Your Compiler Understands It, But Does Anyone Else ?

10 Tips for Cleaner C++ 20 Code

DAVID SACKSTEIN

```
9617        // clang-format off
9618        template <input_range _Rng, class _Pj = identity,
9619            indirect_strict_weak_order<projected<iterator_t<_Rng>, _Pj>> _Pr = ranges::less>
9620            requires indirectly_copyable_storable<iterator_t<_Rng>, range_value_t<_Rng>*>
9621        _NODISCARD constexpr minmax_result<range_value_t<_Rng>> operator()(
9622            _Rng&& _Range, _Pr _Pred = {}, _Pj _Proj = {}) const {
9623            // clang-format on
9624            auto _UFirst = _Ubegin(_Range);
9625            auto _ULast  = _Uend(_Range);
9626            _STL_ASSERT(
9627                _UFirst != _ULast, "A range passed to std::ranges::minmax must not be empty. (N4861 [alg.min.max]/21)");
9628            using _Vty = range_value_t<_Rng>;
9629            if constexpr (forward_range<_Rng> && _Prefer_iterator_copies<iterator_t<_Rng>>) {
9630                const auto _Found = _RANGES _Minmax_element_unchecked(
9631                    _STD move(_UFirst), _STD move(_ULast), _Pass_fn(_Pred), _Pass_fn(_Proj));
9632                return {static_cast<_Vty>(*_Found.min), static_cast<_Vty>(*_Found.max)};
9633            } else {
9634                minmax_result<_Vty> _Found = {static_cast<_Vty>(*_UFirst), static_cast<_Vty>(*_UFirst)};
9635                if (_UFirst == _ULast) {
9636                    return _Found;
9637                }
9638
9639                while (++_UFirst != _ULast) { // process one or two elements
9640                    auto _Prev = *_UFirst;
9641                    if (++_UFirst == _ULast) { // process last element
9642                        if (_STD invoke(_Pred, _STD invoke(_Proj, _Prev), _STD invoke(_Proj, _Found.min))) {
9643                            _Found.min = _STD move(_Prev);
9644                        } else if (!_STD invoke(_Pred, _STD invoke(_Proj, _Prev), _STD invoke(_Proj, _Found.max))) {
9645                            _Found.max = _STD move(_Prev);
9646                        }
9647
9648                        break;
9649                    }
9650
9651                    // process next two elements
9652                    if (_STD invoke(_Pred, _STD invoke(_Proj, *_UFirst), _STD invoke(_Proj, _Prev))) {
9653                        // test _UFirst for new smallest
9654                        if (_STD invoke(_Pred, _STD invoke(_Proj, *_UFirst), _STD invoke(_Proj, _Found.min))) {
9655                            _Found.min = *_UFirst;
9656                        }
9657
9658                        if (!_STD invoke(_Pred, _STD invoke(_Proj, _Prev), _STD invoke(_Proj, _Found.max))) {
9659                            _Found.max = _STD move(_Prev);
9660                        }
```

```
2875    namespace views {
2876        class _Drop_fn {
2877        private:
2878            enum class _St { _Empty, _Reconstruct_span, _Reconstruct_subrange, _Reconstruct_other, _Drop_view };
2879
2880            template <class _Rng>
2881            _NODISCARD static _CONSTEVAL _Choice_t<_St> _Choose() noexcept {
2882                using _Ty = remove_cvref_t<_Rng>;
2883
2884                if constexpr (_Is_specialization_v<_Ty, empty_view>) {
2885                    return {_St::_Empty, true};
2886                } else if constexpr (_Is_span_v<_Ty>) {
2887                    return {_St::_Reconstruct_span, true};
2888                } else if constexpr (_Is_specialization_v<_Ty, basic_string_view>) {
2889                    return {_St::_Reconstruct_other, true};
2890                } else if constexpr (_Random_sized_range<_Ty> && _Is_subrange<_Ty>) {
2891                    if constexpr (sized_sentinel_for<sentinel_t<_Ty>, iterator_t<_Ty>>) {
2892                        return {_St::_Reconstruct_subrange,
2893                            noexcept(_Ty(_RANGES begin(_STD declval<_Rng&>()) + _RANGES distance(_STD declval<_Rng&>()),
2894                                _RANGES end(_STD declval<_Rng&>())))};
2895                    } else {
2896                        return {_St::_Reconstruct_subrange,
2897                            noexcept(_Ty(_RANGES begin(_STD declval<_Rng&>()) + _RANGES distance(_STD declval<_Rng&>()),
2898                                _RANGES end(_STD declval<_Rng&>()), range_difference_t<_Rng>{0}))};
2899                    }
2900                } else if constexpr (_Random_sized_range<_Ty> && _Is_specialization_v<_Ty, iota_view>) {
2901                    return {_St::_Reconstruct_other,
2902                        noexcept(_Ty(_RANGES begin(_STD declval<_Rng&>()) + _RANGES distance(_STD declval<_Rng&>()),
2903                            _RANGES end(_STD declval<_Rng&>())))};
2904                } else {
2905                    return {_St::_Drop_view, noexcept(drop_view(_STD declval<_Rng>(), range_difference_t<_Rng>{0}))};
2906                }
2907            }
2908
2909            template <class _Rng>
2910            static constexpr _Choice_t<_St> _Choice = _Choose<_Rng>();
```

Introduction

- My name is David Sackstein (davids@codeprecise.com)
- I am an independent consultant, developer and instructor.
- I work with C++ and a few other languages
- And I am passionate about writing clean code!

Implementation of the STL

- APIs are extremely well designed and thoroughly reviewed.
- Consistent and compatible over many platforms
- Efficient as possible.
- Complies with many of the core guidelines
- But…

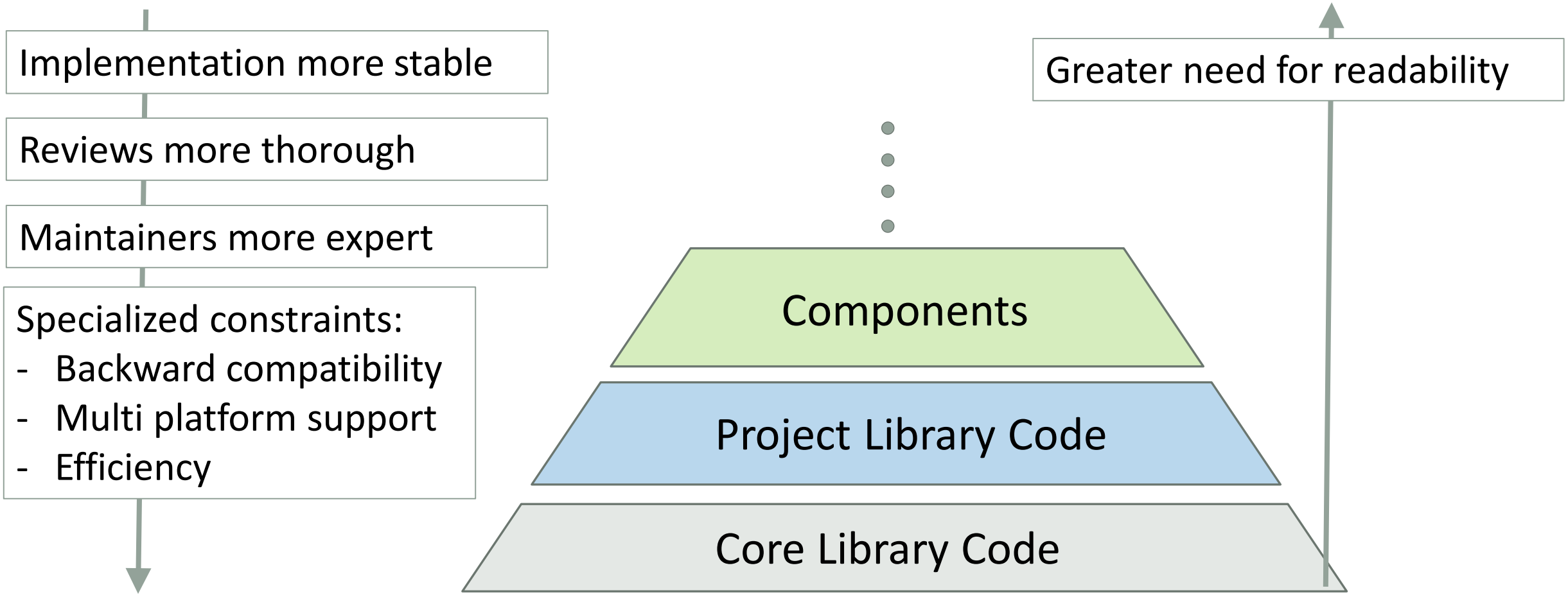
Implementation of the STL

- Classes and functions are long.
- Cyclomatic complexity is often high
- Header files are very long
  - chrono has almost 6000 lines
  - algorithm over 9000.
- Excessive use of macros and compilation flags.
- Very repetitive

Agenda

- Clean Code in Context
- Best Practices for C++ Programming
- Develop an application and clean it.
- Lessons learnt.
- Discussion

Clean code tradeoffs



Best Practices for Software Design

- SOLID - Coined by Robert Martin and Michael Feathers
  - S ingle Responsibility Principle – small pieces that do one thing.
  - O pen Closed Principle – define abstractions for extensibility.
  - L iskov Substitution Principle – be careful with what you inherit.
  - I nterface Segregation Principle – like SRP but with interfaces.
  - D ependency Inversion Principle – depend on abstractions.
- In addition, for C++
  - The C++ Core Guidelines https://isocpp.github.io/CppCoreGuidelines/CppCoreGuidelines

The clean-it repo

- https://github.com/david-sackstein/clean-it.git
- Uses
  - Platform Toolset: Visual Studio 2022 (v143)
  - Language setting: ISO C++20 Standard (/std:c++20)
  - GoogleTest for unit tests
  - Resharper Tools for C++ with clang-tidy checks and fixes enabled.
- Code lives in the `ci` namespace
- There are tags for each step – these appear on the slides.

The VOD Application

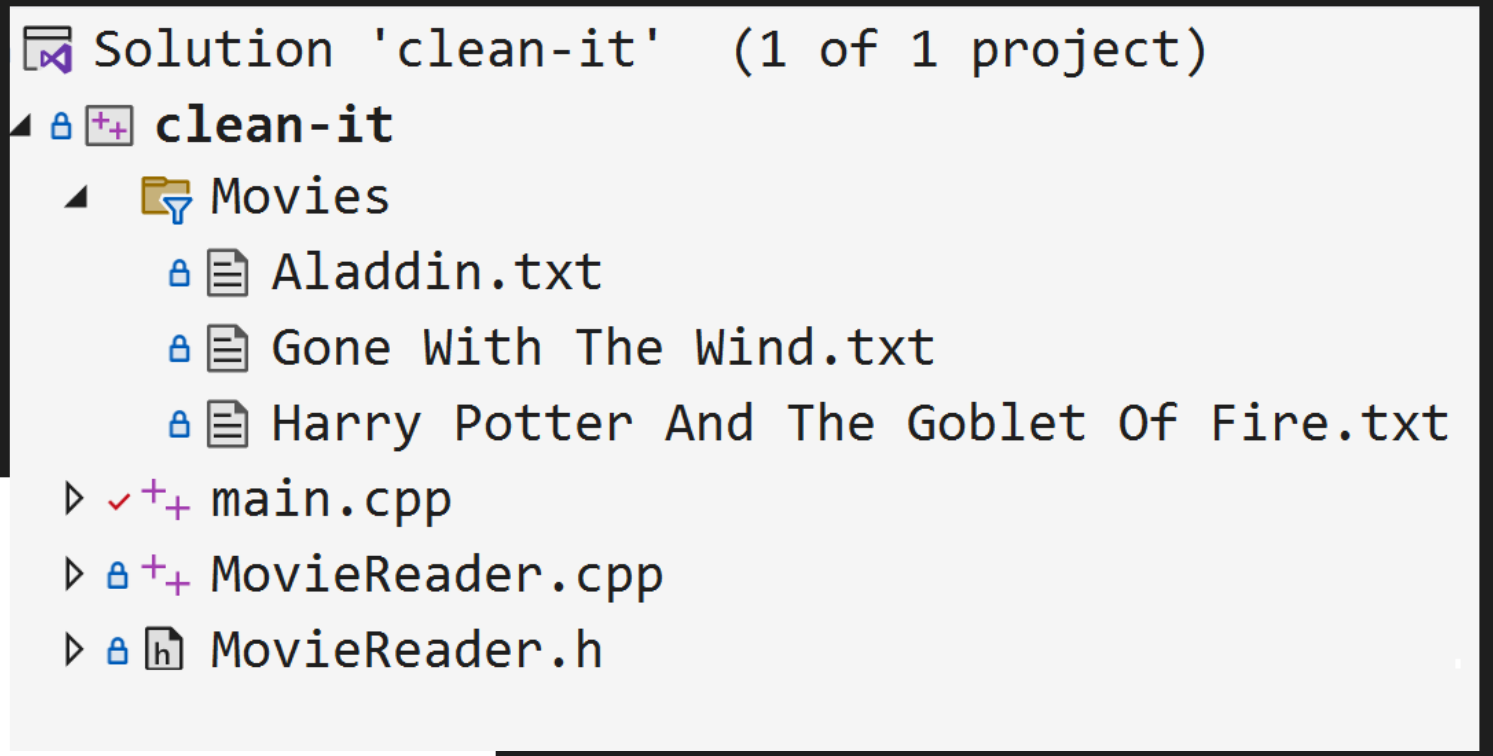
- The VODServer
  - Accepts a connection from VODClients
  - Provides a list of available movies to a connected client.
  - Streams a movie to a connected client.
  - A movie can be stopped by the client and the client can disconnect.
- We will develop:
  - MovieReader – reads the movies from disk.
  - VODServer, VODClient

Use the compiler and tooling

- Make warnings errors
- Run clang-tidy directly or through a tool.
- Turn on warnings and fix options (Resharper does this well).
- Apart from cleaning the code you can learn from them too.

The initial solution

… and a test




```cpp
#include <gtest/gtest.h>

#include <vector>

#include "MovieReader.h"

TEST(MovieReader, ReadSucceeds)
{
    const auto movies = ci::readMovies();
    ASSERT_EQ(3, movies.size());
}
```

git tag: v1-MovieReader

MovieReader.h

```
1  #pragma once
2
3  #include <string>
4
5  namespace ci {
6
7      struct Movie
8      {
9          std::string Name;
10         int Duration;
11     };
12
13     std::vector<Movie> readMovies();
14 }
15
```

❗Missing include file

Managing headers

- A header should include all the headers it depends on.
- A header should NOT include headers it does not depend on.
- Use pimpl or an interface to represent dependencies to avoid needing to include implementation headers.
- Separate internal headers from API headers.
- See the implementation at git tag: v2-header-ordering

Use modules!

MovieReader.cpp

```cpp
std::vector<Movie> readMovies()
{
    std::vector<Movie> movies;
    std::string path = ".\\movies";

    for (const auto& entry : std::filesystem::directory_iterator(path)) {
        // read the movie file and determine its duration.
        std::ifstream file(entry.path());

        // open the file
        std::string name = entry.path().string();
        if (!file.is_open())
            throw std::runtime_error("error while opening file");

        // read a line
        std::string line;
        if (getline(file, line)) {

            // parse the line and read the duration
            std::stringstream linestream(line);
            int seconds{};
            linestream >> seconds;
            if (linestream.fail()) {
                throw std::runtime_error("error reading file");
            }
            // if successful, store the Movie in the vector
            movies.emplace_back(entry.path().string(), seconds);
        }
        else {
            // if reading the file failed, throw
            if (file.bad()) {
                throw std::runtime_error("error while reading file");
            }
            // if reading the line failed, throw
            throw std::runtime_error("file has the wrong format");
        }
    }
    return movies;
}
```

❗ readMovies does many things:

1. Loops over movie files
2. Opens a file
3. Reads a line
4. Parses the line

❗ Comments are needed to explain what the code does

❗ An exception in the loop ends the loop.

❗ High cyclomatic complexity

Improving error handling

- Advantages of exceptions
  - Allow programmers to focus on the business logic.
  - Exceptions require immediate and exclusive attention.
- Disadvantages
  - The exceptional path is slow (not necessarily a problem)
  - Local handling litters the code with try/catch clauses.
  - Not always supported in free-standing environments.
- Proposed guideline:
  - Use exceptions for the exceptional
  - Use expected for the expected.

std::expected is expected in C++ 23

- Proposed for the standard in 2017, expected in C++23
- See CppCon 2018: Andrei Alexandrescu “Expect the expected”
- `std::expected<T,E>` is a class template that contains either:
  - A value of type `T` - the expected value type; or
  - A value of type `E` - an error type used when an unexpected outcome occurs
- We will use an implementation by Sy Brand at: https://github.com/TartanLlama/expected

MovieReader.cpp

```cpp
    // read all movie files from the folder named path without explicitly throwing.
    expected <std::vector<expected<Movie>>> readMovies(const std::string& path)
    {
        std::vector<expected<Movie>> movies;

        // use the overload that excepts an error_code. Note: may still throw std::bad_alloc.
        std::error_code ec;
        for (const auto& entry : std::filesystem::directory_iterator(path, ec)) {
            movies.emplace_back(readMovie(entry.path().string()));
        }

        // if an error occurred return unexpected
        if (ec){
            return unexpected{ ec.message() };
        }
        return movies;
    }
```

git tag: v3-with-tl-expected

MovieReader.cpp

```cpp
34      // read a movie file and determine its duration without explicitly throwing
35      expected<Movie> readMovie(const std::string& fileName) {
36
37          // Using the functional syntax provided by tl::expected
38
39          return openFile(fileName)
40              .and_then([&](std::ifstream file)
41                  {
42                      return readLine(file);
43                  })
44              .and_then([&](const std::string& line)
45                  {
46                      return parseMovie(line, fileName);
47                  });
48      }
```

MovieReader.cpp

```cpp
// opens the file without explicit throw
auto openFile(const std::string& fileName) -> expected<std::ifstream>
{
    std::ifstream file(fileName);
    if (!file.is_open()) {
        return unexpected{ "error while opening file" };
    }
    return file;
}
```

```cpp
// read the first line in the file without explicit throw
auto readLine(std::ifstream& file) -> expected<std::string>
{
    std::string line{};
    if (!getline(file, line)) {
        const auto* error = file.bad()
            ? "error while reading file"
            : "file has the wrong format";
        return unexpected{ error };
    }
    return line;
}
```

```cpp
// read the movie file and determine its duration without explicit throw
auto parseMovie(const std::string& line, const std::string& fileName) -> expected<Movie> {
    std::stringstream linestream(line);
    int seconds{};
    linestream >> seconds;
    if (linestream.fail()) {
        return unexpected{ "error reading file" };
    }
    return Movie{ fileName, seconds };
}
```

✔ Each function does only one thing!

Use lazy iteration with a generator

MovieReader.cpp

```
// read all movie files from the folder named path without explicitly throwing.
// Note: the directory_iterator may throw on the first iteration of the generator.
auto MovieReader::readMovies(
    const std::string& path) -> generator<expected<Movie>> {

    for (const auto& entry : std::filesystem::directory_iterator(path)) {
        co_yield readMovie(entry.path().string());
    }
}
```

git tag: v4-using-generator

main.cpp

```cpp
#include "MovieReader.h"

#include <gtest/gtest.h>

TEST(MovieReader, ReadSucceeds)
{
    const std::string path = ".\\movies";

    ASSERT_TRUE(std::ranges::all_of(
        ci::MovieReader::readMovies(path),
        [](const auto& m)
        {
            return m.has_value();
        }
    ));
}

```

The test can use ranges algorithms on the generator

Use generators for lazy iteration

- Lazy evaluation avoids storing all elements in memory.
- Generators can be used to invert dependencies
- Use a library rather then reinvent
- The famous cppcoro library by Lewis Baker is not maintained.
- The following fork has many bug fixes and supports C++20

https://github.com/andreasbuhr/cppcoro

More improvements for error handling

- Using the generator, we actually overlooked the exception that might be thrown by the `directory_iterator`.
- We have no choice but to catch the exception.
- Here is a fail-fast method to catch and return an expected.

expected.h

✓ Extracted expected.h

```
template<typename F>
auto expect(F&& func) noexcept -> expected<std::invoke_result_t<F>>
{
    try
    {
        return std::invoke(std::forward<F>(func));
    }
    catch (std::exception& ex)
    {
        return unexpected(ex.what());
    }
}
```

Avoid Primitive Obsession

- Primitive Obsession is a code smell in which primitive data types are used excessively to represent data models.
- Movie uses an int to represent seconds.
- Movie itself does not validate its invariants (the duration must be within a certain range)
- We can fix this by hiding the constructor and use expect() on a static factory method.

MovieReader.h

```
11      class Movie
12      {
13      public:
14          const std::chrono::seconds MaxDuration = 120s;
15
16          std::string Name;
17          std::chrono::seconds Duration;
18
19          static expected<Movie> create(
20              const std::string& name, int seconds) noexcept {
21              return expect([&] {
22                  return Movie(name, std::chrono::seconds(seconds));
23              });
24          }
25
26      private:
27
28          Movie(std::string name, std::chrono::seconds seconds) :
29              Name(std::move(name)),
30              Duration(seconds) {
31
32              if (Duration > MaxDuration || Duration < 1s) {
33                  throw std::runtime_error("invalid duration");
34              }
35          }
36      };
```

✔ A static factory method uses expect to handle validation exceptions

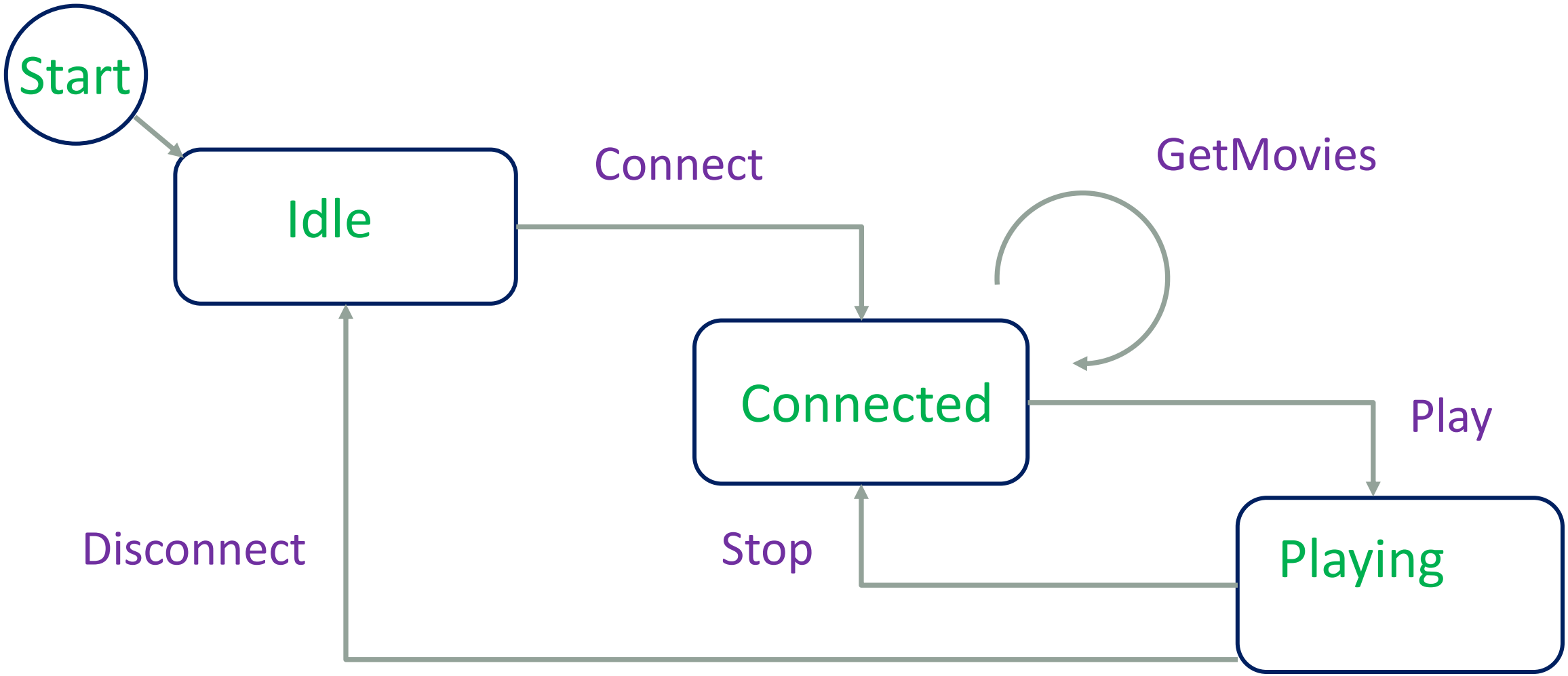
✔ The constructor is private. It validates or throws.

git tag:
v5-avoid-primitive-obsession

Building the VODServer and VODClient

- The VODServer will use the MovieReader to stream movies to the clients.
- It stores a **weak_ptr** to the client so that it can provide async notifications regarding the streaming of a movie.
- The client must call Connect, Disconnect, Play and GetMovies in the right order to get a positive response.

VOD Server State Machine



VODServer.h

```
#include "Movie.h"

#include <format>
#include <iostream>
#include <memory>
#include <vector>
#include <cppcoro/generator.hpp>

namespace ci {
    class IMovieObserver { ... };

    enum class LogLevel { ... };

    class ConsoleLogger { ... };

    class VODServer : public ConsoleLogger
    {
    public:

        VODServer();
        ~VODServer();

        VODServer(VODServer&& other) noexcept = default;
        VODServer(const VODServer& other) noexcept = delete;

        VODServer& operator=(VODServer&&) = default;
        VODServer& operator=(const VODServer&) = delete;

        bool Connect(std::weak_ptr<IMovieObserver>);
        void Disconnect();

        [[nodiscard]] expected<std::vector<Movie>> GetMovies() const;

        bool Play(const Movie&);
        bool Stop();
```

❗ Declares too much in one file

❗ Implementation inheritance is not recommended.

git tag: v5-VODServer-VODClient

VODServer.cpp

```cpp
bool VODServer::Play(const Movie& movie)
{
    if (_isPlaying) {
        return false;
    }

    _playThread = std::jthread([this, movie](const std::stop_token& token)
        {
            if (const auto c = _client.lock())
            {
                c->OnPlaying(movie);
            }

            std::mutex mutex;
            std::unique_lock lock(mutex);

            std::condition_variable_any().wait_for(
                lock,
                token,
                movie.Duration,
                [&token] { return token.stop_requested(); });

            if (const auto c = _client.lock())
            {
                c->OnCompleted(movie);
            }
        });

    _isPlaying = true;
    return true;
}
```

❗ Play does many things:
1. Manages state
2. Starts a thread

❗ The lambda does many things:
1. Manages notifications
2. Does the streaming

Refactoring Step

- Minimize the API (ISP)
  - Define an interface for IVODServer
  - Split interface definitions into separate files.
  - Define a factory method that returns an IVODServer
- Use extract method so that methods do one thing (SRP).
- Separation of test and production code.
  - The clean-it project is now a DLL
  - The DLL exposes one method only – the factory method

IMovieObserver.h

```cpp
#pragma once

#include "Movie.h"

namespace ci {
    class IMovieObserver
    {
    public:
        virtual ~IMovieObserver() = default;

        virtual void OnPlaying(Movie) = 0;
        virtual void OnCompleted(Movie) = 0;
    };
}

```

IVODServer.h

```cpp
#pragma once

#include "IMovieObserver.h"

#include <memory>
#include <vector>

namespace ci {
    class IVODServer
    {
    public:
        virtual ~IVODServer() = default;

        virtual bool Connect(std::weak_ptr<IMovieObserver>) = 0;
        virtual void Disconnect() = 0;

        [[nodiscard]] virtual expected<std::vector<Movie>> GetMovies() const = 0;

        virtual bool Play(const Movie&) = 0;
        virtual bool Stop() = 0;
    };
}
```

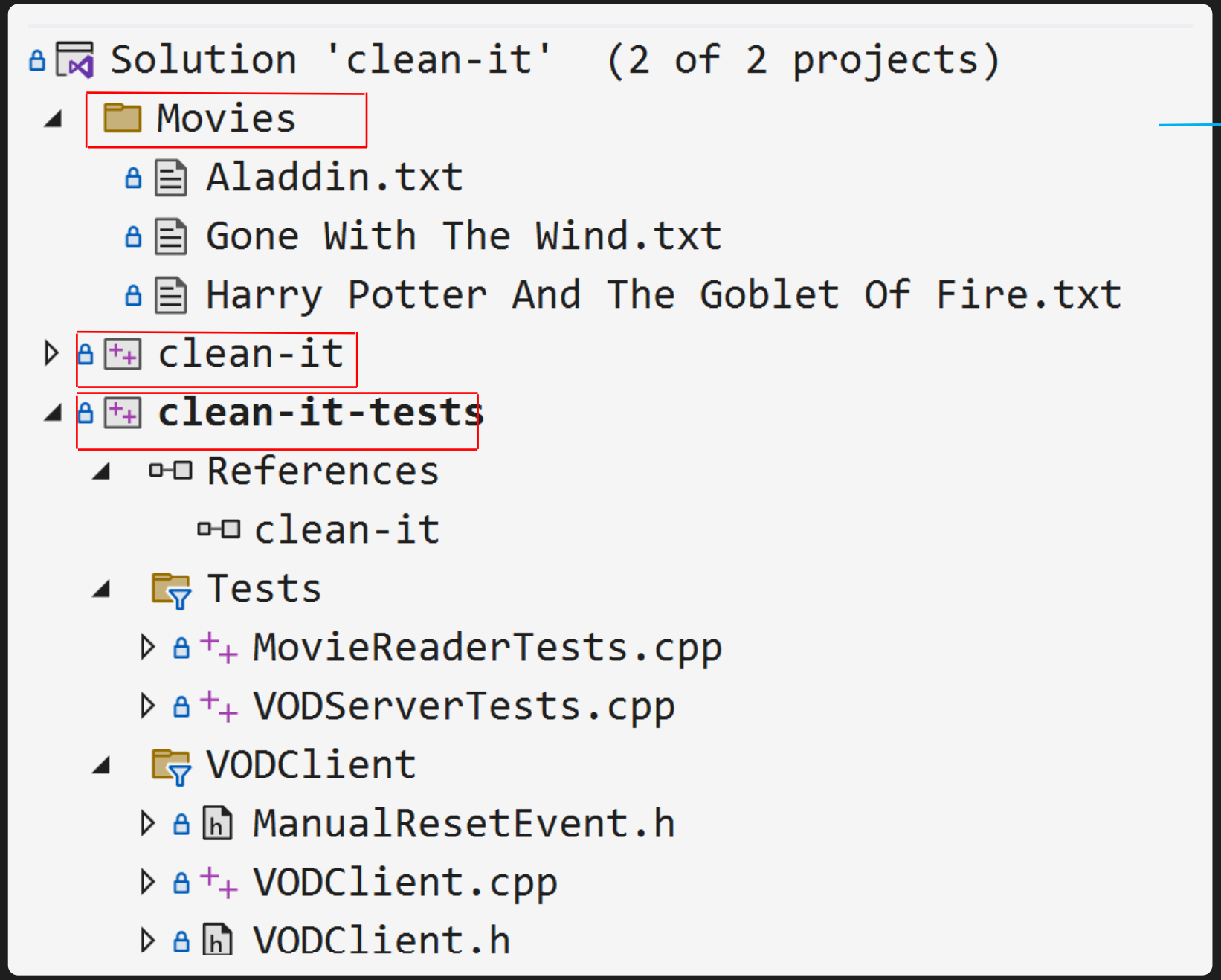
VODExport.h

```cpp
#pragma once

#include "IVODServer.h"

#include <memory>

#define VODEXPORT __declspec(dllexport)

namespace ci {
    std::shared_ptr<IVODServer> VODEXPORT CreateVODServer();
}
```

✔ Define an interface

✔ Expose a factory method to create the interface

git tag: v6-use-dependency-injection



✔ Separated tests from implementation

git tag:
v5-split-test-production

VODServerTests.cpp

```cpp
#include "VODExport.h"
#include "VODClient.h"

#include <gtest/gtest.h>

using namespace ci;

TEST(TestServer, GetMovies) {
    const auto server = CreateVODServer();
    const auto client = std::make_shared<VODClient>(server);

    client->TestGetMovies();
}

TEST(TestServer, Login) {
    const auto server = CreateVODServer();
    const auto client = std::make_shared<VODClient>(server);

    client->TestLogin();
}

TEST(TestServer, StartStop) {
    const auto server = CreateVODServer();
    const auto client = std::make_shared<VODClient>(server);

    client->TestStartStop();
}

TEST(TestServer, CompleteDuration) {
    const auto server = CreateVODServer();
    const auto client = std::make_shared<VODClient>(server);

    client->TestCompleteDuration();
}
```

✔ Tests use the factory to create a IVODServer

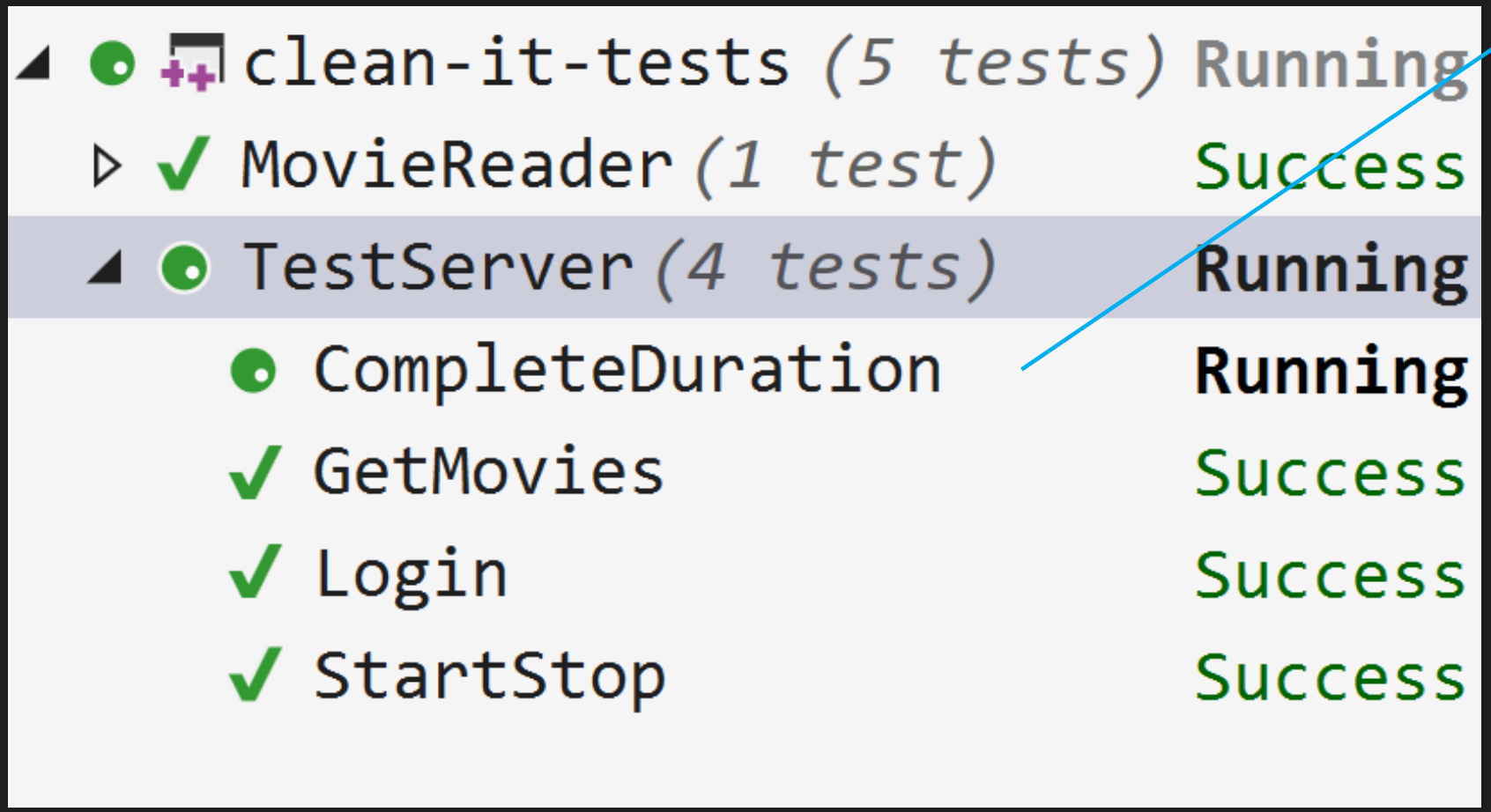
VODClient.cpp

✔ The VODClient is part of the test project

```cpp
void VODClient::TestCompleteDuration() {
    const bool loggedIn = _server->Connect(weak_from_this());
    ASSERT_TRUE(loggedIn);

    const auto movies = _server->GetMovies();
    ASSERT_TRUE(movies && !movies->empty());

    const auto first = *movies->begin();

    bool isPlaying = _server->Play(first);
    ASSERT_TRUE(isPlaying);

    isPlaying = _server->Play(first);
    ASSERT_FALSE(isPlaying);

    const auto receivedPlaying = _wait_for_started.wait_for(1s);
    ASSERT_TRUE(receivedPlaying);

    const auto receivedStopped = _wait_for_stopped.wait_for(20s);
    ASSERT_TRUE(receivedStopped);
}
```

❗ But we have a problem: Tests are slow.

❗ The streaming test takes 20 seconds!



Dependency Injection

- The objective is to decrease coupling between components and their implementations.
- Components should depend on abstractions not concrete types.
- C++ provides two abstraction models:
  - Compile time using templates.
  - Run time using virtual functions.
- We will use virtual functions in this project.
- See git tag: v6-dependency injection for the implementation

Constructor injection

- A component declares the implementations it requires as interfaces which are arguments to its constructor.
- The component stores a pointer or reference to the interface.
- The lifetime of the interface must cover the lifetime of the component.
- The benefits:
  - The caller can specify which implementation will be used.
  - Dependencies are explicit and easy to find.
- The challenge:
  - Composing objects is complex and … breaks encapsulation.
  - When constructors change, the wiring up code needs to change.

Inversion of Control Container

- IOC containers resolve the challenges of DI.
- An IOC container is a factory with two aspects:
  - Register methods:
    Specify which objects should be instantiated for which interface.
  - Resolve methods:
    Build objects specified by interface based on the specifications.
- How does this solve the complexities of composition?
  - The number of specifications is proportional to the number of abstractions – not to the number of types that need to be resolved.

Hypodermic

- https://github.com/ybainier/Hypodermic
- Hypodermic is a non-intrusive header-only IoC container for C++.
- It provides dependency injection to existing designs by managing the creation of components and their dependencies in the right order
- This spares you the trouble of writing and maintaining boiler plate code.

Sample Code for Register and Resolve

```
15      // Register where configuration is being done
16
17      ContainerBuilder builder;
18
19      builder.registerType<VODServer>().as<IVODServer>();
20
21      const auto container = builder.build();
22
23      // Then use somewhere else
24
25      auto   vodServer  = container->resolve<IVODServer>();
```

git tag: v7-use-ioc-container

Usage in clean-it

- VODServer accepts an IStreamer in its constructor.
- The test defines an IStreamer mock.
- The test code uses the IOC container to register the mock as the implementation of IStreamer.
- The test requests the container to resolve an IVODServer.
- This container
  - Resolves the IVODServer type to the VODServer type.
  - Creates an instance of Streamer and passes it to the constructor of VODServer.
  - Returns the instance as an IVODServer.
- See git tag: v7-use-ioc-container

So, what about inheritance?

- Cpp Core Guidelines (C.129): “When designing a class hierarchy, distinguish between implementation inheritance and interface inheritance”.
- Implementation inheritance can be replaced with private composition. This is better because it hides the implementation details.
- Interface inheritance can be achieved using:
  - Run time polymorphism: Pure virtual methods.
  - Static polymorphism: CRTP

Implementation Inheritance introduces tight coupling

1. The Derived class reuses the Base class code but the Derived class cannot be reused independently of the Base.
2. The Derived class cannot be tested with different implementations.

```
12      class VODServer : public IVODServer, public ConsoleLogger
13      {
14      public:
15
16          VODServer(std::shared_ptr<IStreamer> streamer);
17          ~VODServer() override;
```

git tag: v8-prefer-composition-over-inheritance

Logger.h

```
16      class Logger {
17      public:
18          Logger(std::shared_ptr<ILogWriter> writer, LogLevel level = LogLevel::Error) :
19              _writer(move(writer)),
20              _level(level)
21          {}
22
23          template <typename... Args>
24          void Write(LogLevel level, std::string_view format_str, Args&&... args) {
25              if (level >= _level) {
26                  _writer->Write(std::vformat(
27                      format_str,
28                      std::make_format_args(std::forward<Args>(args)...)));
29              }
30          }
31
32      private:
33
34          std::shared_ptr<ILogWriter> _writer;
35          LogLevel _level;
36      };
```

✔ The implementation ILogWriter is injected

git tag: v7-use-ioc-container

Using Modules

- The final two tags in the repo make two improvements in the code:
- v9-use-header-unit-for-std
  - In this commit, a static library is added which contains all standard headers as a header unit, which are then imported as a module in the clean-it dynamic link library.
- v10-VOD-as-a-module
  - In this commit, the VODServer is exported as a module and imported by the clean-it-tests as a module.

Lessons Learnt

- Use SOLID principles to guide your design.
- Use the Core CPP Guidelines and tools that help you implement them.
- Organize headers to reduce coupling.
- Use modules for new projects.
- Consider using std::expected for expected errors.
- Use generators for lazy iteration and for inversion of dependencies.
- Avoid Primitive Obsession.
- Use interfaces on component boundaries.
- Inject implementation dependencies.
- Prefer composition over implementation inheritance.
- Consider the use of Inversion of Control Containers to construct objects.

Lessons Learnt

1. Use SOLID principles to guide your design.
2. Use the Core CPP Guidelines and tools that help you implement them.
3. Organize headers to reduce coupling.
4. Use modules for new projects.
5. Consider using std::expected for expected errors.
6. Use generators for lazy iteration and for inversion of dependencies.
7. Avoid Primitive Obsession.
8. Use interfaces on component boundaries.
9. Inject implementation dependencies.
10. Prefer composition over implementation inheritance.
11. Consider the use of Inversion of Control Containers to construct objects.

Discussion