COSC345 Week 20

Reuse

19 August 2014

Richard A. O'Keefe

Traditional lifecycle

Gather requirements.

Construct specifications.

Build code || tests || documentation.

Test.

Deliver.

Maintain

Basic Reuse-aware lifecycle

Gather requirements.

Construct specifications.

Find reusable component(s).

Adapt reusable component(s).

Build code || tests || documentation.

Test.

Deliver.

Maintain.

Update reused component(s).

Smarter Reuse-aware lifecycle

Gather requirements.
Seek suitable reusable component(s).
Revise requirements around what you found.
Construct specifications.
Find more reusable component(s).
Adapt reusable component(s).
Build code || tests || documentation.
Test.
Deliver.
Maintain.
Update reused component(s).

Why reuse?

Improve quality (code already tested)

* but you still have to test it

Reduce costs (code already written)

* but you still have to adapt it

Reduce time to market (code already written)

* but you still have to find and learn it

Improve portability

* but you must find something portable to reuse

Problems of reuse

Poor quality (you get what you pay for)

Poor maintenance (if you are lucky)

Orphaning (no maintenance at all)

Coupling to original context

You still have to do your own tests

It's not your fault but it is your problem!

Mechanisms for reuse

Reuse via ld

Reuse via text editor

Reuse via macro processor (cpp, m4, *etc*)

Reuse via inheritance

Reuse via generics/templates (Haskell, Ada, C++ *etc*)

Reuse via aspects (AspectJ, AspectS, *etc*)

Reuse via generator

Reuse via ld

How we use the math library in C (`cc ... -lm`)

Components are functions, abstract data objects, or ADTs

Components have data parameters

Components may have function parameters (Fortran, C, *etc*)

Components may not have type parameters.

Components cannot be renamed.

Call backs cannot be renamed.

Reuse via text editor

Common technique

"Low technology"; easy to do

Breaks link between original and revised code

Bug duplication

Reuse via macro processor

Like reuse via text editor,

but *programmed* text transformations.

Regenerate derived version when original changes.

Old technology, effective, doesn't duplicate bugs.

Messy, can be error prone.

Reuse via generics/templates

Like reuse via macro processor

but compiler knows what is going on

semantic transformation done by compiler,

not arbitrary text transformations.

Ada: late 70s.

Consistent with strong static type checking.

Reuse via generator

Metaprogramming

With 20% of the work, design high level notation

and generator to low level language (*e.g.*, Java)

generating 80% of the code.

Derivation process reused.

Levels of reuse

Language and operating system

Function libraries (old, successful)

Class libraries

CORBA-style components

Frameworks (MacApp, MFC, "applet")

Domain-specific frameworks (SAP)

Architectures

The success of reuse

Market growth

$\rightarrow$ C, TeX, Perl, R, Java, ...

$*$ "You can't reuse it if you can't use it."

The internet

$\rightarrow$ `comp.sources`, `netlib`, `statlib`, `archie`, CTAN, CPAN, CRAN, web searching

$*$ "You can't reuse it if you can't find it."

Development for reuse

Low coupling & high cohesion essential

ADTs should be "algebraically complete"

Names should be generic, not application-specific

Need test cases & documentation

It's not reusable until it has been reused

If it's a binary release it's not portable

Resources

Web search engines.

Repositories such as CPAN, CTAN, www.r-project.org, Jungerl, CodeHaus.

m4, antique but still worth knowing; GNU auto-configuration uses it.

AWK and other scripting languages for writing generators.

DSL tools like JetBrains MPS http://www.jetbrains.com/mps/