Programming Guide

Presentation: Framework and Guidance for ARP HMI.pdf

Coding Convention

- Put your name on the class you created. Add your name if you have done large modification on class created by others.
- Don't repeat yourself.
- No throw and exception. Catch all exception if it comes from other lib.
- Think twice before using template.
- Use const and reference when applicable.
- Avoid C style casts.
- Follow Qt's public interface. Use camelCase. Naming the function like “doSomething()”.
- First alphabet for member variable should be “m”. It helps to make code more readable, and avoid variable shadowing.
- Set tab = 4 spaces.
- If not specified, follow the style of the class you are modifying.

Directory structure under "src"

Common	Common and general utility classes
Event	Event subsystem (command pattern)
InputMethod	IME
Presentation	Presentation subsystem, GUI (MVC pattern)
Resource	Resource subsystem
Service	Service subsystem, integration of AutoSDK, adapter for other system services or 3rd party libraries
Widget	UI subsystem

Guide on View

Normally, a View is composed of a View class and a Widget class. Widget is inherited from QWidget, which will show on screen. View class is an adapter to adapt it to IView interface.

1. Call "Tn::Ui::UiFactory::createWidget()" to create Widget from ui config.
2. Additional config for the Widget (e.g., set label text).
3. Call "setUiWidget()" to put Widget to View.
4. Additional signal/slot connection from Widget to View.

UI of a View is largely done by 9-patch image, Qt translate for text, ui config file, and Qt signal/slot
Sample: AutoClient/src/Presentation/Map/DashboardView.cpp

9-patch image
Provided by UX team (as well as normal image). AutoRef/trunk/AutoClient/resource/image
http://developer.android.com/tools/help/draw9patch.html

Qt translate
http://qt-project.org/doc/qt-4.8/linguist-manual.html
QObject::tr("Choose Route to %1")
Note: in "tr()", it must be a raw string, not a variable (i.e., not std::string, nor QString).

ui config file
AutoClient/resource/uiconfig/uiconfig.xml
Tn::Ui::WidgetBuilder will read this config to generate Widgets to populate the View.
What is Important is tag name (e.g., <Splash>), and class name (e.g., class="ImageLabel"). The rest is config for this Widget, based on the class.
The float numbers are in percentage, refer to VDD. Location is specified as the relative location to its parent Widget.

Qt signal/slot
http://doc.qt.digia.com/qt/signalsandslots.html
Similar to observer pattern, but it is very flexible and easy to use.
Note: in header, "Q_OBJECT" macro is needed.

Guide on control flow

2 stage state machine. Lower one is for major state (e.g. map, nav, poi). Higher one is for minor state (e.g. dashboard, full map).
Sample: AutoClient/src/Presentation/Map/MapStateDashboard.cpp as class for a minor state

1. Inherit a ViewState class from ControllerState. Insert it in controller state machine in "setupStateMachine()" of Controller.
2. Override "onEnterState()", "onExitState()" in ViewState class to show / hide the View.
3. Override "executeCommand()" in related states, in it call "transitionToState()" of FiniteStateMachine to complete the flow.

Guide on command handling

View send Command to Controller for UI request. Call sendCommand() method in BaseView to send a command to controller.
The command will be first dispatched to current State in Controller. If current State does not handle it, it will be handled in Controller.
“back” and “close” command are by default handled in BaseController. “back” will cause the state machine go back to a previous state, and “close” will close the Controller. If this is not what you want, handle it explicitly in "executeCommand()" method in State or Controller, this will override the default handler.

1. Add command name to some common class (e.g. MapNaming.h).
2. Send it out by "sendCommand()" method in View class.
3. Override "executeCommand()" in State or Controller class to handle it.

Guide on service request

View asks for data from Model, but when there is some thing new which Model cannot provide (e.g., user press a button to search new poi), View need to send Command for updated data.
Service request from View is also sent by command. But there is one thing which is different. A service command will be parsed in ServiceCommandParser and a CommandImpl object will be attached.
For view, it still send command by "sendCommand()" method, and optionally register a Qt slot for callback. The slot will be called when the request is finished.
For controller, normally it needs to do nothing.
Sample: AutoClient/src/Service/NavigationService/NavCalcRouteCommandImpl.cpp

If the service request is not already implemented, you need to implement it in Service class, and add a new CommandImpl class for it.

1. Add interface in Service class to process the request.
2. Inherit a CommandImpl class from ServiceCommandImpl.
3. Override "parseCommand()" method to parse the request from command arguments.
4. Override "execute()" and "cancel()" method to execute or cancel the request.
5. Override "parseCommand()" in ServiceCommandParser to attach the CommandImpl to a proper Command.