Here Come UNO,
All Shiny and New

Agenda

- You Know UNO?
- Ugly UNO/Shiny UNO
- Multiple-Inheritance Interfaces
- Services
- Type-Safe Properties
- Parametric Polymorphism
- Increased Type Safety Improves Client Code
- Published APIs
- Q & A

You Know UNO?

- UNO is the component model underlying OpenOffice.org.
- The basic UNO concept is an *object*, which implements one or more *interfaces*, which in turn offer *methods* that can be *called* on the object. Clients navigate among the various interfaces of an object via `queryInterface`.
- UNO binds to various programming languages, both statically and dynamically typed ones (Java, C++, Python, OOo BASIC).

Ugly UNO

```
XComponentContext context = ...
XServiceIfc1 serviceIfc1 = (XServiceIfc1)
  UnoRuntime.queryInterface(
    XServiceIfc1.class,
    context.getServiceManager().
    createInstanceWithArgumentsAndContext(
      "foo.bar.SomeService",
      new Any[] { new Integer(10), "Whatever" },
      context));
serviceIfc1.fn1();
XServiceIfc2 serviceIfc2 = (XServiceIfc2)
  UnoRuntime.queryInterface(
    XServiceIfc2.class, serviceIfc1);
serviceIfc2.fn2();
```

Shiny UNO

```
XComponentContext context = ...
XService service = foo.bar.SomeService.create(
  context, 10, "Whatever");
service.fn1();
service.fn2();
```

```
interface XServiceIfc1 {
  void fn1();
};
interface XServiceIfc2 {
  void fn2();
};
service foo.bar.SomeService {
  interface XServiceIfc1;
  interface XServiceIfc2;
};
```

```
interface XServiceIfc1 {
  void fn1();
};
interface XServiceIfc2 {
  void fn2();
};
interface XService {
  interface XServiceIfc1;
  interface XServiceIfc2;
};
service foo.bar.SomeService:
  XService {
  create([in] long arg1,
         [in] string arg2);
};
```

Connecting to the Office

- Many applications want to access OOo via UNO. The code to do that used to be

```
XComponentContext local = Bootstrap.createInitialComponentContext();
XConnector connector = (XConnector) UnoRuntime.queryInterface(
  XConnector.class, local.getServiceManager().createInstanceWithContext(
    "com.sun.star.connection.Connector", local));
XBridgeFactory factory = (XBridgeFactory) UnoRuntime.queryInterface(
  XBridgeFactory.class, local.getServiceManager().createInstanceWithContext(
    "com.sun.star.bridge.BridgeFactory", local));
XConnection connection = XConnector.connect(
  "socket,host=localhost,port=12345");
XBridge bridge = factory.createBridge("", "urp", connection, null);
XComponentContext context = (XComponentContext) UnoRuntime.queryInterface(
  XComponentContext.class, bridge.getInstance
("StarOffice.ComponentContext"));
// make sure OOo -accept=... is running
```

- With the new bootstrap facility, that has

```
XComponentContext context =
  com.sun.star.comp.helper.Bootstrap.bootstrap();
```

Multiple-Inheritance Interfaces

- Allowing multiple-inheritance for interface types opened up a bunch of possibilities:
  - *API designers can group related interfaces together. The resulting super-interfaces can be passed around as method parameters etc.*
  - *Client code gets rid of `queryInterface` to navigate among related interfaces of an object.*
  - *Interface types can take over most of the roles service descriptions previously had; service descriptions focus on a single task now.*

Services of the Past

- In the past, UNOIDL service descriptions were merely documentation for various concepts:
  - *Some services are available at the global service manager (e.g., `com.sun.star.bridge.UnoUrlResolver`).*
  - *Other services are available through specific factories (e.g., `com.sun.star.bridge.Bridge` and `com.sun.star.bridge.XBridgeFactory`).*
  - *Yet other services merely represent abstract entities (e.g., `com.sun.star.document.OfficeDocument`).*
  - *A few services are just documentation for sequences of properties (e.g., `com.sun.star.document.MediaDescriptor`).*

Services of the Future

- A new-style service unambiguously expresses one thing: “Instances of this service are available at the global service manager.”
- A new-style service corresponds to exactly one interface type, so services are now better integrated with the type system:
  - *You obtain a service with its specific interface type.*
  - *You pass a service instance into and out of methods by its specific interface type. (This can also make method specifications more self-documenting.)*

Services Have Constructors

- Each new-style service has a constructor that creates an instance of the service's interface type (no more `queryInterface`):

```
XUnoUrlResolver resolver = UnoUrlResolver.create(context);
```

- Additionally, a new-style service can have specific constructors with arguments (no more `ANY`s):

```
XService service = foo.bar.SomeService.create(
  context, 10, "Whatever");
```

- Analogously, new-style singletons have getters.

Type-Safe Properties

- The properties of old-style services are manipulated via `XPropertySet` etc., which involves handling type-unsafe `ANY`s.
- Attributes of interface types (until now a shadowy feature) are the type-safe counterparts of properties. They just had to be improved a little:
  - *The getters and setters of attributes can raise specific exceptions.*
  - *New types (`Ambiguous<T>`, `Default<T>`, `Optional<T>`) can be used to model `maybeambiguous`, `maybedefault`, and `maybevoid` properties.*
  - *Attributes can be `bound`, to notify listeners about changes.*

Optional et al.

- An old-style `maybevoid` property can either have a value of a certain type *T*, or be void.
- This concept is modelled in a more type-safe way with an attribute of type `com.sun.star.beans.Optional<`*T*`>`:

```
x.optChar = new Optional<char>(true, 'a');
x.optFloat = new Optional<float>(false, 0.0f);
if (x.optChar.isPresent) {
  char c = x.optChar.Value;
}
if (x.optFloat.isPresent) {
  float f = x.optFloat.Value;
}
```

- `Optional` uses the new feature of *polymorphic struct types*.

Parametric Polymorphism

- A few notes:
  - *The feature is kept initially simple, to avoid unpredictable problems.*
  - *For now, only available for struct types, not for interface types (which, alas, means no polymorphic, homogeneous containers).*
  - *Polymorphic struct types are mapped to template classes in C++.*
  - *Polymorphic struct types are mapped to Java's `Object`-polymorphism (instances of type parameters are replaced by `java.lang.Object`) in Java 1.4, and to generic classes in Java 1.5.*

Increased Type Safety

- The main theme of the presented features is increased type safety:
  - *Multiple-inheritance interface types reduce need for `queryInterface`.*
  - *Extended interface type attributes, together with polymorphic struct types, reduce need for `ANY`.*
  - *Constructors of single-interface–based services take exactly typed arguments, and return references of exact type.*
  - *Getters of interface-based singletons return references of exact type.*
- All this leads to simpler, more natural client code.

Published Things

- The dilemma of the OOo API:
  - *On the one hand, the API should be stable across releases.*
  - *On the other hand, freezing an API too early often leads to a poor API.*
- The solution: UNOIDL entities can be marked as either published or unpublished.
  - *A published entity is guaranteed to remain unchanged in future releases. Most parts of the OOo API are published (so that they can be used by clients with confidence).*
  - *An unpublished entity is not yet mature, and still subject to change. New OOo features may start out unpublished.*

The OOo API

- How does all this affect the existing OOo API?
  - *To remain backwards compatible, the existing APIs are generally not affected by the new features.*
  - *Designers of new APIs are encouraged to use the new features, so they should become more and more common post OpenOffice.org 2.0.*
  - *A handful of existing services have been migrated to the new form, and they can be obtained through constructors (e.g.,* `com.sun.star.bridge.UnoUrlResolver`*).*

Q & A

Wenn zu perfekt,
liebe Gott böse!

—*Nam June Paik*