Meta-Programming in KDE

The technology behind KConfig XT and friends

Cornelius Schumacher

The KDE Project
Overview

- What is Meta-Programming?
- Flavors of Meta-Programming
- Code Generators in KDE
  - libkabc
  - KConfig XT
  - kxml_compiler
- libkode
- Other Code Generators
  - Torque
  - gSOAP
- Conclusion
What is Meta-Programming?

Definitions:

- The art of programming programs that read, transform, or write other programs
- Automated Programming
- Creating program code automatically from meta descriptions rather than programming directly in a programming language
Flavors of Meta-Programming

- Assemblers, Compilers - High-level language code is transformed into lower-level language code.

- C++ Templates - Generic programming by describing repeating C++ code in generalized form.

- Aspect Weavers - Additional code described by aspects is injected into code (Aspect-oriented programming).

- Code Generators - Generating source code based on specific input data or commands.
Code Generators used in KDE

- moc (Qt Meta Object Compiler)
- uic (Qt Designer)
- dcopidl (DCOP stubs and skeletons)
- kdewidgets (KDE plugins for Qt Designer)
- kapptemplate (kdesdk)
- umbrello (Code Generation from UML)
- kdebindings (Language bindings)
- makeaddressee (libkabc)
- kconfig_compiler (KConfig XT)
- kxml_compiler
Generation of Addressee class which provides access to all fields of a contact.

- C++ Template Files
- Text file describing fields
- Generator script "makeaddressee" (Perl)
- Output: Addressee class
### libkabc Meta Sources

#### C++ Template (addressee.src.h)

```cpp
/**
 * Return translated label for uid field.
 */
static QString uidLabel();

/**
 * Set name fields by parsing the given string and trying to associate the parts of the string with according fields. This function should probably be a bit more clever.
 */
void setNameFromString( const QString &);
```

#### Control File (entrylist)

```
# This file describes the fields of an address book entry.
#
# The following comma-separated fields are used:
#
# Control: A generates accessor functions.
# L generates a static function for returning a translated label
# F generates a Field id and object for generic field handling
# E generate an equality test in Address::operator==().
# Field Name : A descriptive name which is shown to the user.
# Type : C++ type of field.
# Identifier : A string used in code as variable name etc.
# Field Category : Categories the field belongs to (see Field::FieldCategory).
# Output function: Function used to convert type to string for debug output (optional)

ALE,name,QString,name

ALFE,formatted name,QString,formattedName,Frequent
```
libkabc Generated Code

Generated C++ code (addressee.h)

/**
   * Return translated label for uid field.
   */
static QString uidLabel();

/**
   * Set name.
   */
void setName( const QString &name );

/**
   * Return name.
   */
QString name() const;

/**
   * Return translated label for name field.
   */
static QString nameLabel();

/**
   * Set formatted name.
   */
void setFormattedName( const QString &formattedName );

/**
   * Return formatted name.
   */
QString formattedName() const;

/**
   * Return translated label for formattedName field.
   */
static QString formattedNameLabel();

(...
**libkabc Discussion**

**Benefits:**
- No more need to write error-prone repetitive code
- Type-safe API for all fields
- Strong typing, errors can be found at compile-time, not at run-time
- Consistency is enforced (field names, labels, API docs, etc.)
- Field list easily extendable

**Problems:**
- Increased complexity of build process
- People change generated code instead of sources.
- API docs are not at usual place
Lessons from libkabc approach

- Saves work and improves quality of code.
- Works around limitations of C++ (type-safety can’t be ensured in generic way, C++ identifier aren’t accessible to program at run-time).
- Crude ad-hoc approach for special case.
- Increased complexity and non-standard meta descriptions might confuse developers.
- Template approach could be done by more generic system.
KDE 3.2 introduced KConfig XT (extended technology)

- Abstract description of configuration options in XML
- Code generator for translating XML files to C++ code
- Application has convenient and type-safe access to configuration options
- Loading and saving is handled in the background
- Generic access to configuration options including meta information
- Automatic connection of GUI designer generated dialogs to configuration backend
KConfig XT Code Generation

- XML description of configuration options (.kcfg)
- Code generation options from separate file (.desktop-style)
- kconfig_compiler creates C++ code for classes encapsulating configuration information
KConfig XT Generated Code

KConfigSkeleton
- readConfig()
- writeConfig()
- addItemBool()
- addItemString()
- ...

KConfigSkeletonItem
- name
- label
- group
- whatThis
- key

template < T >
KConfigSkeletonGenericItem
- setValue( T )
- T value()
- defaultValue

MyConfig
- myOption1
- myOption2
- myOption3

ItemBool

ItemString

...
KConfig XT: XML -> C++

Control File (kontact.kcfgc):

```xml
<?xml version="1.0" encoding="UTF−8"?>
<!DOCTYPE kcfg SYSTEM "http://www.kde.org/standards/kcfg/1.0/kcfg.dtd">
<kcfg>
  <kcfgfile name="kontactrc"/>
  ...
</kcfg>
```

Generate C++ code (prefs.h):

```cpp
namespace Kontakt {
  class Prefs : public KConfigSkeleton {
    private:
    ...
    Prefs() : KConfigSkeleton( "kontactrc" ) { ... }
    ItemString *activePluginItem() {
      return mActivePluginItem;
    }
    QStringList mActivePlugins;
  }
public:
  static Prefs *self();
  ~Prefs();
  ** Set ActivePlugin */
  static void setActivePlugin( const QString &v ) {
    if ( !self()->isImmutable( "ActivePlugin" ) )
      self()->mActivePlugin = v;
  }
  ** Get ActivePlugin */
  static QString activePlugin()
  {
    return self()->mActivePlugin;
  }
  ** Get Item object for ActivePlugin */
  ItemString *activePluginItem()
  {
    return mActivePluginItem;
  }
};
```
Benefits of Code Generation

- All definitions are at one place
- Type-safe interface to config options
- Eliminates potential errors caused by inconsistent config keys or default values.
- Less code to be written.
- Meta-data for config options at run-time (labels, whatsthis, info for kconfigeditor)
- Cleaner config files (default values aren’t written)
- More extensible
- Better KIOSK integration (immutability etc.)
- Easier to add GUIs.
Discussion of kconfig_compiler

- Increases complexity of build system
- Designer for .kcfg files (kcfgcreator), XML code is easy to create
- kconfig_compiler internally is an ugly piece of software
- kconfig_compiler itself is easy to test.
- Interesting experience to work on kconfig_compiler. Thinking very abstract. Simple changes can have big effects.
Configuration Wizards

- Based on KConfig XT XML meta data
- Additional rules for propagation of configuration values
- Extendable by custom code
- Dialog for setting options to propagate, optionally including views for the rules and preview of changes
- Used for setting up groupware access etc.
<kcfg>

<kcfgfile name="kolabrc"/>

<group name="General">
    <entry name="User" type="String">
        <label>Kolab user name</label>
        <default>\</default>
    </entry>
</group>

<group name="Constants">
    <entry name="EnableFreeBusy">
        <default>true</default>
    </entry>
</group>

<propagation source="kolabrc/Constants/EnableFreeBusy"
             target="korganizerrc/FreeBusy/FreeBusyPublishAuto" />

<propagation source="kolabrc/General/User"
             target="korganizerrc/FreeBusy/FreeBusyPublishUser" />

</kcfg>
Screenshot Configuration Wizard
Config Propagation Potential

- Simple configuration for specific purposes including logic and know-how of configuration options
- Fine-grained configuration is unaffected
- User levels (home users, former Windows users, enterprise users)
- Meta information is available
  - Show changes (overview page)
  - Record changes as transactions
  - Undo of configuration changes
  - Indicate in GUI which options are set by wizards
**kxml_compiler**

- Generate C++ classes representing XML data
- Source: RelaxNG scheme (maybe converted from DTD)
- Generator: kxml_compiler (C++ program using code generation library libkode)
- Output: Classes representing XML data, including parser for corresponding XML data files and output as XML (serializing/deserializing)
Generated Classes

XMLTag
metadata

Element1
attributes

Element2
attributes

Element3
attributes

MySpecial
myFunction()
myAttribute

Element4
attributes
Discussion

Benefits:

- No hand-written code necessary for parsing XML.
- Parsing code can be optimized for specific scheme.
- Validating parser.
- Enable persistence or streaming of objects.
- Meta data can be used for example to create editor or viewer GUIs.

Problems:

- Not all XML schemes can easily be transferred to class representations.
- Working on kxml_comiler itself is challenging because it introduces an additional level of abstraction.
Feature Plan - DTD

```xml
<!ELEMENT features (category+)>  
<!ELEMENT category (feature|category)*>  
<!ATTLIST category name CDATA #REQUIRED>  
<!ELEMENT feature (summary?, responsible*)>  
<!ATTLIST feature status (inprogress|todo|done) "todo"  
  target CDATA #REQUIRED>  
<!ELEMENT responsible EMPTY>  
<!ATTLIST responsible name CDATA #IMPLIED  
  email CDATA #IMPLIED>  
<!ELEMENT summary (#PCDATA|i|a|b|em|strong|br)*>  
<!ELEMENT i (#PCDATA)>  
<!ELEMENT b (#PCDATA)>  
<!ELEMENT em (#PCDATA)>  
<!ELEMENT strong (#PCDATA)>  
<!ELEMENT br EMPTY>  
<!ELEMENT a (#PCDATA)>  
<!ATTLIST a href CDATA #IMPLIED>  
<!ATTLIST a title CDATA #IMPLIED> 
```
Feature Plan - Relax NG Scheme

```xml
<?xml version="1.0" encoding="UTF-8"?>
<grammar xmlns:a="http://relaxng.org/ns/compatibility/annotations/1.0"
    xmlns="http://relaxng.org/ns/structure/1.0">
  <define name="features">
    <element name="features">
      <ref name="attlist.features" />
      <oneOrMore>
        <ref name="category" />
      </oneOrMore>
    </element>
  </define>
  <define name="attlist.features" combine="interleave">
    <empty/>
  </define>
  <define name="category">
    <element name="category">
      <ref name="attlist.category" />
      <zeroOrMore>
        <choice>
          <ref name="feature" />
          <ref name="category" />
        </choice>
      </zeroOrMore>
    </element>
  </define>
  <define name="attlist.category" combine="interleave">
```

KDE Developer Conference 2004 – p.25
Feature Plan - Generated Code

- XML Elements -> C++ Classes
- Child Elements -> Aggregations
- Attributes -> Properties
- Parsing of XML files to C++ objects

Features
- parse()
- parseFile()

Category
- parse()
- name

Feature
- parse()
- status
- target
- summary

Responsible
- parse()
- name
- email
bool ResourceFeaturePlan::isFeatureEnabled() {
    if (!features) {
        return false;
    } else {
        const auto categories = features->categoryList();
        bool todo = isEnabled( feature );
        if (!todo) {
            return false;
        } else {
            int i = 0;
            auto category = categories.at( i++ );
            while ( !category->empty() ) {
                int j = 0;
                auto feature = features->categoryList().at( j++ );
                while ( !feature->empty() ) {
                    if ( feature->isEnabled() ) {
                        return true;
                    } else {
                        j++;
                    }
                }
                i++;
            }
            return false;
        }
    }
    return true;
}

void ResourceFeaturePlan::insertCategories( CategoryList &categories, Todo *parent ) {
    auto category = categories.begin();
    while ( !category->empty() ) {
        auto todo = new Todo;
        todo->setSummary( category->name() );
        todo->setCategory( parent );
        m_categories.insert( todo->setCategory());
        for ( auto feature : features ) {
            if ( feature->isEnabled() ) {
                todo->setFeature( feature );
            } else {
                todo->setFeature();
            }
        }
    }
}
Screenshot KOrganizer
<table>
<thead>
<tr>
<th>Lines of Code</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>DTD</td>
<td>25</td>
</tr>
<tr>
<td>Relax NG Scheme</td>
<td>151</td>
</tr>
<tr>
<td>kcfg File</td>
<td>18</td>
</tr>
<tr>
<td>Generated XML Handling Code</td>
<td>242</td>
</tr>
<tr>
<td>Generated Configuration Code</td>
<td>101</td>
</tr>
<tr>
<td>Boilerplate Code in KResource</td>
<td>330</td>
</tr>
<tr>
<td>Functional Code in KResource</td>
<td>63</td>
</tr>
<tr>
<td>Total Code</td>
<td>736</td>
</tr>
<tr>
<td>Generated Code</td>
<td>343 (47 %)</td>
</tr>
</tbody>
</table>
libkode

- Library for supporting code generation driven by C++ programs.
- Representations for classes, functions, files, headers, code blocks etc.
- Classes for creating C++ files from the code representation
- Styles for customizing appearance code generation
- Semi-automatic handling of indentation, includes etc.
- Application "kode" for generation of program templates, e.g. header and implementation files including license header, optional singleton code, etc.
Torque

Example from another world

- Java library for generating object-relational mappings for database access
- Data-base scheme described in XML
- Generates classes representing data base elements of relational database
- Customizable by inheriting from generated base classes
- Automatic creation of data base tables
gSOAP

- Generator Tools for Coding SOAP/XML Web Services in C and C++
- Takes WSDL definitions and generates SOAP bindings from them
- Generates stubs and skeletons for convenient strong-typed client and server implementations
- Fast and efficient because gSOAP uses streaming XML parsing techniques
- Saves a lot of work when developing applications making use of SOAP
- Doesn’t seamlessly integrate in KDE code (strings and containers, event loop)
Novell Groupwise KResource

[Image of a calendar software interface with tasks and appointments]

KDE Developer Conference 2004 – p.33
Conclusion

- Meta-Programming is a widely used technique.
- Code generation is a powerful tool.
- Central pieces of KDE like KConfig XT make heavy use of code generation.
- New code generator ‘kxml_compiler’ for generating C++ code from RelaxNG schemes representing XML data.
- Code generation helper library ‘libkode’.
- Applications of meta-programming: KResources for XML feature plan and Novell Groupwise access.
- Think of meta-programming techniques like code-generation and use them where possible.