CS130: Software Engineering

Lecture 9: API Design

https://forms.gle/phLY8fNttBps9vp66

A tweet: In your past experience, what made an API bad?
A word: What design principle would you invoke for a good API?
A tweet: Predict a question on the midterm.

Assignment 5

- Should still be doing CLs

  Find some stuff you can clean up or refactor
- Don’t fret about the preso

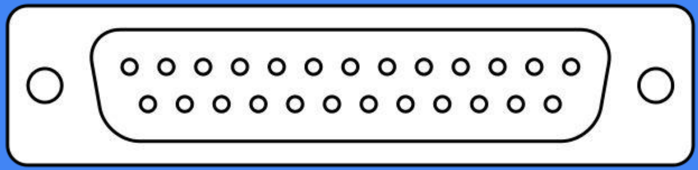
  Time during discussion is limited

  Document is more important

API Design

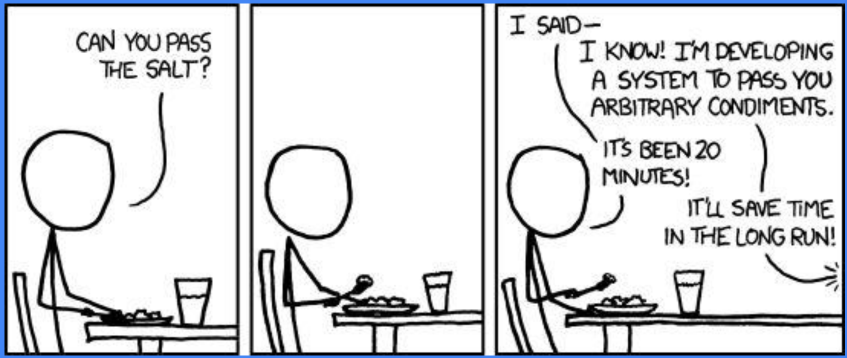
What is an API?

- Application Program Interface
- The de-facto boundary between parts of the software you are building.
- You should define this “public” interface carefully and intentionally.
- If you are writing software, you are designing APIs by default.

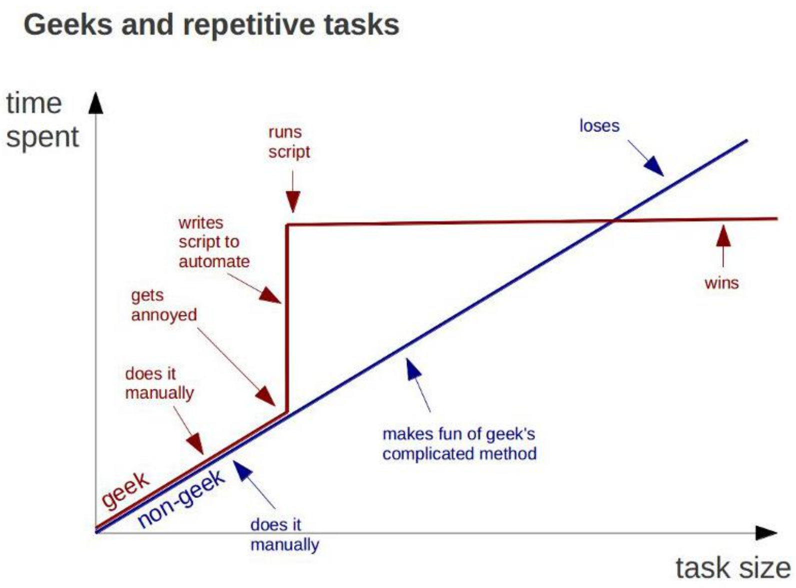


API Design

- People often think about API design when attempting to extract some common functionality that might be reused.
- Resist the temptation to immediately generalize; copy-n-paste
- Wait until you have enough examples (3+) to determine that something is indeed generalizable



API Design



- Once you've seen enough examples, seek to generalize
- Creating these building blocks will ultimately help you move faster
- However, if your building blocks have poor APIs, they will be challenging to reuse and may even slow you down.

Properties of Good API Design

Ease-of-use should be a priority:

- Easy to use correctly
- Hard to use incorrectly
- Intuitive to learn even with limited docs (but you should document them!)

Properties of Good API Design

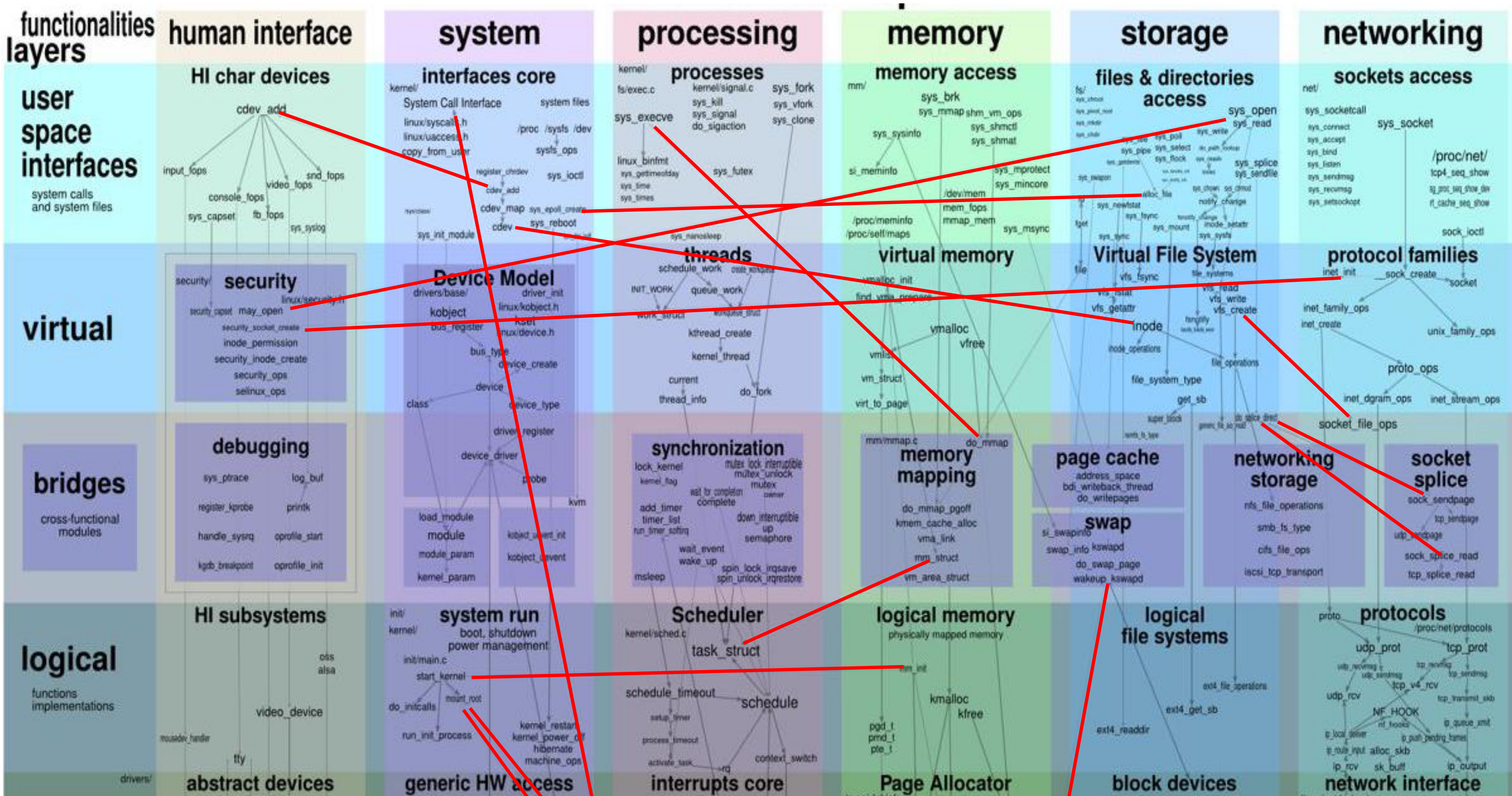
Should represent a singular coherent concept:

- Do one thing well.
- Expose a uniform level of abstraction
  - For example, an API that exposes both `UpdatePersonRecord()` and `CreateDatabaseIndex()` is operating at multiple levels.
- Sufficiently powerful to satisfy the requirements (but no more powerful).

Properties of Good API Design

Extensibility should be possible:

- Easy to extend/augment when needed.
- Exposed methods should allow multiple potential implementations.
- The implementation details shouldn't leak through the interface.
- Members should have limited visibility whenever possible.



Example: APIs within the Linux Kernel

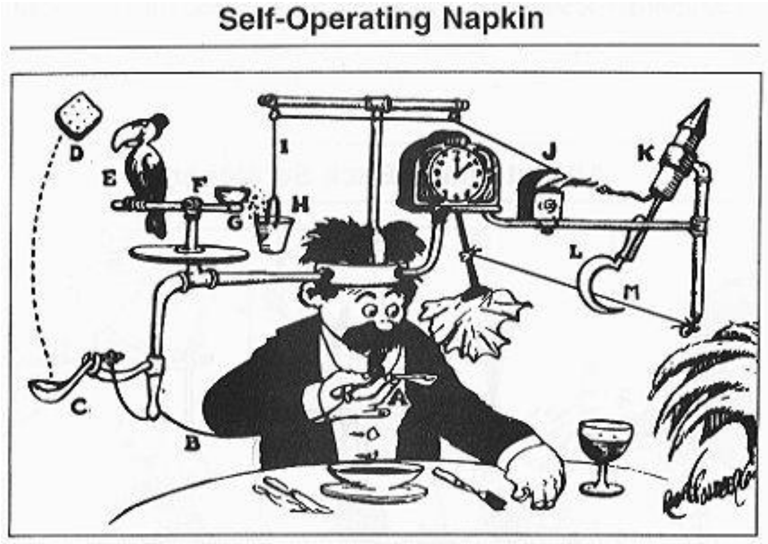
Properties of Bad API Design

Doing too many things:

- Represents several concepts. You can often identify these because they have “and” in the name.
    - A `CompressAndEncrypt` library should probably be split into two components.
- Kitchen sink methods
    - `ioctl()` is an example, but for good reason; it is a generic interface to device drivers.

```
int ioctl(
    int fd,
    unsigned long request,
    ... /* pointer to memory with further data */);
```

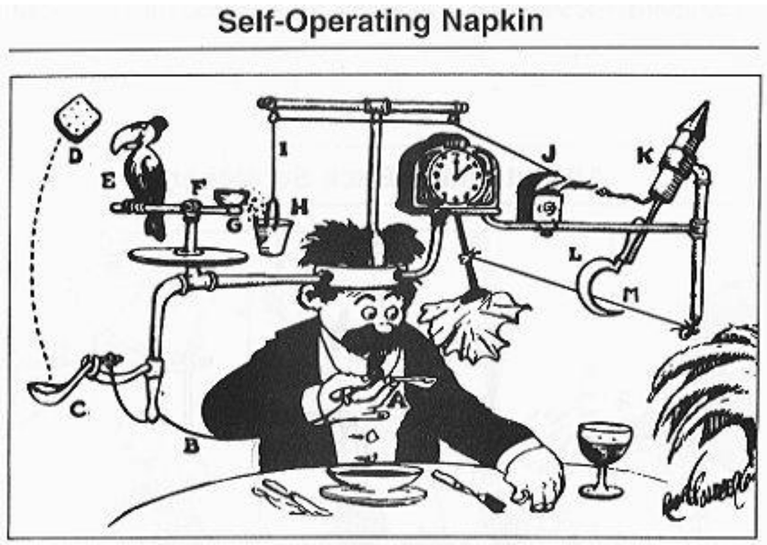
Properties of Bad API Design



Usage is awkward:

- Annoying error handling
  - For example, many linux system calls return errors via errno and strerror()
- Requires careful orchestration by the caller
  - Methods must be called in a certain order
  - Constantly handing memory back and forth
  - Caller has to maintain lots of state

Properties of Bad API Design



Usage is awkward:

- Small changes in usage result in unexpected large changes in behavior
  - For example, change one param and the performance degrades dramatically

API Hall of Shame

```java
Stack<String> stack = new Stack<String>();
stack.push("1");
stack.push("2");
stack.push("3");
stack.insertElementAt("squeeze me in!", 1);

while (!stack.isEmpty()) {
  System.out.println(stack.pop());
}
// prints "3", "2", "squeeze me in!", "1"
```

Java Stack inherits from Vector

- Results in the stack object having some awkward methods like `insert()`
- Reveals implementation details and makes it difficult to build alternate implementations

API Hall of Shame

```
vector <bool> v;
bool* pb = &v[0];
```

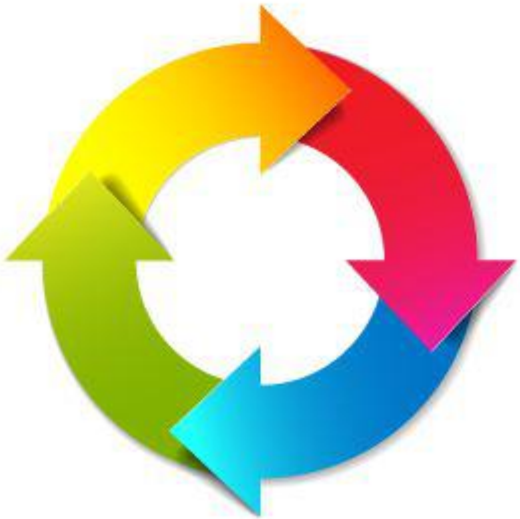
will not compile, violating requirement of STL containers.

```
cannot convert
'std::vector<bool>::reference* {aka
std::_Bit_reference*}' to 'bool*' in
initialization
```

C++ `vector<bool>` uses template specialization to swap implementation for a bit field.

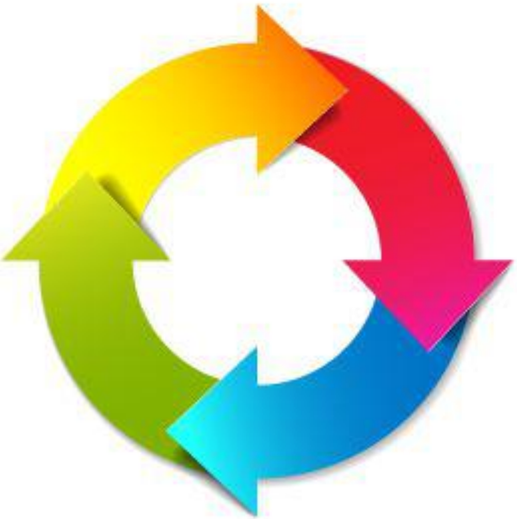
- Tried to be clever and provide an alternate implementation of vector that was more memory efficient.
- Resulted in a container that leaked its implementation in certain cases.
- Also results in small changes (`int` -> `bool`) to the usage of `vector<>` causing large changes in behavior.

API Lifecycle



- APIs are hard to kill.
  - You often don't control all the callers so there is no way to fix them all
  - For example: iOS or Linux Kernel APIs
- Design errors are hard to repair without breaking existing users.
- APIs typically only get bigger over time as use cases evolve.
- As a result, prefer to start small and simple.

Corollaries of the API Lifecycle



Because APIs are generally long lived:

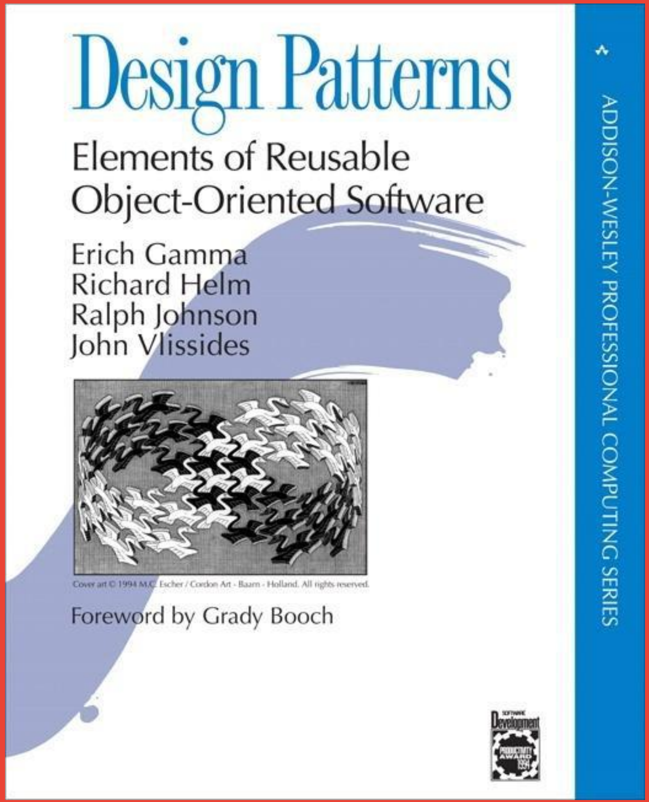
- You want to spend lots of time polishing and documenting APIs.
- Should be especially attentive when reviewing changes.
- When in doubt, leave it out; prefer to push modifications to the caller until there are sufficient examples that this usage is common.

Break

Design Patterns

Design patterns

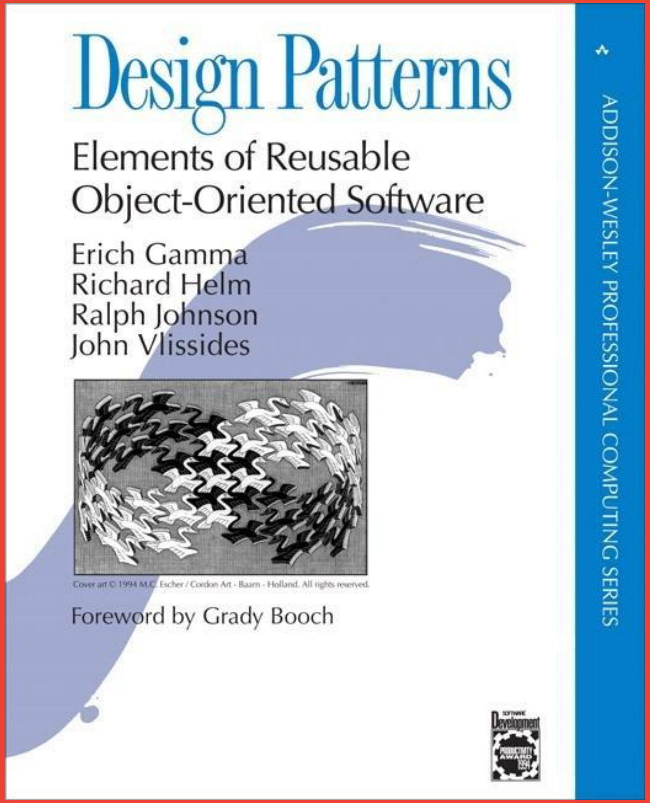
- Like refactoring, there is a book about design patterns
- A vocabulary for particular API patterns
- Helps when discussing these concepts
- We've shown you some of these already



Design patterns: An observation

These patterns are often don't seem useful when you first learn about them:

- But, it is important to have these patterns in the back of your mind when you are building things
- Over time, you'll notice places where these patterns can be used and how they can apply to your travels



Design Patterns: Observer

```
class FileReader {
public:
  void OpenFile(string file) {
    observer_->FileOpened(file);
  }

  string ReadFile() {
    observer_->FileRead(contents);
  }

  void CloseFile() {
    observer_->FileClosed(file);
  }

private:
  Observer observer_;
}
```

Good for:

- One-to-many notifications
- When notifications should be logically handled by the same remote entity
- Adding instrumentation or policy pieces to code

Con:

- You end up repeating yourself somewhat

Design Patterns: Lazy Initialization

```
class Database {
  void init() {
    if (!initialized) {
      setUpDatabase();
    }
  }

  void writeRecord(Record r) {
    init();
    reallyWriteRecord();
  }
}
```

Good for:

- Faster startup times
- Simpler initialization logic

Con:

- Can lead to unexpected and/or unpredictable runtime behavior

Design Patterns: Factories

```
Object* Build(Properties properties) {
  [...]
}
```

Good for:

- Self-documenting construction
- Named constructors
- Decoupling the construction of dependencies

Cons:

- Overused

Design Patterns: Singleton

```
template<class T> class Singleton {

  static T* Get();

  [...]
}
```

Good for:

- Process-wide state
- System access

Cons:

- Essentially a global variable
- Often considered harmful, particularly when testing

Design Patterns: Pools/Freelists

```
class ObjectPool {

  Object* take();

  void replace(Object* o);
}
```

Good for:

- Managing expensive objects (i.e. database connections)
- Central management of a scarce resource (with blocking policies)

Design Patterns: RAII

```
template<class T> class unique_ptr {
public:
  unique_ptr(T* ptr) : ptr_(ptr) {}
  ~unique_ptr() { delete ptr_; }

private:
  T* ptr_;
}

unique_ptr<MemoryBuffer> buffer;
```

Good for:

- Automatically managing resources

Con:

- Verbose and error-prone in many GC languages, though they are gradually adding some auto-closing. Use auto-closing where available!

Design Patterns: Decorators

```
Reader* reader = new BufferedReader(
  new FileReader("f"));

Handler* h = new GzipHandler(
  new StaticFileHandler(...));
```

Good for:

- Separating concerns of layered implementations that share an interface, and composing them together in a flexible way.
- Adds behavior at runtime (vs subclassing that adds at compile time)

Cons:

- Can't always hide this layering behind the same interface, so you end up with some other form of composition.

Design Pattern: Continuation

```
class Reader {
   void ReadFile(const string& fname,
                 Closure* done) {

      string contents = internalRead();

      [...]

      done->Run();
   }
}
```

Good for:

- Compositional lightweight interfaces
- Functional-style programming
- Better in languages with anonymous methods

Cons:

- Can turn your code into spaghetti like using gotos in the wrong way. So, think structured

Design Pattern: Strategy

```
void Run() {
  switch (strategy_) {
    case FOO: DoFooThing(); break;
    case BAR: DoBarThing(); break;
  }
}
```

Versus:

```
class FooThing : public Thing {}
class BarThing : public Thing {}

void Run(Thing* strategy) {
  strategy->Run();
}
```

Good for:

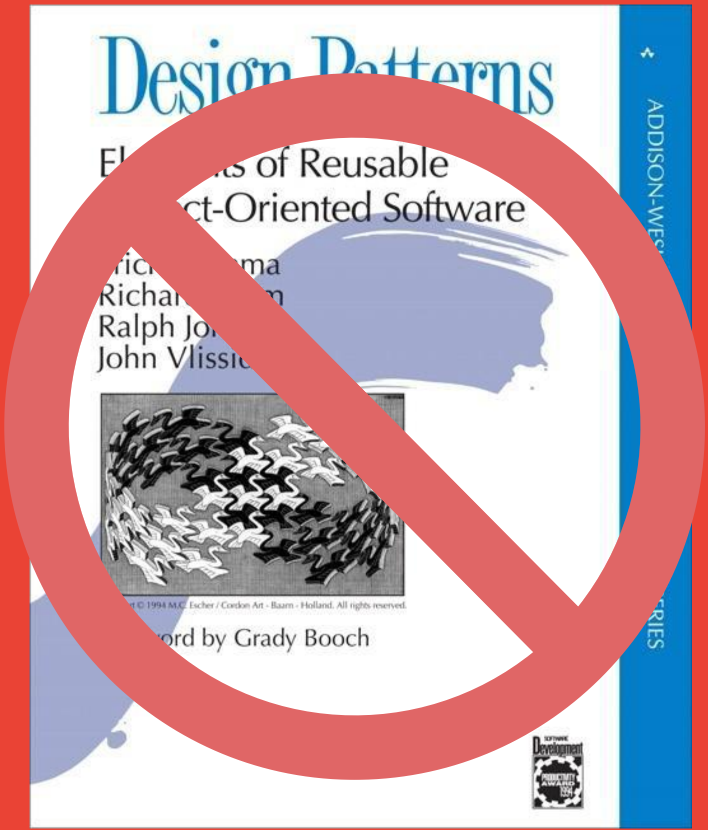
- When you have multiple implementations of a particular operation.
- Allows you to move code into a polymorphic set of objects.

Cons:

- Overuse causes unnecessary or convoluted class hierarchies.
- Makes it harder to know what's going on.

Antipatterns

- Just like design patterns, there are patterns known to be harmful
- These patterns usually result in the interfaces being more complex or hard to use



Antipattern: Stringly typed

```
void HttpHandler(const string& request,
                 string* response) {

    [...]
}
```

- When all params are strings rather than more meaningful types
- It is problematic because you have to do parsing and translation at every layer
- Also means that callers will need documentation to be sure how to interact with this API
- Prefer to use properly typed params or sensible container objects

Antipattern: Unclear Object Lifetime

```
void Foo() {
  int* i = new int;
  *i = 20;

  // Was ownership passed here?
  Bar(i);
}
```

- Lack of RAII object makes ownership less clear.
- Could be partially solved if the new'd object is a member of another object.
- `shared_ptr<>`s also an option, but ownership can still be hard to follow.
- The best option is still `unique_ptr<>` or something similar, which provides a clear understanding of lifetime.

API Example

CSV Parser Interface Critique

```
typedef vector<vector<string>> Container;

class CsvReader {
public:
  CsvReader(const string& fname,
            const string& delimiter,
            const string& quote_char,
            const string& newline);

  Container ReadLines();
}
```

- What would you change?
- How would you expect this object to be used?

CSV Parser Interface Critique

```
typedef vector<vector<string>> Container;

class CsvReader {
public:
  CsvReader(const string& fname,
            const string& delimiter,
            const string& quote_char,
            const string& newline);

  Container ReadLines();
}
```

```
void Client() {
  CsvReader r("foo", ",", "'", "\n");
  const auto c = r.ReadLines();

  for (int i = 0; i < c.size(); ++i) {
    const string& val = c[i][2];
    [...]
  }
}
```

CSV Parser Interface Critique

```
typedef vector<vector<string>> Container;

class CsvReader {
public:
  CsvReader(const string& fname,
            const string& delimiter,
            const string& quote_char,
            const string& newline);

  Container ReadLines();
}
```

```
void Client() {
  CsvReader r("foo", ",", "'", "\n");
  const auto c = r.ReadLines();

  for (int i = 0; i < c.size(); ++i) {
    const string& val = c[i][2];
    [...]
  }
}
```

CSV Parser Interface Critique

```
class CsvBuilder {
public:
  void SetFilename(const string& s);
  void SetDelimiter(const string& s);
  void SetQuote(const string& s);
  void SetNewline(const string& s);

  CsvReader* build();
}

class CsvReader {
public:
  Container ReadLines();

private:
  CsvReader();
}
```

- We can add a builder to simplify creation
- Allows us to set defaults and get rid of a method with many similar params
- In languages with named parameters (like python), might be able to avoid a builder

```
void Client() {
  CsvBuilder b;
  b.SetFilename("foo");

  unique_ptr<CsvReader> r(b.build());
}
```

CSV Parser Interface Critique

```
class CsvReader {
public:
  bool eof();
  Container Read();
}

void Client() {
  CsvReader* r = [...]

  while (!r.eof()) {
    const auto& line = r->Read();
    [...]
  }
}
```

- We could support a more natural client workflow by reading line-at-a-time
- Also allows us to stream the file rather than having to read the whole file at once

CSV Parser Interface Critique

```
typedef map<string, string> Container;

class CsvReader {
public:
  Container Read();
}

void Client() {
  [...]

  while (!r.eof()) {
    const auto& line = reader->Read();
    printf(line["First Name"].c_str();
  }
}
```

- Could use a map for the container
- Key'd by the header line and valued by the values from the current line
- Note that if we weren't streaming, this type would have to be more complex, perhaps `vector<map<string, string>>`

CSV Parser Interface (Before)

```cpp
typedef vector<vector<string>> Container;

class CsvReader {
public:
  CsvReader(const string& fname,
            const string& delimiter,
            const string& quote_char,
            const string& newline);

  Container ReadLines();
}
```

```cpp
void Client() {
  CsvReader r("foo", ",", "'", "\n");
  const auto c = r.ReadLines();

  for (int i = 0; i < c.size(); ++i) {
    const string& val = c[i][2];
    [...]
  }
}
```

CSV Parser Interface (After)

```
typedef map<string, string> Container;

class CsvBuilder {
public:
  void SetFilename(const string& fname);
  void SetDelimiter(const string& delim);
  void SetQuote(const string& quote);
  void SetNewline(const string& newline);

  CsvReader* build();
}

class CsvReader {
public:
  bool eof();
  Container Read();
}
```

```
void Client() {
  CsvBuilder b;
  b.SetFilename(“foo”);

  unique_ptr<CsvReader> r(b.build());
  while (!r.eof()) {
    const auto& line = r->Read();
    printf(line["First Name"].c_str();
  }
}
```

Midterm Preview

Impact

- 10% of your grade
- 1 assignment → 9% of your grade.

Scope

- Lectures 1-8
- Assignments 1-4

Format

- In class
- Expected time 1 hour
- Short answers
- Closed book / closed notes / closed internet
- E-mail ucla-cs130-admin@googlegroups.com if quarantining for alternate exam option

Topics we've covered

- Source control
- Testing
- Code reviews
- Tools for web server development
- Build systems
- Deployment
- Refactoring and debugging the web server
- Testability
- Static analysis
- Logging and exception handling

Source control

- We discussed and compared different revision control systems
  - How they work
  - What properties they have
- You've been using git

Testing

- Picking good test cases
- Unit testing, using fixtures and mocks
- Refactoring for testability
- Integration testing
- Other kinds of testing

Code reviews

- Why and how to do code reviews
- In-class code reviews

Webserver development

- You've started developing a web server

Build systems

- CMake
- Google's build system
- Similarities, differences, tradeoffs

Static and runtime analysis

- How they work/their applications
- What kinds of problems you can catch with each

Exception handling

- Crashing
- Logging (error logs, request logs, etc.)
- Communicating errors up the stack.

https://bit.ly/3KiDufY

We’re ½ way through and got a midterm coming...
A tweet: How do you feel about the course so far?
A tweet: What’s your study plan?

No “right” answer for all situations.
Design is about trade-offs.