Design Patterns:
The Visitor

Nick Sumner
wsumner@sfu.ca
Recall: Design Patterns

- Capture programming idioms
- Exploit polymorphism in well understood ways
Recall: Design Patterns

- Capture programming idioms
- Exploit polymorphism in well understood ways

3 primary 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
- Exploit polymorphism in well understood ways
- 3 primary 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, ...
A New Problem

- Different classes can perform the same action differently
A New Problem

- Different classes can perform the same action differently
A New Problem

- Different classes can perform the same action differently

Manager manager;
manager.updatePay();

Underling underling;
underling.updatePay();
A New Problem

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

• 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();
A New Problem

- 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
A New Problem

- 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`
A New Problem

- 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
```
A New Problem

- 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`
A New Problem

- 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
...

updatePay
serialize
printPerformanceReview
...
A New Problem

- 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!*
A New Problem

- 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?
A New Problem

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;
};
A New Problem

• 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?
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
  - 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);`
}
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);
How Do We Invoke It?

employees = new vector<Employee*>(3);

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

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

EmployeeSerializer
How Do We Invoke It?

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

```cpp
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);
```

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)

```
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);
```
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

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 \texttt{Employee} is known!
Calls \texttt{visit(Manager \&manager)} here.
The Visitor Pattern

Use the new behaviors through their classes:

```cpp
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:

```cpp
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
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?