Today's Topics

1. About OSGi
2. Architecture
3. The CTK Plug-in
4. Programming Basics
5. Dealing with services
About OSGi

- The OSGi Alliance is a non-profit corporation founded in March 1999.
- More than 35 companies from various areas
- Roots in embedded systems
- The OSGi specification is at Release 4 with numerous implementations in Java
- Specification for the core framework and a compendium of service interfaces
A plug-in is started by the *Plugin Activator* class.

The Activator gets a *Plug-in Context* which represents the *Framework*.

Plug-in Context objects should not be shared.
The Framework provides a dynamic service model for communication between plug-ins.

Active plug-ins may (un)register 0 or more services with the Framework at any time.

A service registration is a published interface with optional registration properties.

Service references are obtained from the FW by interface and filter expressions.

The Framework publishes service lifecycle events.
The CTK Plug-in

- A plug-in is a shared library with additional meta-data and resources.
- It must provide a Plugin Activator class which is called by the Framework.
- The FW invokes the start method when the plug-in enters the ACTIVE state.
- The FW invokes the STOP method when the plug-in leaves the ACTIVE state.
The CTK Plug-in

Each plug-in receives a unique `ctkPluginContext` for accessing the FW.

```cpp
class MyActivator : public QObject, public ctkPluginActivator
{
    Q_OBJECT
    Q_INTERFACES(ctkPluginActivator)

public:
    void start(ctkPluginContext* context)
    {
        myPC = context;
    }

    void stop(ctkPluginContext* context);

private:
    ctkPluginContext* myPC;
};
```
Programming Basics
Providing a Service

• Services are registered with the FW through the Plug-in Context
• (Un)Registration may be done at any time

```java
void registerSomeService() {
    mySomeService = new SomeServiceImpl();
    ctkDictionary props;
    props.insert("myvalue", 20);
    mySR = myPC->registerService<SomeService>(someServiceImpl, props);
}

void unregisterSomeService() {
    mySR.unregister();
}
```
Consuming a Service

- Services are retrieved from the FW through the Plug-in Context
- The FW returns a ctkServiceReference object which can be kept for future ref.
- Consumers must unget the service ref.

```cpp
void consumeSomeService() {
    ctkServiceReference sr = myPC->getServiceReference<SomeService>();
    if (sr) {
        SomeService* si = myPC->getService<SomeService>(sr);
        if (si) {
            // ...
            myPC->ungetService(sr);
        }
    }
}
```
Using Service Listeners

- Service listeners can be (un)registered
- A filter can be specified

```cpp
class A : public QObject {
    Q_OBJECT

    slots:
    void someServiceListener(const ctkServiceEvent& event) { ... }

    public:
    void registerServiceListener() {
        myPC->connectServiceListener(this, "someServiceListener", "filterExpr");
    }

    private:
    ctkPluginContext* myPC;
};
```
Using ctkServiceFactory

- Allows customized service instances
- The Framework caches service instances

```cpp
struct MyServiceFactory : public ctkServiceFactory {

    QObject* getService(QSharedPointer<ctkPlugin> plugin, ctkServiceRegistration reg) {
        return new SomeServiceImpl(plugin->getSymbolicName()); }

    void ungetService(QSharedPointer<ctkPlugin> plugin, ctkServiceRegistration reg, QObject* service) {
        delete service; }
};

void A::registerServiceFactory() {
    myServiceFactory = new MyServiceFactory();
    myPC->registerService<SomeService>(myServiceFactory);
}
```
Using ctkServiceTracker

- Convenience class making life easier
- The tracker holds all currently available services

```cpp
class B {

private:
    ctkServiceTracker<SomeService*> myServiceTracker;

public:
    B(ctkPluginContext* context) :
        myServiceTracker(context) { }

    void useSomeService() {
        SomeService* ss = myServiceTracker.getService();
        if (ss) { ... }
    }
};
```
Using Filters

- Service lookups and events can be constrained by the use of filters
- Filters are defined in LDAP query syntax

```cpp
try {
    QList<ctkServiceReference> refs = 
        myPC->getServiceReferences<SomeService>
            ("(&(myvalue>10)(myvalue<30))");
    foreach (ctkServiceReference sr, refs) {
        ...
    }
} catch (const std::invalid_argument& e) {
    // filter expression cannot be parsed
}
```
Questions?