# Automated Test Framework

Dmitry Chmerev

March 3, 2015

## Contents

# 1 ATF structure

ATF core is based on events idea. There are event publisher and subscriber objects, a mechanism to connect them, and event queue, containing emitted events.

The core uses Qt signals/slots mechanism for events model.

# 2 Dynamic objects

To have ability to connect to signals, we need an object with dynamic slots. Lua script creates this object, defines its slots in Lua and connects needed signals to these slots. The class is mostly stolen from http://doc.qt.digia.com/qq/qq16-dynamicqobject.html

## 2.1 Declaration

2a ⟨*file:qtdynamic.h* 2a⟩≡ 2b▷

```
#include <QObject>
#include <QMetaObject>
#include <QHash>
#include <QByteArray>
⟨Lua headers 18a⟩
#include "marshal.h"
```

`DynamicSlot` object contains the index of receiver object in the Lua Registry, the list of marshallers (see 3.1) used to marshal data from Qt type system to Lua, and the name of slot.

2b ⟨*file:qtdynamic.h* 2a⟩+≡ ◁2a 2c▷

```
class DynamicSlot
{
 public:
  DynamicSlot(lua_State *L, int objidx, const char* signature);
  void call(QObject *sender, void **arguments);
 private:
  lua_State *lua_state;
  QList<Marshaller*> marshallers_;
  int objidx_;
  QString slot_;
};
```

`DynamicObject` provides a set of functions to connect Qt signals to its slots and vice versa. Sometimes its useful even to connect two `DynamicObjects`, the method to do this is here too.

There is also `emitDynamicSignal` method to emit signals from Lua.

2c ⟨*file:qtdynamic.h* 2a⟩+≡ ◁2b

```
class DynamicObject : public QObject {
 public:
  DynamicObject(QObject *parent);
  virtual int qt_metacall(QMetaObject::Call c, int id, void **arguments);
  bool emitDynamicSignal(const char *signal, void **arguments);
  bool connectDynamicSignal(const char *signal, QObject *obj, const char *slot);
  bool connectDynamicSlot(QObject *obj, const char *signal, const char *slot, DynamicSlot *s);
  static bool connectDynamicSignalToDynamicSlot(
    DynamicObject* sender,
    const char *signal,
    DynamicObject* receiver,
    const char *slot,
```

```
    DynamicSlot *s);
 private:
  QHash<QByteArray, int> slotIndices;
  QList<DynamicSlot *> slotList;
  QHash<QByteArray, int> signalIndices;
};
```

## 2.2 Implementation

3 ⟨*file:qtdynamic.cc* 3⟩≡ 5 ▷

```
#include "qtdynamic.h"
#include <QObject>
#include <QDebug>
⟨Lua headers 18a⟩
DynamicObject::DynamicObject(QObject *parent)
  : QObject(parent) { }

static QList<QByteArray> typesFromString(const char *str, const char *end)
{
  QList<QByteArray> result;
  const char *p = str;
  while (p != end)
  {
    if (*p == ',') ++p;
    const char *b = p;
    while (p != end && *p++ != ',');
    if (p > b)
      result.append(QByteArray(b, p - b));
  }
  return result;
}

static int *queuedConnectionTypes(const QList<QByteArray> &typeNames)
{
    int *types = new int [typeNames.count() + 1];
    Q_CHECK_PTR(types);
    for (int i = 0; i < typeNames.count(); ++i) {
        const QByteArray typeName = typeNames.at(i);
        if (typeName.endsWith('*'))
            types[i] = QMetaType::VoidStar;
        else
            types[i] = QMetaType::type(typeName);

        if (!types[i]) {
            qWarning("QObject::connect: Cannot queue arguments of type '%s'\n"
                     "(Make sure '%s' is registered using qRegisterMetaType().)",
                     typeName.constData(), typeName.constData());
            delete [] types;
            return 0;
        }
    }
    types[typeNames.count()] = 0;

    return types;
}

bool DynamicObject::connectDynamicSlot(QObject *obj, const char *signal, const char *slot,
    DynamicSlot *s)
{
  ⟨Sanity check for connect arguments 4⟩

  int signalId = obj->metaObject()->indexOfSignal(theSignal);
  if (signalId < 0) {
    qWarning() << "No such signal " << theSignal;
    return false;
  }
```

```
  int slotId = slotIndices.value(theSlot, -1);
  if (slotId < 0) {
    slotId = slotList.size();
    slotIndices[theSlot] = slotId;
    slotList.append(s);
  }

  return QMetaObject::connect(obj, signalId,
          this, slotId + metaObject()->methodCount(),
          Qt::QueuedConnection,
          queuedConnectionTypes(typesFromString(theSlot.constData() + theSlot.indexOf('(') + 1,
                                                theSlot.constData() + theSlot.indexOf(')'))));
}

bool DynamicObject::connectDynamicSignal(const char *signal, QObject *obj, const char *slot)
{
  ⟨Sanity check for connect arguments 4⟩

  int slotId = obj->metaObject()->indexOfSlot(theSlot);
  if (slotId < 0) {
    qWarning() << "Cannot find slot " << theSlot;
    return false;
  }

  int signalId = signalIndices.value(theSignal, -1);
  if (signalId < 0) {
      signalId = signalIndices.size();
      signalIndices[theSignal] = signalId;
  }
  return QMetaObject::connect(this, signalId + metaObject()->methodCount(), obj, slotId,
    Qt::QueuedConnection,
    queuedConnectionTypes(typesFromString(theSignal.constData() + theSignal.indexOf('(') + 1,
                                          theSignal.constData() + theSignal.indexOf(')'))));
}

bool DynamicObject::connectDynamicSignalToDynamicSlot(
  DynamicObject* sender,
  const char *signal,
  DynamicObject* receiver,
  const char *slot,
  DynamicSlot *s)
{
  ⟨Sanity check for connect arguments 4⟩

  int signalId = sender->signalIndices.value(theSignal, -1);
  if (signalId < 0) {
      signalId = sender->signalIndices.size();
      sender->signalIndices[theSignal] = signalId;
  }

  int slotId = receiver->slotIndices.value(theSlot, -1);
  if (slotId < 0) {
    slotId = receiver->slotList.size();
    receiver->slotIndices[theSlot] = slotId;
    receiver->slotList.append(s);
  }

  return QMetaObject::connect(sender,
    signalId + sender->metaObject()->methodCount(),
    receiver,
    slotId + receiver->metaObject()->methodCount(),
    Qt::QueuedConnection,
    queuedConnectionTypes(typesFromString(theSignal.constData() + theSignal.indexOf('(') + 1,
                                          theSignal.constData() + theSignal.indexOf(')'))));
}
```

4 ⟨*Sanity check for connect arguments* 4⟩≡ (3)

```
QByteArray theSignal = QMetaObject::normalizedSignature(signal);
QByteArray theSlot   = QMetaObject::normalizedSignature(slot);
```

```
  if (!QMetaObject::checkConnectArgs(theSignal, theSlot)) {
    qWarning() << "Cannot connect signal" << theSignal << "to slot" << theSlot;
    return false;
  }
```

5 ⟨*file:qtdynamic.cc* 3⟩+≡ ◁3

```
  int DynamicObject::qt_metacall(QMetaObject::Call c, int id, void **arguments)
  {
    id = QObject::qt_metacall(c, id, arguments);
    if (id < 0 || c != QMetaObject::InvokeMetaMethod)
        return id;
    Q_ASSERT(id < slotList.size());

    slotList[id]->call(sender(), arguments);
    return -1;
  }

  bool DynamicObject::emitDynamicSignal(const char *signal, void **arguments)
  {
    QByteArray theSignal = QMetaObject::normalizedSignature(signal);
    int signalId = signalIndices.value(theSignal, -1);
    if (signalId >= 0) {
        QMetaObject::activate(this, metaObject(), signalId + metaObject()->methodCount(),
            arguments);
        return true;
    } else {
        return false;
    }
  }

  DynamicSlot::DynamicSlot(lua_State *L, int objidx, const char *signature)
    : lua_state(L),
      objidx_(objidx)
  {
    QByteArray theSignal = QMetaObject::normalizedSignature(signature);
    marshallers_ = get_marshalling_list(theSignal);
    QByteArray slotName = signature;
    slot_ = QString(QByteArray(slotName, slotName.indexOf('(')));
  }

  void DynamicSlot::call(QObject *sender, void **arguments)
  {
    (void)sender;
    lua_rawgeti(lua_state, LUA_REGISTRYINDEX, objidx_);
    lua_getfield(lua_state, -1, slot_.toUtf8().constData());
    if (lua_isfunction(lua_state, -1)) {
      lua_pushnil(lua_state);
      lua_copy(lua_state, -3, -1);
      ++arguments;  // skip return value
      int argc = 1;
      for (auto m : marshallers_) {
        m->Unmarshal(*arguments++, lua_state);
        ++argc;
      }
      lua_call(lua_state, argc, 0);
    } else {
      lua_pop(lua_state, 1); // Remove slot from stack
    }
    lua_pop(lua_state, 1);   // Remove object from stack
  }
```

# 3 Qt to Lua interaction

To provide `connect` function to Lua I made the global userdata object `qt` with 3 methods:

- `dynamic()`
- `connect(sender: qobject, signal: string, receiver: qobject, slot: string)`
- `disconnect(...)`

6a ⟨*file:qtlua.h* 6a⟩≡
```
#pragma once
⟨Lua headers 18a⟩
int luaopen_qt(lua_State *L);
```

6b ⟨*file:qtlua.cc* 6b⟩≡ 6c▷
```
#include <QObject>
⟨Lua headers 18a⟩
#include "qtlua.h"
#include "qtdynamic.h"
#include "marshal.h"
#include <QMetaObject>
#include <QDebug>
static int qtlua_createdynamic(lua_State *L);
static int qtlua_connect(lua_State *L);
static int qtlua_disconnect(lua_State *L);
static int dynamicTableId = 0; // Registry index of table used to mark dynamic userdata

int luaopen_qt(lua_State *L) {
  lua_newtable(L);
  dynamicTableId = luaL_ref(L, LUA_REGISTRYINDEX);
  luaL_Reg functions[] = {
    { "dynamic", &qtlua_createdynamic },
    { "connect", &qtlua_connect },
    { "disconnect", &qtlua_disconnect },
    { NULL, NULL }
  };
  luaL_newlib(L, functions);
  return 1;
}
```

Only `DynamicObject` objects may participate in signal/slot connections. To distinguish dynamic objects from others, I use the special value - empty table registered in Lua registry.

6c ⟨*file:qtlua.cc* 6b⟩+≡ ◁6b 7a▷
```
int qtlua_deletedynamic(lua_State *L) {
  DynamicObject * li = *static_cast<DynamicObject**>(lua_touserdata(L, 1));
  delete li;
  return 0;
}
int qtlua_createdynamic(lua_State *L) {
  lua_getglobal(L, "interp");  // QObject LuaInterpreter must have been saved in global 'interp'
  QObject * interp = *(static_cast<QObject**>(lua_touserdata(L, -1)));
  QObject **p =  static_cast<QObject**>(lua_newuserdata(L, sizeof(QObject*)));
  *p = new DynamicObject(interp);
  // Set metatable with "__index" and "__newindex" set to one empty table
  lua_newtable(L); // metatable
  lua_newtable(L); // __index and __newindex table
  lua_pushnil(L);
  lua_copy(L, -2, -1);
  lua_setfield(L, -3, "__index");
  lua_setfield(L, -2, "__newindex");
```

```
    lua_pushcfunction(L, &qtlua_deletedynamic);
    lua_setfield(L, -2, "__gc");
    lua_setmetatable(L, -2);
    // Mark userdatum with the specific uservalue
    lua_rawgeti(L, LUA_REGISTRYINDEX, dynamicTableId);
    lua_setuservalue(L, -2);
    return 1;
  }
```

7a ⟨*file:qtlua.cc* 6b⟩+≡ ◁6c 7b▷

```
  // Helper function, pushes onto top of stack
  // __index value of metatable of value on given index
  // If value doesn't have metatable or there is no __index field,
  // function creates them.
  static void get_index_table(lua_State *L, int idx);
  // Function emits the given signal of dynamic object.
  // It takes two  upvalues: signal name and list of argument types to marshal them properly
  static int qtlua_emit_signal(lua_State *L);
```

There is no use to connect a dynamic signal to a dynamic slot, so I prohibited more than one dynamic object in `connect` call. Connecting dynamic signal to a real slot should result in appearance of new method in the dynamic object, named as the signal, that emits new signal on call. Connecting a real signal to a dynamic slot is similar, but in this case no public methods added, instead the invisible slot that calls the existing method is added.

7b ⟨*file:qtlua.cc* 6b⟩+≡ ◁7a 8a▷

```
  int qtlua_connect(lua_State *L) {
    QObject *sender, *receiver;
    bool senderIsDynamic = false;
    bool receiverIsDynamic = false;
    const char *signal = luaL_checkstring(L, 2);
    const char *slot = luaL_checkstring(L, 4);

    sender = *static_cast<QObject**>(lua_touserdata(L, 1));
    receiver = *static_cast<QObject**>(lua_touserdata(L, 3));
    if (!sender) {
      return luaL_error(L, "connect: sender must be a qt object");
    }
    if (!receiver) {
      return luaL_error(L, "connect: receiver must be a qt object");
    }

    // We use uservalue bound to the userdata to identify dynamic qobjects
    lua_getuservalue(L, 1);
    lua_getuservalue(L, 3);
    lua_rawgeti(L, LUA_REGISTRYINDEX, dynamicTableId);
    senderIsDynamic = lua_rawequal(L, -1, -3);
    receiverIsDynamic = lua_rawequal(L, -1, -2);
    lua_pop(L, 3);

    if (senderIsDynamic && receiverIsDynamic) {
      DynamicObject * d_sender = static_cast<DynamicObject*>(sender);
      DynamicObject * d_receiver = static_cast<DynamicObject*>(receiver);

      ⟨Push signal emitter function 8b⟩
      ⟨Register the receiver 8e⟩
      bool res = DynamicObject::connectDynamicSignalToDynamicSlot(d_sender,
        signal,
        d_receiver,
        slot,
        new DynamicSlot(L, objref, slot));
      lua_pushboolean(L, res);
    } else if (senderIsDynamic) {
```

When connecting a dynamic signal to a real Qt object's slot, I must know when this object is destroyed to disconnect it. To achieve this, I connect to

`QObject::destroyed()` signal. See ⟨*Disconnecting dynamic signal* 25b⟩.

8a ⟨*file:qtlua.cc* 6b⟩+≡ ◁7b 8c▷
```
DynamicObject * d_sender = static_cast<DynamicObject*>(sender);
bool res = d_sender->connectDynamicSignal(signal, receiver, slot);

⟨Push signal emitter function 8b⟩
```

8b ⟨*Push signal emitter function* 8b⟩≡ (7b 8a)
```
QByteArray theSignal = QMetaObject::normalizedSignature(signal);

get_index_table(L, 1);
// TODO: check if this function already is bound
// Add signal emitter function
//  Push signal name (upvalue 1)
lua_pushstring(L, theSignal);
//  Push marshallers table (upvalue 2)
auto marshallers = get_marshalling_list(theSignal);
lua_newtable(L);
int i = 1;  // In Lua 1-based arrays are common
for (auto m : marshallers) {
  lua_pushlightuserdata(L, m);
  lua_rawseti(L, -2, i++);
}
//  Push emitter
lua_pushcclosure(L, &qtlua_emit_signal, 2);
int idx = theSignal.indexOf('(');
theSignal.truncate(idx);
lua_setfield(L, -2, theSignal);
```

8c ⟨*file:qtlua.cc* 6b⟩+≡ ◁8a 8d▷
```
⟨Disconnecting dynamic signal 25b⟩
lua_pushboolean(L, res);
} else if (receiverIsDynamic) {
  DynamicObject * d_receiver = static_cast<DynamicObject*>(receiver);
```

I need the receiver to stay alive even when the only reference to it is destroyed, so I register it in the Lua Registry. But when the last signal is unconnected the object must be unregistered. See ⟨*Unreference dynamic object* (never defined)⟩.

8d ⟨*file:qtlua.cc* 6b⟩+≡ ◁8c 8f▷
```
⟨Register the receiver 8e⟩
```

8e ⟨*Register the receiver* 8e⟩≡ (7b 8d)
```
// Register the receiver object in the registry and store its index
lua_pushnil(L);
lua_copy(L, 3, -1);
int objref = luaL_ref(L, LUA_REGISTRYINDEX);
```

8f ⟨*file:qtlua.cc* 6b⟩+≡ ◁8d 8g▷
```
bool res = d_receiver->connectDynamicSlot(sender, signal, slot,
    new DynamicSlot(L, objref, slot));
lua_pushboolean(L, res);
} else {
  bool res = QObject::connect(sender, signal, receiver, slot);
  lua_pushboolean(L, res);
}
return 1;
}
```

This helper routine pushes `getmetatable(x)"[__index"]` value on stack. If argument `x` doesn't have metatable or metatable doesn't have `__index` field, routine creates them.

8g ⟨*file:qtlua.cc* 6b⟩+≡ ◁8f 11b▷
```
void get_index_table(lua_State *L, int idx)
{
  lua_getmetatable(L, idx);
```

```
  if (lua_isnil(L, -1)) {
    lua_newtable(L);
    lua_copy(L, -1, -2);
    lua_setmetatable(L, idx > 0 ? idx : idx - 2);
  }
  lua_getfield(L, -1, "__index");
  if (lua_isnil(L, -1)) {
    lua_pop(L, 1);
    lua_newtable(L);
    lua_pushnil(L);
    lua_copy(L, -2, -1);
    lua_setfield(L, -3, "__index");
  }
  lua_remove(L, -2);
}
```

## 3.1 Arguments marshalling

To properly emit a signal from Lua code, we have to marshal Lua arguments to C++. I use a list of functions converting input Lua argument to C++ object. When Lua script emits a signal, these functions are run consequently to get array of arguments ready to be passed to the C++ signal.

9a ⟨*file:marshal.h* 9a⟩≡

```
#pragma once
⟨Lua headers 18a⟩
#include <QList>
#include <QMap>
#include <QString>

class Marshaller
{
 public:
  static Marshaller *get(const QString& type);
  virtual void* Marshal(lua_State *L, int index) = 0;
  virtual void Dispose(void *obj) = 0;
  virtual void Unmarshal(void *obj, lua_State *L) = 0;
};
// These functions create lists of marshallers
// to bind them to signal emitters and slot callers
QList<Marshaller*> get_marshalling_list(const char* signature);
```

9b ⟨*file:marshal.cc* 9b⟩≡ 9c▷

```
#include "marshal.h"
⟨Lua headers 18a⟩
#include <QMap>
#include <QList>
#include <QString>
```

For convinience we have the function returning marshallers list by a signal signature.

9c ⟨*file:marshal.cc* 9b⟩+≡ ◁9b 10a▷

```
QList<Marshaller*> get_marshalling_list(const char* signature)
{
  QList<Marshaller*> retval;
  char buffer[80];
  // Assumed that signature is normalized
  const char *p = signature;
  while (*p && *p++ != '(');
  while (*p && *p != ')') {
    char *b = buffer;
    while (*p && *p != ',' && *p != ')') {
      *b++ = *p++;
    }
    *b = '\0';
```

```
      if (*p == ',') ++p;
      retval.append(Marshaller::get(buffer));
    }
    return retval;
  }
```

A set of predefined marshallers

10a ⟨*file:marshal.cc* 9b⟩+≡ ◁9c 10b▷

```
  namespace {
    // int Marshaller
    class : public Marshaller {
     public:
      void* Marshal(lua_State *L, int index) {
        int isnum = 0;
        int val = lua_tointegerx(L, index, &isnum);
        if (!isnum)
          return NULL;
        return new int(val);
      }
      void Dispose(void *obj) {
        delete static_cast<int*>(obj);
      }
      void Unmarshal(void *obj, lua_State *L) {
        lua_pushinteger(L, *static_cast<int*>(obj));
      }
    } intMarshaller;
```

The rest are defined the same way.

10b ⟨*file:marshal.cc* 9b⟩+≡ ◁10a 11a▷

```
    // QString Marshaller
    class : public Marshaller {
     public:
      void* Marshal(lua_State *L, int index) {
        const char * val = lua_tostring(L, index);
        if (!val)
          return NULL;
        return new QString(val);
      }
      void Dispose(void *obj) {
        delete static_cast<QString*>(obj);
      }
      void Unmarshal(void *obj, lua_State *L) {
        lua_pushstring(L, static_cast<QString*>(obj)->toUtf8().constData());
      }
    } QStringMarshaller;
    // QByteArray Marshaller
    class : public Marshaller {
     public:
      void* Marshal(lua_State *L, int index) {
        size_t size;
        const char * val = lua_tolstring(L, index, &size);
        if (!val)
          return NULL;
        return new QByteArray(val, size);
      }
      void Dispose(void *obj) {
        delete static_cast<QByteArray*>(obj);
      }
      void Unmarshal(void *obj, lua_State *L) {
        QByteArray *ba = static_cast<QByteArray*>(obj);
        lua_pushlstring(L, ba->constData(), ba->size());
      }
    } QByteArrayMarshaller;
    // bool Marshaller
    class : public Marshaller {
     public:
      void* Marshal(lua_State *L, int index) {
        bool val = lua_toboolean(L, index);
```

```
      return new bool(val);
    }
    void Dispose(void *obj) {
      delete static_cast<bool*>(obj);
    }
    void Unmarshal(void *obj, lua_State *L) {
      lua_pushboolean(L, *static_cast<bool*>(obj));
    }
  } boolMarshaller;
```

11a ⟨*file:marshal.cc* 9b⟩+≡ ◁10b

```
  QMap<QString, Marshaller*> marshallers = {
    { "int", &intMarshaller },
    { "bool", &boolMarshaller },
    { "QString", &QStringMarshaller },
    { "QByteArray", &QByteArrayMarshaller }
  };
} // anonymous namespace

Marshaller* Marshaller::get(const QString& type)
{
  return marshallers[type];
}
```

## 3.2 Signals emitting

I bind one simple closure to every dynamic signal added. It goes without saying, the only difference between signals is in upvalues. Signal emitter closure has two upvalues: the signal name, necessary to find a signal, and the `Marshaller` objects list. The latter is used to construct C++ arguments from the values on Lua stack.

11b ⟨*file:qtlua.cc* 6b⟩+≡ ◁8g 11c▷

```
int qtlua_emit_signal(lua_State *L)
{
  DynamicObject * li = *static_cast<DynamicObject**>(lua_touserdata(L, 1));
  const char* theSignal = lua_tostring(L, lua_upvalueindex(1));
  int mid = lua_upvalueindex(2); // Marshallers table pseudoindex
  lua_len(L, mid);
  int msize = lua_tointegerx(L, -1, NULL);
  lua_pop(L, 1);
  void *stackargs[8];
  void **args = stackargs;
  if (msize > 8) {
    args = new void*[msize];
  }
  args[0] = NULL;
  for (int i = 1; i <= msize; ++i) {
    lua_rawgeti(L, mid, i);
    auto marshaller = static_cast<Marshaller*>(lua_touserdata(L, -1));
    lua_pop(L, 1);
    args[i] = marshaller->Marshal(L, i + 1);
  }
  li->emitDynamicSignal(theSignal, args);
  if (args != stackargs) {
    delete[] args;
  }
  return 0;
}
```

11c ⟨*file:qtlua.cc* 6b⟩+≡ ◁11b

```
int qtlua_disconnect(lua_State *L)
{
  (void)L;
  return 0;
}
```

# 4 Networking

All functions here just wraps corresponding `TcpSocket` and `TcpServer` functions.

12a ⟨*file:network.h* 12a⟩≡
```
#pragma once
⟨Lua headers 18a⟩
#include <QObject>
#include <QAbstractSocket>
#include <QTcpSocket>
#include <QTcpServer>
int luaopen_network(lua_State *L);
```

12b ⟨*file:network.cc* 12b⟩≡ 12c▷
```
#include "network.h"

#include <QAbstractSocket>
#include <QTcpSocket>
#include <QTcpServer>
#include <QWebSocket>
```

## 4.1 Connection classes

`TcpClient` class provides methods to connect a socket, send and receive data from socket and close it.

12c ⟨*file:network.cc* 12b⟩+≡ ◁12b 13b▷
```
// TcpClient functions
int network_tcp_client(lua_State *L) {
  QTcpSocket  *tcpSocket = new QTcpSocket();
  QTcpSocket **p = static_cast<QTcpSocket**>(lua_newuserdata(L, sizeof(QTcpServer*)));
  *p = tcpSocket;
  luaL_getmetatable(L, "network.TcpSocket");
  lua_setmetatable(L, -2);
  return 1;
}
int tcp_socket_connect(lua_State *L) {
  ⟨Get TcpSocket object 13a⟩
  const char* ip   = luaL_checkstring(L, 2);
  int         port = luaL_checkinteger(L, 3);
  tcpSocket->connectToHost(ip, port);
  return 0;
}
int tcp_socket_read(lua_State *L) {
  ⟨Get TcpSocket object 13a⟩
  int maxSize = luaL_checkinteger(L, 2);
  QByteArray result = tcpSocket->read(maxSize);
  lua_pushlstring(L, result.data(), result.count());
  return 1;
}
int tcp_socket_write(lua_State *L) {
  ⟨Get TcpSocket object 13a⟩
  size_t size;
  const char* data = luaL_checklstring(L, 2, &size);
  int result = tcpSocket->write(data, size);
  lua_pushinteger(L, result);
  return 1;
}
int tcp_socket_close(lua_State *L) {
  ⟨Get TcpSocket object 13a⟩
  tcpSocket->close();
  return 0;
}
int tcp_socket_delete(lua_State *L) {
  ⟨Get TcpSocket object 13a⟩
```

```
  delete tcpSocket;
  return 0;
}
```

13a ⟨*Get TcpSocket object* 13a⟩≡ (12c)

```
QTcpSocket *tcpSocket =
  *static_cast<QTcpSocket**>(luaL_checkudata(L, 1, "network.TcpSocket"));
```

`TcpServer` and `WebSocket` objects are defined the same way.

13b ⟨*file:network.cc* 12b⟩+≡ ◁12c 13d▷

```
// TcpServer functions
int network_tcp_server(lua_State *L) {
  QTcpServer *tcpServer = new QTcpServer();
  QTcpServer **p = static_cast<QTcpServer**>(lua_newuserdata(L, sizeof(QTcpServer*)));
  *p = tcpServer;
  luaL_getmetatable(L, "network.TcpServer");
  lua_setmetatable(L, -2);
  return 1;
}
int tcp_server_listen(lua_State *L)
{
  ⟨Get TcpServer object 13c⟩
  const char * ip = luaL_checkstring(L, 2);
  QHostAddress addr(ip);
  int port = luaL_checkinteger(L, 3);
  lua_pushboolean(L, tcpServer->listen(QHostAddress::Any, port));

  return 1;
}
int tcp_server_get_connection(lua_State *L)
{
  ⟨Get TcpServer object 13c⟩
  QTcpSocket *tcpSocket = tcpServer->nextPendingConnection();
  if (tcpSocket) {
    QTcpSocket **p = static_cast<QTcpSocket**>(lua_newuserdata(L, sizeof(QTcpServer*)));
    *p = tcpSocket;
    luaL_getmetatable(L, "network.TcpSocket");
    lua_setmetatable(L, -2);
  } else {
    lua_pushnil(L);
  }

  return 1;
}
int tcp_server_delete(lua_State *L) {
  ⟨Get TcpServer object 13c⟩
  delete tcpServer;
  return 0;
}
```

13c ⟨*Get TcpServer object* 13c⟩≡ (13b)

```
QTcpServer *tcpServer =
  *static_cast<QTcpServer**>(luaL_checkudata(L, 1, "network.TcpServer"));
```

13d ⟨*file:network.cc* 12b⟩+≡ ◁13b 14b▷

```
// WebSocket functions
int network_web_socket(lua_State *L) {
  QWebSocket *webSocket = new QWebSocket();
  QWebSocket **p = static_cast<QWebSocket**>(lua_newuserdata(L, sizeof(QWebSocket*)));
  *p = webSocket;
  luaL_getmetatable(L, "network.WebSocket");
  lua_setmetatable(L, -2);
  return 1;
}
int web_socket_open(lua_State *L) {
```

```
    ⟨Get WebSocket object 14a⟩
    QUrl url(luaL_checkstring(L, 2));
    url.setPort(lua_tointegerx(L, 3, NULL));
    webSocket->open(url);
    return 0;
  }
  int web_socket_close(lua_State *L) {
    ⟨Get WebSocket object 14a⟩
    webSocket->close();
    return 0;
  }
  int web_socket_write(lua_State *L) {
    ⟨Get WebSocket object 14a⟩
    size_t size;
    const char* data = luaL_checklstring(L, 2, &size);
    QByteArray b(data, size);
    int res = webSocket->sendTextMessage(b);
    lua_pushinteger(L, res);
    return 1;
  }

  int web_socket_delete(lua_State *L) {
    ⟨Get WebSocket object 14a⟩
    delete webSocket;
    return 0;
  }
```

14a ⟨Get WebSocket object 14a⟩≡ (13d)

```
  QWebSocket *webSocket =
    *static_cast<QWebSocket**>(luaL_checkudata(L, 1, "network.WebSocket"));
```

## 4.2 Network module initialization

14b ⟨file:network.cc 12b⟩+≡ ◁13d

```
  int luaopen_network(lua_State *L) {
    lua_newtable(L);

    // TcpServer metatable
    luaL_newmetatable(L, "network.TcpServer");

    lua_newtable(L);
    luaL_Reg tcp_server_functions[] = {
      { "listen", &tcp_server_listen },
      { "get_connection", &tcp_server_get_connection },
      { NULL, NULL }
    };
    luaL_setfuncs(L, tcp_server_functions, 0);
    lua_setfield(L, -2, "__index");
    lua_pushcfunction(L, tcp_server_delete);
    lua_setfield(L, -2, "__gc");
    // TcpSocket metatable
    luaL_newmetatable(L, "network.TcpSocket");

    lua_newtable(L);
    luaL_Reg tcp_socket_functions[] = {
      { "connect", &tcp_socket_connect },
      { "read", &tcp_socket_read },
      { "write", &tcp_socket_write },
      { "close", &tcp_socket_close },
      { NULL, NULL }
    };
    luaL_setfuncs(L, tcp_socket_functions, 0);

    lua_setfield(L, -2, "__index");
    lua_pushcfunction(L, tcp_socket_delete);
    lua_setfield(L, -2, "__gc");
```

```
  // WebSocket metatable
  luaL_newmetatable(L, "network.WebSocket");
  lua_newtable(L);
  luaL_Reg web_socket_functions[] = {
    { "open", &web_socket_open },
    { "close", &web_socket_close },
    { "write", &web_socket_write },
    { NULL, NULL }
  };
  luaL_setfuncs(L, web_socket_functions, 0);
  lua_setfield(L, -2, "__index");
  lua_pushcfunction(L, web_socket_delete);
  lua_setfield(L, -2, "__gc");

  luaL_Reg network_functions[] = {
    { "TcpClient", &network_tcp_client },
    { "TcpServer", &network_tcp_server },
    { "WebSocket", &network_web_socket },
    { NULL, NULL }
  };
  luaL_newlib(L, network_functions);
  return 1;
}
```

# 5 Timer

Timer object is used to raise periodic signals.

16a ⟨*file:timers.h* 16a⟩≡

```
#pragma once
⟨Lua headers 18a⟩
#include <QObject>

int luaopen_timers(lua_State *L);
```

16b ⟨*file:timers.cc* 16b⟩≡

```
#include "timers.h"
#include <QTimer>

int timer_create(lua_State *L) {
  QTimer **p = static_cast<QTimer**>(lua_newuserdata(L, sizeof(QTimer*)));
  *p = new QTimer();
  luaL_getmetatable(L, "timers.Timer");
  lua_setmetatable(L, -2);
  return 1;
}

int timer_start(lua_State *L) {
  QTimer *timer =
    *static_cast<QTimer**>(luaL_checkudata(L, 1, "timers.Timer"));
  if (lua_isnumber(L, 2)) {
    int msec = lua_tonumberx(L, 2, NULL);
    timer->start(msec);
  } else {
    timer->start();
  }
  return 0;
}

int timer_stop(lua_State *L) {
  QTimer *timer =
    *static_cast<QTimer**>(luaL_checkudata(L, 1, "timers.Timer"));
  timer->stop();
  return 0;
}

int timer_set_interval(lua_State *L) {
  QTimer *timer =
    *static_cast<QTimer**>(luaL_checkudata(L, 1, "timers.Timer"));
  int msec = luaL_checknumber(L, 2);
  timer->setInterval(msec);
  return 0;
}

int timer_delete(lua_State *L) {
  QTimer *timer =
    *static_cast<QTimer**>(luaL_checkudata(L, 1, "timers.Timer"));
  delete timer;
  return 0;
}

int luaopen_timers(lua_State *L) {
  lua_newtable(L);

  luaL_newmetatable(L, "timers.Timer");

  lua_newtable(L);
  luaL_Reg timer_functions[] = {
    { "start", &timer_start },
    { "stop", &timer_stop },
    { "setInterval", &timer_set_interval },
    { NULL, NULL }
  };
```

```
  luaL_setfuncs(L, timer_functions, 0);
  lua_setfield(L, -2, "__index");
  lua_pushcfunction(L, timer_delete);
  lua_setfield(L, -2, "__gc");

  luaL_Reg timers_functions[] = {
    { "Timer", &timer_create },
    { NULL, NULL }
  };
  luaL_newlib(L, timers_functions);
  return 1;
}
```

# 6 Runtime

Here is the very common chunks of code, used by most of sources.

18a ⟨*Lua headers* 18a⟩≡ (2a 3 6 9 12a 16a 18 19a 21)

```
extern "C" {
#include <lua5.2/lua.h>
#include <lua5.2/lualib.h>
#include <lua5.2/lauxlib.h>
}
```

## 6.1 Main program

It's quite simple. Just take command line arguments, create `LuaInterpeter` object and run event loop.

18b ⟨*file:main.cc* 18b⟩≡

```
⟨Lua headers 18a⟩
#include <QObject>
#include <QCoreApplication>
#include <assert.h>
#include "lua_interpreter.h"
#include <iostream>

int main(int argc, char** argv)
{
  if (argc < 2) {
    std::cerr << "Path to Lua script needed" << std::endl;
    return 1;
  }

  QCoreApplication app(argc, argv);

  LuaInterpreter lua_interpreter(&app);
  int res = lua_interpreter.load(argv[1]);
  if (res) {
    return res;
  }
  if (lua_interpreter.quitCalled) {
    return lua_interpreter.retCode;
  }
  return app.exec();
}
```

## 6.2 Lua interpreter

This class instance is use to be parent for all Qt objects created by ATF, so it inherits `QObject`.

18c ⟨*file:lua_interpreter.h* 18c⟩≡

```
#pragma once
⟨Lua headers 18a⟩
#include <QObject>
#include <QMetaObject>
#include <QHash>
#include <QByteArray>

class LuaInterpreter : public QObject {
  Q_OBJECT
 private:
  lua_State* lua_state;
  int testObject;
 public:
  bool quitCalled = false;
```

```
  int retCode = 0;
  LuaInterpreter(QObject *parent);
  int load(const char *filename);
 public slots:
  void quit();
 public:
  ~LuaInterpreter();
};
```

19a ⟨*file:lua_interpreter.cc* 19a⟩≡ 19b▷

```
#include <time.h> // for clock_gettime
#include "lua_interpreter.h"
#include "qtdynamic.h"
#include "network.h"
#include "timers.h"
#include "qtlua.h"
#include <assert.h>
#include <iostream>
#include <stdexcept>
⟨Lua headers 18a⟩
#include <QObject>
#include <QTimer>
#include <QCoreApplication>

namespace {
```

`app_quit` function may be called before application event loop is started, so I had to add some tricks to not to start this loop if `app_quit` was called (`quitCalled` boolean flag indicates it).

19b ⟨*file:lua_interpreter.cc* 19a⟩+≡ ◁19a 20▷

```
int app_quit(lua_State *L) {
  lua_getglobal(L, "interp");
  LuaInterpreter *li = *static_cast<LuaInterpreter**>(lua_touserdata(L, -1));
  li->quitCalled = true;
  if (lua_isnumber(L, 1)) {
    li->retCode = lua_tointegerx(L, 1, NULL);
  }
  QCoreApplication::exit(li->retCode);
  return 0;
}

int timestamp(lua_State *L) {
  struct timespec ts;
  clock_gettime(CLOCK_MONOTONIC, &ts);
  lua_pushnumber(L, ts.tv_sec * 1000 + ts.tv_nsec / 1000000);
  return 1;
}
}  // anonymous namespace
LuaInterpreter::LuaInterpreter(QObject *parent)
  : QObject(parent) {
  lua_state = luaL_newstate();

  luaL_requiref(lua_state, "base", &luaopen_base, 1);
  luaL_requiref(lua_state, "package", &luaopen_package, 1);
  luaL_requiref(lua_state, "network", &luaopen_network, 1);
  luaL_requiref(lua_state, "timers", &luaopen_timers, 1);
  luaL_requiref(lua_state, "string", &luaopen_string, 1);
  luaL_requiref(lua_state, "table", &luaopen_table, 1);
  luaL_requiref(lua_state, "debug", &luaopen_debug, 1);
  luaL_requiref(lua_state, "math", &luaopen_math, 1);
  luaL_requiref(lua_state, "io", &luaopen_io, 1);
  luaL_requiref(lua_state, "bit32", &luaopen_bit32, 1);
  luaL_requiref(lua_state, "qt", &luaopen_qt, 1);
```

I'd like to locate my extensions in the "modules" subdirectory and name them as it common for libraries — starting with "lib", but don't want Lua interpreter

to look for “luaopen_libmodule” procedure.

20 ⟨*file:lua_interpreter.cc* 19a⟩+≡ ◁19b

```
    // extend package.cpath
    lua_getglobal(lua_state, "package");
    assert(!lua_isnil(lua_state, -1));
    lua_getfield(lua_state, -1, "cpath");
    assert(!lua_isnil(lua_state, -1));
    lua_pushstring(lua_state, ";./modules/lib?.so;./lib?.so");
    lua_concat(lua_state, 2);
    lua_setfield(lua_state, -2, "cpath");

    lua_getfield(lua_state, -1, "path");
    assert(!lua_isnil(lua_state, -1));
    lua_pushstring(lua_state, ";./modules/?.lua");
    lua_concat(lua_state, 2);
    lua_setfield(lua_state, -2, "path");

    lua_pushcfunction(lua_state, &app_quit);
    lua_setglobal(lua_state, "quit");

    lua_pushcfunction(lua_state, &timestamp);
    lua_setglobal(lua_state, "timestamp");

    // Adding global 'interp'
    QObject **p = static_cast<QObject**>(lua_newuserdata(lua_state, sizeof(QObject*)));
    *p = this;
    lua_setglobal(lua_state, "interp");
  }
  int LuaInterpreter::load(const char *filename) {

    int res = luaL_dofile(lua_state, filename);
    if (res != 0) {
      std::cerr << lua_tostring(lua_state, -1) << std::endl;
    }
    return quitCalled ? retCode : res;
  }
  void LuaInterpreter::quit() {
    QCoreApplication::quit();
  }

  LuaInterpreter::~LuaInterpreter() {
    lua_close(lua_state);
  }
```

# 7 Tests

The tests are performed every build iteration to verify that nothing was broken during last modification.

## 7.1 `qt.connect` tests

In this test case I check if `qtlua_connect` function works correctly. I need to check three cases:

- Connect dynamic signal to a normal slot
- Connect normal signal to dynamic slot
- Connect normal signal and slot

To do this, I need a way to create test Qt objects. I've put them to the dynamic library `qttest.so`. This library is loaded from within Lua test script using `require` keyword.

21 ⟨*file:test/qttest.cc* 21⟩≡

```
⟨Lua headers 18a⟩
#include <QObject>
#include <QDebug>
#include "object1.h"

// Creates a Qt object with a set of various signals
static int qttest_mkObject1(lua_State *L) {
  qDebug() << "test.Object1 ctor";
  lua_getglobal(L, "interp"); // LuaInterpreter obejct must have been saved in global 'interp'
  QObject * interp = *(static_cast<QObject**>(lua_touserdata(L, -1)));
  QObject **p = static_cast<QObject**>(lua_newuserdata(L, sizeof(QObject*)));
  *p = new TestObject1(interp);
  luaL_getmetatable(L, "test.Object1");
  lua_setmetatable(L, -2);
  return 1;
}

static int qttest_deleteObject1(lua_State *L) {
  qDebug() << "test.Object1 __gc metamethod";
  TestObject1 *obj = *(static_cast<TestObject1**>(lua_touserdata(L, 1)));
  delete obj;
}

static int qttest_Object1_raiseSignal(lua_State *L) {
  qDebug() << "test.Object1 raiseSignal";
  TestObject1 *obj = *(static_cast<TestObject1**>(lua_touserdata(L, 1)));
  obj->raiseSignal();
  return 0;
}

static int qttest_Object1_raiseStringSignal(lua_State *L) {
  qDebug() << "test.Object1 raiseStringSignal";
  TestObject1 *obj = *(static_cast<TestObject1**>(lua_touserdata(L, 1)));
  QString s = lua_tostring(L, 2);
  obj->raiseStringSignal(s);
  return 0;
}

extern "C"
int luaopen_qttest(lua_State *L) {
  // Object1 metatable
  luaL_newmetatable(L, "test.Object1");
```

```
  lua_newtable(L);
  luaL_Reg object1_functions[] =
  {
    { "raiseSignal", &qttest_Object1_raiseSignal },
    { "raiseStringSignal", &qttest_Object1_raiseStringSignal },
    { NULL, NULL }
  };
  luaL_setfuncs(L, object1_functions, 0);

  lua_setfield(L, -2, "__index");

  lua_pushcfunction(L, qttest_deleteObject1);
  lua_setfield(L, -2, "__gc");

  luaL_Reg qttest_functions[] = {
    { "Object1", qttest_mkObject1 },
    { NULL, NULL }
  };

  luaL_newlib(L, qttest_functions);

  return 1;
}
```

22a ⟨*file:test/object1.h* 22a⟩≡

```
#pragma once

#include <QObject>
#include <QDebug>

class TestObject1 : public QObject
{
  Q_OBJECT
  public:
  TestObject1(QObject *parent)
    : QObject(parent) { }
  ~TestObject1() {
    qDebug() << "TestObject::dtor";
  }
  signals:
  void Signal();
  void StringSignal(QString s);
  public slots:
    void Slot();
    void StringSlot(QString);
  public:
  void raiseSignal();
  void raiseStringSignal(QString s);
};
```

22b ⟨*file:test/object1.cc* 22b⟩≡

```
#include "object1.h"

void TestObject1::raiseSignal() {
  emit Signal();
}

void TestObject1::raiseStringSignal(QString s) {
  emit StringSignal(s);
}

void TestObject1::Slot() {
  qDebug() << "TestObject1::Slot()";
}

void TestObject1::StringSlot(QString s) {
  qDebug() << "TestObject1::StringSlot(" << s << ")";
}
```

23a ⟨*file:test/connect.lua* 23a⟩≡
```
q = require("qttest")

obj1 = q.Object1()
dyn = qt.dynamic()
qt.connect(obj1, "Signal()", dyn, "TestSlot()")
qt.connect(obj1, "StringSignal(QString)", dyn, "TestStringSlot(QString)")
function dyn:TestSlot()
  print("TestSlot")
end
function dyn:TestStringSlot(str)
  print("TestStringSlot ", str)
  quit()
end
qt.connect(dyn, "Sig()", obj1, "Slot()")
qt.connect(dyn, "SigString(QString)", obj1, "StringSlot(QString)")
dyn:Sig()
dyn:SigString("Test string here")
obj1:raiseSignal()
obj1:raiseStringSignal("Test string here")
```

23b ⟨*file:test/dynamic.lua* 23b⟩≡
```
d = qt.dynamic()
function d.test()
  print("d.test()")
  quit()
end
d:test()
```

## 7.2 Network testing

It's kind of impossible to test networking automatically, so I have some prepared scripts to do it manually and to check how it does.

23c ⟨*file:test/network.lua* 23c⟩≡
```
local server = network.TcpServer()
if not server then
  print("TcpServer returns nothing")
  quit()
end
local client = network.TcpClient()
if not client then
  print("TcpClient returns nothing")
  quit()
end

local input = qt.dynamic()
local output = qt.dynamic()

qt.connect(client, "connected()", input, "connected()")
qt.connect(client, "readyRead()", input, "dataReady()")

function input.connected()
  print("Client connected")
  client:write("Hello")
end

function input.dataReady()
  data = client:read(5000)
  print("Client received: ", data)
  client:close()
  quit()
end

if not server:listen("localhost", 5200) then
  print("Listen failed")
```

```
  quit(1)
end

qt.connect(server, "newConnection()", output, "newConnection()")

function output.newConnection()
  output.socket = server:get_connection()
  if not output.socket then
    print("server.get_connection returns nil")
    quit(1)
  end
  qt.connect(output.socket, "readyRead()", output, "dataReady()")
end
function output.dataReady()
  data = output.socket:read(5000)
  print("Server received: ", data)
  output:write("Response")
end
function output:write(data)
  output.socket:write(data)
end

client:connect("localhost", 5200);
```

## 7.3 Testing base Lua library

24 ⟨*file:test/testbase.lua* 24⟩≡

```
require("base")
local tcp = require("tcp_connection")
function printtable(t)
  for k,v in pairs(t) do
    print(k,v)
  end
end

x = EXPECT_CALL()

local mobile = tcp.Connection("localhost", 80)
c = mobile:Connect()
mobile:OnDataAvailable(function()
  s = mobile:Recv(10000)
  print(#s .. " bytes received successfully")
  mobile:Close()
  quit()
end)
mobile:Send("GET / HTTP/1.0\r\n\r\n")
print("Connection: ", c)
```

# 8 Memory management

## 8.1 DynamicObject

Page 6:

25a ⟨*DynamicObject deleting function* 25a⟩≡

25b ⟨*Disconnecting dynamic signal* 25b⟩≡ (8c)

# 9 Automated Test Framework

## 9.1 Glossary

**Expectation** Object describing what test is waiting for.

**Complied expectation** The expectation is complied when the event bound to it occurs. Despite compliance, the expectation still may not be satisfied with incoming data, so it can be failed.

**Satisfied expectation** The expectation is satisfied if it has been complied and `verifyData` method hasn't set its status to `FAILED`.

**Pinned expectation** Expectation could be pinned to prevent its removing from `ExpectationsList` when test case is complete. Pinned expectation may fail a test if some broken data come, but its compliance doesn't affect on test case result.

## 9.2 Expectation

Expectations are what the test made of. When some event occured, ATF examines if it was expected and marks corresponding `Expectation` object as complied. Expectations provide a way to set type and number of events expected, to specify expected order of events and to add some event handling.

Nevertheless the expectation has been complied, it still can fail if it was not completely satisfied with incoming data. In this case `Expectation:verifyData` function marks the expectation as failed.

26 ⟨*file:modules/expectations.lua* 26⟩≡ 28 ▷

```
local module = { }
module.FAILED  = { }
module.SUCCESS = { }

local cardinalities = require('cardinalities')
function module.Expectation(name, connection)
  local mt = { __index = { } }
  function mt.__index:Action(data)
    for i = 1, #self.actions do
      self.actions[i](self, data)
    end
  end
  function mt.__index:Times(c)
    if type(c) == 'table' and getmetatable(c) == cardinalities.mt then
      self.timesLE = c.lower
      self.timesGE = c.upper
    elseif type(c) == 'string' then
      self.timesLE = tonumber(c)
      self.timesGE = tonumber(c)
    elseif type(c) == 'number' then
      self.timesLE = c
      self.timesGE = c
    else
      error("Expectation:Times() must be called with number or Cardinality argument")
    end
    return self
  end
  function mt.__index:Pin()
    self.pinned = true
```

```
  if list then list:Pin(self) end
  return self
end
function mt.__index:Unpin()
  self.pinned = false
  if list then list:Unpin(self) end
  return self
end
function mt.__index:DoOnce(func)
  local idx = #self.actions + 1
  table.insert(self.actions,
               function(self, data)
                 func(self, data)
                 table.remove(self.actions, idx)
               end)
  return self
end
function mt.__index:Do(func)
  table.insert(self.actions, func)
  return self
end
function mt.__index:Timeout(ms)
  self.timeout = ms
  return self
end
function mt.__index:validate()
  -- Check Timeout status
  if timestamp() - self.ts > self.timeout then
    self.status = module.FAILED
    self.errorMessage["Timeout"] = string.format("%s: Timeout expired", self)
  end
  if self.occurences >= self.timesLE then
    -- Check if Times criteria is valid
    if self.timesGE and self.occurences > self.timesGE then
      self.status = module.FAILED
      self.errorMessage["Times"] = "The most allowed occurences boundary exceed"
    elseif not self.status then
      self.status = module.SUCCESS
    end
    -- Now check out the Sequence criteria
    for _, e in ipairs(exp.after) do
      if not e.status then
        exp.status = module.FAILED
        exp.errorMessage["Sequence"] =
          string.format("\nSequence order violated:\n\"%s\""..
            " must have got occured before \"%s\"", e, exp)
      end
    end
  end
end
function mt.__index:ValidIf(func)
  self.verifyData = function(self, data)
    local valid, msg = func(self, data)
    if not valid then
      self.status = module.FAILED
      self.errorMessage["ValidIf"] = msg
    end
  end
  return self
end
function mt:__tostring() return self.name end
local e =
{
  timesLE    = 1,    -- Times Less or Equal
  timesGE    = 1,    -- Times Greater or Equal
  after      = { },  -- Expectations that should get complied before this one
  ts         = timestamp(), -- Timestamp
  timeout    = 10000, -- Maximum allowed age
```

```
    name       = name,  -- Name to display in error message if failed
    connection = connection, -- Network connection
    occurences = 0,     -- Expectation complience times
    errorMessage = { }, -- If failed, error message to display
    actions    = { },  -- Sequence of actions to be executed when complied
    pinned     = false, -- True if the expectation is pinned
    list       = nil   -- ExpectationsList the expectation belongs to
  }

  setmetatable(e, mt)
  return e
end
```

## 9.3 Expectations List

The `ExpectationsList` class stores expectations and provides methods to add, remove expectations from the list, and to pin/unpin them. Pinned expectations are not to be removed due `ExpectationsList:Clear` call. They are used for some persistent messages processing.

28 ⟨*file:modules/expectations.lua* 26⟩+≡ ◁26

```
function module.ExpectationsList()
  local mt = { __index = {} }
  function mt.__index:Add(e)
    if e.pinned then
      table.insert(self.pinned, e)
      e.index = #self.pinned
    else
      table.insert(self.expectations, e)
      e.index = self.expectations
    end
  end
  function mt.__index:Remove(e)
    if e.pinned then
      table.remove(self.pinned, e.index)
      for i = e.index, #self.pinned do
        self.pinned[i].index = i
      end
    else
      table.remove(self.expectations)
      for i = e.index, #self.expectations do
        self.expectations[i].index = i
      end
    end
  end
  function mt.__index:Clear()
    self.expectations = { }
  end
  function mt.__index:Empty()
    return #self.expectations == 0
  end
  function mt.__index:Any(func)
    for _, e in ipairs(self.expectations) do
      if func(e) then return true end
    end
    return false
  end
  function mt.__index:List()
    return pairs(self.expectations)
  end
  function mt:__pairs() return pairs(self.expectations) end
  function mt:__ipairs()
    local function expnext(t, i)
      if self.expectations[i + 1] then
        return i + 1, self.expectations[i + 1]
```

```
      elseif self.pinned[i - #self.expectations + 1] then
        return i - #self.expectations + 1, self.pinned[i - #self.expectations + 1]
      else
        return nil
      end
    end
    return expnext, self.expectations, 0
  end
  function mt.__index:Pin(e)
    for i = 1, #self.expectations do
      if self.expectations[i] == e then
        table.remove(self.expectations, i)
        table.insert(self.pinned, e)
        break
      end
    end
  end
  function mt.__index.Unpin(e)
    for i = 1, #self.pinned do
      if self.pinned[i] == e then
        table.remove(self.pinned, i)
        table.insert(self.expectations, e)
        break
      end
    end
  end
  local res = { pinned = { }, expectations = { } }
  setmetatable(res, mt)
  return res
end

return module
```

## 9.4 Event dispatcher

Event Dispatcher object manages Event pool. `Expectation` objects (see 9.2) register expected events here. Each Event corresponds to received data — it might be an answer on StartSession request or Heartbeat control message, or RPC answer etc. When an event occurs, Dispatcher calls function bound to the event.

There are three events vectors: one (id 3) for the most definitive events, such as "Answer on Alert request tagged with correlationId = 5", second for the less definitive ones, such as "An OnAudioPathThru notification", and the lowest priority pool (id 1) for the most unspecific events, e. g. "A control packet". These pools are looked through one by one, searching for a matching event.

29 ⟨*file:modules/event_dispatcher.lua* 29⟩≡

```
expectations = require('expectations')
events = require('events')
local module = {}

local mt = { __index = { } }

function mt.__index:GetHandler(conn, ev)
  res = self._pool3[conn][ev] or
        self._pool2[conn][ev] or
        self._pool1[conn][ev]
  return res
end

function mt.__index:FindHandler(conn, data)
  -- Visit all event pools and find matching event
```

```
  local pool = nil
  local event = nil
  for e, h in pairs(self._pool3[conn]) do
    if e:matches(data) then
      return h
    end
  end
  for e, h in pairs(self._pool2[conn]) do
    if e:matches(data) then
      return h
    end
  end
  if not pull then
    for e, h in pairs(self._pool1[conn]) do
      if e:matches(data) then
        return h
      end
    end
  end
  return nil
end

function mt.__index:OnPreEvent(func)
  self.preEventHandler = func
end

function mt.__index:OnPostEvent(func)
  self.postEventHandler = func
end

function mt.__index:validateAll()
  local function iter(pool)
    for e, exp in pairs(pool) do
      exp:validate()
    end
  end

  for c, pool in pairs(self._pool3) do iter(pool) end
  for c, pool in pairs(self._pool2) do iter(pool) end
  for c, pool in pairs(self._pool1) do iter(pool) end
end

-- Function takes a Connection object and subscribes on its [[OnInputData]] signal
function mt.__index:AddConnection(connection)
  local this = self
  self._pool1[connection] = { }
  self._pool2[connection] = { }
  self._pool3[connection] = { }
  connection:OnConnected(function (self)
                           if this.preEventHandler then
                             this.preEventHandler(events.connectedEvent)
                           end
                           exp = this:GetHandler(self, events.connectedEvent)
                           if exp then
                             exp.occurences = exp.occurences + 1
                             exp:Action()
                             this:validateAll()
                           end
                           if this.postEventHandler then
                             this.postEventHandler(events.connectedEvent)
                           end
                         end)
  connection:OnDisconnected(function (self)
                              if this.preEventHandler then
                                this.preEventHandler(events.disconnectedEvent)
                              end
                              exp = this:GetHandler(self, events.disconnectedEvent)
                              if exp then
```

```
                              exp.occurences = exp.occurences + 1
                              exp:Action()
                              this:validateAll()
                            end
                            if this.postEventHandler then
                              this.postEventHandler(events.disconnectedEvent)
                            end
                          end)
  connection:OnInputData(function (self, data)
                           if this.preEventHandler then
                             this.preEventHandler(data)
                           end
                           exp = this:FindHandler(self, data)
                           if exp then
                             exp.occurences = exp.occurences + 1
                             if exp.verifyData then
                               exp:verifyData(data)
                             end
                             exp:Action(data)
                             this:validateAll()
                           end
                           if this.postEventHandler then
                             this.postEventHandler(data)
                           end
                         end)
end
function mt.__index:AddEvent(connection, event, expectation)
  if event.level == 3 then
    self._pool3[connection][event] = expectation
  elseif event.level == 2 then
    self._pool2[connection][event] = expectation
  elseif event.level == 1 then
    self._pool1[connection][event] = expectation
  end
end
function mt.__index:RemoveEvent(connection, event)
  self._pool3[connection][event] = nil
  self._pool2[connection][event] = nil
  self._pool1[connection][event] = nil
end
function module.EventDispatcher()
  local res =
  {
    _pool1 = { },
    _pool2 = { },
    _pool3 = { },
    preEventHandler = nil,
    postEventHandler = nil,
    timer  = timers.Timer()
  }
  res._pool1[res.timer] = { }
  res._pool2[res.timer] = { }
  res._pool3[res.timer] = { }
  local d = qt.dynamic()
  qt.connect(res.timer, "timeout()", d, "timeout()")
  function d.timeout()
    if res.preEventHandler then
      res.preEventHandler(events.timeoutEvent)
    end
    for c, _ in pairs(res._pool3) do
      exp = res:GetHandler(c, events.timeoutEvent)
      if exp then
        exp.occurences = exp.occurences + 1
        exp:Action()
        res:validateAll()
      end
    end
    if res.postEventHandler then
```

```
      res.postEventHandler(events.timeoutEvent)
    end
  end
  res.timer:start(400)
  setmetatable(res, mt)
  return res
end
return module
```

## 9.5 Events

32a ⟨*file:modules/events.lua* 32a⟩≡

```
local module = {}
module.connectedEvent = { level = 3 }
module.disconnectedEvent = { level = 3 }
module.timeoutEvent = { level = 3 }
local event_mt = { __index = { } }
setmetatable(module.connectedEvent, event_mt)
setmetatable(module.disconnectedEvent, event_mt)
setmetatable(module.timeoutEvent, event_mt)
function event_mt.__index:matches() return false end
function module.Event()
  local ret = { level = 1 }
  setmetatable(ret, event_mt)
  return ret
end
return module
```

### 9.5.1 Cardinalities

Cardinalities may be used as arguments of `Times()` call instead of plain number. There should be the following cardinalities:

32b ⟨*file:modules/cardinalities.lua* 32b⟩≡

```
local module = { }
module.mt = { __index = { } }
function module.Cardinality(lower, upper)
  local c = { }
  c.lower = lower
  c.upper = upper
  setmetatable(c, module.mt)
  return c
end

function AnyNumber(num)
  return module.Cardinality(0, nil)
end

function AtLeast(num)
  if num <= 0 then
    error("AtLeast: number must be greater than 0")
  end
  return module.Cardinality(num, nil)
end

function AtMost(num)
  if num <= 0 then
    error("AtMost: number must be greater than 0")
  end
  return module.Cardinality(0, num)
end

function Between(a, b)
  if (a > b) then
```

```
    error("Between: ‘from’ must be less than ‘to’")
  end
  return module.Cardinality(a, b)
end
function Exactly(num)
  return module.Cardinality(num, num)
end
return module
```

### 9.5.2 Sequences

Sequences are tables that may be passed to expectation function `InSequence` and used inside functions `Before` and `After`. They are needed to define expectations activation order. As expectations may take a part in several sequences, expectations have a field `sequences` - table of sequences.

33 ⟨*Sequence methods* 33⟩≡ (34a)

```
function seq:add(expectation)
  -- Registers new expectation in the tail of the sequence
  self.count = self.count + 1
  self.list[expectation] = self.count
end
function seq:insertBefore(item, before)
  -- Registers new expectation in the sequence before given expectation
  local beforeNum = self.list[before]
  if (beforeNum == nil) then
    error("Sequence.InsertBefore failed: Given item doesn’t belong to the sequence")
  end
  local count = #self.list
  for i = count,beforeNum,-1 do
    local exp = self.list[i]
    self.list[i + 1] = exp;
    self.list[exp] = i + 1;
  end
  self.list[item] = beforeNum + 1;
  self.list[beforeNum + 1] = item;
end
function seq:insertAfter(item, after)
  -- Registers new expectation in the sequence after given expectation
  local afterNum = self.list[after]
  if (afterNum == nil) then
    error("Sequence.InsertAfter failed: Given item doesn’t belong to the sequence")
  end
  local count = #self.list
  for i = count,afterNum+1,-1 do
    local exp = self.list[i]
    self.list[i + 1] = exp;
    self.list[exp] = i + 1;
  end
  self.list[item] = afterNum;
  self.list[afterNum] = item;
end
function seq:advance(expectation)
  -- Goes through the sequence. Returns true if the expected item activated,
  -- false otherwise
  while self.list[self.iterator] ~= expectation and
        expectation.times == 0 do
    -- skip all AnyNumber and not to be called expectations
    self.iterator = self.iterator + 1
  end
  return self.list[self.iterator] == expectation
end
```

34a ⟨*Sequence table* 34a⟩≡
```
local seq = {}

⟨Sequence methods 33⟩

seq.__index = seq

function Sequence()
  local s =
  {
    iterator = 0,
    count    = 0
  }
  setmetatable(s, seq)
  return s
end
```

Now we can define expectation functions:

34b ⟨*expectation:Sequence related functions* 34b⟩≡
```
function exp:InSequence(seq)
  return self
end
function exp:After(other)
  return self
end
function exp:Before(other)
  return self
end
```

34c ⟨*Comparisons* 34c⟩≡
```
function Gt(num)
  ⟨global::function Gt (never defined)⟩
end

function Lt(num)
  ⟨global::function Lt (never defined)⟩
end

function Ge(num)
  ⟨global::function Ge (never defined)⟩
end

function Le(num)
  ⟨global::function Le (never defined)⟩
end

function Eq(num)
  ⟨global::function Eq (never defined)⟩
endG

function Ne(num)
  ⟨global::function Ne (never defined)⟩
end
```

### 9.5.3 Output

34d ⟨*file:modules/format.lua* 34d⟩≡
```
local module = { }
console = require('console')
function module.PrintCaseResult(caseName, success, errorMessage)
  caseName = tostring(caseName)
  if #caseName > 35 then
    caseName = string.sub(caseName, 1, 22) .. "..."
  else
    caseName = caseName .. string.rep(' ', 35 - #caseName)
  end
```

```
    str = string.format("%s    %s", caseName, console.setattr(
       success and "[SUCCESS]" or "[FAIL]",
       success and "green"     or "red",
       2, false))
    print(str)
    if not success and errorMessage then
      for k, v in pairs(errorMessage) do
        print(console.setattr("  " .. k .. ": " .. v, "cyan", 1))
      end
    end
    return module
  end
  return module
```

35 ⟨*file:modules/console.lua* 35⟩≡

```
  local module = { }
  function module.setattr(string, color, bold, underline)
    if     color == "black"   then c = '30'
    elseif color == "red"     then c = '31'
    elseif color == "green"   then c = '32'
    elseif color == "brown"   then c = '33'
    elseif color == "blue"    then c = '34'
    elseif color == "magenta" then c = '35'
    elseif color == "cyan"    then c = '36'
    elseif color == "white"   then c = '37'
    end
    if bold == 1 then b = '2' end
    if bold == 2 then b = '22' end
    if bold == 3 then b = '1' end
    if underline then u = '4' else u = '24' end
    local prefix = nil
    if c then
      prefix = c
    end
    if b then
      if prefix then prefix = prefix .. ';' end
      prefix = prefix .. b
    end
    if u then
      if prefix then prefix = prefix .. ';' end
      prefix = prefix .. u
    end
    if prefix then
      prefix = '\27[' .. prefix .. 'm'
      suffix = '\27[0m'
    else
      prefix = ''
      suffix= ''
    end
    return prefix .. string .. suffix
  end
  return module
```

# 10 Lua network connections

Network connections conform one contract, so they could be superseded transparenly.

**Connect()** Opens connection

**Send(data)** Writes `data` to connection

**OnInputData(func)** Registers `func` to be called when the connection receives new data. `func` is called with `connection` as first argument (`self`) and received data as the second argument.

**OnConnected(func)** Registers `func` to be called when connection is established.

**OnDisconnected(func)** Registers `func` to be called when connection is closed. `func` is called with the only argument `self` = `connection` object.

**Close()** Closes connection.

## 10.1 Tcp Connection

36 ⟨*file:modules/tcp_connection.lua* 36⟩≡

```
local module = { mt = { __index = {} } }
function module.Connection(host, port)
  local res =
  {
    host = host,
    port = port
  }
  res.socket = network.TcpClient()
  setmetatable(res, module.mt)
  res.qtproxy = qt.dynamic()
  return res
end
local function checkSelfArg(s)
  if type(s) ~= "table" or
    getmetatable(s) ~= module.mt then
    error("Invalid argument 'self': must be connection (use ':', not '.')")
  end
end
function module.mt.__index:Connect()
  checkSelfArg(self)
  self.socket:connect(self.host, self.port)
end
function module.mt.__index:Send(data)
  checkSelfArg(self)
  return self.socket:write(data)
end
function module.mt.__index:OnInputData(func)
  checkSelfArg(self)
  local d = qt.dynamic()
  local this = self
  function d.readyRead()
    while true do
      data = self.socket:read(81920)
      if data == '' then break end
      func(this, data)
    end
  end
```

```
  qt.connect(self.socket, "readyRead()", d, "readyRead()")
end
function module.mt.__index:OnConnected(func)
  checkSelfArg(self)
  if self.qtproxy.connected then
    error("Tcp connection: connected signal is handled already")
  end
  local this = self
  self.qtproxy.connected = function() func(this) end
  qt.connect(self.socket, "connected()", self.qtproxy, "connected()")
end
function module.mt.__index:OnDisconnected(func)
  checkSelfArg(self)
  if self.qtproxy.disconnected then
    error("Tcp connection: disconnected signal is handled already")
  end
  local this = self
  self.qtproxy.disconnected = function() func(this) end
  qt.connect(self.socket, "disconnected()", self.qtproxy, "disconnected()")
end
function module.mt.__index:Close()
  checkSelfArg(self)
  self.socket:close();
end
return module
```

## 10.2 Mobile Connection

This connection acts as if the mobile communicates using Lua tables as data transfer units. The `MobileConnection` object envelops `Tcp connection`, but generates its signals on every Applink packet is available, not when raw data appears in the socket. `data` arguments of `Send` function and `OnInputData` callback function are tables described in the 11 section.

37 ⟨*file:modules/mobile_connection.lua* 37⟩≡

```
local tcp = require("tcp_connection")
require("protocol_handler")
local module = { mt = { __index = {} } }
local protocol_handler = ProtocolHandler()

function module.MobileConnection(host, port)
  res = { }
  res.connection = tcp.Connection(host, port)
  setmetatable(res, module.mt)
  return res
end
function module.mt.__index:Connect()
  self.connection:Connect()
end
function module.mt.__index:Send(data)
  local binary = protocol_handler:Compose(data)
  self.connection:Send(binary)
end
function module.mt.__index:OnInputData(func)
  local this = self
  local f =
  function(self, binary)
    local msg = protocol_handler:Parse(binary)
    for _, v in ipairs(msg) do
      func(this, v)
    end
  end
  self.connection:OnInputData(f)
end
```

```
function module.mt.__index:OnConnected(func)
  self.connection:OnConnected(function() func(self) end)
end
function module.mt.__index:OnDisconnected(func)
  self.connection:OnDisconnected(function() func(self) end)
end
function module.mt.__index:Close()
  self.connection:Close()
end
return module
```

## 10.3 WebSocket connection

38 ⟨*file:modules/websocket_connection.lua* 38⟩≡

```
json = require("json")
local module = { mt = { __index = {} } }
function module.WebSocketConnection(url, port)
  local res =
  {
    url = url,
    port = port
  }
  res.socket = network.WebSocket()
  setmetatable(res, module.mt)
  res.qtproxy = qt.dynamic()
  return res
end

function module.mt.__index:Connect()
  self.socket:open(self.url, self.port)
end

function module.mt.__index:Send(data)
  text = json.encode(data)
  --print("ws output:", text)
  self.socket:write(text)
end

function module.mt.__index:OnInputData(func)
  local d = qt.dynamic()
  local this = self
  function d:textMessageReceived(text)
    local data = json.decode(text)
    --print("ws input:", text)
    func(this, data)
  end
  qt.connect(self.socket, "textMessageReceived(QString)", d, "textMessageReceived(QString)")
end
function module.mt.__index:OnConnected(func)
  if self.qtproxy.connected then
    error("Websocket connection: connected signal is handled already")
  end
  local this = self
  self.qtproxy.connected = function() func(this) end
  qt.connect(self.socket, "connected()", self.qtproxy, "connected()")
end
function module.mt.__index:OnDisconnected(func)
  if self.qtproxy.disconnected then
    error("Websocket connection: disconnected signal is handled already")
  end
  local this = self
  self.qtproxy.disconnected = function() func(this) end
  qt.connect(self.socket, "disconnected()", self.qtproxy, "disconnected()")
end
return module
```

# 11 Protocol Handler

Protocol handler provides methods to parse and create Applink protocol messages.

`Parse` function takes binary string and returns an array of fully parsed messages, including all Applink (see Applink protocol specification [1]) protocol fields, and json payload fields if available.

`Compose` function does the inverse operation of `Parse`ing. It takes a table with all fields named as the result of `Parse` call and returns binary string with the serialized message.

39a ⟨*file:modules/protocol_handler.lua* 39a⟩≡ 39b▷
```
json = require("json")
local mt = { __index = { } }
function ProtocolHandler()
  ret =
  {
    buffer = "",
    frames = { }
  }
  setmetatable(ret, mt)
  return ret
end
local function int32ToBytes(val)
  local res = string.char(
    bit32.rshift(bit32.band(val, 0xff000000), 24),
    bit32.rshift(bit32.band(val, 0xff0000), 16),
    bit32.rshift(bit32.band(val, 0xff00), 8),
    bit32.band(val, 0xff)
  )
  return res
end
local function bytesToInt32(val, offset)
  local res = bit32.lshift(string.byte(val, offset),     24) +
              bit32.lshift(string.byte(val, offset + 1), 16) +
              bit32.lshift(string.byte(val, offset + 2),  8) +
                           string.byte(val, offset + 3)
  return res
end
```

## 11.1 Parse

`Parse` function returns an array of complete messages. If no messages was found, returned array is empty.

39b ⟨*file:modules/protocol_handler.lua* 39a⟩+≡ ◁39a 39c▷
```
function mt.__index:Parse(binary)
  self.buffer = self.buffer .. binary
  local res = { }
```

If there is no header in buffer, we have nothing to parse yet. Wait for more data.

39c ⟨*file:modules/protocol_handler.lua* 39a⟩+≡ ◁39b 39d▷
```
  while #self.buffer >= 12 do
    local msg = {}
    local c1 = string.byte(self.buffer, 1)
    msg.size        = bytesToInt32(self.buffer, 5)
```

The same thing if the current message is not read to end yet.

39d ⟨*file:modules/protocol_handler.lua* 39a⟩+≡ ◁39c 40a▷
```
    if #self.buffer < msg.size + 12 then break end
    msg.version     = bit32.rshift(bit32.band(c1, 0xf0), 4)
```

```
msg.frameType   = bit32.band(c1, 0x07)
msg.encryption  = bit32.band(c1, 0x08) == 0x08
msg.serviceType = string.byte(self.buffer, 2)
msg.frameInfo   = string.byte(self.buffer, 3)
msg.sessionId   = string.byte(self.buffer, 4)
msg.messageId   = bytesToInt32(self.buffer, 9)
msg.binaryData  = string.sub(self.buffer, 13, msg.size + 12)
```

The message is cut off from the accumulator, so it is the next message on the beginning.

40a ⟨*file:modules/protocol_handler.lua* 39a⟩+≡ ◁39d 40b▷

```
self.buffer     = string.sub(self.buffer, msg.size + 13)
```

Binary data should be parsed with respect to message fields: if the message is a frame of multiframe message, we just store its data into the buffer, if it is a RPC message, its data must be parsed as JSON. Add this message to `res` array or not, we decide here. The message should be emitted if it is a single frame or it is the last frame of mutiframe message.

First frame message initializes the buffer for consecutive frames only. No other data is used here. I just don't know what to do with frames count and full message size.

Consecutive (not last) frame data should be saved into the buffer and not be emitted.

The header of the result message made from consecutive frames is the header of last frame.

40b ⟨*file:modules/protocol_handler.lua* 39a⟩+≡ ◁40a 41▷

```
if #msg.binaryData == 0 or msg.frameType == 0 then
  table.insert(res, msg)
else
  if msg.frameType == 1 or (msg.frameType == 3 and msg.frameInfo == 0) then
    if msg.frameType == 3 then
      msg.binaryData = self.frames[msg.messageId] .. msg.binaryData
      self.frames[msg.messageId] = nil
    end
    if msg.serviceType == 7 then
      ⟨Parse RPC data 40c⟩
    end
    table.insert(res, msg)
  elseif msg.frameType == 2 then
    self.frames[msg.messageId] = ""
  elseif msg.frameType == 3 then
    self.frames[msg.messageId] = self.frames[msg.messageId] .. msg.binaryData
  end
 end
 return res
end
```

Now we need to parse JSON data located in the data section of message with `sessionId` = 7 (RPC message).

40c ⟨*Parse RPC data* 40c⟩≡ (40b)

```
msg.rpcType          = bit32.rshift(string.byte(msg.binaryData, 1), 4)
msg.rpcFunctionId    = bit32.band(bytesToInt32(msg.binaryData, 1), 0x0fffffff)
msg.rpcCorrelationId = bytesToInt32(msg.binaryData, 5)
msg.rpcJsonSize      = bytesToInt32(msg.binaryData, 9)
msg.payload          = json.decode(string.sub(msg.binaryData, 13, msg.rpcJsonSize + 12))
if msg.size > msg.rpcJsonSize + 12 then
  msg.binaryData = string.sub(msg.binaryData, msg.rpcJsonSize + 13)
else
  msg.binaryData = ""
end
```

## 11.2 Compose

`Compose` function produces Applink protocol message

41 ⟨*file:modules/protocol_handler.lua* 39a⟩+≡ ◁40b

```
function mt.__index:Compose(message)
  local payload = nil
  if message.frameType ~= 0 and message.serviceType == 7 and message.payload then
    payload = json.encode(message.payload)
    payload = string.char(
      bit32.lshift(message.rpcType, 4) + bit32.band(bit32.rshift(message.rpcFunctionId, 24), 0x0f),
      bit32.rshift(bit32.band(message.rpcFunctionId, 0xff0000), 16),
      bit32.rshift(bit32.band(message.rpcFunctionId, 0xff00), 8),
      bit32.band(message.rpcFunctionId, 0xff))  ..
      int32ToBytes(message.rpcCorrelationId) ..
      int32ToBytes(#payload) ..
      payload
  end
  local res = string.char(
    bit32.bor(
      bit32.lshift(message.version, 4),
      (message.encryption and 0x80 or 0),
      bit32.band(message.frameType, 0x07)),
    message.serviceType,
    message.frameInfo,
    message.sessionId) ..
    (payload and int32ToBytes(#payload) or string.char(0, 0, 0, 0)) .. -- size
    int32ToBytes(message.messageId)
  if payload then
    res = res .. payload;
  end
  return res
end
```

# 12 Test base class

This class manages test control flow.

42a ⟨*file:modules/testbase.lua* 42a⟩≡ 42b▷

```
local ed           = require("event_dispatcher")
local events       = require("events")
local expectations = require('expectations')
local console      = require('console')
local fmt          = require('format')

local module = { }

local Expectation = expectations.Expectation
local SUCCESS = expectations.SUCCESS
local FAILED = expectations.FAILED

local control = qt.dynamic()

local mt =
{
  __index =
  {
    test_cases = { },
    case_names = { },
    current_case_name = nil,
    current_case_index = 0,
    expectations_list = expectations.ExpectationsList(),
    AddExpectation = function(self,e)
                       self.expectations_list:Add(e)
                     end,
    RemoveExpectation = function(self, e)
                          self.expectations_list:Remove(e)
                        end,
  },
  __newindex = function(t, k, v)
                 if type(v) == "function" then
                   table.insert(t.test_cases, v)
                   t.case_names[v] = k
                 else
                   rawset(t, k, v)
                 end
               end,
  __metatable = { }
}
function control.runNextCase()
  module.current_case_index = module.current_case_index + 1
  local testcase = module.test_cases[module.current_case_index]
  if testcase then
    module.current_case_name = module.case_names[testcase]
    testcase(module)
  else
    quit()
  end
end
setmetatable(module, mt)

qt.connect(control, "next()", control, "runNextCase()")
```

`CheckStatus` function verifies status of test expectations and executes test cases.

42b ⟨*file:modules/testbase.lua* 42a⟩+≡ ◁42a 43a▷

```
local function CheckStatus()
  if module.current_case_name == nil or module.current_case_name == '' then return end
  -- Check the test status
  if module.expectations_list:Any(function(e) return not e.status end) then return end
  local success = true
```

```
  local errorMessage = {}
  for _, e in ipairs(module.expectations_list) do
    if e.status ~= SUCCESS then
      success = false
    end
    if not e.pinned then
      module.event_dispatcher:RemoveEvent(e.connection, e.event)
    end
    for k, v in pairs(e.errorMessage) do
      errorMessage[k] = v
    end
  end
  fmt.PrintCaseResult(module.current_case_name, success, errorMessage)
  module.expectations_list:Clear()
  module.current_case_name = nil
  if not success then quit() return end
  control:next()
end

module.event_dispatcher = ed.EventDispatcher()
module.event_dispatcher:OnPostEvent(CheckStatus)
```

I use `control` object to send signals to the Test object, it is the only way to run test cases from within event loop.

43a ⟨*file:modules/testbase.lua* 42a⟩+≡ ◁42b

```
control:next()
return module
```

### 12.0.1 Mobile session

`MobileSession` table allows to control one RPC session. It contains methods to send RPC queries.

At first I declare some constant values as table of function identifiers, as declared in `HMI_API.xml`.

43b ⟨*file:modules/mobile_session.lua* 43b⟩≡ 44a▷

```
local expectations = require('expectations')
local events = require('events')
local Expectation = expectations.Expectation
local Event = events.Event
local SUCCESS = expectations.SUCCESS
local FAILED = expectations.FAILED
local module = {}
local functionIds = {
  ["RegisterAppInterface"]       =    1,
  ["UnregisterAppInterface"]     =    2,
  ["SetGlobalProperties"]        =    3,
  ["ResetGlobalProperties"]      =    4,
  ["AddCommand"]                 =    5,
  ["DeleteCommand"]              =    6,
  ["AddSubMenu"]                 =    7,
  ["DeleteSubMenu"]              =    8,
  ["CreateInteractionChoiceSet"] =    9,
  ["PerformInteraction"]         =    10,
  ["DeleteInteractionChoiceSet"] =    11,
  ["Alert"]                      =    12,
  ["Show"]                       =    13,
  ["Speak"]                      =    14,
  ["SetMediaClockTimer"]         =    15,
  ["PerformAudioPassThru"]       =    16,
  ["EndAudioPassThru"]           =    17,
  ["SubscribeButton"]            =    18,
  ["UnsubscribeButton"]          =    19,
  ["SubscribeVehicleData"]       =    20,
```

```
    ["UnsubscribeVehicleData"]     =    21,
    ["GetVehicleData"]             =    22,
    ["ReadDID"]                    =    23,
    ["GetDTCs"]                    =    24,
    ["ScrollableMessage"]          =    25,
    ["Slider"]                     =    26,
    ["ShowConstantTBT"]            =    27,
    ["AlertManeuver"]              =    28,
    ["UpdateTurnList"]             =    29,
    ["ChangeRegistration"]         =    30,
    ["GenericResponse"]            =    31,
    ["PutFile"]                    =    32,
    ["DeleteFile"]                 =    33,
    ["ListFiles"]                  =    34,
    ["SetAppIcon"]                 =    35,
    ["SetDisplayLayout"]           =    36,
    ["DiagnosticMessage"]          =    37,
    ["SystemRequest"]              =    38,
    ["SendLocation"]               =    39,
    ["OnHMIStatus"]                = 32768,
    ["OnAppInterfaceUnregistered"] = 32769,
    ["OnButtonEvent"]              = 32770,
    ["OnButtonPress"]              = 32771,
    ["OnVehicleData"]              = 32772,
    ["OnCommand"]                  = 32773,
    ["OnTBTClientState"]           = 32774,
    ["OnDriverDistraction"]        = 32775,
    ["OnPermissionsChange"]        = 32776,
    ["OnAudioPassThru"]            = 32777,
    ["OnLanguageChange"]           = 32778,
    ["OnKeyboardInput"]            = 32779,
    ["OnTouchEvent"]               = 32780,
    ["OnSystemRequest"]            = 32781,
    ["OnHashChange"]               = 32782
}
```

Auxiliary function `compareValues` takes two arguments and returns `true` if they are equal or pair `false`, `string` if not. `string` contains a message with list of unequal fields.

44a ⟨*file:modules/mobile_session.lua* 43b⟩+≡ ◁43b 44b▷

```
local function compareValues(a, b, name)
  local function iter(a, b, name, msg)
    if type(a) == 'table' and type(b) == 'table' then
      local res = true
      for k, v in pairs(a) do
        res = res and iter(v, b[k], name .. "." .. k, msg)
      end
      return res
    else
      if a == b then
        return true
      else
        table.insert(msg, string.format("%s: expected: %s, actual value: %s", name, a, b))
        return false
      end
    end
  end
  local message = { }
  local res = iter(a, b, name, message)
  return res, table.concat(message, '\n')
end
```

Now I declare metable for `MobileSession` object.

44b ⟨*file:modules/mobile_session.lua* 43b⟩+≡ ◁44a 45▷

```
local mt = { __index = { } }
function mt.__index:ExpectEvent(event, name)
  local ret = Expectation(name, self.connection)
```

```
  ret.event = event
  self.event_dispatcher:AddEvent(self.connection, event, ret)
  self.exp_list:Add(ret)
  return ret
end
```

Function `ExpectResponse` takes function name and arguments list and adds an expectation of response to call of this function. If `arguments` specified, the payload of response if checked against this value during test validation.

45 ⟨*file:modules/mobile_session.lua* 43b⟩+≡ ◁44b 46▷

```
function mt.__index:ExpectResponse(correlationId, ...)
  local args = table.pack(...)
  local event = events.Event()
  event.matches = function(self, data)
                    return data.rpcCorrelationId == correlationId
                  end
  local ret = Expectation("response to " .. correlationId, self.connection)
  if #args > 0 then
    ret:ValidIf(function(self, data)
                  local arguments
                  if self.occurences > #args then
                    arguments = args[#args]
                  else
                    arguments = args[self.occurences]
                  end
                  return compareValues(arguments, data.payload, "payload")
                end)
  end
  ret.event = event
  self.event_dispatcher:AddEvent(self.connection, event, ret)
  self.exp_list:Add(ret)
  return ret
end
function mt.__index:ExpectNotification(funcName, ...)
  local args = table.pack(...)
  local event = events.Event()
  event.matches = function(self, data)
                    return data.rpcFunctionId == functionIds[funcName]
                  end
  local ret = Expectation(funcName .. " notification", self.connection)
  if #args > 0 then
    ret:ValidIf(function(self, data)
                  local arguments
                  if self.occurences > #args then
                    arguments = args[#args]
                  else
                    arguments = args[self.occurences]
                  end
                  return compareValues(arguments, data.payload, "payload")
                end)
  end
  ret.event = event
  self.event_dispatcher:AddEvent(self.connection, event, ret)
  self.exp_list:Add(ret)
  return ret
end
function mt.__index:Send(message)
  if not message.serviceType then
    error("MobileSession:Send: sessionId must be specified")
  end
  if not message.frameInfo then
    error("MobileSession:Send: frameInfo must be specified")
  end
  self.messageId = self.messageId + 1
  self.connection:Send(
  {
    version          = message.version or self.version,
```

```
    encryption       = message.encryption or false,
    frameType        = message.frameType or 1,
    serviceType      = message.serviceType,
    frameInfo        = message.frameInfo,
    sessionId        = self.sessionId,
    messageId        = self.messageId,
    rpcType          = message.rpcType,
    rpcFunctionId    = message.rpcFunctionId,
    rpcCorrelationId = message.rpcCorrelationId,
    payload          = message.payload,
    binaryData       = message.binaryData
  })
end
function mt.__index:SendRPC(func, arguments)
  self.correlationId = self.correlationId + 1
  self:Send(
  {
    serviceType      = 7,
    frameInfo        = 0,
    rpcType          = 0,
    rpcFunctionId    = functionIds[func],
    rpcCorrelationId = self.correlationId,
    payload          = arguments
  })
  return self.correlationId
end
```

`StartService` function sends `StartService` message to SDL and adds an expectation for response. There is a trouble if you need to start two or more services of one type, because it's impossible to dispatch responses. It's a protocol problem, not ATF's.

46 ⟨*file:modules/mobile_session.lua* 43b⟩+≡ ◁45

```
function mt.__index:StartService(service)
  local startSession =
  {
    frameType   = 0,
    serviceType = service,
    frameInfo   = 1,
    sessionId   = self.sessionId,
  }
  self:Send(startSession)
  -- prepare event to expect
  local startserviceEvent = Event()
  startserviceEvent.matches = function(_, data)
                                return data.frameType   == 0 and
                                       data.serviceType == service and
                                       (data.frameInfo  == 2 or  -- Start Service ACK
                                        data.frameInfo  == 3)    -- Start Service NACK
                              end

  local ret = self:ExpectEvent(startserviceEvent, "StartService ACK")
    :ValidIf(function(s, data)
               if data.frameInfo == 2 then return true
               else return false, "StartService NACK received" end
             end)
  if service == 7 then
    ret:Do(function(s, data)
             if s.status == FAILED then return end
             self.sessionId = data.sessionId
           end)
  end
  return ret
end
function mt.__index:Start()
  self:StartService(7)
    :Do(function()
          local correlationId = self:SendRPC("RegisterAppInterface",
```

```
          {
            syncMsgVersion =
            {
              majorVersion = 3,
              minorVersion = 0
            },
            appName = "Test Application",
            isMediaApplication = true,
            languageDesired = 'EN-US',
            hmiDisplayLanguageDesired = 'EN-US',
            appHMIType = { "NAVIGATION" },
            appID = "8675308",
            deviceInfo =
            {
              os = "Android",
              carrier = "Megafon",
              firmwareRev = "Name: Linux, Version: 3.4.0-perf",
              osVersion = "4.4.2",
              maxNumberRFCOMMPorts = 1
            }
          })
          self:ExpectResponse(correlationId, { success = true })
        end)
end

function module.MobileSession(event_dispatcher, exp_list, connection)
  local res = { }
  res.connection = connection
  res.event_dispatcher = event_dispatcher
  res.exp_list = exp_list
  res.messageId  = 1
  res.sessionId  = 0
  res.correlationId = 1
  res.version = 2
  setmetatable(res, mt)
  return res
end
return module
```

### 12.0.2 EXPECT_CALL

This function simplifies the most common `Expectation` object creation. It creates `Event` object matching specified RPC call, and binds it to the new expectation. The expectation verifies all given arguments of function call.

47 ⟨*file:modules/connecttest.lua* 47⟩≡

```
local module = require('testbase')
local mobile       = require("mobile_connection")
local mobile_session = require("mobile_session")
local websocket    = require('websocket_connection')
local events       = require("events")
local expectations = require('expectations')
local config       = require('config')
local Event = events.Event

local Expectation = expectations.Expectation
local SUCCESS = expectations.SUCCESS
local FAILED = expectations.FAILED

module.hmiConnection = websocket.WebSocketConnection(config.hmiUrl, config.hmiPort)
module.mobileConnection = mobile.MobileConnection(config.mobileHost, config.mobilePort)
module.event_dispatcher:AddConnection(module.hmiConnection)
module.event_dispatcher:AddConnection(module.mobileConnection)

function module.hmiConnection:EXPECT_HMIRESPONSE(id)
  local event = events.Event()
```

```
  event.matches = function(self, data) return data.id == id end
  local ret = Expectation("HMI response " .. id, self)
  ret.event = event
  module.event_dispatcher:AddEvent(module.hmiConnection, event, ret)
  module:AddExpectation(ret)
  return ret
end

function EXPECT_HMIRESPONSE(id)
  return module.hmiConnection:EXPECT_HMIRESPONSE(id)
end

function EXPECT_HMICALL(methodName, ...)
  local args = table.pack(...)
  -- TODO: Avoid copy-paste
  local function compareValues(a, b, name)
    local function iter(a, b, name, msg)
      if type(a) == 'table' and type(b) == 'table' then
        local res = true
        for k, v in pairs(a) do
          res = res and iter(v, b[k], name .. "." .. k, msg)
        end
        return res
      else
        if a == b then
          return true
        else
          table.insert(msg, string.format("%s: expected: %s, actual value: %s", name, a, b))
          return false
        end
      end
    end
    local message = { }
    local res = iter(a, b, name, message)
    return res, table.concat(message, '\n')
  end
  local event = events.Event()
  event.matches =
    function(self, data) return data.method == methodName end
  local ret = Expectation("HMI call " .. methodName, module.hmiConnection)
  if #args > 0 then
    ret:ValidIf(function(self, data)
                  local arguments
                  if self.occurences > #args then
                    arguments = args[#args]
                  else
                    arguments = args[self.occurences]
                  end
                  return compareValues(arguments, data.params, "params")
                end)
  end
  ret.event = event
  module.event_dispatcher:AddEvent(module.hmiConnection, event, ret)
  module:AddExpectation(ret)
  return ret
end

function EXPECT_NOTIFICATION(func, ...)
  return module.mobileSession:ExpectNotification(func, ...)
end

function EXPECT_RESPONSE(correlationId, ...)
  return module.mobileSession:ExpectResponse(correlationId, ...)
end

function EXPECT_EVENT(event, name)
  local ret = Expectation(name, module.mobileConnection)
  ret.event = event
```

```
  module.event_dispatcher:AddEvent(module.mobileConnection, event, ret)
  module:AddExpectation(ret)
  return ret
end

function EXPECT_HMIEVENT(event, name)
  local ret = Expectation(name, module.hmiConnection)
  ret.event = event
  module.event_dispatcher:AddEvent(module.hmiConnection, event, ret)
  module:AddExpectation(ret)
  return ret
end

function module:InitHMI()
  local idCounter = 100
  local function registerComponent(name, subscriptions)
    local expId = idCounter
    idCounter = idCounter + 1
    local exp = EXPECT_HMIRESPONSE(expId)
    if subscriptions then
      local subscriptionIdCounter = idCounter
      for _, s in ipairs(subscriptions) do
        EXPECT_HMIRESPONSE(idCounter)
        local id = idCounter
        idCounter = idCounter + 1
        exp:Do(function()
                 module.hmiConnection:Send({
                   jsonrpc = "2.0",
                   id      = id,
                   method  = "MB.subscribeTo",
                   params  = { propertyName = s }
                 })
               end)
      end
    end
    module.hmiConnection:Send({
      jsonrpc = "2.0",
      id      = expId,
      method  = "MB.registerComponent",
      params = { componentName = name }
    })
  end


  EXPECT_HMIEVENT(events.connectedEvent, "Connected websocket")
    :Do(function()
          registerComponent("Buttons")
          registerComponent("TTS")
          registerComponent("VR")
          registerComponent("BasicCommunication",
          {
            "BasicCommunication.OnPutFile",
            "SDL.OnStatusUpdate",
            "SDL.OnAppPermissionChanged",
            "BasicCommunication.OnSDLPersistenceComplete",
            "BasicCommunication.OnFileRemoved",
            "BasicCommunication.OnAppRegistered",
            "BasicCommunication.OnAppUnregistered",
            "BasicCommunication.PlayTone",
            "BasicCommunication.OnSDLClose",
            "SDL.OnSDLConsentNeeded",
            "BasicCommunication.OnResumeAudioSource"
          })
          registerComponent("UI",
          {
            "UI.OnRecordStart"
          })
          registerComponent("VehicleInfo")
```

```
          registerComponent("Navigation")
        end)
  self.hmiConnection:Connect()
end

function module:InitHMI_onReady()
  local function ExpectRequest(name, mandatory, response)
    local event = events.Event()
    event.matches = function(self, data) return data.method == name end
    response.code = 0
    response.method = name
    return
    EXPECT_HMIEVENT(event, name)
      :Times(mandatory and 1 or AnyNumber())
      :Do(function(_, data)
            self.hmiConnection:Send({
              id = data.id,
              jsonrpc = "2.0",
              result = response
            })
          end)
  end

  ExpectRequest("BasicCommunication.MixingAudioSupported",
                true,
                { attenuatedSupported = true })
  ExpectRequest("BasicCommunication.GetSystemInfo", false,
  {
    ccpu_version = "ccpu_version",
    language = "EN-US",
    wersCountryCode = "wersCountryCode"
  })
  ExpectRequest("UI.GetLanguage", false, { language = "EN-US" })
  ExpectRequest("UI.ChangeRegistration", false, { }):Pin()
  ExpectRequest("TTS.SetGlobalProperties", false, { }):Pin()
  ExpectRequest("BasicCommunication.UpdateDeviceList", false, { }):Pin()
  ExpectRequest("VR.ChangeRegistration", false, { }):Pin()
  ExpectRequest("TTS.ChangeRegistration", false, { }):Pin()
  ExpectRequest("VR.GetSupportedLanguages", false, {
    languages =
    {
      "EN-US","ES-MX","FR-CA","DE-DE","ES-ES","EN-GB","RU-RU","TR-TR","PL-PL",
      "FR-FR","IT-IT","SV-SE","PT-PT","NL-NL","ZH-TW","JA-JP","AR-SA","KO-KR",
      "PT-BR","CS-CZ","DA-DK","NO-NO"
    }
  }):Pin()
  ExpectRequest("TTS.GetSupportedLanguages", false, {
    languages =
    {
      "EN-US","ES-MX","FR-CA","DE-DE","ES-ES","EN-GB","RU-RU","TR-TR","PL-PL",
      "FR-FR","IT-IT","SV-SE","PT-PT","NL-NL","ZH-TW","JA-JP","AR-SA","KO-KR",
      "PT-BR","CS-CZ","DA-DK","NO-NO"
    }
  }):Pin()
  ExpectRequest("TTS.GetSupportedLanguages", false, {
    languages =
    {
      "EN-US","ES-MX","FR-CA","DE-DE","ES-ES","EN-GB","RU-RU","TR-TR","PL-PL",
      "FR-FR","IT-IT","SV-SE","PT-PT","NL-NL","ZH-TW","JA-JP","AR-SA","KO-KR",
      "PT-BR","CS-CZ","DA-DK","NO-NO"
    }
  }):Pin()
  ExpectRequest("VehicleInfo.GetVehicleType", false, {
    vehicleType =
    {
      make = "Ford",
      model = "Fiesta",
      modelYear = "2013",
```

```
    trim = "SE"
  }
}):Pin()
ExpectRequest("VehicleInfo.GetVehicleData", false, { vin = "52-452-52-752" }):Pin()

local function button_capability(name, shortPressAvailable, longPressAvailable, upDownAvailable)
  return
  {
    name = name,
    shortPressAvailable = shortPressAvailable == nil and true or shortPressAvailable,
    longPressAvailable = longPressAvailable == nil and true or longPressAvailable,
    upDownAvailable = upDownAvailable == nil and true or upDownAvailable
  }
end
local buttons_capabilities =
{
  capabilities =
  {
    button_capability("PRESET_0"),
    button_capability("PRESET_1"),
    button_capability("PRESET_2"),
    button_capability("PRESET_3"),
    button_capability("PRESET_4"),
    button_capability("PRESET_5"),
    button_capability("PRESET_6"),
    button_capability("PRESET_7"),
    button_capability("PRESET_8"),
    button_capability("PRESET_9"),
    button_capability("OK", true, false, true),
    button_capability("SEEKLEFT"),
    button_capability("SEEKRIGHT"),
    button_capability("TUNEUP"),
    button_capability("TUNEDOWN")
  },
  presetBankCapabilities = { onScreenPresetsAvailable = true }
}
ExpectRequest("Buttons.GetCapabilities", true, buttons_capabilities):Pin()
ExpectRequest("VR.GetCapabilities", false, { vrCapabilities = { "TEXT" } }):Pin()
ExpectRequest("TTS.GetCapabilities", false, {
  speechCapabilities = { "TEXT", "PRE_RECORDED" },
  prerecordedSpeechCapabilities =
  {
      "HELP_JINGLE",
      "INITIAL_JINGLE",
      "LISTEN_JINGLE",
      "POSITIVE_JINGLE",
      "NEGATIVE_JINGLE"
  }
}):Pin()

local function text_field(name, characterSet, width, rows)
  return
  {
    name = name,
    characterSet = characterSet or "TYPE2SET",
    width = width or 500,
    rows = rows or 1
  }
end
local function image_field(name, width, heigth)
  return
  {
    name = name,
    imageTypeSupported =
    {
      "GRAPHIC_BMP",
      "GRAPHIC_JPEG",
      "GRAPHIC_PNG"
```

```
      },
      imageResolution =
      {
        resolutionWidth = width or 64,
        resolutionHeight = height or 64
      }
    }

end

ExpectRequest("UI.GetCapabilities", false, {
  displayCapabilities =
  {
    displayType = "GEN2_8_DMA",
    textFields =
    {
        text_field("mainField1"),
        text_field("mainField2"),
        text_field("mainField3"),
        text_field("mainField4"),
        text_field("statusBar"),
        text_field("mediaClock"),
        text_field("mediaTrack"),
        text_field("alertText1"),
        text_field("alerText2"),
        text_field("alertText3"),
        text_field("scrollableMessageBody"),
        text_field("initialInteractionText"),
        text_field("navigationText1"),
        text_field("navigationText2"),
        text_field("ETA"),
        text_field("totalDistance"),
        text_field("navigationText"),
        text_field("audioPassThruDisplayText1"),
        text_field("audioPassThruDisplayText2"),
        text_field("sliderHeader"),
        text_field("sliderFooter"),
        text_field("notificationText"),
        text_field("menuName"),
        text_field("secondaryText"),
        text_field("tetriaryText"),
        text_field("timeToDestination"),
        text_field("turnText"),
        text_field("menuTitle")
    },
    imageFields =
    {
      image_field("softButtonImage"),
      image_field("choiceImage"),
      image_field("choiceSecondaryImage"),
      image_field("vrHelpItem"),
      image_field("turnIcon"),
      image_field("menuIcon"),
      image_field("cmdIcon"),
      image_field("imageTypeSupported"),
      image_field("showConstantTBTIcon"),
      image_field("showConstantTBTNextTurnIcon")
    },
    mediaClockFormats =
    {
        "CLOCK1",
        "CLOCK2",
        "CLOCK3",
        "CLOCKTEXT1",
        "CLOCKTEXT2",
        "CLOCKTEXT3",
        "CLOCKTEXT4"
    },
```

```
      graphicSupported = true,
      imageCapabilities = { "DYNAMIC", "STATIC" },
      templatesAvailable = { "TEMPLATE" },
      screenParams =
      {
        resolution = { resolutionWidth = 800, resolutionHeight = 480 },
        touchEventAvailable =
        {
          pressAvailable = true,
          multiTouchAvailable = true,
          doublePressAvailable = false
        }
      },
      numCustomPresetsAvailable = 10
    },
    audioPassThruCapabilities =
    {
      samplingRate = "44KHZ",
      bitsPerSample = "8_BIT",
      audioType = "PCM"
    },
    hmiZoneCapabilities = "FRONT",
    softButtonCapabilities =
    {
      shortPressAvailable = true,
      longPressAvailable = true,
      upDownAvailable = true,
      imageSupported = true
    }
  }):Pin()

  ExpectRequest("VR.IsReady", true, { available = true })
  ExpectRequest("TTS.IsReady", true, { available = true })
  ExpectRequest("UI.IsReady", true, { available = true })
  ExpectRequest("Navigation.IsReady", true, { available = true })
  ExpectRequest("VehicleInfo.IsReady", true, { available = true })

  self.hmiConnection:Send { jsonrpc = "2.0", method = "BasicCommunication.OnReady" }
end

function module:ConnectMobile()
  -- Connected expectation
  self.mobileSession = mobile_session.MobileSession(
    self.event_dispatcher,
    self.expectations_list,
    self.mobileConnection)
  self.mobileSession:ExpectEvent(events.timeoutEvent, "timeout")
    :Times(AnyNumber())
    :Pin()
  self.mobileSession:ExpectEvent(events.connectedEvent, "Connection started")
  self.mobileConnection:Connect()
end
function module:StartSession()
  self.mobileSession:Start()

  local function ExpectRequest(name, mandatory, response)
    local event = events.Event()
    event.matches = function(self, data) return data.method == name end
    response.code = 0
    response.method = name
    return
    EXPECT_HMIEVENT(event, name)
      :Times(mandatory and 1 or AnyNumber())
      :Do(function(_, data)
            self.hmiConnection:Send({
              id = data.id,
              jsonrpc = "2.0",
              result = response
```

```
            })
          end)
  end

  ExpectRequest("BasicCommunication.UpdateAppList", true, { })
    :Pin()
    :ValidIf(function(_, data) return #data.params.applications > 0 end)
    :Do(function(_, data)
          if #data.params.applications > 0 then
            self.appId = data.params.applications[1].appID
          end
        end)
end

return module
```

54 ⟨*file:modules/config.lua* 54⟩≡

```
local config = { }

config.hmiUrl = "ws://localhost"
config.hmiPort = 8087
config.mobileHost = "localhost"
config.mobilePort = 12345

config.application1 =
{
  appName = "Test Application",
  isMediaApplication = true,
  appId   = "8675308"
}

return config
```

## References

[1] Ford protocol specification, revision 8, 2015