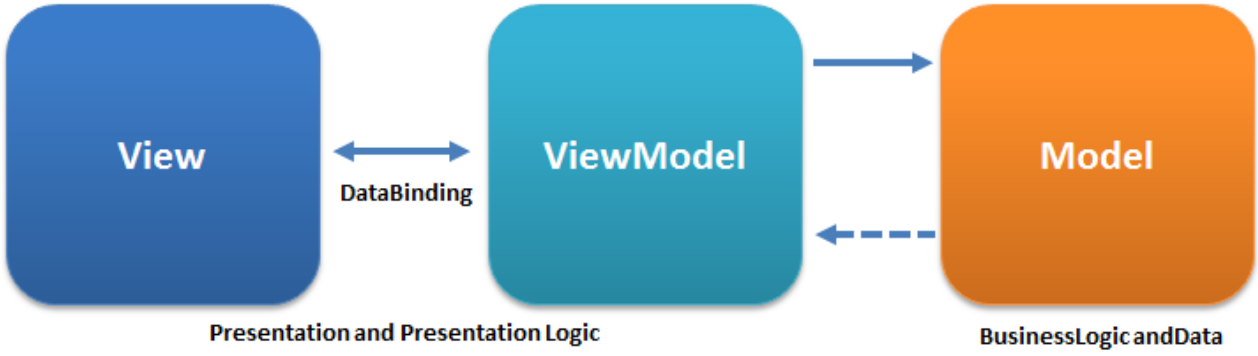
2.1.2. Model-View-Presenter:MVP



Model-View-Presenter (MVP) 是用户界面设计模式的一种，被广泛用于便捷自动化单元测试和在呈现逻辑中改良分离关注点（separation of concerns).

- Model 定义用户界面所需要被显示的数据模型，一个模型包含着相关的业务逻辑。
- View 视图为呈现用户界面的终端，用以表现来自 Model 的数据，和用户命令路由再经过 Presenter 对事件处理后的数据。
- Presenter 包含着组件的事件处理，负责检索 Model 获取数据，和将获取的数据经过格式转换与 View 进行沟通。MVP 设计模式通常会再加上 Controller 做为整体应用程序的后端程序工作。

2.1.3. Model-View-ViewModel:MVVM



- Model(模型)

  模型是指代表真实状态内容的领域模型（面向对象），或指代表内容的数据访问层(以数据为中心)。

- View(视图)

  就像在MVC和MVP模式中一样，视图是用户在屏幕上看到的结构、布局和外观(UI).

- ViewModel(视图模型)

  视图模型是暴露公共属性和命令的视图的抽象。MVVM没有MVC模式的控制器，也没有MVP模式的presenter，有的是一个绑定器。在视图模型中，绑定器在视图和数据绑定器之间进行通信。

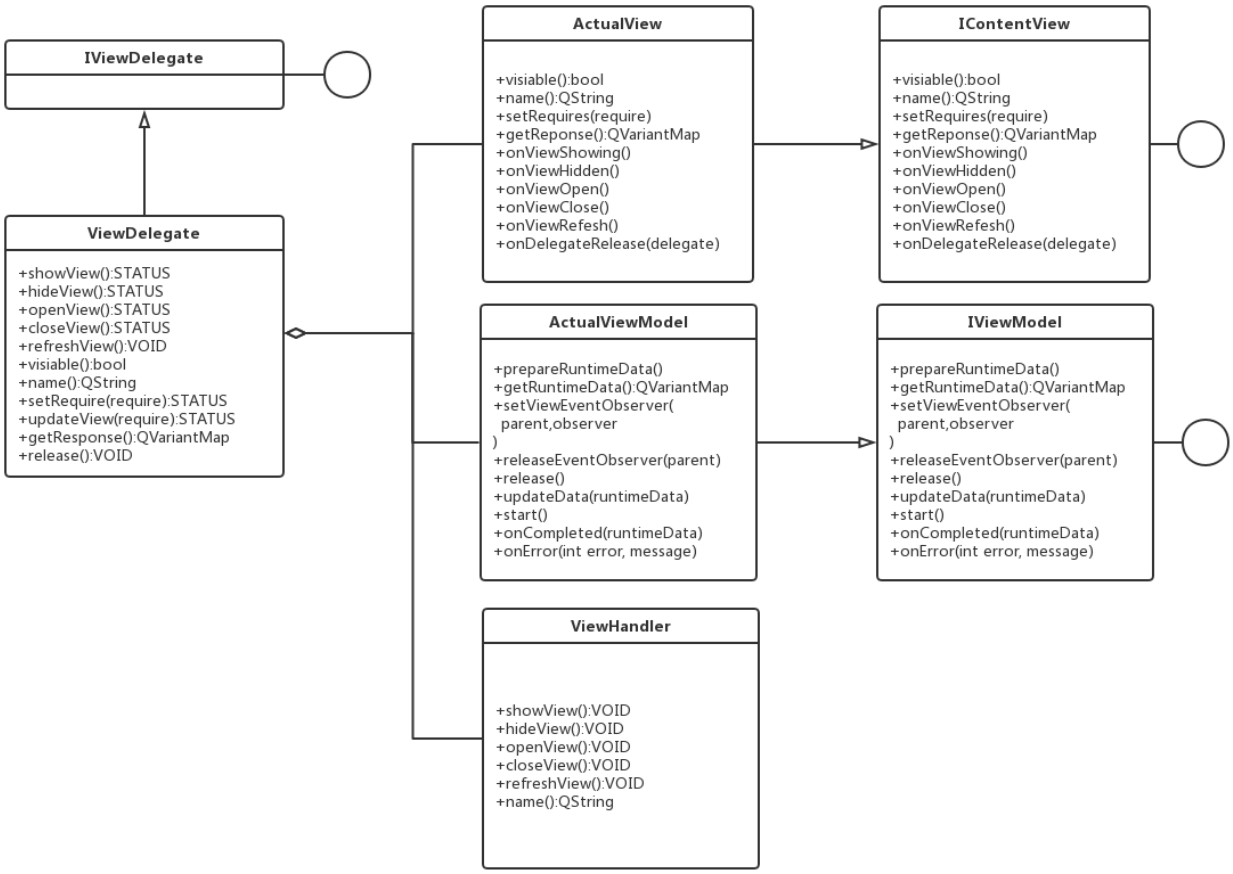
- Binder(绑定器)

  声明性数据和命令绑定隐含在MVVM模式中。

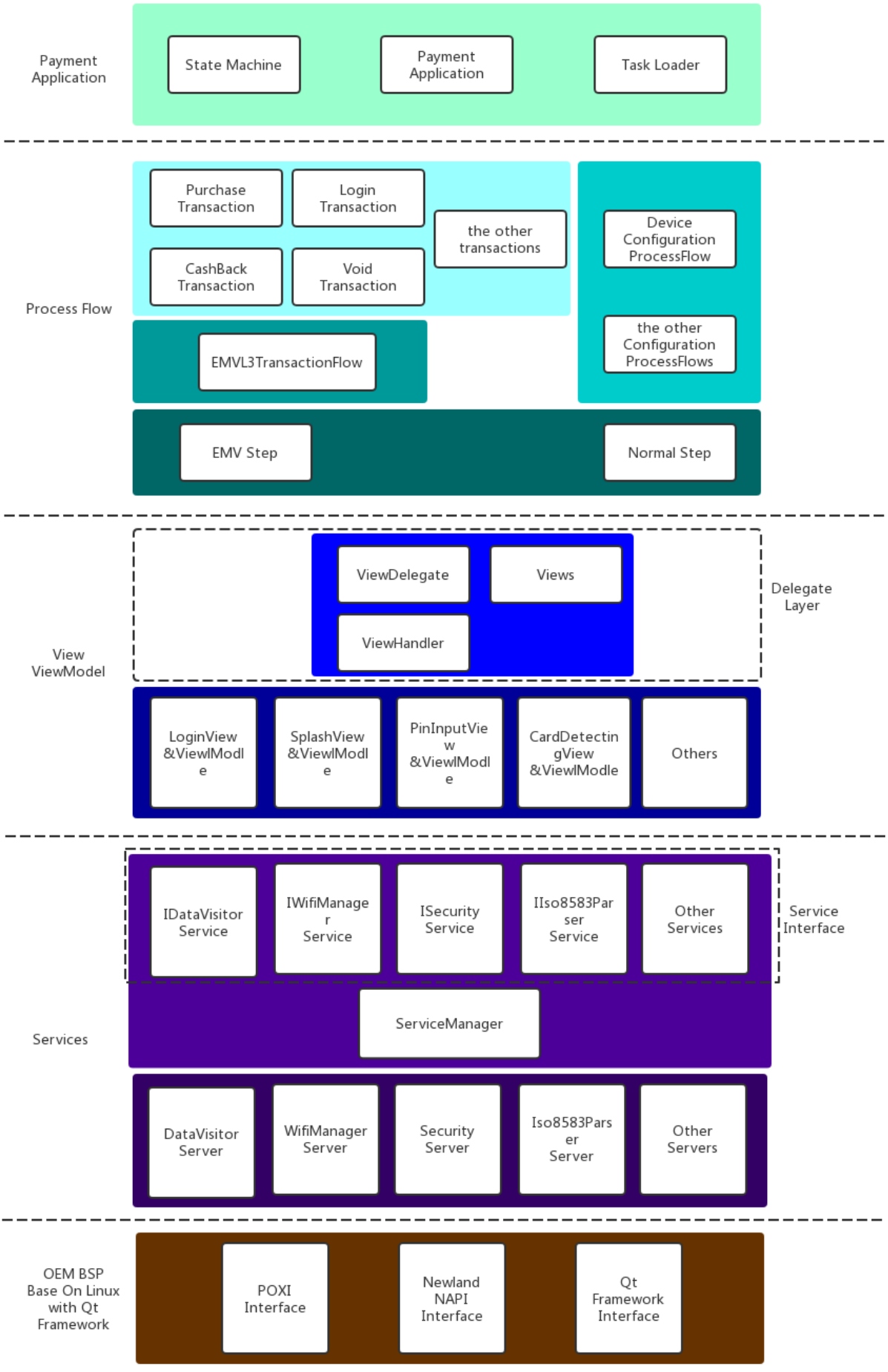
- BANKDEMO中的MVVM结构:

class name	**role**
ViewDelegete	Binder
ActualView	View
DataVisitService	Model
ActualViewModel	ViewModel

- The Inheritance of View and ViewModel, View Delegate.



2.1.4. The Architecture of Qt Bankdemo Diagram

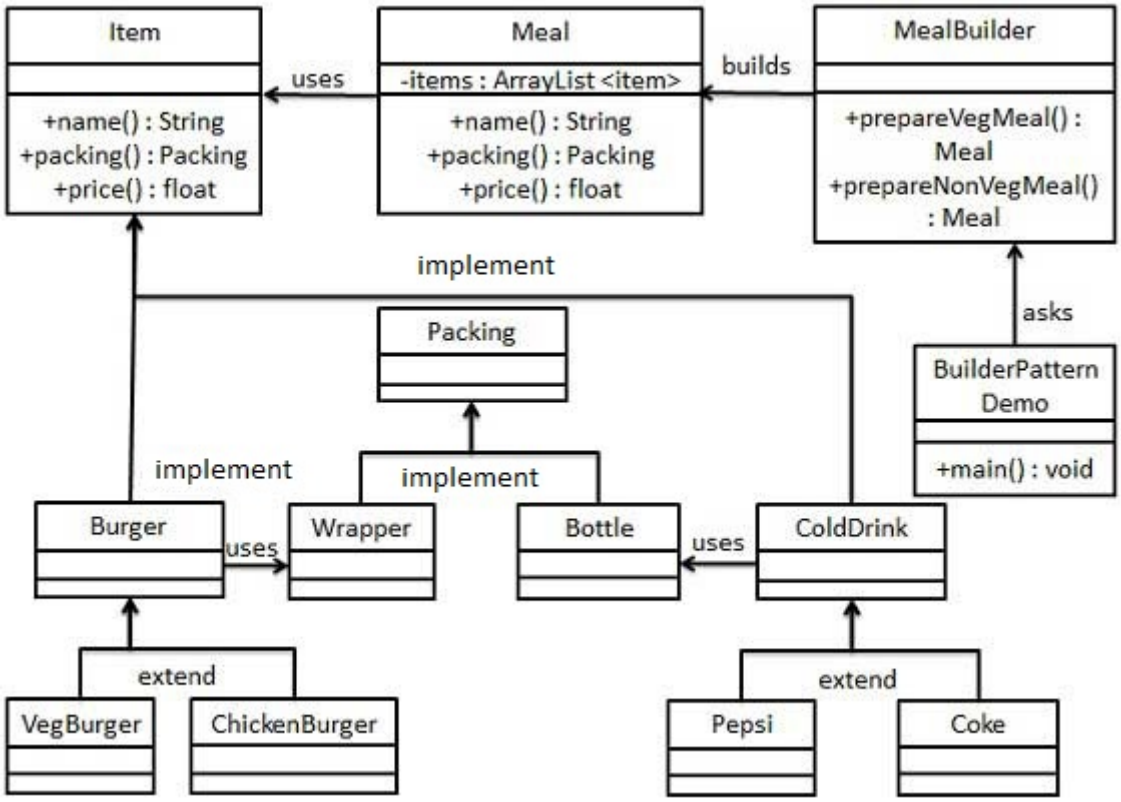

Payment Application
State Machine
Payment Application
Task Loader
Process Flow
Purchase Transaction
Login Transaction
the other transactions
CashBack Transaction
Void Transaction
EMVL3TransactionFlow
Device Configuration ProcessFlow
the other Configuration ProcessFlows
EMV Step
Normal Step
View ViewModel
ViewDelegate
Views
ViewHandler
Delegate Layer
LoginView &ViewlModle
SplashView &ViewlModle
PinInputView &ViewlModle
CardDetectingView &ViewlModle
Others
IDataVisitor Service
IWifiManager Service
ISecurity Service
IIso8583Parser Service
Other Services
Service Interface
Services
ServiceManager
DataVisitor Server
WifiManager Server
Security Server
Iso8583Parser Server
Other Servers
OEM BSP Base On Linux with Qt Framework
POXI Interface
Newland NAPI Interface
Qt Framework Interface

2.2. some classic design patterns-经典的设计模式介绍

2.2.1. creational-创建型

Name	Description
Abstract factory	Provide an interface for creating *families* of related or dependent objects without specifying their concrete classes.
Builder	Separate the construction of a complex object from its representation, allowing the same construction process to create various representations.
Dependency Injection	A class accepts the objects it requires from an injector instead of creating the objects directly.
Factory method	Define an interface for creating a *single* object, but let subclasses decide which class to instantiate. Factory Method lets a class defer instantiation to subclasses.
Lazy initialization	Tactic of delaying the creation of an object, the calculation of a value, or some other expensive process until the first time it is needed. This pattern appears in the GoF catalog as "virtual proxy", an implementation strategy for the Proxy pattern.
Multiton	Ensure a class has only named instances, and provide a global point of access to them.
Object pool	Avoid expensive acquisition and release of resources by recycling objects that are no longer in use. Can be considered a generalisation of connection pool and thread pool patterns.
Prototype	Specify the kinds of objects to create using a prototypical instance, and create new objects from the 'skeleton' of an existing object, thus boosting performance and keeping memory footprints to a minimum.
Resource acquisition is initialization (RAII)	Ensure that resources are properly released by tying them to the lifespan of suitable objects.
Singleton	Ensure a class has only one instance, and provide a global point of access to it.

- Buider - 建造者
    - 意图：将一个复杂的构建与其表示相分离，使得同样的构建过程可以创建不同的表示。
    - 主要解决：主要解决在软件系统中，有时候面临着"一个复杂对象"的创建工作，其通常由各个部分的子对象用一定的算法构成；由于需求的变化，这个复杂对象的各个部分经常面临着剧烈的变化，但是将它们组合在一起的算法却相对稳定。
    - 何时使用：一些基本部件不会变，而其组合经常变化的时候。
    - 如何解决：将变与不变分离开。
    - 关键代码：建造者：创建和提供实例，导演：管理建造出来的实例的依赖关系。



- builder in Qt BankDemo:

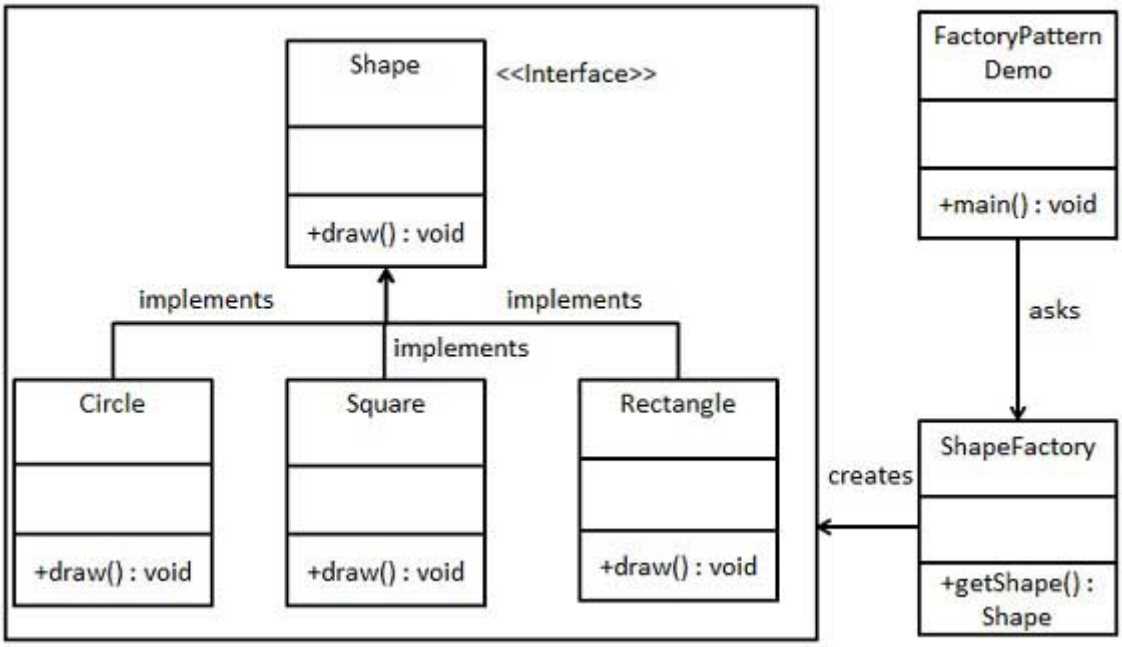
```
class DataVisitService : public QObject, IDataVisitor, IService
{
    Q_OBJECT
  public:
    class Builder
    {
    public:
         typedef enum DATA_SOURCE
         {
            SOURCE_FROME_FILE        = 1,
            SOURCE_FROME_DATABASE    = 2,
         }DataSource_t;

         Builder(DataSource_t source):mSource(source){}
         Builder &IP(const QString& ip){mIp = ip; return *this;}
         Builder &DatabaseName(const QString & name){mDataBaseName=name;
return *this;}
         Builder &FilePath(const QString& path="default"){mFilePath =
path; return *this;}

         DataVisitService *building(){
            return new DataVisitService(*this);
         }

         DataSource_t mSource;
         QString      mDataBaseName;
         QString      mIp;
         QString      mFilePath;
```

```
    };


    DataVisitService(const Builder &buidler);

    /// ... ...
};
```

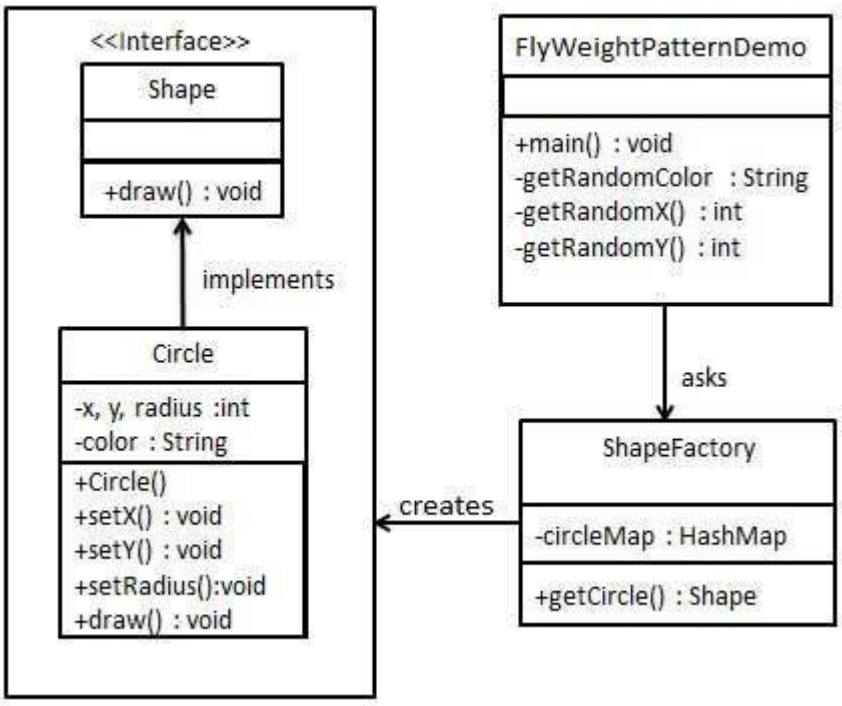
demo code fragment

```
DataVisitService* dataVisitingService = nullptr;

if(EnvReaderHelper::getBoolean(envReader, ENV_TAG_FIRST_RUN_STATUS, true))
{
    /// first running.
    dataVisitingService =
DataVisitService::Builder(DataVisitService::Builder::DATA_SOURCE::SOURCE_F
ROME_FILE)

.FilePath(EnvReaderHelper::getString(envReader,ENV_TAG_CONFIG_FILE_PATH,
"default")).building();
}
else{
    dataVisitingService =
DataVisitService::Builder(DataVisitService::Builder::DATA_SOURCE::SOURCE_F
ROME_DATABASE).building();
}

dataVisitingService->service_initialize();
dataVisitingService->service_start();
ServiceManager::getInstance()->registerService(dataVisitingService-
>ServiceName(), dataVisitingService);
```

- Factory Method - 工厂
    - 意图：定义一个创建对象的接口，让其子类自己决定实例化哪一个工厂类，工厂模式使其创建过程延迟到子类进行。
    - 主要解决：主要解决接口选择的问题。
    - 何时使用：我们明确地计划不同条件下创建不同实例时。
    - 如何解决：让其子类实现工厂接口，返回的也是一个抽象的产品。

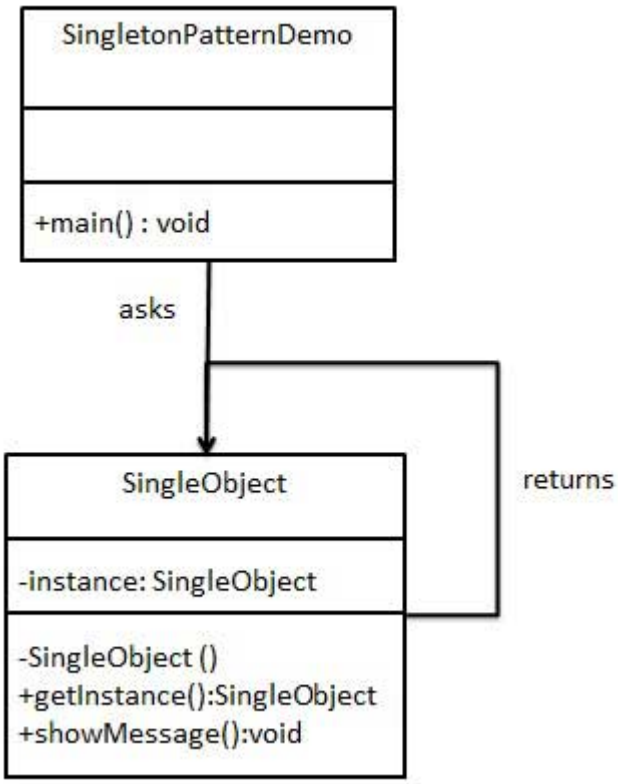


- Object Pool - 对象池 & FlyWeight - 享元
    - 意图：运用共享技术有效地支持大量细粒度的对象。
    - 主要解决：在有大量对象时，有可能会造成内存溢出，我们把其中共同的部分抽象出来，如果有相同的业务请求，直接返回在内存中已有的对象，避免重新创建。
    - 何时使用： 1、系统中有大量对象。 2、这些对象消耗大量内存。 3、这些对象的状态大部分可以外部化。 4、这些对象可以按照内蕴状态分为很多组，当把外蕴对象从对象中剔除出来时，每一组对象都可以用一个对象来代替。 5、系统不依赖于这些对象身份，这些对象是不可分辨的。
    - 如何解决：用唯一标识码判断，如果在内存中有，则返回这个唯一标识码所标识的对象。
    - 关键代码：用 HashMap/Map 存储已生产的对象并管理其生命周期。



- Singleton - 单例

- 意图：保证一个类仅有一个实例，并提供一个访问它的全局访问点。
- 主要解决：一个全局使用的类频繁地创建与销毁。
- 何时使用：当您想控制实例数目，节省系统资源的时候。
- 如何解决：判断系统是否已经有这个单例，如果有则返回，如果没有则创建。
- 关键代码：构造函数是私有的。



- Factory, Object Pool, Singletone in Qt BankDemo:

```
class ProcessFlowManager : public QObject
{
    Q_OBJECT
 public:
    ProcessFlow* getProcessFlow(const QString& nameOfProcessFlow);
    void init();

    ~ProcessFlowManager();

    static ProcessFlowManager* getInstance();

private:
    typedef std::function<ProcessFlow*(const QString&,QObject* )>
InstanceProvider;

    ProcessFlowManager(QObject* parent=0);
    ProcessFlowManager& registerProvider(const QString& providerName,
InstanceProvider instanceProvider);

    static ProcessFlowManager* mInstance;
```

```
    QMap<QString,  InstanceProvider>   mProcessFlowProviders;
    QMap<QString,  ProcessFlow*>       mProcessFlows;
};
```

```
/// in processflow.cpp:
ProcessFlowManager* ProcessFlowManager::mInstance = nullptr;
```

```
void ProcessFlowManager::init()
{
    registerProvider("Sale", [](const QString& name,QObject* parent)-
>ProcessFlow*{LOGGING_LEVEL("warn")<<"create Sale processflow";return (new
SaleTransactionFlow(parent));});
    registerProvider("Login",[](const QString& name,QObject* parent)-
>ProcessFlow*{LOGGING_LEVEL("warn")<<"create Login processflow";return
(new LoginProcessFlow(parent));});
}
```

Get The Process Flow:

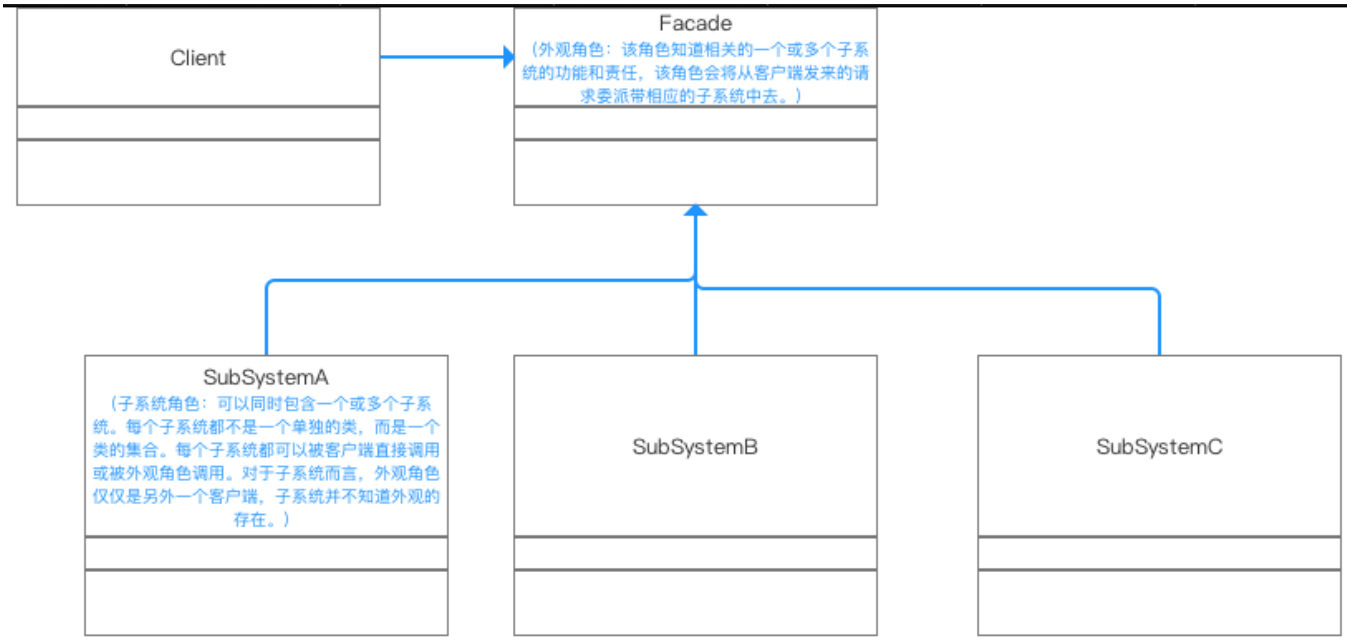
```
ProcessFlow *ProcessFlowManager::getProcessFlow(const QString
&nameOfProcessFlow)
{
    LOGGING_LEVEL("debug")<<"process name "<<nameOfProcessFlow;
    if(mProcessFlows.contains(nameOfProcessFlow))
    {
        LOGGING_LEVEL("debug")<<nameOfProcessFlow<<" process name contain
in process flows!";
        return mProcessFlows[nameOfProcessFlow];
    }

    if(!mProcessFlowProviders.contains(nameOfProcessFlow)){
        /// log error.
        LOGGING_LEVEL("warn")<<nameOfProcessFlow<<" not exist in
providers";
        return nullptr;
    }

    ProcessFlow* _object = mProcessFlowProviders[nameOfProcessFlow]
(nameOfProcessFlow, nullptr);
    _object->moveToThread(thread());
    mProcessFlows[nameOfProcessFlow]=_object;
    return _object;
}
```

```
ProcessFlow* idleProcessFlow =  ProcessFlowManager::getInstance()-
>getProcessFlow("Idle");

idleProcessFlow->installListener(new ProcessFlow::StateListener{
        .OnComplete = [&](Bundle& resultsOfProcessFlow){

            LOGGING_LEVEL("DEBUG")<<"@lambda function";
            LOGGING_LEVEL("DEBUG")<<"resultsOfProcessFlow "
<<resultsOfProcessFlow.toVariatMap();

            /// to test existed card event.
            if(resultsOfProcessFlow.contains(BUNDLETAG_USER_PAN)){
                /// read from idle then pass it to transaction.
                QString UserPAN
=resultsOfProcessFlow.getStringData(BUNDLETAG_USER_PAN,"null");

                SMRuntimeData runtimeData =
stateMechine.getRuntimeData();

                runtimeData->put(BUNDLETAG_USER_PAN, UserPAN);
                runtimeData->put("StateMachine.Transaction", "Sale");

                stateMechine.doTransaction();
                return;
            }

            stateMechine.halt();
            },

        .OnError=[&](IProcessFlow::ErrorCode error, Bundle&
resultsOfProcessFlow){
                Q_UNUSED(error)
            stateMechine.halt();
        }
});

idleProcessFlow->prepareRuntimeData();
idleProcessFlow->start();
```

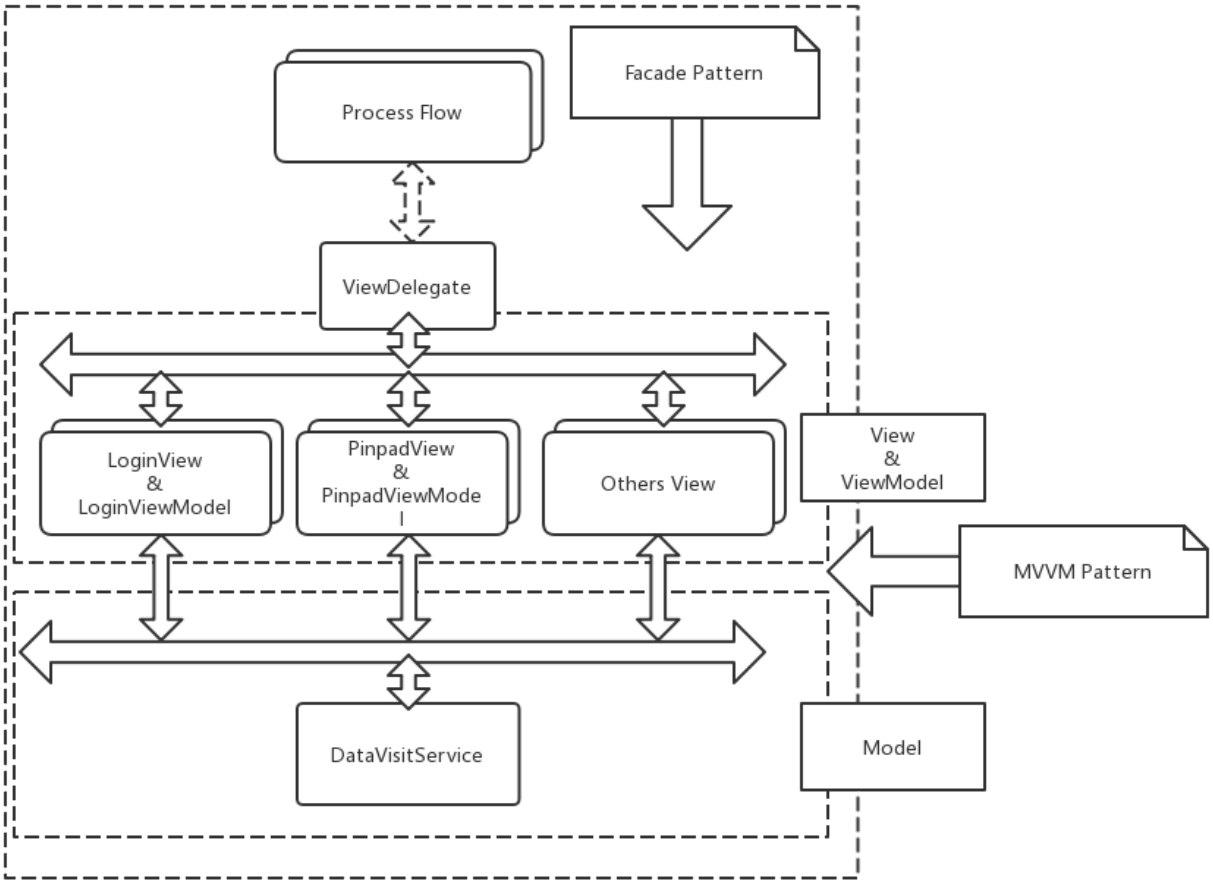
2.2.2 structural-结构型

Name	Description
Adapter, Wrapper, or Translator	Convert the interface of a class into another interface clients expect. An adapter lets classes work together that could not otherwise because of incompatible interfaces. The enterprise integration pattern equivalent is the translator.
Bridge	Decouple an abstraction from its implementation allowing the two to vary independently.
Composite	Compose objects into tree structures to represent part-whole hierarchies. Composite lets clients treat individual objects and compositions of objects uniformly.
Decorator	Attach additional responsibilities to an object dynamically keeping the same interface. Decorators provide a flexible alternative to subclassing for extending functionality.
Extension object	Adding functionality to a hierarchy without changing the hierarchy.
Facade	Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use.
Flyweight	Use sharing to support large numbers of similar objects efficiently.
Front controller	The pattern relates to the design of Web applications. It provides a centralized entry point for handling requests.
Marker	Empty interface to associate metadata with a class.
Module	Group several related elements, such as classes, singletons, methods, globally used, into a single conceptual entity.
Proxy	Provide a surrogate or placeholder for another object to control access to it.
Twin [19]	Twin allows modeling of multiple inheritance in programming languages that do not support this feature.

- Facade - 外观
    - 意图：为子系统中的一组接口提供一个一致的界面，外观模式定义了一个高层接口，这个接口使得这一子系统更加容易使用。
    - 主要解决：降低访问复杂系统的内部子系统时的复杂度，简化客户端与之的接口。
    - 何时使用： 1、客户端不需要知道系统内部的复杂联系，整个系统只需提供一个"接待员"即可。2、定义系统的入口。
    - 如何解决：客户端不与系统耦合，外观类与系统耦合。
    - 关键代码：在客户端和复杂系统之间再加一层，这一层将调用顺序、依赖关系等处理好。



- Facade in Qt BankDemo



demo code fragment:

```
class BindingStepView // this is a template method pattern.
{
```

```
public:
    typedef std::function<void(IViewDelegate&, Bundle& ,const
QVariantMap&)>      OnComplete;
    typedef std::function<void(IViewDelegate& ,int ,const QVariantMap&)>
OnError;

    BindingStepView(BaseStep* step, IViewDelegate* view)
                   :mStep(step),mView(view)
    {

    }

    virtual ~BindingStepView(){
        LOGGING_LEVEL("debug")<<"destory it!";
    }

    BindingStepView& bindOnComplete(OnComplete onComleted)
    {
        mOnCompleted = onComleted;
        return *this;
    }

    BindingStepView& bindOnError(OnError onError)
    {
        mOnError = onError;
        return *this;
    }

    BindingStepView& bind(){
        bind(mStep, mStep->getRuntimeDataBundle(), mView);
        return *this;
    }

    void start(){
        mView->openView();
        mView->showView();
    }

    void release()
    {
        delete this;
    }

private:
    void bind(BaseStep* step, Bundle& runtimeData, IViewDelegate* theView)
{
            BindingStepView* self = this;

            theView->setObserver(new IViewDelegate::IViewEventObserver{
                                    .onShowed  = [](IViewDelegate&){},
                                    .onComplete= [=](IViewDelegate& view,
const QVariantMap& response){
                                         Q_UNUSED(view)
                                         Q_UNUSED(response)
```

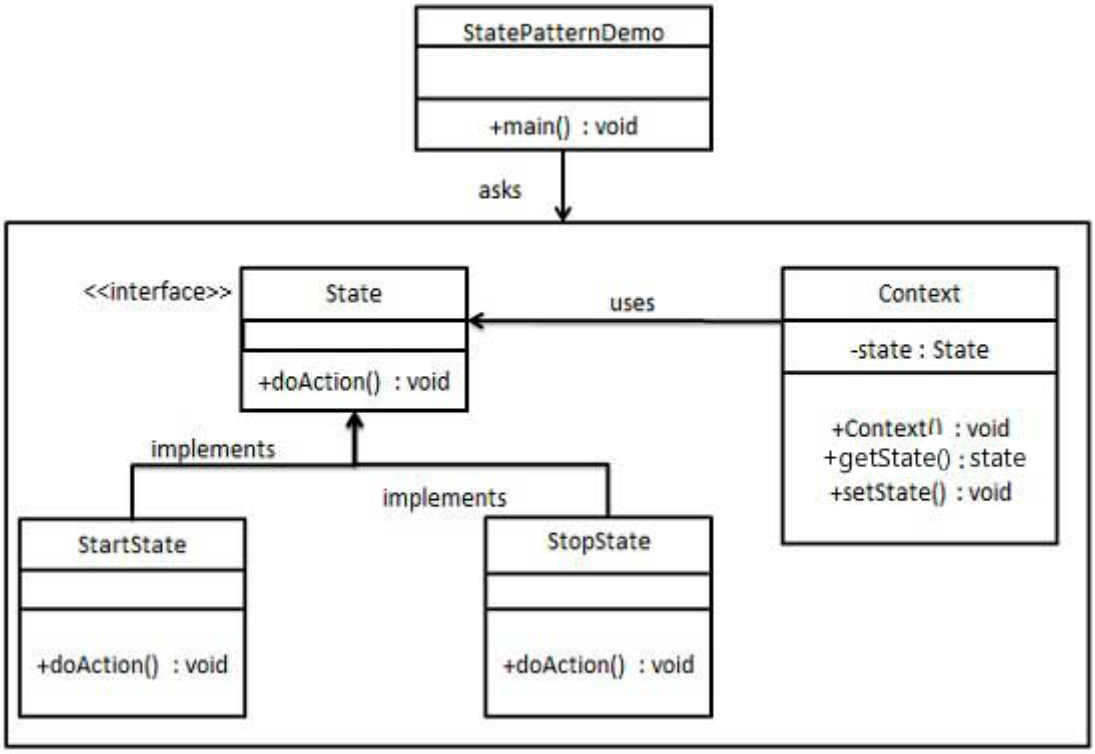
```
                                        LOGGING_LEVEL("debug")
<<"IViewDelegate::IViewEventObserver.onComplete";
                                        // todo add some implement to
handle response.
                                        // call runtimeData put some data
from response.
                                        if(self->mOnCompleted) self-
>mOnCompleted(view, const_cast<Bundle&>(runtimeData), response);

                                        step->completeIt();
                                        view.release();
                                    },
                                    .onHidden = [&](IViewDelegate& view){
                                        Q_UNUSED(view)
                                    },
                                    .onReleased = [&](IViewDelegate& view){
                                        Q_UNUSED(view)
//                                        self->release();
                                    },
                                    .onError  = [=](IViewDelegate& view,
                                                    int errCode,
                                                    const QVariantMap&
response) {
                                        Q_UNUSED(view)
                                        Q_UNUSED(errCode)
                                        Q_UNUSED(response)
                                        if(self->mOnError) self-
>mOnError(view, errCode, response);
                                        step->notifyError();

                                    }
                                });

          theView->setRequire(runtimeData.toVariatMap());
    }

    BaseStep*       mStep;
    IViewDelegate*  mView;
    OnComplete      mOnCompleted;
    OnError         mOnError;
};
```

```
IViewDelegate* splashView = Views::getView("SplashScreenDlg");

BindingStepView* binding = new BindingStepView(step, splashView);
binding->bindOnComplete(
    [binding](IViewDelegate& theView, Bundle& runtimeData,const
QVariantMap& viewsResponse){
        binding->release();//release binding when it completed
    }
).bind().start();
```

2.2.3 beharvioral-行为型

Name	Description
Blackboard	Artificial intelligence pattern for combining disparate sources of data (see blackboard system)
Chain of responsibility	Avoid coupling the sender of a request to its receiver by giving more than one object a chance to handle the request. Chain the receiving objects and pass the request along the chain until an object handles it.
Command	Encapsulate a request as an object, thereby allowing for the parameterization of clients with different requests, and the queuing or logging of requests. It also allows for the support of undoable operations.
Interpreter	Given a language, define a representation for its grammar along with an interpreter that uses the representation to interpret sentences in the language.
Iterator	Provide a way to access the elements of an aggregate object sequentially without exposing its underlying representation.
Mediator	Define an object that encapsulates how a set of objects interact. Mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it allows their interaction to vary independently.
Memento	Without violating encapsulation, capture and externalize an object's internal state allowing the object to be restored to this state later.
Null object	Avoid null references by providing a default object.
Observer or Publish/subscribe	Define a one-to-many dependency between objects where a state change in one object results in all its dependents being notified and updated automatically.
Servant	Define common functionality for a group of classes. The servant pattern is also frequently called helper class or utility class implementation for a given set of classes. The helper classes generally have no objects hence they have all static methods that act upon different kinds of class objects.
Specification	Recombinable business logic in a Boolean fashion.
State	Allow an object to alter its behavior when its internal state changes. The object will appear to change its class.
Strategy	Define a family of algorithms, encapsulate each one, and make them interchangeable. Strategy lets the algorithm vary independently from clients that use it.
Template method	Define the skeleton of an algorithm in an operation, deferring some steps to subclasses. Template method lets subclasses redefine certain steps of an algorithm without changing the algorithm's structure.
Visitor	Represent an operation to be performed on the elements of an object structure. Visitor lets a new operation be defined without changing the classes of the elements on which it operates.

- State & StateMachine - 状态与状态机:
    - 意图：允许对象在内部状态发生改变时改变它的行为，对象看起来好像修改了它的类。
    - 主要解决：对象的行为依赖于它的状态（属性），并且可以根据它的状态改变而改变它的相关行为。
    - 何时使用：代码中包含大量与对象状态有关的条件语句。
    - 如何解决：将各种具体的状态类抽象出来。



- State Pattern in Qt BankDemo:

  class declear.

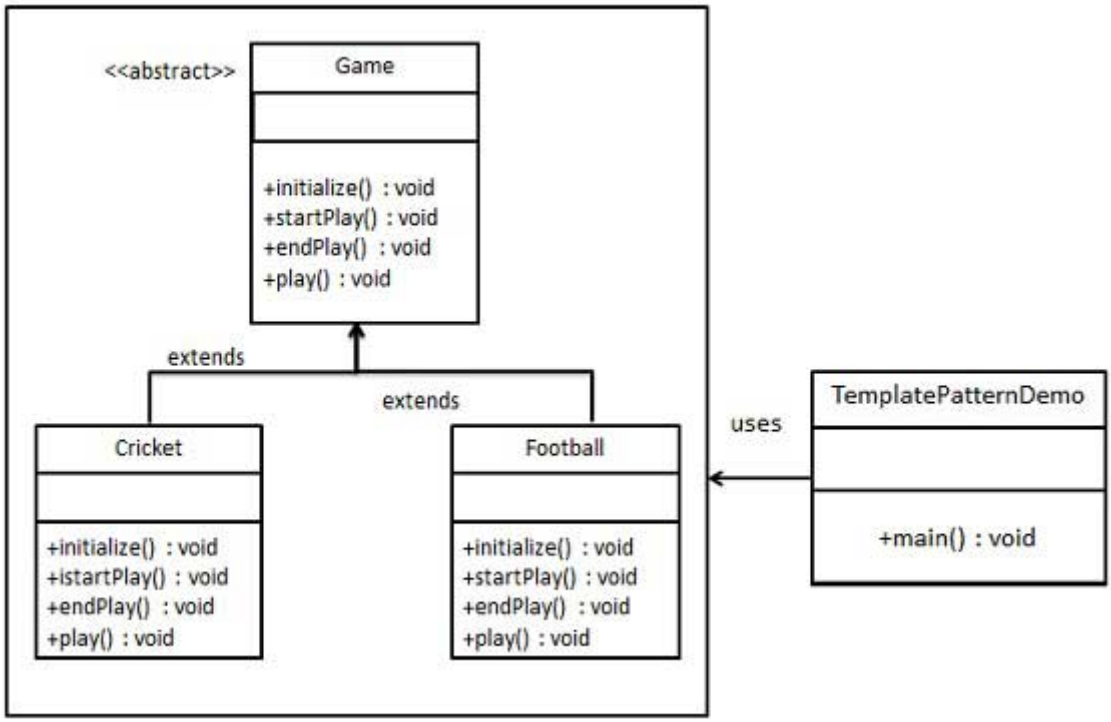
```
    StateMachine -> StateMachineBase

    class StateMachine : public StateMachineBase{

        StateMachine();

        // ... ... ignore more implementation.

        void halt();
        void doMainMenuSelection();
        void doTransaction();
        void tamper();
    }
```

  usage code fragment.

```
StateMachine    stateMechine;
stateMechine.addState(new StateMachine::ApplicationState(
                                StateMachine::STATE_ON_IDLE,
                                [](StateMachine &stateMechine){
                                    LOGGING_LEVEL("DEBUG")<<"@lambda
function";

                                    ProcessFlow* idleProcessFlow =
ProcessFlowManager::getInstance()->getProcessFlow("Idle");
```

```
                                    idleProcessFlow-
>installListener(new ProcessFlow::StateListener{
                                            .OnComplete = [&](Bundle&
resultsOfProcessFlow){

                                                LOGGING_LEVEL("DEBUG")
<<"@lambda function";
                                                LOGGING_LEVEL("DEBUG")
<<"resultsOfProcessFlow "<<resultsOfProcessFlow.toVariatMap();

                                                /// to test existed
card event.

if(resultsOfProcessFlow.contains(BUNDLETAG_USER_PAN)){
                                                    /// read from idle
then pass it to transaction.
                                                    QString UserPAN
=resultsOfProcessFlow.getStringData(BUNDLETAG_USER_PAN,"null");

                                                    SMRuntimeData
runtimeData =  stateMechine.getRuntimeData();

                                                    runtimeData-
>put(BUNDLETAG_USER_PAN, UserPAN);
                                                    runtimeData-
>put("StateMachine.Transaction", "Sale");


stateMechine.doTransaction();
                                                    return;
                                                }

                                                stateMechine.halt();
                                            },

                                            .OnError=[&]
(IProcessFlow::ErrorCode error, Bundle& resultsOfProcessFlow){
                                                Q_UNUSED(error)
                                                stateMechine.halt();
                                            }
                                    });

                                idleProcessFlow->prepareRuntimeData();
                                idleProcessFlow->start();
            })).addState(new StateMachine::ApplicationState(
                                StateMachine::STATE_ON_TAMPER,
                                [](StateMachine& stateMachine){
                                    /// handling something on tamper.
                            }));
```

- Template Method - 模板方法:
    - 意图：定义一个操作中的算法的骨架，而将一些步骤延迟到子类中。模板方法使得子类可以不改变一个算法的结构即可重定义该算法的某些特定步骤。
    - 主要解决：一些方法通用，却在每一个子类都重新写了这一方法。
    - 何时使用：有一些通用的方法。
    - 如何解决：将这些通用算法抽象出来。
    - 关键代码：在抽象类实现，其他步骤在子类实现。
    - 优点： 1、封装不变部分，扩展可变部分。 2、提取公共代码，便于维护。 3、行为由父类控制，子类实现。
    - 缺点：每一个不同的实现都需要一个子类来实现，导致类的个数增加，使得系统更加庞大。
    - 使用场景： 1、有多个子类共有的方法，且逻辑相同。 2、重要的、复杂的方法，可以考虑作为模板方法。



- The Template Method in Qt BankDemo:
    - EMV L3 Transaction Flow:

```
    class EmvLV3TransactionFlow : public QObject, IProcessFlow
    {
        Q_OBJECT
    public:
        /// level 3 event callback;
        typedef std::function<int(const QString&, const QString&)>
OnShowCardProcessingMessage;
        typedef std::function<int(const QString&)>
OnShowUerPANInfo;
        typedef std::function<int(void)>
OnWaitAnyKeyPressed;

        /// -----
```

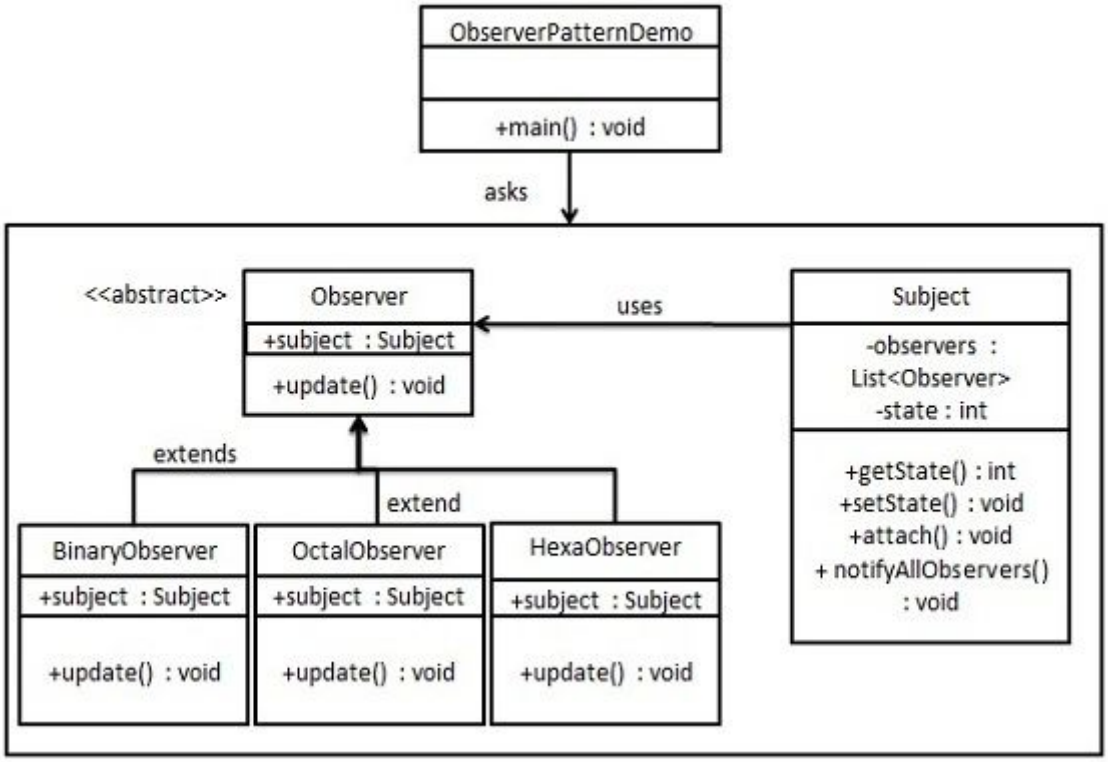
```
        explicit EmvLV3TransactionFlow(L3EventCallBack* callBack, QObject
*parent = 0);

        void prepareRuntimeData(const Bundle& runtimeData);
        void completeIt(Bundle &runtimeData);
        virtual void start() override;

    signals:
        void    sig1stGACCompleted(Bundle& runtimeData);
        void    sigTransactionCompleted(Bundle& runtimeData);
        void    sigError(int errorCode, Bundle& runtimeData);

    private:
        virtual void completeIt() override;

        int onShowCardProcessingMessage(L3_UIID uiEventId, unsigned char*
uiEventData);
        int onRequiring2InputUserPIN(L3_PIN_TYPE type, unsigned int cnt,
publicKey* pinPK, unsigned char *sw1sw2);
        int onRequiring2InputAmount(L3_AMOUNT_TYPE type, const char* amt);
        int onRequiringUserSelectCandidateList(L3_CANDIDATE_LIST
candidateList[], int listNum,  int *pSelect);
        int onRequiringSelectAccountType(unsigned char* acct_type);
        int onRequiringSelectLanguage();

        int onRequiringcheckCredentials();
        int onRequiringVoiceReferrrals(int* result);
        int onRequiringDekDet(unsigned char type, unsigned char* buf, int*
len);
        int afterFinalSelect(L3_CARD_INTERFACE interface, unsigned char
*aid, int aidLen);

        void onTransactionPerformCompleted(int status, L3_TXN_RES result);
        void onTransactionCompleted(int status, L3_TXN_RES result);
        void onTransactionTerminateCompleted(int status);
        void onDetectCardCallBack(int status, int res);

        Bundle&        getRuntimeData(){return mRuntimeData;}

    private:
        Bundle                          mRuntimeData;
        L3EventCallBack*                mCallBacks;

    };
```

usage code fragment:

```
mEmvTransactionFlow = new EmvLV3TransactionFlow(
                            new EmvLV3TransactionFlow::L3EventCallBack{
                                    .onShowCardProcessingMsg=[self](const
QString& tip, const QString& amount)->int{
                                            EmvStep* step = new
```

```
EmvStep("CardTechProcess", self);
                                            step->stateMachine()
                                                    ->addState(
                                                    new
BaseStep::State(BaseStep::STATE_ON_STARED,
                                                        [](BaseStep* step,
Bundle& runtimeData){

BindingStepView* binding = new BindingStepView(step,
Views::getView("CardTechProcessView"));
                                                            binding-
>bindOnComplete(
                                                                []
(IViewDelegate& view, Bundle& runtimeData,const QVariantMap& viewResponse)
{
                                                                    ///
TODO add implement
                                                                }
                                                            ).bindOnError(
                                                                []
(IViewDelegate& ,int ,const QVariantMap&) {

                                                                }

).bind().start();
                                                        })
                                                    );


TransactionHelper::setAmountWithTip(step, tip, amount);
                                            step->start();

                                            int ret =
TransactionHelper::getResponse(step);
                                            step->deleteLater();
                                            return ret;
                                    },
                                    .onShowUserPAN=[self](const QString&
PAN)->int{
                                        BaseStep* step =
newStepInstance("emv", "ShowCardPAN", self);

TransactionHelper::setUserPAN(step, PAN);
                                        step->start();
                                        return
TransactionHelper::getResponse(step);
                                    },

                                    .onWaitForKeyPressed=[](void)-
>int{return -1;}
                        }, self);
```

- Observer - 观察者
    - 意图：定义对象间的一种一对多的依赖关系，当一个对象的状态发生改变时，所有依赖于它的对象都得到通知并被自动更新。
    - 主要解决：一个对象状态改变给其他对象通知的问题，而且要考虑到易用和低耦合，保证高度的协作。
    - 何时使用：一个对象（目标对象）的状态发生改变，所有的依赖对象（观察者对象）都将得到通知，进行广播通知。
    - 如何解决：使用面向对象技术，可以将这种依赖关系弱化。



- The Observer in Qt BankDemo:

```
class IDataVisitor
{
public:
    static QString const SERVICE_NAME;


    virtual ~IDataVisitor(){}

    class DataVisitResponseObserver : public virtual
IServiceResponseObserver{

    public:
            typedef std::function<void(const QVariantMap&)> OnCompleted;
            typedef std::function<void(int)>  OnError;

           DataVisitResponseObserver(OnCompleted onCompleted, OnError
onError)
                                    :mOnCompleted(onCompleted)
```

```
                                    ,mOnError(onError)
            {

            }


            void onItCompleted(const QVariantMap& response){
                 this->mOnCompleted(response);
            }
            void onItError(int error){
                this->mOnError(error);
            }

    private:
            OnCompleted mOnCompleted;
            OnError     mOnError;
    };


    virtual void    getData(const QString& tag, DataVisitResponseObserver*
observer)                =0;
    virtual void    getDatas(const QStringList& tags,
DataVisitResponseObserver* observer)           =0;

    virtual void saveDatas(const QVariantMap& datas,
DataVisitResponseObserver* observer)                =0;
    virtual void saveData(const QString& tag, const QVariant& data,
DataVisitResponseObserver* observer) =0;

};
```

demo code fragment

```
IDataVisitor* dataVisitor = ServiceManager::getInstance()
                                ->getService<IDataVisitor>
(IDataVisitor::SERVICE_NAME);

             dataVisitor->getData("MerchaintName",
                 new IDataVisitor::DataVisitResponseObserver(
                     [runtimeData](const QVariantMap& response){
///onCompleted
                         // read data from response.
                     },
                     [runtimeData](int error){
                         // notify the error.
                     }
                 )
             );
```