Dynamic Binding

Object-oriented Programming

236703

Spring 2016
Reminder: Strict Inheritance

- Strict inheritance allows *extension*:
  - Additional operations
  - Additional structure elements
  - But no *overriding*

- Full conformance — Derived *is a* base, always
- No performance penalty — binding is known at compile time (i.e., *static*)
- Limited abstraction mechanism — can’t redefine behavior
  - No abstract classes, no interfaces etc.

- Real-world example?
  - C# default behavior
We now discuss non-strict inheritance

- AKA Inheritance

More powerful abstraction mechanism: a subclass is basically like the superclass, but more changes are allowed

- Operations implemented differently – overriding

Bear in mind:

- there is usually no way to redefine structure elements
- Interface may sometimes be modified in a safe way (covariant return, removing exceptions)
Strict vs. Non-strict

<table>
<thead>
<tr>
<th></th>
<th>Strict</th>
<th>Non-strict</th>
</tr>
</thead>
<tbody>
<tr>
<td>Forge</td>
<td>New</td>
<td>New</td>
</tr>
<tr>
<td>Mill</td>
<td>New (usually call old)</td>
<td>New (usually call old)</td>
</tr>
<tr>
<td>Structure</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>Protocol</td>
<td>Add</td>
<td>Add</td>
</tr>
<tr>
<td>Behavior</td>
<td>Added only</td>
<td>Added and overriding</td>
</tr>
</tbody>
</table>

Can the protocol be modified in awkward ways?
Reminder: Static Binding

- Binding – linking between messages and methods
- Static Binding (AKA Early Binding) – the compiler uses the type of variables to do the binding
  - Compile time feature
  - Link time (external symbols, static libraries)

```c
struct A {
    void f() {}
};

A a;
a.f();
```

```
... f_label:
...

... call f_label
```
Reminder: Dynamic Binding

- Dynamic Binding (AKA Late Binding) – the decision is made at run time based on the type of the actual values
  - Execution time feature
  - Used in both statically and dynamically typed languages
  - (related but different: dynamic libraries)

```ruby
Object subclass: #MyClass
  
  MyClass new f ...
```
Reminder: Static & Dynamic Types

Consider the call:
Employee* e = new Manager();
e->print();

What is the type of e?
- Static type: Employee
- Dynamic type: Manager

Which version of print will be called?
- C++: static binding by default (zero overhead)
- Java: dynamic binding by default
- C#: static binding by default (strict inheritance)
Static Typing Binding Challenge

- Statically typed languages rely on compile time work.
- Inclusion Polymorphism – The same entity may refer to objects of different classes, each of which has a different implementation of the same method.
- Inclusion Polymorphism + Overriding = Binding Challenge

```cpp
struct A {
    virtual void f() {}
};

A* a = g();
a->f();
```
Dynamic Binding and Static Typing

- **Static typing:** guarantee that a method exists
  - A variable of type T can contain only objects of type T or of type T', where T' is derived from T
  - A message to an object is legal only if its static type recognizes it
- **Dynamic binding:** the right method is selected
  - The method invoked for an object depends on its dynamic type
- **Static typing + Dynamic binding:** the right combination of safety and power
  - Examples: Eiffel, C++, Java, C#, etc.
struct X {
    virtual void f() { ... }
    void g() { ... }
};

struct Y : public X {
    void f() override { ... } // Override f() of X
    void g() { ... } // Hide g() of X
    void h() { ... }
};

X* x = new Y();

x->f(); // The right method
x->g(); // The wrong method
x->h(); // Compile time error – no guarantee that the method exists
Java Example

```java
public class X {
    public String toString() { return "X"; }
    public final int addThree(int n) { return n + 3; }
    public static int mul(int a, int b) { return a * b; }
}
public class Y extends X {
    public String toString() {
        return "Y" + super.toString();
    }
}

X x = new Y();
x.toString();
x.addThree(5);
x.mul(3, 9);
```

What's the binding of `super.toString()`? And this `toString()`?

What's the binding of a `final` method?

What's the binding of a `static` method? Does this call make sense?

And what's the binding of a `private` method?
Forms of Overriding – Replacement

Replacement: The new implementation of an operation replaces the implementation of the operation in the base class

- The only kind of overriding in earlier versions of Eiffel

Use case:

- Java’s Reader.read() picks a single character from input
- Overriding BufferedReader.read() picks a single character from an internal buffer, which holds chunks of input

Downsides: disallows code reuse, use case doesn’t justify language limitation

- Hence never used anymore
Forms of Overriding - Refinement

- **Refinement**: The new implementation of an operation refines the implementation of the operation in the base class.

- Overridden method can be called even though an overriding method exists.
  - Overriding method need only contain the additional logic – no duplication due to the replacement of the overridden method.
  - Java, C#, Squeak: only last overridden method can be called, and only by overriding class (why have this limitation?)
  - C++: any overridden function can be called by anyone...
Refinement Strategies

Given overriding and refinement, we have two “versions” of a method: the parent’s and the child’s

Who’s in charge?

- Alpha refinement – child’s method called first, internally calls parent’s method if it wants
- Beta refinement – Parent’s method called first, internally calls child’s method if it wants
Alpha Refinement and Subobjects

- Recall that each object of a subclass has a subobject of the superclass (recursively)
- Alpha refinement involves a forwarding the message to the subobject
- Message forwarding mechanisms:
  - `super` in Smalltalk, Java
  - `base` in C#
  - `Scope resolution operator` in C++ (Base::foo)

```cpp
void Derived::foo() override {
    ...
    Base::foo();
    ...
}
```

Why not super/base?
Alpha Refinement and Conformance

The overriding method decides whether to call the overridden method

- If overridden not called, semantics more likely to differ
  - Even if not from the get go, how can overriding method keep up if overridden method is changed?

```cpp
char Reader::read() {
    return getFromInput(1);
}
```

```cpp
char Reader::read() {
    readChars += 1;
    return getFromInput(1);
}
```

```cpp
char BufferedReader::read() {
    verifyBuffer();
    return getFromBuffer(1);
}
```
Beta Refinement

- Encourages *semantic* conformance
  - Overridden method decide if and when to call overriding method and if and when to do base class work
  - Alpha refinement only supports *syntactic* conformance
- Excludes replacement
  - Base class method always invoked
- Design challenge: can a base method deal with any overriding method?
  - *It’s hard to make predictions, especially about the future...*
Beta Refinement Example

- Beta, Simula and CLOS allow calling the overriding method using the keyword `inner`
- A C++ emulation:

```cpp
class Connection {
    virtual void innerSend() {}
public:
    void send() {
        loadMsg();
        innerSend();
        sendMsgToServer();
    }
};
```

```cpp
class SecureConnection : public Connection {
    void innerSend() {
        encryptMsg();
    }
};
```

Why should `innerSend` be private?

Why shouldn’t `send` be virtual?

Why should overriding `innerSend` be final?
Inner vs. Super

- **Flexibility:**
  - Super – maximal, can call any of the base’s methods (but do we want to call *any* of them?)
  - Inner – minimal, can only call the current method’s overriding method

- **Readability:**
  - Super usually much easier to understand

- **Binding:**
  - Super – always static
  - Inner – it’s complicated...
Pattern vs. Feature

- The C++ example is a pattern (similar to the Template Pattern)
- Patterns are fragile – rely on good will, discipline and documentation
- The C++ example is weaker than the language feature:
  - Language allows innerSend to be overridden
  - Further deriving requires innerInnerSend...
- Language features are designed to encourage quality code
  - Here – conformant overriding
const_cast

Just get this out of our way (irrelevant to this course)

► Purpose: cast away const
  ► (and volatile, but that’s even less related to OOP)

► Targets: pointers and references
  ► Otherwise we get a copy, which eliminates const anyway

► Run-time overhead: none – merely instructs the compiler to allow mutation of the target

► Use case:
  ► Breaking const-correctness and invoking UB 😞
reinterpret_cast

- **Purpose:** instruct the compiler to interpret a bunch of bits differently
  - Origin and target types need not be related
- **Targets:** mostly pointer and integral types
- **Run-time overhead:** none
- **Use case:**
  - Converting actual types to or from an opaque or general (void*, uintptr_t) type in an API
- **Should not be used on most cases**
static_cast

- **Purpose:** cast related types and numeric types
  - numeric conversions are well-defined (possibly changing bit patterns), type conversions are lightweight
- **Targets:** mostly pointers and numeric types
- **Run-time overhead:** low (no RTTI)
- **Use case:**
  - When type conversion is known to be legal
- **Shortcomings compared to dynamic_cast:**
  - Can’t down-cast from virtual base class
  - No indication whether cast was legal
dynamic_cast

- **Purpose:** cast to a different sub-object of a polymorphic type
  - Non-polymorphic types will yield a compiler error
  - Sub-object type usually derived from target’s type, but may be anywhere in the dynamic type’s inheritance graph
- **Targets:** pointers, references
- **Run-time overhead:** RTTI lookup, possibly noticeable
- **Use case:**
  - Down-cast, indicating if cast was legal or not
C-style cast

- In C++, does the first C++ cast that is legal within a well defined series of attempts
  - Possibly not what you were expecting – say remove the const due to a type typo
- Not self-documenting
- Use case:
  - You’re too lazy to type the long C++ cast 😞
Downcasting vs. Dynamic Binding

RTTI considered harmful:
- Order of classes in the if chains is significant
- Code must change whenever new class is added to the hierarchy

What’s the safer, simpler, cleaner solution?