Typing Analysis

Dima Trushin

Contents

1 General information

The application uses Qt environment as an ecosystem. It does not use RTTI but uses exceptions. Application uses static linking and presents a single binary file as a result. All components of the program must be inside `NSApplication` namespace. Nested namespaces should be placed in a subfolder of the project.

1.1 Addressable objects

I use a notion of an addressable object. It means an object that can be referenced by other objects. There are several types of addressable objects:

1. `QObject`.
2. `CApplicationImpl` or any other implementations located on the heap (PIMPL).
3. Observer/Observable.

Addressable objects must be created on the heap via `std::make_unique` or similar mechanism. An exception to this rule: you may create an addressable object inside another addressable object.

1.2 Exceptions

The strategy is to catch an exception, show a message, and die. Since Qt environment is not totally exception safe, it is not allowed to throw exceptions in an event loop. If a `QObject` requires to throw an exception and terminate the program it catches the exception by itself and then sends a signal to `CQtLoopException` object. `CQtLoopException` shows a message and stops the event loop. `CQtLoopException` is a singleton, it uses `CAnyGlobalAccess` template (see 7.1).

2 Application behavior

The global logic of the entry point is the following:

1. Application acquires all the required resources.
2. Application starts the Qt event loop.

When started the application intercepts keyboard events system-wide, translates them into OS independent form and shows the debug output via `qDebug()`. If debugging macros are enabled (see 4), additional debugging windows are shown to output debug information.

(TO DO) The KeyboardHandler must notify the kernel of the application. The kernel computes the required information and notifies the interface of the application. From the other hand the interface elements control the kernel. All elements of the kernel are connected using Observer pattern 7.3. The interface and the kernel are connected using MVC pattern. The MVC pattern is based on the Observer pattern as written here by Sean Parent.

3 Application Structure

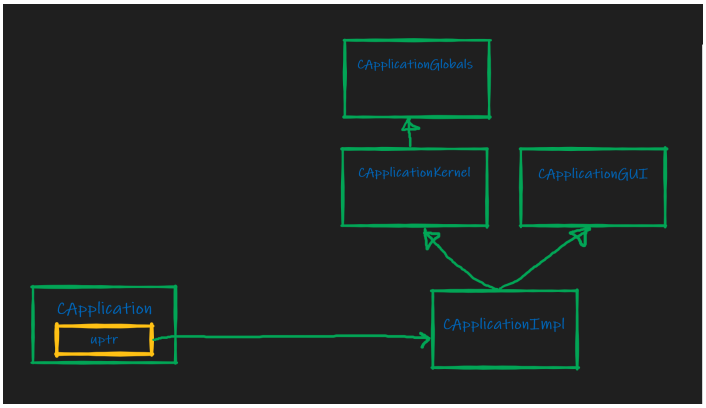
3.1 Overview

`CApplication` object initializes all required resources for the application including the ones to interact with Qt ecosystem. It may throw an exception while constructing. In order to minimize stack usage `CApplication` contains `std::unique_ptr` to `CApplicationImpl` (it is an addressable object). `CApplicationImpl` consists of four parts:

1. `CApplicationGlobals`. Its purpose is to initialize global resources, e.g., timers, loggers, thread pools, etc. Application initializes all global resources (basically singletons) at the start. This allows not to wast time on the first call. Also, it is inconvenient to initialize timers via the first call.
2. `CApplicationKernel`. Its purpose is to initialize the kernel of the application. The kernel does not depend on the GUI and uses MVC via observer pattern to interact with GUI.
3. `CApplicationGUI`. Its purpose is to provide View wrappers over Qt resources compatible with MVC pattern.

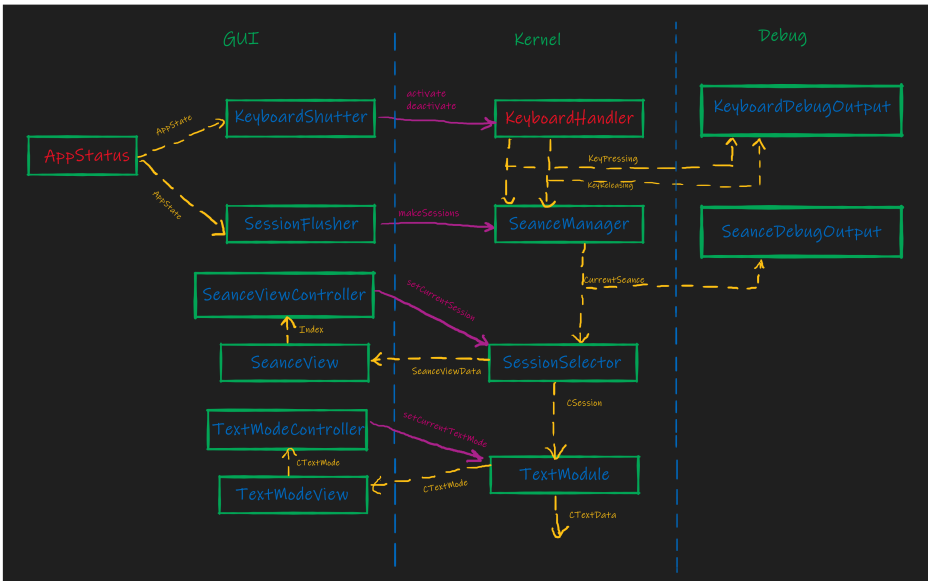
4. `CApplicationImpl`. Its purpose is to connect the kernel and the GUI via MVC.

The order of construction is ensured by the inheritance mechanism.



3.2 Application data flow

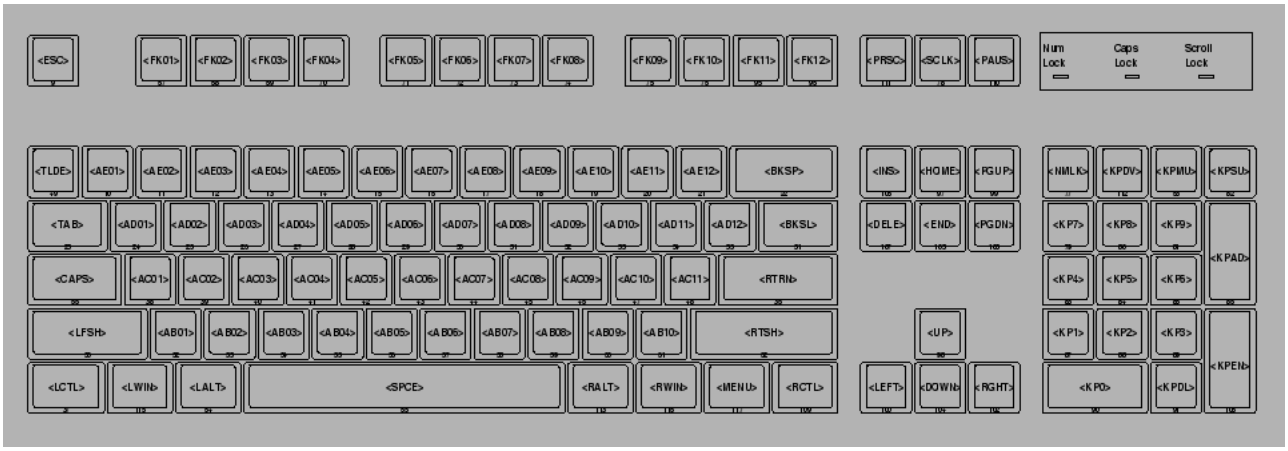
Currently the following modules are implemented.



The boxes are application components. Red color means that the objects are global and blue color means that the objects are local. Yellow lines denote Observable/Observer relationship. Violet lines represent Control/Model relationship. The arrows denote the direction of the data flow.

3.3 Keyboard

Each key on a keyboard has its position identifier `CKeyPosition` and a key identifier `CKeyId`. `CKeyPosition` points out a physical location on a keyboard (e.g. a row and a column where the key is located). `CKeyId` identifies the key depending on the keyboard layout (qwerty, Dvorak, etc.) `CKeyPosition` is an enum with the following identifiers (xkb identifiers):



The picture is taken from here

It should be noted that the numeric values of the identifiers are slightly different from the ones in xkb.

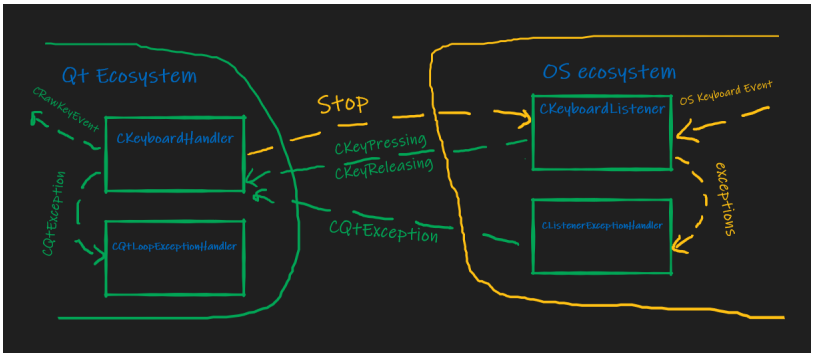
3.4 Keyboard Interception

The application must intercept keyboard events system-wide even if application is on the background. Qt ecosystem does not allow doing this. Hence, we need to implement our own mechanism. System wide interception usually means that raw keyboard data is being intercepted. Hence, we need to implement translation of the key events to generated text. The main problem here is dead keys and ligatures (left directional and bidirectional text is not supported in any way, shape, or form). Current implementation supports dead keys but not ligatures (but the design allows to extend implementation to ligatures TO DO). Since system-wide interception must be done on the level of current OS, we implement this mechanism separately for each supported OS. Currently the list of the supported OS is the following

1. Windows 8 or higher.
2. Linux with X11 (there is a working prototype, not implemented yet, need to specify all the details TO DO).
3. macOS 10.14 (Mojave) or higher (there is a working prototype, not implemented yet, need to specify all the details TO DO).

Current design The design is based on ideas of Alexey Sokolovsky. He is also in charge of Linux and macOS implementations.

The main object here is `CKeyboardHandler`. It serves as an object receiving all key events in a system independent form. This is a singleton described in section 3.7.2. Let us consider the following figure:



Green arrows represent queued Qt signals and yellow arrows represent OS dependent signals.

`CKeyboardHandler` acts on the main GUI thread. It lives in Qt ecosystem and spawns an additional thread while construction. This additional thread lives in the OS ecosystem.

CKeyboardListener The main object acting on the additional thread is `CKeyboardListener`. Its roles are:

1. Intercept key events system-wide.
2. Extract all the information from OS key events and translate them to OS independent form.
3. Resend OS independent key events to `CKeyboardHandler` via Qt event system.
4. It receives only one message from `CKeyboardHandler` via OS ecosystem message system. This is a "stop" message. Receiving the message, it stops the additional thread.

There are two types of messages sent by `CKeyboardListener`:

1. Key pressing message.
2. Key releasing message.

Key pressing message contains more information.

CListenerExceptionHandler The second object acting on the additional thread is `CListenerExceptionHandler`. Its roles are:

1. Extract exception information.
2. Resend special exception messages to `CKeyboardHandler`.

If an exception happens the event loop on the additional thread stops. `CKeyboardHandler` resend the message to `CQtEventLoopExceptionHandler`. The latter one shows the error message and terminates the program.

When we need to shut down the keyboard interception temporary, we make `CKeyboardHandler` to ignore the messages from `CKeyboardListener`. The latter object is always on and operational.

3.5 System independent Key event

The `CKeyEvent` contains the following information (TO DO not fully implemented yet, subject to change):

1. Pressing time.
2. Releasing time.
3. `CKeyPosition` is an identifier of the Key position on the keyboard (independent on the layout, this is a physical instance of a key). It identifies the key uniquely.
4. `CKeyId` is an identifier of the Key depending on the layout being used. It is based on the Windows Virtual Key table.
5. `CLabelData`. Key Label. This one describes how to label a key on the keyboard. It contains at most two `QChar`s (the symbol for the key and the symbol for the key with shift being pressed). For example, if a key is a letter or a number key, then the label contains two symbols. For system keys like shifts, backspace, enter, etc, the label information usually contains one special symbol.
6. `CKeyTextData`. Key Text. This is a sequence of symbols appearing after pressing this key. It can be 0, 1, or 2 symbols. The exact rules are explained below.
7. Flags (subject to change, currently are not implemented TO DO). These flags are reserved and can be needed to pass some additional information.

`CKeyboardHandler` receives the following structures from `CKeyboardListener`:

1. `CKeyPressing`.
2. `CKeyReleasing`.

The structure `CKeyPressing` contains the following fields:

1. `CTime` The pressing time of the key.

2. `CKeyPosition` The OS independent position of the key.

3. `CKeyID` The OS independent ID of the key.

4. `CLabelData`. The key label information. These symbols are shown on the keyboard. It contains `LowSymbol`, `HighSymbol`, and `Size` fields.

5. `CKeyTextData`. The text generated by the key. It contains `Symbol[2]` array of `QChar`s and `Size` fields.

The structure `CKeyReleasing` contains the following fields:

1. `CTime` The releasing time of the key.

2. `CKeyPosition` The OS independent position of the key.

3. `CKeyID` The OS independent ID of the key. (subject to change, probably need to exclude it TO DO)

When we need to pair the pressing and releasing events into one key, we identify the key by its position. It should be noted that the ID of the key can change. For example, if we press a number on the numpad (and keep pressing), then press NumLock, and only then release the numpad number key, the numpad key will have different IDs while pressing and releasing. This happens because NumLock changes the mapping for physical keys to the corresponding IDs.

3.6 Key mapping

This section is mostly influenced by a series of blogs by Michael S. Kaplan. His posts explain all subtleties of the keyboard handling on Windows. However, the principals he explains are applicable to any keyboard handling on any OS. I do not implement all the features (TO DO need a full list of implemented and non-implemented features).

We split keys into the following categories:

1. Producing symbols keys (not necessarily printable symbols). These keys may produce symbols depending on the sate of the keyboard (not necessarily in all possible states).

2. Shifters. These are: Shift (left, right), Ctrl (left, right), Alt (left, right), Capslock.

3. Ignorable. All other keys.

The key Capslock has a toggled state. Also NumLock and ScrollLock have toggled state but we ignore toggled states of these keys.

The symbol appearing on the screen depends on the sequence of keys pressed and not just one key. There are several reasons for that: dead keys or chained dead keys, some modifiers like Capslock or NumLock. I am currently focused on the dead key handling but keep in mind chained dead keys and other possible features for the future implementation. Modifiers behave as expected. In order to handle a key pressing, we need to know:

1. Key.

2. Shifters.

3. Layout.

These triples may be classified as follows:

1. Undefined. These combinations do not produce any symbols and are ignored by the symbol producing system.

2. Control. These combinations do not produce any symbols[1] and are ignored by the symbol producing system. However, this combinations may copy and past some text. We cannot handle these events and just ignore them. They are usually combinations of Ctrl or Ctrl+Shift with some other keys.

3. Printable Key. These keys usually produce a single symbol. There are cases when they produce a ligature, that is, a sequence of two UTF-16 characters combined into one symbol on print (ligatures currently are not supported but the support is possible).

[1] The OS may actually produce a control symbol in this case.

4. Printable Dead Key. These are special keys. They do not produce a symbol on its own but can be combined with other keys if compatible. The symbol producing system has a special buffer to store a previous dead key. This dead key is flushed from the buffer if the symbol producing system accepts one of the next keys. Theoretically, it is possible to use chained dead keys, that is, to store a sequence of dead keys and then compose them with other characters. However, this feature is not considered[2] and is not implemented.

5. Non-printable Key. (On Windows they are: cancel, backspace, tab, enter, esc). These keys are a pain in the neck. They behave differently in different Text Editors.

These triples behave as follows.

1. Undefined. These combinations do not produce any symbols and are ignored by the symbol producing system.

2. Control. These combinations do not produce any symbols and are ignored by the symbol producing system.

3. Printable Key. This key behaves as follows:

   (a) If there is no active dead key (the dead key was not pressed before this key or was flushed from the buffer of the symbol producing system). In this case Printable Key results in a single printable symbol.

   (b) If there is an active dead key (the dead key was pressed and is in the buffer of the symbol producing system). In this case the behavior is different on all three OS:

Windows: If the Printable Key is compatible with the dead key they result in a single composed character. In other case, they produce a pair of symbols: a symbol for the dead key and a symbol for the Printable Key. The dead key is flushed from the buffer.

Linux: If the Printable Key is compatible with the dead key they result in a single composed character. In other case, they produce no symbols. The dead key is flushed from the buffer.

macOS: It should be noted that in this case, the symbol of the dead key was printed. If the Printable Key is compatible with the dead key, the dead key symbol is deleted and a single composed character is produced. In other case, the symbol of the Key being pressed is produced. The dead key is flushed from the buffer.

4. Printable Dead Key. This key behaves as follows:

   (a) If there is no active dead key (the dead key was not pressed before this key or was flushed from the buffer of the symbol producing system). In this case the behavior is different on all three OS:

Windows: This dead key is stored in the buffer of the symbol producing system. No symbol is generated.

Linux: This dead key is stored in the buffer of the symbol producing system. No symbol is generated.

macOS: This dead key is stored in the buffer of the symbol producing system. The symbol of the dead key is generated.

   (b) If there is an active dead key (the dead key was pressed and is in the buffer of the symbol producing system). In this case the behavior is different on all three OS:

Windows: The dead keys are never compatible. They produce two symbols: a symbol for the first dead key and a symbol for the second dead key. The dead key is flushed from the buffer.

Linux: The dead keys are never compatible. They produce no symbols. The dead key is flushed from the buffer.

macOS: (TO DO) Need to explore the behavior.

5. Non-printable Key. (On Windows they are: cancel, backspace, tab, enter, esc). This key behaves as follows:[3]

   (a) If the key is Enter (Tab). In case of an active dead key, it produces two symbols: a symbol for the dead key and a new line symbol (Tab symbol), then it flushes the dead key from the buffer. In case of no active dead key, it produces a new line symbol (Tab symbol).

   (b) Any other key. It flushes dead key from the buffer if any and produces no symbols.

[2] partially because it is not supported on Windows.
[3] Need to explore the behavior in all OSes.

3.6.1 Keyboard Interception on Windows

`CKeyboardListenerWin` substitutes `CKeyboardListener` on Windows. This object is addressable, hence it stores a unique pointer to its implementation. When created it does the following operations:

1. It registers and creates a background message Window.
2. It provides a hook to RAW INPUT system with global keyboard interception.
3. It sends a special Killer object to `CKeyboardHandler` via `std::promise` to receive the "stop" messages.
4. It connects its Qt signals to the corresponding Qt slots of `CKeyboardHandler` to send messages about key events.

This object starts a Windows message loop on its thread and reacts to two messages:

1. `WM_STOP_LISTENING`. This is an application defined event for the "stop" message.
2. `WM_INPUT`. It corresponds to RAW INPUT mechanism.

In the event loop

1. On `WM_STOP_LISTENING` message. It posts the Quit message to terminate the Windows thread loop. Since it receives such a message the main thread is about to terminate the application, hence we do not need to execute any additional steps.
2. On `WM_INPUT` message. It performs the following steps:
   (a) Get time of the message. There are two options: get current time from the global Timer or get OS provided time. I stick to the first option to have one consistent time line in the application.
   (b) Extract `RAWKEYBOARD` data from the OS key event.
   (c) Compute `CKeyPosition` and if it is unknown, then return.
   (d) Compute `CKeyID` and if it is unknown or ignorable, then return.
   (e) Check if the event is pressing or releasing:
      i. If pressing. It computes `CLabelData` and `CKeyTextData`, then it generates `CKeyPressing` structure and sends the corresponding Qt signal to `CKeyboardHandler`.
      ii. If releasing. It generates `CKeyReleasing` structure and sends the corresponding Qt signal to `CKeyboardHandler`.

Structure of CKeyboardListenerWin The object `CKeyboardListenerWin` contains an `std::unique_ptr` to `CKeyboardListenerWinImpl`. The latter object contains (this is not a complete list but represents the key parts of the object):

1. static method `WndProc`.
2. `CWinRawInputHook KeyboardHook_`.
3. `CRawInputReader RawInputReader_`.
4. `CKeyPositionWin KeyPosition_` .
5. `CKeyTextMaker KeyTextMaker_`.

- `WndProc` is a windows procedure. It is used by the message window created on the additional thread to react to OS messages. This function calls an appropriate method of `CKeyboardListenerWinImpl`.
- `CWinRawInputHook` provides integration of `CKeyboardListenerWinImpl` into Windows ecosystem. It registers and creates a non-gui message window and then register this window in RAW INPUT system to receive all keyboard messages system-wide.
- `CRawInputReader` provides a convenient way of extraction `RAWKEYBOARD` data from a key event.
- `CKeyPositionWin` computes `CKeyPosition` for a key.
- `CKeyTextMaker` gets a text produced by the current key event and also produces labels for keys.

Known issues In order to get a symbol from a key event `KeyTextMaker_` needs to know the current keyboard layout. It gets the layout of the current foreground window. However, certain windows do not receive broadcast messages, hence are unaware of the layout changes. For example, all console windows, calc.exe, etc. If you type while a console window has focus and switch the layout, the application will not notice this. This is a well known issue. This happens because a console application is not a window from the OS point of view. It has a parent window that does receive the broadcast messages, however it is a complicated task to find out the handle to the parent window. I am leaving this issue as it is for now. In order to solve the issue, we need to modify the `getForegroundLayout()` method in `CWinKeyboardApi` facade.

3.6.2 Keyboard Interception on Linux

TO DO

3.6.3 Keyboard Interception on macOS

TO DO

3.7 Global Objects

3.7.1 Timer

This module was written by Tagir Kuskarov.

Application uses one main timer. This is a singleton with respect to the template here 7.1. It starts in the constructor of CApplicationGlobal. Timer return CTime object. It does not depend on particular units but you may convert it to any units you want. Internally `std::chrono` is used.

3.7.2 KeyboardHandler

Application uses `CKeyboardHandler` object to intercept keyboard system-wide, that is, it intercepts the keyboard even if application has no focus. Qt does not support such functionality. Thus the corresponding mechanism is implemented.

`CKeyboardHandler` is wrapped into a singleton according to Section 7.1. It is initialized in `CApplicationGlobal`. From the user point of view `CKeyboardHandler` is able to send the following information to the application objects:

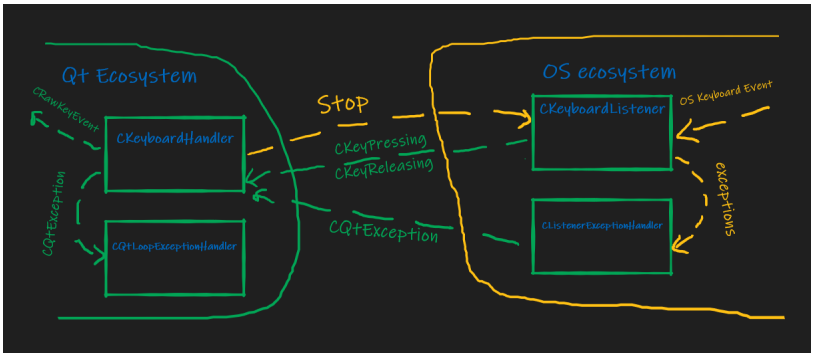
1. `CKeyPressing` event. This is a system independent representation of a key pressing event. This event is sent via Observer pattern.
2. `CKeyReleasing` event. This is a system independent representation of a key releasing event. This event is sent via Observer pattern.
3. `CQtException`. This is a message with information about an exception encountered. `CKeyboardHandler` sends a Qt signal to `CQtLoopExceptionHandler` with the corresponding `CQtException` object. The application terminates on any exception.

Interception of the keyboard is implemented as follows. `CKeyboardHandler` spans a worker thread with OS dependent message loop. There are two objects operating on the worker thread:

1. `CKeyboardListener`. This object starts an OS dependent event loop with `exec()` function. It listens to two type of messages:
   (a) OS key events.
   (b) `CKeyboardHandler` "stop" event.

   `CKeyboardListener` intercepts any key events in OS, transforms system dependent key events into `CKeyPressing` and `CKeyReleasing` events, and sends the events to `CKeyboardHandler`. If the "stop" event is encountered the event loop on the worker thread is terminated and the thread stops.
2. `CListenerExceptionHandler`. This object intercepts any exceptions on the working thread, transforms the exceptions to `CQtException`-s, and sends them to `CKeyboardHandler`. If an exception is encountered the event loop is terminated and the worker thread stops.

`CKeyboardListener` is OS dependent. Currently support for Windows, macOS, and Linux is provided (TO DO only Windows listener is implemented). In order to send the "stop" signal in and OS independent fashion the application uses `CAnyKeyboardKiller` object. It is based on `CAnyMoveable` object as described in 7.2.1.



Green arrows represent queued Qt signals and yellow arrows represent OS dependent signals.

3.7.3 AppStatus

This object is a wrapper over the Qt application instance. It has observable values in terms of section 7.3. It is currently used to react on switching between active and inactive states of the application. For example, I do not want to intercept keyboard, while the user interacts with the application GUI. Also, when user switch back to the application, we upload all intercepted sessions.

This object is a singleton according to section 7.1. It has one output sending the application current status. There are two options: active and inactive.

3.8 Kernel

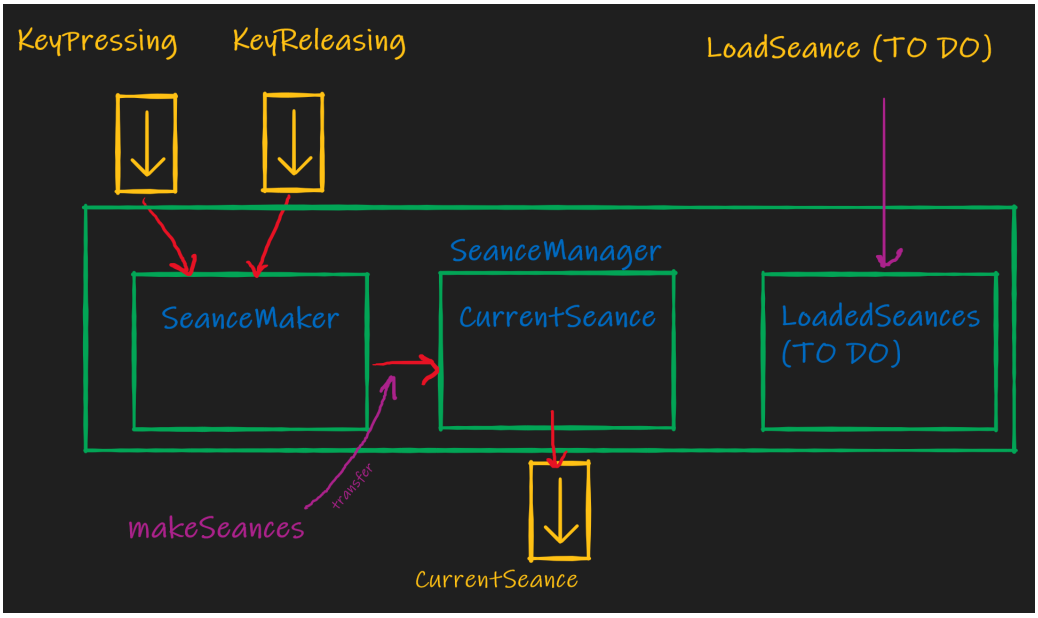
3.9 Sessions and Seances

When the application runs it uses a continuous time scale. Each new run of the application has its one time scale starting from zero. All key events on the same time line are grouped into a Seance. Each Seance can be separated into peaces called Sessions. The Session is considered as a peace of information to analyze. Each session is analyzed independently. When we load a file it contains key events on a different time scale. The notion of Seance allows us to separate current Seance intercepted while application is running and other Seances that are loaded from files.

The Seance is presented by `CSeance` class. Currently this is an extension of `std::deque` of `CSession`-s (TO DO partially implemented). Session is presented by `CSession` class. Currently this is an extension of `std::vector` of `CKeyEvent`-s. (TO DO partially implemented).

3.10 CSeanceManager

To understand `CSeanceManater` and its responsibilities let us look on the following picture:



Green boxes represent objects. Yellow boxes are input/outputs according to the observer pattern. Violet arrows are methods to control the manager via MVC. Red arrows represent the data flow.

The manager has the following components:

1. `SeanceMaker`. This object receives key pressing and releasing events and store them in a raw form. It also divides the stored events into `CRawSessions`. The `CRawSessions` use `std::list` container to have low cost modification actions.
2. `CurrentSeance`. Current seance stores all sessions intercepted by the application.
3. `LoadedSeances`. (TO DO not implemented yet) This object manages all loaded seances, e.g., loaded from files.

The manager's responsibilities:

1. It receives Key Pressing and Key Releasing events from `CKeyboardHandler` via observer pattern.
2. It receives new seances from a file loader (TO DO not implemented yet).
3. It returns current Seance to other parts of the application via observer pattern.
4. It returns current Seance and loaded Seances data (TO DO not implemented yet) via observer pattern. This is used by GUI to represent the set of seances visually.

The manager has the following control options:

1. `makeSessions`. This method transforms `CRawSessions` from `SeanceMaker` to `CSessions`, appends them to `CurrentSeance`, and clears the internal buffer, that is, `CRawSeance`.

SeanceMaker This object eats `CKeyPressing` and `CKeyReleasing` events. It tries to pair pressing and releasing events to form `CKeyEvent`. Internally, `SeanceMaker` contains `RawSeance` that is `std::list<CRawSession>` and `CRawSession` is `std::list<CKeyEvent>`. There are several things you should be aware off:

1. If you press a key and hold it for some time, the key may generate artificial key pressing events in specific time intervals. The artificial key pressing events are marked by `AutoRepeat` flag and do not have releasing time (releasing time is set to be zero). `SeanceMaker` distinguishes the keys with autorepeat support and without. Currently autorepeat is not allowed for the shifters (left and right shift, alt, and ctrl and capslock).
2. The logic for division of `CRawSeance` into `CRawSessions` is the following. `SeanceMaker` keep track the time of the last event and the keys that are being pressed currently. If the current `CRawSession` is not empty, there are no pressed keys, and the time since the last event exceeds the time limit (if it is set) the new `CRawSession` is opened. Currently, the time limit is set to be 10 seconds. (TO DO) In the future, there will be an option to set the time limit via application settings.

3.11 Qt Resources

3.12 ApplicationImpl

There are currently the following control elements:

1. `KeyboardShutter`. This object listens to the application status. If application becomes active it shuts down the keyboard interception. When application becomes inactive it turns the keyboard interception on.
2. `SessionFlusher`. This object listens to the application status. If application becomes active it flushes intercepted sessions currently stored in the buffer of `SeanceManager`.

4 Debugging

In order to turn on debugging options, you need to define a macro in `TypeingAnalysis.pro` file. Currently the following macros are available:

1. `KEYBOARD_HANDLER_DEBUG`
2. `SEANCE_MANAGER_DEBUG`

When you comment or uncomment a macro in the pro file you will probably need to go through: clear, run qmake, and rebuild sequence.

KEYBOARD_HANDLER_DEBUG This macro activates an additional window showing the data received by the Keyboard Handler of the application. Currently it is connected to key pressing and releasing events of the handler and prints the messages information. The connection is done via Observer pattern 7.3. The debug window closes automatically when the main window is closing.

SEANCE_MANAGER_DEBUG This macro activates an additional window showing the structure of the current Seance of the application. Currently it is connected to the current seance output of the seance manager and prints some information about the seance. The connection is done via Observer pattern 7.3. The debug window closes automatically when the main window is closing.

5 Code

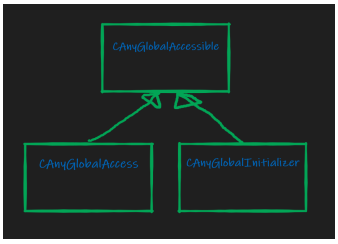
6 Implementation

7 Library

7.1 Singleton

CAnyGlobalAccess template consists of three parts:

1. **CAnyGlobalAccessible**. It provides a static storage for a global object. You do not access it directly.
2. **CAnyGlobalAccess**. This object is used to access the global object. The global object must be initialized before access object is created.
3. **CAnyGlobalInitializer**. This object is used to initialize the global object. You need to create one instance of this object in order to initialize the global object.



Description The pattern is used to make a global object with non-trivial constructor without explicitly defining the object globally. **WARNING** this thing is NOT thread safe! In order to use the pattern we must provide:

1. `TAccessible` – the class for the global object
2. `TID` – identification class. If we want to have several global objects of a class Type, we must distinguish them by a dummy class `TID`. For example, `CAnyGlobalAccessible<Type, A>` and `CAnyGlobalAccessible<Type, B>` store different instances of objects of type Type in static storage. Since static objects defined by the class they belong to, classes `A` and `B` are required to distinguish the instances.

   Example of a dummy class declaration:

   ```
   class CGlobalAccessibleID;
   ```

3. A class for initialization of the global object. It must be inherited from `CAnyGlobalInitializer`. The class must inherit all the constructors of the base class.

   ```
   class CMyInitializer : CAnyGlobalInitializer<TAccessible, TID> {
      using CBase = CAnyGlobalInitializer<TAccessible, TID>;
    public:
      using CBase::CBase;
    };
   ```

4. A class for getting access to the global object must be publicly inherited from `CAnyGlobalAccess`. It has only a default constructor. It asserts if the global object has not yet been initialized.

   ```
   class CMyAccessor : public CAnyGlobalAccess<TAccessible, TID> {};
   ```

Example Suppose we want a global `int` variable for logging:

```
class CLoggerCounterID;
class CLoggerCounterInitializer :
  CAnyGlobalInitializer<int, CLoggerCounterID> {
  using CBase = CAnyGlobalInitializer<int, CLoggerCounterID>;
public:
  using CBase::CBase;
};
class CLoggerCounterAccess : public CAnyGlobalAccess<int, CLoggerCounterID> {};
```

Then in code we write something like that:

```
...
CLoggerCounterInitializer Init(0);
...
CLoggerCounterAccess LogCounter;
++(*LogCounter);
std::cout << *LogCounter << std::endl;
...
```

7.2 AnyObject

7.2.1 AnyMovable

Description `CAnyMovable` allows you to store any movable only class without any restrictions on the class. It also allows you to provide an interface to a class and implementations for different classes of the method.

The `operator->()` goes without any checks and may fail if the object is empty, that is, does not store anything. It is your responsibility to call `isDefined()` method before accessing the interface.

The class `CAnyMovable` does not use Small Object Optimization. In particular, move operations are always cheap. `CAnyMovable` has value semantics and extends the ideas of Sean Parent's talk on cppcon about Run-time Polymorphism.

How to use In order to use the template you need:

1. Create an interface class:

```
template<class TBase>
class IAny : public TBase {
public:
  virtual void print() const = 0;
};
```

   The class describes the interface of an abstract object you want to support. Do not use names with prefix underscore here, e.g.,

BAD: `virtual void _print() const = 0;`
GOOD: `virtual void print() const = 0;`

   This interface will be accessible when using `operator->()` on `CAnyObject`. The example is below.

2. Create an implementation class:

```
template<class TBase, class TObject>
class CAnyImpl : public TBase {
  using CBase = TBase;
public:
  using CBase::CBase;
  void print() const override {
    std::cout << "data = " << CBase::Object() << std::endl;
  }
};
```

   Here you implement all the functions from the interface. The parameter `TObject` is used to reimplement the behaviour for different types of objects if needed. In the example above, it may happen that `TObject` does not support stream `operator<<` and you want to define a specialization of the implementation class. Access to the stored object is provided by the method `CBase::Object()`.

3. Create your Any class:

```
class CAny : public CAnyMovable<IAny, CAnyImpl> {
  using CBase = CAnyMovable<IAny, CAnyImpl>;
public:
  using CBase::CBase;
  friend bool operator==(const CAny&, const CAny&) {
    ...
  }
};
```

   You can add any additional functionality to your `CAny` class.

Now you can use it like this:

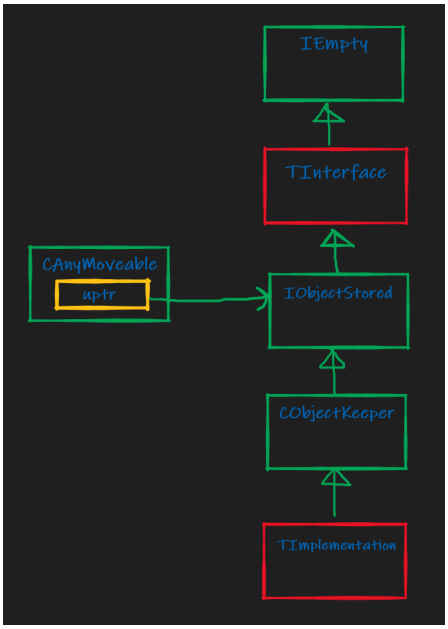
```
CAny x = 'c';
x->print();
x = std::move("123");
x->print();
x = 1.45;
x->print();
```

If you want to store an object of Type `R` and construct it on the fly from the data: `x, y, z`. Then use emplace function or emplace constructor like this:

```
CAny s;
s.emplace<R>(x, y, z);
CAny y(std::in_place_type_t<R>(), x, y, z);
```

Then the object of type `R` will be created without creation of intermediate objects.

Implementation Internally `CAnyMovable` stores an `std::unique_ptr` to interface `IObjectStored`. Let us look at the diagram below:



The red rectangles are templates provided by the user. `IEmpty` is an empty interface with virtual distructor. Its only purpose is to avoid user having to provide a virtual distructor in the `TInterface` template. Then `IObjectStored` adds some virtual methods to provide required functionality. `CObjectKeeper` is a template storing an instance of an object you want to move into `CAnyMovable`. `TImplementation` is a user defined template implementing all virtual methods from `TInterface`.

7.3 Observer

Description The library provides several primitives to use Observer pattern. The most basic primitives are:

1. `CObservable<TData>`
2. `CObserver<TData>`
3. `CSource<TData>`

All primitives are synchronous. Observable notify events return after all its observers perform their actions.

All these objects are addressable. This means you must follow the rules for addressable objects. Usually this means you put this objects into unique pointer or inside other addressable object. Never store addressable objects in a vector or on the stack.

Observable `CObservable<TData>` is an observable value of type `TData`. The Observable need not to contain the data. Instead, it has a method to get access to the data. You may provide such a method in the constructor. `CObservable<TData>` provides the following methods:

1. `subscribe(CObserver<TData>*)`. This method subscribe an observer to the observable. The observable may have many subscribed observers. When you subscribe an observer that is already subscribed, the observer is unsubscribed first (even if this is the same observable). On subscription, every observable is put into the end of the list of subscribers. This is indeed an `std::list<CObserver<TData>*>`.
2. `notify()`. This method notify all subscribed observers about the data change. The subscribers are notified in the order they are presented in the list of the subscribers.

There are three types of notifications from `CObservable<TData>`

1. Notification on subscribe.

2. Notification on calling `notify()`.

3. Notification on unsubscribe.

An observer may and can react differently in all three cases.

Source The observer pattern allows you to pass undefined data via std::optional. If TData is one of the following

1. arithmetic type

2. pointer type

3. enum type

then it is passed by value and all other types are passed by const reference. In this case `isPassedByValue` variable is set to true. In order to deal with these variety of cases in a convenient way, the library provides `CSource<TData>` class. This class is used internally inside a connection between an observer and an observable. An observer get data from observable through an object of type `CSource<TData>`. The source object does not store the data, instead it has an action that returns the data. You may provide the action in the constructor. If you do not provide the action, the source will return undefined data. All methods are safe. You do not need to worry about run-time errors. Undefined data is handled with `std::optional` on a regular basis.

`CSource<TData>` has the following type aliases

1. `CReturnValueType`. If `TData` is passed by value, then `CReturnValueType` is equal `TData` otherwise it coincides with `std::reference_wrapper<const TData>`.

2. `CGetType`. This is the type return by `CSource<TData>` object. It is defined as `std::optional<CReturnValueType>`. This allows to handle the case of no data passing from the observable if the data is not defined at the moment.

3. `CGetSignature`. This is the signature of the action returning a value from the source.

4. `CGetAction`. This is the type of the action returning a value from the source.

`CSource<TData>` has the following methods:

1. `hasGetter`. This check if the source has a getter action returning a value.

2. `operator()` or `get()`. These methods are synonymous and return the value of `CGetType` an observer. This actions are safe to call even if the getter is not defined. In the latter case the result of static `getNothing` action is returned.

3. `set(CGetAction)`. This method allows to change the source getter action.

4. `Getter()`. This method return the getter action by value. It is safe to call this method even if the action is not defined. In this case the static `getNothing` action is returned.

5. `hasValue()`. This method checks if the source returns a well-defined value. It returns true only if the action is defined and the action returns a well-defiled value, that is, the optional indeed has a value.

6. static `getNothing()`. This is a static default getter action returning undefined optional as a result.

Observer `CObserver<TData>` is an observer of a value of type `TData`. The observer contains three actions to react to an observable notifications. You must provide them in the constructor of the observer. Each observer can be subscribed to at most one observable.

`CObserver<TData>` provides the following methods (all methods are safe to call in any state):

1. `isSubscribed()`. Checks if the observer is subscribed to an observable.

2. `hasValue()`. Checks if the observer has a source returning a well-defined value, that is, the observer is subscribed to an observable and the observable provides a well-defined value (the optional result has a value).

3. `data()`. Returns a value from the connection with observable. If not connected, the result is the undefined optional.

4. `Getter()`. Returns the getter action of the connection with an observable. Returns `getNothing` default action if not connected.
5. `unsubscribe()`. Unsubscribe from the current observable if any.
6. `setSubscribe(CMethod)`. This methods changes the action on subscription.
7. `setNotify(CMethod)`. This method changes the action on notification.
8. `setUnsubscribe(CMethod)`. This method chages the action on unsubscription.

Higher level primitives The library provides higher level observable and observers. Currently the following are implemented:

1. `CObservableData<TData>`
2. `CNotifier`
3. `CObserverStrict<TData>`
4. `CObserverHot<TData>`
5. `CObserverCold<TData>`
6. `CObserverHotStrict<TData>`
7. `CHotInput<TData>`
8. `CObserverColdStrict<TData>` or `CColdInput<TData>`
9. `CHotActiveInput<TData>`
10. `CColdActiveInput<TData>`

- `CObservableData<TData>` contains data of type `TData` (the type must be an appropriate type that can be stored in `std::optional`). It has a method to change the data and notifies the observers immediately after the change.
- `CNotifier` this is `CObservable<void>`. It is used when you need to send a dataless signal to an observer.
- `CObserverStrict<TData>` this is an observer reacting to notifications only in case the data is well-defined, that is, when the optional value sent by an observable has a value.
- `CObserverHot<TData>`. This is an observer reacting on subscribe and notify events and doing nothing on unsubscribe event.
- `CObserverCold<TData>`. This is an observer reacting on notify event only and doing nothing otherwise.
- `CObserverHotStrict<TData>`. This observer behaves like the hot observer and reacts only on well-defined data.
- `CHotInput<TData>`. This `CObserverHotStrict<TData>` with the same actions for subscription and notification events.
- `CObserverColdStrict<TData>` or `CColdInput<TData>`. This is cold observer that reacts on well-defined data only.
- `CHotActiveInput<TData>`. This observer can be in two states: active or inactive. It has one action. When the observer is subscribed or notified, if the data from the observable is well-defined, it becomes active and reacts using the action it has. When the observer is unsubscribed it becomes inactive. It also has a method `deactivate()` to make it inactive. You can check its status using `isActive()` method.

  The use case. This input is useful when you want to treat the input as an instance of something. So, when you get a well-defined data the input becomes active, that means it "holds the data from the observable". When you read the data from the input, you set it to inactive state to simulate "the value is consumed" process.

  It is also useful when you need to subscribe different inputs to the same observable and react only once when both inputs get their values.

- `CColdActiveInput<TData>`. This observer can be in two states: active or inactive. It has one action. When the observer is subscribed it does nothing. When observer is notified, if the data from the observable is well-defined, it becomes active and reacts using the action it has. When the observer is unsubscribed it becomes inactive. It also has a method `deactivate()` to make it inactive. You can check its status using `isActive()` method.

The use case is similar to the one for the hot version. It allows you to ignore the subscription event in the similar scenarios.

How to use Here is an example of usage:

```
void print_sub(int x) {
  std::cout << "sub value = " << x << std::endl;
}
void print_not(int x) {
  std::cout << "not value = " << x << std::endl;
}
void print_unsub(int x) {
  std::cout << "unsub value = " << x << std::endl;
}
CObservableData<int> value;
CObserverStrict<int> printer(print_sub, print_not, print_unsub);
value.subscribe(&printer);
value.set(1);
value.set(2);
```

In this example all the objects are created on stack. However, you must not do that. If you need to create an observer or observable, either use `std::unique_ptr` to hold them or create them inside another addressable object.

Remarks (TO DO) Need to implement value semantics wrappers for the primitives to avoid explicit using of `std::unique_ptr` all the time. The wrapper will be a movable object.