ConfigurableFactory
update
+ a few related topics

Martin Woudstra (UMass)

A-team meeting

ATLAS Software & Computing Workshop

April 7, 2011

Reminder: the issue

- The problem:
  - Not all configuration dependencies are setup explicitly
    - Example: ToolA uses ToolB
    - the python that configures ToolA should make sure that ToolB is configured, e.g. by calling the corresponding python.
      Often not done. Assumes someone else already did it
  - Makes the configuration fragile
- The solution:
  - Every tool/service/algorithm configuration should explicitly call the configuration setup for all the tools/services it uses
  - The ConfigurableFactory is a way to streamline and automate this

Setting up the dependencies

- The dependencies should be setup in C++ by
  - using ToolHandles and ServiceHandles (not ToolSvc directly)
  - declaring all *Handles as Properties (with declareProperty)
- Then dependencies are known on the python side
  - ‘just’ loop over all *Handles of a configurable and call the corresponding python code that does their configuration
- How to know which python config to call for which tool?
  - leverage Gaudi feature: tool/service/algorithm instance names have to be unique in the job
  - Make a database that links instance names to configuration code

Auto-dependencies: ingredients

- Python code that configures tools/services/algorithms
  - Already existing, possibly not in a form directly usable by Factory
- Database that links instance names to config code
  - To be added. First example version exists in MuonRecExample.
- A central manager: ConfigurableFactory
  - Working prototype exists (in MuonRecExample)

- Possible bootstrap
  - Get all algorithms in topSequence from the ConfigurableFactory
    - should trigger configuration of all used tools and services in the job !

ConfigurableFactory

- Creates instances on-demand (by name)
- Loops over all Tool/ServiceHandles of the instance and configures the used tools/services (by name)
  - if tool/service name is found in ToolSvc/ServiceMgr: just use it
  - else if found in the configuration database: configure and use it
    - recursive: configures everything that is used
    - automatically choses between public and private (for tools)
  - else make default configurable (optional, off by default)
- Loads the configuration databases
  - all modules with name pattern <package>/<package>ConfigDb.py
- Single instance in module CfgGetter
  - accessor functions at module level: CfgGettter.getPublicTool(), getPrivateTool(), getService(), getAlgorithm()

Python configuration code

- Different ways to declare to the ConfigurableFactory
  - an auto-generated configurable class name or C++ class name
  - a class derived from an auto-generated Configurable
  - a ‘factory function’ which returns a Configurable instance
  - a Configurable instance (not preferred but possible)
- Can use Flags as before
  - May need less flags and less ‘ifs’: do not need to decide which tools to configure and which not
- Configure job dependent dependencies runtime
  - by simply setting the correct instance name in the *Handle property
- Add dependencies missing or not possible in C++
  - by adding explicit calls to CfgGetter.getPublicTool() etc.
  - by importing other configuration modules
  - by including other jobOptions files

Configuration Database

- Python module with only CfgGetter.addTool(), addService(), addAlgorithm() statements to fill the Factory
  - string based to minimize dependencies on configuration modules
- Static (hand-coded) or maybe one day auto-generated at build time
- No run-time dependencies!
  - Use of Flags (JobProperties) not allowed inside database module
    - Flags can be used in configuration code, either in the factory function, in the class __init__ or even at the module level
- No loading of any configuration code!
  - no imports of modules with configuration code
  - no inclusion of jobOptions files

‘Database’ example

https://svnweb.cern.ch/trac/atlasoff/browser/MuonSpectrometer/MuonReconstruction/MuonRecExample/trunk/python/MuonRecExampleConfigDb.py

```
from CfgGetter import addTool, addToolClone, addService


addTool( "MuonRecExample.MuonRecTools.MuonClusterOnTrackCreator", "MuonClusterOnTrackCreator" )
addTool("Muon::MuonClusterOnTrackCreator", "CscBroadClusterOnTrackCreator",
        DoFixedErrorCscEta = True, FixedErrorCscEta = 5. )


addTool( "MuonRecExample.MuonRecTools.MdtDriftCircleOnTrackCreator", "MdtDriftCircleOnTrackCreator" )
addTool( "MuonRecExample.MuonRecTools.MdtDriftCircleOnTrackCreator", "MdtTubeHitOnTrackCreator",
         CreateTubeHit = True )


addToolClone( "MdtDriftCircleOnTrackCreator", "MdtDriftCircleOnTrackCreatorUpdateError_s2_c05",
              DoFixedError = False, FixedError = 0.2 )


addTool("TrkExTools.AtlasExtrapolator.AtlasExtrapolator","AtlasExtrapolator")


addService( "MuonRecExample.MuonRecTools.MboySvcConfig","MboySvc" )
```

Factory function

Instance name

C++ class name

some properties

Already declared instance name (‘original’)

Instance name (‘clone’)

Configurable instance

Class derived from auto-gen Configurable class

Database modules: where

- Two possible strategies
  - Every package that has configuration code has a configuration database module
    - could probably be auto-generated at build time (specify signatures)
    - always up-to-date
  - Database modules are concentrated in a few packages (MuonRecExample, InDetRecExample, ...)
    - needs to be hand-coded, as they probably include config code from other packages (true for MuonRecExample)
    - could make a tool that generates a database to start from
    - needs manual update each time a new piece of configuration code is added
- Merging
  - The database modules (hand-coded or auto-generated) could be merged into one file per project (a-la-CfgMgr) => faster reading
  - To be done at build time

Python modules vs. include files

- All configuration code declared to the ConfigurableFactory must be defined in python modules, not in jobOptions include files
  - include files could be made to work, but not very clean because then include files need to be called from within a python module
    - Possible but not recommended.
    - if multiple tools/services/algorithms are setup in one include file, then it may get included multiple times (unless include.block() is used)
- changing jobOptions file into python module is easy
  - just move the file from share/ to python/
    - May need to add import of ToolSvc, ServiceMgr, include, ...
  - Backwards compat simple: the new jobOptions file imports all needed global variables from the new module:
    from MyPackage.MyBrandNewModule import myOldVar1,var2,var3

Benefits of the scheme

- User doesn’t need to know where a tool/service is configured, just get it from the Factory:
  - from AthenaCommon.CfgGetter import getPublicTool
  - theTool = getPublicTool(“MyFavouriteToolName”)
- If consistently used, for any given job it automatically configures everything that is used and nothing more

TODO

- Move ConfigurableFactory class and instance (CfgGetter.py)
  from MuonRecExample to AthenaCommon
  - request: maintenance/further development taken over by A-team (I can and probably will still contribute, but much less)
- Other domains should try it out
- Muons: move all tools/services/algorithms to the factory (not yet fully done)

Different topic... Multiple defaults

- C++ defaults are not sufficient
  - needed configuration only known at runtime,
    - depends on: type of input, type of output, trigger on/off, subdetectors on/off, etc.
- “User” defaults:
  - a correct configuration for a specific job, without end-user tweaks
  - provided by tool/service/algorithm configuration experts
  - different config for different use cases
- I extended the Configurable interface with functions to set the UserDefaults
  - Example use on next page
  - Implemented (for now) in classes ConfiguredBase(Meta)
  - Could be moved to class Configurable (if desired & A-team agrees)

```
class DCMathSegmentMaker(CfgMgr.Muon__DCMathSegmentMaker,ConfiguredBase):
    __slots__ = ()

    def __init__(self,name = 'DCMathSegmentMaker',**kwargs):
        self.applyUserDefaults(kwargs,name)
        super(DCMathSegmentMaker,self).__init__(name,**kwargs)

DCMathSegmentMaker.setDefaultProperties(
    MdtSegmentFinder = "MdtMathSegmentFinder",
    RefitSegment = True,
    AssumePointingPhi = beamFlags.beamType() != 'cosmics',
    OutputFittedT0 = True,
    DCFitProvider = "MdtSegmentT0Fitter" )

if (beamFlags.beamType() == 'singlebeam' or beamFlags.beamType() == 'cosmics'):
    DCMathSegmentMaker.setDefaultProperties(
        SinAngleCut           = 0.9 ,
        AddUnassociatedPhiHits = True ,
        RecoverBadRpcCabling   = True,
        CurvedErrorScaling     = False,
        UsePreciseError        = True,
        PreciseErrorScale      = 2. )

if muonRecFlags.doSegmentT0Fit():
    DCMathSegmentMaker.setDefaultProperties(
        MdtCreatorT0 = "MdtDriftCircleOnTrackCreatorAdjustableT0Moore",
        TofTool      = "AdjustableT0ToolMoore" )
# end of class DCMathSegmentMaker
```

And some more topics...

- Proposal for extra functions for Configurable
  - def getProperty(configurable,property):
    - get property value: the set value, user default or C++ default
  - def getPropertyType(configurable,property):
    - get type of property (taken from C++ default)
  - def hasPropertyBeenSet(configurable,property):
    - return whether property has been set (in python)
    - for non-Handles: same as hasattr(configurable,property)
    - for Handles: not the same, because hasattr() will always return True because of auto-retrieval of Handles
- Auto-retrieval of Handles
  - also triggered when printing configurable and when getting defaults
    - may change state of configurable => not desirable
  - Proposal: do not change state of configurable in these cases

For discussion

- Support (temporarily) jobOptions files as source for configuration code for the ConfigurableFactory?
- Where do we store the configurable databases?
  - Per package or more central (few packages)?
- Do we want to auto-generate the databases?
  - Only possible if all properties are set in the configuration code
  - If hand-coded, then properties can also be set in the database module
- Add extented ‘user defaults’ interface?
- Add extra Configurable functions?