Software Design Foundations

Nick Sumner
wsumner@sfu.ca
Why care about software design?

- Software Design
Why care about software design?

- **Software Design**
  - The components into which a problem is broken down

Diagram:

- Input
- Audio
- Client Logic
- Network
- Server Logic
- Persistence
- Graphics
Why care about software design?

**Software Design**
- The components into which a problem is broken down
- The ways those components interact
Why care about software design?

- **Software Design**
  - The components into which a problem is broken down
  - The ways those components interact
  - The interfaces and abstractions they expose or hide
Why care about software design?

- **Software Design**
  - The components into which a problem is broken down
  - The ways those components interact
  - The interfaces and abstractions they expose or hide

- **Design affects the value of software**
  - Understandability
  - Performance
  - Reliability
  - Ease of change
Why care about software design?

- **Software Design**
  - The components into which a problem is broken down
  - The ways those components interact
  - The interfaces and abstractions they expose or hide

- **Design affects the value of software**
  - Understandability
  - Performance
  - Reliability
  - **Ease of change**

Most programming is “brown field” programming
Why care about software design?

- **Software Design**
  - The components into which a problem is broken down
  - The ways those components interact
  - The interfaces and abstractions they expose or hide

- **Design affects the value of software**
  - Understandability
  - Performance
  - Reliability
  - Ease of change

  - Poor value on these metrics is a significant risk
  - Good design can mitigate these risks
Why care about software design?

- **Software Design**
  - The components into which a problem is broken down
  - The ways those components interact
  - The interfaces and abstractions they expose or hide

- **Design affects the value of software**
  - Understandability
  - Performance
  - Reliability
  - Ease of...

  My goal is to have you able read and understand design decisions at FAANG....

  - Poor value on these metrics is a significant risk
  - Good design can mitigate these risks
What is bad design?

- Several red flags [Ousterhout 2018, ...]
What is bad design?

- Several red flags [Ousterhout 2018, ...]
  - Seemingly simple changes require modifying many locations
What is bad design?

- **Several red flags** [Ousterhout 2018, ...]
  - Seemingly simple changes require modifying many locations
  - A developer needs to know a great deal to complete a task
What is bad design?

• **Several red flags** [Ousterhout 2018, ...]
  – Seemingly simple changes require modifying many locations
  – A developer needs to know a great deal to complete a task
  – What code must be modified is unclear
What is bad design?

- **Several red flags** [Ousterhout 2018, ...]
  - Seemingly simple changes require modifying many locations
  - A developer needs to know a great deal to complete a task
  - What code must be modified is unclear
  - The impact of a change is unclear
What is bad design?

- **Several red flags** [Ousterhout 2018, ...]
  - Seemingly simple changes require modifying many locations
  - A developer needs to know a great deal to complete a task
  - What code must be modified is unclear
  - The impact of a change is unclear

- **Possible causes** [Ousterhout 2018]
  - *Dependencies* – Code cannot be understood in isolation
  - *Obscurity* – Important information is not obvious
What is bad design?

- Several red flags [Ousterhout 2018, ...]
  - Seemingly simple changes require modifying many locations
  - A developer needs to know a great deal to complete a task
  - What code must be modified is unclear
  - The impact of a change is unclear

- Possible causes [Ousterhout 2018]
  - Dependencies – Code cannot be understood in isolation
  - Obscurity – Important information is not obvious

- Design *complexity* arises from many portions of code interacting
  - Think of a basket or a braid. [Hickey 2011]
  Changing one strand is hard....
What is common in good designs?

- Loose Coupling (connectivity)
What is common in good designs?

- Loose Coupling (connectivity)
What is common in good designs?

- Loose Coupling (connectivity)

worse  –  Content

better
What is common in good designs?

- Loose Coupling (connectivity)

\[ \text{worse} \quad \text{– Content} \quad \text{better} \]
What is common in good designs?

- Loose Coupling (connectivity)

  worse
  - Content
  - Common global data

  better
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data

```
int global = ...

... = global

global = ...

... = global
```
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data

Note: “Solutions” like singletons have these constraints and worse.
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing

```cpp
class Parent {
public:
    virtual void foo() { bar(); }
    virtual void bar() {}
};
```

[Bloch, “Effective Java”]
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing

```cpp
class Parent {
public:
    virtual void foo() { bar(); }
    virtual void bar() {}
};

class Child : public Parent {
public:
    virtual void bar() { foo(); }
};
```

[Bloch, “Effective Java”]
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing

```
class Parent {
    public:
        void foo() { barImpl(); }
        void bar() { barImpl(); }
    private:
        virtual void barImpl() = 0;
};
```

Non Virtual Interfaces (NVI) help clarify & are common in C++.

```
class Parent {
    public:
        virtual void foo() { bar(); }
        virtual void bar() {}
    private:
        virtual void barImpl() = 0;
};
```

[Bloch, “Effective Java”]
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing
  - Temporal

worse

better
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing
  - Temporal

```
Cat cat = new Cat;
...
delete cat;
```
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing
  - Temporal

```
Cat cat = new Cat;
...
delete cat;
```

```
Process p;
p.doStep1();
p.doStep2();
p.doStep3();
```
What is common in good designs?

- **Loose Coupling (connectivity)**
  - Content
  - Common global data
  - Subclassing
  - Temporal

This is more insidious!

```
Cat cat = new Cat;
...
delete cat;
```

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

```
Process p;
p.foo();
p.bar();
p.baz();
```

Cat
What is common in good designs?

- **Loose Coupling (connectivity)**
  - Content
  - Common global data
  - Subclassing
  - Temporal
  - Passing data to/from each other

```python
x = foo(1, 2)
def foo(a, b):
    ...
```
What is common in good designs?

- Loose Coupling (connectivity)
  - Content
  - Common global data
  - Subclassing
  - Temporal
  - Passing data to/from each other
  - Independence
What is common in good designs?

- Loose Coupling
- High fan in / low fan out
What is common in good designs?

- Loose Coupling
- High fan in / low fan out
- Layers / Stratification
What is common in good designs?

- Loose Coupling
- High fan in / low fan out
- Layers / Stratification
What is common in good designs?

- Loose Coupling
- High fan in / low fan out
- Layers / Stratification

Layers are just a form of decoupling.
What is common in good designs?

- Loose Coupling
- High fan in / low fan out
- Layers / Stratification
- Cohesion
What is common in good designs?

- Loose Coupling
- High fan in / low fan out
- Layers / Stratification
- Cohesion

These attributes promote ease of change
What are our tools in creating designs?

- The same tools arise across languages
  - Polymorphism
  - Composition
What are our tools in creating designs?

- The same tools arise across languages
  - Polymorphism
  - Composition

- Understanding and leveraging these can enable safe, efficient, modifiable, and clear designs
What are our tools in creating designs?

- The same tools arise across languages
  - Polymorphism
  - Composition

- Understanding and leveraging these *can* enable safe, efficient, modifiable, and clear designs
- So we need to understand them....
Polymorphism

- What is polymorphism?
Polymorphism

- What is polymorphism?
  - A component is polymorphic if it may operate on multiple types
Polymorphism

- What is polymorphism?
  - A component is polymorphic if it may operate on multiple types

- What kinds of polymorphism are there?
  - At least 4(ish) broad classes that people should be familiar with
  - Even more (and further subdivision) in richer languages
Polymorphism

- **What is polymorphism?**
  - A component is polymorphic if it may operate on multiple types

- **What kinds of polymorphism are there?**
  - At least 4(ish) broad classes that people should be familiar with
  - Even more (and further subdivision) in richer languages

1) **Runtime polymorphism** (e.g. via inheritance in OOP)
Polymorphism

- What is polymorphism?
  - A component is polymorphic if it may operate on multiple types

- What kinds of polymorphism are there?
  - At least 4(ish) broad classes that people should be familiar with
  - Even more (and further subdivision) in richer languages

  1) **Runtime polymorphism** (e.g. via inheritance in OOP)
  2) **Parametric polymorphism** (e.g. via generics / templates)
Polymorphism

- **What is polymorphism?**
  - A component is polymorphic if it may operate on multiple types

- **What kinds of polymorphism are there?**
  - At least 4(ish) broad classes that people should be familiar with
  - Even more (and further subdivision) in richer languages

1. **Runtime polymorphism** (e.g. via inheritance in OOP)
2. **Parametric polymorphism** (e.g. via generics / templates)
3. **Overloading** (e.g. via classic overloading / type classes / traits)
Polymorphism

- What is polymorphism?
  - A component is polymorphic if it may operate on multiple types

- What kinds of polymorphism are there?
  - At least 4(ish) broad classes that people should be familiar with
  - Even more (and further subdivision) in richer languages

1) **Runtime polymorphism** (e.g. via inheritance in OOP)
2) **Parametric polymorphism** (e.g. via generics / templates)
3) **Overloading** (e.g. via classic overloading / type classes / traits)
4) **Coercion** (e.g. via implicit conversion)
5) ...
Polymorphism

- What is polymorphism?
  - A component is polymorphic if it may operate on multiple types

- What kinds of polymorphism are there?
  - At least 4(ish) broad classes that people should be familiar with
  - Even more (and further subdivision) in richer languages

1) Runtime polymorphism (e.g. via inheritance in OOP)
2) Parametric polymorphism (e.g. via generics / templates)
3) Overloading (e.g. via classic overloading / type classes / traits)
4) Coercion* (e.g. via implicit conversion)
5) ...

Different forms of polymorphism have different design trade offs
Polymorphism via Inheritance
(a quick review)
Polymorphism via Inheritance

- Inheritance
  - An approach of constructing a new entity in terms of an existing one
Polymorphism via Inheritance

- **Inheritance**
  - An approach of constructing a new entity in terms of an existing one
  - Can apply to classes, objects, ...
Polymorphism via Inheritance

- **Inheritance**
  - An approach of constructing a new entity in terms of an existing one
  - Can apply to classes, objects, ...
  - Most familiar nowadays through Object Oriented Programming (OOP)
Polymorphism via Inheritance

- **Inheritance**
  - An approach of constructing a new entity in terms of an existing one
  - Can apply to classes, objects, ...
  - Most familiar nowadays through Object Oriented Programming (OOP)

- **Class Inheritance**
  - Creates a new class in terms of an existing class
Polymorphism via Inheritance

- **Inheritance**
  - An approach of constructing a new entity in terms of an existing one
  - Can apply to classes, objects, ...
  - Most familiar nowadays through Object Oriented Programming (OOP)

- **Class Inheritance**
  - Creates a new class in terms of an existing class
  - Shares properties and behaviors with the new class

```
List
+ add()

ArrayList
+ add()
```
Polymorphism via Inheritance

- **Inheritance**
  - An approach of constructing a new entity in terms of an existing one
  - Can apply to classes, objects, ...
  - Most familiar nowadays through Object Oriented Programming (OOP)

- **Class Inheritance**
  - Creates a new class in terms of an existing class
  - Shares properties and behaviors with the new class
  - Can establish a subtyping relationship
Polymorphism via Inheritance

- **Inheritance**
  - An approach of constructing a new entity in terms of an existing one
  - Can apply to classes, objects, ...
  - Most familiar nowadays through Object Oriented Programming (OOP)

- **Class Inheritance**
  - Creates a new class in terms of an existing class
  - Shares properties and behaviors with the new class
  - Can establish a subtyping relationship

Java

```java
List list = new ArrayList();
void foo(List& someList);
...
ArrayList list;
foo(list);
```

C++

```cpp
List + add()  
is-a  
ArrayList + add()
```
What does good inheritance look like?

- Initial guidelines:
What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\phi$ is true for the base, then $\phi$ is true for the derived

```
Derived is substitutable for Base
```

```
Base
  A foo(B b)

Derived
  C foo(D d)
```
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be *more general*

```
Base
A foo(B b)
```
```
Derived
C foo(D d)
```
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general

```
B <: D

Base
A foo(B b)

Derived
C foo(D d)
```

Arguments are *contravariant*
What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general

- Arguments are \textit{contravariant}
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true for the derived
    - Arguments in the subtype may be more general

```
Base
A foo(B b)
```

```
Derived
C foo(D d)
```

```
B <: D
```

```
+ void draw(Rectangle)
```

```
+ void draw(Shape)
```

Arguments are **contravariant**
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If \( \varphi \) is true for the base, then \( \varphi \) is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained

C <: A

Base
A foo(B b)

Derived
C foo(D d)

Return types are *covariant*
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained

$C <: A$

Return types are **covariant**
What does good inheritance look like?

- Initial guidelines:
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained

```
Base A foo(B b)
```

```
Derived C foo(D d)
```

```
C <: A
```

```
LinuxDrawer + Shape getBounds()
```

```
LinuxDrawer + Rectangle getBounds()
```

Return types are **covariant**
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger

  ```
  assert(x > 0)  assert(x != 0)
  
  Base
  A foo(B b)
  
  Derived
  C foo(D d)
  ```
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger

```plaintext
Base
A foo(B b)

assert(x > 0)

Derived
C foo(D d)

assert(x != 0)
```
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If \( \varphi \) is true for the base, then \( \varphi \) is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger
    - Postconditions are not weaker

```plaintext
Base
A foo(B b)

Derived
C foo(D d)

assert(result != 0)    assert(result > 0)
```
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\varphi$ is true for the base, then $\varphi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger
    - Postconditions are not weaker

```
Base
A foo(B b)

Derived
C foo(D d)
```

assert (result != 0)  assert (result > 0)
What does good inheritance look like?

- **Initial guidelines:**
  - Prefer composition to inheritance
  - Liskov Substitution Principle
    - If $\phi$ is true for the base, then $\phi$ is true the derived
    - Arguments in the subtype may be more general
    - Return values in the subtype may be more constrained
    - Preconditions are not stronger
    - Postconditions are not weaker
    - Invariants must still hold

Base
A foo(B b)

Derived
C foo(D d)
So why is inheritance hard?
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?
- Every is-a relationship could instead be has-a!
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and *has-a* relationships *unambiguously* tell us how to apply inheritance?

- Every *is-a* relationship could instead be *has-a*!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

Professor *has-a* Researcher
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships \textit{unambiguously} tell us how to apply inheritance?

- Every \textit{is-a} relationship could instead be \textit{has-a}!
  - These often capture finer grained relationships
  - Break individual responsibilities into components
So why is inheritance hard?

- Do the LSP and has-a relationships unambiguously tell us how to apply inheritance?

- Every is-a relationship could instead be has-a!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

Note, these are now roles, not people.
So why is inheritance hard?

- Do the LSP and has-a relationships *unambiguously* tell us how to apply inheritance?

- Every *is-a* relationship could instead be *has-a*!
  - These often capture finer grained relationships
  - Break individual responsibilities into components

Note, these are now *roles*, not *people*.

- Whenever *is-a* applies, you must still make a decision
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Inheritance often *precludes* it
Choosing is-a or has-a

- Guide 1: Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic
Choosing is-a or has-a

- **Guide 1: Might the behavior need to change?**
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

- **Guide 2:** Might the type be used polymorphically?
  - Composition does not intrinsically aid it
Choosing is-a or has-a

- **Guide 1: Might the behavior need to change?**
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

- **Guide 2: Might the type be used polymorphically?**
  - Composition does not intrinsically aid it
  - Inheritance can enable it
Choosing is-a or has-a

- **Guide 1: Might the behavior need to change?**
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

- **Guide 2: Might the type be used polymorphically?**
  - Composition does not intrinsically aid it
  - Inheritance can enable it
  - **Consider** inheritance when a reference to a general type may point to a more specific one.
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic
  
  ```
  std::vector<People*> folks;
  ```

- **Guide 2:** Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance can enable it
  - **Consider** inheritance when a reference to a general type may point to a more specific one.

0) Student
1) Student
2) Lecturer
3) Professor
4) Student
Choosing is-a or has-a

- **Guide 1:** Might the behavior need to change?
  - Inheritance often precludes it
  - Composition often simplifies it
  - Use composition if the relationship is dynamic

- **Guide 2:** Might the type be used polymorphically?
  - Composition does not intrinsically aid it
  - Inheritance can enable it
  - *Consider* inheritance when a reference to a general type may point to a more specific one.

We will revisit this in the context of *algebraic data types.*
So let’s try it out...

- I need
  - Many different types of animals.

This should sound familiar...
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to move() and speak().
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

What does my design look like based on the rules?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

Is this good?
So let’s try it out...

I need

- Many different types of animals.
- Each should be able to \texttt{move()} and \texttt{speech()}.
- An \texttt{Animal} should be able to refer to any of them.

Does \texttt{Cat} serve a purpose?

Is this good?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to \texttt{move()} and \texttt{speak()}.
  - An \texttt{Animal} should be able to refer to any of them.

\begin{tikzpicture}
  \node (animal) {Animal};
  \node [below left=0.5cm and 1cm of animal] (parrot) {Parrot};
  \node [below right=0.5cm and 1cm of animal] (cat) {Cat};
  \node [below left=0.5cm and -1cm of animal] (professor) {Professor};
  \node [below right=0.5cm and -1cm of animal] (corgi) {Corgi};
  \node [below left=0.5cm and 1cm of professor] (mainecoon) {Maine Coon};
  \node [below right=0.5cm and 1cm of professor] (bengal) {Bengal};

  \draw [->] (animal) -- (parrot);
  \draw [->] (animal) -- (cat);
  \draw [->] (animal) -- (professor);
  \draw [->] (animal) -- (corgi);
  \draw [->] (professor) -- (mainecoon);
  \draw [->] (professor) -- (bengal);

\end{tikzpicture}
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Is this good?
Does it achieve reuse?
What if I want a new Animal at run time?
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

Can we do better?
So let’s try it out...

• I need
  – Many different types of animals.
  – Each should be able to `move()` and `speak()`.
  – An `Animal&` should be able to refer to any of them.

  Can we do better?

If someone on my team did this multiple times, I would consider firing them.

Hierarchies in data need not be hierarchies in the type system!
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal&` should be able to refer to any of them.

Can we do better?  
Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?  Recall: identify & isolate change

Animal

has-a

Movement
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better? Recall: identify & isolate change

- Movement
  - Movement selects from the ways any Animal can move.
  - Animal
  - has-a Movement
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better? Recall: identify & isolate change

```
Animal
  has-a Movement
has-a Vocalization
```
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

```
Animal
  has-a
  Movement
  has-a
  Crawl
  has-a
  Vocalization
```
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

![Diagram](attachment:diagram.png)

- Animal
  - has-a Movement
    - Crawl
    - Fly
  - has-a Vocalization
So let’s try it out...

I need

- Many different types of animals.
- Each should be able to move() and speak().
- An Animal should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

Animal

has-a

Movement

Crawl
Fly
Saunter

has-a

Vocalization
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?  Recall: identify & isolate change
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better? Recall: identify & isolate change

Animal

- Movement
  - Crawl
  - Fly
  - Saunter

- Vocalization
  - Tweet
  - Meow
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

```
Animal
  has-a Movement
    Crawl
    Fly
    Saunter
  has-a Vocalization
    Tweet
    Meow
    Ramble
```
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

```
Animal
  - has-a Movement
    - Crawl
    - Fly
    - Saunter

  - has-a Vocalization
    - Tweet
    - Meow
    - Ramble
    - Bark
```
So let’s try it out...

- I need
  - Many different types of animals.
  - Each should be able to `move()` and `speak()`.
  - An `Animal` should be able to refer to any of them.

Can we do better?

Recall: identify & isolate change

```java
class Animal {
    Movement& m;
    void move() {
        m.move();
    }
};
```
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

- Inheritance contracts for fine grained policies
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

- Inheritance contracts for fine grained policies

- Enables dynamic selection & configuration of which policies are desired
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

- Inheritance contracts for fine grained policies

- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out **Stationary**, then **Run**, then be **Stationary**
Shallow, fine grained inheritance

- Avoids reimplementation of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out Stationary, then Run, then be Stationary

Previously static requirements will often become dynamic.
Shallow, fine grained inheritance

- Avoids reimplementaion of common behavior
  - e.g. Common aspects of Animal are just fields of Animal

- Inheritance contracts for fine grained policies

- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out Stationary, then Run, then be Stationary

- Directly identifies & addresses risks of change in class design
Shallow, fine grained inheritance

- Avoids reimplementations of common behavior
  - e.g. Common aspects of Animal are just fields of Animal
- Inheritance contracts for fine grained policies
- Enables dynamic selection & configuration of which policies are desired
  - e.g. A Cat may start out Stationary, then Run, then be Stationary
- Directly identifies & addresses risks of change in class design
- Does not focus on reusing from the base class. Instead makes the derived class reusable.
Violating Examples

- There are a few bad examples that students & text books have brought to me.
Violating Examples

- There are a few bad examples that students & text books have brought to me.
Violating Examples

- There are a few bad examples that students & textbooks have brought to me.
Violating Examples

- There are a few bad examples that students & textbooks have brought to me.
Parametric Polymorphism
(a quick review?)
Parametric Polymorphism

- Parametric polymorphism enables defining generic components over a family of types using type parameters.
Parametric Polymorphism

- Parametric polymorphism enables defining generic components over a family of types using type parameters

Commonly referred to as **generics** or **templates**
Parametric Polymorphism

- Parametric polymorphism enables defining generic components over a family of types using type parameters

Java
```java
public class ArrayList<E> {...}
```

C++
```cpp
template <class E>
class vector;
```

Typescript
```typescript
class ArrayList<E> {...}
```

Python
```python
from typing import TypeVar
t = TypeVar('T')
class SpecialList(Generic[T]):
    def __init__(self, value: T) -> None:
        ...
```

Commonly referred to as *generics* or *templates*
Parametric Polymorphism

- Parametric polymorphism enables defining generic components over a family of types using type parameters.

Java
```
public class ArrayList<E> {...}
```

Typescript
```
class ArrayList<E> {...}
```

Python
```
T = TypeVar('T')
class SpecialList(Generic[T]):
    def __init__(self, value: T) -> None: ...
```

C++
```
template <class E>
class vector;
```

C++
```
std::vector<int> v1 = {1, 2, 3, 4, 5};
```
Parametric Polymorphism

- Parametric polymorphism enables defining generic components over a family of types using type parameters.

Java

```java
public class ArrayList<E> {...}
```

C++

```cpp
template <class E>
class vector;
```

Typescript

```typescript
class ArrayList<E> {...}
```

Python

```python
T = TypeVar('T')
class SpecialList(Generic[T]):
    def __init__(self, value: T) -> None:
        ...
```

```
Parameters can sometimes be inferred.
```

```cpp
std::vector<int> v1 = {1, 2, 3, 4, 5};
```
Parametric Polymorphism

- Parametric polymorphism enables defining generic components over a family of types using type parameters.

- Enables careful *abstraction* of design components:
  - A class/function/data structure/algorithm can be written & validated once
  - Intentions can be clearer within code
Suppose an algorithm needs to find an element in a collection & increment it.
Suppose an algorithm needs to find an element in a collection & increment it.

```cpp
void bigAlgorithm(...) {
    std::vector<int> c;
    ...
    for (auto i = begin(c), e = end(c); i != e; ++i) {
        if (*i == v) {
            ++*i;
            break;
        }
    }
    ...
}
```
Suppose an algorithm needs to find an element in a collection & increment it.

```cpp
void bigAlgorithm(...) {
    std::vector<int> c;
    ...
    for (auto i = begin(c), e = end(c); i != e; ++i) {
        if (*i == v) {
            ++*i;
            break;
        }
    }
    ...
}
```

This is awful.
Intentions are unclear.
Modifiability is low.
Reusability is low.
Suppose an algorithm needs to find an element in a collection &
increment it.

```cpp
void bigAlgorithm(...) {
    std::vector<int> c;
    ...
    for (auto i = begin(c), e = end(c); i != e; ++i) {
        if (*i == v) {
            ++*i;
            break;
        }
    }
    ...
}
```

```cpp
template<typename C, typename V>
size_t find(const C& c, const V& v) {
    for (auto& [index, element] : enumerate(c)) {
        if (element == v) {
            return index;
        }
    }
    return (size_t)-1;
}
```
Parametric Polymorphism

- Suppose an algorithm needs to find an element in a collection & increment it.

```cpp
template<typename C, typename V>
size_t find(const C& c, const V& v) {
    for (auto& [index, element] : enumerate(c)) {
        if (element == v) {
            return index;
        }
    }
    return (size_t)-1;
}
```

```cpp
void bigAlgorithm(...) {
    std::vector<int> c;
    ...
    auto index = find(c, v);
    ++c[index];
    ...
}
```
Suppose an algorithm needs to find an element in a collection & increment it.

```cpp
void bigAlgorithm(...) {
    std::vector<int> c;
    ...
    for (auto i = begin(c), e = end(c); i != e; ++i) {
        if (*i == v) {
            ++*i;
            break;
        }
    }
    ...
}
```

```cpp
template<typename C, typename V>
size_t find(const C& c, const V& v) {
    for (auto& [index, element] : enumerate(c)) {
        if (element == v) {
            return index;
        }
    }
    return (size_t)-1;
}
```

```cpp
void bigAlgorithm(...) {
    std::vector<int> c;
    ...
    auto index = find(c, v);
    ++c[index];
    ...
}
```

```cpp
void otherAlgorithm(...) {
    std::vector<string> d = ...;
    auto index = find(d, w);
    ...
}
```
Parametric Polymorphism

- Type variables can also be bounded / restricted
  - Consider `find(C,V)`, it should require that `ElementType(C) = V`
  - Restricting to subtypes / supertypes is common

Java
```java
class D <T extends A & B & C> { }
class F <? extends E> { }
```
Parametric Polymorphism

- Type variables can also be bounded / restricted
  - Consider `find(C, V)`, it should require that `ElementType(C) = V`
  - Restricting to subtypes / supertypes is common

Java

```java
class D <T extends A & B & C> { }
class F <? extends E> { }
```

public interface `Map<K, V>` {
  void putAll(`Map<? extends K, ? extends V>` m)
}

Parametric Polymorphism

- Type variables can also be bounded / restricted
  - Consider `find(C,V)`, it should require that `ElementType(C) = V`
  - Restricting to subtypes / supertypes is common

Java

```java
class D <T extends A & B & C> { }
class F <? extends E> { }
```

C++

```cpp
template <typename T, typename=std::enable_if_t<std::is_class_v<T>>>
void foo(const T& t) {
    std::cout << "T is a class type\n";
}
```

Such constraints can be cleaner in C++20.
Parametric Polymorphism

- Type variables can also be bounded / restricted
  - Consider `find(C,V)`, it should require that `ElementType(C) = V`
  - Restricting to subtypes / supertypes is common

Java

```java
class D <T extends A & B & C> { }
class F <? extends E> { }
```

C++

```cpp
template <typename T, typename=std::enable_if_t<std::is_class_v<T>>>
void foo(const T& t) {
  std::cout << "T is a class type\n";
}
template <typename C>
C::iterator_type find(const C& c, C::element_type v) {
  ...
}
```

Some scenarios are bounded by convention.
Parametric Polymorphism

- *Specialized* instances can sometimes be created
  - Sometimes domain knowledge allows more efficient implementations
Parametric Polymorphism

- *Specialized* instances can sometimes be created

```cpp
template <class PointedTo, class Value>
class PointerValuePair {
    PointedTo* p;
    Value v;
    PointedTo* getP();
    Value getV();
};
```
Parametric Polymorphism

- *Specialized* instances can sometimes be created

```cpp
template <class PointedTo, class Value>
class PointerValuePair {
    PointedTo* p;
    Value v;
    PointerTo* getP();
    Value getV();
};

template <class PointedTo>
class PointerValuePair<PointedTo,int> {
    uintptr_t compact;
    PointerTo* getP() {
        return reinterpret_cast<PointedTo*>(compact & ~0xFFFFFFF8);
    }
    int getV() { return compact & 0x00000007; }
};
```
Parametric Polymorphism

- *Specialized* instances can sometimes be created

```cpp
template <class PointedTo, class Value>
class PointerValuePair {
    PointedTo* p;
    Value v;
    PointedTo* getP();
    Value getV();
};

template <class PointedTo>
class PointerValuePair<PointedTo,int> {
    uintptr_t compact;
    PointedTo* getP() {
        return reinterpret_cast<PointedTo*>(compact & ~0xFFFFFFFF);
    }
    int getV() { return compact & 0x00000007; }
};

Note, this example is still too simple to be safe.
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};
```
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
    void printImpl() { printf("Yo\n"); }
};
```
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```c++
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
    void printImpl() { printf("Yo\n"); }
};
```
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
    void print() { getDerived().printImpl(); }
private:
    T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
    void printImpl() { printf("Yo\n"); }
};
```
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
  void print() { getDerived().printImpl(); }
private:
  T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
  void printImpl() { printf("Yo\n"); }
};
```

What other approaches could we have used? What are the trade offs?
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```cpp
template<class T>
class Base {
public:
   void print() { getDerived().printImpl(); }
private:
   T& getDerived() { return *static_cast<T*>(this); }
};

class Specific : public Base<Specific> {
public:
   void printImpl() { printf("Yo\n"); }
};
```

What other approaches could we have used?
What are the trade offs?

Flexibility vs Efficiency
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

Have those of you familiar with Java seen this before?
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```
public class LocalTime implements Comparable<LocalTime> {
    ...
}
```

Have those of you familiar with Java seen this before?
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```java
public class LocalTime implements Comparable<LocalTime> {
    ...
}
```

Have those of you familiar with Java seen this before?

```java
public interface Comparable<T> {
    int compareTo(T o);
}
```
Selecting forms of polymorphism

- Sometimes information needs to flow from a derived class to a base class.

```java
public class LocalTime implements Comparable<LocalTime> {
    ...
}
```

```java
public interface Comparable<T> {
    int compareTo(T o);
}
```

This Curiously Recurring Template Pattern (CRTP) can help in building more robust APIs.

Have those of you familiar with Java seen this before?
Selecting forms of polymorphism

- There are richer interactions between polymorphisms that enable clean & simple API design.
Selecting forms of polymorphism

- There are richer interactions between polymorphisms that enable clean & simple API design.
  - These issues are not the focus of this class
  - They are discussed more in CMPT 373
  - Feel free to ask questions about them on our discussion fora
Ad-hoc Polymorphism
Ad-hoc Polymorphism

- Ad-hoc polymorphism can occur on a case by case basis
  - Overloading
  - Type conversions / coercion
  - Type traits & type classes for flexible & structured overloading
Coercion

- Defining allowed conversions can lead to safe & intuitive APIs
- Example: Suppose we want APIs that can operate on contiguous collections.
Coercion

- Defining allowed conversions can lead to safe & intuitive APIs
- Example:
  Suppose we want APIs that can operate on contiguous collections.

```cpp
template<class E, auto N>
void foo(const E(&c)[N]);
```
Coercion

- Defining allowed conversions can lead to safe & intuitive APIs
- Example:
  Suppose we want APIs that can operate on contiguous collections.

```cpp
template<class E, auto N>
void foo(const E(&c)[N]);

template<class E, auto N>
void foo(const std::array<E,N>& c);
```
Coercion

- Defining allowed conversions can lead to safe & intuitive APIs

- Example:
  Suppose we want APIs that can operate on contiguous collections.

```cpp
template<class E, auto N>
void foo(const E(&c)[N]);

template<class E, auto N>
void foo(const std::array<E,N>& c);

template<class E>
void foo(const std::vector<E>& c);
```
Coercion

- Defining allowed conversions can lead to safe & intuitive APIs
- **Example:**
  Suppose we want APIs that can operate on contiguous collections.

```cpp
template<class E, auto N>
void foo(const E(&c)[N]);

template<class E, auto N>
void foo(const std::array<E,N>& c);

template<class E>
void foo(const std::vector<E>& c);

void foo(const std::string& c);
```
Coercion

- Defining allowed conversions can lead to safe & intuitive APIs

- Example:
  Suppose we want APIs that can operate on contiguous collections.
Coercion

- Defining allowed conversions can lead to safe & intuitive APIs

- Example:
  Suppose we want APIs that can operate on contiguous collections.

```cpp
template<class E, auto N>
void foo(const E(&c)[N]);

template<class E, auto N>
void foo(const std::array<E,N>& c);

template<class E>
void foo(const std::vector<E>& c);

void foo(const std::string& c);
```

Yuck.
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

We can start by thinking what is common.
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

```cpp
template<class E>
struct Span {
    E* first;
    size_t count;
};
```
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

```cpp
template<class E>
struct Span {
    template<class E, auto N>
    Span(const std::array<E,N>& c);

    E* first;
    size_t count;
};
```

In C++, a non explicit 1 arg constructor defines a compatible conversion.
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

```cpp
template<class E>
struct Span {
    template<class E, auto N>
    Span(const std::array<E,N>& c);

    template<class E>
    Span(const std::vector<E>& c);

    E* first;
    size_t count;
};
```
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

```c++
template<class E>
struct Span {
    template<class E, auto N>
    Span(const std::array<E,N>& c);

    template<class E>
    Span(const std::vector<E>& c);

    E* first;
    size_t count;
};
```

void foo(Span<E> c);
void bar(Span<E> c);
...

std::vector v = {1, 2, 3, 4, 5};
foo(v);

int v[] = {1, 2, 3, 4, 5};
bar(v);

foo(“This works for free”);

This enables convenient & efficient generic APIs.
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

```cpp
template<class E>
struct Span {
    template<class E, auto N>
    Span(const std::array<E,N>& c);
    template<class E>
    Span(const std::vector<E>& c);
    E* first;
    size_t count;
};

void foo(Span<E> c);
void bar(Span<E> c);
...
std::vector v = {1, 2, 3, 4, 5};
foo(v);
bar(v);
int v[N] = {1, 2, 3, 4, 5};
foo(v);
bar(v);
foo("This works for free");
```

This enables convenient & efficient generic APIs.
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

```cpp
template<class E>
struct Span {
    template<class E, auto N>
    Span(const std::array<E,N>& c);

    template<class E>
    Span(const std::vector<E>& c);

    E* first;
    size_t count;
};

void foo(Span<E> c);
void bar(Span<E> c);
...
std::vector v = {1, 2, 3, 4, 5};
foo(v);
int v[] = {1, 2, 3, 4, 5};
bar(v);
foo("This works for free");
```

This enables convenient & efficient generic APIs.
Coercion

- Perhaps we can construct a new type that is conversion compatible with all desired types...

```cpp
template<class E>
struct Span {
    template<class E, auto N>
    Span(const std::array<E,N>& c);
    template<class E>
    Span(const std::vector<E>& c);
    E* first;
    size_t count;
};
```

```cpp
... std::vector v = {1, 2, 3, 4, 5}; foo(v);
```

```cpp
int v[] = {1, 2, 3, 4, 5};
bar(v);
```

```cpp
foo("This works for free");
```
Type Traits

- Careful use of specialization can structure overloading & extend behaviors
Type Traits

- Careful use of specialization can structure overloading & extend behaviors
- Suppose we want to implement graph algorithms to traverse arbitrary data structures.
Type Traits

- Careful use of specialization can structure overloading & extend behaviors

- Suppose we want to implement graph algorithms to traverse arbitrary data structures.
  - What constraints exist?
Type Traits

- Careful use of specialization can structure overloading & extend behaviors

- **Suppose we want to implement graph algorithms to traverse arbitrary data structures.**
  - What constraints exist?
  - How might we design a nice API?
    - Via inheritance?
    - Via parametric polymorphism?
Type Traits

- Careful use of specialization can structure overloading & extend behaviors

- Suppose we want to implement graph algorithms to traverse arbitrary data structures.
  - What constraints exist?
  - How might we design a nice API?
    - Via inheritance?
    - Via parametric polymorphism?

- *Type traits* and specialization can convey details about a type that enable generic algorithms
  - Specializations carry the extra details for an overload
template<typename GraphKind>
struct GraphTraits {
   using Error = typename GraphKind::ABCD;
};
template<typename GraphKind>
struct GraphTraits {
    using Error = typename GraphKind::ABCD;
};

template<>
struct GraphTraits<SocialGraph> {

};
Type Traits

template<typename GraphKind>
struct GraphTraits {
    using Error = typename GraphKind::ABCD;
};

template<>
struct GraphTraits<SocialGraph> {
    using NodeRef = ...;
    using ChildIterator = ...;
    static NodeRef get_entry(SocialGraph&) {...}
    static ChildIterator child_begin(NodeRef&) {...}
    static ChildIterator child_end(NodeRef&) {...}
};
Type Traits

template<typename GraphKind>
struct GraphTraits {
    using Error = typename GraphKind::ABCD;
};

template<>
struct GraphTraits<SocialGraph> {
    using NodeRef = ...;
    using ChildIterator = ...;
    static NodeRef get_entry(SocialGraph&) {...}
    static ChildIterator child_begin(NodeRef&) {...}
    static ChildIterator child_end(NodeRef&) {...}
};
Type Traits

template<typename GraphKind>
struct GraphTraits {
    using Error = typename GraphKind::ABCD;
};

template<>
struct GraphTraits<SocialGraph> {
    using NodeRef = ...;
    using ChildIterator = ...;
    static NodeRef get_entry(SocialGraph&) {...}
    static ChildIterator child_begin(NodeRef&) {...}
    static ChildIterator child_end(NodeRef&) {...}
};

template<class Kind, class GT=GraphTraits<Kind>>
void visualizeGraph(Kind& graph);
template<typename GraphKind>
struct GraphTraits {
    using Error = typename GraphKind::ABCD;
};

template<>
struct GraphTraits<SocialGraph> {
    using NodeRef = ...;
    using ChildIterator = ...;
    static NodeRef get_entry(SocialGraph&) {...}
    static ChildIterator child_begin(NodeRef&) {...}
    static ChildIterator child_end(NodeRef&) {...}
};

template<class Kind, class GT=GraphTraits<Kind> >
void visualizeGraph(Kind& graph);
Type Traits

template<typename GraphKind>
struct GraphTraits {
  using Error = typename GraphKind::ABCD;
};

template<>
struct GraphTraits<SocialGraph> {
  using NodeRef = ...;
  using ChildIterator = ...;
  static NodeRef get_entry(SocialGraph&) {...}
  static ChildIterator child_begin(NodeRef&) {...}
  static ChildIterator child_end(NodeRef&) {...}
};

template<class Kind, class GT=GraphTraits<Kind>>
void visualizeGraph(Kind& graph);

SocialGraph g;
...
visualizeGraph(g);
Type Traits

Regardless of the actual graph data structure, or even its API, traits allow generic algorithms to work!
Type Traits

- They are even common in the C++ standard library
Type Traits

- They are even common in the C++ standard library `<functional>`

```cpp
namespace std {
  template< class Key >
  struct hash;
}
```
Type Traits

- They are even common in the C++ standard library `<functional>`

```cpp
namespace std {
    template< class Key >
    struct hash;
}
```

```cpp
<unordered_set>
```

```cpp
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
> class unordered_set;
```
Type Traits

- They are even common in the C++ standard library `<functional>`

```cpp
namespace std {
    template< class Key >
    struct hash;
}
```

This doesn’t implement hashing for custom types.

What if I want to add a `Cat` to an `unordered_set`?

```cpp
template<
    class Key,
    class Hash = std::hash<Key>,
    class KeyEqual = std::equal_to<Key>,
    class Allocator = std::allocator<Key>
>
class unordered_set;
```
Type Traits

- They are even common in the C++ standard library
  `<functional>`
  `<unordered_set>`

```cpp
namespace std {
    template< class Key >
    struct hash;
}
```

```cpp
<unordered_set>
namespace std {
    template<
        class Key,
        class Hash = std::hash<Key>,
        class KeyEqual = std::equal_to<Key>,
        class Allocator = std::allocator<Key>
    > class unordered_set;
}
```

```cpp
<Cats.h>
namespace std {
    template<>
    struct hash<Cat> {
        std::size_t
        operator()(Cat const& s) const noexcept {
            return ...;
        }
    };
}
```
Type Traits

- They are even common in the C++ standard library

```cpp
#include <functional>
namespace std {
    template< class Key >
    struct hash;
}

#include <unordered_set>
namespace std {
    template<
        class Key,
        class Hash = std::hash<Key>,
        class KeyEqual = std::equal_to<Key>,
        class Allocator = std::allocator<Key>
    > class unordered_set;
}

#include <Cats.h>
namespace std {
    template<>
    struct hash<Cat> {
        std::size_t
        operator()(Cat const& s) const noexcept {
            return ...;
        }
    };

    std::unordered_set<Cat> bigBagOfCats;
}
Composition
Composition

- The Principle of Compositionality (roughly)
  - The meaning of a complex entity is determined by the meanings of its constituents and the rules used to combine them.
Composition

- **The Principle of Compositionality (roughly)**
  - The meaning of a complex entity is determined by the meanings of its constituents and the rules used to combine them.

  **Or in software**
  - The meaning of a component should be clear from the meanings of its constituents and how they are used.
Composition

- **The Principle of Compositionality (roughly)**
  - The meaning of a complex entity is determined by the meanings of its constituents and the rules used to combine them.

  Or in software
  - The meaning of a component should be clear from the meanings of its constituents and how they are used.

- **But how can we achieve this? We’ll look at a few approaches**
  - Region / scope bounded behavior
  - Ownership
  - Algebraic data types
Region based behaviors

- Consider functions as a unit of abstraction
  - Possible incoming data
  - Behavior
  - Possible outgoing data
Region based behaviors

- Consider functions as a unit of abstraction
  - Possible incoming data
  - Behavior
  - Possible outgoing data
Region based behaviors

- Consider functions as a unit of abstraction
  - Possible incoming data
  - Behavior
  - Possible outgoing data
Region based behaviors

- Consider functions as a unit of abstraction
  - Possible incoming data
  - Behavior
  - Possible outgoing data
- Good abstractions tend to be *self contained*, but bad ones will leak obligations on their users
Region based behaviors

- Consider functions as a unit of abstraction
  - Possible incoming data
  - Behavior
  - Possible outgoing data

- Good abstractions tend to be self contained, but bad ones will leak obligations on their users

```c
Mutex m;
lock(m);
...
unlock(m);
```
Region based behaviors

- Consider functions as a unit of abstraction
  - Possible incoming data
  - Behavior
  - Possible outgoing data

- Good abstractions tend to be *self contained*, but bad ones will leak obligations on their users

```c
Mutex m;
lock(m);
...
unlock(m);
```

What if we don’t unlock the mutex?
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- Examples
  - Java: synchronized blocks/methods, try-with-resources

```java
synchronized (this) {
    ...
}

try (BufferedReader br =
    new BufferedReader(new FileReader(path))) {
    return br.readLine();
}
```
Region based behaviors

• Modern languages enable denoting the region for an abstraction
  – Helps to bound the impact and provide composable interfaces.
  – Design the inconsistency and lack of hygiene out of a system

• Examples
  – Java: synchronized blocks/methods, try-with-resources
  – Python: with

```python
with open(path) as infile:
    ...
```
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- Examples
  - Java: `synchronized` blocks/methods, `try-with-resources`
  - Python: `with`
  - C#: `using`

```csharp
using (var reader = new StreamReader(path)) {
    ...
}
```
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- **Examples**
  - Java: `synchronized` blocks/methods, `try-with-resources`
  - Python: `with`
  - C#: `using`
  - C++: RAII (*Resource Acquisition Is Initialization*)
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- Examples
  - Java: synchronized blocks/methods, try-with-resources
  - Python: with
  - C#: using
  - C++: RAII

```cpp
void memoryResource() {
    auto w = std::make_unique<Widget>(3, "bofrot");
    foo(*w);
}
```
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- Examples
  - Java: `synchronized` blocks/methods, `try-with-resources`
  - Python: `with`
  - C#: `using`
  - C++: RAII

```cpp
void memoryResource() {
    auto w = std::make_unique<Widget>(3, "bofrot");
    // Use w...
}
```

```cpp
void memoryResource() {
    Widget w(3, "bofrot");
    foo(w);
}
```

Or better...
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- Examples
  - Java: `synchronized` blocks/methods, `try-with-resources`
  - Python: `with`
  - C#: `using`
  - C++: RAII

```cpp
void memoryResource() {
    auto w = std::make_unique<Widget>(3, "bofrot");
    foo(*w);
}

void fileResource() {
    std::ofstream out{"output.txt"};
    out << "Boston cream\n";
}
```
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- **Examples**
  - Java: *synchronized* blocks/methods, *try-with-resources*
  - Python: *with*
  - C#: *using*
  - C++: RAII

```cpp
void memoryResource() {
  std::unique_ptr<Widget> w = std::make_unique<Widget>(3, "bofrot");
  foo(*w);
}

void fileResource() {
  std::ofstream out{"output.txt"};
  out << "Boston cream\n";
}

std::mutex m;
void synchronization() {
  std::lock_guard<std::mutex> guard(g_pages_mutex);
  out << "Thread safe fritter\n";
}
```
Region based behaviors

- Modern languages enable denoting the region for an abstraction
  - Helps to bound the impact and provide composable interfaces.
  - Design the inconsistency and lack of hygiene out of a system

- **Examples**
  - Java: *synchronized* blocks/methods, *try-with-resources*
  - Python: *with*
  - C#: *using*
  - C++: RAII
  - Rust: lifetimes, borrowing, RAII, ...
Ownership

- Sometimes lexical bounds are not known
  - Ownership *designates* whose *responsibility* it is to manage a resource

makes explicit & obvious
Ownership

- Sometimes lexical bounds are not known
  - Ownership designates whose responsibility it is to manage a resource
  - Applies when a resource has uncertain lifetimes
Ownership

- Sometimes lexical bounds are not known
  - Ownership *designates* whose *responsibility* it is to manage a resource
  - Applies when a resource has uncertain lifetimes
  - Combines region abstractions to clean up automatically
Ownership

- Sometimes lexical bounds are not known
  - Ownership designates whose responsibility it is to manage a resource
  - Applies when a resource has uncertain lifetimes
  - Combines region abstractions to clean up automatically

```cpp
std::unique_ptr<Widget> memoryResource() {
    auto w = std::make_unique<Widget>(3, "bofrot");
    ...
    return w;
}
```

Whose responsibility is it to clean w? When does it happen?
Ownership

- Sometimes lexical bounds are not known
  - Ownership *designates* whose *responsibility* it is to manage a resource
  - Applies when a resource has uncertain lifetimes
  - Combines region abstractions to clean up automatically

What do these signatures connote?

```c++
void foo(unique_ptr<Widget> w);
void foo(unique_ptr<Widget>& w);
void foo(Widget& w);
```
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

```c
struct Cat {
    enum Activity {RUNNING, SLEEPING};
    Activity activity;
    uint64_t runningSpeed;
};
```

What *problems* does this design enable?
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

```haskell
type Bool = True | False
```
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

```plaintext
type Bool = True | False

type Activity = Running(int speed) | Sleeping
```
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

```plaintext
type Bool = True | False

type Activity = Running(int speed) | Sleeping

Note: it is impossible to ask for the running speed of something sleeping!
```
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system.

- **Algebraic Data Types** enable the composable construction of types through combining types.
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

- *Algebraic Data Types* enable the composable construction of types through combining types
  - *Sum types* express disjoint alternatives

```plaintext
type Activity = Running(int speed) | Sleeping
```
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

- **Algebraic Data Types** enable the composable construction of types through combining types
  - **Sum types** express disjoint alternatives

```plaintext
type Activity = Running(int speed) | Sleeping
```

How would you express this is C? In C++?
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

- *Algebraic Data Types* enable the composable construction of types through combining types
  - *Sum types* express disjoint alternatives

```plaintext
type Activity = Running(int speed) | Sleeping
```
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

- **Algebraic Data Types** enable the composable construction of types through combining types
  - **Sum types** express disjoint alternatives
  - **Product types** express combinations

```c
struct MapEntry { Key key; Value value; };
```
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system

- *Algebraic Data Types* enable the composable construction of types through combining types
  - Sum types express disjoint alternatives
  - Product types express combinations

- Note, the preferred way of extracting from an ADT is through pattern matching
Algebraic Data Types

- Carefully combining types can design more inconsistent & erroneous states out of a system
- Algebraic Data Types enable the composable construction of types through combining types
  - Sum types express disjoint alternatives
  - Product types express combinations
- Note, the preferred way of extracting from an ADT is through pattern matching

```rust
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)
}
let msg = Message::Quit;
match msg {
    Message::Quit => {
        println!("The Quit variant has no data to destruct."),
    },
    Message::Move { x, y } => {
        println!("Move {} and {}", x, y);
    },
    Message::Write(text) => println!("Text message: {}", text),
}
```

[From the Rust Book]
Algebraic Data Types

Carefully combining types can design more inconsistent & erroneous states out of a system.

Algebraic Data Types enable the composable construction of types through combining types.

- Sum types express disjoint alternatives
- Product types express combinations

Note, the preferred way of extracting from an ADT is through pattern matching.

```rust
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)
}

let msg = Message::Quit;
match msg {
    Message::Quit => {
        println!("The Quit variant has no data to destructure."");
    },
    Message::Move { x, y } => {
        println!("Move {} and {}", x, y);
    },
    Message::Write(text) => println!("Text message: {}", text),
}
```

[From the Rust Book]
Carefully combining types can design more inconsistent & erroneous states out of a system. Algebraic Data Types enable the composable construction of types through combining types. Sum types express disjoint alternatives—Product types express combinations.

Note, the preferred way of extracting from an ADT is through pattern matching.

```
enum Message {
    Quit,
    Move { x: i32, y: i32 },
    Write(String)
}

let msg = Message::Quit;
match msg {
    Message::Quit => {
        println!("The Quit variant has no data to destructure.")
    },
    Message::Move { x, y } => {
        println!("Move {} and {}", x, y);
    },
    Message::Write(text) => println!("Text message: {}", text),
}
```

[From the Rust Book]
Designing Design Patterns
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
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.

Blind use of patterns is another reason why people dislike OOP.
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?

```cpp
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);
... 
```
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.

```cpp
... 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.

Create some work.

Do the created work.
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.

Create some work.

Do the created work.

- 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.

```cpp
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.

```cpp
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.

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

```cpp
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.

```cpp
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.

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

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

```cpp
class WorkKind2 : public Work {
    Result doWork() override {...}
};
```
e.g. **Behavioral Pattern: Command**

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

- This is the *command pattern*

```cpp
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

```cpp
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

```cpp
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
  – Parameterizable via constructor
The Command Pattern

• Benefits
  – Decouples a request / behavior from the invoker
  – Invoker decides when to invoke without caring what
  – Parameterizable via constructor

```cpp
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
  - Parameterizable via constructor

```cpp
... 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
  - Parameterizable via constructor
  - Sequences of commands can be easily batched
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently

Diagram:

- Employee
  - Manager
  - Underling
**Problem:** Add new behaviors to a set of types

- Different classes can perform the same action differently

```java
Manager manager;
manager.updatePay();

Underling underling;
underling.updatePay();
```
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently.
- Sometimes you want to add a *new kind of action* to a set of related classes.
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently
- Sometimes you want to add a *new kind of action* to a set of related classes

```java
Manager manager;
manager.serialize();

Underling underling;
underling.serialize();
```
**Problem:** Add new behaviors to a set of types

- Different classes can perform the same action differently.
- Sometimes you want to add a *new kind of action* to a set of related classes.
- There may be *many* different types of actions to add.
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently
- Sometimes you want to add a new kind of action to a set of related classes
- There may be many different types of actions to add

Operations for Employees

updatePay
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently
- Sometimes you want to add a new kind of action to a set of related classes
- There may be many different types of actions to add

Operations for Employees

```
updatePay
serialize
```
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently
- Sometimes you want to add a new kind of action to a set of related classes
- There may be many different types of actions to add

Operations for Employees

- updatePay
- serialize
- printPerformanceReview
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently
- Sometimes you want to add a new kind of action to a set of related classes
- There may be many different types of actions to add

Operations for Employees

updatePay
serialize
printPerformanceReview
Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently
- Sometimes you want to add a new kind of action to a set of related classes
- There may be many different types of actions to add
- Sometimes, you can't even know all of the actions in advance!
**Problem: Add new behaviors to a set of types**

- Different classes can perform the same action differently
- Sometimes you want to add a *new kind of action* to a set of related classes
- There may be many different types of actions to add
- Sometimes, you can't even know all of the actions in advance!

Why are these problems?
Problem: Add new behaviors to a set of types

- Let us take a look at our `Employee` base class...

```cpp
class Employee {
public:

    ... 
    virtual void updatePay() = 0;
    virtual void performJob() = 0;
    virtual void serialize() = 0;
    virtual void displayAvatar() = 0;
    virtual void printPerformanceReview() = 0;
    virtual void findFavoriteOfficeMate() = 0;
    virtual void procrastinate() = 0;
};
```
Problem: Add new behaviors to a set of types

- Let us take a look at our Employee base class...

```cpp
class Employee {
public:
    ...
    virtual void updatePay() = 0;
    virtual void performJob() = 0;
    virtual void serialize() = 0;
    virtual void displayAvatar() = 0;
    virtual void printPerformanceReview() = 0;
    virtual void findFavoriteOfficeMate() = 0;
    virtual void procrastinate() = 0;
};
```

Why does this feel so wrong?
Problem: Add new behaviors to a set of types

- Let us take a look at our Employee base class...

```cpp
class Employee {
public:
    ...
    virtual void updatePay() = 0;
    virtual void performJob() = 0;
    virtual void serialize() = 0;
    virtual void displayAvatar() = 0;
    virtual void printPerformanceReview() = 0;
    virtual void findFavoriteOfficeMate() = 0;
    virtual void procrastinate() = 0;
};
```

Why does this feel so wrong?
Solutions

- We need to find a better way
Solutions

- We need to find a better way

Employee* employee = ...  
...  
auto result = employee->foo(x, y, z);  
...
Solutions

- We need to find a better way

```cpp
Employee* employee = ...;
auto result = employee->foo(x, y, z);
```
Solutions

- We need to find a better way

```cpp
Employee* employee = [REDACTED];
auto result = employee->[REDACTED](x, y, z);
```

We want to be able to add new behaviors, so we should not need to know them.
Solutions

- We need to find a better way

Employee* employee = ...

auto result = employee->foo(x, y, z);

We also want possibly different behavior for different subtypes.
Solutions

- We need to find a better way
  - What are the tools at our disposal?
Solutions

- We need to find a better way
  - What are the tools at our disposal?
    - Classes
    - Polymorphism
Solutions

- We need to find a better way
  - What are the tools at our disposal?
    - Classes
    - Polymorphism
  - How can we use them to attack the problem?
Solutions

- **We need to find a better way**
  - What are the tools at our disposal?
    - Classes
    - Polymorphism
  - How can we use them to attack the problem?
    - Group related behaviors into classes
    - Invoke them when desired
Grouping Related Behavior

- How should we group related behaviors?

What does SRP dictate?
Grouping Related Behavior

- How should we group related behaviors?
  - Each offending method becomes a new class
Grouping Related Behavior

- How should we group related behaviors?
  - Each offending method becomes a new class

```cpp
class EmployeeSerializer {
public:
    void serialize(Manager &manager);
    void serialize(Underling &underling);
};

class PerformanceReviewPrinter {
public:
    void printReview(Manager &manager);
    void printReview(Underling &underling);
};
```
How Do We Invoke It?
How Do We Invoke It?

EmployeeSerializer serializer;
std::vector<Employee*> employees;

for (auto *employee : employees) {
    serializer.serialize(*employee);
}
EmployeeSerializer serializer;
std::vector<Employee*> employees;

for (auto *employee : employees) {
    serializer.serialize(*employee);
}
EmployeeSerializer serializer;
std::vector<Employee*> employees;

for (auto *employee : employees) {
    serializer.serialize(*employee);
}
How Do We Invoke It?

EmployeeSerializer serializer;
std::vector<Employee*> employees;

for (auto *employee : employees) {
    serializer.serialize(*employee);
}
How Do We Invoke It?

EmployeeSerializer serializer;
std::vector<Employee*> employees;

for (auto *employee : employees) {
    serializer.serialize(*employee);
}

No!

What is the core problem?
How Do We Invoke It?

- **Problem:**
  - We want to call a method based on *multiple dynamic types*

```java
serializer.serialize(*employee);
```
How Do We Invoke It?

- **Problem:**
  - We want to call a method based on multiple dynamic types

```java
serializer.serialize(*employee);
```

EmployeeSerializer
How Do We Invoke It?

- Problem:
  - We want to call a method based on multiple dynamic types

```java
serializer.serialize(employee);
```

EmployeeSerializer Manager/Underling
How Do We Invoke It?

- Problem:
  - We want to call a method based on multiple dynamic types
  - *Multiple Dispatch* (or double dispatch in this case)

```java
serializer.serialize(employee);
```

```
EmployeeSerializer  Manager/Underling
```
How Do We Invoke It?

• Problem:
  - We want to call a method based on multiple dynamic types
  - Multiple Dispatch (or double dispatch in this case)

```java
serializer.serialize(employee);
```

EmployeeSerializer  Manager/Underling

But we only know that employee is an Employee*
How Do We Invoke It?

- **Problem:**
  - We want to call a method based on multiple dynamic types
  - *Multiple Dispatch* (or double dispatch in this case)

```cpp
for (auto* employee : employees) {
    serializer.serialize(*employee);
}
```

But we only know that `employee` is an `Employee*`
How Do We Invoke It?

- **Problem:**
  - We want to call a method based on multiple dynamic types
  - *Multiple Dispatch* (or double dispatch in this case)

```java
serializer.serialize(employee);
```

But we only know that `employee` is an `Employee*`

How can we resolve the issue?
How Do We Invoke It?

- **Problem:**
  - We want to call a method based on multiple dynamic types
  - *Multiple Dispatch* (or double dispatch in this case)

- **Solution:**
  - The Visitor Pattern

```java
serializer.serialize(*employee);
```
How Do We Invoke It?

- **Problem:**
  - We want to call a method based on multiple dynamic types
  - *Multiple Dispatch* (or double dispatch in this case)

```
serializer.serialize(*employee);
```

- **Solution:**
  - The Visitor Pattern
  - Goal:

```
base->method(whatever);
```
How Do We Invoke It?

- **Problem:**
  - We want to call a method based on multiple dynamic types
  - *Multiple Dispatch* (or double dispatch in this case)

- **Solution:**
  - The Visitor Pattern
  - Goal:

```
serializer.serialize(*employee);
```

```
base->xxxxxx(xxx);
```

Invoke the correct behavior regardless of the dynamic type!
The Visitor Pattern

Abstract away the added behaviors:

class EmployeeSerializer : public Visitor {
public:
    void visit(Manager &manager) override;
    void visit(Underling &underling) override;
};
The Visitor Pattern

Abstract away the added behaviors:

class EmployeeSerializer : public Visitor {
    public:
        void visit(Manager &manager) override;
        void visit(Underling &underling) override;
};

Giving behaviors a common API allows us to use all behaviors in the same way
The Visitor Pattern

Change the original classes:

```cpp
class Employee {
public:
    virtual void accept(Visitor &v) = 0;
};

class Manager : public Employee {
    ...
    void accept(Visitor &v) override {
        v.visit(*this);
    }
};
```
The Visitor Pattern

Change the original classes:

class Employee {
public:
    virtual void accept(Visitor &v) = 0;
}
class Manager : public Employee {
    ...
    void accept(Visitor &v) override {
        v.visit(*this);
    }
};

The dynamic type of Employee is known!
Calls visit(Manager &manager) here.
The Visitor Pattern

Use the new behaviors through their classes:

```cpp
EmployeeSerializer serializer;
PerformanceReviewPrinter reviewer;
std::vector<Employee*> employees;

for (auto* employee : employees) {
    employee->accept(serializer);
    employee->accept(reviewer);
}
```
The Visitor Pattern

Use the new behaviors through their classes:

```cpp
EmployeeSerializer serializer;
PerformanceReviewPrinter reviewer;
std::vector<Employee*> employees;

for (auto* employee : employees) {
    employee->accept(serializer);
    employee->accept(reviewer);
}
```

What if we want a return value?
The Visitor Pattern

Use the new behaviors through their classes:

```cpp
std::vector<Visitor*> actions;
std::vector<Employee*> employees;
...
for (auto* employee : employees) {
    for (auto* action : actions) {
        employee->accept(*action);
    }
}
```
The Visitor Pattern

- A behavioral pattern
The Visitor Pattern

- A behavioral pattern
- Useful for adding new behaviors to a collection of related classes
The Visitor Pattern

- A behavioral pattern

- Useful for adding new behaviors to a collection of related classes
  - *It also keeps those behaviors isolated!*
The Visitor Pattern

- A behavioral pattern

- Useful for adding new behaviors to a collection of related classes
  - *It also keeps those behaviors isolated!*
  - Useful for designing APIs open to extension
The Visitor Pattern

- A behavioral pattern
- Useful for adding new behaviors to a collection of related classes
- But what are the downsides?
  - Can we overcome them?
Making tradeoffs

- **The visitor pattern**
  - makes adding new behaviors trivial
  - can leave adding new types challenging
Making tradeoffs

- The visitor pattern
  - makes adding new behaviors trivial
  - can leave adding new types challenging

- What if we expect adding new types to be more common?
  - A similar pattern called the *interpreter* emerges
  - Each behavior is just a method of the type involved
Making tradeoffs

- The visitor pattern
  - makes adding new behaviors trivial
  - can leave adding new types challenging

- What if we expect adding new types to be more common?
  - A similar pattern called the *interpreter* emerges
  - Each behavior is just a method of the type involved

- Choose between them by likelihood of change & maintainability
Making tradeoffs

- The visitor pattern
  - makes adding new behaviors trivial
  - can leave adding new types challenging

- What if we expect adding new types to be more common?
  - A similar pattern called the *interpreter* emerges
  - Each behavior is just a method of the type involved

- Choose between them by likelihood of change & maintainability

You can help or hurt an open/closed design
Making tradeoffs

• The visitor pattern
  – makes adding new behaviors trivial
  – can leave adding new types challenging

• What if we expect adding new types to be more common?
  – A similar pattern called the interpreter emerges
    – Each behavior is just a method of the type involved

• Choose between them by likelihood of change & maintainability

• Adding new types vs adding new behaviors is a common tension when designing maintainable software
  – This is classically known as the expression problem.
Designing Design Patterns

- Instead of memorizing them, you should be able to create them
Summary

- Careful software design focuses responsibilities & makes changes easier
Summary

- Careful software design focuses responsibilities & makes changes easier
- Polymorphism & composition help provide clear abstractions