CMPT 373
Software Development Methods

Design Patterns

Nick Sumner
wsumner@sfu.ca

Recall: Managing Complexity

The most fundamental issue in software development
is *managing complexity*

Recall: Managing Complexity

The most fundamental issue in software development
is *managing complexity*

Complexity:

- Has many forms

Recall: Managing Complexity

The most fundamental issue in software development
is *managing complexity*

Complexity:

- Has many forms
- One broad notion is *coupling*

Recall: Managing Complexity

The most fundamental issue in software development
is *managing complexity*

Complexity:

- Has many forms
- One broad notion is *coupling*
  - Can one component be ***understood*** without others?
  - Can one component be ***changed*** without changing others?

Recall: Managing Complexity

The most fundamental issue in software development
is *managing complexity*

Complexity:

- Has many forms
- One broad notion is *coupling*
  - Can one component be ***understood*** without others?
  - Can one component be ***changed*** without changing others?

Solutions are built using:

Recall: Managing Complexity

The most fundamental issue in software development
is *managing complexity*

Complexity:

- Has many forms
- One broad notion is *coupling*
  - Can one component be ***understood*** without others?
  - Can one component be ***changed*** without changing others?

Solutions are built using:

- Abstraction
- Encapsulation
- Information hiding

Recall: Managing Complexity

The most fundamental issue in software development
is *managing complexity*

Complexity:

- Has many forms
- One broad notion is *coupling*
  - Can one component be ***understood*** without others?
  - Can one component be ***changed*** without changing others?

Solutions are built using:

- Abstraction
- Encapsulation
- Information hiding

Strive for components that:
- interact minimally
- know minimal information

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.
  - *Archetypes*

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.
  - *Archetypes*



What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.
  - *Archetypes*
    - Their trade-offs are well understood
    - New solutions can be *modelled after* them effectively

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.
  - *Archetypes*
    - Their trade-offs are well understood
    - New solutions can be *modelled after* them effectively

Note:

- As in literature, you ***do not copy the archetype*** directly.

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.
  - *Archetypes*
    - Their trade-offs are well understood
    - New solutions can be *modelled after* them effectively

Note:

- As in literature, you do not copy the archetype directly.
- Adapt it to your specific needs & trade offs.

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.
  - *Archetypes*
    - Their trade-offs are well understood
    - New solutions can be *modelled after* them effectively

Note:

- As in literature, you do not copy the archetype directly.
- Adapt it to your specific needs & trade offs.
- ***Why*** a pattern exists is more important than just knowing that pattern

What are design patterns?

- ***Design patterns*** are reusable solutions and metaphors for addressing problems
- They provide
  - *Common Language*
    - discuss complex solutions more easily by name.
  - *Archetypes*
    - Their trade-offs are well understood
    - New solutions can be *modelled after* them effectively

Note:

- As in literature, you do not copy the archetype directly.
- Adapt it to your specific needs & trade offs.
- ***Why*** a pattern exists is more important than just knowing that pattern !

So what is their benefit?

- Design patterns…

So what is their benefit?

- Design patterns…
  - have clear formulations of the problems they attack

Your problems will usually be slightly different

So what is their benefit?

- Design patterns…
  - have clear formulations of the problems they attack
  - enable efficient communication

So what is their benefit?

- Design patterns…
  - have clear formulations of the problems they attack
  - enable efficient communication
  - have well understood strengths & weaknesses

So what is their benefit?

- Design patterns…
  - have clear formulations of the problems they attack
  - enable efficient communication
  - have well understood strengths & weaknesses
  - provide *anchor points* in the design space that you can *explore*

What are their risks?

What are their risks?

- Solutions can be *built around* design patterns rather than *informed by* them.

What are their risks?

- Solutions can be built around design patterns rather than informed by them.
- Emergent tradeoffs can be hidden by adopting a pattern too early.

What are their risks?

- Solutions can be built around design patterns rather than informed by them.
- Emergent tradeoffs can be hidden by adopting a pattern too early.

Start simple and adopt or move toward design patterns as their utility becomes clear.

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism
- ***Polymorphism***
  - Using a common interface for many types

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism
- ***Polymorphism***
  - Using a common interface for many types
- 4(ish) common types of polymorphism:

What are they?

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism
- ***Polymorphism***
  - Using a common interface for many types
- 4 common types of polymorphism:
  - Inheritance / Subtyping (at runtime)

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism
- ***Polymorphism***
  - Using a common interface for many types
- 4 common types of polymorphism:
  - Inheritance / Subtyping (at runtime)
  - Parametric polymorphism (at compile time)

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism
- ***Polymorphism***
  - Using a common interface for many types
- 4 common types of polymorphism:
  - Inheritance / Subtyping (at runtime)
  - Parametric polymorphism (at compile time)
  - Ad hoc polymorphism / overloading / type classes

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism
- ***Polymorphism***
  - Using a common interface for many types
- 4 common types of polymorphism:
  - Inheritance / Subtyping (at runtime)
  - Parametric polymorphism (at compile time)
  - Ad hoc polymorphism / overloading / type classes
  - Coercion / casting (ish)

What are the puzzle pieces?

- Design patterns are largely built around exploiting
  - composition
  - polymorphism
- ***Polymorphism***
  - Using a common interface for many types
- 4 common types of polymorphism:
  - Inheritance / Subtyping (at runtime)
  - Parametric polymorphism (at compile time)
  - Ad hoc polymorphism / overloading / type classes
  - Coercion / casting

Choosing one form of polymorphism over another yields trade-offs

3 classical categories

- ***Creational***
  - Support creation of objects within a program

3 classical categories

- ***Creational***
  - Support creation of objects within a program
- ***Structural***
  - Organize object composition for creating new behavior

3 classical categories

- *Creational*
  - Support creation of objects within a program
- *Structural*
  - Organize object composition for creating new behavior
- *Behavioral*
  - Focus on communication between entities

3 classical categories

- ***Creational***
  - Support creation of objects within a program
- ***Structural***
  - Organize object composition for creating new behavior
- ***Behavioral***
  - Focus on communication between entities

Other categories exist for specific domains.
These are general.

Deriving Designs & Recognizing Patterns

- We will derive a handful of patterns in these categories

Deriving Designs & Recognizing Patterns

- We will derive a handful of patterns in these categories
- I want us to try to construct them from first principles

Deriving Designs & Recognizing Patterns

- We will derive a handful of patterns in these categories
- I want us to try to construct them from first principles
  - Identify goals
  - Understand the constraints of a scenario
  - Derive a design that does what you want

Deriving Designs & Recognizing Patterns

- We will derive a handful of patterns in these categories
- I want us to try to construct them from first principles
  - Identify goals
  - Understand the constraints of a scenario
  - Derive a design that does what you want
- I expect the patterns to be obvious in retrospect….

Problem: Flexibly creating objects

- How would you normally create an instance of an object?

Problem: Flexibly creating objects

- How would you normally create an instance of an object?

```
Animal animal{"Zebra", RunPolicy, WinnyPolicy};
```

Problem: Flexibly creating objects

- How would you normally create an instance of an object?
- What are the coupled constraints in this approach?

```
Animal animal{"Zebra", RunPolicy, WinnyPolicy};
```

Problem: Flexibly creating objects

- How would you normally create an instance of an object?
- What are the coupled constraints in this approach?

```
Animal animal{"Zebra", RunPolicy, WinnyPolicy};
```

Note, there are also temporal constraints!
When are the arguments & types known?

Problem: Flexibly creating objects

- How would you normally create an instance of an object?
- What are the coupled constraints in this approach?
- What if you want to allow the user to define their own kinds of objects to create? (custom paintbrush for objects)

```
Animal animal{"Zebra", RunPolicy, WinnyPolicy};
```

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change
  - Might be created far from where arguments are known

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change
  - Might be created far from where arguments are known

How would you attack this?

```
...
Animal animal{"Zebra", RunPolicy, WinnyPolicy};
...
```

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change
  - Might be created far from where arguments are known

How would you attack this?

```
Animal animal{"Zebra", RunPolicy, WinnyPolicy};
```

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change
  - Might be created far from where arguments are known

How would you attack this?

```
Animal animal = maker.makeOne();
```

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change
  - Might be created far from where arguments are known

How would you attack this?

```
class ThingMaker{
  //info about
  //thing to make
  Animal makeOne();
} maker;
```

```
Animal animal = maker.makeOne();
```

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change
  - Might be created far from where arguments are known

How would you attack this?

```
class ThingMaker{
  Animal toCopy;
public:
  Animal makeOne();
} maker;
```

```
Animal animal = maker.makeOne();
```

Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be *costly to build*
  - First instance might be user created
  - Actual type may need to change
  - Might be created far from where arguments are known

- Register an instance as a template & make clones

e.g. Creational Pattern: Prototype

- Goal: Create new objects based on a configuration.

e.g. Creational Pattern: Prototype

- Goal: Create new objects based on a configuration.

An inheritance version:

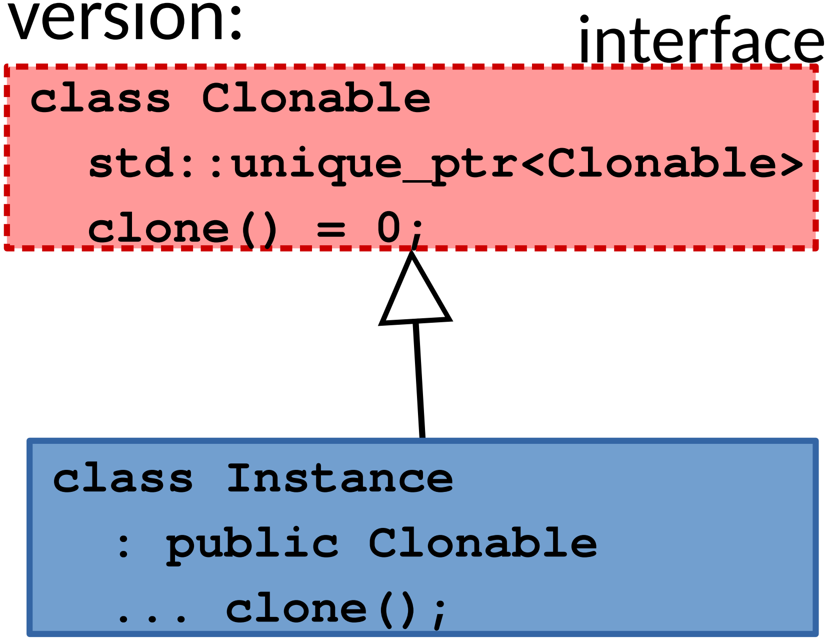
interface

```
class Clonable
  std::unique_ptr<Clonable>
  clone() = 0;
```

e.g. Creational Pattern: Prototype

- Goal: Create new objects based on a configuration.

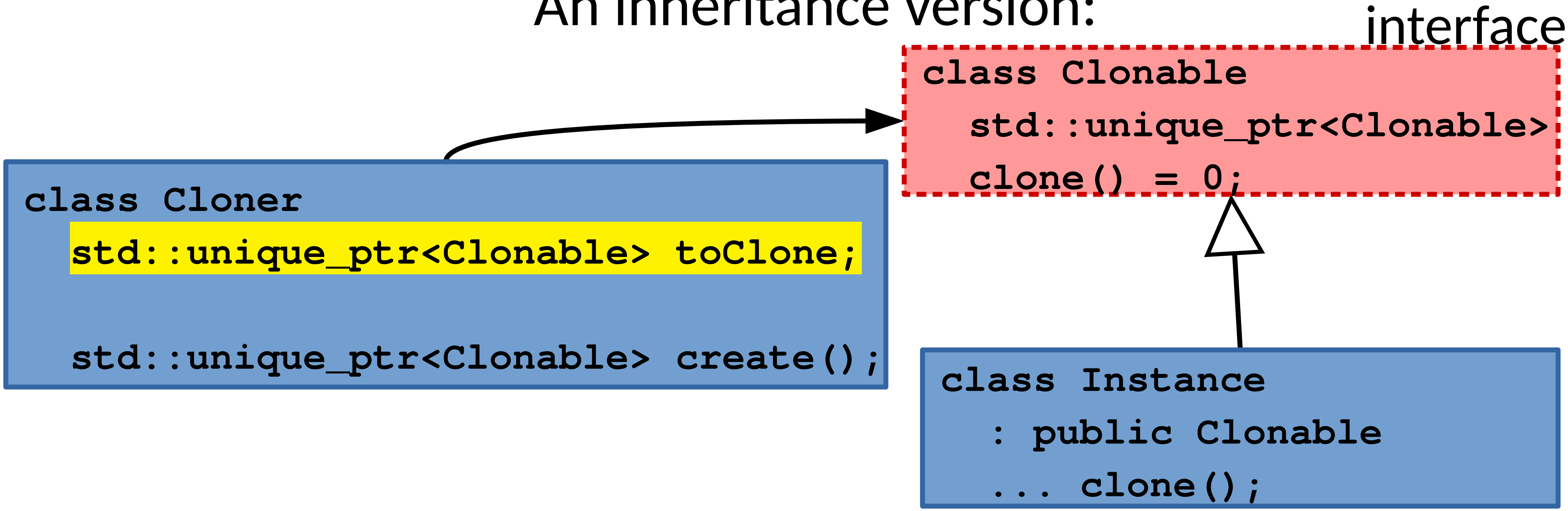
An inheritance version:



e.g. Creational Pattern: Prototype

- Goal: Create new objects based on a configuration.

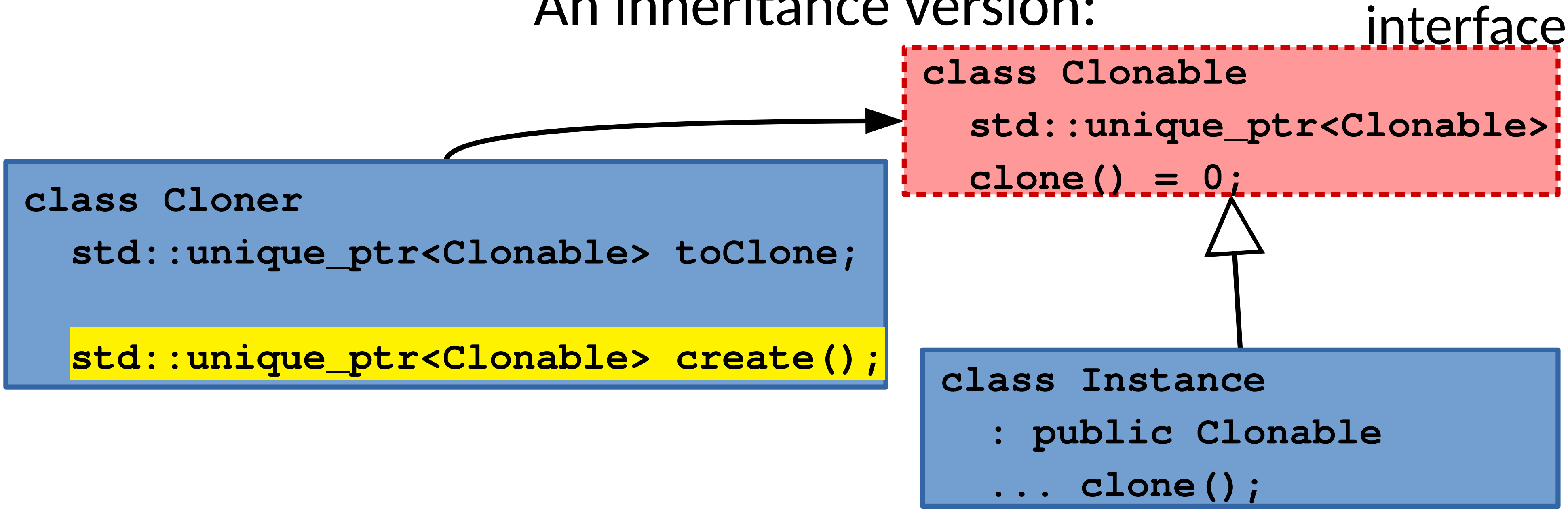
An inheritance version:



e.g. Creational Pattern: Prototype

- Goal: Create new objects based on a configuration.

An inheritance version:



e.g. Creational Pattern: Prototype

- Goal: Create new objects based on a configuration.

An inheritance version:

interface

```
class Clonable
  std::unique_ptr<Clonable>
  clone() = 0;
```

```
class Cloner
  std::unique_ptr<Clonable> toClone;

  std::unique_ptr<Clonable> create();
```

```
class Instance
  : public Clonable
          e();
```

What risks are there?
Can you see better ways?

e.g. Creational Pattern: Prototype

- Benefits:
  - User defined objects become easier

e.g. Creational Pattern: Prototype

- Benefits:
  - User defined objects become easier

- Downsides:
  - Managing the cloning becomes critical

e.g. Creational Pattern: Prototype

- Benefits:
  - User defined objects become easier
- Downsides:
  - Managing the cloning becomes critical
  - Inheritance based approaches require clone implementations

e.g. Creational Pattern: Prototype

- Benefits:
  - User defined objects become easier

- Downsides:
  - Managing the cloning becomes critical
  - Inheritance based approaches require clone implementations
  - Deep copy vs shallow copy?

Problem: Adding Behavior/State

- How do you normally add behaviors or state to an object?

Problem: Adding Behavior/State

- How do you normally add behaviors or state to an object?

```
class ByteStream {
public:
  Byte getNextByte();
};
```

Problem: Adding Behavior/State

- How do you normally add behaviors or state to an object?

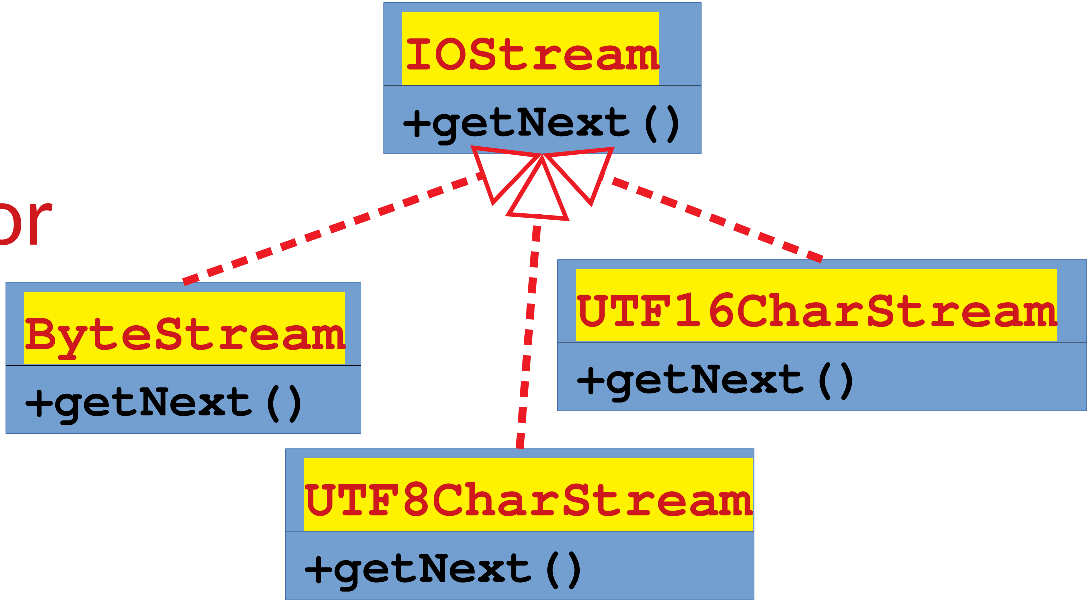
```
class ByteStream {
public:
  Byte getNextByte();
  UT8Char getNextUTF8Char();
  UTF16Char getNextUTF16Char();
};
```

Problem: Adding Behavior/State

- How do you normally add behaviors or state to an object?

```
class ByteStream {
public:
   Byte getNextByte();
   UT8Char getNextUTF8Char();
   UTF16Char getNextUTF16Char();
};
```

or

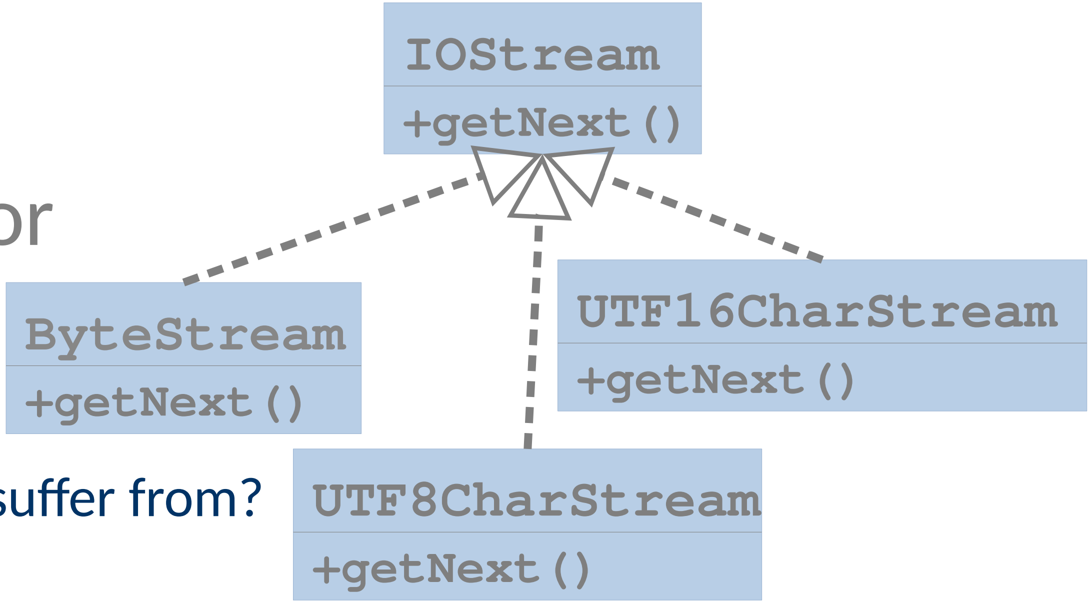


Problem: Adding Behavior/State

- How do you normally add behaviors or state to an object?

```
class ByteStream {
public:
  Byte getNextByte();
  UT8Char getNextUTF8Char();
  UTF16Char getNextUTF16Char();
};
```

or



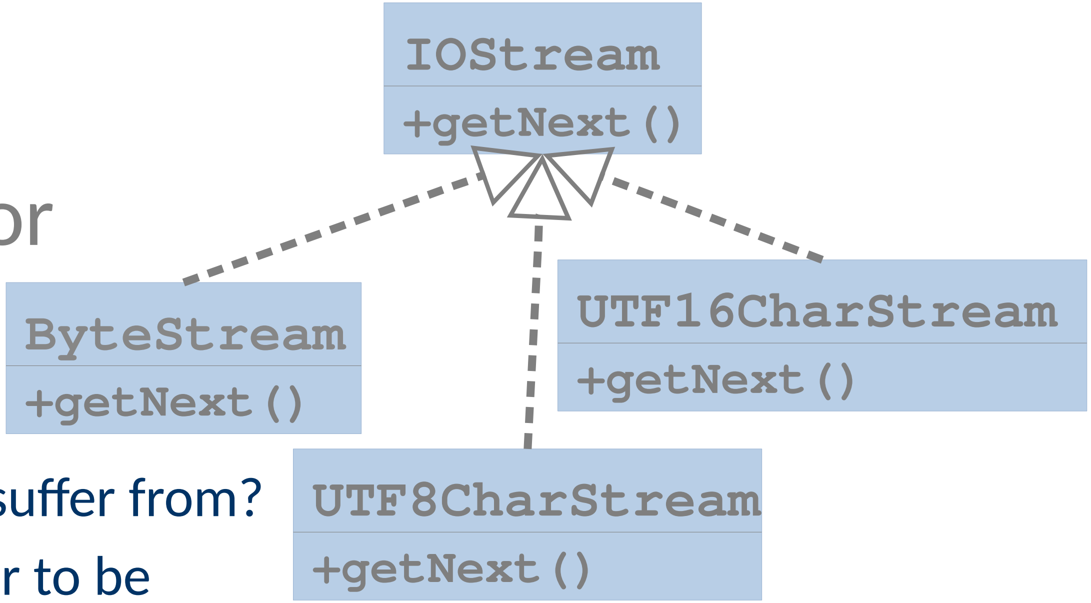
  – What issues do these solutions suffer from?

Problem: Adding Behavior/State

- How do you normally add behaviors or state to an object?

```
class ByteStream {
public:
   Byte getNextByte();
   UT8Char getNextUTF8Char();
   UTF16Char getNextUTF16Char();
};
```

or



  - What issues do these solutions suffer from?
  - What if you wanted the behavior to be dynamic?

Problem: Adding Behavior/State

- Let us consider another example:

```
class VideoStream {
public:
  Frame getNextFrame();
};
```

Problem: Adding Behavior/State

- Let us consider another example:
  - What if we want the ability to scale/resize frames?

```
class VideoStream {
public:
  Frame getNextFrame();
};
```

Problem: Adding Behavior/State

- Let us consider another example:
  - What if we want the ability to scale/resize frames?
  - What if we want to add a banner ad?

```
class VideoStream {
public:
  Frame getNextFrame();
};
```

Problem: Adding Behavior/State

- Let us consider another example:
  - What if we want the ability to scale/resize frames?
  - What if we want to add a banner ad?
  - What if we want to log slow to acquire frames?

```
class VideoStream {
public:
  Frame getNextFrame();
};
```

Problem: Adding Behavior/State

- Let us consider another example:
  - What if we want the ability to scale/resize frames?
  - What if we want to add a banner ad?
  - What if we want to log slow to acquire frames?
  - And the combined behavior is chosen at runtime.

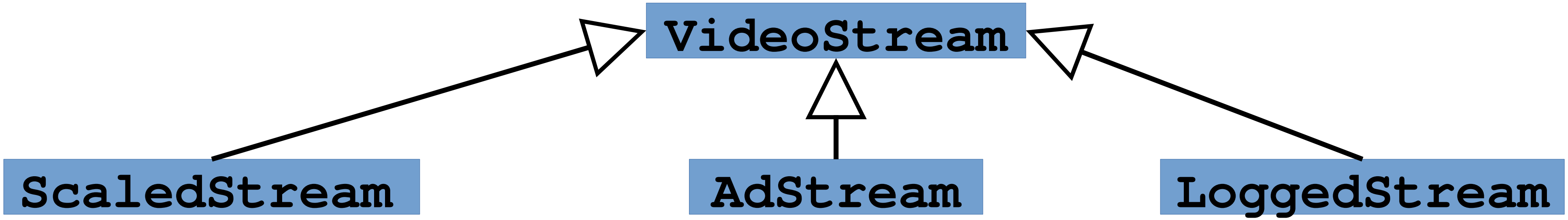
```
class VideoStream {
public:
  Frame getNextFrame();
};
```

Problem: Adding Behavior/State

- What if we use inheritance?

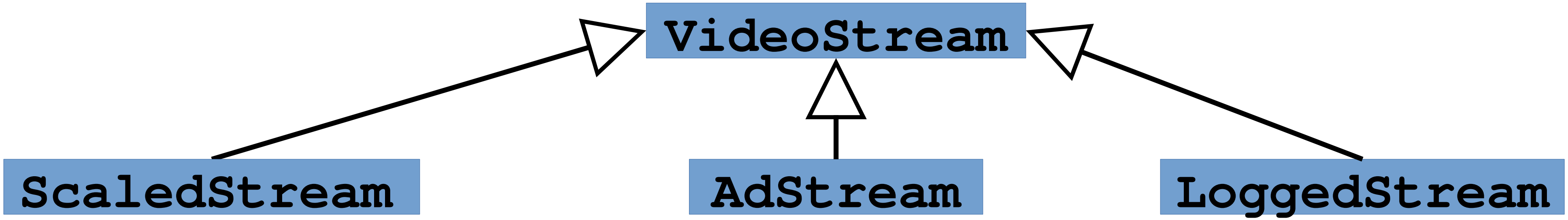
Problem: Adding Behavior/State

- What if we use inheritance?



Problem: Adding Behavior/State

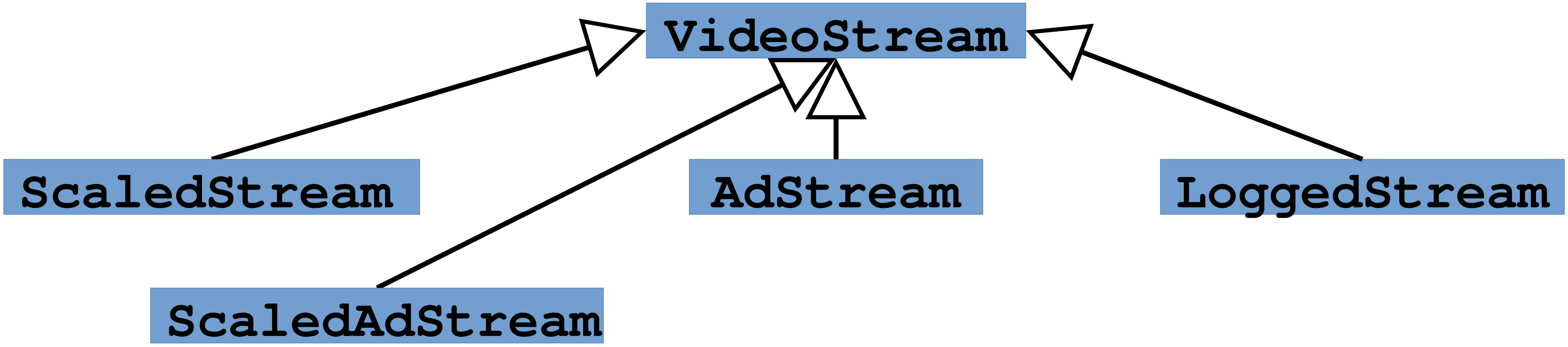
- What if we use inheritance?



Is this sufficient?

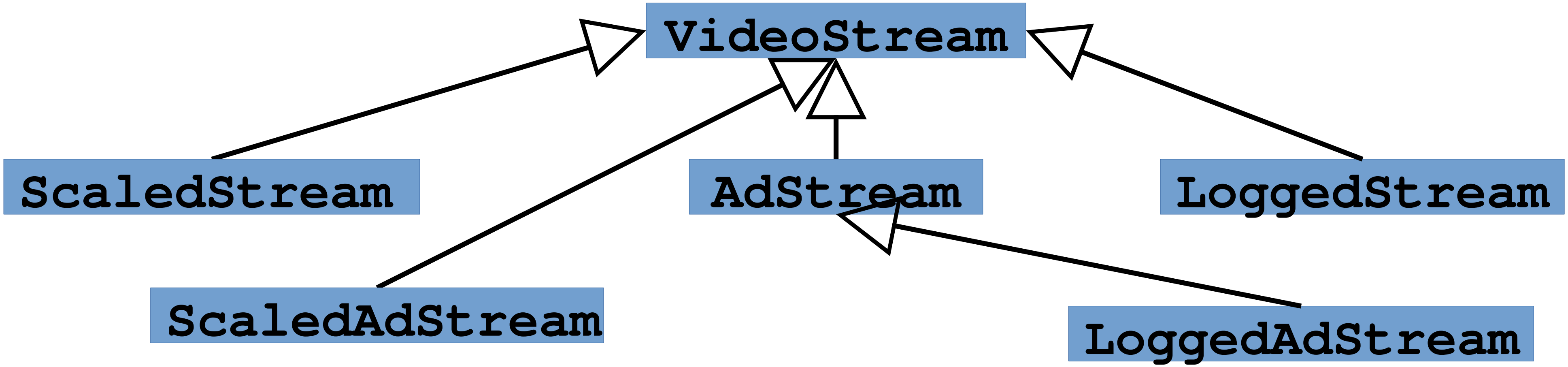
Problem: Adding Behavior/State

- What if we use inheritance?



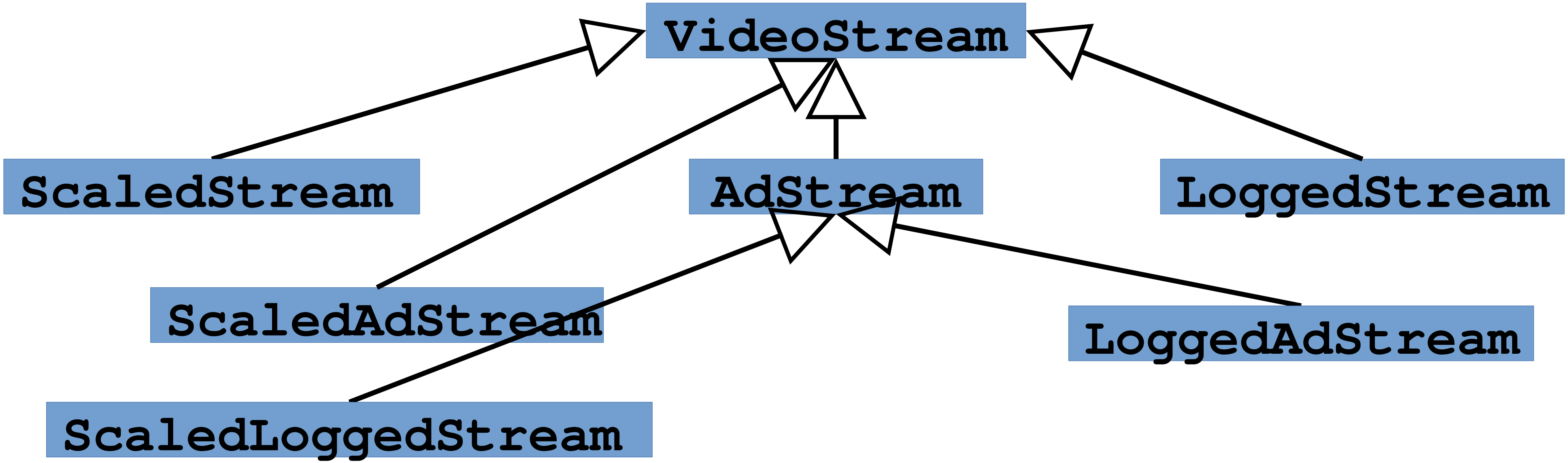
Problem: Adding Behavior/State

- What if we use inheritance?



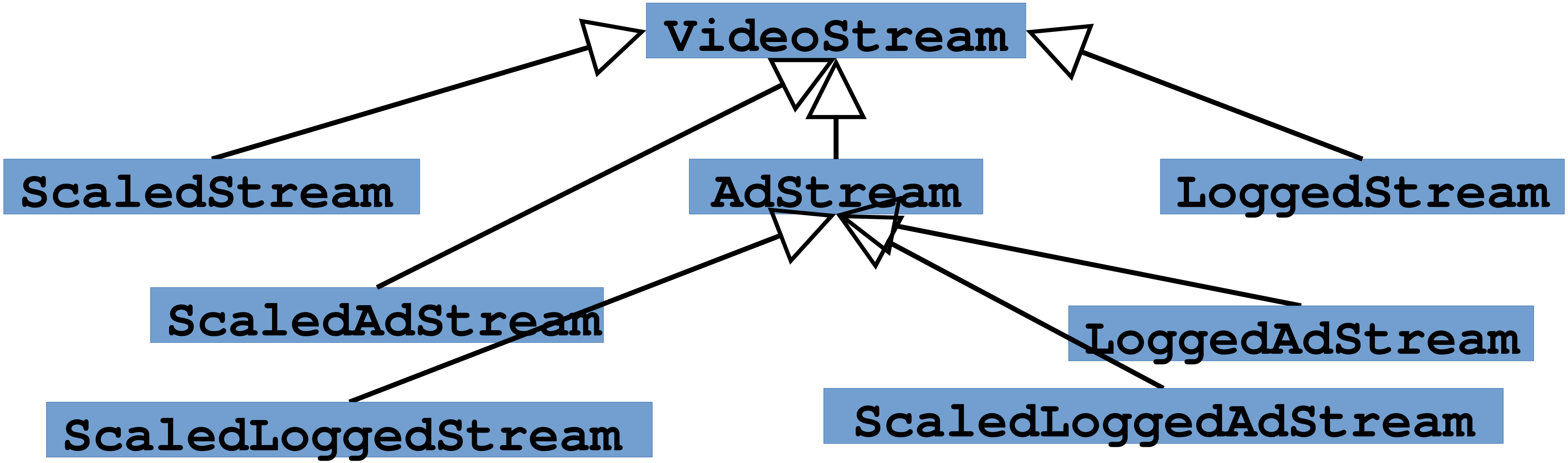
Problem: Adding Behavior/State

- What if we use inheritance?



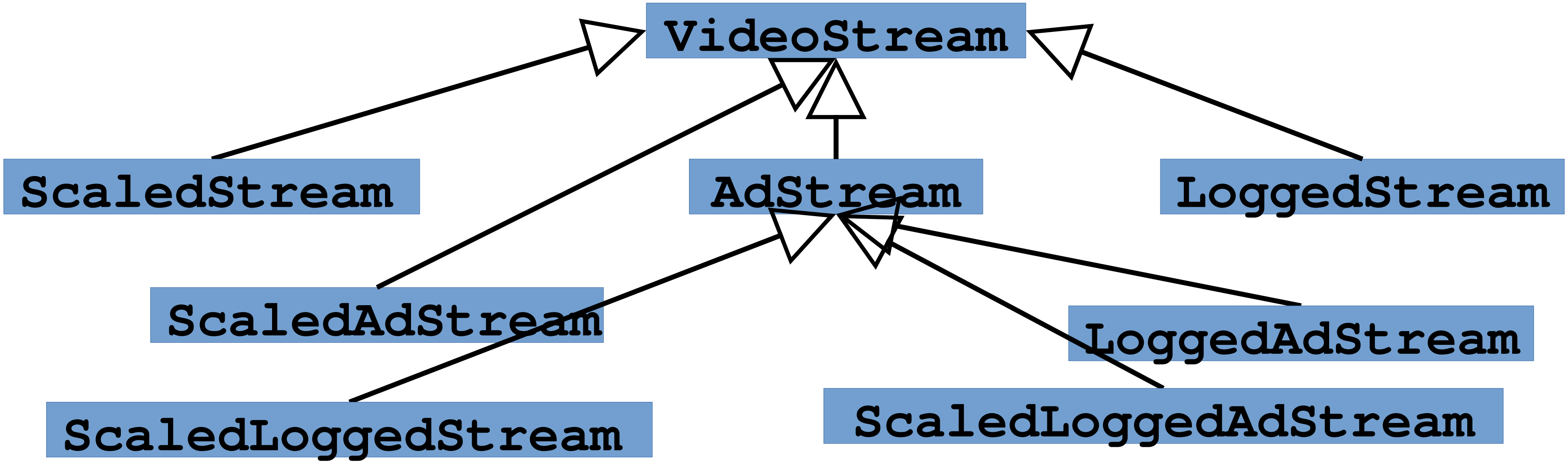
Problem: Adding Behavior/State

- What if we use inheritance?



Problem: Adding Behavior/State

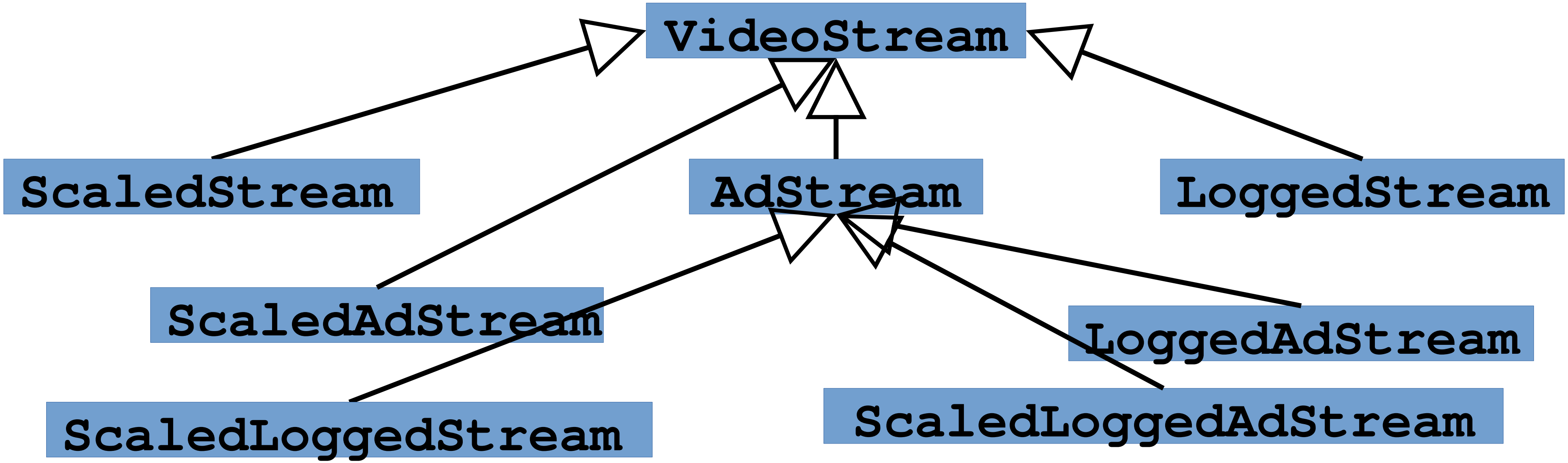
- What if we use inheritance?



- For *k* additions: 2^k classes

Problem: Adding Behavior/State

- What if we use inheritance?



- For *k* additions: 2^k classes
  - And you may not know which even make sense right away…

Problem: Adding Behavior/State

- Goal:
  - Decouple the addition of behavior from the `VideoStream` class

Problem: Adding Behavior/State

- Goal:
  - Decouple the addition of behavior from the **`VideoStream`** class
  - But inheritance of implementation is strongly coupling!

Problem: Adding Behavior/State

- Goal:
  - Decouple the addition of behavior from the **`VideoStream`** class
  - But inheritance of implementation is strongly coupling!

Problem: Adding Behavior/State

- Goal:
  - Decouple the addition of behavior from the `VideoStream` class
  - But inheritance of implementation is strongly coupling!

  - So what can we do instead?

Let’s work through it on the board…

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object

```
class FrameProvider
  getNextFrame() = 0;
```

interface

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object

```
class FrameProvider
  getNextFrame() = 0;
```

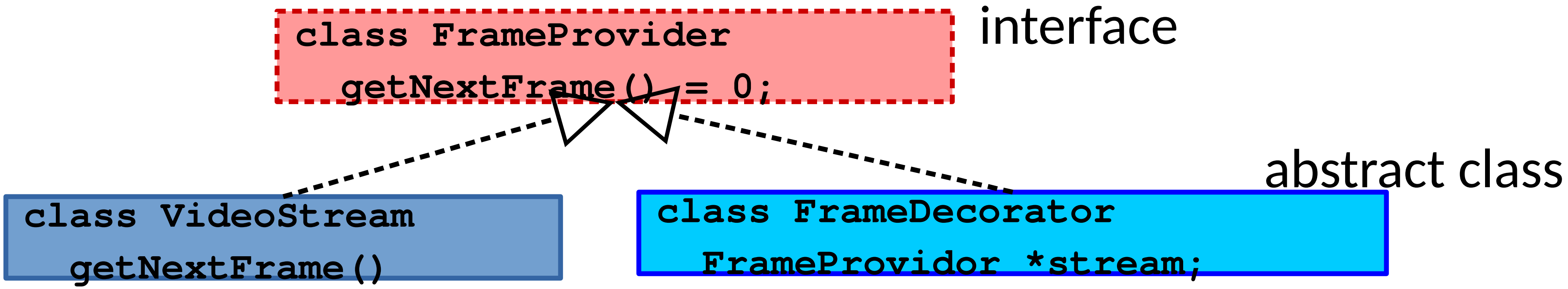
interface

```
class VideoStream
  getNextFrame()
```

The core/simplest behavior will always be necessary

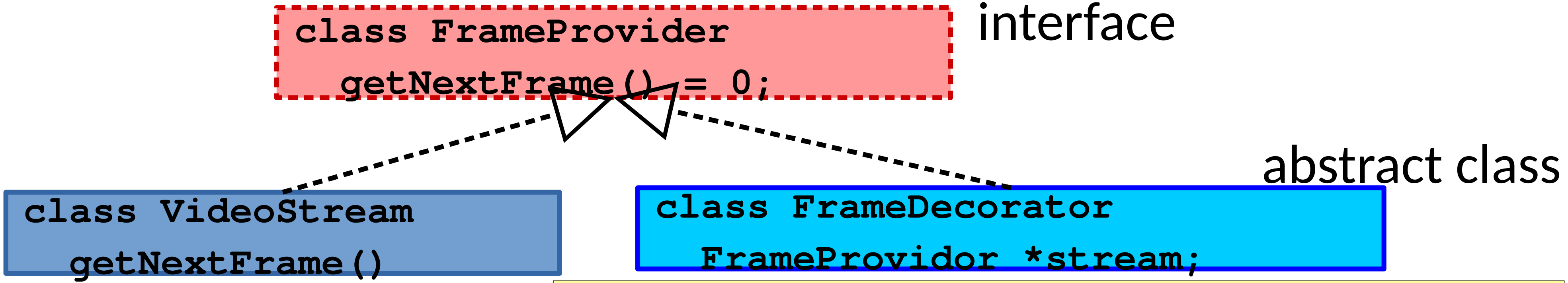
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



e.g. Structural Pattern: Decorator

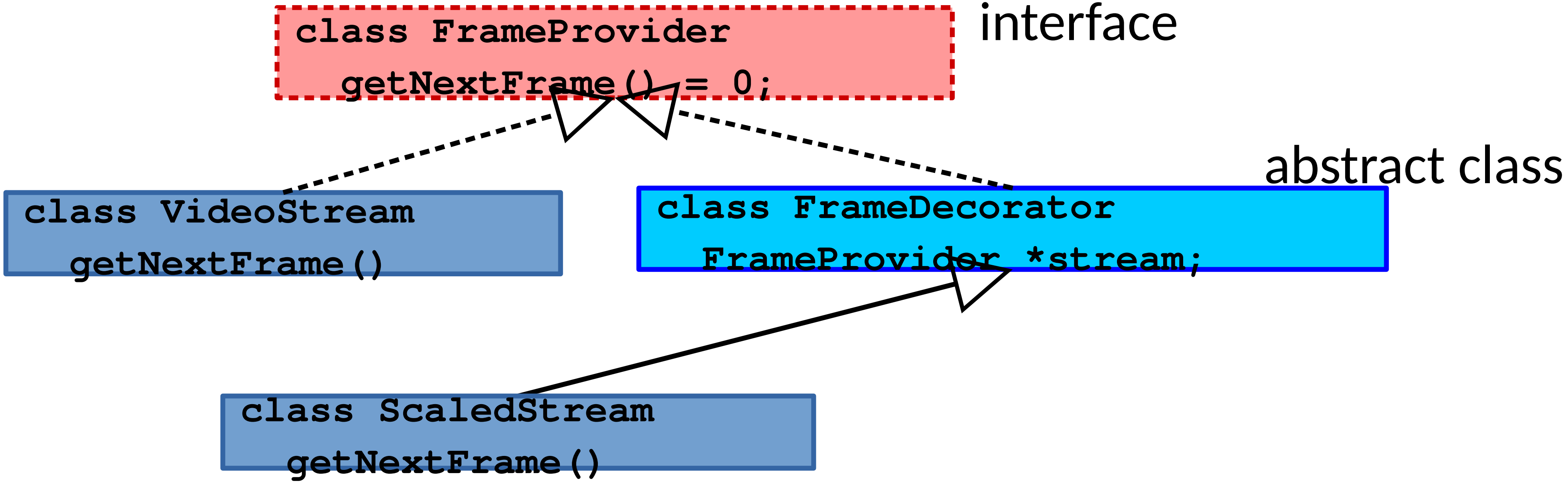
- Goal: Flexibly add state/behavior to an object



This only exists to provide the **`*stream`** to concrete decorations!

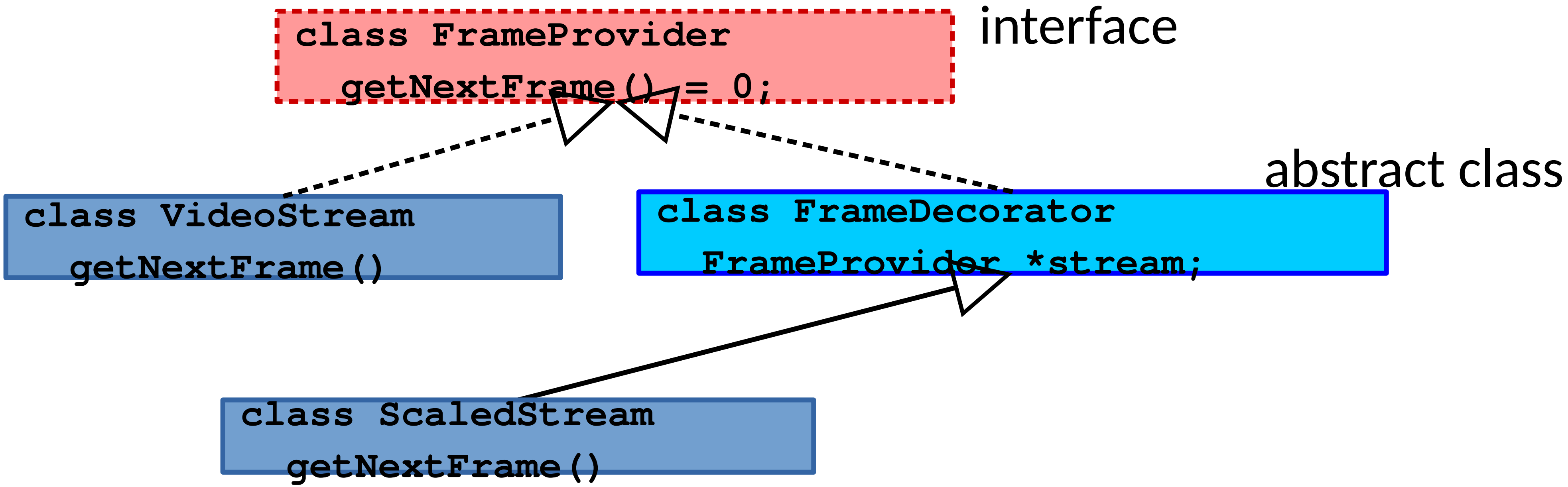
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



e.g. Structural Pattern: Decorator

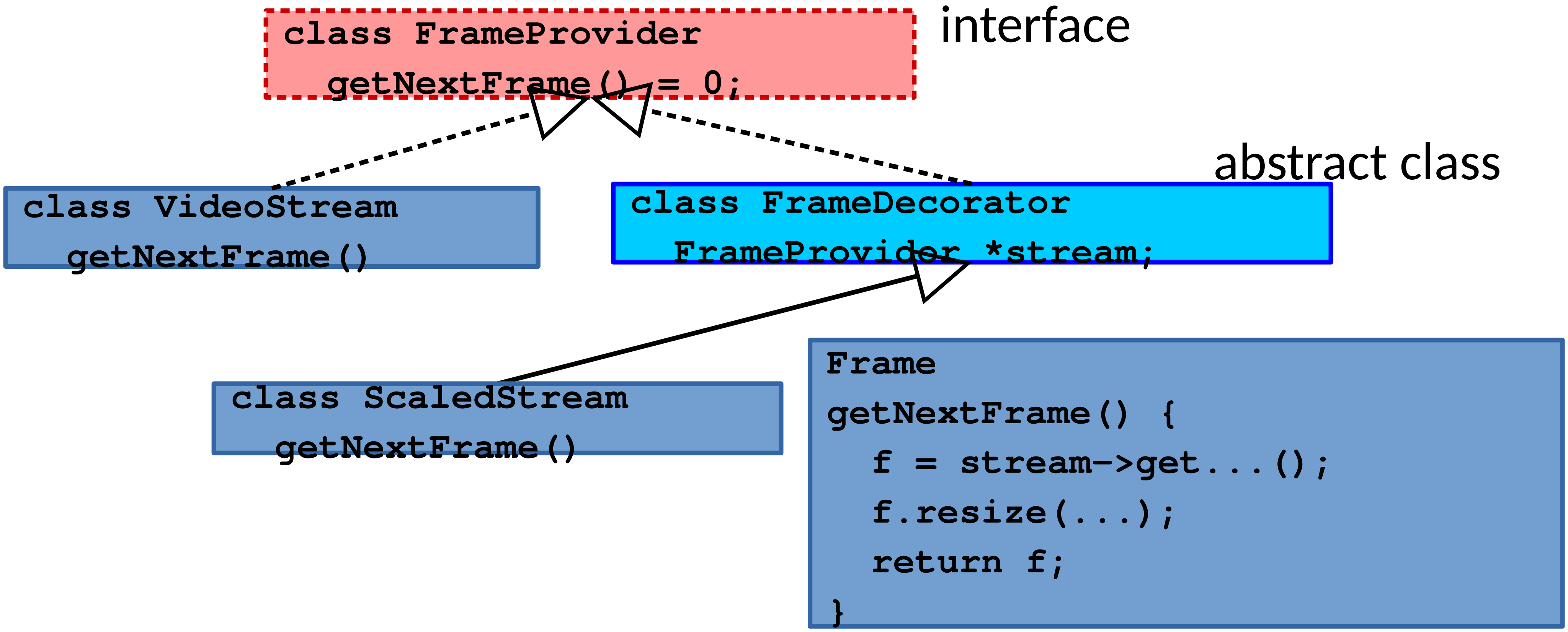
- Goal: Flexibly add state/behavior to an object



What does its **getNextFrame()** look like?

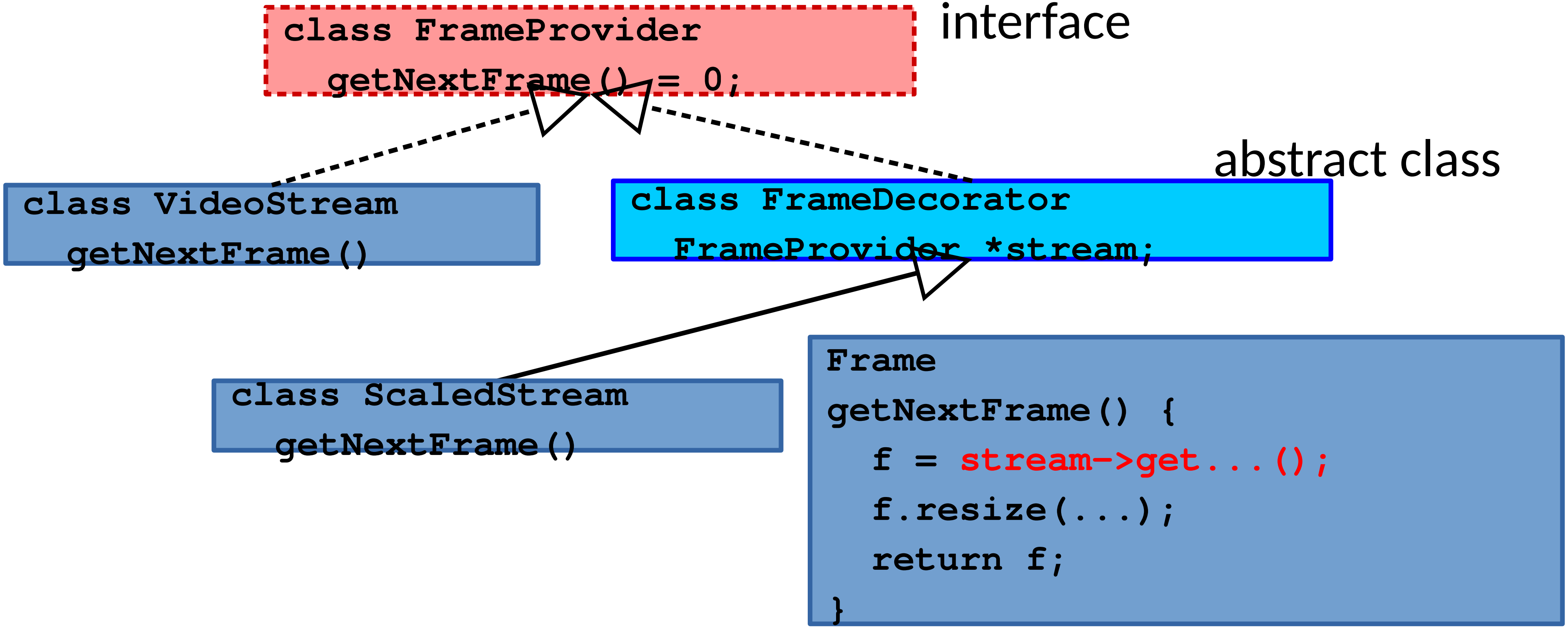
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



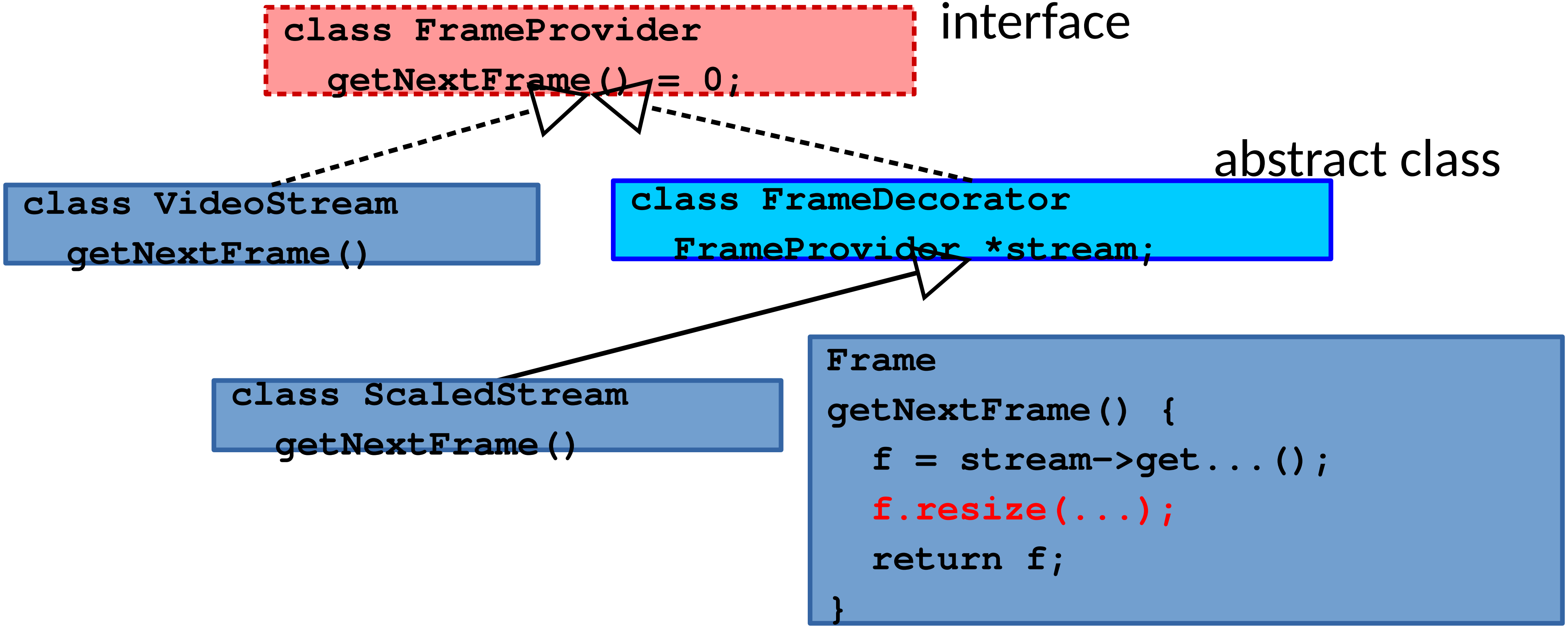
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



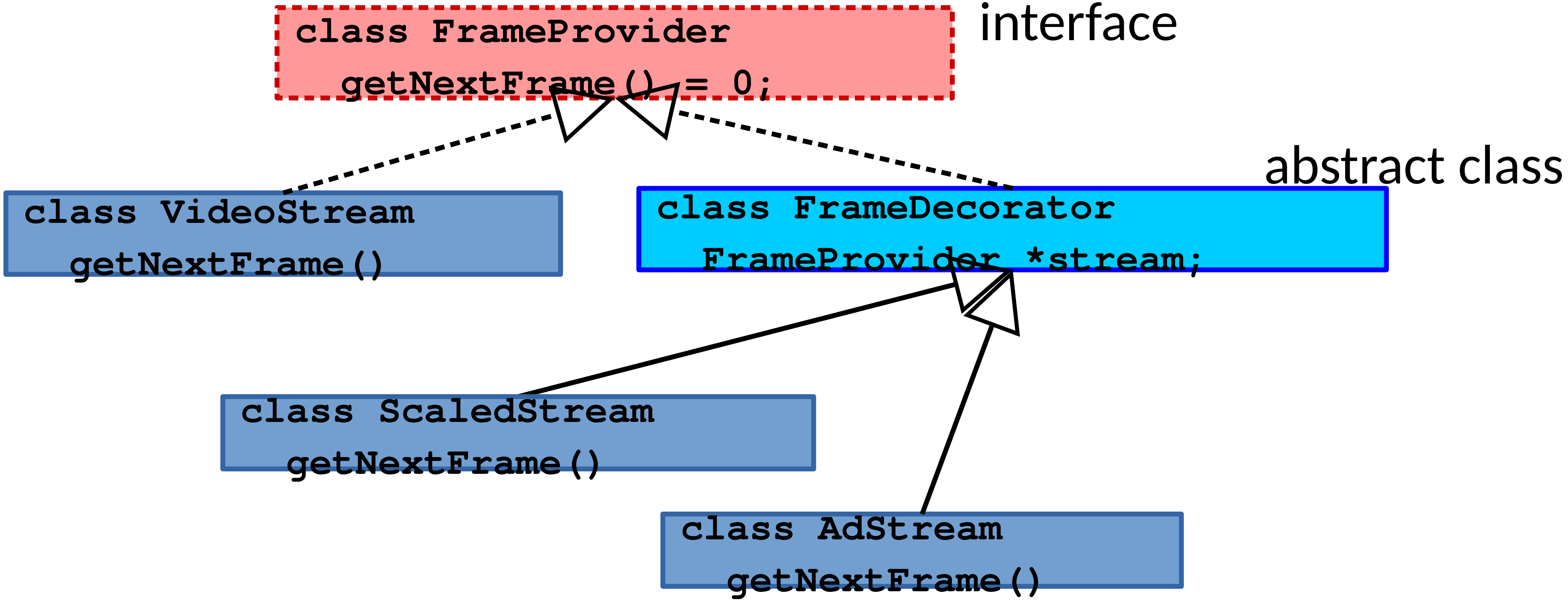
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



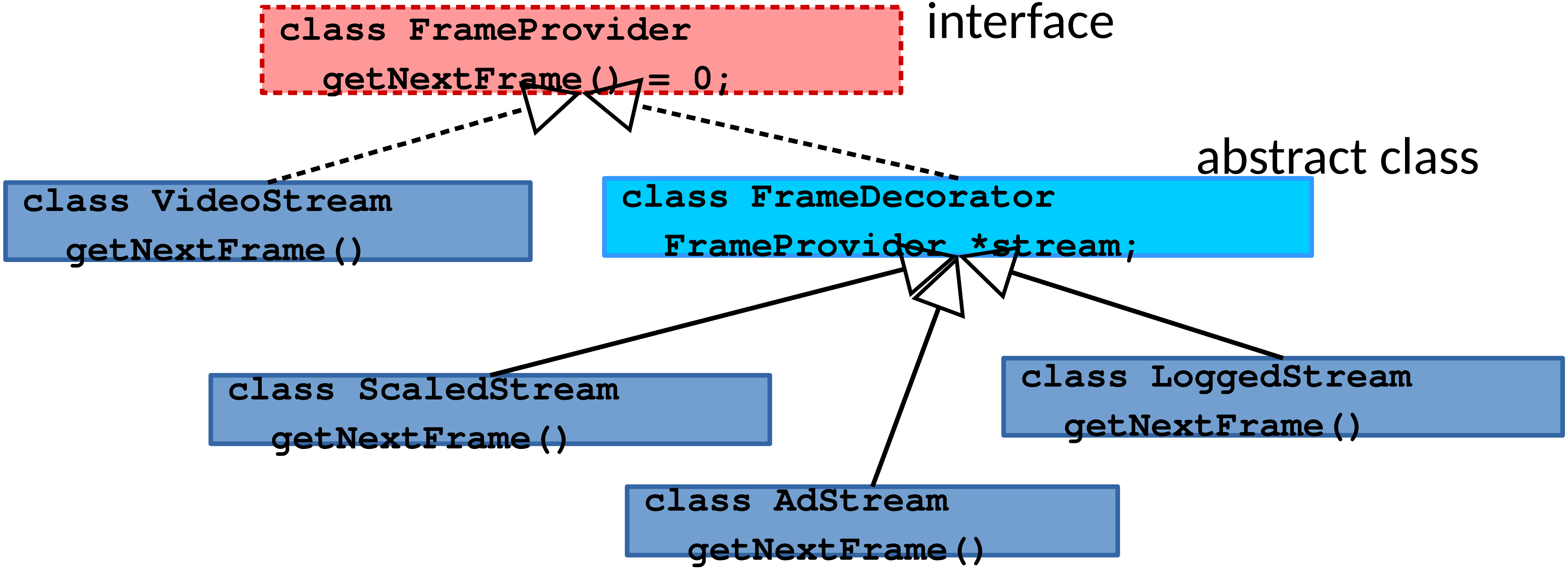
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



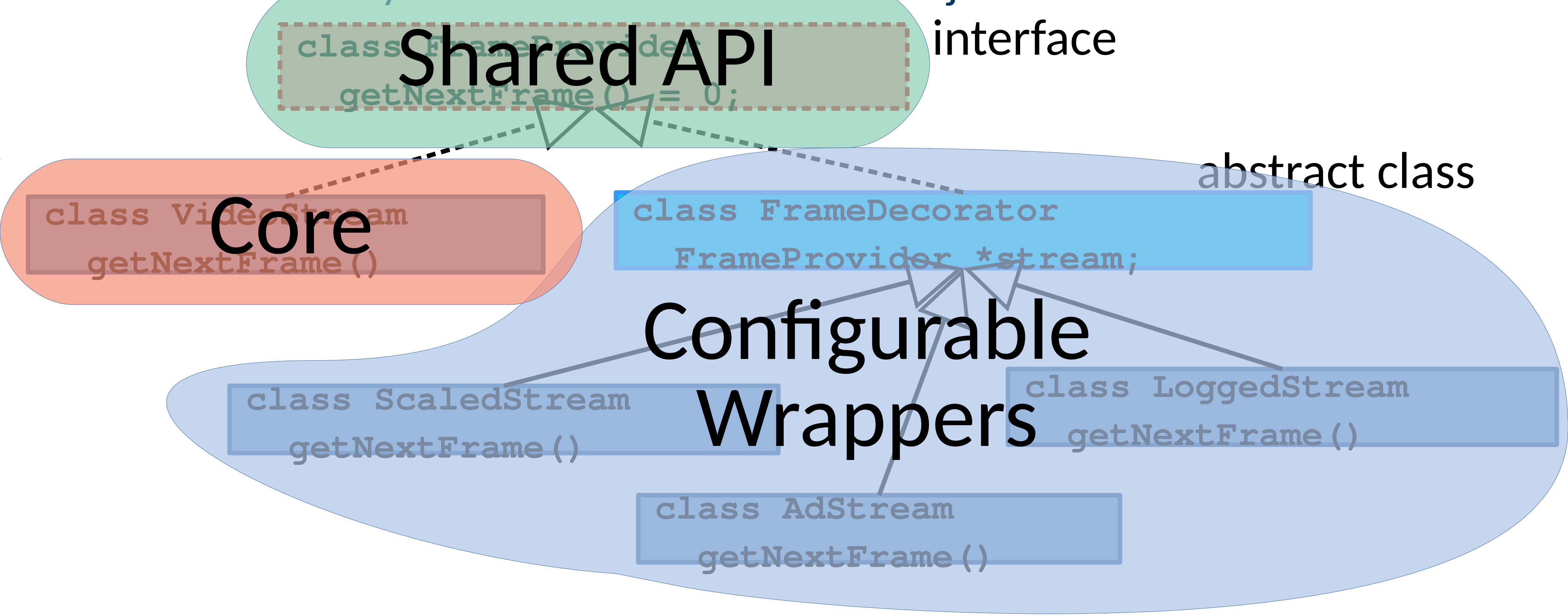
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object



e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object
- Also called **`Wrapper`** (for now obvious reasons)

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object
- Also called `Wrapper` (for now obvious reasons)
- Benefits

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object
- Also called `Wrapper` (for now obvious reasons)
- Benefits
  - Avoid class explosion

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object
- Also called `Wrapper` (for now obvious reasons)
- Benefits
  - Avoid class explosion
  - Works when inheritance on core is prohibited

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object
- Also called `Wrapper` (for now obvious reasons)
- Benefits
  - Avoid class explosion
  - Works when inheritance on core is prohibited
  - Enables dynamically adding/removing behavior!

e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object
- Also called `Wrapper` (for now obvious reasons)
- Benefits
  - Avoid class explosion
  - Works when inheritance on core is prohibited
  - Enables dynamically adding/removing behavior!
- Can the added & original behaviors change independently?

e.g. Structural Pattern: Decorator

- Downsides?

e.g. Structural Pattern: Decorator

- Downsides?
  - Address no longer gives object identity
    - How might you resolve this?

e.g. Structural Pattern: Decorator

- Downsides?
  - Address no longer gives object identity
    - How might you resolve this?
  - The indirection is itself a form of complexity
    - Debugging why one link in a chain fails is more complex

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?

```
...
auto result = foo(x, y, z);
...
```

What are the forms of coupling that arise?

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?

```
...
auto result = foo(x, y, z);
...
```

What are the forms of coupling that arise?

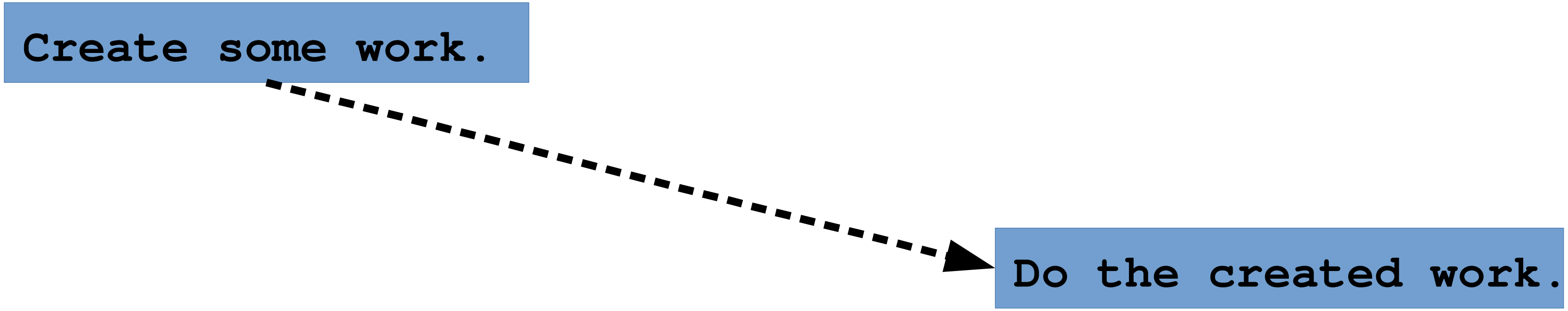
Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.

```
...
auto result = foo(x, y, z);
...
```

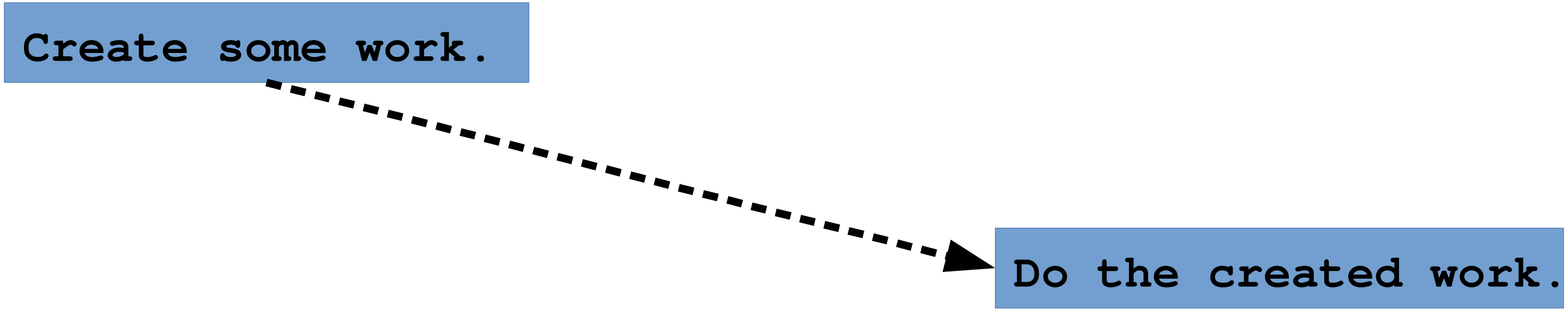
Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.



Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.



  - What interface captures this?

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.

```
[redacted]
auto result = [redacted]([redacted]);
[redacted]
```

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.

```
auto result = worker.doWork();
```

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.

```
auto result = worker.doWork();
```

```
class Work {
  // Information about work
  // ...
  Result doWork() {...}
};
```

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.

```
auto result = worker.doWork();
```

```
class Work {
  // Information about work
  // ...
  Result doWork() {...}
};
```

```
class OtherKindOfWork {
  Result doWork() {...}
};
```

Problem: Separate Caller & Callee

- What if we want to fully decouple actions to be taken from their call sites?
  - Sometimes you must execute an action without any knowledge of what that action is.

```
auto result = worker.doWork();
```

```
class Work {
  virtual Result doWork() = 0;
};
```

```
class WorkKind1 : public Work {
  Result doWork() override{...}
};
```

```
class WorkKind2 : public Work {
  Result doWork() override{...}
};
```

e.g. Behavioral Pattern: Command

```
class Command {
public:
  virtual void execute() = 0;
};
```

e.g. Behavioral Pattern: Command

- This is the ***command pattern***

```
class Command {
public:
  virtual void execute() = 0;
};
```

e.g. Behavioral Pattern: Command

- This is the ***command pattern***
- It is nothing more than an object oriented callback

```
class Command {
public:
  virtual void execute() = 0;
};
```

e.g. Behavioral Pattern: Command

- This is the ***command pattern***
- It is nothing more than an object oriented callback

```
class Command {
public:
  virtual void execute() = 0;
};
```

Why not just use a lambda?

The Command Pattern

- Benefits
  - **Decouples a request / behavior from the invoker**

The Command Pattern

- Benefits
  - Decouples a request / behavior from the invoker
  - Invoker decides when to invoke without caring what

The Command Pattern

- Benefits
  - Decouples a request / behavior from the invoker
  - Invoker decides when to invoke without caring what
  - Parametrizable via constructor

The Command Pattern

- Benefits
  - Decouples a request / behavior from the invoker
  - Invoker decides when to invoke without caring what
  - Parametrizable via constructor

```
...
auto result = foo(x, y, z);
...
```

The Command Pattern

- Benefits
  - Decouples a request / behavior from the invoker
  - Invoker decides when to invoke without caring what
  - Parametrizable via constructor

```
...
auto result = foo(x, y, z);
...
```

```
...
auto command = FooCommand(x, y, z);
...
```

```
command.execute();
```

The Command Pattern

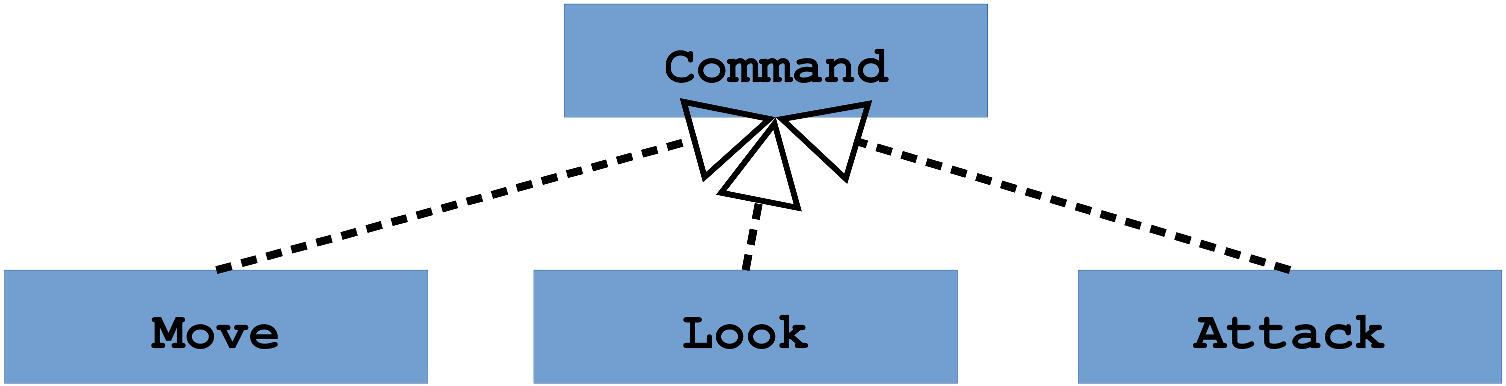
- Benefits
  - Decouples a request / behavior from the invoker
  - Invoker decides when to invoke without caring what
  - Parametrizable via constructor
  - Sequences of commands can be easily batched

The Command Pattern

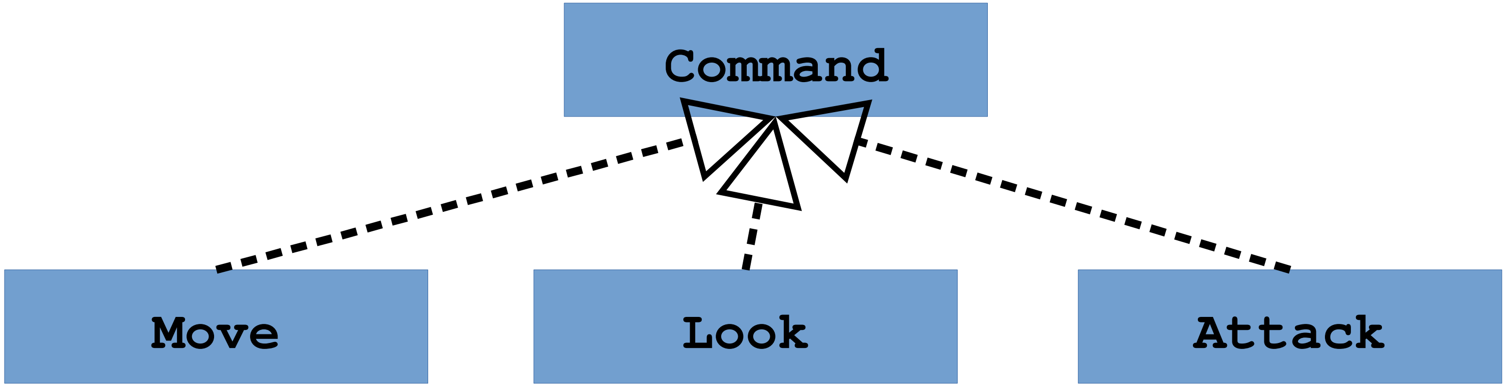
- Benefits
  - Decouples a request / behavior from the invoker
  - Invoker decides when to invoke without caring what
  - Parametrizable via constructor
  - Sequences of commands can be easily batched

How can this be used in the project?

The Command Pattern



The Command Pattern



Is only one Move necessary?

The Command Pattern

- Issues
  - How much state should it hold? (Passed to constructor vs passed to execute)

The Command Pattern

- Issues
  - How much state should it hold?
  - Does it perform undo/redo?

The Command Pattern

- Issues
  - How much state should it hold?
  - Does it perform undo/redo?
  - Can you batch commands?

The Command Pattern

- Issues
  - How much state should it hold?
  - Does it perform undo/redo?
  - Can you batch commands?
  - How does temporal decoupling affect operation logic?

The Big Picture

- There is nothing *special* about design patterns!

The Big Picture

- There is nothing *special* about design patterns!
  - What is the API you want?

```
...
auto result = foo(x, y, z);
...
```

VS

```
auto result = worker.doWork();
```

The Big Picture

- There is nothing *special* about design patterns!
  - What is the API you want?
  - What do you know, what do you need to know, & when?

I know `x, y, z` here

I want to know `x, y, z`
but hide them here.

```
auto result = worker.doWork();
```

The Big Picture

- There is nothing *special* about design patterns!
  - What is the API you want?
  - What do you know, what do you need to know, & when?
  - How can you hide design decisions to get the API you want?

I know `x, y, z` here

```
class Command {
public:
  virtual void doWork() = 0;
};
```

I want to know `x, y, z`
but hide them here.

```
auto result = worker.doWork();
```

Design Patterns

- They provide a common language for design decisions

Design Patterns

- They provide a common language for design decisions
- They illustrate common trade offs & how to solve them

Design Patterns

- They provide a common language for design decisions
- They illustrate common trade offs & how to solve them
- I heartily recommend learning State, Strategy, & Visitor as well
  - We will explore these a little in class.