Easymock: Innovative mocking for unit tests

Problem statement

Some mocking frameworks, such as Google Mock, require your code base to **conform to specific constraints imposed by the framework.**

Innovative solution

By leveraging **code generation**, Easymock imposes no restrictions, allowing any **class or free function to be mocked** seamlessly.

Additionally, Easymock **utilizes libclang and LLVM**, ensuring compatibility with the latest C++ standards.

Smarter mock, better tests

The generated mocks include "ExpectAndReturn" functions to define the expectation and return values, enabling precise testing of edge cases and failure scenarios.

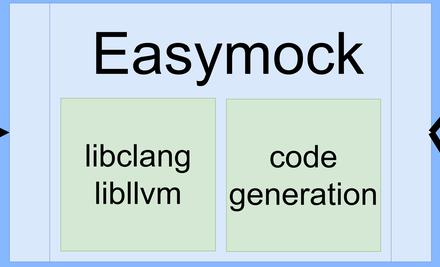
By generating code, Easymock provides a maintainable, efficient, and developer-friendly approach to writing fast and debuggable unit tests

Use code generation, not virtual functions

database.hpp

```
#include <string>

class Database {
public:
    bool connect(const std::string url);
    int fetchData(std::string id);
};
```



easymock_database.hpp

```
namespace EasyMock::Database { // Namespace for the Database mocked class
namespace connect { // Namespace for configuring the connect mocked function

void ExpectAndReturn
        (::Database* instance, const std::string url, bool to_return,
         ::EasyMock::EasyMock_Matcher_Cpp<::std::string> easyMock_match_url);

} // namespace connect
namespace fetchData { // Namespace for configuring the fetchData mocked function

void ExpectAndReturn
  (::Database* instance, ::std::;string id, int to_return,
   ::EasyMock::EasyMock_Matcher_Cpp<::std::string> easyMock_match_id);

} // namespace fetchData
} // namespace EasyMock::Database
```

easymock_database.cpp

```
// Mock implementation ...
namespace EasyMock::Database {
// Mock ExpectAndReturn implementation...
}
```

Code under test

```
class DataProcessor {
    Database& db;
public:
    DataProcessor(Database& database) : db(database) {}

    int process(std::string id) {
        if (!db.connect("db://test")) {
            return -1;
        }
        int value = db.fetchData(id);
        return value * 2; // some processing
    }
};
```

Test DB access OK scenario

```
#include "easymock_database.hpp"

TEST(DataProcessorTest, ProcessesCorrectly) {
    Database db;
    DataProcessor processor(db);

    // Expect the call to connect on our db instance --> return true
    EasyMock::Database::connect::ExpectAndReturn
        (&db, "db://test", true,
         &::EasyMock::EasyMock_Matcher_Cpp<::std::string>);
    // Expect the call to fetchData on our db instance.
    // Verify the parameter is "Alice" and return 10
    EasyMock::Database::fetchData::ExpectAndReturn
        (&db, "Alice", 10, &::EasyMock::EasyMock_Matcher_Cpp<::std::string>);

    EXPECT_EQ(processor.process("Alice"), 20); // 10 * 2 == 20
}
```

Test DB connection fail scenario

```
#include "easymock_database.hpp"

TEST(DataProcessorTest, DBConnectionFail) {
    Database db;
    DataProcessor processor(db);

    // Expect the call to connect on our db instance --> return false
    EasyMock::Database::connect::ExpectAndReturn
        (&db, "db://test", false,
        &::EasyMock::EasyMock_Matcher_Cpp<::std::string>);

    EXPECT_EQ(processor.process("Alice"), -1); // -1 == error
}
```

Advantages

- ✅ No interface needed
- ✅ Easily usable on legacy code
- ✅ Compatible with any unit test framework

What can you mock?

- Access to databases
- Access to file system
- Access to HW register
- Linux Kernel driver
- -...

More information

FOSDEM talk **Github**

Author: Laurent Carlier