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
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 change

  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)

worse  →  Content

better

... goto yourcode ...

... yourcode: ...

...
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

Better:

```c
int global = ...

... = global
```

Worse:

```c
global = ...

global = ...

... = global
```
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

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

- **Loose Coupling (connectivity)**
- **worse**
  - Content
  - Common global data
  - Subclassing
- **better**
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:
    virtual void foo() { bar(); }
    virtual void bar() {};
};
```

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

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

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

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

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

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

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

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

```java
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

```
Cat cat = new Cat;
...
deleter cat;
```

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

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

This is more insidious!
What is common in good designs?

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

\[ x = \text{foo}(1, 2) \]

```
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

![Comparison of good and bad designs]

VS
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)
2) Parametric polymorphism (e.g. via generics/templates)
3) Ad-hoc polymorphism (e.g. via 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) Ad-hoc polymorphism (e.g. via 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
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
List list = new ArrayList();
void foo(List& someList);
...
ArrayList list;
foo(list);
```

```cpp
List + add()
ArrayList + add()
```

Java

C++
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 $\varphi$ is true for the base, then $\varphi$ is true 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 $\phi$ is true for the base, then $\phi$ is true the derived
    - Arguments in the subtype may be more general
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
  - 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
    - Preconditions are not stronger

```
assert(x > 0) assert(x != 0)
```

<table>
<thead>
<tr>
<th>Base</th>
<th>Derived</th>
</tr>
</thead>
<tbody>
<tr>
<td>A foo(B b)</td>
<td>C foo(D d)</td>
</tr>
</tbody>
</table>
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 $\varphi$ 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
    - 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

Professor
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

Researcher

is-a

Professor
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

![Diagram showing relationships between Researcher, Teacher, and Professor](image-url)
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 \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

\begin{center}
\begin{tikzpicture}
    \node[draw] (Professor) {Professor} ;
    \node[draw, right of=Professor, xshift=1cm] (Researcher) {Researcher} ;
    \node[draw, below of=Professor, yshift=-1cm] (Teacher) {Teacher} ;
    \draw[->, above of=Professor, yshift=0.5cm] (Professor) -- (Researcher) node[midway, above] {has-a} ;
    \draw[->, below of=Professor, yshift=-0.5cm] (Professor) -- (Teacher) node[midway, above] {has-a} ;
\end{tikzpicture}
\end{center}
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

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

```cpp
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.
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.
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.

Does 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 `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?
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 \texttt{move()} and \texttt{speak()}
  - An \texttt{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.

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

Animal has-a Movement

Movement selects from the ways any Animal can move.
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*
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
  - 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
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
```
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
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 reimplemention 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 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
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 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
- Directly identifies & addresses risks of change in class design
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
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

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

Typescript

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

Python

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

C++

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

C++

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

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

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

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

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

**Python**
```
Parameters can sometimes be inferred.
```
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.
Parametric Polymorphism

- 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.

```c++
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 and 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;
        }
    }
    ...
}
```

```
template<typename C, typename V>
auto find(const C& c, const V& v) {
    for (auto i = begin(c), e = end(c); i != e; ++i) {
        if (*i == v) {
            return i;
        }
    }
    return end(c);
}
```
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
// Parametric Polymorphism

```cpp
template<typename C, typename V>
auto find(const C& c, const V& v) {
    for (auto i = begin(c), e = end(c); i != e; ++i) {
        if (*i == v) {
            return i;
        }
    }
    return end(c);
}
```

```cpp
void bigAlgorithm(...) {
    std::vector<int> c;
    ...
    auto element = find(c, v);
    ++*element;
    ...
}
```
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>
auto find(const C& c, const V& v) {
    for (auto i = begin(c), e = end(c); i != e; ++i) {
        if (*i == v) {
            return i;
        }
    }
    return end(c);
}
```

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

```cpp
void otherAlgorithm(...) {
    std::vector<string> d = ...;
    auto element = 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
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> { }
```

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
Specialized instances can sometimes be created

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 & ~0xFFFFFFFFFFFF8);
    }
    Value 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 & ~0xFFFFFFFF8);
    }
    Value 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.

```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"); }
};
```
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.

```java
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.

Have those of you familiar with Java seen this before?

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

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.

Have those of you familiar with Java seen this before?

```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.
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.

```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);
```

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

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

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

void bar(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.

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

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...

```
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...

```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
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);
```

```cpp
std::array<E,N> std::vector<E>
E[N]
```

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.
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
Type Traits

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

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

template<>
struct GraphTraits<SocialGraph> {
};
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

```cpp
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

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

```cpp
<functional>

```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.

```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`?
Type Traits

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

```cpp
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;
}
```

```cpp
#include <Cats.h>

namespace std {
    namespace 
        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
<functional>

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

<unordered_set>

```std::unordered_set`<class 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.
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");
    foo(*w);
}
```

Or better...

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

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

void synchronization() {
    std::mutex m;
    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?

```cpp
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

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

How would you express this in C or 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
disenum 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]
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 destructu"e.")
    },
    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]
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 modeled 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?

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

What are the forms of coupling that arise?
Problem: Separate Caller & Callee

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

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

What are the forms of coupling that arise?
Problem: Separate Caller & Callee

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

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

```cpp
class Work {
  // Information about work
  // ...

  Result doWork() { ... }
};
```
Problem: Separate Caller & Callee

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

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

auto result = worker.doWork();

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

class Work {
    virtual Result doWork() = 0;
};
```
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
  - Parametrizable via constructor
The Command Pattern

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

```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);
...
```

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

```cpp
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
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