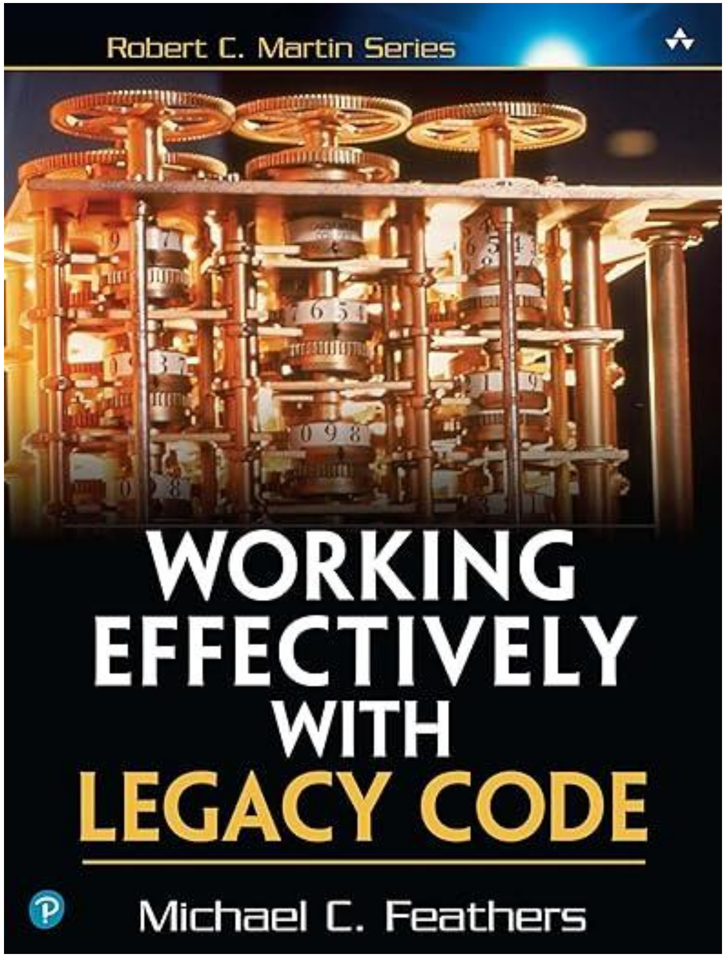
How to Improve Legacy Code

David Sackstein

Agenda

- What is legacy code?
- Problems with legacy code
- Evolution of legacy code
- Make minimal changes
- Testing code without a clear API
- Addressing hard dependencies

Recommended Book on this Topic



What is Legacy Code?

- Legacy code often starts out as high quality code.
- With time it degrades due to:
  - Not enough tests
  - “Quick and dirty” bug fixes
  - Adding new features without redesigning as needed

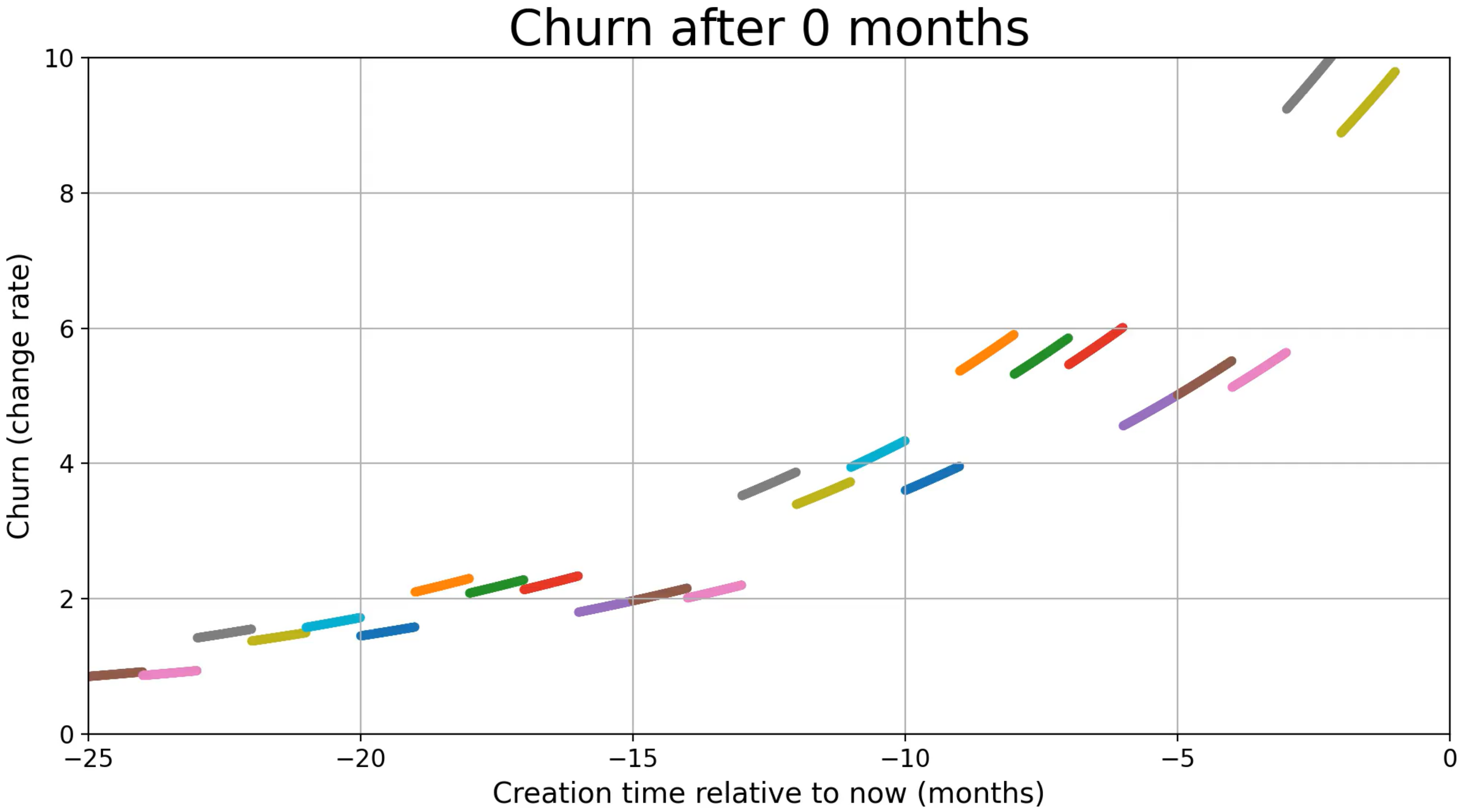
Problems with Legacy Code

- Legacy code is often over complex due to ~~quick and~~ dirty changes made over time resulting in:
  - Highly coupled components
  - Violations of the Single Responsibility Principle
  - Violations of encapsulation
  - Inheritance all over the place
- Our manager will not let us change legacy code because “If it works, don’t fix it”
- Legacy code typically does not have tests so we can’t safely change it, anyway.

Working with Legacy Code

- Betty bought a bit of butter but the butter was bitter, so she bought a better bit of butter, put it with the bitter butter, and made the bitter butter better.
- The legacy code is bitter, so write new code, with tests that adheres to the principles we are learning, and your code base will incrementally improve.
- This is because the churn (rate of change) of code decreases with time so after a year, most of the code you will need to change will be new code.
- The challenge is then to make sure that the changes you make during the year preserve its quality.

Evolution of Legacy Code



Recommendations

- There is often no need to rewrite legacy code
- Because, as it gets older:
  - Maintenance decreases (bugs get solved)
  - The probability of breaking it increases.
- Make small incremental changes.
- Add a test, do some refactoring
- Add more tests

Sprout Method or Sprout Class

- Create new functionality in a new method or class.
- Do not copy the style and fallacies of the legacy code. Don’t mimic just for consistency.
- Modify just a few lines from the legacy code so it can call the new code.
- Demonstrate in a review that
  - The changes to the existing code were minimal
  - The new code is well tested.

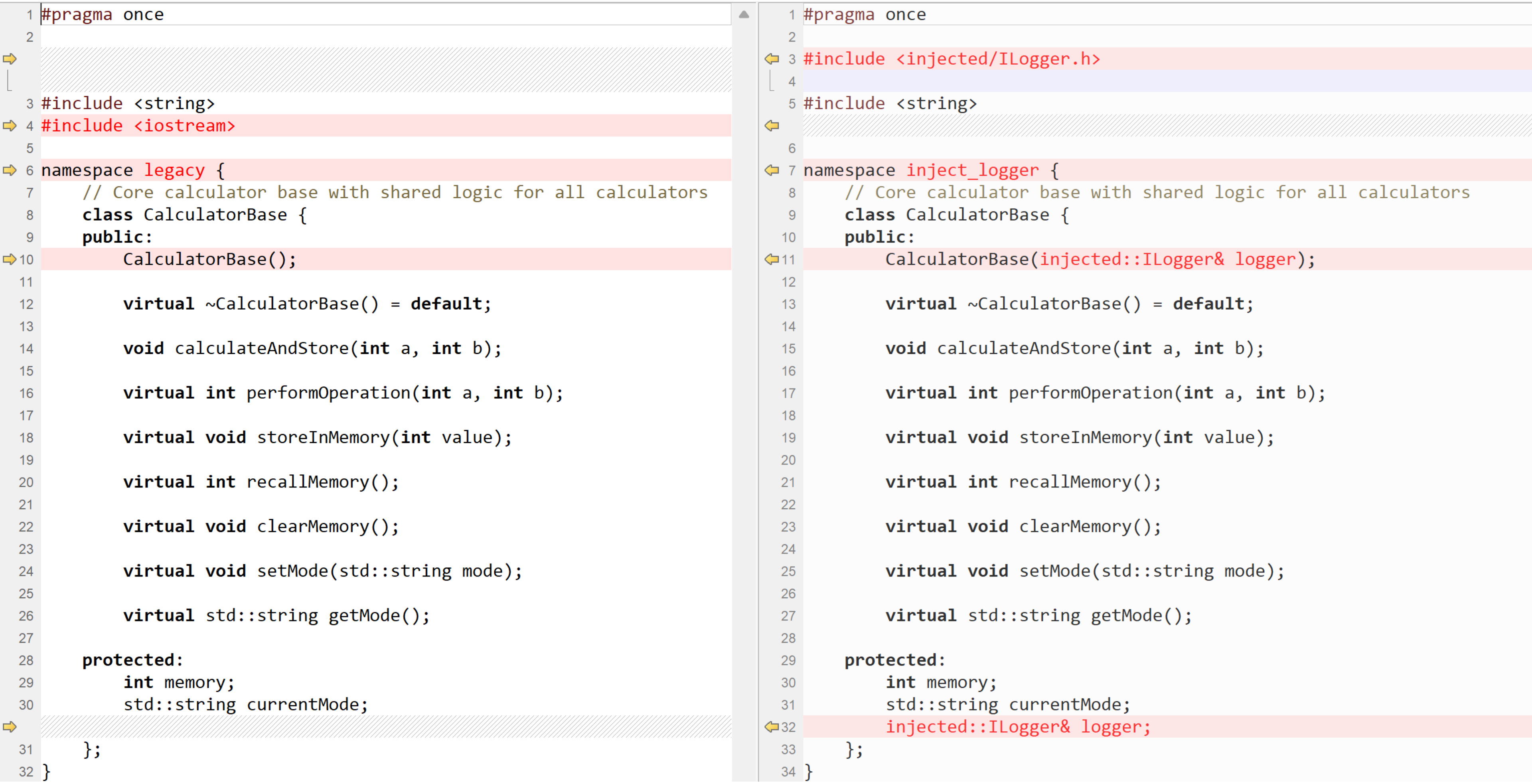
Difficulties writing tests

- The public APIs are not well-defined so we cannot run a test that calls them.
  - For instance, they are called in varying order from different parts of the system
- There are hard dependencies that are not conducive to testing, for instance:
  - External systems that are not available during tests (e.g. database and network devices)
  - Systems that are not deterministic (making our tests not repeatable)

Addressing unclear APIs

- Write Characterization Tests
  - A test that runs the code and captures its current behavior.
  - The behavior should be a part of the system that would certainly change if the system functionality were changed.
- For instance:
  - Define an interface for a logger.
  - Make a minimal change to the code to inject the logger.
  - Inject an implementation that records all logs to a file.
  - Run the code and verify that the recorded logs did not change.
  - This can be done in a non-test environment at first

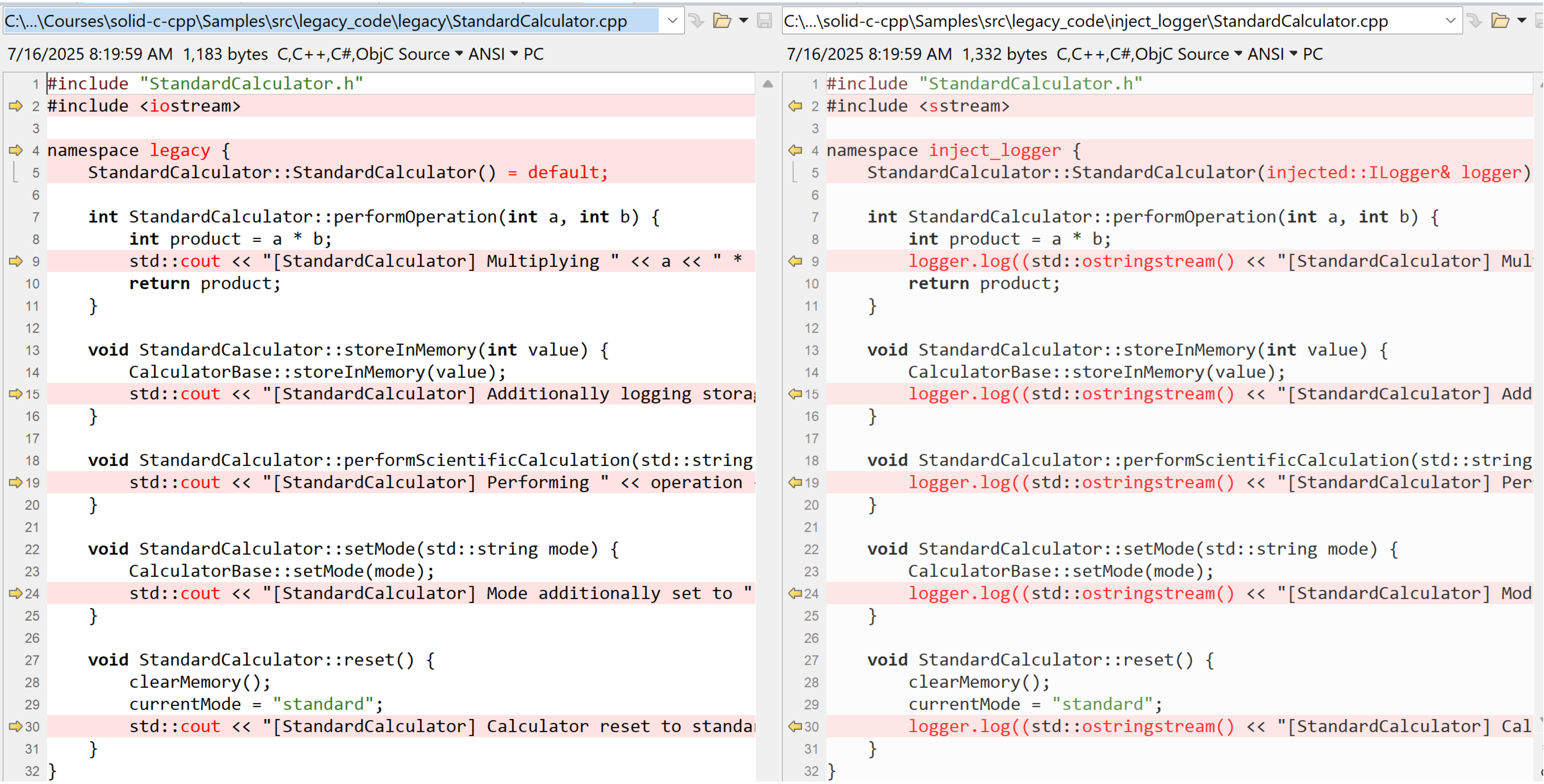
Inject an ILogger



```cpp
#pragma once

#include <string>
#include <iostream>

namespace legacy {
    // Core calculator base with shared logic for all calculators
    class CalculatorBase {
    public:
        CalculatorBase();

        virtual ~CalculatorBase() = default;

        void calculateAndStore(int a, int b);

        virtual int performOperation(int a, int b);

        virtual void storeInMemory(int value);

        virtual int recallMemory();

        virtual void clearMemory();

        virtual void setMode(std::string mode);

        virtual std::string getMode();

    protected:
        int memory;
        std::string currentMode;
    };
}
```

```cpp
#pragma once

#include <injected/ILogger.h>

#include <string>

namespace inject_logger {
    // Core calculator base with shared logic for all calculators
    class CalculatorBase {
    public:
        CalculatorBase(injected::ILogger& logger);

        virtual ~CalculatorBase() = default;

        void calculateAndStore(int a, int b);

        virtual int performOperation(int a, int b);

        virtual void storeInMemory(int value);

        virtual int recallMemory();

        virtual void clearMemory();

        virtual void setMode(std::string mode);

        virtual std::string getMode();

    protected:
        int memory;
        std::string currentMode;
        injected::ILogger& logger;
    };
}
```

Intercept logs



C:\...\Courses\solid-c-cpp\Samples\src\legacy_code\legacy\StandardCalculator.cpp

7/16/2025 8:19:59 AM 1,183 bytes C,C++,C#,ObjC Source ▾ ANSI ▾ PC

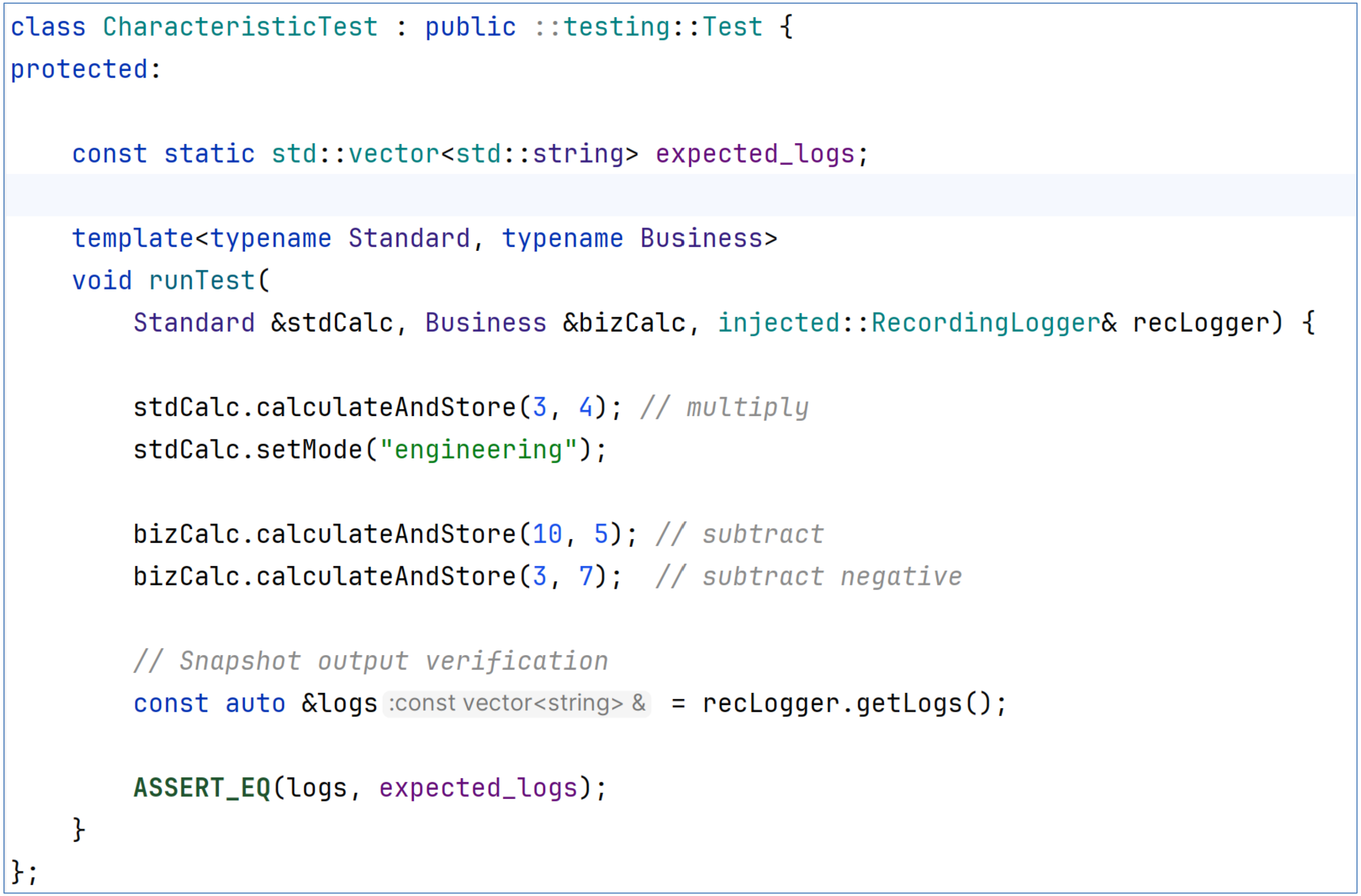
```cpp
#include "StandardCalculator.h"
#include <iostream>

namespace legacy {
    StandardCalculator::StandardCalculator() = default;

    int StandardCalculator::performOperation(int a, int b) {
        int product = a * b;
        std::cout << "[StandardCalculator] Multiplying " << a << " *
        return product;
    }

    void StandardCalculator::storeInMemory(int value) {
        CalculatorBase::storeInMemory(value);
        std::cout << "[StandardCalculator] Additionally logging stora
    }

    void StandardCalculator::performScientificCalculation(std::string
        std::cout << "[StandardCalculator] Performing " << operation
    }

    void StandardCalculator::setMode(std::string mode) {
        CalculatorBase::setMode(mode);
        std::cout << "[StandardCalculator] Mode additionally set to "
    }

    void StandardCalculator::reset() {
        clearMemory();
        currentMode = "standard";
        std::cout << "[StandardCalculator] Calculator reset to standa
    }
}
```

C:\...\solid-c-cpp\Samples\src\legacy_code\inject_logger\StandardCalculator.cpp

7/16/2025 8:19:59 AM 1,332 bytes C,C++,C#,ObjC Source ▾ ANSI ▾ PC

```cpp
#include "StandardCalculator.h"
#include <sstream>

namespace inject_logger {
    StandardCalculator::StandardCalculator(injected::ILogger& logger)

    int StandardCalculator::performOperation(int a, int b) {
        int product = a * b;
        logger.log((std::ostringstream() << "[StandardCalculator] Mul
        return product;
    }

    void StandardCalculator::storeInMemory(int value) {
        CalculatorBase::storeInMemory(value);
        logger.log((std::ostringstream() << "[StandardCalculator] Add
    }

    void StandardCalculator::performScientificCalculation(std::string
        logger.log((std::ostringstream() << "[StandardCalculator] Per
    }

    void StandardCalculator::setMode(std::string mode) {
        CalculatorBase::setMode(mode);
        logger.log((std::ostringstream() << "[StandardCalculator] Mod
    }

    void StandardCalculator::reset() {
        clearMemory();
        currentMode = "standard";
        logger.log((std::ostringstream() << "[StandardCalculator] Cal
    }
}
```

Record and compare logs



```
class CharacteristicTest : public ::testing::Test {
protected:

    const static std::vector<std::string> expected_logs;

    template<typename Standard, typename Business>
    void runTest(
        Standard &stdCalc, Business &bizCalc, injected::RecordingLogger& recLogger) {

        stdCalc.calculateAndStore(3, 4); // multiply
        stdCalc.setMode("engineering");

        bizCalc.calculateAndStore(10, 5); // subtract
        bizCalc.calculateAndStore(3, 7);  // subtract negative

        // Snapshot output verification
        const auto &logs :const vector<string> & = recLogger.getLogs();

        ASSERT_EQ(logs, expected_logs);
    }
};
```

Addressing hard dependencies

- Mocks and stubs will help here:
  - Inject interfaces to constructors
  - Replace the hard dependencies with simple mocks that allow the tests to run.

Key Takeaways

Legacy Code	Legacy code could be today’s code - tomorrow
No rewrites	It is better to improve incrementally
Add tests	Use characteristic tests to capture behavior
	Refactor, and add more tests
Dependencies	Use mocks and stubs to allow testing