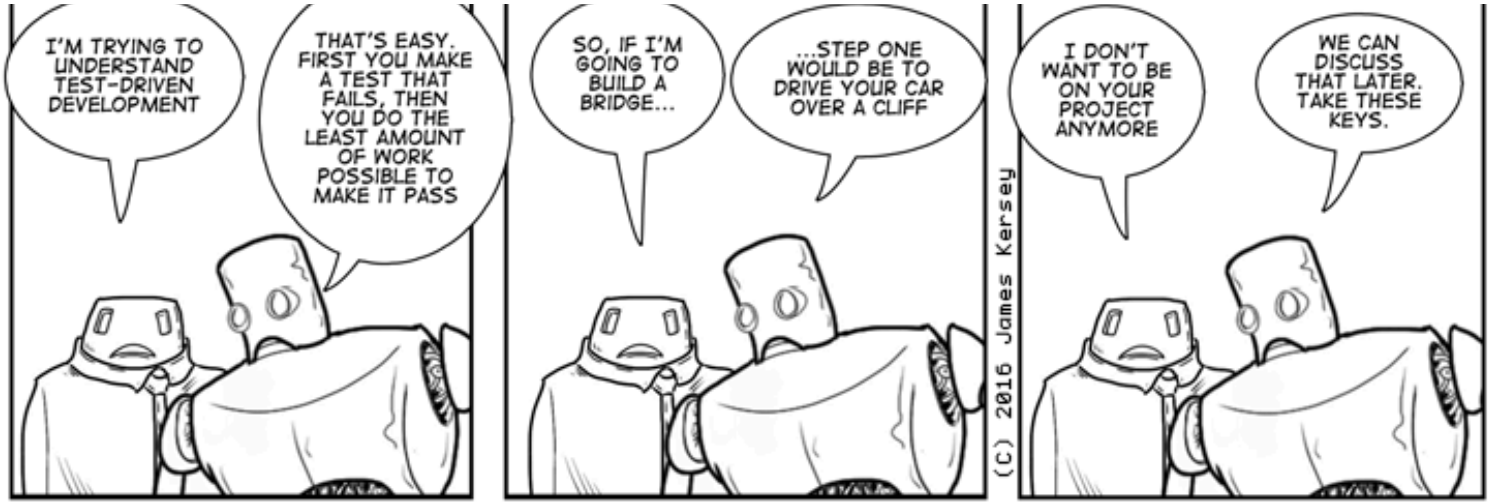
Application Of Unit-Test for GENIE

James Jones
Harvard university
15 August 2016

Test-Driven Development

- Test cases written before program
- Requirements turned into test cases
- Write program to pass test cases
- Continue the cycle

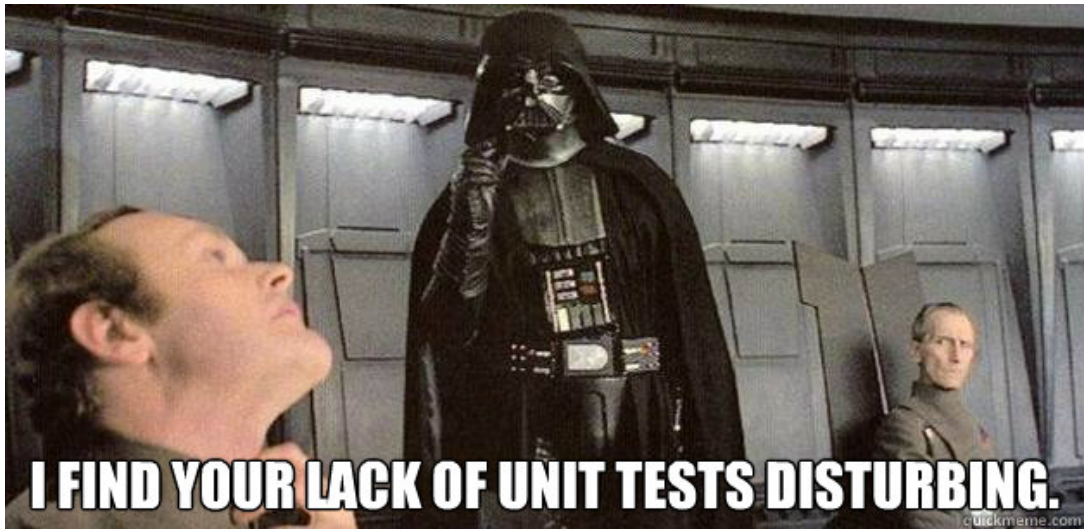


Unit-Test

- Testing method used to test units of code.
- Usually used to test functions and procedures
  - Numbers, return value, booleans
- Allows for testing of functionality of code
- Unit-Test Frameworks

Unit-Test

- Testing method used to test units of code.
- Usually used to test functions and procedures
  - Numbers, return value, booleans
- Allows for testing of functionality of code
- Unit-Test Frameworks



Why Use Unit Tests

- Test functions to see if they work
  - Give the test something you know the answer too
- Allows one to test variables
- Protects from errors made during sharing and editing

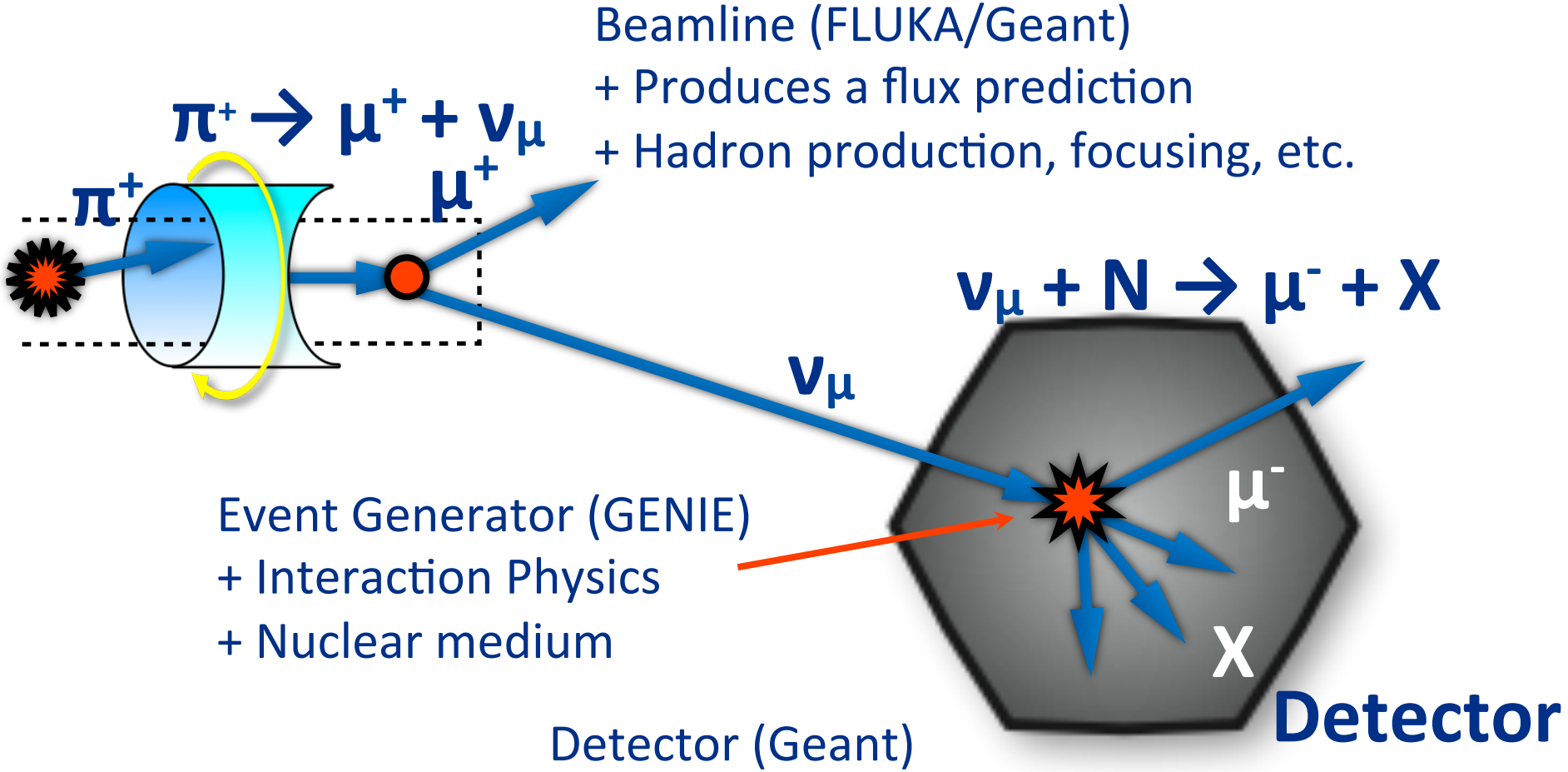
"If we learn from our mistakes, shouldn't I try to make as many mistakes as possible?"

GENIE

- Neutrino Monte Carlo
  - simulate detailed experimental setups involving neutrinos
- 200,000 lines of C++
  - Developed and Supported by scientist all over the world
    - (covering low energy reactor experiments, solar, supernova, meson decay at rest, accelerator-based experiments, and all the way through PeV+ cosmic experiments)

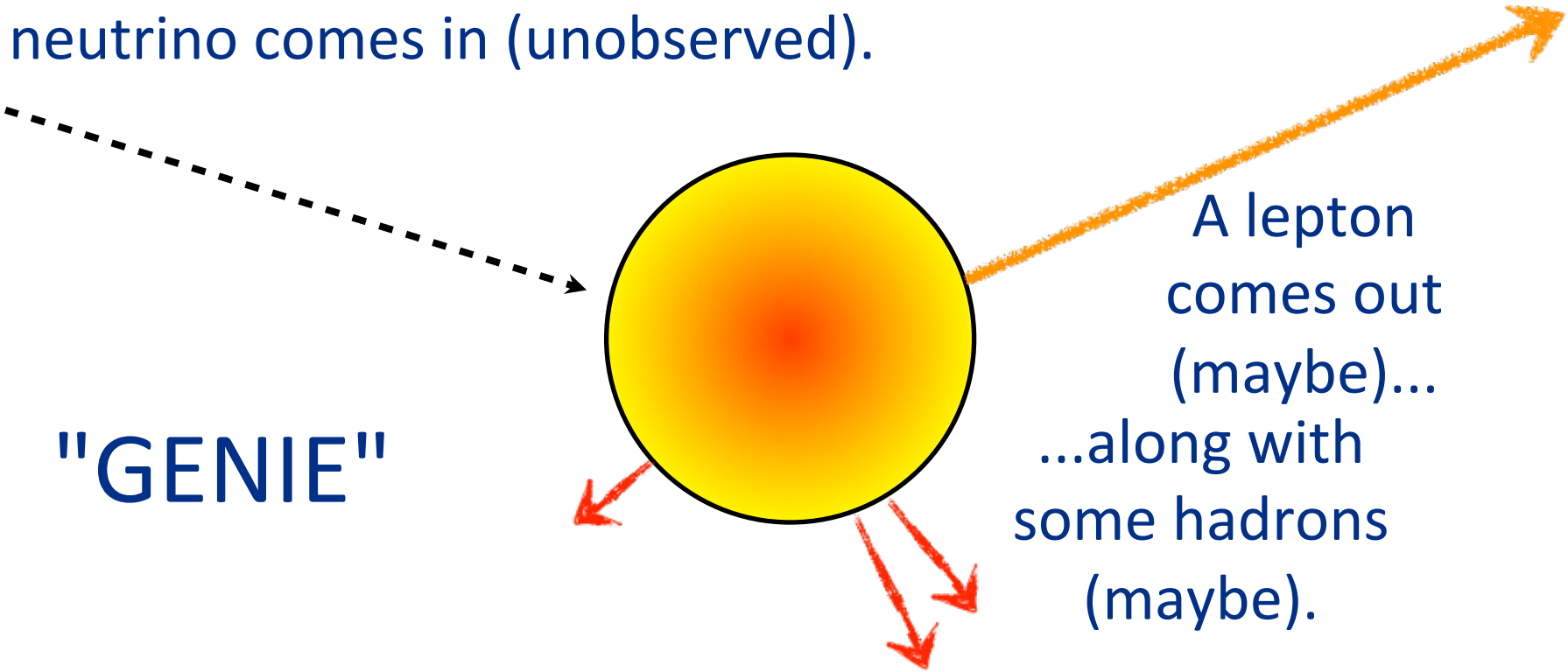
Neutrino Simulations:

A Three-Part Software Stack



The Problem

A neutrino comes in (unobserved).



"GENIE"

A lepton comes out (maybe)...

...along with some hadrons (maybe).

What was the neutrino's energy?

GENIE at FNAL

- GENIE is the primary event generator for:
  - ArgoNeut
  - SBND
  - DUNE
  - MicroBooNE
  - MINERvA
  - NOvA
- GENIE is being considered for special studies by MINOS and MiniBooNE (they use previous generation software for their main generators).

BOOST

- A set of libraries for C++
  - Supports number generation, linear algebra and unit testing
- Chosen framework for GENIE unit-test
  - + Easy to create tests
  - + Handles crashes well

Suites and cases

- BOOST_AUTO_TEST_SUITE
  - Used to organize unit test
- BOOST_AUTO_TEST_CASE
  - Organizes more specific tests in a Suite

```
BOOST_AUTO_TEST_SUITE(Physics)
BOOST_AUTOO_TEST_CASE(Acceleration)
{
        // Here we would test acceleration based funcions.
}
BOOST_AUTO_TEST_SUITE(Force)
{
        // here we  would test functions having to do with force.
}
BOOST_AUTO_TEST_SUITE_END
```

Simple BOOST Functions

- BOOST_CHECK
  - Makes sure a function returns to right value
- BOOST_REQUIRE
  - Same as BOOST_CHECK but stops test if false
- BOOST_ERROR
  - Uses to warn of an error
- BOOST_FAIL
  - Same as BOOST_ERROR but stops test when used

Using BOOST

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Hello
#include <boost/test/unit_test.hpp>
#include <iostream>
#include <iterator>
#include "fun.h"

using namespace std;
// int divide(int x, int y);
int add( int i, int j)
{
        return i + j;
}
string word(string x)
{
         return x;
}
BOOST_AUTO_TEST_SUITE(Math)
numbers numbrs;
        BOOST_AUTO_TEST_CASE(Addition)
        {
                BOOST_CHECK(add(1, 1) < 3);
                BOOST_REQUIRE(add(4, 3) > 6 && add(4,3) < 4);
                BOOST_CHECK(add(3, 2) == 5);
                }
        BOOST_AUTO_TEST_CASE(Division)
        {
                BOOST_CHECK(divide(4, 3) == 2
        }
BOOST_AUTO_TEST_SUITE_END()
```

Fermilab

Using BOOST

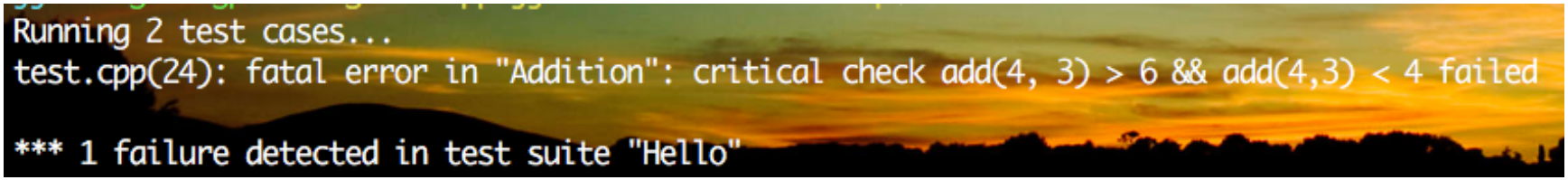
```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Hello
#include <boost/test/unit_test.hpp>
#include <iostream>
#include <iterator>
#include "fun.h"

using namespace std;
// int divide(int x, int y);
int add( int i, int j)
{
        return i + j;
}
string word(string x)
{
         return x;
}
BOOST_AUTO_TEST_SUITE(Math)
numbers numbrs;
        BOOST_AUTO_TEST_CASE(Addition)
        {
                BOOST_CHECK(add(1, 1) < 3);
                BOOST_REQUIRE(add(4, 3) > 6 && add(4,3) < 4);
                BOOST_CHECK(add(3, 2) == 5);
                }
        BOOST_AUTO_TEST_CASE(Division)
        {
                BOOST_CHECK(divide(4, 3) == 2
        }
BOOST_AUTO_TEST_SUITE_END()
```

Fermilab

James Bad Pun Counter: 1

```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Hello
#include <boost/test/unit_test.hpp>
#include <iostream>
#include <iterator>
#include "fun.h"

using namespace std;
// int divide(int x, int y);
int add( int i, int j)
{
        return i + j;
}
string word(string x)
{
         return x;
}
BOOST_AUTO_TEST_SUITE(Math)
numbers numbrs;
        BOOST_AUTO_TEST_CASE(Addition)
        {
                BOOST_CHECK(add(1, 1) < 3);
                BOOST_REQUIRE(add(4, 3) > 6 && add(4,3) < 4);
                BOOST_CHECK(add(3, 2) == 5);
                }
        BOOST_AUTO_TEST_CASE(Division)
        {
                BOOST_CHECK(divide(4, 3) == 2
        }
BOOST_AUTO_TEST_SUITE_END()

BOOST_AUTO_TEST_SUITE(Words)
        BOOST_AUTO_TEST_CASE(thesaurus)
        {
                BOOST_CHECK(word("delta") == "delta");
                if (word("pancakes") != "pancakes")
                {
                BOOST_ERROR("Something is Wwrong");
                }
        }
BOOST_AUTO_TEST_SUITE_END()
```

Running the Test



Booleans

- A data type with only two possible values
  - True (1)
  - False (0)

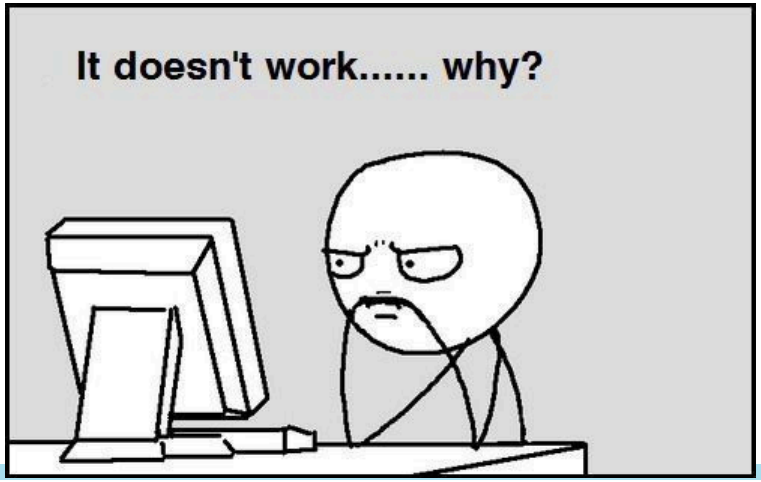
```
#define BOOST_TEST_DYN_LINK
#define BOOST_TEST_MODULE Genie
#include <boost/test/unit_test.hpp>
#include <iostream>
#include <iterator>
#include "LlewellynSmith/LwlynSmithQELCCPXSec.h"
#include "Interaction/Interaction.h"
#include "Interaction/KPhaseSpace.h"
#include "Interaction/XclsTag.h"
#include "Interaction/Kinematics.h"
#include "PDG/PDGUtils.h"


using namespace std;
namespace genie{
BOOST_AUTO_TEST_SUITE(True or False)
        BOOST_AUTO_TEST_CASE(Mr.T)
        {
                BOOST_CHECK(PityTheFool() = "True");
                BOOST_REQUIRE(MohawkGameStrong() = "True");
        }
}
```

Applying to GENIE

- Different than normal test-driven development
  - GENIE already exists
- Have to apply unit-test to it.
- More difficult
  - Working with already made classes
  - Need to access the correct pointers
  - Find the function that depends on the function the depends on the function

Applying to GENIE

- Different than normal test-driven development
  - GENIE already exists
- Have to apply unit-test to it.
- More difficult
  - Working with already made classes
  - Need to access the correct pointers
  - Find the function that depends on the function the depends on the function



Testing XSEC

- Attempted to test a cross section in the LlewellynSmith class

```
BOOST_AUTO_TEST_SUITE(Lwlyn)
// Build things from class Interaction for testing
LwlynSmithQELCCPXSec * lwlyn = new LwlynSmithQELCCPXSec();
Interaction * inter = new Interaction();
KinePhaseSpace_t kps = kPSQ2fE;
// int A nd Z represent carbon
int A = 12;
int Z = 6;
int probe_pdgc = 14;
int target_pdgc = pdg::IonPdgCode(A,Z);
// int for GeV
double Ev = 3;
//Setting up TLorentzVector
TLorentzVector nu_p4(0., 0., Ev, Ev); //px, py, pz, E
ProcessInfo proc_info(kScQuasiElastic, kIntWeakCC);
BOOST_AUTO_TEST_CASE(doubles)
{
inter->InitStatePtr()->SetPdgs(target_pdgc, probe_pdgc);
inter->InitStatePtr()->SetProbeP4(nu_p4);
std::cout << "The XSec is " << lwlyn->XSec(inter, kps) << std::endl;
BOOST_CHECK(lwlyn->XSec(inter, kps) == 5.3365984441833672E-11);


}
BOOST_AUTO_TEST_SUITE_END()
}
```

```cpp
//____________________________________________________________________________
double LwlynSmithQELCCPXSec::XSec(
    const Interaction * interaction, KinePhaseSpace_t kps) const
{
  if(! this -> ValidProcess    (interaction) ) return 0.;
  if(! this -> ValidKinematics (interaction) ) return 0.;

  // Get kinematics & init-state parameters
  const Kinematics &   kinematics = interaction -> Kine();
  const InitialState & init_state = interaction -> InitState();
  const Target & target = init_state.Tgt();

  double E  = init_state.ProbeE(kRfHitNucRest);
  double E2 = TMath::Power(E,2);
  double ml = interaction->FSPrimLepton()->Mass();
  double M  = target.HitNucMass();
  double q2 = kinematics.q2();

  // One of the xsec terms changes sign for antineutrinos
  bool is_neutrino = pdg::IsNeutrino(init_state.ProbePdg());
  int sign = (is_neutrino) ? -1 : 1;

  // Calculate the QEL form factors
  fFormFactors.Calculate(interaction);

  double F1V   = fFormFactors.F1V();
  double xiF2V = fFormFactors.xiF2V();
  double FA    = fFormFactors.FA();
  double Fp    = fFormFactors.Fp();

#ifdef __GENIE_LOW_LEVEL_MESG_ENABLED__
  LOG("LwlynSmith", pDEBUG) << "\n" << fFormFactors;
#endif

  // Calculate auxiliary parameters
  double ml2     = TMath::Power(ml,    2);
  double M2      = TMath::Power(M,     2);
  double M4      = TMath::Power(M2,    2);
  double FA2     = TMath::Power(FA,    2);
  double Fp2     = TMath::Power(Fp,    2);
  double F1V2    = TMath::Power(F1V,   2);
  double xiF2V2  = TMath::Power(xiF2V, 2);
  double Gfactor = M2*kGF2*fCos8c2 / (8*kPi*E2);
  double s_u     = 4*E*M + q2 - ml2;
  double q2_M2   = q2/M2;

  // Compute free nucleon differential cross section
  double A = (0.25*(ml2-q2)/M2) * (
             (4-q2_M2)*FA2 - (4+q2_M2)*F1V2 - q2_M2*xiF2V2*(1+0.25*q2_M2)
             -4*q2_M2*F1V*xiF2V - (ml2/M2)*(
              (F1V2+xiF2V2+2*F1V*xiF2V)+(FA2+4*Fp2+4*FA*Fp)+(q2_M2-4)*Fp2));
  double B = -1 * q2_M2 * FA*(F1V+xiF2V);
  double C = 0.25*(FA2 + F1V2 - 0.25*q2_M2*xiF2V2);

  double xsec = Gfactor * (A + sign*B*s_u/M2 + C*s_u*s_u/M4);

  double xsec = Gfactor * (A + sign*B*s_u/M2 + C*s_u*s_u/M4);

#ifdef __GENIE_LOW_LEVEL_MESG_ENABLED__
  LOG("LwlynSmith", pDEBUG)
     << "dXSec[QEL]/dQ2 [FreeN](E = "<< E << ", Q2 = "<< -q2 << ") = "<< xsec;
  LOG("LwlynSmith", pDEBUG)
                << "A(Q2) = " << A << ", B(Q2) = " << B << ", C(Q2) = " << C;
#endif

  //----- The algorithm computes dxsec/dQ2
  //      Check whether variable tranformation is needed
  if(kps!=kPSQ2fE) {
    double J = utils::kinematics::Jacobian(interaction,kPSQ2fE,kps);

#ifdef __GENIE_LOW_LEVEL_MESG_ENABLED__
    LOG("LwlynSmith", pDEBUG)
     << "Jacobian for transformation to: "
                << KinePhaseSpace::AsString(kps) << ", J = " << J;
#endif
    xsec *= J;
  }

  //----- if requested return the free nucleon xsec even for input nuclear tgt
  if( interaction->TestBit(kIAssumeFreeNucleon) ) return xsec;

  //----- compute nuclear suppression factor
  //      (R(Q2) is adapted from NeuGEN - see comments therein)
  double R = nuclear::NuclQELXSecSuppression("Default", 0.5, interaction);

  //----- number of scattering centers in the target
  int nucpdgc = target.HitNucPdg();
  int NNucl = (pdg::IsProton(nucpdgc)) ? target.Z() : target.N();

#ifdef __GENIE_LOW_LEVEL_MESG_ENABLED__
  LOG("LwlynSmith", pDEBUG)
      << "Nuclear suppression factor R(Q2) = " << R << ", NNucl = " << NNucl;
#endif

  xsec *= (R*NNucl); // nuclear xsec

  return xsec;
}
//
```

Testing XSEC

- Testing a cross section in the LlewellynSmith class

```
BOOST_AUTO_TEST_SUITE(Lwlyn)
// Build things from class Interaction for testing
LwlynSmithQELCCPXSec * lwlyn = new LwlynSmithQELCCPXSec();
Interaction * inter = new Interaction();
KinePhaseSpace_t kps = kPSQ2fE;
// int A nd Z represent carbon
int A = 12;
int Z = 6;
int probe_pdgc = 14;
int target_pdgc = pdg::IonPdgCode(A,Z);
// int for GeV
double Ev = 3;
//Setting up TLorentzVector
TLorentzVector nu_p4(0., 0., Ev, Ev); //px, py, pz, E
ProcessInfo proc_info(kScQuasiElastic, kIntWeakCC);
BOOST_AUTO_TEST_CASE(doubles)
{
inter->InitStatePtr()->SetPdgs(target_pdgc, probe_pdgc);
inter->InitStatePtr()->SetProbeP4(nu_p4);
std::cout << "The XSec is " << lwlyn->XSec(inter, kps) << std::endl;
BOOST_CHECK(lwlyn->XSec(inter, kps) == 5.3365984441833672E-11);


}
BOOST_AUTO_TEST_SUITE_END()
}
```

Testing XSEC

- Testing a cross section in the LlewellynSmith class

```
BOOST_AUTO_TEST_SUITE(Lwlyn)
// Build things from class Interaction for testing
LwlynSmithQELCCPXSec * lwlyn = new LwlynSmithQELCCPXSec();
Interaction * inter = new Interaction();
KinePhaseSpace_t kps = kPSQ2fE;
// int A nd Z represent carbon
int A = 12;
int Z = 6;
int probe_pdgc = 14;
int target_pdgc = pdg::IonPdgCode(A,Z);
// int for GeV
double Ev = 3;
//Setting up TLorentzVector
TLorentzVector nu_p4(0., 0., Ev, Ev); //px, py, pz, E
ProcessInfo proc_info(kScQuasiElastic, kIntWeakCC);
BOOST_AUTO_TEST_CASE(doubles)
{
inter->InitStatePtr()->SetPdgs(target_pdgc, probe_pdgc);
inter->InitStatePtr()->SetProbeP4(nu_p4);
std::cout << "The XSec is " << lwlyn->XSec(inter, kps) << std::
BOOST_CHECK(lwlyn->XSec(inter, kps) == 5.3365984441833672E-11);


}
BOOST_AUTO_TEST_SUITE_END()
}
```

```
July 14 2016
    Not gona lie the past 6 days have been a mess.
    Set up constructors and fixed the test
    Fixed the Makefile
    Test runs
    But fails at Valid process.
    Need to go into that and find out why it failed.
    Set up a different file called gtest1check.cpp
        Sets up code inder a main so I can gdb it
        Need to add that to the makeile.
    gdb ing to try and find our what's wrond with ValidProcess
        gdb -tui --args ./gtest1check
    Failed
    Really?
    AAHAHSGHASHAJSKDASHSDKJHAALKSJD!!!!!
```

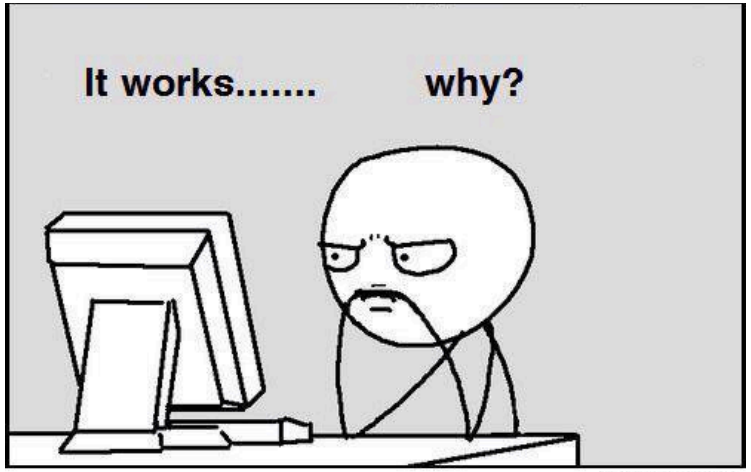

Fermilab

Testing Booleans

```
BOOST_AUTO_TEST_CASE(bools)
{
        inter->InitStatePtr()->SetPdgs(target_pdgc, probe_pdgc);
        inter->InitStatePtr()->SetProbeP4(nu_p4);
        std::cout<< "hmmm " << kISkipProcessChk  << std::endl;
        std::cout<<"testing recoil nucleon"<<recoil_nuc<<std::endl;
        std::cout<<"testing proton "<<isP<<"target_pdgc = " << target_pdgc<<std::endl;
        cout<<kScQuasiElastic<<endl;
        cout<<"TESTING BOOLS"<<endl;
        BOOST_CHECK(probe_pdgc == 14);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsQuasiElastic() == 1);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsDeepInelastic() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsResonant() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsCoherent() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsCoherentElas() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsSingleKaon() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsElectronScattering() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsNuElectronElastic() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsInverseMuDecay() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsIMDAnnihilation () == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsInverseBetaDecay() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsGlashowResonance() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsAMNuGamma() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsMEC() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsDiffractive() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsEM() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeak() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeakCC() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeakNC() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeakMix () == 0);
        cout<<"CLEAR"<<endl;
}
```

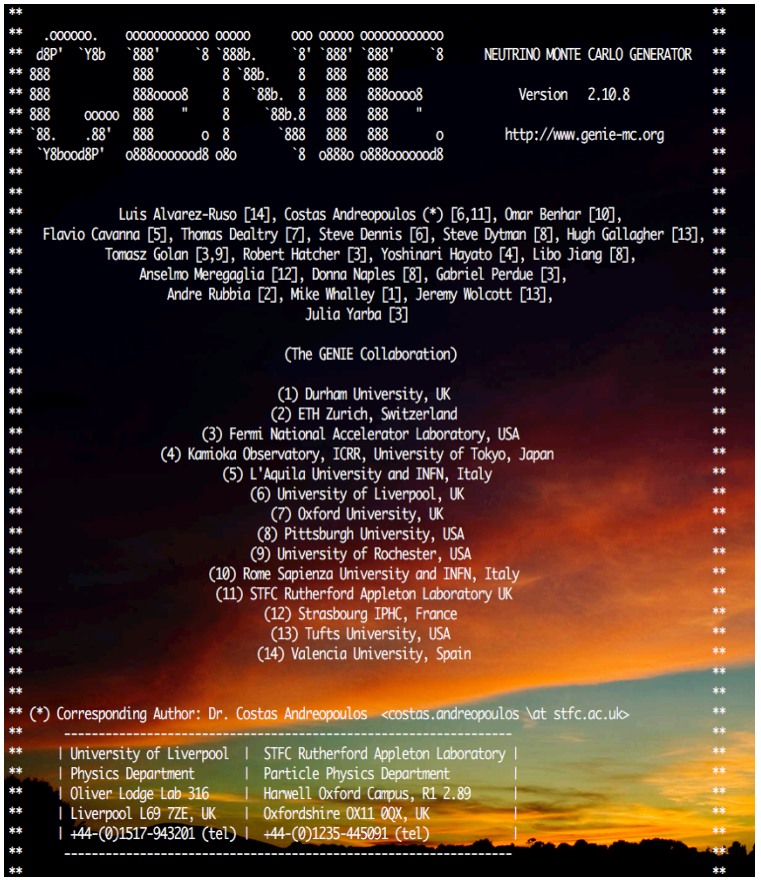
Testing Booleans

```
BOOST_AUTO_TEST_CASE(bools)
{
        inter->InitStatePtr()->SetPdgs(target_pdgc, probe_pdgc);
        inter->InitStatePtr()->SetProbeP4(nu_p4);
        std::cout<< "hmmm " << kISkipProcessChk  << std::endl;
        std::cout<<"testing recoil nucleon"<<recoil_nuc<<std::endl;
        std::cout<<"testing proton "<<isP<<"target_pdgc = " << target_pdgc<<std::endl;
        cout<<kScQuasiElastic<<endl;
        cout<<"TESTING BOOLS"<<endl;
        BOOST_CHECK(probe_pdgc == 14);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsQuasiElastic() == 1);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsDeepInelastic() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsResonant() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsCoherent() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsCoherentElas() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsSingleKaon() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsElectronScattering() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsNuElectronElastic() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsInverseMuDecay() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsIMDAnnihilation () == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsInverseBetaDecay() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsGlashowResonance() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsAMNuGamma() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsMEC() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsDiffractive() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsEM() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeak() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeakCC() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeakNC() == 0);
        cout<<"MOVING ON"<<endl;
        BOOST_CHECK(proc_info.IsWeakMix () == 0);
        cout<<"CLEAR"<<endl;
}
```



Future of Unit-Test for GENIE

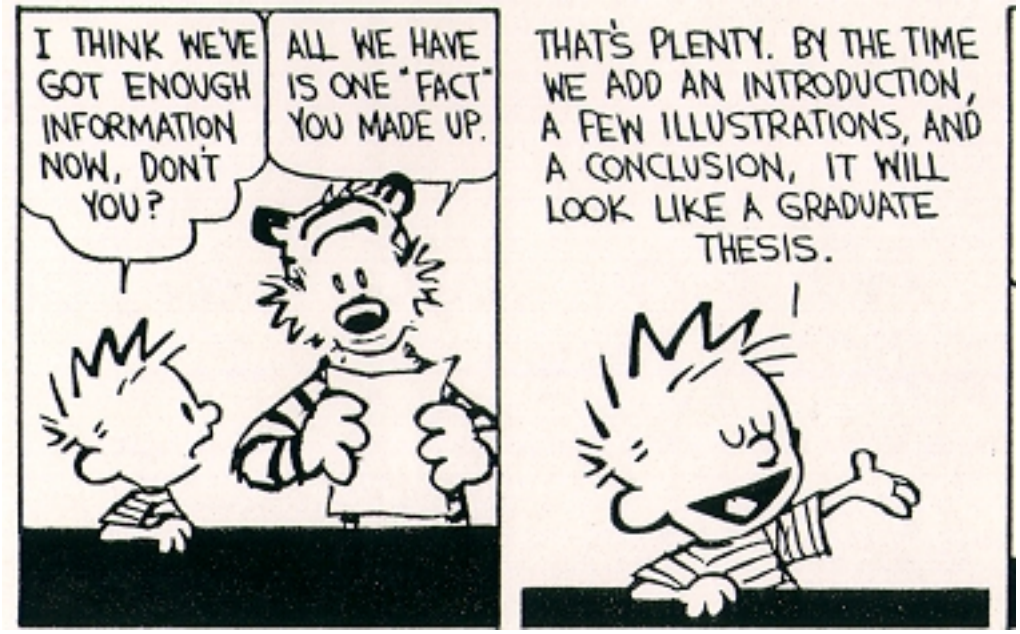
- So far:
  - Showed BOOST can work with Genie
- Develop an infrastructure
  - Currently doing
  - Allow for testing by all users
  - Protect Genie from deadly errors



Acknowledgements

- SIST Committee
  - Sandra Charles, Elliott McCrory, Judy Nuñez
  - Mayling Wong, Charles Orozco
- GENIE
  - Gabriel Perdue and the GENIE Team
- Fellow Interns

Questions?


I THINK WE'VE GOT ENOUGH INFORMATION NOW, DON'T YOU?
ALL WE HAVE IS ONE "FACT" YOU MADE UP.
THAT'S PLENTY. BY THE TIME WE ADD AN INTRODUCTION, A FEW ILLUSTRATIONS, AND A CONCLUSION, IT WILL LOOK LIKE A GRADUATE THESIS.