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
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
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**
    - 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 don’t 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 don’t 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 don’t 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 don’t 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
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 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 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
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.
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?
- What are the coupled constraints in this approach?
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 for very flexible objects created by the user?
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*
  - ‘user created’
Problem: Flexibly creating objects

- Sometimes you want to create new objects patterned off another
  - First instance might be costly to build
  - ‘user created’
Problem: Flexibly creating objects

• Sometimes you want to create new objects patterned off another
  – First instance might be costly to build
  – ‘ ’ user created

• 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:

```cpp
class Clonable
{
    std::unique_ptr<Clonable> clone() = 0;
};
```
e.g. **Creational Pattern: Prototype**

- **Goal:** Create new objects based on a configuration.

**An inheritance version:**

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

class Instance
: public Clonable
... clone();
```

e.g. Creational Pattern: Prototype

- Goal: Create new objects based on a configuration.

An inheritance version:

```cpp
class Clonable

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

class Cloner

std::unique_ptr<Clonable> toClone;

std::unique_ptr<Clonable> create();

class Instance

: public Clonable
...
... clone();
```
e.g. **Creational Pattern: Prototype**

- **Goal:** Create new objects based on a configuration.

An inheritance version:

```cpp
class Clonable

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

```cpp
class Cloner

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

```cpp
class Instance

: public Clonable

clone();
```

**What risks are there? Are there 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

What better ways can you imagine?
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

- Sometimes you want to add behavior / state to an object.
Problem: Adding Behavior/State

- Sometimes you want to add behavior / state to an object.
  - How would you do it?
Problem: Adding Behavior/State

- Sometimes you want to add behavior / state to an object.
  - How would you do it?
  - At runtime?
Sometimes you want to add behavior / state to an object.
- How would you do it?
- At runtime?

```cpp
class VideoStream {
public:
    Frame getNextFrame();
};
```
Problem: Adding Behavior/State

- Sometimes you want to add behavior / state to an object.
  - How would you do it?
  - At runtime?

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

- What if we want the ability to scale/resize frames?
Problem: Adding Behavior/State

- Sometimes you want to add behavior / state to an object.
  - How would you do it?
  - At runtime?

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

- What if we want the ability to scale/resize frames?
- What if we want to add a banner ad?
Problem: Adding Behavior/State

Sometimes you want to add behavior / state to an object.
  – How would you do it?
  – At runtime?

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

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?
Problem: Adding Behavior/State

- What if we use inheritance?
Problem: Adding Behavior/State

- What if we use inheritance?

Diagram:

- VideoStream
  - ScaledStream
  - AdStream
  - LoggedStream
Problem: Adding Behavior/State

- What if we use inheritance?

![Diagram](source_url)
Problem: Adding Behavior/State

- What if we use inheritance?
Problem: Adding Behavior/State

- What if we use inheritance?

```
VideoStream
  |   |
  |   |
ScaledStream     AdStream     LoggedStream
  |   |           |   |
  |   |           |   |
ScaledAdStream   LoggedAdStream
```
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 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

```java
class FrameProvider

interface

g getNextFrame() = 0;
```
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object

```java
interface FrameProvider
{
    getNextFrame() = 0;
}

class VideoStream
{
    getNextFrame()
}
```
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object

```cpp
class FrameProvider
{
    public:
        virtual ~FrameProvider() = 0;
        virtual FrameProvider* getNextFrame() = 0;
};

class VideoStream
{
    FrameProvider* stream;
    public:
        ~VideoStream() override{ delete stream; }
        Resource* getNextFrame() override{ return stream->getNextFrame(); }
};

class FrameDecorator : public FrameProvider
{
    private:
        FrameProvider* stream;
    public:
        FrameDecorator(FrameProvider* _stream) : stream(_stream) { }
        FrameProvider* getNextFrame() override{ return stream; }
};
```
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object

```cpp
class FrameProvider {
    public:
        virtual ~FrameProvider() {}
        virtual ~FrameProvider() {}
    public:
        virtual void getNextFrame() = 0;
};

class VideoStream {
    public:
        ~VideoStream() {}
    public:
        virtual void getNextFrame() {}
};

class FrameDecorator {
    private:
        FrameProvider *stream;
    public:
        FrameDecorator() : stream(nullptr) {
        }
        ~FrameDecorator() {}
        void getNextFrame() {

            // This only exists to provide the *stream
            // to concrete decorations!
        }
    }
```
e.g. **Structural Pattern: Decorator**

- **Goal:** Flexibly add state/behavior to an object

```cpp
class FrameProvider
{
    getNextFrame() = 0;
}

class VideoStream
{
    getNextFrame();
}

class FrameDecorator
{
    FrameProvider *stream;
}

class ScaledStream
{
    getNextFrame();
}
```
e.g. **Structural Pattern: Decorator**

- **Goal:** Flexibly add state/behavior to an object

```cpp
class FrameProvider
{
    virtual getNextFrame() = 0;
};

class VideoStream
{
    virtual getNextFrame() = 0;
};

class FrameDecorator
{
    FrameProvider *stream;
};

class ScaledStream
{
    virtual getNextFrame() = 0;
};

What does its `getNextFrame()` look like?
```
e.g. Structural Pattern: Decorator

- Goal: Flexibly add state/behavior to an object

```cpp
class FrameProvider
{
    virtual Frame getNextFrame() = 0;
};

class VideoStream
{
    Frame getNextFrame();
};

class FrameDecorator
{
    FrameProvider *stream;

    Frame getNextFrame()
    {
        Frame f = stream->getNextFrame();
        f.resize(...);
        return f;
    }
};

class ScaledStream
{
    Frame getNextFrame();
};
```
e.g. **Structural Pattern: Decorator**

- **Goal:** Flexibly add state/behavior to an object

```cpp
class VideoStream {
    public:
        virtual ~VideoStream(){ };
        virtual frame getNextFrame() = 0;
}
class FrameProvider {
    public:
        virtual frame getNextFrame() = 0;
}
class FrameDecorator {
    private:
        FrameProvider *stream;
    public:
        frame getNextFrame() {
            frame f = stream->getNextFrame();
            f.resize(...);
            return f;
        }
}
class ScaledStream : public VideoStream {
    public:
        frame getNextFrame() {
            frame f = stream->getNextFrame();
            f.resize(...);
            return f;
        }
}
```
e.g. **Structural Pattern: Decorator**

- **Goal:** Flexibly add state/behavior to an object

```cpp
class VideoStream {
    getNextFrame();
};
```

interface

```cpp
abstract class FrameProvider {
    getNextFrame();
};
```

```cpp
class FrameDecorator : public FrameProvider {
    FrameProvider *stream;
};
```

```cpp
class ScaledStream {
    getNextFrame();
};
```

```cpp
class Frame {
    getNextFrame() {
        f = stream->get...();
        f.resize(...);
        return f;
    }
};
```
e.g. **Structural Pattern: Decorator**

- **Goal:** Flexibly add state/behavior to an object

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

class VideoStream
    getNextFrame()

class FrameDecorator
    FrameProvider *stream;

class ScaledStream
    getNextFrame()

class AdStream
    getNextFrame()
```
e.g. **Structural Pattern: Decorator**

- **Goal:** Flexibly add state/behavior to an object

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

class VideoStream
    getNextFrame()

class FrameDecorator
    FrameProvider *stream;

class ScaledStream
    getNextFrame()

class AdStream
    getNextFrame()

class LoggedStream
    getNextFrame()
```

```java
interface abstract class
```
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 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 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 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: What if we want to fully decouple actions to be taken from their call sites?
e.g. **Behavioral Pattern: Command**

- Sometimes you must execute an action without any knowledge of what that action is.
e.g. **Behavioral Pattern: Command**

- Sometimes you must execute an action without any knowledge of what that action is.
- What is an interface that encapsulates this?
e.g. **Behavioral Pattern: Command**

- Sometimes you must execute an action without any knowledge of what that action is.
- What is an interface that encapsulates this?

```cpp
class Command {
public:
    virtual void execute() = 0;
};
```
e.g. Behavioral Pattern: Command

- Sometimes you must execute an action without any knowledge of what that action is.
- What is an interface that encapsulates this?

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

- This is the *command pattern*
e.g. Behavioral Pattern: Command

- Sometimes you must execute an action without any knowledge of what that action is.
- What is an interface that encapsulates this?

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

- This is the command pattern
- It is nothing more than an object oriented callback
e.g. Behavioral Pattern: Command

- Sometimes you must execute an action without any knowledge of what that action is.
- What is an interface that encapsulates this?

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

- This is the *command pattern*
- It is nothing more... 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
  - 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

- Command
  - Move
  - Look
  - Attack
The Command Pattern

Is only one Move necessary?
The Command Pattern

- Issues
  - How much state should it hold?
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
Design Patterns

- The provide a common language for design decisions
Design Patterns

- The provide a common language for design decisions
- I heartily recommend learning State, Strategy, & Visitor as well