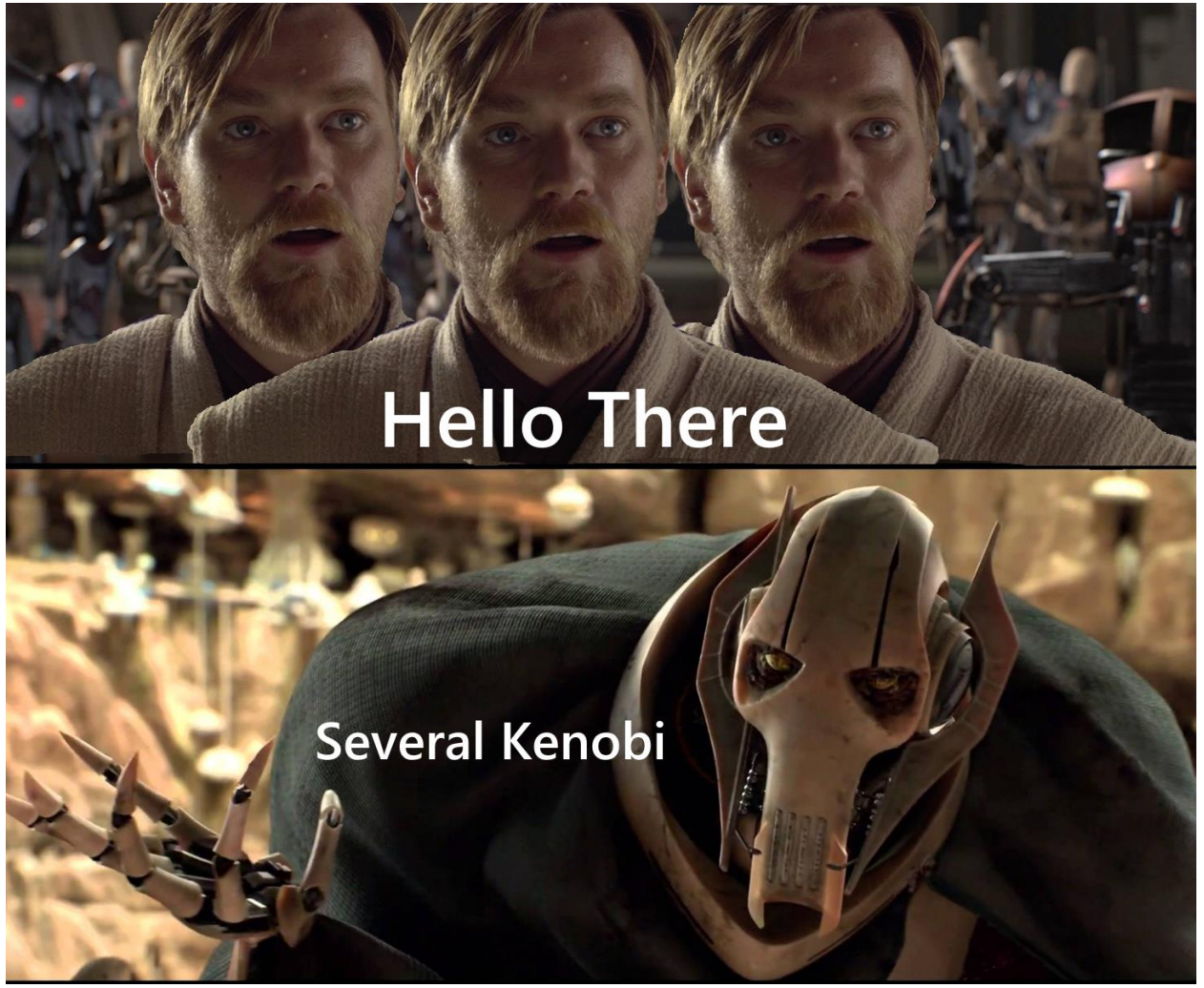
Multiple interface class inheritance and name-hiding

>How to select the proper method?_

Introduction



Requirements

- Interface/Abstract class to permit dynamic dispatch
- Finite set of types
- Templated Single Point of contact (due to templated clients)
- Type-Safety
- (Ab)use the compiler and generate stuff…
- DRY
- SOLID

An interface in C++

```cpp
class IInterface {
public:
  IInterface(IInterface const&other)            = delete;
  IInterface(IInterface &&other)                = delete;
  IInterface& operator=(IInterface const&other) = delete;
  IInterface& operator=(IInterface &&other)     = delete;

  virtual ~IInterface() = default;

  // Public API
  virtual void foo() = 0;

protected:
  IInterface() = default;
};
```

An interface is just a declaration of capabilities.

No construction, no copy, no move!

All methods are pure virtual declarations!

Basic Interface

```
template <typename TType>
class ITaskBackendPartDecl {
public:
  virtual uint8_t spawnCreationTask(typename TType::Descriptor const&) = 0;
};
```

ITaskBackendPartDecl<TType> will be inherited for each supported type, declaring a *spawnCreationTask*-method for a specific type's Descriptor-structure.

"Several Kenobis" - Supported Types

```
template <typename... TTypes>
class AbstractTaskBackend
  : public ITaskBackendPartDecl<TTypes>...
{
public:
  template <typename TType>
  uint8_t spawnCreationTask(typename TType::Descriptor const&desc) {
    // How to select the proper override?
  }
};
```

Inheriting the base-interface for each supported type makes all versions **visible** in the derived class' scope.

BEWARE: **Name-Hiding** due to consistency

The *single point of contact* method hides the base class' declarations…

Option 1 – Just call 'this'

```
template <typename TType>
uint8_t spawnCreationTask(typename TType::Descriptor const&desc) {
  // Try simple overload-resolution using the descriptor.
  // Fails, as TType is part of the nested-name-specifier and
  // TType::Descriptor is non-deduced scope
  // -> Deduction fails
  // -> Overload resolution fails
  return this->spawnCreationTask(desc);
}
```

All interface method's names are visible (no name hiding), so overload resolution for *TType::Descriptor* will take place, but:

nested-name-specifier

Option 2 – Qualified Name Selection

```
template <typename TType>
uint8_t spawnCreationTask(typename TType::Descriptor const&desc) {
  // Qualified Name Lookup in explicitly defined scope of ITaskBackendPartDecl<TType>.
  // Qualified name look up is never virtual.
  // Just finds a pure-virtual method --> Undefined Reference
  return this->ITaskBackendPartDecl<TType>::spawnCreationTask(desc);
}
```

Being explicit is good, as long as it doesn’t cancel the purpose…

The declaration is now properly selected, but the *most derived overrider* won’t be detected, as *dynamic dispatch* is deactivated.

Option 3 – static_cast + virtual call

```
template <typename TType>
uint8_t spawnCreationTask(typename TType::Descriptor const&desc) {
  // Type cast to desired interface and perform virtual function call
  // -> Selects the proper override
  return static_cast<ITaskBackendPartDecl<TType>*>(this)->spawnCreationTask(desc);
}
```

Above:

The combination of being explicit and dynamic dispatch!

Example Backend

```
struct TypeA { struct Descriptor {}; };
struct TypeB { struct Descriptor {}; };
struct TypeC { struct Descriptor {}; };

#define SupportedTypes TypeA, TypeB, TypeC

class ExampleTaskBackend
  : public AbstractTaskBackend<SupportedTypes>
{
private:
  uint8_t spawnCreationTask(TypeA::Descriptor const&);
  uint8_t spawnCreationTask(TypeB::Descriptor const&);
  uint8_t spawnCreationTask(TypeC::Descriptor const&);
};

uint8_t ExampleTaskBackend::spawnCreationTask(TypeA::Descriptor const&)
{
  return 1;
}
```

This implementation can be specialized however you want…

It can also be wrapped inside macros to reduce redundancy even more…
(Remind debuggability though…)

How to use it?

```
int main()
{
  Ptr<AbstractTaskBackend<SupportedTypes>> backend = MakePtr<ExampleTaskBackend>();

  TypeA::Descriptor descA={ };
  print(backend->spawnCreationTask<TypeA>(descA));
  TypeB::Descriptor descB={ };
  print(backend->spawnCreationTask<TypeB>(descB));
  TypeC::Descriptor descC={ };
  print(backend->spawnCreationTask<TypeC>(descC));

  system("PAUSE");
}
```

Simple usage example…

Trying to use any other type, e.g. TypeD, will prevent compilation very early.

The big question?

Is this really necessary?

Why not just declare all pure virtual methods in a single interface class?

DRY > No redundant declaration, rely on compiler generation

SOLID > Interface segregation, Dependency Inversion

SLOC > Keep SLOC small, no significant increase in complexity

And subjectively… I just love templates… ;)

Questions?

>Craft|Deploy_

Marc-Anton Boehm-von Thenen

Twitter: @DottiDeveloper
Github: BoneCrasher
StackOverflow: mabvt

craft-deploy.github.io/