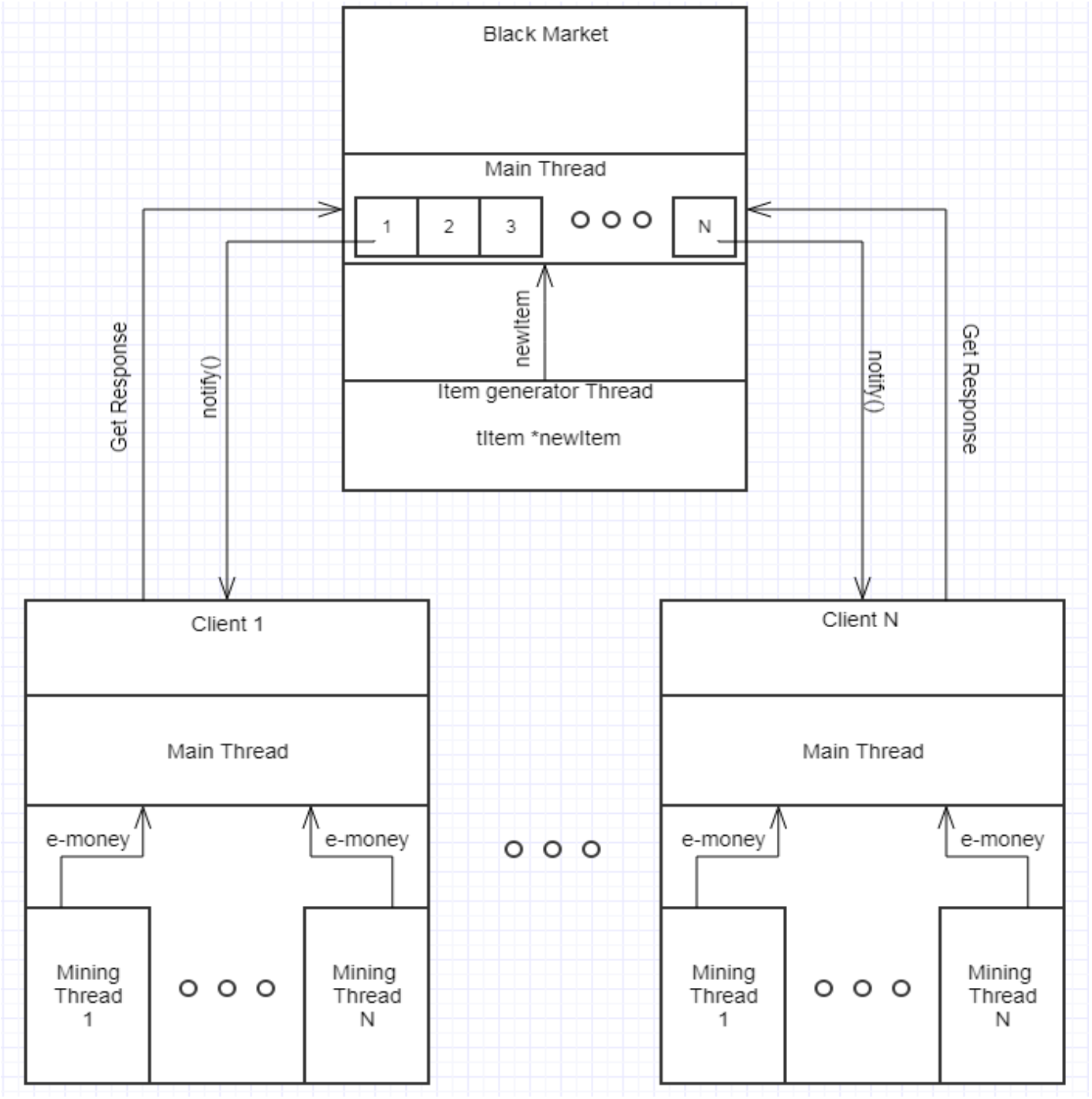
Test task: “Evil Corporation”

Done by Maxim Karpovich

1. Concept

The idea is to create a software system that imitates client-server application representing clients and “black market” objects as a separate threads respectively. After “black market” object is created, every new client can be connected (subscribed) to it. When “black market” generates new item it notifies every subscribed client and according to their answer performs random choice between them. The behavior of such random choice is similar as when clients saw updates simultaneously .

2. Application scheme



3. Realization

3.1 Implemented Patterns

3.1.1. Observer.

"Black market" represents itself an observer class. Lets clients to subscribe on it`s updates and unsubscribe. Also has own private methods to process clients data. When the object of this class is created, first of all it registers available item types (Coin, Jug, Helm, Sword). Then it launches a separate thread that is responsible for generating new items through external API.

```
class BlackMarket
{
public:
	BlackMarket();
	~BlackMarket();
	tClientId subscribe(tClientPtr pNewClient);
	bool unsubscribe(tClientId id);
private:
	/*Begin generating items*/
	void start_trading();
	/*Tell all subscribers that new item is available*/
	void notify(tItem);
	/*Send a pointer to new item*/
	void send_item(tItem);
	/*Collect answers from clients about readiness to buy item*/
	void collect_responses();
	/*Generate new rare item*/
	void generate_item();
private:
	/*Storage for all subscribed clients*/
	tClientList mClientList;
	/*Storage for all subscribed clients that ready to by current item*/
	tClientList mClientsReadyToBuy;
	/*IPC for accessing shared objects*/
	tMutex mMutex;
	/*Thread that generates items*/
	tThread mItemGeneratorThread;
	/*Templated factory for creating needed item*/
	tItemFactory *mItemsFactory;

	};
```

3.1.2. Template Fabric.

"Item Fabric" serves as an object that is responsible for creating specified item object. It is initialized in *BlackMarket* constructor with all available item types (Coin, Jug,

Helm, Sword). When *BlackMarket* gets newly generated item, it created corresponding item through Item Fabric. Idea of the fabric is to create *Creator* class that would be responsible for creating an item object.

```
emplate <class BaseType, class Key>
class EquipmentFactory
{
public:
	void register_item(Key Id, Creator<BaseType>* Fn)
	{
		FunctionMap[Id] = Fn;
	}
	BaseType* create(Key Id)
	{
		return FunctionMap[Id]->create();
	}
	~EquipmentFactory()
	{
		std::map<Key, Creator<BaseType>*>::iterator i = FunctionMap.begin();
		while (i != FunctionMap.end())
		{
			delete (*i).second;
			++i;
		}
	}
private:
	std::map<Key, Creator<BaseType>*> FunctionMap;
};
```

3.1.3. Singleton.

"ApiSingleton" serves as single object that is responsible for providing interface to external API. It loads the api.dll library into the application address space and also provides class *api*.

Implementation is so called Meyers singleton with lazy instantiation.

```
class ApiSingleton {
private:
	ApiSingleton() {
		try {
			auto hd1 = LoadLibraryA("headers/ExternalAPI/api.dll");
			if (hd1) {
				mApiInterface = reinterpret_cast<apiPtr>(GetProcAddress(hd1, "api"));

				if (!mApiInterface) {
					throw ApiAccessException("Could not get proc address");
				}
			}
			else
				throw MemoryException("Could not load api.dll");
		}
```

```
                catch (MemoryException const &ex) {
                        std::cerr << "ApiSingleton: " << ex.what() << std::endl;
                        throw;
                }
                catch (ApiAccessException const &ex) {
                        std::cerr << "ApiSingleton: " << ex.what() << std::endl;
                        throw;
                }
        }
        ~ApiSingleton() {};
public:
        ApiSingleton(ApiSingleton const&) = delete;
        ApiSingleton& operator=(ApiSingleton const&) = delete;

        /*Create new rare item. Means invoke api generator function with custom callback*/
        void generate_new_item() {
                mApiInterface->get_raw_rare(&ApiSingleton::process_item);
        }
        /*Save currently generated rare thing*/
        static void process_item(const byte *thing, const size_t size) {
                mThing = (byte*)malloc(size);
                strcpy((char *)ApiSingleton::mThing, (char *)thing);
                mSize = size;
        };
        static ApiSingleton &get_instance() {
                static ApiSingleton mInstance;
                return mInstance;
        }
private:
        apiPtr mApiInterface;
        static ApiSingleton *mInstance;

public:
        static byte *mThing;
        static size_t mSize;
          };
```

3.2 Implemented idioms

3.2.1 RAII

3.2.1.1. ThreadPool

ThreadPool is a class that is responsible for creating/deleting threads. It also gives a thread a function to execute. User does not have to worry about non finished threads as when threadpool is deleted, it finishes all created jobs.

```
class ThreadPool {
public:
    ThreadPool(size_t);
```

```
    template<class F, class... Args>
    auto enqueue(F&& f, Args&&... args) -> std::future<typename
std::result_of<F(Args...)>::type>;
    ~ThreadPool();
private:
    /*Need to keep track of threads so we can join them*/
    std::vector< std::thread > workers;
    /*The task queue*/
    std::queue< std::function<void()> > tasks;

    /*Synchronization*/
    std::mutex queue_mutex;
    std::condition_variable condition;
    bool stop;
};
```

3.2.1.2. Locks

Places where shared objects are changed within threads are zoned with lockers. Lockers itself represent an object that manages blocking resource, such as mutex. So user does not have to release resource manually as when the locker will be out of it`s namespace it will release the source.

```
{ //lock
      tLockGuard lock(mMutex);
      this->notify(item);
      this->collect_responses();
      this->send_item(item);
} //Release
```

3.2.2. STL usage

C++ is getting more and more usefull features with new standards. In this project following features were used:

- std::vector
- std::string
- std::queue
- std::map

- std::mutex
- std::lock_guard
- std::thread
- std::chrono

- std::future

- *std::result_of*
- *std::condition_variable*
- *std::packaged_task*
- *std::forward*

3.2.3 Smart Pointers

Usage of smart pointers sets the responsibility for deleting the pointer on the object that created this pointer. That leads to decreasing memory leaks.

- *std::unique_ptr*
- *std::shared_ptr*

3.2.4. Exception handling

Mechanism for runtime problems detecting. In this project two custom classes where inherited from std::exception for demonstration purposes.

```
class MemoryException : public std::runtime_error {
public:
	MemoryException (std::string mess) : std::runtime_error(mess) {}
};
```

3.2.5 OOP principles

Good example in this project may be the item family inheritance. Base item class is an interface.

```
class Equipment {
public:
	Equipment() {};
	virtual ~Equipment() {};

	/*Construct object depending on it`s type*/
	virtual void set_item_info(const byte* thing, const size_t size) = 0;
	/*Return description depending on type*/
	virtual tString get_description() = 0;
	virtual Value get_cost() = 0;
};
```

Each heir implements abstract functions in its own way.

```
using tHelmData = struct HelmData
{
	Value cost;
	Size  size;
	Value armor;
};
```

```
class Helm : public Equipment
{
public:
	Helm() {};
	~Helm() {};

	virtual void set_item_info(const byte* thing, const size_t size);
	virtual tString get_description();
	virtual Value get_cost();
private:
	tHelmData mHelmData;
	evil::RareType mType;
	 };

void Helm::set_item_info(const byte* thing, const size_t size) {
	memcpy((void*)&mHelmData, (void*)&thing[1], sizeof(tHelmData));
	mType = (evil::RareType)thing[0];
}
tString Helm::get_description() {
	return "Helm"; //can be any string containing object data.
}

Value Helm::get_cost() {
	return mHelmData.cost;
}
```