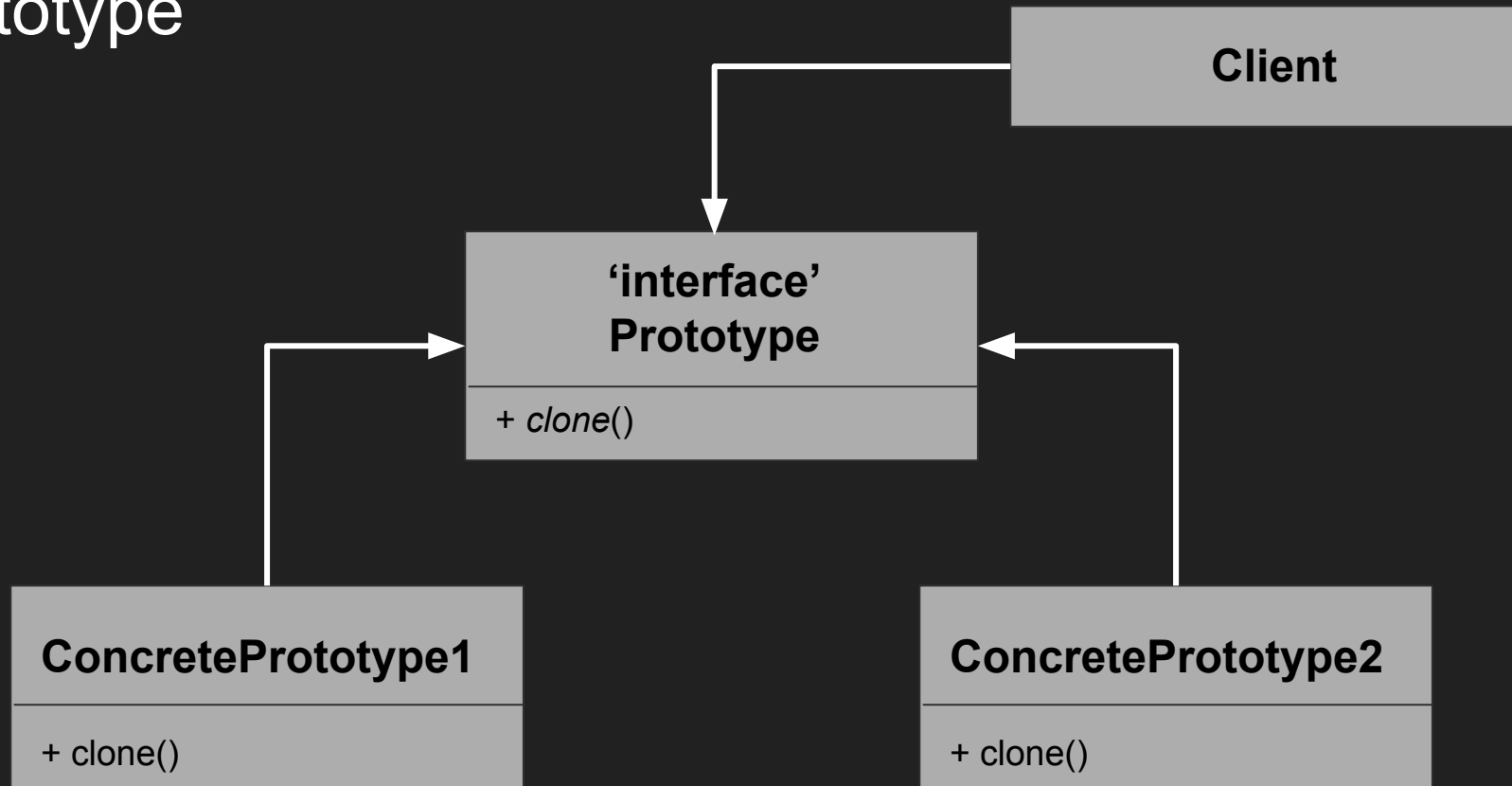


# Технологии программирования

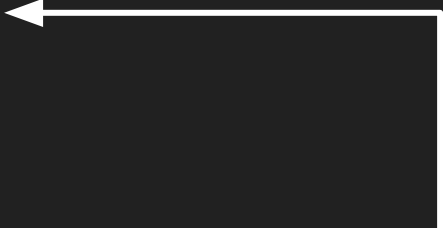
Паттерны проектирования. Порождающие  
паттерны. Prototype, Singleton. Системы сборки.  
CMake.

# Prototype



# Prototype

```
class ISpaceshipPrototype {  
public:  
    virtual ~CSpaceship();  
    virtual void do_something();  
    virtual CSpaceship* clone() const = 0;  
};
```



```
class CFighter : public ISpaceshipPrototype {  
public:  
    CFighter(const CFighter&);  
    void do_something();  
    CFighter* clone() const { return new CFighter(*this); }  
};
```

# Prototype VS copy constructor



# Prototype VS copy constructor

взаимодействие с объектом через интерфейс

# Java: Cloneable interface

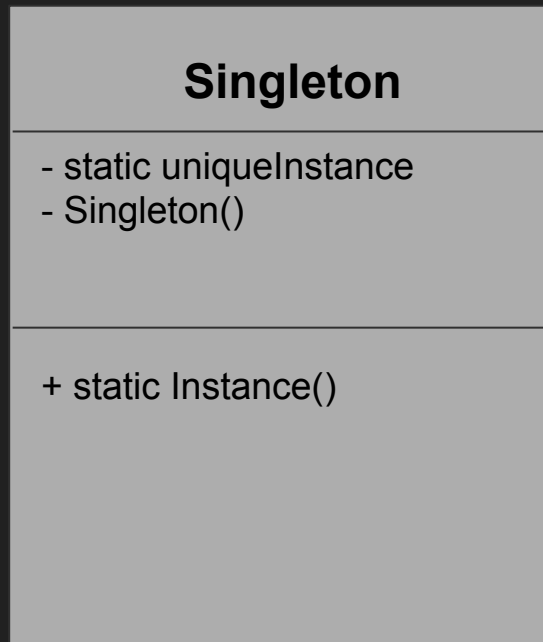
```
ArrayList<Integer> list = new ArrayList<Integer>();  
for (int i = 0; i < 10; i++) {  
    list.add(i);  
}  
....
```

```
ArrayList<Integer> duplicatedSet = (ArrayList<Integer>) list.clone();
```

# Порождающие паттерны

Factory Method	создание объектов, объединенных одним интерфейсом
Abstract Factory	множество фабричных методов
Builder	пошаговое создание объекта
Prototype	создание через копирование

# Singleton



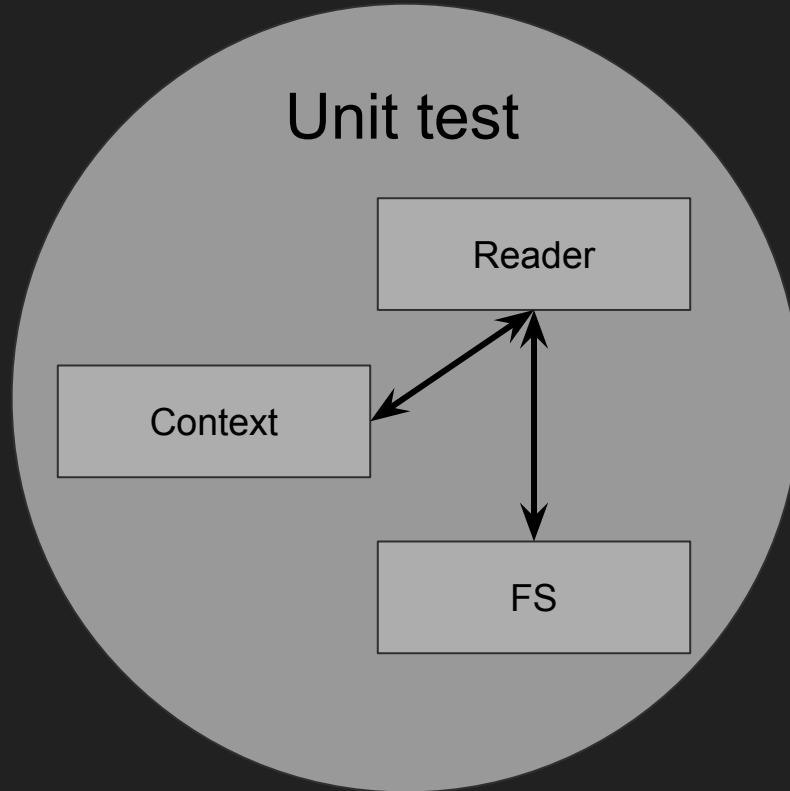


# Singleton

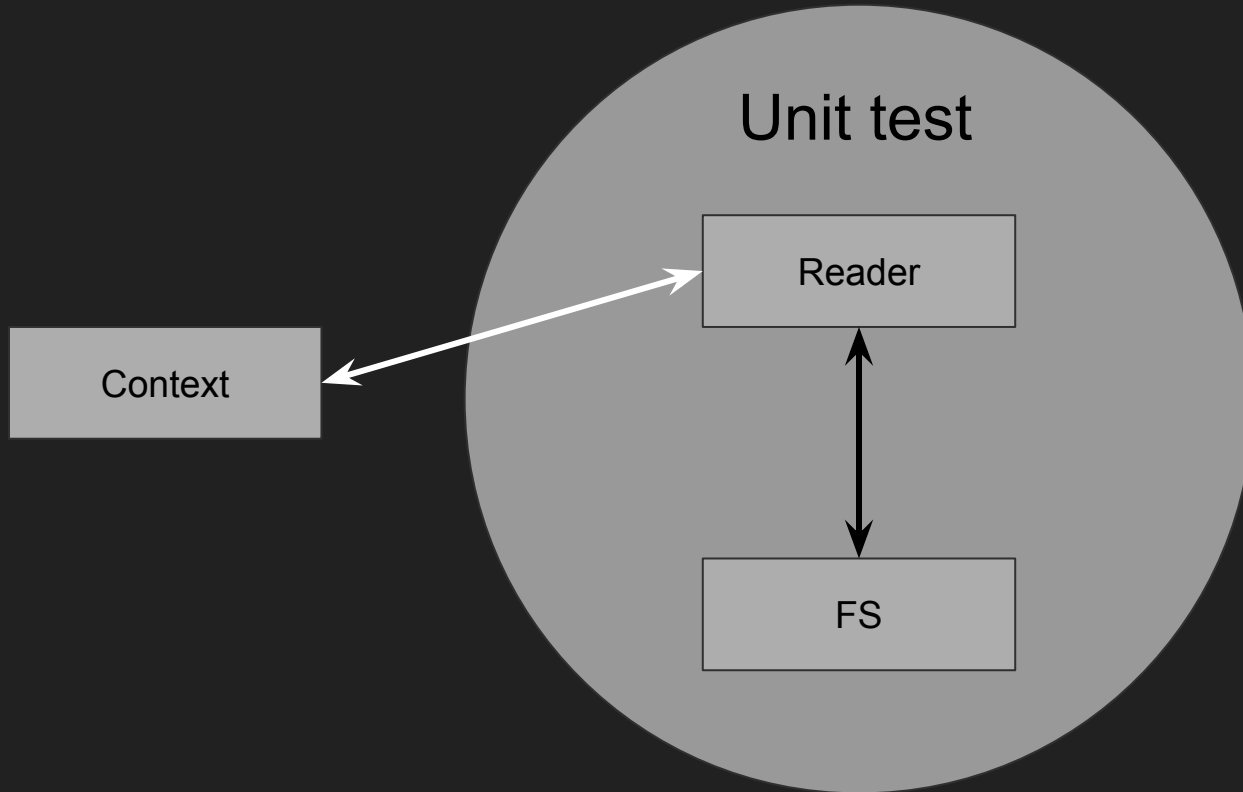
```
template<class T>
class CSingleton {
private:
    static std::shared_ptr<T> m_instance;
public:
    CSingleton() = delete;
    ~CSingleton() = delete;
    CSingleton(const CSingleton&) = delete;
    CSingleton& operator=(const CSingleton&) = delete;

    static std::shared_ptr<T> instance() {
        if (m_instance.empty())
            m_instance = std::shared_ptr<T>(new T);
        return m_instance;
    }
};
```

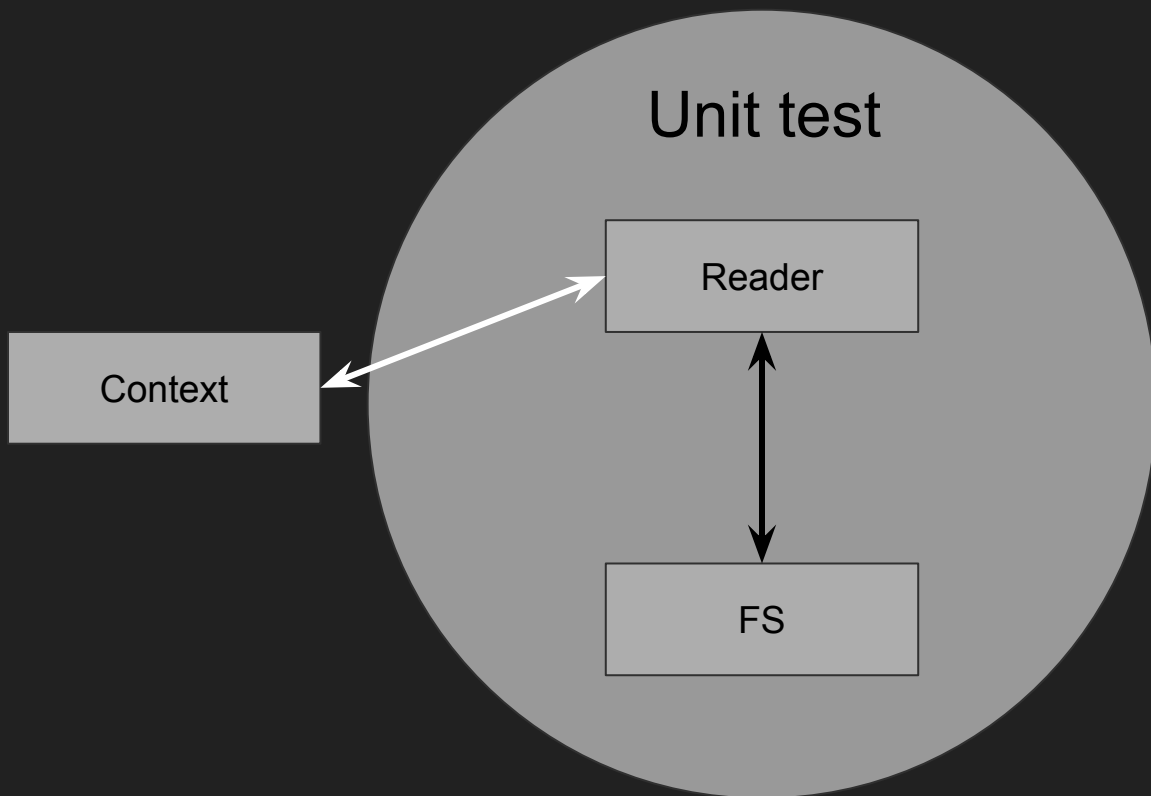
# Singleton. Cons.



# Singleton. Cons.



# Singleton. Cons.



**Состояние синглтона  
может меняться!**

- порядок тестов имеет значение
- нельзя запустить тесты параллельно
- воспроизводимость тестов нарушена

# Сборка. Сборочные скрипты

```
#!/bin/bash
```

```
CFLAGS="-std=c++11 -Wall"
```

```
SOURCES="hello.cpp"
```

```
echo "Build started"
```

```
g++ $CFLAGS $SOURCES -o bot
```

```
...
```

```
echo "Build finished"
```

# Сборка. Сборочные скрипты

```
#!/bin/bash
```

```
CFLAGS="-std=c++11 -Wall"
```

```
SOURCES="hello.cpp bye.cpp"
```

```
echo "Build started"
```

```
g++ $CFLAGS $SOURCES -o bot
```

```
...
```

```
echo "Build finished"
```

# Сборка. Сборочные скрипты

```
#!/bin/bash
```

```
CFLAGS="-std=c++11 -Wall"  
SOURCES="hello.cpp bye.cpp"
```

```
echo "Build started"  
g++ $CFLAGS $SOURCES -o bot
```

```
...
```

```
echo "Build finished"
```

```
$ chmod +x build.sh
```

```
$ ./build.sh
```

```
$ ./bot
```

# Сборка. Сборочные скрипты

```
#!/bin/bash
```

```
CFLAGS="-std=c++11 -Wall"  
SOURCES="hello.cpp bye.cpp"
```

```
echo "Build started"  
g++ $CFLAGS $SOURCES -o bot  
...  
echo "Build finished"
```

подходят для  
небольших проектов

плохо подходят для  
сборки больших  
проектов



# Сборка. Makefiles

```
CFLAGS="-std=c++11 -Wall"
```

```
hello.o: hello.cpp
```

```
    g++ ${CFLAGS} -c -o hello.o hello.cpp
```

```
bye.o: bye.cpp
```

```
    g++ ${CFLAGS} -c -o bye.o bye.cpp
```

```
bot: bye.o hello.o
```

```
    g++ -o bot hello.o bye.o
```

# Сборка. Makefiles

```
CFLAGS="-std=c++11 -Wall"
```

```
hello.o: hello.cpp
```

```
    g++ ${CFLAGS} -c -o hello.o hello.cpp
```

```
bye.o: bye.cpp
```

```
    g++ ${CFLAGS} -c -o bye.o bye.cpp
```

```
bot: bye.o hello.o
```

```
    g++ -o bot hello.o bye.o
```

```
$ make bot
```

```
$ ./bot
```

# Сборка. Makefiles

```
CFLAGS="-std=c++11 -Wall"
```

```
hello.o: hello.cpp
```

```
    g++ ${CFLAGS} -c -o hello.o hello.cpp
```

```
bye.o: bye.cpp
```

```
    g++ ${CFLAGS} -c -o bye.o bye.cpp
```

```
bot: bye.o hello.o
```

```
    g++ -o bot hello.o bye.o
```

инкрементальная  
сборка хороша для  
больших проектов

не подходят для  
кроссплатформенных  
продуктов

# Сборка. CMake

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```

# Сборка. CMake

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```

```
$ ls
bye.cpp CMakeLists.txt hello.cpp
$ cmake
$ make
$ ./bot
```

# CMake. Директории с заголовочными файлами

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
# укажем каталог с заголовочными файлами
include_directories("include")
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```

# CMake. Подпроекты

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
# укажем каталог с подпроектом
add_subdirectory("ai_module")
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```

# CMake. Флаги

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
if(WITH_AI)
    add_subdirectory("ai_module")
endif()
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```



# CMake. Флаги

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
if(WITH_AI)
    add_subdirectory("ai_module")
endif()
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```

```
$ cmake -DWITH_AI=1
$ make
$ ./bot
```

# CMake. Использование сторонних библиотек

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
# найдем Boost
find_package(Boost REQUIRED)
if(NOT Boost_FOUND)
    message(SEND_ERROR "Failed to find boost library")
    return()
else()
    include_directories(${Boost_INCLUDE_DIRS})
endif()
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```

# CMake. Генераторы

```
# проверка версии
cmake_minimum_required(VERSION 2.8)
# название проекта
project(bot)
# установим флаги компилятора
set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -std=c++11
-Wall")
# перечислим все исходники
set(SOURCES_LIST "hello.cpp bye.cpp")
if(WITH_AI)
    add_subdirectory("ai_module")
endif()
# создать исполняемый файл
add_executable(bot ${SOURCES_LIST})
```

```
$ cmake -G "KDevelop3 — Unix Makefiles"
$ make
$ ./bot
```

# CMake. Список генераторов

```
$ cmake --help
```

```
...
```

```
Generators
```

The following generators are available on this platform:

Unix Makefiles = Generates standard UNIX makefiles.

Ninja = Generates build.ninja files.

Watcom WMake = Generates Watcom WMake makefiles.

Sublime Text 2 - Ninja = Generates Sublime Text 2 project files.

Sublime Text 2 - Unix Makefiles = Generates Sublime Text 2 project files.

Kate - Ninja = Generates Kate project files.

Kate - Unix Makefiles = Generates Kate project files.

Eclipse CDT4 - Ninja = Generates Eclipse CDT 4.0 project files.

Eclipse CDT4 - Unix Makefiles = Generates Eclipse CDT 4.0 project files.

KDevelop3 = Generates KDevelop 3 project files.

KDevelop3 - Unix Makefiles = Generates KDevelop 3 project files.