Designing APIs for Simplicity and Preventing Errors

Nick Sumner
wsumner@sfu.ca
What is an API?

- API – Application Programming Interface
  - A specification of how things interact
What is an API?

- API – Application Programming Interface
  - A specification of how things interact
- Crosses many levels of design
What is an API?

- API – Application Programming Interface
  - A specification of how things interact

- Crosses many levels of design
  - Web Apps: REST, GraphQL, OpenAPI spec
What is an API?

- API – Application Programming Interface
  - A specification of how things interact
- Crosses many levels of design
  - Web Apps: REST, GraphQL, OpenAPI spec
  - Library interfaces
What is an API?

- API – Application Programming Interface
  - A specification of how things interact

- Crosses many levels of design
  - Web Apps: REST, GraphQL, OpenAPI spec
  - Library interfaces
  - Class & function definitions
What is an API?

- API – Application Programming Interface
  - A specification of how things interact
- Crosses many levels of design
  - Web Apps: REST, GraphQL, OpenAPI spec
  - Library interfaces
  - Class & function definitions
  - For some functions, even just the code within the function....
What is an API?

- **API – Application Programming Interface**
  - A specification of how things interact
- **Crosses many levels of design**
  - Web Apps: REST, GraphQL, OpenAPI spec
  - Library interfaces
  - Class & function definitions
  - For some functions, even just the code within the function....
- **An API just describes some boundary within the design process**
What makes an API good?

- Some guidance from leaders with significant experience [Bloch 2008]
  - Easy to use and hard to misuse
  - Self documenting
  - Structured by use cases
  - Strong examples
  - Displease clients with equally
  - Avoids fixed limits
  - Minimal
  - Immutable
  - Fail fast
  - ...

What makes an API good?

- Some guidance from leaders with significant experience [Bloch 2008]
  - *Easy to use and hard to misuse*
  - Self documenting
  - Structured by use cases
  - Strong examples
  - Displease clients with equally
  - Avoids fixed limits
  - Minimal
  - Immutable
  - Fail fast
  - ...

- Many of these can be seen as a vision of the first criterion
What makes an API good?

- Some guidance from leaders with significant experience [Bloch 2008]
  - Easy to use and hard to misuse
  - Self documenting
  - Structured by use cases
  - Strong examples
  - Displease clients with equally
  - Avoids fixed limits
  - Minimal
  - Immutable
  - Fail fast
  - ...

- Many of these can be seen as a vision of the first criterion
  - That will be our goal today: easy to use & hard to misuse
  - The topic expands well beyond what we have time to cover
Let us consider a problematic API

```cpp
bool isFasterThanSound(double speed) {
    return speed > MACH1;
}
```

Is this easy or hard to use? Why?
Let us consider a problematic API

```c
bool isFasterThanSound(double speed) {
    return speed > MACH1;
}
```

Is this easy or hard to use? Why?
Let us consider a problematic API

- Exposing primitive types on an API boundary leaves the user guessing
  - What are the units? Which argument is which? ...

```c
bool isFasterThanSound(double speed) {
  return speed > MACH1;
}
```
Let us consider a problematic API

- Exposing primitive types on an API boundary leaves the user guessing
  - What are the units? Which argument is which? ...
- One common form of this is a *stringly typed* API. Don’t.

```c
bool isFasterThanSound(double speed) {
    return speed > MACH1;
}
```
Let us consider a problematic API

- Exposing primitive types on an API boundary leaves the user guessing
  - What are the units? Which argument is which? ...

- One common form of this is a stringly typed API. Don’t.
Let us consider a problematic API

- Exposing primitive types on an API boundary leaves the user guessing
  - What are the units? Which argument is which? ...

- One common form of this is a stringly typed API. Don’t.

```cpp
bool isFasterThanSound(double speed) {
  return speed > MACH1;
}

(void (double speed, double angle) {
}

void feed(string food, string user) {
  feed("John Smith", "chicken");
}
Let us consider a problematic API

- Exposing primitive types on an API boundary leaves the user guessing
  - What are the units? Which argument is which? ...
- One common form of this is a stringly typed API. Don’t.

```java
void feed(string food, string user) {
    // ...
    feed("John Smith", "chicken");
}

bool isFasterThanSound(double speed) {
    return speed > MACH1;
}
```

- Ideally, only the set of appropriate values should even be possible
Let us consider a problematic API

- Exposing primitive types on an API boundary leaves the user guessing
  - What are the units? Which argument is which? ...
- One common form of this is a stringly typed API. Don’t.

```c++
bool isFasterThanSound(double speed) {
    return speed > MACH1;
}

(void (double speed, double angle) {
}

void feed(string food, string user) {
    feed("John Smith", "chicken");
}
```

- Ideally, only the set of appropriate values should even be possible
  - What name do we give to a set of values?
Use strong types to make APIs clear & prevent bugs

```c
struct User {
    ...
};

struct Food {
    ...
};

void feed(Food food, User user) {

}
```
Use strong types to make APIs clear & prevent bugs

```cpp
struct User {
    ...
};

struct Food {
    ...
};

void
feed(Food food, User user) {
}

feed(Food{"chicken"}, User{"John Smith"});
```
Use strong types to make APIs clear & prevent bugs

- Misusing the API results in a compile time error

```c
struct User {
    ...
};

struct Food {
    ...
};

void
feed(Food food, User user) {
    // Code
}

feed(Food{"chicken"}, User{"John Smith"});
```
Use strong types to make APIs clear & prevent bugs

- Misusing the API results in a compile time error
- Most IDEs will even make it particularly clear

```c
struct User {
    ...
};

struct Food {
    ...
};

void feed(Food food, User user) {
}

feed(Food{"chicken"}, User{"John Smith"});
```
Use strong types to make APIs clear & prevent bugs

- Misusing the API results in a compile time error
- Most IDEs will even make it particularly clear
- This is sometimes called a “tiny types” idiom

```c
struct User {
    ...
};

struct Food {
    ...
};

void feed(Food food, User user) {
    ...
}

feed(Food{"chicken"}, User{"John Smith"});
```
Use strong types to make APIs clear & prevent bugs

- Misusing the API results in a compile time error
- Most IDEs will even make it particularly clear
- This is sometimes called a “tiny types” idiom
- NOTE: In C++, normal type aliases are insufficient, but we have already seen strongly typed aliases

```cpp
struct User {
    ...
};

struct Food {
    ...
};

void
feed(Food food, User user) {
    ...}

feed(Food{"chicken"}, User{"John Smith"});
```
Use strong types to make APIs clear & prevent bugs

| struct User { ... }; | struct Food { ... }; | void feed(Food food, User user) { 
| | | |
| feed(Food{"chicken"}, User{"John Smith"}); |

- Misusing the API results in a compile time error
- Most IDEs will even make it particularly clear
- This is sometimes called a “tiny types” idiom
- **NOTE:** In C++, normal type aliases are insufficient, but we have already seen **strongly typed aliases**

```cpp
template<
typename Value, typename Tag>
struct StrongAlias {
  ... 
  const Value value;
};
```
Use strong types to make APIs clear & prevent bugs

- Misusing the API results in a compile time error
- Most IDEs will even make it particularly clear
- This is sometimes called a “tiny types” idiom
- **NOTE:** In C++, normal type aliases are insufficient, but we have already seen *strongly typed aliases*

```cpp
template<typename Value, typename Tag>
struct StrongAlias {
    ... 
    const Value value;
};

using Side = StrongAlias<int, struct SideTag>;
using Angle = StrongAlias<double, struct AngleTag>;

struct User {
    ...
};

struct Food {
    ...
};

void feed(Food food, User user) {
    feed(Food{"chicken"}, User{"John Smith"});
}
```
Use strong types to make APIs clear & prevent bugs

- Misusing the API results in a compile time error
- Most IDEs will even make it particularly clear
- This is sometimes called a “tiny types” idiom
- NOTE: In C++, normal type aliases are insufficient, but we have already seen strongly typed aliases

```cpp
struct User {
    ...
};
struct Food {
    ...
};
void
feed(Food food, User user) {
    // Example usage
    feed(Food{"chicken"}, User{"John Smith"});
}

template<typename Value, typename Tag>
struct StrongAlias {
    ...  
    const Value value;
};

using Side = StrongAlias<int, struct SideTag>;
using Angle = StrongAlias<double, struct AngleTag>;
```
Bool on a boundary

- Avoid booleans across an interface boundary
Avoid booleans across an interface boundary
Avoid booleans across an interface boundary
   - These are designs that frequently cause problems in practice
• Avoid booleans across an interface boundary
  – These are designs that frequently cause problems in practice
  – Does add return true when there is an error or on success?
  – Does passing true choose policy A or policy B?
Avoid booleans across an interface boundary
- These are designs that frequently cause problems in practice
- Does add return true when there is an error or on success?
- Does passing true choose policy A or policy B?
- What if I need to add another policy?!
Bool on a boundary

- Avoid booleans across an interface boundary
  - These are designs that frequently cause problems in practice
  - Does add return true when there is an error or on success?
  - Does passing true choose policy A or policy B?
  - What if I need to add another policy?!

- How can we limit the set of values on the boundary while being clearer?

```c
bool add(Element e);
void setPolicy(bool enabled);

bool result = add(e);
setPolicy(true);
```
Bool on a boundary

- Avoid booleans across an interface boundary
  - These are designs that frequently cause problems in practice
  - Does add return true when there is an error or on success?
  - Does passing true choose policy A or policy B?
  - What if I need to add another policy?!

- How can we limit the set of values on the boundary while being clearer?

```cpp
enum class AddResult {
  SUCCESS, FAILURE
};

enum class Policy {
  OptionA, OptionB, OptionC
};

bool add(Element e);
void setPolicy(bool enabled);
bool result = add(e);
setPolicy(true);
```
Avoid booleans across an interface boundary
- These are designs that frequently cause problems in practice
- Does add return true when there is an error or on success?
- Does passing true choose policy A or policy B?
- What if I need to add another policy?!

How can we limit the set of values on the boundary while being clearer?

- Recall that *sum types* capture a finite set cleanly!
- They can also force the compiler to warn when new options are unhandled!
Bool on a boundary

- Avoid booleans across an interface boundary
  - These are designs that frequently cause problems in practice
  - Does add return true when there is an error or on success?
  - Does passing true choose policy A or policy B?
  - What if I need to add another policy?!

- How can we limit the set of values on the boundary while being clearer?

```cpp
enum class AddResult {
  SUCCESS, FAILURE
};

enum class Policy {
  OptionA, OptionB, OptionC
};
```

- Recall that *sum types* capture a finite set cleanly!
- They can also force the compiler to warn when new options are unhandled!
double distanceTraveled(double speed, double time) {
    return speed * time;
}

What can go wrong?
double
distanceTraveled(double speed, double time) {
  return speed * time;
}

// Miles per hour * seconds?
... = distanceTraveled(3, 5);

d1 = ...; // Meters
d2 = ...; // Miles
... = d1 + d2; // Uh oh.

What can go wrong?
double
distanceTraveled(double speed, double time) {
    return speed * time;
}

// Miles per hour * seconds?
... = distanceTraveled(3, 5);

d1 = ...; // Meters
d2 = ...; // Miles
... = d1 + d2; // Uh oh.
Phantom Types

- Parameterize your types by unique type names...

```cpp
struct Meters {};
struct Miles {};
struct Seconds {};
struct Hours {};

template<typename T, typename U>
struct Speed { double speed; };

template<typename T>
struct Distance { double distance; };

template<typename T>
struct Time { double time; };
```
Phantom Types

- Parameterize your types by unique type names...

```cpp
struct Meters {};
struct Miles {};
struct Seconds {};
struct Hours {};

template <typename T, typename U>
struct Speed { double speed; };

template <typename T>
struct Distance { double distance; };

template <typename T>
struct Time { double time; };
```

Speed is parameterized by time & a unit of length
Phantom Types

- Consistent units are enforced via template arguments

```cpp
template <typename T, typename U>
Distance<T>
distanceTraveled(Speed<T,U> speed, Time<U> time) {
    return {speed.speed * time.time};
}

template <typename T>
Distance<T>
operator+(Distance<T> d1, Distance<T> d2) {
    return d1.distance + d2.distance;
}
```
Phantom Types

- Consistent units are enforced via template arguments

```cpp
template <typename T, typename U>
Distance<T>
distanceTraveled(Speed<T, U> speed, Time<U> time) {
    return {speed.speed * time.time};
}

template <typename T>
Distance<T>
operator+(Distance<T> d1, Distance<T> d2) {
    return d1.distance + d2.distance;
}
```
Phantom Types

distanceTraveled(Speed<Miles, Hours>{3}, Time<Seconds>{5});
Phantom Types

distanceTraveled(Speed<Miles, Hours>{3}, Time<Seconds>{5});

phantom.cpp:37:19: error: no matching function for call to 'distanceTraveled'
... deduced conflicting types for parameter 'U' ('Hours' vs. 'Seconds')
Phantom Types

d1 = distanceTraveled(Speed<Miles, Hours>{3}, Time<Seconds>{5});
d2 = distanceTraveled(Speed<Meters, Seconds>{3}, Time<Seconds>{5});
d3 = d2 + d3;

phantom.cpp:37:19: error: no matching function for call to 'distanceTraveled'
... deduced conflicting types for parameter 'U' ('Hours' vs. 'Seconds')
Phantom Types

distanceTraveled(Speed<Miles, Hours>{3}, Time<Seconds>{5});

d1 = distanceTraveled(Speed<Miles, Hours>{3}, Time<Hours>{5});
d2 = distanceTraveled(Speed<Meters, Seconds>{3}, Time<Seconds>{5});
d3 = d2 + d3;

phantom.cpp:37:19: error: no matching function for call to 'distanceTraveled'
... deduced conflicting types for parameter 'U' ('Hours' vs. 'Seconds')

phantom.cpp:41:30: error: invalid operands to binary expression
... deduced conflicting types for parameter 'T' ('Miles' vs. 'Meters')
Phantom Types

distanceTraveled(Speed<Miles, Hours>{3}, Time<Seconds>{5});

phantom.cpp:37:19: error: no matching function for call to 'distanceTraveled'
... deduced conflicting types for parameter 'U' ('Hours' vs. 'Seconds')

d1 = distanceTraveled(Speed<Miles, Hours>{3}, Time<Hours>{5});
d2 = distanceTraveled(Speed<Meters, Seconds>{3}, Time<Seconds>{5});
d3 = d2 + d3;

phantom.cpp:41:30: error: invalid operands to binary expression
... deduced conflicting types for parameter 'T' ('Miles' vs. 'Meters')

What are the trade offs for using this technique?
Avoiding Inconsistent State
Avoiding Inconsistent State

```cpp
enum class CurrentState { SLEEP, PLAY, WORK }

class Student {
    CurrentState state;
    uint64_t timeWorked;
};
```
Avoiding Inconsistent State

define class CurrentState {
SLEEP, PLAY, WORK
};

class Student {
CurrentState state;
uint64_t timeWorked;
};
Avoiding Inconsistent State

- How can we fix it?

```cpp
class Student {
    unique_ptr<CurrentState> state;
};
```
Avoiding Inconsistent State

- How can we fix it?

class Student {
    unique_ptr<CurrentState> state;
};

class CurrentState {
    ...
};
Avoiding Inconsistent State

- How can we fix it?

```cpp
class Student {
    unique_ptr<CurrentState> state;
};

class CurrentState {
    ...
};

class Sleep : public CurrentState {
};

class Work : public CurrentState {
    uint64_t timeWorked;
};
```
Avoiding Inconsistent State

- How can we fix it?

```cpp
class Student {
    unique_ptr<CurrentState> state;
};

class CurrentState {
};

This is part of the state pattern!

class Sleep : public CurrentState {
};

class Work : public CurrentState {
    uint64_t timeWorked;
};
```
Avoiding Inconsistent State

- How can we fix it?

```cpp
class Student {
    struct Sleep {};
    struct Play {};
    struct Work { uint64_t timeWorked; };
    std::variant<Sleep, Play, Work> currentState;
};
```
Avoiding Inconsistent State

- How can we fix it?

class Student {
    struct Sleep {};
    struct Play {};
    struct Work { uint64_t timeWorked; };

    std::variant<Sleep, Play, Work> currentState;
};
Avoiding Inconsistent State

- How can we fix it?

class Student {
  struct Sleep {};
  struct Play {};
  struct Work { uint64_t timeWorked; };

  std::variant<Sleep, Play, Work> currentState;
};
Generalize away corner cases

- Sometimes complexity comes because an abstraction is *too specific*!
Generalize away corner cases

- Sometimes complexity comes because an abstraction is too specific!
  - We can generalize the interface to handle corner cases transparently
Generalize away corner cases

- Sometimes complexity comes because an abstraction is too specific!
  - We can generalize the interface to handle corner cases transparently
- Consider a tree that may be traversed
Generalize away corner cases

- Sometimes complexity comes because an abstraction is *too specific*!
  - We can generalize the interface to handle corner cases transparently
- Consider a tree that may be traversed
- Implicitly
  - e.g. the *null object* pattern

**Null Object Pattern**
Create a subtype representing an object with no information.

Any getters/methods effectively perform no-ops.
Generalize away corner cases

- Sometimes complexity comes because an abstraction is *too specific*!
  - We can generalize the interface to handle corner cases transparently

- Consider a tree that may be traversed

- Implicitly
  - e.g. the *null object* pattern

```c
struct Node {
    void traverseInOrder(auto onNode);
    Node* left;
    Node* right
    int value;
};
```
Generalize away corner cases

- Sometimes complexity comes because an abstraction is *too specific*!
  - We can generalize the interface to handle corner cases transparently

- Consider a tree that may be traversed

- Implicitly
  - e.g. the *null object* pattern

```c
struct Node {
    void traverseInOrder(auto onNode);
    Node* left;
    Node* right;
    int value;
};

root->traverseInOrder(printValue);
```
Generalize away corner cases

- Sometimes complexity comes because an abstraction is *too specific*!
  - We can generalize the interface to handle corner cases transparently

- Consider a tree that may be traversed

  - Implicitly
    - e.g. the *null object* pattern

```
struct Node {
    void traverseInOrder(auto onNode);

    Node* left;
    Node* right
    int value;
};

void root->traverseInOrder(printValue);

void Node::traverseInOrder(auto onNode) {
    if (left) left->traverseInOrder(onNode);
    onNode(this);
    if (right) right->traverseInOrder(onNode);
}
```
Sometimes complexity comes because an abstraction is too specific!
  
  We can generalize the interface to handle corner cases transparently

Consider a tree that may be traversed

Implicitly
  
  e.g. the null object pattern

```c++
struct Node {
    virtual void traverseInOrder(auto onNode) = 0;
};

struct Node {
    void traverseInOrder(auto onNode);
    Node* left;
    Node* right;
    int value;
};

root->traverseInOrder(printValue);

void
Node::traverseInOrder(auto onNode) {
    if (left) left->traverseInOrder(onNode);
    onNode(this);
    if (right) right->traverseInOrder(onNode);
}
Sometimes complexity comes because an abstraction is too specific!

- We can generalize the interface to handle corner cases transparently.

Consider a tree that may be traversed implicitly—e.g. the null object pattern.

```cpp
struct Node {
    void traverseInOrder(auto onNode);
    Node* left;
    Node* right;
    int value;
};

void root->traverseInOrder(printValue);

struct InternalNode : public Node {
    void traverseInOrder(auto onNode) override {
        left->traverseInOrder(onNode);
        onNode(this);
        right->traverseInOrder(onNode);
    }
    int value;
};

void Node::traverseInOrder(auto onNode) {
    if (left) left->traverseInOrder(onNode);
    onNode(this);
    if (right) right->traverseInOrder(onNode);
}
```
Sometimes complexity comes because an abstraction is too specific! We can generalize the interface to handle corner cases transparently.

Consider a tree that may be traversed implicitly—e.g. the null object pattern.

```c++
struct Node {
    void traverseInOrder(auto onNode);
    Node* left;
    Node* right;
    int value;
};

root->traverseInOrder(printValue);

void Node::traverseInOrder(auto onNode) {
    if (left) left->traverseInOrder(onNode);
    onNode(this);
    if (right) right->traverseInOrder(onNode);
}

struct InternalNode : public Node {
    void traverseInOrder(auto onNode) override {
        left->traverseInOrder(onNode);
        onNode(this);
        right->traverseInOrder(onNode);
    }
    int value;
};

struct LeafNode : public Node {
    void traverseInOrder(auto onNode) override {
    }
};
```

• Implicitly
  - e.g. the null object pattern
Generalize away corner cases

• Sometimes complexity comes because an abstraction is *too specific*!
  – We can generalize the interface to handle corner cases transparently

• **Consider a tree that may be traversed**

• Implicitly
  – e.g. the *null object* pattern

• Explicitly
  – e.g. `getChildren()` vs `getLeft()` & `getRight()`
Generalize away corner cases

• Sometimes complexity comes because an abstraction is *too specific*!
  – We can generalize the interface to handle corner cases transparently

• Consider a tree that may be traversed

• Implicitly
  – e.g. the *null object* pattern

• Explicitly
  – e.g. `getChildren()` vs `getLeft()` & `getRight()`

What are the trade offs?
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors

```java
ComplexProcess p;
p.doThing1();
p.doThing2();
p.doThing3();
```
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors

```java
ComplexProcess p;
p.doThing1();
p.doThing2();
p.doThing3();
```

```java
ComplexProcess p;
p.doThing1();
p.doThing3();
p.doThing2();
```
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors.
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors.
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.

```c
struct ComplexProcess {
    Stage1 doStep1();
} FONT-size:15px
```

```c
struct Stage1 {
    Stage2 doStep2();
} FONT-size:15px
```

```c
struct Stage2 {
    void doStep3();
} FONT-size:15px
```
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.

```cpp
struct ComplexProcess {
    Stage1 doStep1();
}

struct Stage1 {
    Stage2 doStep2();
}

struct Stage2 {
    void doStep3();
}

ComplexProcess p;
p.doStep1()
  .doStep2()
  .doStep3();
```
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors.
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.

```cpp
struct ComplexProcess {
    Stage1 doStep1();
}

struct Stage1 {
    Stage2 doStep2();
}

struct Stage2 {
    void doStep3();
}

ComplexProcess p;
p.doStep1()
    .doStep2()
    .doStep3();
```

We can make invalid usage a compilation error.
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.

```cpp
struct ComplexProcess {
    Stage1 doStep1();
}

struct Stage1 {
    Stage2 doStep2();
}

struct Stage2 {
    void doStep3();
}

ComplexProcess p;
    p.doStep1()
        .doStep2()
        .doStep3();
```

```cpp
ComplexProcess p;
    p.doStep1()
        .doStep2();
    p.doStep3();
```
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.

```cpp
struct ComplexProcess {
    [[nodiscard]] Stage1 doStep1();
};

struct Stage1 {
    [[nodiscard]] Stage2 doStep2();
};

struct Stage2 {
    void doStep3();
};

ComplexProcess p;
p.doStep1().doStep3();

ComplexProcess p;
p.doStep1().doStep2().doStep3();
```
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.

```cpp
struct ComplexProcess {
    Stage1 doStep1();
}

[[nodiscard]] struct Stage1 {
    Stage2 doStep2();
}

[[nodiscard]] struct Stage2 {
    void doStep3();
}

ComplexProcess p;
ComplexProcess p; p.doStep1()
    .doStep3();
ComplexProcess p; p.doStep1()
    .doStep2();
    .doStep3();
```
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.
- In practice, you can express things like
  - Selecting from options
  - Sequencing
  - Iteration

  state machines

using nothing more than return types!
Fluent APIs

- Fluent APIs use strong return types to enforce correct behaviors
- By returning a new type that controls the available behaviors, you can enforce the protocols you want.
- In practice, you can express things like
  - Selecting from options
  - Sequencing
  - Iteration

\[
\text{InSequence dummy;} \quad \text{EXPECT\_CALL(mockThing, foo(Ge(20)))}
\]
\[
\quad .\text{Times}(2) \ // \text{Can be omitted here}
\quad .\text{WillOnce(Return(100))}
\quad .\text{WillOnce(Return(200))};
\]
\[
\text{EXPECT\_CALL(mockThing, bar(Lt(5)))};
\]
Monadic APIs

- Monadic APIs use patterns from functional languages to hide corner cases behind an API.
Monadic APIs

- Monadic APIs use patterns from functional languages to hide corner cases behind an API
  - There is a rich formalism behind them that provides composability
  - These are increasingly common (Java, Javascript, C++, ...)
  - In fact, we have already seen some in class!
Monadic APIs

- Monadic APIs use patterns from functional languages to hide corner cases behind an API
  - There is a rich formalism behind them that provides composability
  - These are increasingly common (Java, Javascript, C++, ...)
  - In fact, we have already seen some in class!

```cpp
int total = accumulate(view::iota(1)
  | view::transform([](int x){return x*x;})
  | view::take(10), 0);
```

[Milewski 2014]
Monadic APIs

- Monadic APIs use patterns from functional languages to hide corner cases behind an API
  - There is a rich formalism behind them that provides composability
  - These are increasingly common (Java, Javascript, C++, ...)
  - In fact, we have already seen some in class!
- We can create an abstraction for a specific design concern, hide burdens of it within a clean API, & push behaviors into the API that handles the concern.
Monadic APIs

Monadic APIs use patterns from functional languages to hide corner cases behind an API
- There is a rich formalism behind them that provides composability
- These are increasingly common (Java, Javascript, C++, ...)
- In fact, we have already seen some in class!

We can create an abstraction for a specific design concern, hide burdens of it within a clean API, & push behaviors into the API that handles the concern.
- Create: $z \rightarrow A[z]$
- Bind: $(A[x], x \rightarrow A[y]) \rightarrow A[y]$
Monadic APIs

- Monadic APIs use patterns from functional languages to hide corner cases behind an API.
  - There is a rich formalism behind them that provides composability.
  - These are increasingly common (Java, Javascript, C++, ...).
  - In fact, we have already seen some in class!

- We can create an abstraction for a specific design concern, hide burdens of it within a clean API, & push behaviors into the API that handles the concern.
  - Create: \( z \rightarrow A[z] \)
  - Bind: \( (A[x], x \rightarrow A[y]) \rightarrow A[y] \)

```cpp
int total = accumulate(
  view::iota(1),
  view::transform([](int x){return x*x;}),
  view::take(10), 0);
```
Monadic APIs

- Monadic APIs use patterns from functional languages to hide corner cases behind an API
  - There is a rich formalism behind them that provides composability
  - These are increasingly common (Java, Javascript, C++, ...)
  - In fact, we have already seen some in class!
- We can create an abstraction for a specific design concern, hide burdens of it within a clean API, & push behaviors into the API that handles the concern.
  - Create: $z \rightarrow A[z]$
  - Bind: $(A[x], x \rightarrow A[y]) \rightarrow A[y]$
- In fact, `Option` is a monad in many languages
Monadic APIs

Monadic APIs use patterns from functional languages to hide corner cases behind an API. There is a rich formalism behind them that provides composability. These are increasingly common (Java, Javascript, C++, ...). In fact, we have already seen some in class!

```cpp
std::optional<image> get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
        return std::nullopt;
    }

    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
        return std::nullopt;
    }

    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
        return std::nullopt;
    }

    return add_rainbow(make_smaller(*with_sparkles));
}
```

[Brand 2017]
Monadic APIs

- Monadic APIs use patterns from functional languages to hide corner cases behind an API.
- There is a rich formalism behind them that provides composability.
- These are increasingly common (Java, Javascript, C++, ...).
- In fact, we have already seen some in class!

```cpp
std::optional<image> get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
        return std::nullopt;
    }
    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
        return std::nullopt;
    }
    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
        return std::nullopt;
    }
    return add_rainbow(make_smaller(*with_sparkles));
}
```

[Brand 2017]
Monadic APIs

Monadic APIs use patterns from functional languages to hide corner cases behind an API.

- There is a rich formalism behind them that provides composability.
- These are increasingly common (Java, Javascript, C++, ...)
- In fact, we have already seen some in class!

```cpp
std::optional<image>
get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
        return std::nullopt;
    }

    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
        return std::nullopt;
    }

    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
        return std::nullopt;
    }

    return add_rainbow(make_smaller(*with_sparkles));
}
```

[Brand 2017]
Monadic APIs

Monadic APIs use patterns from functional languages to hide corner cases behind an API. There is a rich formalism behind them that provides composability. These are increasingly common (Java, Javascript, C++, ...). In fact, we have already seen some in class!

```cpp
std::optional<image> get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
        return std::nullopt;
    }

    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
        return std::nullopt;
    }

    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
        return std::nullopt;
    }

    return add_rainbow(make_smaller(*with_sparkles));
}
```

[Brand 2017]
Monadic APIs

Monadic APIs use patterns from functional languages to hide corner cases behind an API. There is a rich formalism behind them that provides composability. These are increasingly common (Java, Javascript, C++, ...). In fact, we have already seen some in class!

```c++
std::optional<image> get_cute_cat(const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
        return std::nullopt;
    }
    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
        return std::nullopt;
    }
    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
        return std::nullopt;
    }
    return add_rainbow(make_smaller(*with_sparkles));
}
```

[Brand 2017]
Monadic APIs

Monadic APIs use patterns from functional languages to hide corner cases behind an API. There is a rich formalism behind them that provides composability. These are increasingly common (Java, JavaScript, C++, ...). In fact, we have already seen some in class!

```cpp
std::optional<image> get_cute_cat (const image& img) {
    auto cropped = crop_to_cat(img);
    if (!cropped) {
        return std::nullopt;
    }
    auto with_tie = add_bow_tie(*cropped);
    if (!with_tie) {
        return std::nullopt;
    }
    auto with_sparkles = make_eyes_sparkle(*with_tie);
    if (!with_sparkles) {
        return std::nullopt;
    }
    return add_rainbow(make_smaller(*with_sparkles));
}
```

[Brand 2017]
Monadic APIs

Monadic APIs use patterns from functional languages to hide corner cases behind an API. There is a rich formalism behind them that provides composability. These are increasingly common (Java, Javascript, C++, ...). In fact, we have already seen some in class!

```cpp
std::optional<image>
get_cute_cat (const image& img) {
    return crop_to_cat(img)
        .and_then(add_bow_tie)
        .and_then(make_eyes_sparkle)
        .map(make_smaller)
        .map(add_rainbow);
}
```

[Brand 2017]
Other more advanced topics?

- Versioning
- Performance
- Wire protocols (more like GraphQL, protobufs, etc.)
Summary

- Try to make your APIs
  - express essential complexity of the boundary
  - hide the corner cases of the implementation
Summary

- Try to make your APIs
  - express essential complexity of the boundary
  - hide the corner cases of the implementation

- Use types to your advantage in the process
  - Strong, expressive types
  - Fluent APIs to direct flow
  - Monadic APIs for composability while abstracting out complexity