
Core C++ 2025
19 Oct. 2025 :: Tel-Aviv
Abstraction Addiction:
When good C++ design goes bad
Adi Ben David

Who am I?

- Senior software engineer at Cynet Security
- BSc in Computer Science from the Technion
- About a decade of experience, mostly in C++
- Mother of 2 adorable kids

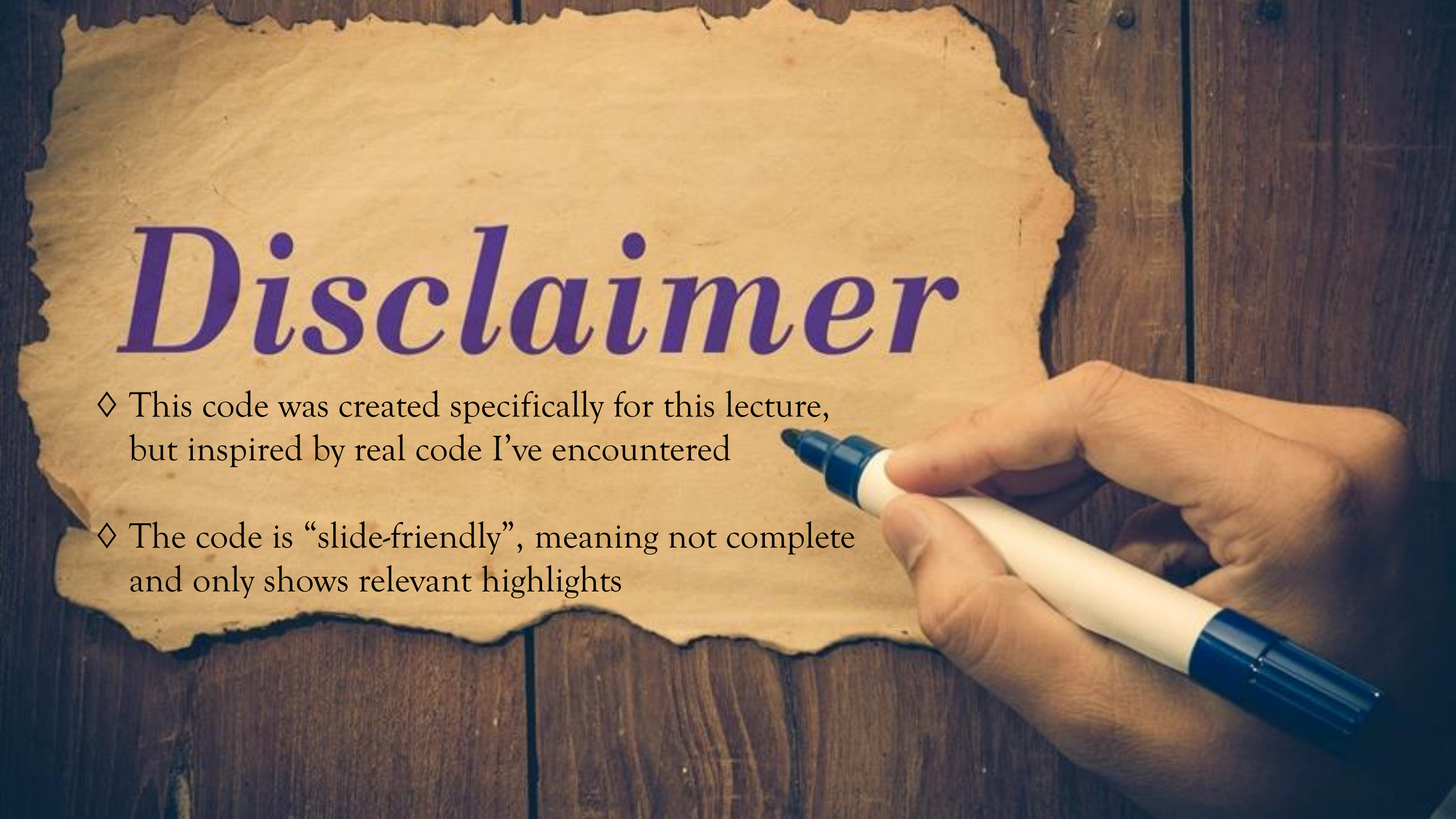
TECHNION
Israel Institute of Technology

cynet

All opinions expressed in this lecture are my own and don't represent the company or anyone else

Why are we here?

- I just came back from paternity leave, but the inspiration for this talk came right before it.
- Reviewing code that sparked some strong emotions …
- We're going to be talking about over-engineering
- … and you may be similarly triggered


Disclaimer
◊ This code was created specifically for this lecture, but inspired by real code I've encountered
◊ The code is "slide-friendly", meaning not complete and only shows relevant highlights

How and why code

How and why code

Solve a problem

"Pretty"

Future proof

Efficient

Maintainable

Readable

Abstract

...

How and why code

How abstract can we go?

How abstract can we go?

How abstract can we go?

```
6   class Data;
7
8   class Event {
9   public:
10      Event(Data& data);
11      /*
12      Some interface to query and adjust the event data
13      */
14  private:
15      void Log() const;
16      std::shared_ptr<Data> m_data;
17  };
```

```
19  enum class ErrorValue {
20      Success = 0,
21      Fail,
22  };
23  struct ErrorCode {
24      ErrorValue value;
25      std::string info;
26      /* Store information about success\fail of methods */
27  };
```

```
29  class EventProcessor {
30  public:
31      EventProcessor();
32      ErrorCode AcceptEvent(std::shared_ptr<Event> event);
33  private:
34      void ReportEvents() const;
35      ErrorCode ProcessEvents();
36      ErrorCode ProcessEvent(std::shared_ptr<Event> event);
37
38      std::mutex m_lock;
39      static std::deque<std::shared_ptr<Event>> m_waitlist;
40      static std::deque<std::shared_ptr<Event>> m_toReport;
41  };
```

How abstract can we go?

```
6   class Data;
7
8   class Event {
9   public:
10      Event(Data& data);
11      /*
12      Some interface to query and adjust the
13      */
14  private:
15      void Log() const;
16      std::shared_ptr<
17  };
```

This is too easy,
Let's bring this up a notch

```
19      enum class ErrorValue {
20          Success = 0,
21          Fail,
22      };
23      struct ErrorCode {
24          ErrorValue value;
25          std::string info;
26          /* Store information about success\fail of methods */
27      };
```

```
                                    {
                                ocessor();
                            rorCode AcceptEvent(std::shared_ptr<Event> event);
34      private:
35          void ReportEvents() const;
36          ErrorCode ProcessEvents();
37          ErrorCode ProcessEvent(std::shared_ptr<Event> event);
38
39          std::mutex m_lock;
40          static std::deque<std::shared_ptr<Event>> m_waitlist;
41          static std::deque<std::shared_ptr<Event>> m_toReport;
    };
```

How abstract can we go?

```
6   class Data;
7
8   class Event {
9   public:
10      Event(Data& data);
11      /*
12      Some interface to query and adjust the event data
13      */
14  private:
15      void Log() const;
16      std::shared_ptr<Data> m_data;
17  };
```

How abstract can we go?

```
61  class IEvent {
62  public:
63      virtual ~IEvent() = default;
64      virtual void Log() const = 0;
65  };
66
67  class EventFoo: public IEvent {
68  public:
69      ~EventFoo() override = default;
70      void Log() const override;
71  };
72
73  class EventBoo: public IEvent {
74  public:
75      ~EventBoo() override = default;
76      void Log() const override;
77  };
78
79  class EventMoo: public IEvent {
80  public:
81      ~EventMoo() override = default;
82      void Log() const override;
83  };
```

How abstract can we go?

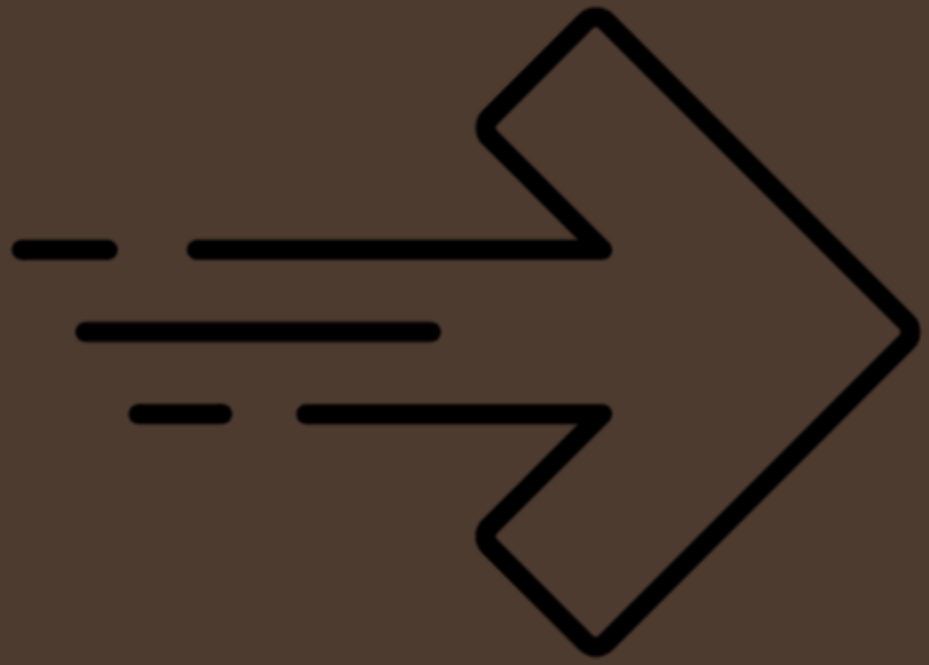
```
61  class IEvent {
62  public:
63      virtual ~IEvent() = default;
64      virtual void Log() const = 0;
65  };
```

```
108 class EventLogger {
109 public:
110     EventLogger(std::shared_ptr<IEvent> event):
111                 m_event(event) {};
112     std::string Log();
113     std::string Report();
114     std::string to_string();
115 private:
116     std::shared_ptr<IEvent> m_event;
117 };
```

We could continue
But I think we get the point

Before we abstract, weigh the act

- Do we have a good reason for abstraction or is this simply for future proofing made up scenarios?
- Is there enough substance to justify a separate class?
- Does this actually make the code better or more readable?



Operators galore

```
7   class MyInFile {
8   public:
9       MyInFile(const std::string& filename, std::ios_base::openmode mode = std::ios::in):
10          m_stream(filename, mode) {};
11      MyInFile& operator++() {
12
13          m_stream.getline(&m_line[0], s_maxString);
14          return *this;
15      }
16      std::string operator*() {
17
18          return &m_line[0];
19      }
20  private:
21      std::fstream m_stream;
22      static constexpr unsigned s_maxString = 200;
23      std::array<char, s_maxString> m_line;
24  };
```

Operators galore

```
26  class MyOutFile {
27  public:
28      MyOutFile(const std::string& filename, std::ios_base::openmode mode
29          m_filename(filename), m_stream(filename, mode) {};
30      MyOutFile& operator++() {
31          // odd - truncate to increase file size by some amount
32          std::filesystem::resize_file(m_filename, std::filesystem::file_size(m_filename) + 64*1024);
33          return *this;
34      }
35      std::string operator*() {
36
37          return &m_line[0];
38      }
39      MyOutFile& operator+(const std::string& str) {
40
41          m_stream << str;
42          return *this;
43      }
44  protected:
45      std::string m_filename;
46      std::fstream m_stream;
47      static constexpr unsigned s_maxString = 200;
48      std::array<char, s_maxString> m_line;
49  };
```

Because "*MyInFile*" had *operator++*

Like in "*MyInFile*"

Operators galore

Why don't I like this?

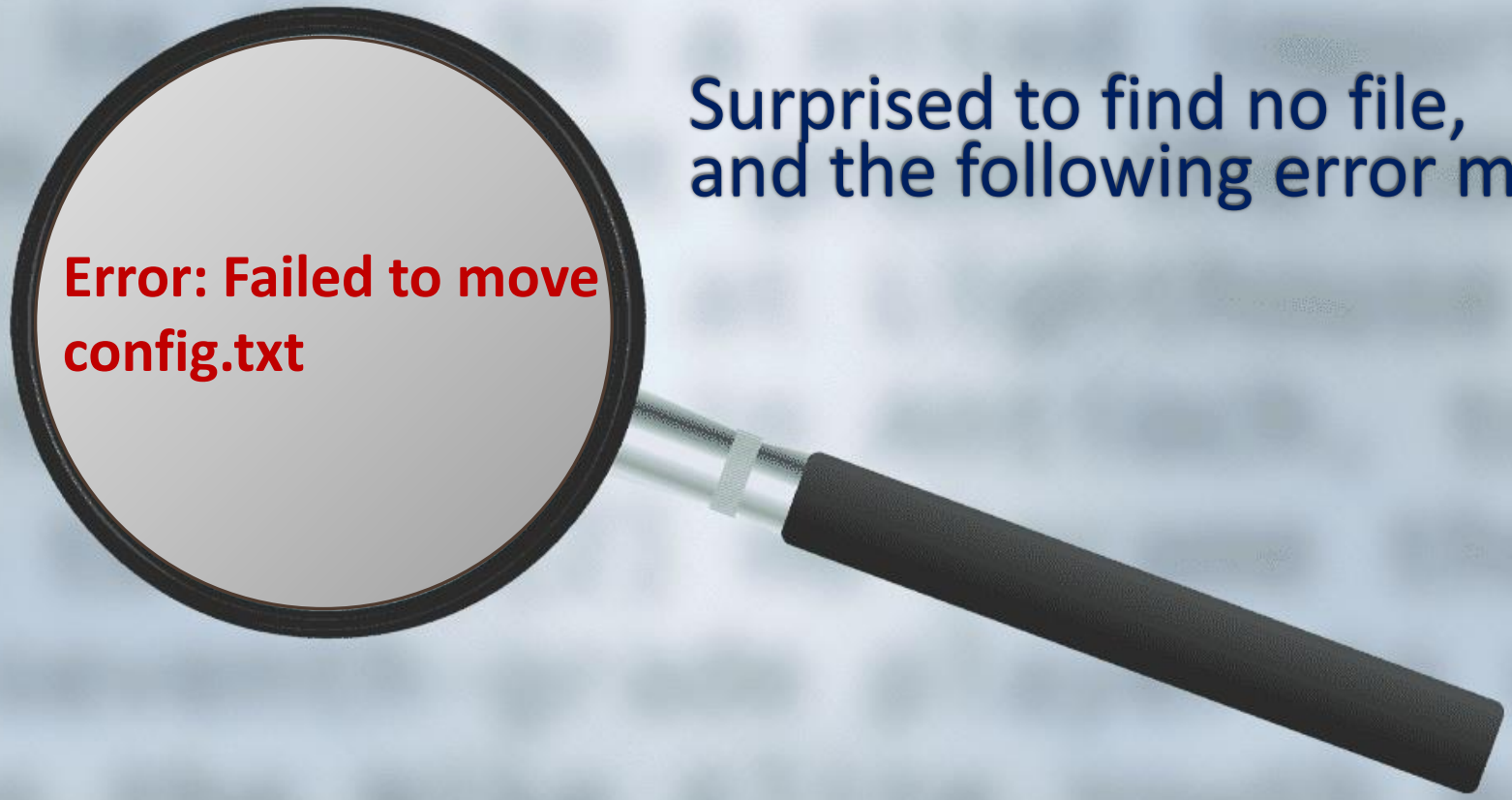
```
26  class MyOutFile {
27  public:
28      MyOutFile(const std::string& filename, std::ios_base::openmode mode = std::ios::app):
29          m_filename(filename), m_stream(filename, mode) {};
30      MyOutFile& operator++() {
31
32          std::filesystem::resize_file(m_filename, std::filesystem::file_size(m_filename) + 64*1024);
33          return *this;
34      }
35      std::string operator*() {
36
37          return &m_line[0];
38      }
39      MyOutFile& operator<<(const std::string& str) {
40
41          m_stream << str;
42          return *this;
43      }
44  protected:
45      std::string m_filename;
46      std::fstream m_stream;
47      static constexpr unsigned s_maxString = 200;
48      std::array<char, s_maxString> m_line;
49  };
```

Beware of operators, they might bite

- Balance what clean and compact means
  - Are we better off with a descriptive (yet longer) method name?
- Is the overload intuitive or forced?
- Intuition -> assumptions
  wrong assumptions -> bug potential

Let's investigate Some code

I tried writing a configuration into a file



Surprised to find no file, and the following error message

Let’s dive in

```
131  class BaseFileOps{
132  public:
133      BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135      ErrorCode Rename(std::string new_name) {
136          return MoveFile(m_fileName, new_name);
137      }
138  private:
139  static ErrorCode MoveFile(const std::string& source, const std::string& dest) {
140          if (0 != std::remove(dest.c_str())) {
141              return {ErrorValue::Fail, "Failed to delete " + source};
142          }
143          if (0 != std::rename(source.c_str(), dest.c_str())) {
144              return {ErrorValue::Fail, "Failed to move " + source};
145          }
146          return {ErrorValue::Success, source + " file moved to " + dest};
147      }
148
149      std::string m_fileName;
150  };
```

Let’s dive in

```
131  class BaseFileOps{
132  public:
133      BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135      ErrorCode Rename(std::string new_name) {
136          return MoveFile(m_fileName, new_name);
137      }
138  private:
139  static ErrorCode MoveFile(const std::string& source, const std::string& dest) {
140          if (0 != std::remove(dest.c_str())) {
141              return {ErrorValue::Fail, "Failed to delete " + source};
142          }
143          if (0 != std::rename(source.c_str(), dest.c_str())) {
144              return {ErrorValue::Fail, "Failed to move " + source};
145          }
146          return {ErrorValue::Success, source + " file moved to " + dest};
147      }
148
149      std::string m_fileName;
150  };
```

Let’s dive in

```
131  class BaseFileOps{
132  public:
133      BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135      ErrorCode Rename(std::string new_name) {
136          return MoveFile(m_fileName, new_name);
137      }
138  private:
139  static ErrorCode MoveFile(const std::string& source, const std::string& dest) {
140          if (0 != std::remove(dest.c_str())) {
141              return {ErrorValue::Fail, "Failed to delete " + source};
142          }
143          if (0 != std::rename(source.c_str(), dest.c_str())) {
144              return {ErrorValue::Fail, "Failed to move " + source};
145          }
146          return {ErrorValue::Success, source + " file moved to " + dest};
147      }
148
149      std::string m_fileName;
150  };
```

Let’s dive in

```
131  class BaseFileOps{
132  public:
133      BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135      ErrorCode Rename(std::string new_name) {
136          return MoveFile(m_fileName, new_name);
137      }
138  private:
139      ErrorCode MoveFile(const std::string& source, const std::string& dest) {
140          if (0 != std::remove(dest.c_str())) {
141              return {ErrorValue::Fail, "Failed to move " + source};
142          }
143          if (0 != std::rename(source.c_str(), dest.c_str())) {
144              return {ErrorValue::Fail, "Failed to move " + source};
145          }
146          return {ErrorValue::Success, source + " file moved to " + dest};
147      }
148
149      std::string m_fileName;
150  };
```

Let’s dive in

```
131   class BaseFileOps{
132   public:
133       BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135       ErrorCode Rename(std::string new_name) {
136           return MoveFile(m_fileName, new_name);
137       }
138   private:
139   static ErrorCode MoveFile(const std::string& source, const std::string& dest) {
140           if (0 != std::remove(dest.c_str())) {
141               return {ErrorValue::Fail, "Failed to delete " + source};
142           }
143           if (0 != std::rename(source.c_str(), dest.c_str()))
144               return {ErrorValue::Fail, "Failed to move " + source};
145           }
146           return {ErrorValue::Success, source + " file moved to " + dest};
147
148   auto confHandle = std::make_shared<ConfigGenerator<MyConfig>>("config.txt", conf);
149       std::string m_fileName;
150   };
```

config.txt

Let’s dive in

```
186  template <typename Config>
187  class ConfigGenerator: public Conf
188  public:
189      ConfigGenerator(std::string file_name, Config& config):
190          ConfigDumper(file_name), _raw_config(file_name, config.Serialize(), config.Size())
191          {};
192
193  private:
194      RawConfig _raw_config;
195  };
```

config.txt

Let’s dive in

```
178   class ConfigDumper: public DataDumper {
179   public:
180       ConfigDumper(const std::string& dump_to): DataDumper(dump_to) {}
181       void WriteNewVersion(RawConfig& config) {
182           Dump(config.configName, config.data, config.dataSize);
183       }
184   };
```

Let’s dive in

```
178  class ConfigDumper: public DataDumper {
179  public:
180      ConfigDumper(const std::string& dump_to): DataDumper(dump_to) {}
181      void WriteNewVersion(RawConfig& config) {
182          Dump(config.configName, config.data, config.dataSize);
183      }
184  };
```

Let’s dive in

```
169  class DataDumper : public FileWriter {
170  public:
171      DataDumper(const std::string& dump_to): FileWriter(dump_to) {}
172      void Dump(const std::string& file_name, uint8_t *data, size_t data_size) {
173          Write(data, data_size);
174          Rename(file_name);
175      }
176  };
```

Let’s dive in

```
169  class DataDumper: public FileWriter {
170  public:
171      DataDumper(const std::string& dump_to): FileWriter(dump_to) {}
172      void Dump(const std::string& file_name, uint8_t *data, size_t data_size) {
173          Write(data, data_size);
174          Rename(file_name);
175      }
176  };
```

Let’s dive in

```
169  class DataDumper: public FileWriter {
170  public:
171      DataDumper(const std::string& dump_to): FileWriter(dump_to) {}
172      void Dump(const std::string& file_name, uint8_t *data, size_t data_size) {
173          Write(data, data_size);
174          Rename(file_name);
175      }
176  };
```

```
131  class BaseFileOps{
132  public:
133      BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135      ErrorCode Rename(std::string new_name) {
136          return MoveFile(m_fileName, new_name);
137      }
```

Let’s dive in

```
152  class FileWriter: public BaseFileOps {
153  public:
154      FileWriter(const std::string& file_name): BaseFileOps(file_name) {};
155      bool Write(uint8_t *data, size_t data_size);
156  };
```

```
169  class Da
170  public:
171      DataDumper(const std::string& dump_to): FileWriter(dump_to) {}
172      void Dump(const std::string& file_name, uint8_t *data, size_t data_size) {
173          Write(data, data_size);
174          Rename(file_name);
175      }
176  };
```

```
131  class BaseFileOps{
132  public:
133      BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135      ErrorCode Rename(std::string new_name) {
136          return MoveFile(m_fileName, new_name);
137      }
```

Let’s dive in

```
152   class FileWriter: public BaseFileOps {
153   public:
154       FileWriter(const std::string& file_name): BaseFileOps(file_name) {};
155       bool Write(uint8_t *data, size_t data_size);
156   };
```

```
169   class Da
170   public:
171       DataDumper(const std::string& dump_to): FileWriter(dump_to) {}
172       void Dump(const std::string& file_name, uint8_t *data, size_t data_size) {
173           Write(data, data_size);
174           Rename(file_name);
175       }
176   };
```

```
131   class BaseFileOps{
132   public:
133       BaseFileOps(const std::string& file_name): m_fileName(file_name) {}
134
135       ErrorCode Rename(std::string new_name) {
136           return MoveFile(m_fileName, new_name);
137       }
```

Let’s dive in

```
186  template <typename Config>
187  class ConfigGenerator: public ConfigDumper {
188  public:
189      ConfigGenerator(std::string file_name, Config& config):
190          ConfigDumper(file_name), _raw_config(file_name, config.Serialize(), config.Size())
191          {};
192
193  private:
194      RawConfig _raw_config;
195  };
```

config.txt

```
169  class DataDumper: public FileWriter {
170  public:
171      DataDumper(const std::string& dump_to): FileWriter(dump_to) {}
172      void Dump(const std::string& file_name, uint8_t *data, size_t data_size) {
173          Write(data, data_size);
174          Rename(file_name);
175      }
176  };
```

Let’s dive in

```
186  template <typename Config>
187  class ConfigGenerator: public ConfigDumper {
188  public:
189      ConfigGenerator(std::string file_name, Config& c
190          ConfigDumper(file_name), _raw_config(file_name, config.Serialize(), config.Size())
191          {}.
139      static ErrorCode MoveFile(const std::string& source, const std::string& dest) {
140          if (0 != std::remove(dest.c_str())) {
141              return {ErrorValue::Fail, "Failed to delete " + source};
142          }
143          if (0 != std::rename(source.c_str(), dest.c_str())) {
144              return {ErrorValue::Fail, "Failed to move " + source};
145          }
146          return {ErrorValue::Success, source + " file moved to " + dest};
147      }
148
149      std::string m_fileName;
150  };
175      }
176  };
```

config.txt

Deep dive into the sun

- Deep hierarchies may hide bugs
- Should we inherit or add a member?
- Shallow -> more code being read
  Deep -> rely on predecessors' logic


WHAT
DO
I DO
NOW?

Summary Checklist

- SRP abuse: Multiple classes with 1-3 substantial methods.
  - Reconsider the splits – some of these classes may make more sense joined.
- Operator overloading – is this the intuitive functionality or should this be a different function?
- Inherit or add a member?
  - Is this really a natural expansion of the API or do we simply need some utility functionality from another class?

Adi Ben David

Medium https://medium.com/@adi.ashour