Appendix C. A message-passing framework and complete ATM example

Back in chapter 4, I presented an example of sending messages between threads using a message-passing framework, using a simple implementation of the code in an ATM as an example. What follows is the complete code for this example, including the message-passing framework.

Listing C.1 shows the message queue. It stores a list of messages as pointers to a base class; the specific message type is handled with a template class derived from that base class. Pushing an entry constructs an appropriate instance of the wrapper class and stores a pointer to it; popping an entry returns that pointer. Because the `message_base` class doesn't have any member functions, the popping thread will need to cast the pointer to a suitable `wrapped_message<T>` pointer before it can access the stored message.

Listing C.1. A simple message queue

[View full size image]

```
#include <mutex>
#include <condition_variable>
#include <queue>
#include <memory>

namespace messaging
{
    struct message_base                    <- Base class of our queue entries
    {
        virtual ~message_base()
        {}
    };
    template<typename Msg>
    struct wrapped_message:                <- Each message type has a specialization
        message_base
    {
        Msg contents;

        explicit wrapped_message(Msg const& contents_):
            contents(contents_)
        {}
    };
    class queue                            <- Our message queue
    {
        std::mutex m;
        std::condition_variable c;
        std::queue<std::shared_ptr<message_base> > q;   <- Actual queue stores pointers to message_base
    public:
        template<typename T>
        void push(T const& msg)
        {
            std::lock_guard<std::mutex> lk(m);
            q.push(std::make_shared<wrapped_message<T> >(msg));   <- Wrap posted message and store pointer
            c.notify_all();
        }

        std::shared_ptr<message_base> wait_and_pop()
        {
            std::unique_lock<std::mutex> lk(m);
            c.wait(lk,[&]{return !q.empty();});   <- Block until queue isn't empty
            auto res=q.front();
            q.pop();
            return res;
        }
    };
}
```

Sending messages is handled through an instance of the `sender` class shown in listing C.2. This is just a thin wrapper around a message queue that only allows messages to be pushed. Copying instances of `sender` just copies the pointer to the queue rather than the queue itself.

Listing C.2. The sender class

```
namespace messaging
{
    class sender
    {
        queue*q;
    public:
        sender():
            q(nullptr)
        {}

        explicit sender(queue*q_):
            q(q_)
        {}

        template<typename Message>
        void send(Message const& msg)
        {
            if(q)
            {
                q->push(msg);
            }
        }
    };
}
```

sender is wrapper around queue pointer

Default-constructed sender has no queue

Allow construction from pointer to queue

Sending pushes message on the queue

Receiving messages is a bit more complicated. Not only do you have to wait for a message from the queue, but you also have to check to see if the type matches any of the message types being waited on and call the appropriate handler function. This all starts with the `receiver` class shown in the following listing.

Listing C.3. The receiver class

```
namespace messaging
{
    class receiver
    {
        queue q;
    public:
        operator sender()
        {
            return sender(&q);
        }

        dispatcher wait()
        {
            return dispatcher(&q);
        }
    };
}
```

A receiver owns the queue

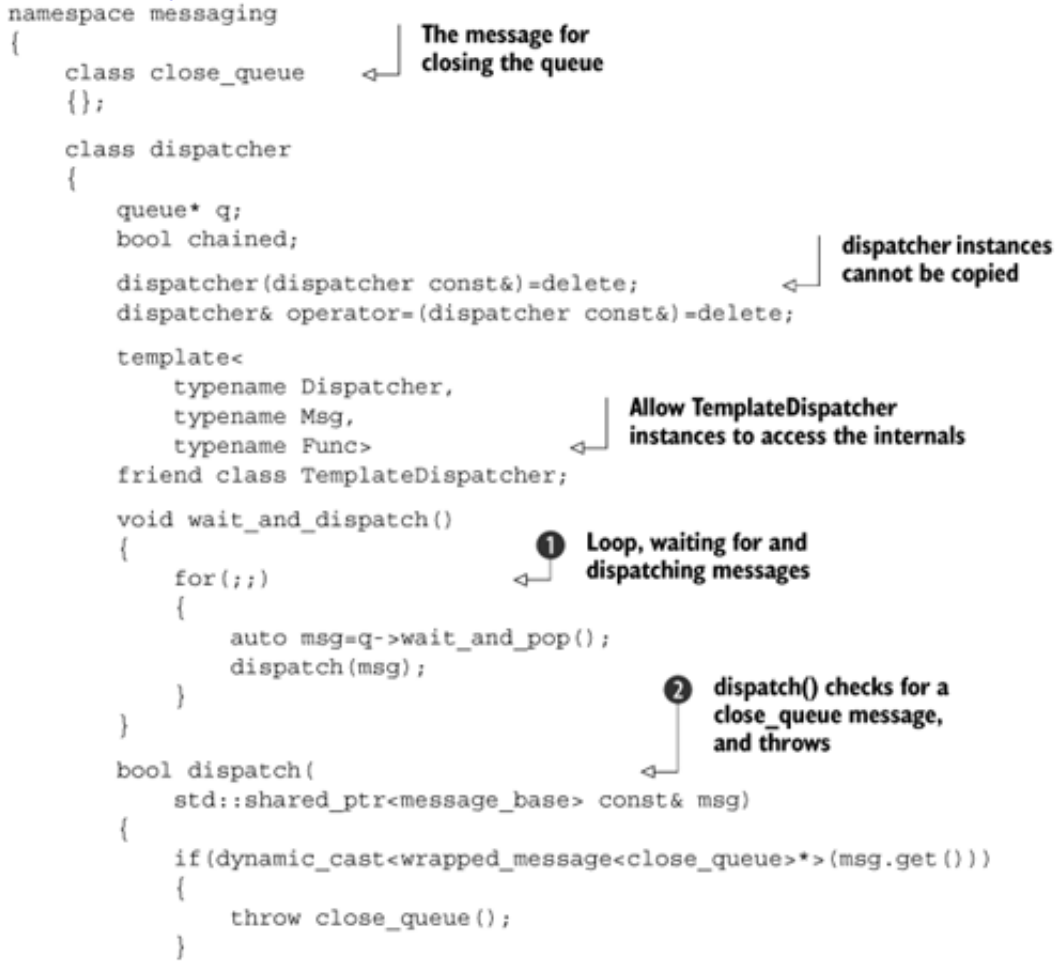
Allow implicit conversion to a sender that references the queue

Waiting for a queue creates a dispatcher

Whereas a `sender` just references a message queue, a `receiver` owns it. You can obtain a `sender` that references the queue by using the implicit conversion. The complexity of doing the message dispatch starts with a call to `wait()`. This creates a `dispatcher` object that references the queue from the `receiver`. The `dispatcher` class is shown in the next listing; as you can see, the work is done in the *destructor*. In this case, that work consists of waiting for a message and dispatching it.

Listing C.4. The dispatcher class

[View full size image]




```
namespace messaging
{
    class close_queue
    {};

    class dispatcher
    {
        queue* q;
        bool chained;

        dispatcher(dispatcher const&)=delete;
        dispatcher& operator=(dispatcher const&)=delete;

        template<
            typename Dispatcher,
            typename Msg,
            typename Func>
        friend class TemplateDispatcher;

        void wait_and_dispatch()
        {
            for(;;)
            {
                auto msg=q->wait_and_pop();
                dispatch(msg);
            }
        }

        bool dispatch(
            std::shared_ptr<message_base> const& msg)
        {
            if(dynamic_cast<wrapped_message<close_queue>*>(msg.get()))
            {
                throw close_queue();
            }
```

[View full size image]

```
            return false;
        }
    public:
        dispatcher(dispatcher&& other):
            q(other.q),chained(other.chained)
        {
            other.chained=true;
        }
        explicit dispatcher(queue* q_):
            q(q_),chained(false)
        {}
        template<typename Message,typename Func>
        TemplateDispatcher<dispatcher,Message,Func>
        handle(Func&& f)
        {
            return TemplateDispatcher<dispatcher,Message,Func>(
                q,this,std::forward<Func>(f));
        }
        ~dispatcher() noexcept(false)
        {
            if(!chained)
            {
                wait_and_dispatch();
            }
        }
    };
}
```

dispatcher instances can be moved

The source mustn't wait for messages

❸ Handle a specific type of message with a TemplateDispatcher

❹ The destructor might throw exceptions

The `dispatcher` instance that's returned from `wait()` will be destroyed immediately, because it's a temporary, and as mentioned, the destructor does the work. The destructor calls ❶ `wait_and_dispatch()`, which is a loop ❷ that waits for a message and passes it to `dispatch()`.

`dispatch()` itself is rather simple; it checks whether the message is a `close_queue` message and throws an exception if it is; otherwise, it returns `false` to indicate that the message was unhandled. This `close_queue` exception is why the destructor is marked `noexcept(false)`; without this ❹ annotation the default exception specification for the destructor would be `noexcept(true)`, indicating that no exceptions can be thrown, and the `close_queue` exception would thus terminate the program.

It's not often that you're going to call `wait()` on its own, though; most of the time you'll be wanting ❸ to handle a message. This is where the `handle()` member function comes in. It's a template, and the message type isn't deducible, so you must specify which message type to handle and pass in a function (or callable object) to handle it. `handle()` itself passes the queue, the current `dispatcher` object, and the handler function to a new instance of the `TemplateDispatcher` class template, to handle messages of the specified type. This is shown in listing C.5. This is why you test the `chained` value in the destructor before waiting for messages; not only does it prevent moved-from objects waiting for messages, but it also allows you to transfer the responsibility of waiting to your new `TemplateDispatcher` instance.

Listing C.5. The TemplateDispatcher class template

[View full size image]

```
namespace messaging
{
    template<typename PreviousDispatcher,typename Msg,typename Func>
    class TemplateDispatcher
    {
        queue* q;
        PreviousDispatcher* prev;
        Func f;
        bool chained;

        TemplateDispatcher(TemplateDispatcher const&)=delete;
        TemplateDispatcher& operator=(TemplateDispatcher const&)=delete;

        template<typename Dispatcher,typename OtherMsg,typename OtherFunc>
        friend class TemplateDispatcher;

        void wait_and_dispatch()
        {
            for(;;)
            {
                auto msg=q->wait_and_pop();
                if(dispatch(msg))
                    break;
            }
        }

        bool dispatch(std::shared_ptr<message_base> const& msg)
        {
            if(wrapped_message<Msg>* wrapper=
                dynamic_cast<wrapped_message<Msg>*>(msg.get()))
            {
                f(wrapper->contents);
                return true;
            }
            else
            {
                return prev->dispatch(msg);
            }
```

TemplateDispatcher instantiations are friends of each other

1 If we handle the message, break out of the loop

Check the message type, and call the function 2

3 Chain to the previous dispatcher

[View full size image]

```
        }
    public:
        TemplateDispatcher(TemplateDispatcher&& other):
            q(other.q),prev(other.prev),f(std::move(other.f)),
            chained(other.chained)
        {
            other.chained=true;
        }
        TemplateDispatcher(queue* q_,PreviousDispatcher* prev_,Func&& f_):
            q(q_),prev(prev_),f(std::forward<Func>(f_)),chained(false)
        {
            prev_->chained=true;
        }

        template<typename OtherMsg,typename OtherFunc>
        TemplateDispatcher<TemplateDispatcher,OtherMsg,OtherFunc>
        handle(OtherFunc&& of)
        {
            return TemplateDispatcher<
                TemplateDispatcher,OtherMsg,OtherFunc>(
                    q,this,std::forward<OtherFunc>(of));
        }

        ~TemplateDispatcher() noexcept(false)
        {
            if(!chained)
            {
                wait_and_dispatch();
            }
        }
    };
}
```

❹ Additional handlers can be chained

❺ The destructor is noexcept(false) again

The `TemplateDispatcher<>` class template is modeled on the `dispatcher` class and is almost identical. In particular, the destructor still calls `wait_and_dispatch()` to wait for a message.

Since you don't throw exceptions if you handle the message, you now need to check whether you did ❶ handle the message in your message loop . Your message processing stops when you've successfully handled a message, so that you can wait for a different set of messages next time. If you ❷ do get a match for the specified message type, the supplied function is called rather than throwing an exception (although the handler function may throw an exception itself). If you don't ❸ get a match, you chain to the previous dispatcher . In the first instance, this will be a `dispatcher`, ❹ but if you chain calls to `handle()` to allow multiple types of messages to be handled, this may be a prior instantiation of `TemplateDispatcher<>`, which will in turn chain to the previous handler if the message doesn't match. Because any of the handlers might throw an exception (including the `dispatcher`'s default handler for `close_queue` messages), the destructor must once again be declared ❺ `noexcept(false)` .

This simple framework allows you to push any type of message on the queue and then selectively match against messages you can handle on the receiving end. It also allows you to pass around a

reference to the queue for pushing messages on, while keeping the receiving end private.

To complete the example from chapter 4, the messages are given in listing C.6, the various state machines in listings C.7, C.8, and C.9, and the driving code in listing C.10.

Listing C.6. ATM messages

```
struct withdraw
{
    std::string account;
     unsigned amount;
     mutable messaging::sender atm_queue;

     withdraw(std::string const& account_,
              unsigned amount_,
              messaging::sender atm_queue_):
        account(account_),amount(amount_),
        atm_queue(atm_queue_)
     {}
};

struct withdraw_ok
{};

struct withdraw_denied
{};

struct cancel_withdrawal
{
    std::string account;
    unsigned amount;

    cancel_withdrawal(std::string const& account_,
                      unsigned amount_):
         account(account_),amount(amount_)
     {}
};

struct withdrawal_processed
{
    std::string account;
    unsigned amount;

    withdrawal_processed(std::string const& account_,
                         unsigned amount_):
         account(account_),amount(amount_)
     {}
};
struct card_inserted
{
    std::string account;

    explicit card_inserted(std::string const& account_):
```

```
        account(account_)
    {}

};

struct digit_pressed
{
    char digit;

    explicit digit_pressed(char digit_):
        digit(digit_)
    {}

};

struct clear_last_pressed
{};

struct eject_card
{};

struct withdraw_pressed
{
    unsigned amount;

    explicit withdraw_pressed(unsigned amount_):
        amount(amount_)
    {}

};

struct cancel_pressed
{};

struct issue_money
{
    unsigned amount;
    issue_money(unsigned amount_):
        amount(amount_)
    {}
};

struct verify_pin
{
    std::string account;
    std::string pin;
    mutable messaging::sender atm_queue;

    verify_pin(std::string const& account_,std::string const& pin_,
               messaging::sender atm_queue_):
        account(account_),pin(pin_),atm_queue(atm_queue_)
    {}
};
```

```
struct pin_verified
{};
struct pin_incorrect
{};

struct display_enter_pin
{};

struct display_enter_card
{};

struct display_insufficient_funds
{};

struct display_withdrawal_cancelled
{};

struct display_pin_incorrect_message
{};

struct display_withdrawal_options
{};

struct get_balance
{
    std::string account;
    mutable messaging::sender atm_queue;

    get_balance(std::string const& account_,messaging::sender atm_queue_):
         account(account_),atm_queue(atm_queue_)
     {}
};

struct balance
{
    unsigned amount;

    explicit balance(unsigned amount_):
         amount(amount_)
     {}
};

struct display_balance
{
    unsigned amount;

    explicit display_balance(unsigned amount_):
         amount(amount_)
     {}
};

struct balance_pressed
{};
```

Listing C.7. The ATM state machine

```
class atm
{
    messaging::receiver incoming;
    messaging::sender bank;
    messaging::sender interface_hardware;
    void (atm::*state)();
    std::string account;
    unsigned withdrawal_amount;
    std::string pin;

    void process_withdrawal()
    {
        incoming.wait()
            .handle<withdraw_ok>(
                [&](withdraw_ok const& msg)
                {
                    interface_hardware.send(
                        issue_money(withdrawal_amount));
                    bank.send(
                        withdrawal_processed(account,withdrawal_amount));
                    state=&atm::done_processing;
                }
                )
            .handle<withdraw_denied>(
                [&](withdraw_denied const& msg)
                {
                   interface_hardware.send(display_insufficient_funds());
                    state=&atm::done_processing;
                }
                )
            .handle<cancel_pressed>(
                [&](cancel_pressed const& msg)
                {
                    bank.send(
                       cancel_withdrawal(account,withdrawal_amount));
                    interface_hardware.send(
                        display_withdrawal_cancelled());
                    state=&atm::done_processing;
                }
                );
    }

    void process_balance()
    {
        incoming.wait()
            .handle<balance>(
                [&](balance const& msg)
                {
                    interface_hardware.send(display_balance(msg.amount));
                    state=&atm::wait_for_action;
                }
```

```
                )
            .handle<cancel_pressed>(
                [&](cancel_pressed const& msg)
                {
                    state=&atm::done_processing;
                }
                );
}

void wait_for_action()
{
    interface_hardware.send(display_withdrawal_options());
     incoming.wait()
        .handle<withdraw_pressed>(
             [&](withdraw_pressed const& msg)
             {
                withdrawal_amount=msg.amount;
                bank.send(withdraw(account,msg.amount,incoming));
                state=&atm::process_withdrawal;
             }
             )
        .handle<balance_pressed>(
            [&](balance_pressed const& msg)
             {
                bank.send(get_balance(account,incoming));
                state=&atm::process_balance;
             }
             )
         .handle<cancel_pressed>(
             [&](cancel_pressed const& msg)
             {
                 state=&atm::done_processing;
             }
             );
}

void verifying_pin()
{
     incoming.wait()
        .handle<pin_verified>(
             [&](pin_verified const& msg)
             {
                 state=&atm::wait_for_action;
             }
             )
         .handle<pin_incorrect>(
             [&](pin_incorrect const& msg)
             {
                 interface_hardware.send(
                    display_pin_incorrect_message());
                 state=&atm::done_processing;
             }
             )
          .handle<cancel_pressed>(
```

```
            [&](cancel_pressed const& msg)
            {
                state=&atm::done_processing;
            }
            );
}

void getting_pin()
{
     incoming.wait()
         .handle<digit_pressed>(
             [&](digit_pressed const& msg)
             {
                 unsigned const pin_length=4;
                 pin+=msg.digit;
                 if(pin.length()==pin_length)
                 {
                    bank.send(verify_pin(account,pin,incoming));
                    state=&atm::verifying_pin;
                 }
             }
             )
         .handle<clear_last_pressed>(
             [&](clear_last_pressed const& msg)
             {
                 if(!pin.empty())
                 {
                    pin.pop_back();
                 }
             }
             )
         .handle<cancel_pressed>(
             [&](cancel_pressed const& msg)
             {
                 state=&atm::done_processing;
             }
             );
 }

 void waiting_for_card()
 {
    interface_hardware.send(display_enter_card());
    incoming.wait()
         .handle<card_inserted>(
            [&](card_inserted const& msg)
             {
                account=msg.account;
                pin="";
                interface_hardware.send(display_enter_pin());
                state=&atm::getting_pin;
             }
             );
 }
```

```
        void done_processing()
        {
           interface_hardware.send(eject_card());
           state=&atm::waiting_for_card;
        }

        atm(atm const&)=delete;
        atm& operator=(atm const&)=delete;

    public:
        atm(messaging::sender bank_,
           messaging::sender interface_hardware_):
           bank(bank_),interface_hardware(interface_hardware_)
        {}
        void done()
        {
           get_sender().send(messaging::close_queue());
        }

        void run()
        {
           state=&atm::waiting_for_card;
           try
           {
               for(;;)
                {
                   (this->*state)();
                }
            }
            catch(messaging::close_queue const&)
            {
            }
        }

        messaging::sender get_sender()
        {
            return incoming;
        }

    };
```

Listing C.8. The bank state machine

```
  class bank_machine
  {
      messaging::receiver incoming;
      unsigned balance;
  public:
      bank_machine():
          balance(199)
      {}

      void done()
```

```
{
    get_sender().send(messaging::close_queue());
}

void run()
{
    try
    {
         for(;;)
         {
            incoming.wait()
                .handle<verify_pin>(
                    [&](verify_pin const& msg)
                    {
                        if(msg.pin=="1937")
                        {
                            msg.atm_queue.send(pin_verified());
                        }

                         else
                        {
                            msg.atm_queue.send(pin_incorrect());
                        }
                     }
                     )
                .handle<withdraw>(
                    [&](withdraw const& msg)
                    {
                         if(balance>=msg.amount)
                         {
                            msg.atm_queue.send(withdraw_ok());
                            balance-=msg.amount;
                         }
                         else
                         {
                            msg.atm_queue.send(withdraw_denied());
                         }
                     }
                     )
                .handle<get_balance>(
                     [&](get_balance const& msg)
                     {
                        msg.atm_queue.send(::balance(balance));
                     }
                     )
                .handle<withdrawal_processed>(
                    [&](withdrawal_processed const& msg)
                    {
                    }
                    )
                .handle<cancel_withdrawal>(
                    [&](cancel_withdrawal const& msg)
                    {
                    }
```

```
                        );
                }
            }
            catch(messaging::close_queue const&)
            {
            }
        }

        messaging::sender get_sender()
        {
            return incoming;
        }
    };
```

Listing C.9. The user-interface state machine

```
class interface_machine
{
    messaging::receiver incoming;
public:
    void done()
    {
        get_sender().send(messaging::close_queue());
    }
    void run()
    {
        try
        {
            for(;;)
            {
                incoming.wait()
                    .handle<issue_money>(
                        [&](issue_money const& msg)
                        {
                            {
                                std::lock_guard<std::mutex> lk(iom);
                                std::cout<<"Issuing "
                                        <<msg.amount<<std::endl;
                            }
                        }
                        )
                    .handle<display_insufficient_funds>(
                        [&](display_insufficient_funds const& msg)
                        {
                            {
                                std::lock_guard<std::mutex> lk(iom);
                                std::cout<<"Insufficient funds"<<std::endl;
                            }
                        }
                        )
                    .handle<display_enter_pin>(
                        [&](display_enter_pin const& msg)
                        {
```

```
        {
             std::lock_guard<std::mutex> lk(iom);
             std::cout
                <<"Please enter your PIN (0-9)"
                <<std::endl;
        }
    }
    )
 .handle<display_enter_card>(
    [&](display_enter_card const& msg)
    {
        {
             std::lock_guard<std::mutex> lk(iom);
             std::cout<<"Please enter your card (I)"
                      <<std::endl;
        }
    }
    )
.handle<display_balance>(
    [&](display_balance const& msg)
    {
        {
             std::lock_guard<std::mutex> lk(iom);
             std::cout
                <<"The balance of your account is "
                <<msg.amount<<std::endl;
        }
     }
     )
.handle<display_withdrawal_options>(
    [&](display_withdrawal_options const& msg)
    {
        {
             std::lock_guard<std::mutex> lk(iom);
             std::cout<<"Withdraw 50? (w)"<<std::endl;
             std::cout<<"Display Balance? (b)"
                      <<std::endl;
             std::cout<<"Cancel? (c)"<<std::endl;
        }
     }
     )
.handle<display_withdrawal_cancelled>(
     [&](display_withdrawal_cancelled const& msg)
     {
         {
             std::lock_guard<std::mutex> lk(iom);
             std::cout<<"Withdrawal cancelled"
                      <<std::endl;
         }
     }
     )
.handle<display_pin_incorrect_message>(
     [&](display_pin_incorrect_message const& msg)
     {
```

```
                                {
                                    std::lock_guard<std::mutex> lk(iom);
                                    std::cout<<"PIN incorrect"<<std::endl;
                                }
                            }
                            )
                        .handle<eject_card>(
                            [&](eject_card const& msg)
                            {
                                {
                                    std::lock_guard<std::mutex> lk(iom);
                                    std::cout<<"Ejecting card"<<std::endl;
                                }
                            }
                            );
                }
            }
            catch(messaging::close_queue&)
            {
            }
        }
        messaging::sender get_sender()
        {
            return incoming;
        }
    };
```

Listing C.10. The driving code

```
    int main()
    {
        bank_machine bank;
        interface_machine interface_hardware;

        atm machine(bank.get_sender(),interface_hardware.get_sender());

        std::thread bank_thread(&bank_machine::run,&bank);
        std::thread if_thread(&interface_machine::run,&interface_hardware);
        std::thread atm_thread(&atm::run,&machine);

        messaging::sender atmqueue(machine.get_sender());

        bool quit_pressed=false;

        while(!quit_pressed)
        {
            char c=getchar();
            switch(c)
            {
            case '0':
            case '1':
            case '2':
            case '3':
```

```
                case '4':
                case '5':
                case '6':
                case '7':
                case '8':
                case '9':
                    atmqueue.send(digit_pressed(c));
                    break;
                case 'b':
                    atmqueue.send(balance_pressed());
                    break;
                case 'w':
                    atmqueue.send(withdraw_pressed(50));
                    break;
                case 'c':
                    atmqueue.send(cancel_pressed());
                    break;
                case 'q':
                    quit_pressed=true;
                    break;
                case 'i':
                    atmqueue.send(card_inserted("acc1234"));
                    break;
                }
            }
        bank.done();
        machine.done();
        interface_hardware.done();
        atm_thread.join();
        bank_thread.join();
        if_thread.join();
    }
```