CMPT 373
Software Development Methods

Visitors

Nick Sumner
wsumner@sfu.ca

Recall: Design Patterns

- Capture programming idioms (not solutions)

Recall: Design Patterns

- Capture programming idioms (not solutions)
- Exploit polymorphism in well understood ways

Recall: Design Patterns

- Capture programming idioms (not solutions)
- Exploit polymorphism in well understood ways
- 3 classic categories
  - **Creational** – provide flexibility in creating objects
  - **Structural** – compose classes to add new behavior
  - **Behavioral** – focus on communication between entities

Recall: Design Patterns

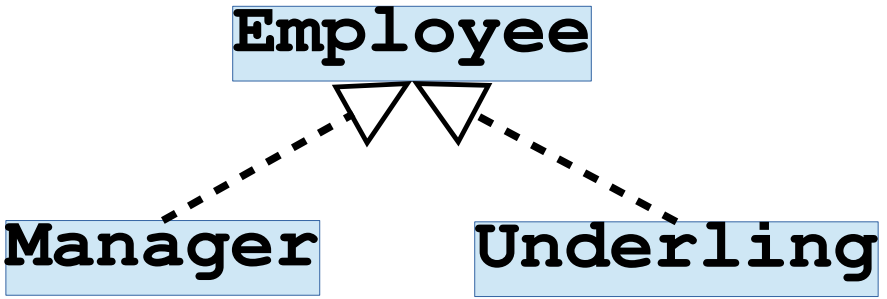
- Capture programming idioms (not solutions)
- Exploit polymorphism in well understood ways
- 3 classic categories
  - Creational – provide flexibility in creating objects
  - Structural – compose classes to add new behavior
  - Behavioral – focus on communication between entities
- We have seen **prototype**, **decorator**, **command**, …

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

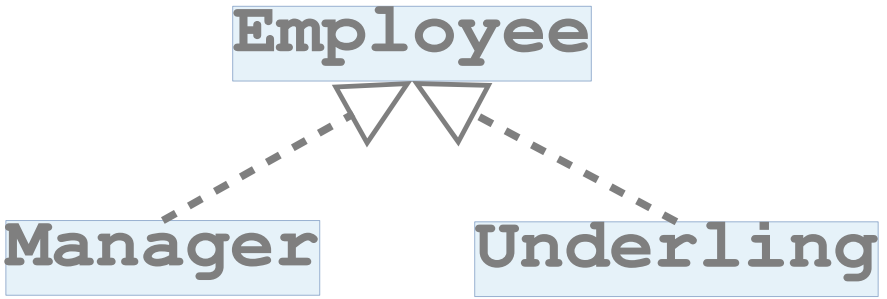
- Different classes can perform the same action differently



Problem: Add new behaviors to a set of types

- Different classes can perform the same action differently

```
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

```
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…

```
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…

```
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…

```
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
  - 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

```
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);
}
```

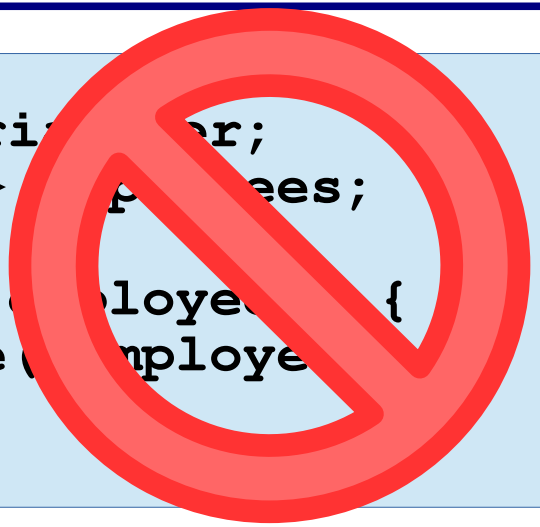
How Do We Invoke It?

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

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

Will this work? Why?

How Do We Invoke It?

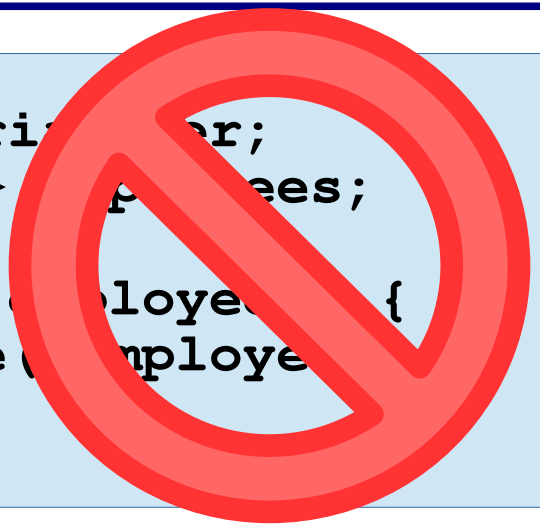
```
EmployeeSerializer seri    er;
std::vector<Employee*>     ees;

for (auto *employee :     loye    {
  serializer.serialize(    mploye
}
```



No!

How Do We Invoke It?

```
EmployeeSerializer seri    er;
std::vector<Employee*>     ees;

for (auto *employee :     loye    {
  serializer.serialize(    mploye
}
```

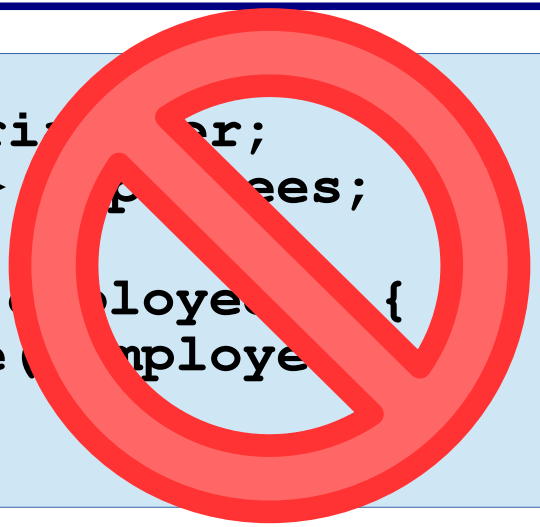


No!

What is the core problem?

How Do We Invoke It?

```
EmployeeSerializer seri    er;
std::vector<Employee*>     ees;

for (auto *employee :     loye    {
  serializer.serialize(    mploye
}
```



No!

What is the core problem?

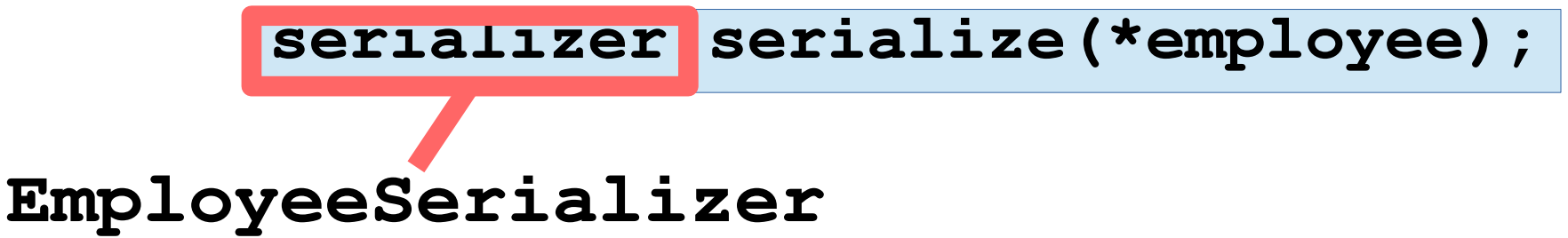
How Do We Invoke It?

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

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

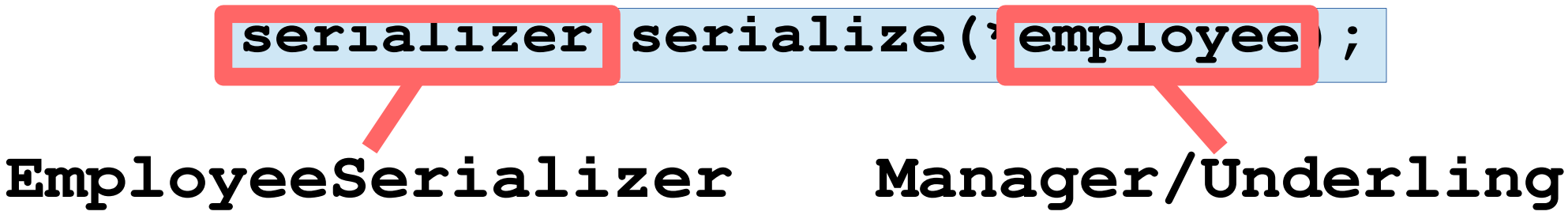
How Do We Invoke It?

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



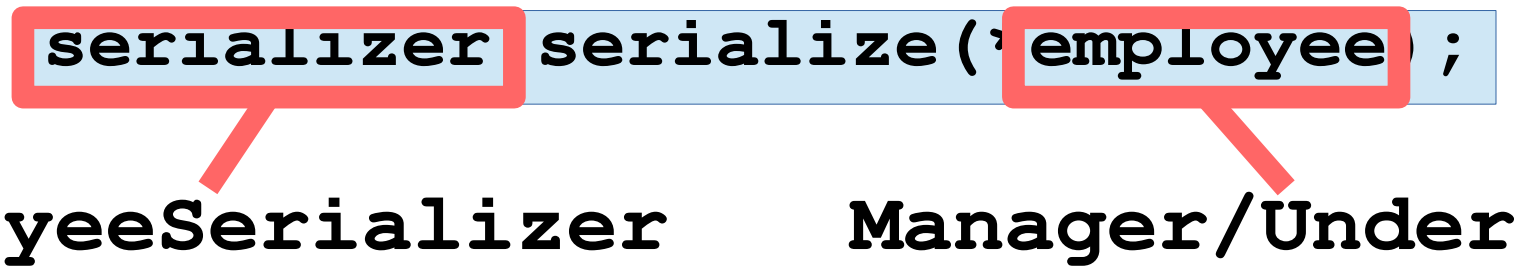
How Do We Invoke It?

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



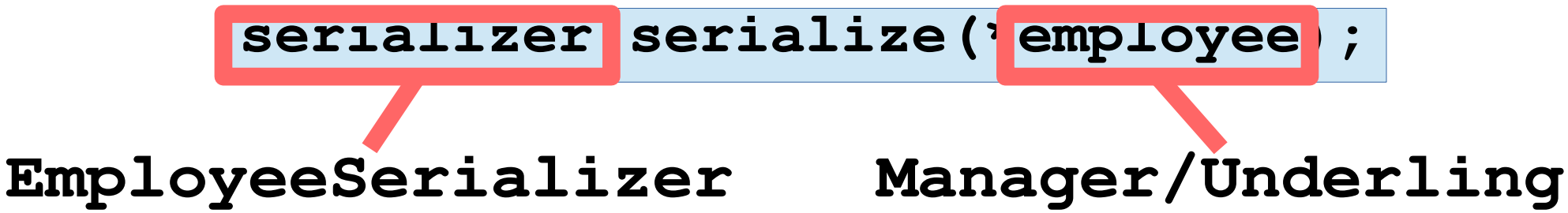
How Do We Invoke It?

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



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)



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)

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

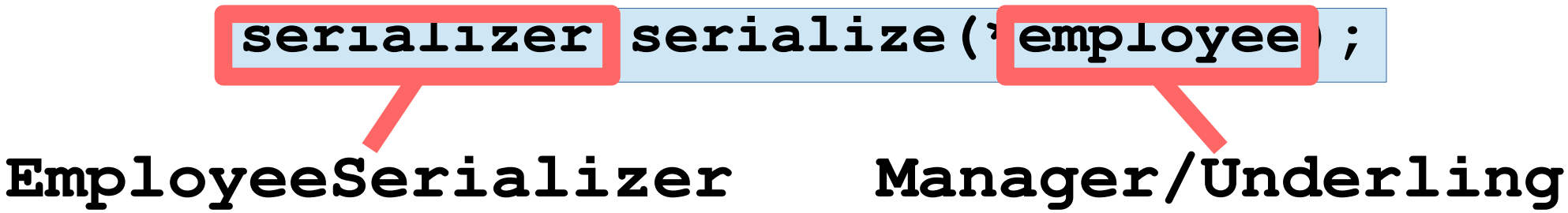
EmployeeSerializer **Manager/Underling**

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

```
for (auto* employee : employees) {
  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)



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)

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

- Solution:
  - The Visitor Pattern

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->[redacted]([redacted]);
```

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->[redacted](*[redacted]);
```

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:

```
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:

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

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

The Visitor Pattern

Use the new behaviors through their classes:

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

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

What if we want a return value?

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 (infinite set of new behaviors)

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 tradoffs

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

Making tradoffs

- 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 tradoffs

- 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 tradoffs

- 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 tradoffs

- 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.

Summary

- The visitor pattern enables adding new behaviors to a set of types

Summary

- The visitor pattern enables adding new behaviors to a set of types
- Types can assist in choosing behavior based on when/where the type is known

Summary

- The visitor pattern enables adding new behaviors to a set of types
- Types can assist in choosing behavior based on when/where the type is known
- Trade offs must still be managed
  - Over-engineering is a serious risk