“Once upon a time…

there was a Good Software Engineer whose Customers knew exactly what they wanted.

The Good Software Engineer worked very hard to design the Perfect System that would solve all the Customers’ problems now and for decades.

When the Perfect System was designed,
implemented and finally deployed,
the Customers were very happy indeed.

The Maintainer of the System had very little to do
to keep the Perfect System up and running,
and the Customers and the Maintainer
lived happily every after.”

It sounds like a fairy tale…

Could it be because
there are no Good Software Engineers?

Could it be because
the Customers/Users don't really know what they want?

Or because **the Perfect System does not exist?**

Lehman laws

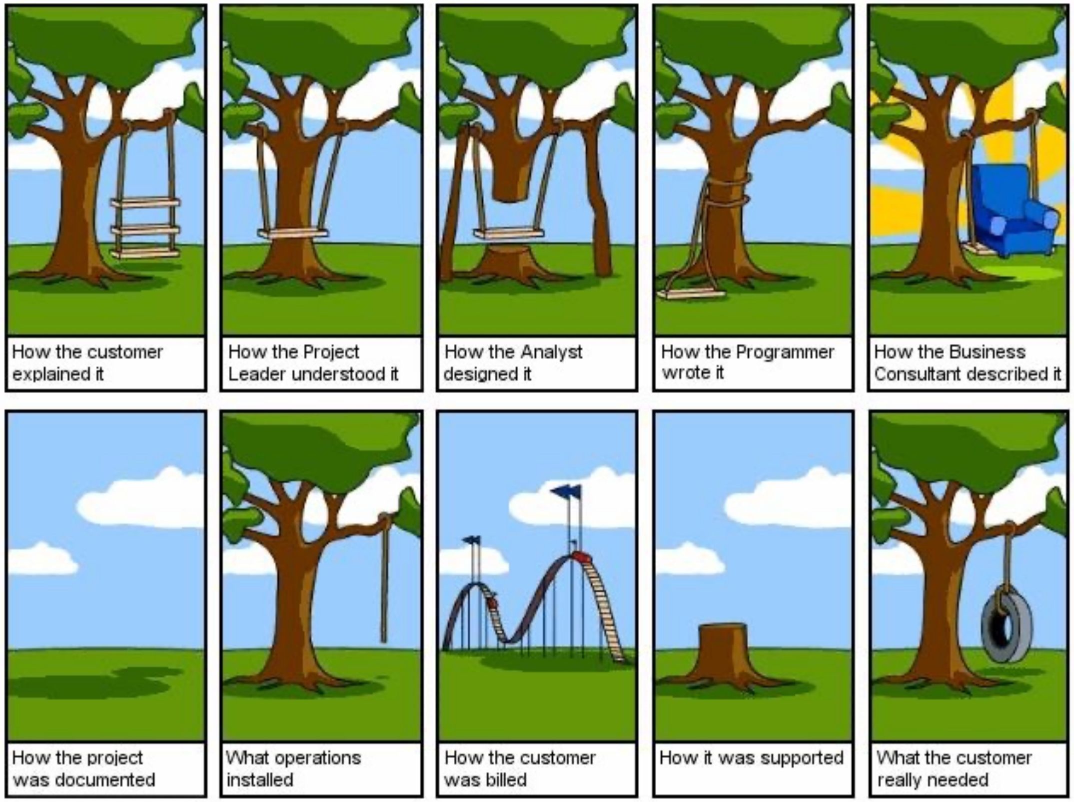
M. M. Lehman,
Programs, Life Cycles, and Laws of Software Evolution,
Proc. IEEE, vol. 68, no. 9, Sep. 1980

1. **Continuing Change**

   A program that is used and that as an implementation of its specification reflects some other reality, **undergoes continual change** or **becomes progressively less useful**. The change or decay process continues until it is judged more cost effective to replace the system with a recreated version.

2. **Increasing Complexity**

   As an evolving program is continually changed, **its complexity**, *reflecting deteriorating structure*, **increases** unless work is done to maintain or reduce it.



Refactoring

Maria Grazia Pia

INFN Genova, Italy

Maria.Grazia.Pia@cern.ch

http://www.ge.infn.it/geant4/training/APC2025/

Refactoring

…is a disciplined technique for
improving the design of an existing code

In the ideal world
there would be hardly any need for refactoring

In the real world of HEP (and related fields)
most software needs to be refactored

By learning refactoring
you also learn writing code
that minimizes the need to be refactored

“If it ain’t broken, don’t fix it”

conventional wisdom

A piece of software can be broken in many ways

Functional	it no longer delivers the function it is designed to perform
Maintenance	it can no longer be maintained

Warnings you are heading into trouble

usually do not occur isolated

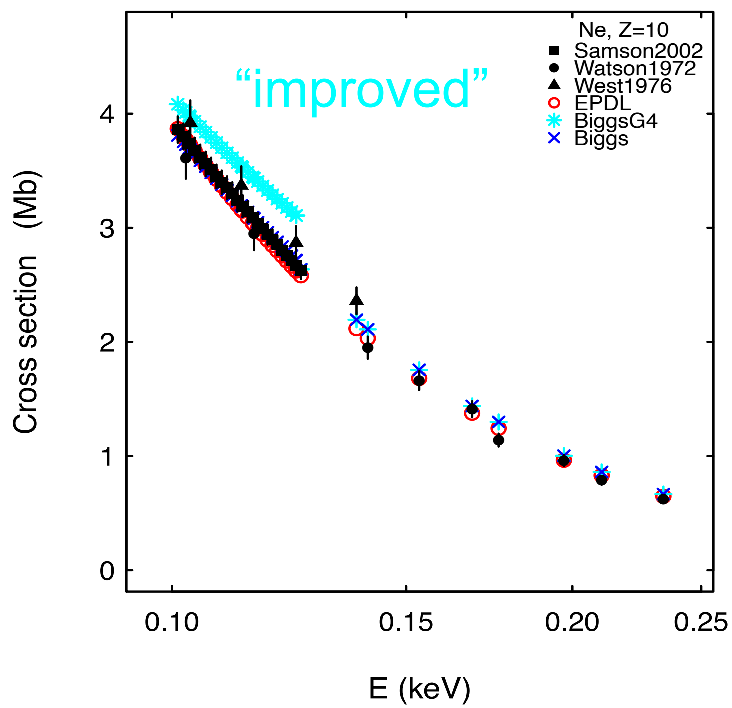
- Obsolete or no **documentation**
- Missing **tests**
- Original **developers** or users have left
- **Inside knowledge** about the system has disappeared
- Limited understanding of the **entire system**
- Too long to turn things over to **production**
- Too much time to make **simple changes**
- Need for constant **bug fixes**
- Big **build times**
- Difficulties **separating** products
- **Duplicated** code
- Code **smells**

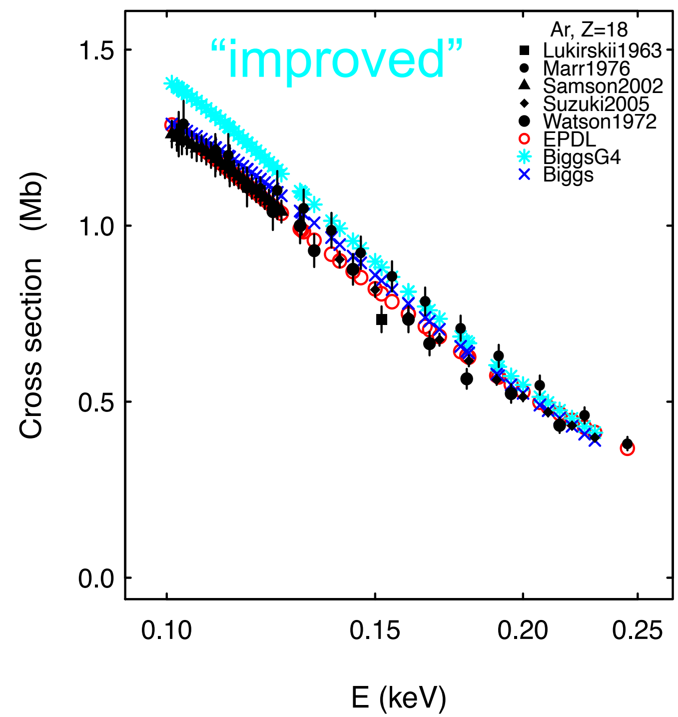
S. Demeyer, S. Ducasse, O. Nierstrasz, Object Oriented Reengineering Patterns

Photoionisation cross section

Cross sections in Geant4 "standard" photoelectric model based on "**improved**" Biggs-Lighthill parameterisation

F. Biggs and R. Lighthill, Analytical Approximation for X-ray Cross Sections III, Sandia Lab. Report SAND-0070, 1988





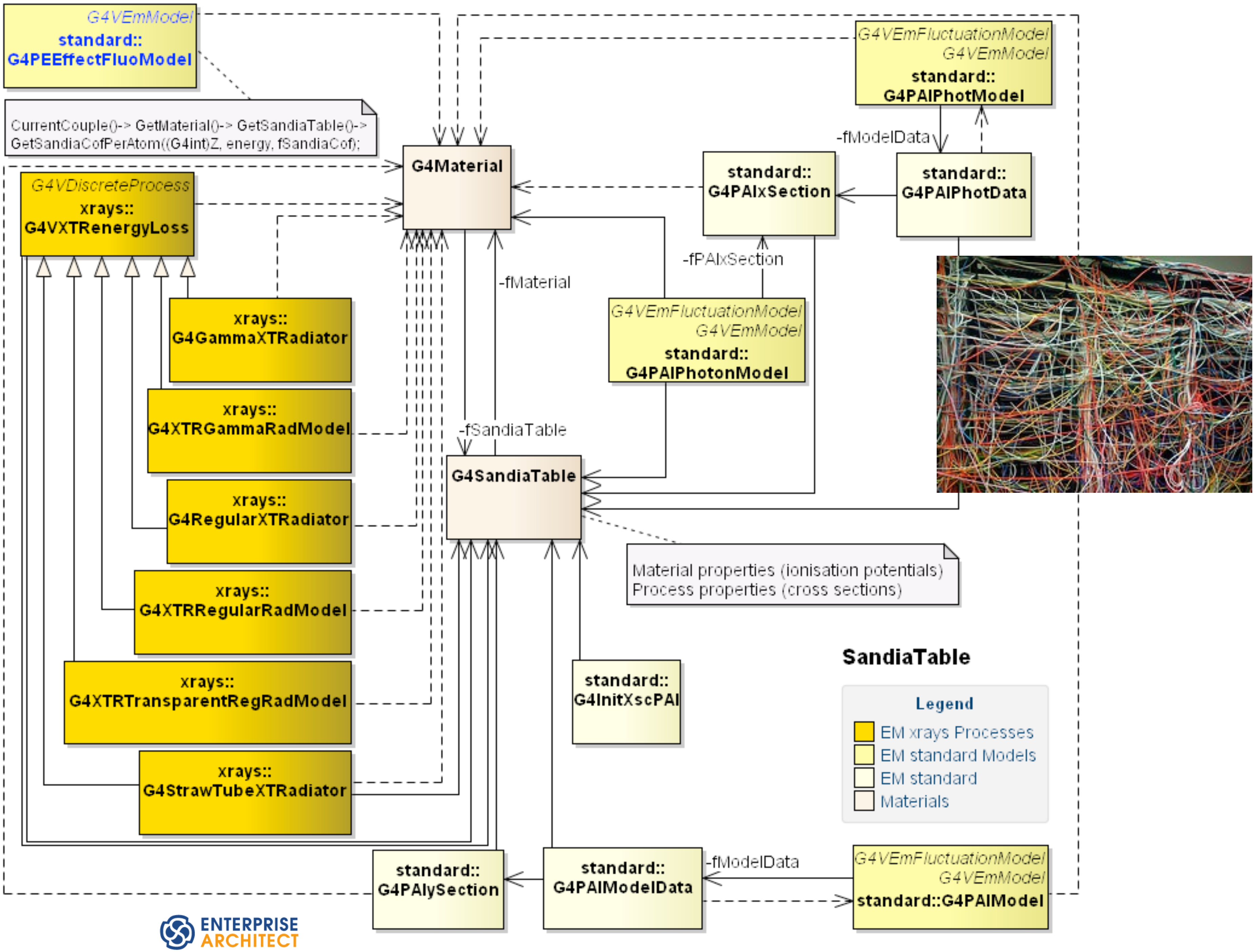
Test made possible by refactoring the code

Validation of cross sections for Monte Carlo simulation of the photoelectric effect

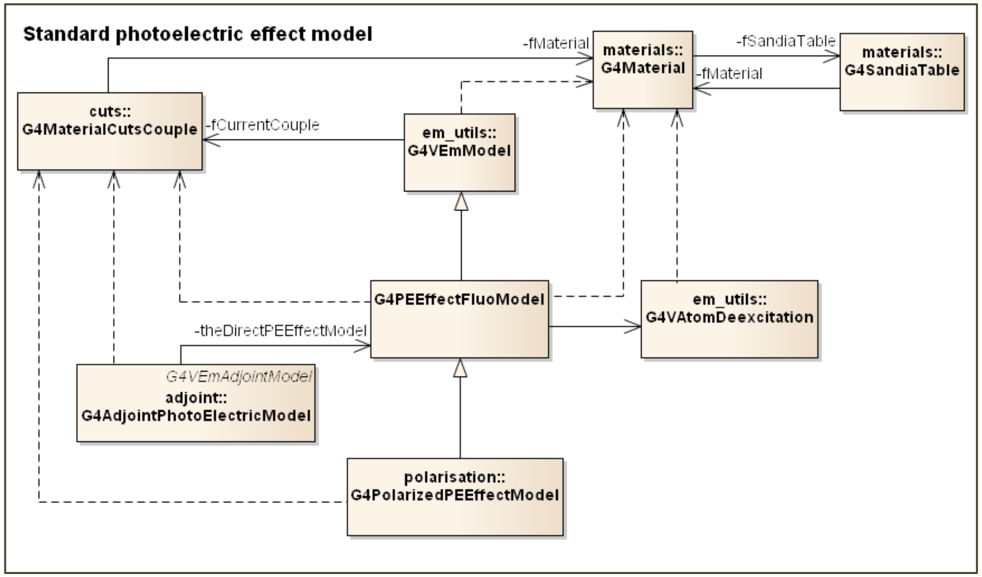
Min Cheol Han, Han Sung Kim, Maria Grazia Pia, Tullio Basaglia, Matej Batič, Gabriela Hoff, Chan Hyeong Kim, and Paolo Saracco

IEEE Trans. Nucl. Sci., vol. 63, no. 2, Apr. 2016

http://arxiv.org/abs/1601.06514


G4VEmModel
standard::
G4PEEffectFluoModel
CurrentCouple()-> GetMaterial()-> GetSandiaTable()->
GetSandiaCofPerAtom((G4int)Z, energy, fSandiaCof);
G4Material
G4VEmFluctuationModel
G4VEmModel
standard::
G4PAIPhotModel
-fModelData
standard::
G4PAIxSection
standard::
G4PAIPhotData
G4VDiscreteProcess
xrays::
G4VXTRenergyLoss
-fPAIxSection
-fMaterial
xrays::
G4GammaXTRadiator
G4VEmFluctuationModel
G4VEmModel
standard::
G4PAIPhotonModel
xrays::
G4XTRGammaRadModel
-fSandiaTable
G4SandiaTable
xrays::
G4RegularXTRadiator
Material properties (ionisation potentials)
Process properties (cross sections)
xrays::
G4XTRRegularRadModel
SandiaTable
standard::
G4InitXscPAI
xrays::
G4XTRTransparentRegRadModel
Legend
EM xrays Processes
EM standard Models
EM standard
Materials
xrays::
G4StrawTubeXTRadiator
standard::
G4PAIySection
standard::
G4PAIModelData
-fModelData
G4VEmFluctuationModel
G4VEmModel
standard::G4PAIModel
ENTERPRISE
ARCHITECT

Hidden dependencies



One needs a geometry
(and a full scale application)
to test a photon cross section

Difficult to test

often…

no testing

Post-RD44 electromagnetic design

evolution

1. one of a set of prescribed movements
2. a process of change in a certain direction
3. the process of working out or developing
4. the historical development of a biological group
5. the extraction of a mathematical root
6. a process in which the whole universe is a progression of interrelated phenomena

"The code slowly sinks from engineering to hacking."
M. Fowler, **Refactoring**

software **evolution** ⟷ **legacy** software

"With rapid development tools and rapid turnover in personnel, software systems can turn into legacies more quickly than you might imagine."

S. Demeyer, S. Ducasse, O. Nierstrasz, Object Oriented Reengineering Patterns

legacy

Merriam-Webster

1. a gift by will especially of money or other personal property
2. something transmitted by or received from an ancestor or predecessor or from the past

"A legacy is something *valuable* that you have *inherited.*"

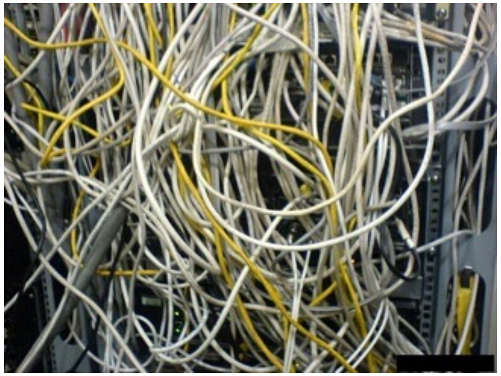
S. Demeyer, S. Ducasse, O. Nierstrasz, Object Oriented Reengineering Patterns

Refactoring   Reengineering

Methods and techniques

- to deal with software **evolution**
- to manage **complexity**
- to work with **legacy** code

in a **disciplined** and **effective** way



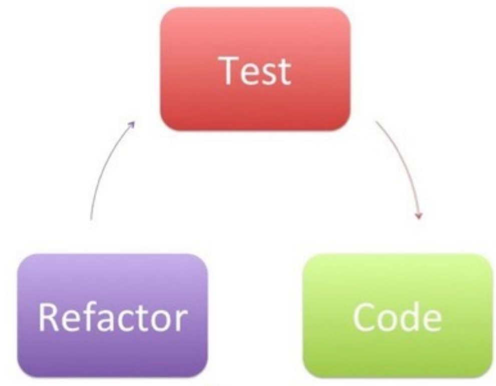
Evolution of legacy systems

New Functionality

Hack it in ?

- duplicated code
- complex conditionals
- abusive inheritance
- large classes/methods

First …
- refactor
- restructure
- reengineer



Take a loan on your software
⇒ pay back via reengineering

Investment for the future
⇒ paid back during maintenance

S. Demeyer, S. Ducasse, O. Nierstrasz,
Object Oriented Reengineering Patterns

Software maintenance

"The modification of a software product after delivery
to **correct faults**,
to **improve** performance or other attributes,
or to **adapt** the product to a modified environment."

IEEE Standard 1219

- Accommodate changes in the software environment
- Incorporate new user requirements
- Fix errors
- Prevent future problems

OO techniques promise better

- flexibility,
- reusability,
- maintainability
- …

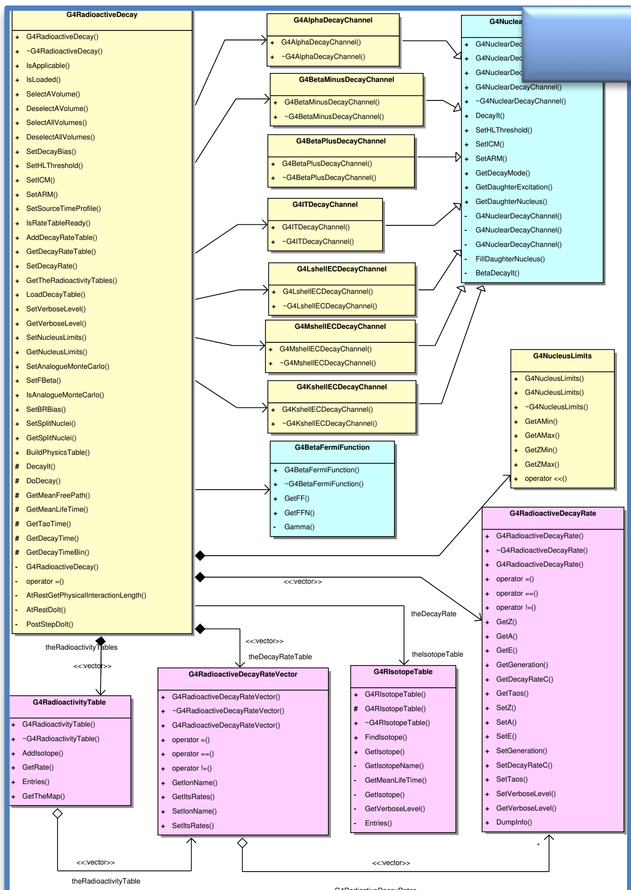
but they do not come for free!

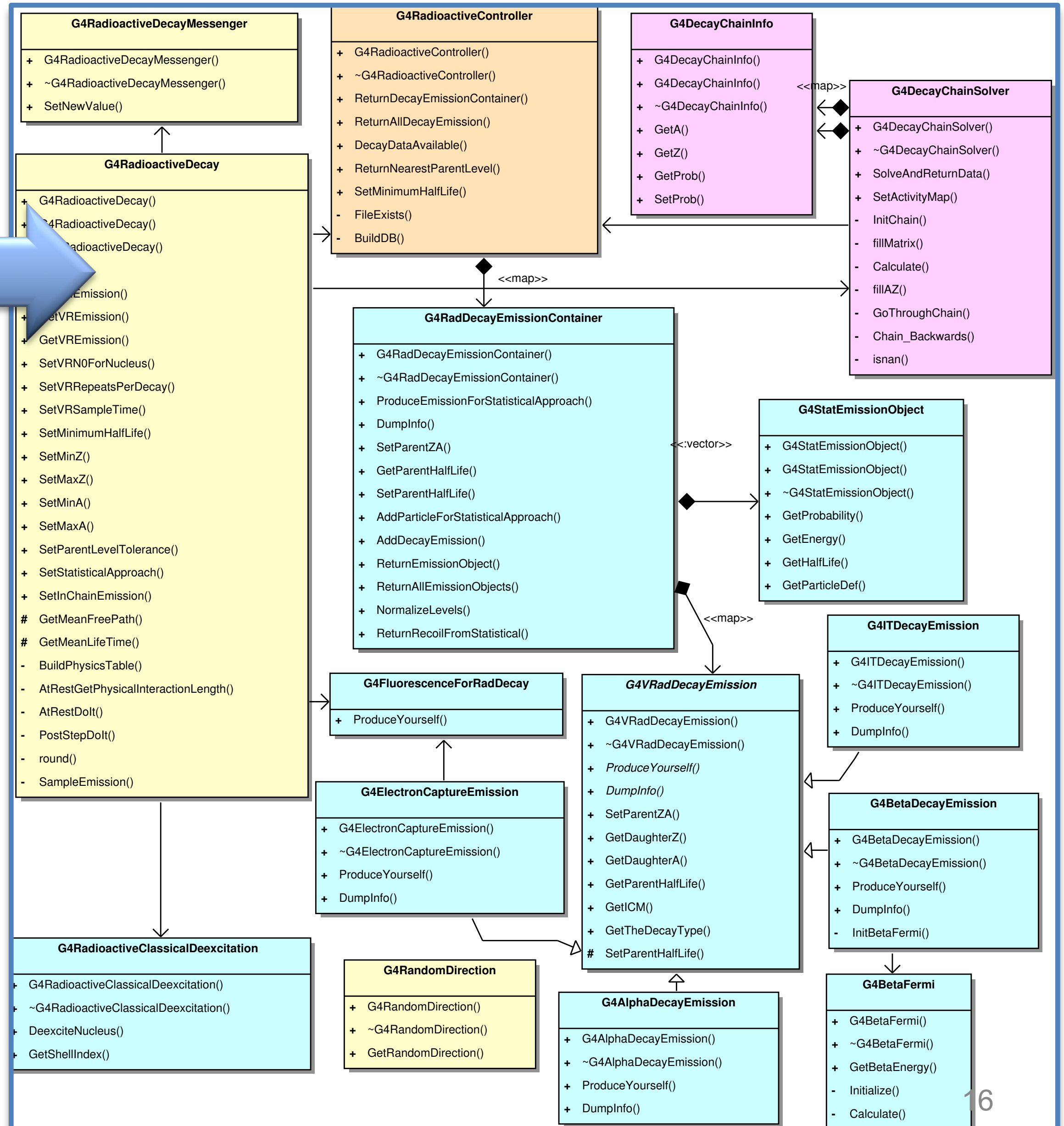
Does refactoring pay back?

Steffen Hauf
PhD Thesis

Refactoring Geant4 Radioactive Decay

Well defined **responsibilities** and **interactions**

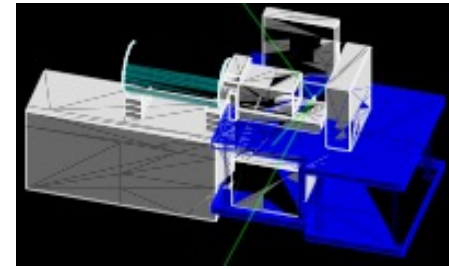




Maria Grazia Pia, *INFN Genova*

Motivations

- Gain understanding of the code
- Assess its capabilities and accuracy
- Improve physics performance
- Improve computational performance

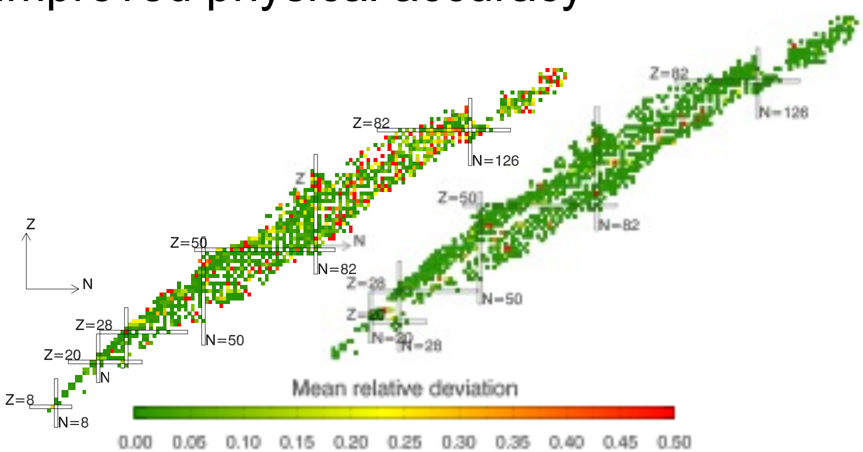



Experiment: Z. W. Bell *(ORNL)*

Enabled by refactoring

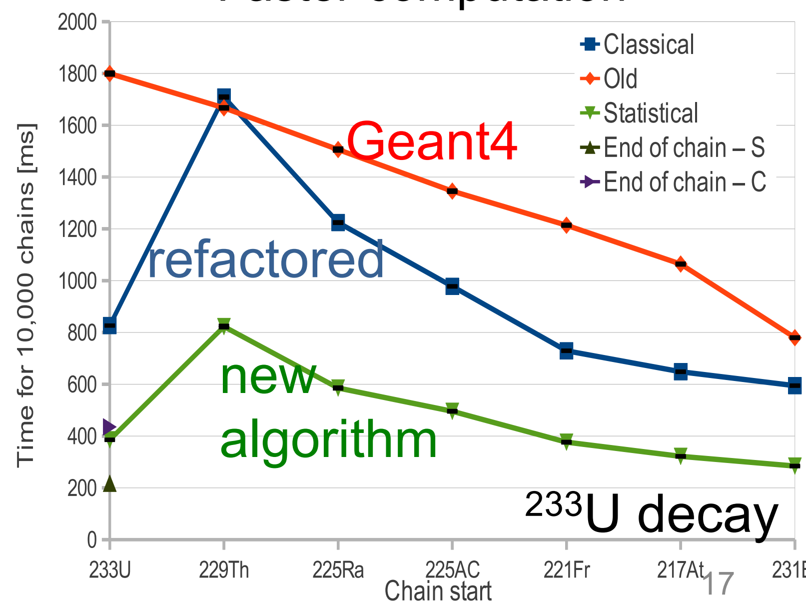
New algorithm

Improved physical accuracy



Performance

Faster computation



Maria Grazia Pia, *INFN Genova*

2966 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 60, NO. 4, AUGUST 2013

Radioactive Decays in Geant4

Steffen Hauf, Markus Kuster, Matej Batič, Zane W. Bell, Dieter H. H. Hoffmann, Philipp M. Lang, Stephan Neff, Maria Grazia Pia, Georg Weidenspointner, and Andreas Zoglauer

2984 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 60, NO. 4, AUGUST 2013

Validation of Geant4-Based Radioactive Decay Simulation

Steffen Hauf, Markus Kuster, Matej Batič, Zane W. Bell, Dieter H. H. Hoffmann, Philipp M. Lang, Stephan Neff, Maria Grazia Pia, Georg Weidenspointner, and Andreas Zoglauer

Steffen Hauf
Head of Software Control Group
XFEL

Outline

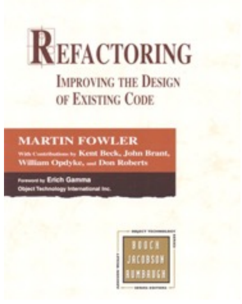
Software technology
to deal with evolving/legacy software

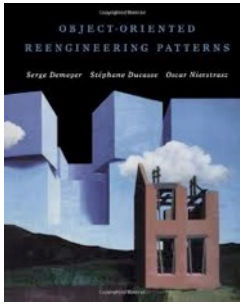
- Problems
- Methods
- Techniques

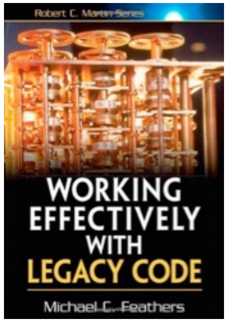
Overview

Focus on basic **concepts**

Guidance for further personal study







Peculiarities of refactoring physics software

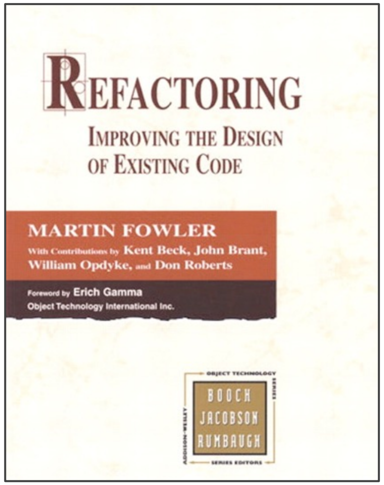
Hands-on practice

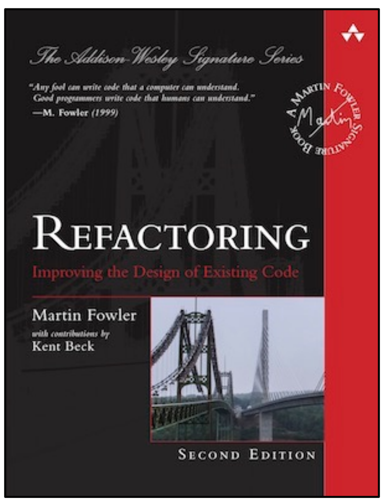
Common problems of legacy code

- **Insufficient documentation**
  - non-existent or out-of-date
- **Improper layering**
  - too few or too many layers
- **Lack of modularity**
  - strong coupling
- **Duplicated code**
  - copy & paste code
- **Duplicated functionality**
  - similar functionality by separate teams
- **Misuse** of inheritance
  - code reuse vs. polymorphism
- **Missing** inheritance
  - duplication, case-statements
- **Misplaced** operations
  - operations outside classes
- **Violation** of encapsulation
  - type-casting; C++ "friends”
- **Class abuse**
  - classes as namespaces

S. Demeyer, S. Ducasse, O. Nierstrasz, Object Oriented Reengineering Patterns

1999



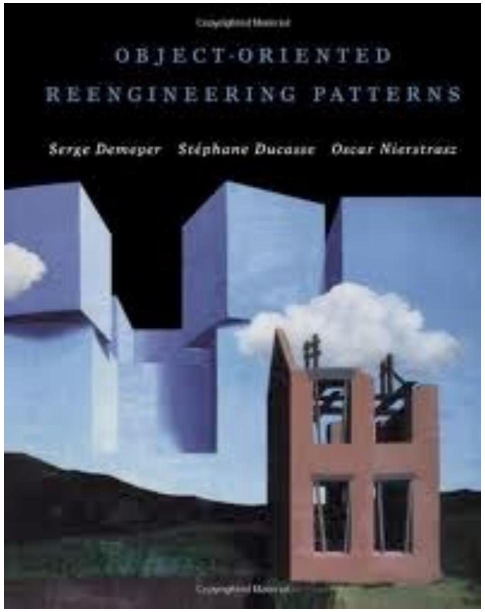


2018

Refactoring

is "the process of changing a software system in such a way that it does not alter the **external behavior** of the code yet improves its **internal structure**."

"When you refactor
you are improving the design of the code
after it has been written."



Reengineering

Reengineering "seeks to transform a legacy system into the system you would have built if you had the luxury of hindsight and could have known all the new requirements that you know today."

"**Reengineering** […] is the examination and alteration of a subject system to reconstitute it in a new form and the subsequent implementation of the new form."

"**Reverse Engineering** is the process of analyzing a subject system
- to identify the system's components and their interrelationships and
- create representations of the system in another form or at a higher level of abstraction."

"**Forward Engineering** is the traditional process of moving from high-level abstractions and logical, implementation-independent designs to the physical implementation of a system."

E.J. Chikofsky, J.H. Cross, Reverse engineering and design recovery: a taxonomy, IEEE Software, vol. 7, no.1, pp. 13-17, 1990.

Basic concepts

Refactoring

Reengineering

Interplay with testing

- What they are
- Why to refactor (reengineer)
- When to refactor
- What NOT to refactor
- When NOT to refactor

Do not refactor what does not pass the tests

Do not refactor immediately before a critical deadline

Why?

To make software **easier to understand** and **modify**

When?

- Refactor when you want to add functionality
  - *before adding it*
- Refactor when you have to fix a bug
- Refactor while doing a code review (!)
- Refactor when you want to gain understanding of some legacy code
- …

Refactoring embedded in an iterative-incremental life-cycle

Set priorities while refactoring

Risks

- The new code may be more "elegant" but it has bugs / is slower
- Instead of just fixing a bug, you refactor the core code and screw up everything

Testing

- When replacing old code that has been working fine for a long time, one risks reintroducing old problems that were fixed by some of the "ugly" (undocumented) code
- Nobody remembers all of the requirements, but break a single one by refactoring, and you can be in deep trouble

Reengineering

- Refactoring increases the amount of verification testing that has to be done: when you refactor a class, you need to retest everything that deals with it (and side effects too)

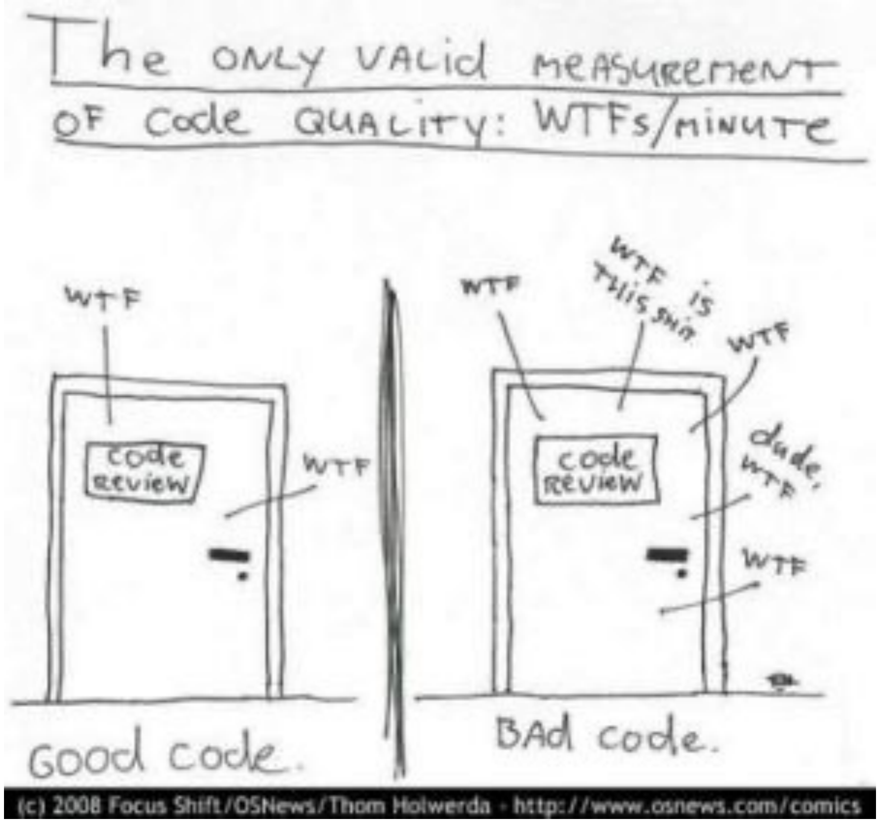
Planning

- Refactoring is an excuse for lazy programmers

Refactor your own code
Refactor your colleague's code

What to refactor

How does one identify code that needs to be refactored?



Identifying code to be refactored

If it stinks, change it.

Grandma Beck, discussing child-rearing philosophy

M. Fowler, K. Beck et al.,
Refactoring: Improving the Design of Existing Code

A **code smell** is a **surface indication** that usually corresponds to a **deeper problem** in the system

Smells are heuristics that help in deciding:

- When to refactor
- What to refactor
- How to refactor

Quick to spot

Don't always indicate a problem

Code smells

- Duplicated code
- Long method
- Large class
- Long parameter list
- Divergent change
- Shotgun surgery
- Feature envy
- Data clumps
- Switch statement
- Parallel inheritance hierarchies
- Lazy class
- Speculative generality
- Temporary field
- Comments
- Refused bequest
- Primitive obsession
- Message chains
- Middle man
- Inappropriate intimacy
- Alternative classes with different interfaces
- Incomplete library class
- Data class

M. Fowler, K. Beck et al.,
Refactoring: Improving the Design of Existing Code

Common code smells

- **Duplicated Code** 

in the stink parade
  - if you modify one instance of duplicated code but not the others, you may introduce a bug!
- **Large class**
  - tries to do too much
- **Long Method**
  - difficult to understand and maintain
  - *Martin's Rule of Performance: Assume costs of lots of short functions are negligible and wait to be proven wrong!*
- **Long Parameter List**
  - hard to understand, can become inconsistent

Code smells: **dispensable**

- **Data Class**
  - A data holder: a class that has attributes, getting and setting methods for the fields, and nothing else
  - *Objects should be about data **and** behavior*
- **Speculative Generality**
  - "I may need the ability to do this kind of thing someday"
- **Lazy Class**
  - A class that no longer "pays its way"
    e.g. a class that was downsized by refactoring, or represented planned functionality that did not materialize
- **Dead Code**
  - Code that is not used

Code smells: **OO abusers**

- **Refused Bequest**
  - A subclass ignores most of the functionality provided by its superclass
- **Switch Statements**
  - Can be replaced by use of polymorphism
- **Temporary Field**
  - An attribute of an object is only set in certain circumstances *but an object should need all of its attributes*
  - Fields used to hold intermediate results
- **Alternative Classes with Different Interfaces**
  - Two or more methods do the same thing but have different signature for what they do

Code smells: **coupling**

- **Middle Man**
  - A class delegates most of its responsibilities to another class
  - *Does it really have a reason to exist?*
- **Message Chains**
  - A client asks an object for another object, then asks that object for another object etc.
- **Feature Envy**
  - A method requires lots of information from some other object
- **Inappropriate Intimacy**
  - Classes that know too much about each other’s private details

Code smells: **hindering change**

- **Divergent Change**
  - Lack of cohesion:
    one type of change requires changing one subset of methods; another type of change requires changing another subset
- **Shotgun Surgery**
  - A change requires lots of little changes in a lot of different objects
- **Parallel Inheritance Hierarchies**
  - Similar to Shotgun Surgery; each time I add a subclass to one hierarchy, I need to do it for all related hierarchies

…and more

- **Data Clumps**
  - Attributes that are used together, but are not part of the same object
- **Primitive Obsession**
  - A reluctance to use classes instead of primitive data types
- **Magic Number**
  - A literal value that appears in a program
- **Combinatorial Explosion**
  - Lots of code that does *almost* the same thing

How to refactor

Methods
Techniques

Reverse engineering

Not limited to deriving a UML class diagram from the code…
Motivation: understanding other people's code

Refactoring is not meant to alter the behaviour of the code

To be tested!

Refactoring begins by designing a **solid set of tests** for the portion of code under analysis

(usually unit tests)

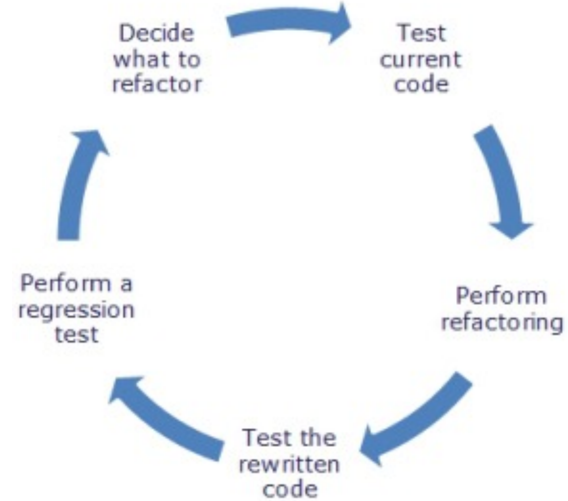
Refactoring occurs as a **series of small changes**

Test original code

Apply one action at a time

Test

. . .



→ **hands-on exercise**

What may happen…

…if the software is changed *("improved")* without testing

Available on CMS information server

CMS NOTE 2000/016

The Compact Muon Solenoid Experiment

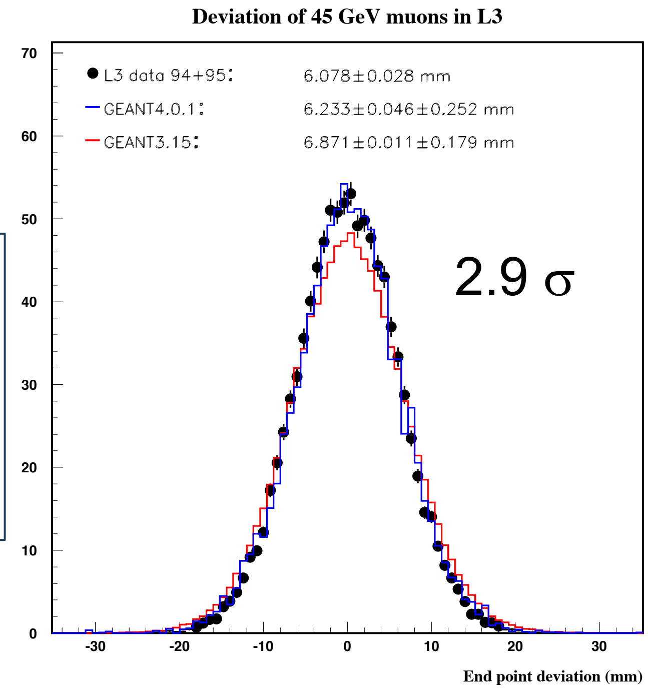
CMS Note

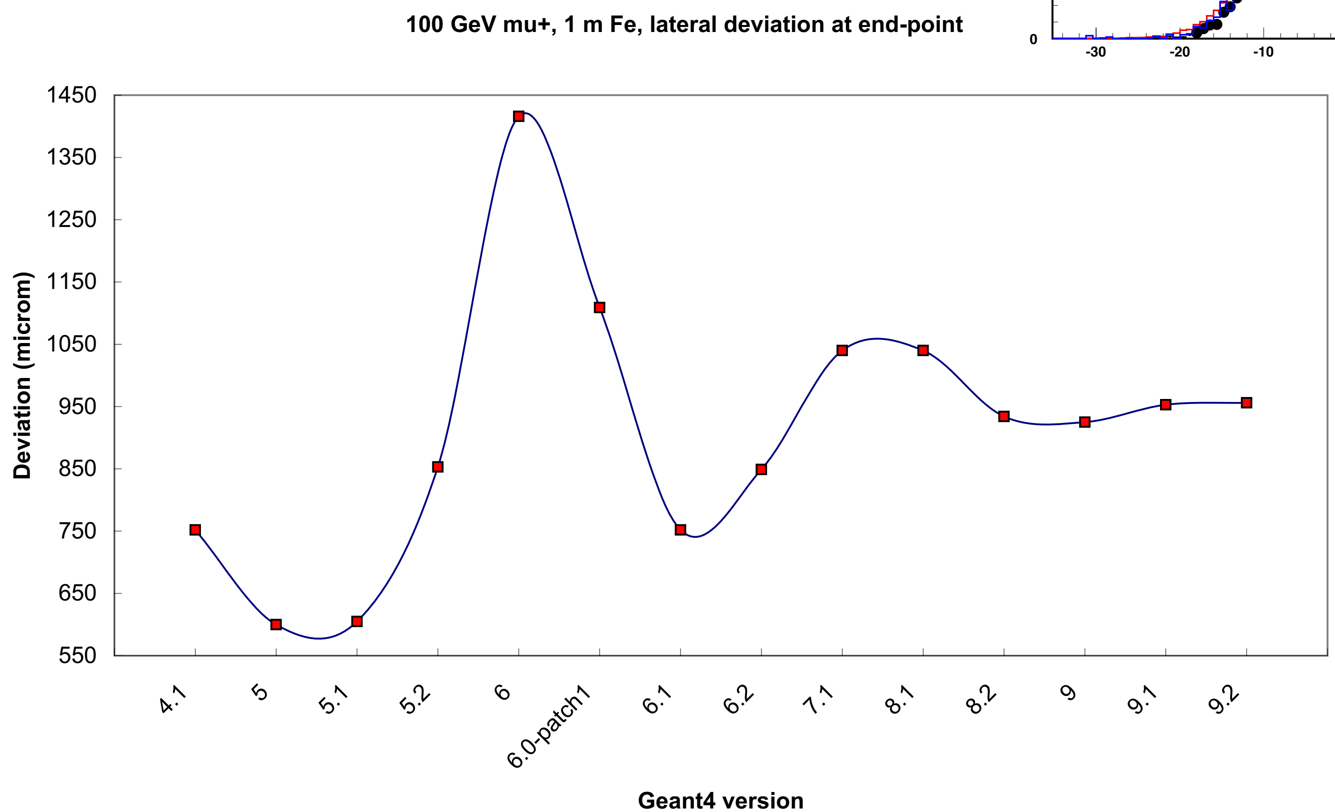
Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

11 January 2000

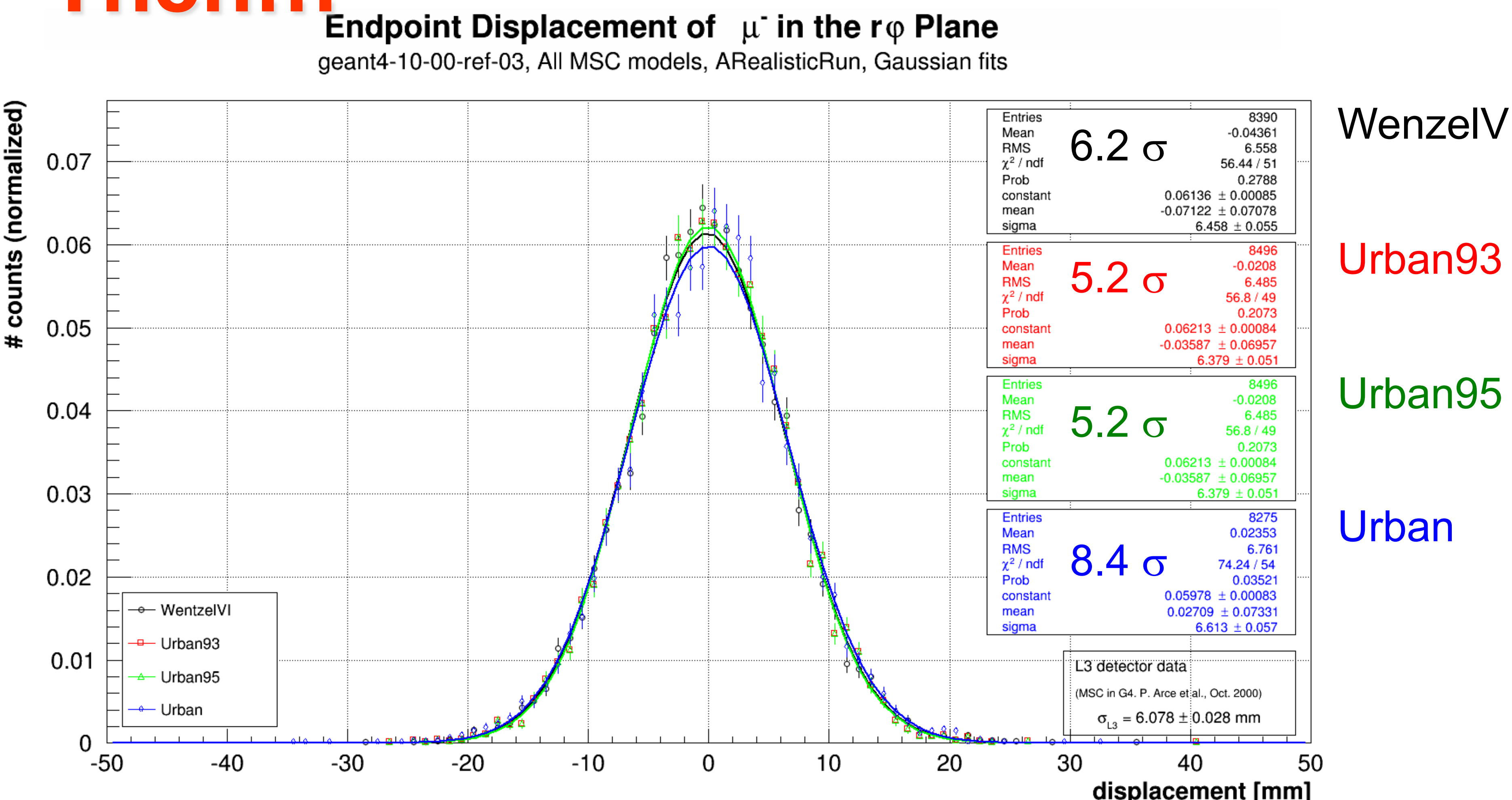
P. Arce, M. Wadhwa

Deviation in matter of 45 GeV muons in GEANT3 and GEANT4. A comparison with L3 data





Then…



V. Ivanchenko, 14/4/2014

Beware:

experimental geometry is not reproducible!

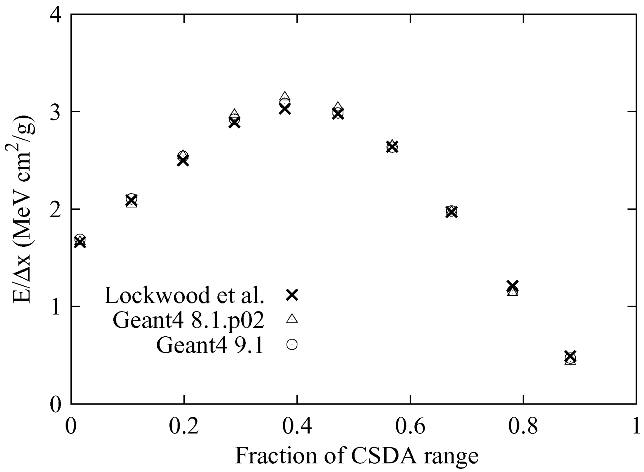


Fig. 8. Energy deposition divided by density-scaled calorimeter thickness Δx versus depth in C for 1 MeV e^- at 0° incidence: simulations with Geant4 library-based processes, Geant4 version 8.1p02 (triangles) and 9.1 (circles), experimental data from [5] (crosses).

398 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 56, NO. 2, APRIL 2009

Validation of Geant4 Low Energy Electromagnetic Processes Against Precision Measurements of Electron Energy Deposition

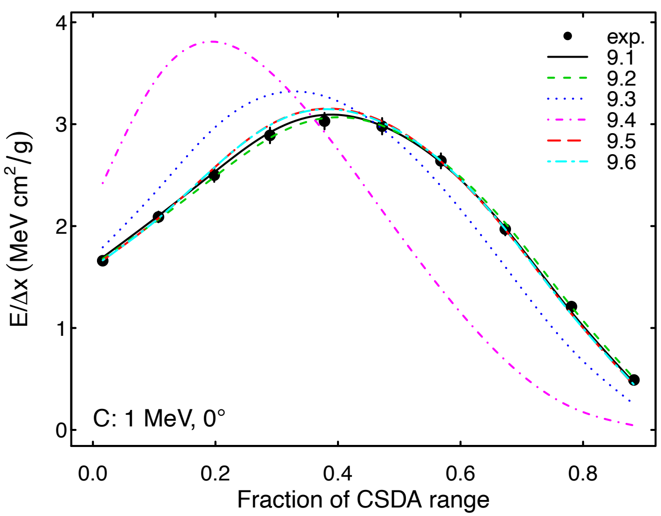
Anton Lechner, Maria Grazia Pia, and Manju Sudhakar

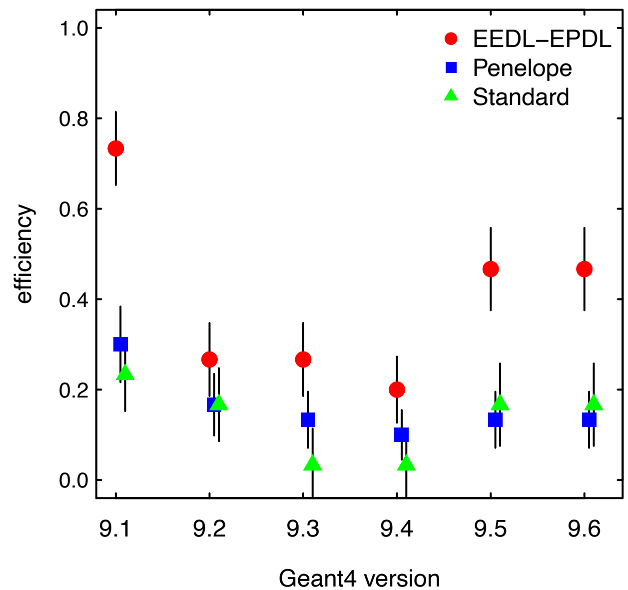
IEEE NSS Best Student Paper, 2007

2934 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 60, NO. 4, AUGUST 2013

Validation of Geant4 Simulation of Electron Energy Deposition

Matej Batič, Gabriela Hoff, Maria Grazia Pia, Paolo Saracco, and Georg Weidenspointner





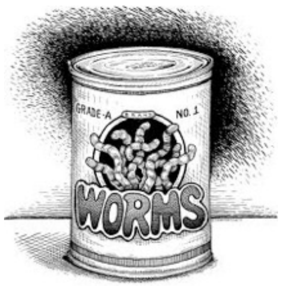
Negative improvements…

Refactoring aims to preserve correctness

Sweeping under the carpet?

Was the original code **verified**?

Was the original code **validated**?



IEEE Standard 1012
Software Verification & Validation

What was the test coverage?

Were the test process and the test results documented?

Std 1012 evolution
IEEE Standard for **System**, **Software**, and **Hardware** Verification and Validation

Risk mitigation

Do not mix!

One of the motivations for refactoring may be the need to introduce new features in the code

Refactoring

Does not modify the code behaviour

Adding new features

Modifying the code behaviour

- Test
- Refactor
- Test
- Add new feature
- Test
- Add new feature
- Test
- …

→ **hands-on exercise**

Basic actions for common smells

- **Method invocation**
  - Consolidate **recurring code** into a single method
  - OK when the recurring code does not span methods and all methods containing code belong to the same class

```
commonCode() {
…
}
```

- **Inheritance**
  - Common code in two different classes
  - New superclass introduced

```
class Child : public Super {
…
};
```

Extract commonality

Abstract interfaces
are classes with no implementation
Abstract classes
represent mixed design and implementation

- **Introduce abstract class**
  - a class with no or partial implementation

```
class Common {
  void commonCode(...) {...}
  virtual void contextSpecificCode () = 0;
  ...
} ;
```

- **Add delegation**
  - Delegate the recurring code segments to a helper class

```
class Extension : public Common {
  void method1(...) {
   ...
  helper.SomeMoreCommonCode();
  }
...
} ;
```

…sounds like common sense?

Many refactoring techniques are just
good practices of code hygiene

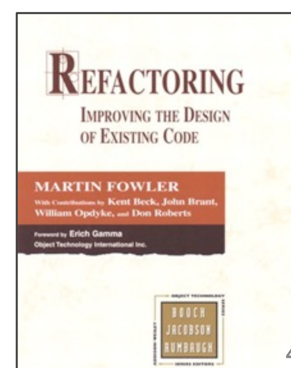
Apply them when writing new code!

The “legacy code” you may have to refactor in a few months/years may **be your own**…
A colleague collaborating at your project may have to refactor your “legacy code”…

Refactoring Techniques

Composing Methods	Extract Method, Inline Method,…
Moving Features Between Objects	Move Method, Move Field, Hide Delegate, ...
Organizing Data	Replace Data Value with Object, ...
Simplifying Conditional Expressions	Decompose Conditionals, ...

http://www.refactoring.com/catalog/index.html



Duplicated code

in the stink parade

- Same expression in two methods of the **same class**
  - Use **Extract Method** refactoring
- Same expression in two methods of **sibling classes**
  - Use **Extract Method** and **Pull Up Method**
- If code is similar, but not same
  - Consider **Form Template Method**
- Duplicated code in **unrelated classes**
  - May need to **Extract Class**
  - Or eliminate one of the versions

G4WentzelVIModel.cc

```
G4double G4WentzelVIModel::ComputeGeomPathLength(G4double truelength)
{
  tPathLength  = truelength;
  zPathLength  = tPathLength;

  if(lambdaeff > 0.0 && lambdaeff != DBL_MAX) {
    G4double tau = tPathLength/lambdaeff;
    //G4cout << "ComputeGeomPathLength: tLength= " << tPathLength
    //   << " Leff= " << lambdaeff << " tau= " << tau << G4endl;
    // small step
    if(tau < numlimit) {
      zPathLength *= (1.0 - 0.5*tau + tau*tau/6.0);

      // medium step
    } else {
      G4double e1 = 0.0;
      if(currentRange > tPathLength) {
    e1 = GetEnergy(particle,currentRange-tPathLength,currentCouple);
      }
      e1 = 0.5*(e1 + preKinEnergy);
      cosTetMaxNuc = wokvi->SetupKinematic(e1, currentMaterial);
      lambdaeff = GetTransportMeanFreePath(particle,e1);
      zPathLength = lambdaeff*(1.0 - G4Exp(-tPathLength/lambdaeff));
    }
  } else { lambdaeff = DBL_MAX; }
  //G4cout<<"Comp.geom: zLength= "<<zPathLength<<" tLength= "<<tPathLength<<G4endl;
  return zPathLength;
}
```

G4WentzelVIRelModel.cc

29

```
G4double G4WentzelVIRelModel::ComputeGeomPathLength(G4double truelength)
{
  tPathLength  = truelength;
  zPathLength  = tPathLength;

  if(lambdaeff > 0.0 && lambdaeff != DBL_MAX) {
    G4double tau = tPathLength/lambdaeff;
    //G4cout << "ComputeGeomPathLength: tLength= " << tPathLength
    //   << " Leff= " << lambdaeff << " tau= " << tau << G4endl;
    // small step
    if(tau < numlimit) {
      zPathLength *= (1.0 - 0.5*tau + tau*tau/6.0);

      // medium step
    } else {
      G4double e1 = 0.0;
      if(currentRange > tPathLength) {
    e1 = GetEnergy(particle,currentRange-tPathLength,currentCouple);
      }
      e1 = 0.5*(e1 + preKinEnergy);
      cosTetMaxNuc = wokvi->SetupKinematic(e1, currentMaterial);
      lambdaeff = GetTransportMeanFreePath(particle,e1);
      zPathLength = lambdaeff*(1.0 - G4Exp(-tPathLength/lambdaeff));
    }
  } else { lambdaeff = DBL_MAX; }
  //G4cout<<"Comp.geom: zLength= "<<zPathLength<<" tLength= "<<tPathLength<<G4endl;
  return zPathLength;
}
```

Identical member function!

G4PAIModel.cc

G4PAIPhotModel.cc

G4PAIModel.cc vs. G4PAIPhotModel.cc

G4PAIModel.cc – /Users/pia/Documents/g4dev/g4_svn/geant4.10.00.p01/source/processes/electromagnetic/standard/src

G4PAIModel::G4PAIModel()

```
}

/////////////////////////////////////////////////////////////////////////////

void G4PAIModel::Initialise(const G4ParticleDefinition* p,
                const G4DataVector& cuts)
{
  if(fVerbose > 0) {
    G4cout<<"G4PAIModel::Initialise for "<<p->GetParticleName()<<G4endl;
  }

  if(isInitialised) { return; }
  isInitialised = true;

  SetParticle(p);
  fParticleChange = GetParticleChangeForLoss();

  if(IsMaster()) {

    InitialiseElementSelectors(p, cuts);

    if(!fModelData) {

      G4double tmin = LowEnergyLimit()*fRatio;
      G4double tmax = HighEnergyLimit()*fRatio;
      fModelData = new G4PAIModelData(tmin, tmax, fVerbose);
    }
    // Prepare initialization
    const G4MaterialTable* theMaterialTable = G4Material::GetMaterialTable();
    size_t numOfMat   = G4Material::GetNumberOfMaterials();
    size_t numRegions = fPAIRegionVector.size();

    for(size_t iReg = 0; iReg < numRegions; ++iReg) {
      const G4Region* curReg = fPAIRegionVector[iReg];
      G4Region* reg = const_cast<G4Region*>(curReg);

      for(size_t jMat = 0; jMat < numOfMat; ++jMat) {
    G4Material* mat = (*theMaterialTable)[jMat];
    const G4MaterialCutsCouple* cutCouple = reg->FindCouple(mat);
    //G4cout << "Couple <" << fCutCouple << G4endl;
    if(cutCouple) {
      /*
        G4cout << "Reg <" <<curReg->GetName() << ">  mat <"
        << fMaterial->GetName() << ">  fCouple= "
        << fCutCouple << " idx= " << fCutCouple->GetIndex()
        <<"  " << p->GetParticleName() <<G4endl;
        // G4cout << cuts.size() << G4endl;
        */
      // check if this couple is not already initialized
      size_t n = fMaterialCutsCoupleVector.size();
      if(0 < n) {
        for(size_t i=0; i<fMaterialCutsCoupleVector.size(); ++i) {
          if(cutCouple == fMaterialCutsCoupleVector[i]) {
        break;
          }
        }
      }
      // initialise data banks
      fMaterialCutsCoupleVector.push_back(cutCouple);
      G4double deltaCutInKinEnergy = cuts[cutCouple->GetIndex()];
      fModelData->Initialise(cutCouple, deltaCutInKinEnergy, this);
    }
      }
    }
  }
}

/////////////////////////////////////////////////////////////////////////////
```

G4PAIPhotModel.cc – /Users/pia/Documents/g4dev/g4_svn/geant4.10.00.p01/source/processes/electromagnetic/standard/src

G4PAIPhotModel::G4PAIPhotModel()

```
/////////////////////////////////////////////////////////////////////////////

void G4PAIPhotModel::Initialise(const G4ParticleDefinition* p,
                const G4DataVector& cuts)
{
  // if(fVerbose > 0)
  {
    G4cout<<"G4PAIPhotModel::Initialise for "<<p->GetParticleName()<<G4endl;
  }

  if(isInitialised) { return; }
  isInitialised = true;

  SetParticle(p);
  fParticleChange = GetParticleChangeForLoss();

  if(IsMaster()) {

    InitialiseElementSelectors(p, cuts);

    if(!fModelData) {

      G4double tmin = LowEnergyLimit()*fRatio;
      G4double tmax = HighEnergyLimit()*fRatio;
      fModelData = new G4PAIPhotData(tmin, tmax, fVerbose);
    }
    // Prepare initialization
    const G4MaterialTable* theMaterialTable = G4Material::GetMaterialTable();
    size_t numOfMat   = G4Material::GetNumberOfMaterials();
    size_t numRegions = fPAIRegionVector.size();

    for(size_t iReg = 0; iReg < numRegions; ++iReg) {
      const G4Region* curReg = fPAIRegionVector[iReg];
      G4Region* reg = const_cast<G4Region*>(curReg);

      for(size_t jMat = 0; jMat < numOfMat; ++jMat) {
    G4Material* mat = (*theMaterialTable)[jMat];
    const G4MaterialCutsCouple* cutCouple = reg->FindCouple(mat);
    //G4cout << "Couple <" << fCutCouple << G4endl;
    if(cutCouple) {
      /*
        G4cout << "Reg <" <<curReg->GetName() << ">  mat <"
        << fMaterial->GetName() << ">  fCouple= "
        << fCutCouple << " idx= " << fCutCouple->GetIndex()
        <<"  " << p->GetParticleName() <<G4endl;
        // G4cout << cuts.size() << G4endl;
        */
      // check if this couple is not already initialized
      size_t n = fMaterialCutsCoupleVector.size();
      if(0 < n) {
        for(size_t i=0; i<fMaterialCutsCoupleVector.size(); ++i) {
          if(cutCouple == fMaterialCutsCoupleVector[i]) {
        break;
          }
        }
      }
      // initialise data banks
      fMaterialCutsCoupleVector.push_back(cutCouple);
      G4double deltaCutInKinEnergy = cuts[cutCouple->GetIndex()];
      fModelData->Initialise(cutCouple, deltaCutInKinEnergy, this);
    }
      }
    }
  }
}

/////////////////////////////////////////////////////////////////////////////
```

11

12

13

Identical but the instantiation of a "model data" object

How to find duplicated code?

- Automated tools
  - Some exist
- By hand
  - Still the most common way
  - Not necessarily the most efficient
    (nor the most inefficient)
- **Reverse engineering**
  - Gain understanding of the code

G4KleinNishinaCompton.cc

```
G4double G4KleinNishinaCompton::ComputeCrossSectionPerAtom(
                                        const G4ParticleDefinition*,
                                              G4double GammaEnergy,
                                              G4double Z, G4double,
                                              G4double, G4double)
{
  G4double xSection = 0.0 ;
  if ( Z < 0.9999 )                 return xSection;
  if ( GammaEnergy < 0.1*keV      ) return xSection;
  //  if ( GammaEnergy > (100.*GeV/Z) ) return xSection;

  static const G4double a = 20.0 , b = 230.0 , c = 440.0;

  G4double p1Z = Z*(d1 + e1*Z + f1*Z*Z), p2Z = Z*(d2 + e2*Z + f2*Z*Z),
           p3Z = Z*(d3 + e3*Z + f3*Z*Z), p4Z = Z*(d4 + e4*Z + f4*Z*Z);

  G4double T0  = 15.0*keV;
  if (Z < 1.5) T0 = 40.0*keV;

  G4double X   = max(GammaEnergy, T0) / electron_mass_c2;
  xSection = p1Z*G4Log(1.+2.*X)/X
               + (p2Z + p3Z*X + p4Z*X*X)/(1. + a*X + b*X*X + c*X*X*X);

  //  modification for low energy. (special case for Hydrogen)
  if (GammaEnergy < T0) {
    G4double dT0 = 1.*keV;
    X = (T0+dT0) / electron_mass_c2 ;
    G4double sigma = p1Z*G4Log(1.+2*X)/X
                    + (p2Z + p3Z*X + p4Z*X*X)/(1. + a*X + b*X*X + c*X*X*X);
    G4double   c1 = -T0*(sigma-xSection)/(xSection*dT0);
    G4double   c2 = 0.150;
    if (Z > 1.5) c2 = 0.375-0.0556*G4Log(Z);
    G4double     y = G4Log(GammaEnergy/T0);
    xSection *= G4Exp(-y*(c1+c2*y));
  }
  //  G4cout << "e= " << GammaEnergy << " Z= " << Z << " cross= " << xSection << G4endl;
  return xSection;
}

//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo....
```

G4KleinNishinaModel.cc

```
//....oooOO0OOooo........oooOO0OOooo........oooOO0OOooo........oooOO0OOooo....

G4double
G4KleinNishinaModel::ComputeCrossSectionPerAtom(const G4ParticleDefinition*,
                                                G4double GammaEnergy,
                                                G4double Z, G4double,
                                                G4double, G4double)
{
  G4double xSection = 0.0 ;
  if ( Z < 0.9999 || GammaEnergy < 0.1*keV) { return xSection; }

  static const G4double a = 20.0 , b = 230.0 , c = 440.0;

  G4double p1Z = Z*(d1 + e1*Z + f1*Z*Z), p2Z = Z*(d2 + e2*Z + f2*Z*Z),
           p3Z = Z*(d3 + e3*Z + f3*Z*Z), p4Z = Z*(d4 + e4*Z + f4*Z*Z);

  G4double T0  = 15.0*keV;
  if (Z < 1.5) { T0 = 40.0*keV; }

  G4double X   = max(GammaEnergy, T0) / electron_mass_c2;
  xSection = p1Z*G4Log(1.+2.*X)/X
               + (p2Z + p3Z*X + p4Z*X*X)/(1. + a*X + b*X*X + c*X*X*X);

  //  modification for low energy. (special case for Hydrogen)
  if (GammaEnergy < T0) {
    G4double dT0 = keV;
    X = (T0+dT0) / electron_mass_c2 ;
    G4double sigma = p1Z*G4Log(1.+2*X)/X
                    + (p2Z + p3Z*X + p4Z*X*X)/(1. + a*X + b*X*X + c*X*X*X);
    G4double   c1 = -T0*(sigma-xSection)/(xSection*dT0);
    G4double   c2 = 0.150;
    if (Z > 1.5) { c2 = 0.375-0.0556*G4Log(Z); }
    G4double     y = G4Log(GammaEnergy/T0);
    xSection *= G4Exp(-y*(c1+c2*y));
  }

  if(xSection < 0.0) { xSection = 0.0; }
  //  G4cout << "e= " << GammaEnergy << " Z= " << Z
  //  << " cross= " << xSection << G4endl;
  return xSection;
}
```

17 18 19 20 21 22 23

Beware:

Sometimes automated tools don't catch code duplication effectively

Code reviews!

Long method

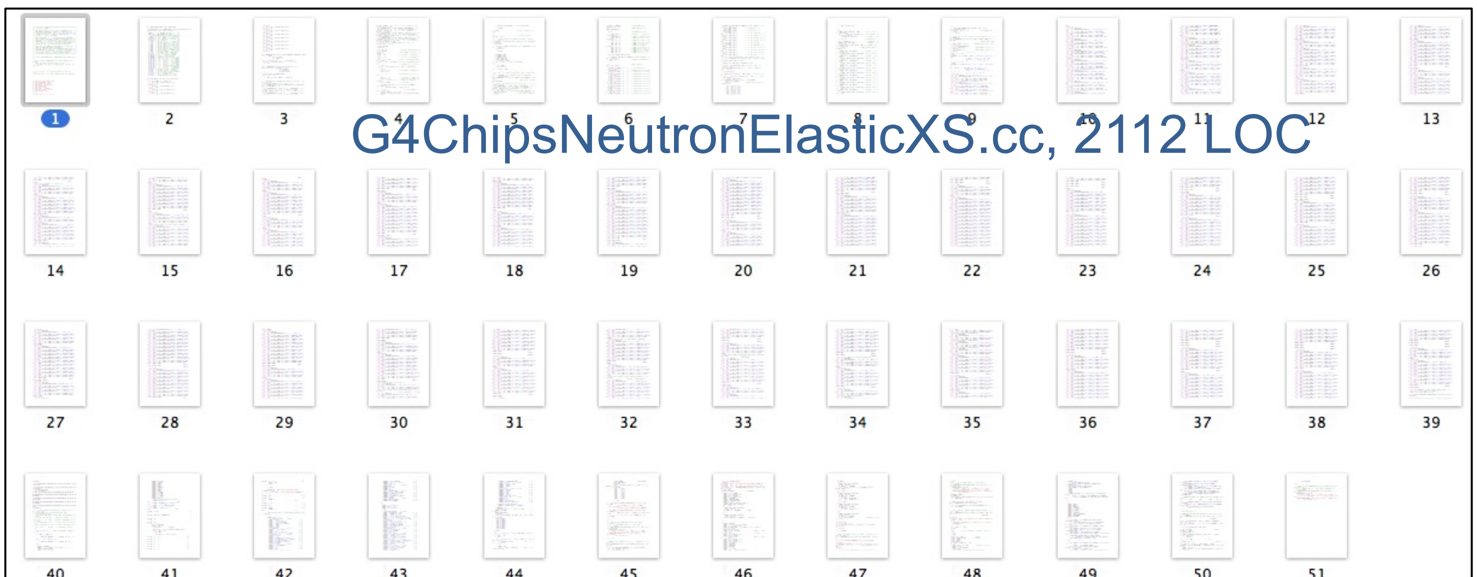
- The longer a method is, the more difficult it is to understand

- Decomposing methods
- Most of the time: just **Extract Method**
  - What to extract?
  - Understand what the code does
  - Comments in the code may help

Large class

- A class that tries to do too much
- Often has too many instance variables
- Prone to duplicated code
- **Extract Class**
- **Extract SubClass**
- **Extract Interface**

G4VEnergyLossProcess

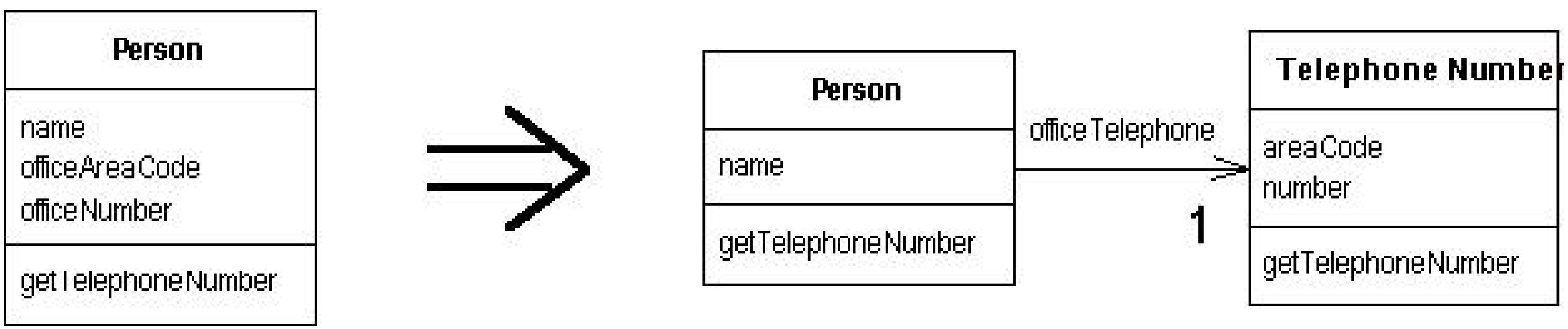


G4ChipsNeutronElasticXS.cc, 2112 LOC

Extract class

One class does work that should be done by two

Create a new class and move the relevant fields and methods from the old class into the new class

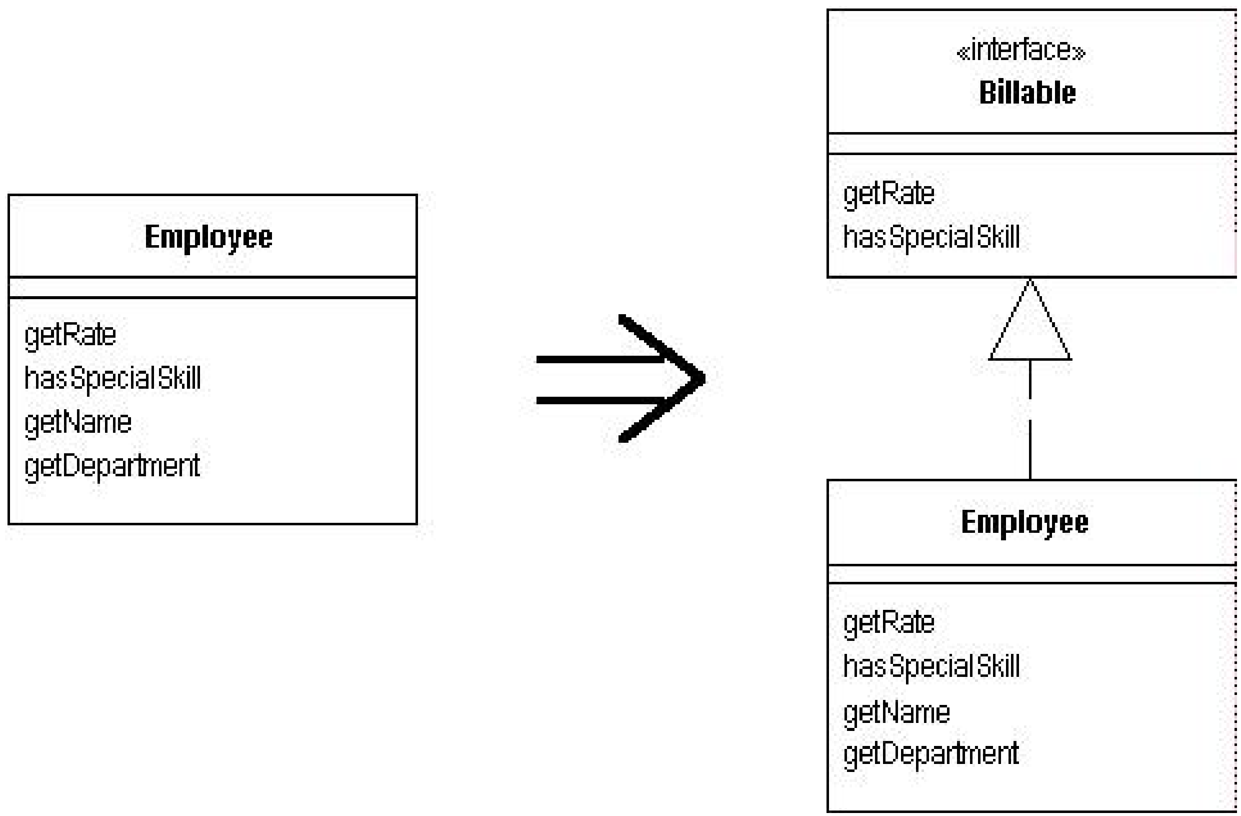


hands-on exercise

Extract interface

Several clients use the same subset of a class's interface, or two classes have part of their interfaces in common

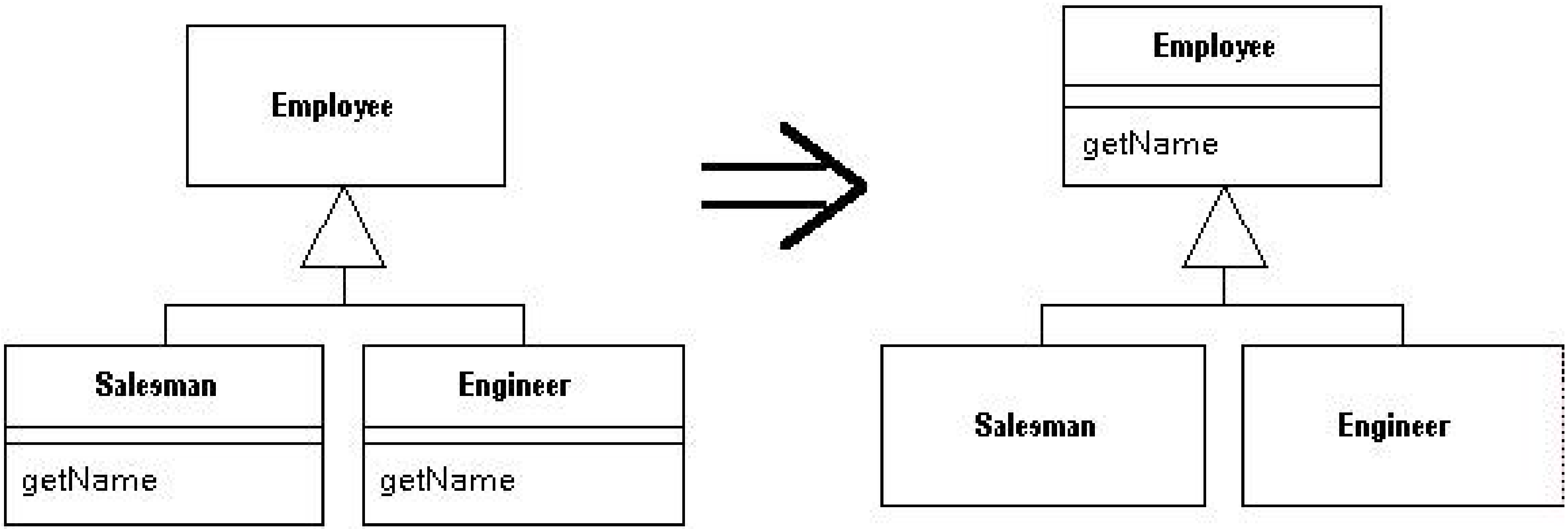
Extract the subset into an abstract interface



Pull up method

Methods with identical results on subclasses

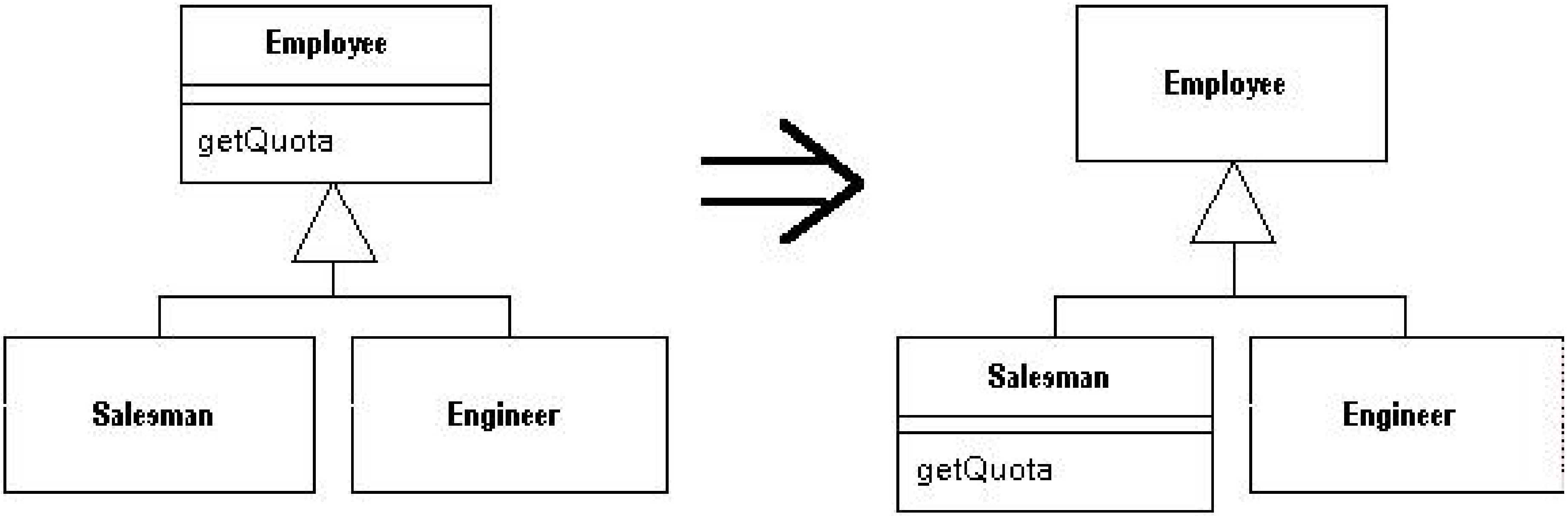
Move them to the superclass



Push down method

Behavior in a superclass is relevant only for some of its subclasses

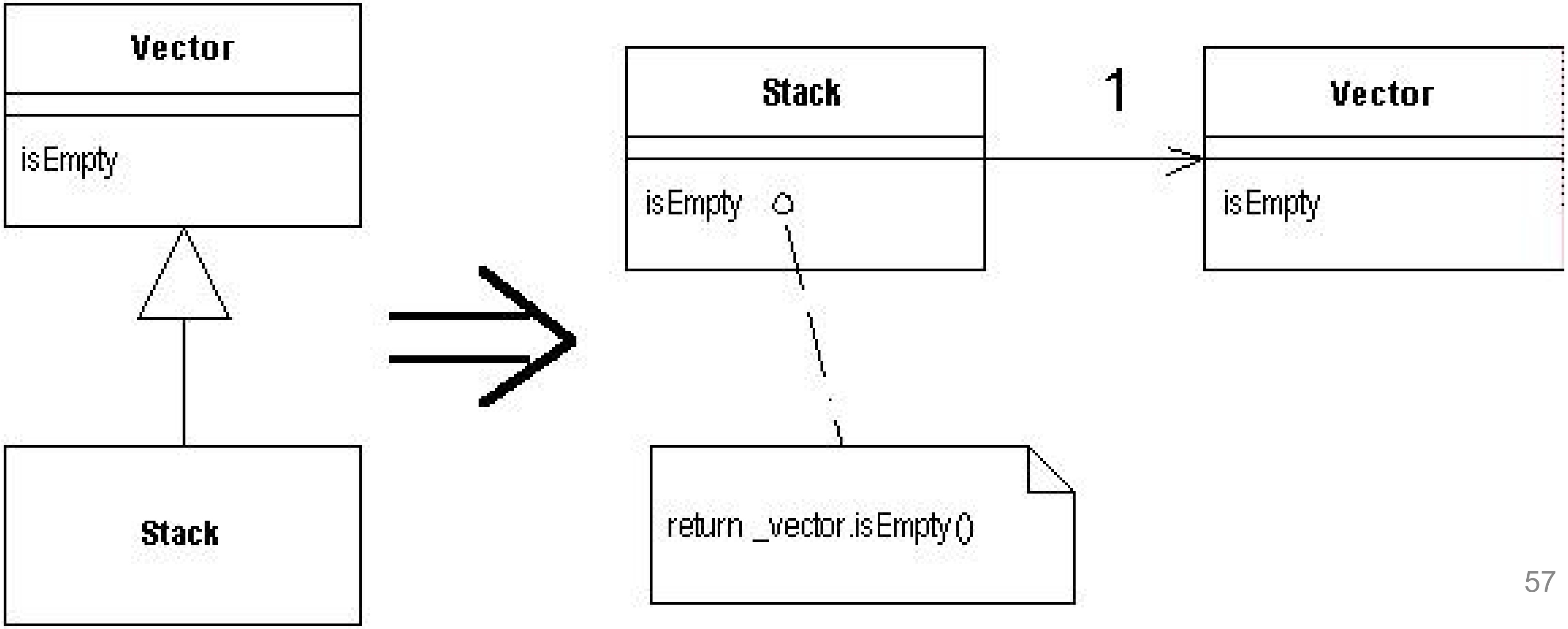
Move it to those subclasses



Replace inheritance with delegation

A subclass uses only part of a superclass interface
or does not want to inherit data

**Create a field for the superclass,
adjust methods to delegate to the superclass,
and remove the subclassing**



Refactoring mechanics

Catalog of refactoring techniques

Reference material in Fowler’s book and web site

Not meant to be learned by heart,
but to be exercised when a pertinent “smell” is identified

Composing Methods

- **Extract Method** turns a code fragment into a function
- **Inline Method** is the opposite of Extract Method
- **Inline Temp** gets rid of a temporary variable by moving the expression to where the temp is used
- **Replace Temp with Query** removes a temporary variable and instead uses a function call where the temp was used
- **Introduce Explaining Variable** replaces comments and complex expressions with a temp variable that is well named
- **Split Temporary Variable** splits a temp that is used for two different things into two different variables
- **Remove Assignments to Parameters** removes assignments to function parameters within the function
- **Replace Method with Method Object** moves a complex function to its own class
- **Substitute Algorithm**

Moving Features Between Objects

- **Move Method** moves a method from one class to a more appropriate class
- **Move Field** moves a field (member object) from one class to another class
- **Extract Class** pulls a set of methods and fields from one class into a new class
- **Inline Class** opposite of Extract Class
- **Hide Delegate:** a class that provides access to an object of another class instead provides the methods of that class by delegation
- **Remove Middle Man** opposite of Hide Delegate
- **Introduce Foreign Method** adds a method to an untouchable class by passing an instance of the untouchable class into the method
- **Introduce Local Extension** adds methods to an untouchable class by deriving from it or by wrapping it

Organizing Data

- **Self Encapsulate Field** creates accessors for private member objects
- **Replace Data Value with Object** turns a member object into a full-fledged class
- **Change Value to Reference**
- **Change Reference to Value**
- **Replace Array with Object** converts an array which has various fields in each entry into an object
- **Duplicate Observed Data** introduces a Document/View architecture into an interactive application
- **Change Unidirectional Association to Bidirectional** introduces a back pointer
- **Change Bidirectional Association to Unidirectional** is the opposite
- **Replace Magic Number with Symbolic Constant**

Organizing Data

- **Encapsulate Field** adds getters and setters
- **Encapsulate Collection** hides a collection within a class
- **Replace Record with Data Class** makes a dumb data class to represent a record structure
- **Replace Type Code with Class** replaces an enumeration type with a class that has a set of global instances of itself, one for each possible value
- **Replace Type Code with Subclasses** introduces a polymorphic hierarchy to replace an enumeration
- **Replace Type Code with State/Strategy** allows change at runtime
- **Replace Subclass with Fields** is used when the subclasses no longer serve any real purpose

Simplifying Conditional Expressions

- **Decompose Conditional** extracts the condition, the "then" part, and the "else" part into functions
- **Consolidate Conditional Expression** combines a series of "if" into one
- **Consolidate Duplicate Conditional Fragments** factors out code that is common to a "then" part and an "else" part
- **Remove Control Flag** replaces flags that trigger exits with return, continue, and break
- **Replace Nested Conditional with Guard Clauses** replaces nested "if" with returns
- **Replace Conditional with Polymorphism** replaces case statements with polymorphism
- **Introduce Null Object** replaces checks for null values with an object
- **Introduce Assertion** uses assertions to describe a function's preconditions

Making Method Calls Simpler

- **Rename Method**
- **Add Parameter** to a function
- **Remove Parameter** from a function
- **Separate Query from Modifier** to avoid side-effects
- **Parametrize Method** reduces a set of similar functions to a single function with a parameter to differentiate amongst the functions
- **Replace Parameter with Explicit Methods**
- **Preserve Whole Object** passes an object to a method instead of selected fields
- **Replace Parameter with Method** reduces a parameter list by using a value that is already available within the class
- **Introduce Parameter Object** groups parameters into a single object
- **Remove Setting Method** makes an attribute read-only
- **Hide Method** makes a method private
- **Replace Constructor with Factory Method** supports polymorphism
  Encapsulate Downcast hides a downcast within a method
- **Replace Error Code with Exception** separates error-handling from normal paths
- **Replace Exception with Test** provides a method for caller to avoid an exception

Dealing with Generalization

- **Pull Up Field** factors a common field into a superclass
- **Pull Up Method** factors a common method into a superclass
- **Push Down Method** moves a unique method down into a subclass
- **Push Down Field** moves a unique field down into a subclass
- **Extract Subclass** creates a subclass and moves features into it
- **Extract Superclass** factors common code into a superclass
- **Extract Interface** promotes decoupling and partitioning of a class's responsibilities by extracting an interface class
- **Collapse Hierarchy** combines a subclass and a superclass into one
- **Form Template Method** factors out common behavior into a superclass
- **Replace Inheritance with Delegation Replace Delegation with Inheritance**

Big Refactorings

- **Tease Apart Inheritance**
  - deals with a messy inheritance hierarchy
- **Convert Procedural Design to Objects**
- **Separate Domain from Presentation**
  - moves domain logic out of the UI classes
- **Extract Hierarchy**
  - introduces polymorphism to replace complex conditional code

Computational performance

Refactoring may make the code slower

conventional wisdom

Yes, sometimes…

“First ~~do it, then~~ do it right, then do it fast”

Refactoring **prepares the ground**
for computational improvement
by providing **clean code**

Refactoring and reengineering physics software

Food for thought…

Future Generation Computer Systems 159 (2024) 411–422

Contents lists available at ScienceDirect

Future Generation Computer Systems

journal homepage: www.elsevier.com/locate/fgcs

ELSEVIER

Geant4: A game changer in high energy physics and related applicative fields

Tullio Basaglia [a,1], Zane W. Bell [b], Daniele D'Agostino [c,d,*], Paul V. Dressendorfer [b], Simone Giani [a], Maria Grazia Pia [d], Paolo Saracco [d]

[a] European Organization for Nuclear Research (CERN), Geneva, CH-1211 Geneva 23, Switzerland
[b] IEEE, 445 Hoes Lane, Piscataway, NJ 08854, USA
[c] Department of Informatics, Bioengineering, Robotics and Systems Engineering (DIBRIS), Università degli Studi di Genova, Via Dodecaneso 35, Genova, Italy
[d] Istituto Nazionale di Fisica Nucleare (INFN), Sezione di Genova, Via Dodecaneso 33, Genova, Italy

ARTICLE INFO

Keywords:
High energy physics
Software engineering
C++
Monte Carlo
Geant4

ABSTRACT

Geant4 is an object-oriented toolkit for the simulation of the passage of particles through matter. Its development was initially motivated by the requirements of physics experiments at high energy hadron colliders under construction in the last decade of the 20th century. Since its release in 1998, it has been exploited in many different applicative fields, including space science, nuclear physics, medical physics and archaeology. Its valuable support to scientific discovery is demonstrated by more than 16000 citations received in the past 25 years, including notable citations for main discoveries in different fields. This accomplishment shows that well designed software plays a key role in enabling scientific advancement. In this paper we discuss the key principles and the innovative decisions at the basis of Geant4, which made it a game changer in high energy physics and related fields, and outline some considerations regarding future directions.

Smells

Duplicated code
Long method
Large class
Long parameter list
Shotgun surgery
Feature envy
Data clumps
Switch statement
Parallel inheritance hierarchies
Lazy class
Speculative generality
Temporary field
Comments
Refused bequest
Primitive obsession
Message chains
Middle man
Inappropriate intimacy
...

Evolution away from RD44 discipline

Maria Grazia Pia, *INFN Genova*

Electromagnetic smells

Coupling

σ_{tot} and final state modeling have been decoupled in hadronic physics design since RD44

"model"

Total cross section

Whether a process occurs

Final state generation

How a process occurs

Dependencies

on other parts of the software

One needs a geometry

(a full-scale simulation application)

to test a cross section

often

Difficult to test ➔ no testing

Problem domain analysis, improve domain decomposition and software design

Magic number

```
// G4HadronElastic
// Author : […]  29 June 2009 (redesign old elastic model)
```

```
G4double dd = 10.;
G4Pow* g4pow = G4Pow::GetInstance();
if (A <= 62) {
bb = 14.5*g4pow->Z23(A);
aa = g4pow->powZ(A, 1.63)/bb;
cc = 1.4*g4pow->Z13(A)/dd;
} else {
bb = 60.*g4pow->Z13(A);
aa = g4pow->powZ(A, 1.33)/bb;
cc = 0.4*g4pow->powZ(A, 0.4)/dd;
}
G4double q1 = 1.0 - std::exp(-bb*tmax);
G4double q2 = 1.0 - std::exp(-dd*tmax);
G4double s1 = q1*aa;
G4double s2 = q2*cc;
```

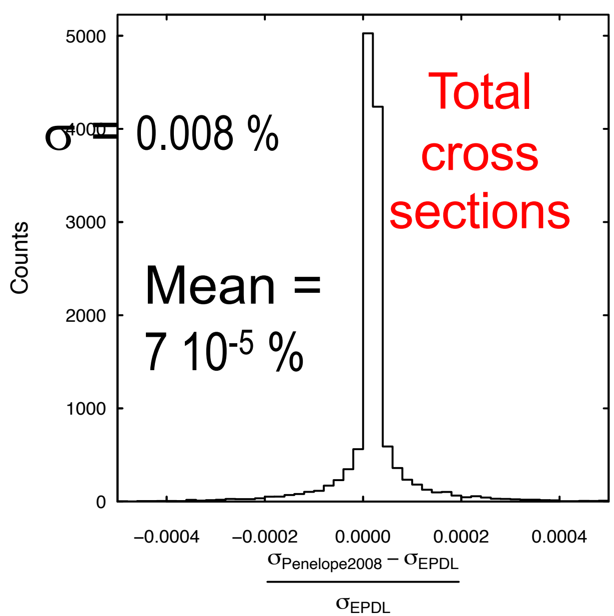
M. Fowler,
Refactoring

Number one in the stink parade is **duplicated ~~code~~** **physics**

Two Geant4 physics models, identical underlying physics content *(it used to be different)*

"Livermore"	Penelope
EPDL97	EPDL97
0.38±0.06	0.38±0.06

Same compatibility with experiment



Code bloat

Burden on

- Software design
- Maintenance
- User support

Unnecessary complexity

M. Fowler, *Refactoring*

Number one in the stink parade is **duplicated ~~code~~ numbers**

Geant 4 Atomic binding energies

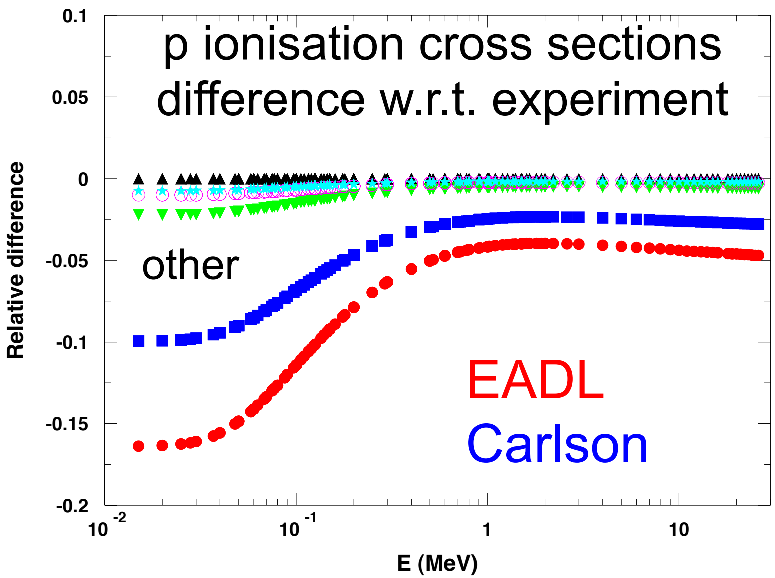
1. Bearden & Burr (1967)
2. Carlson
3. EADL
4. Sevier
5. ToI 1978 (Shirley)
6. ToI 1996 (Larkins)
7. Williams

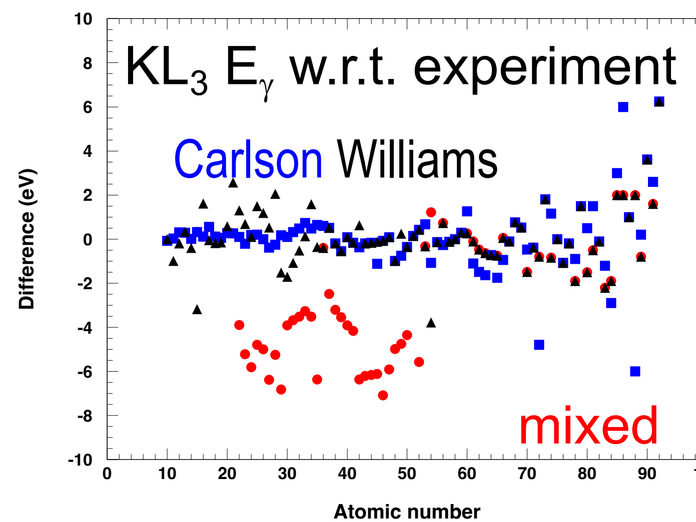
Source of epistemic uncertainties?

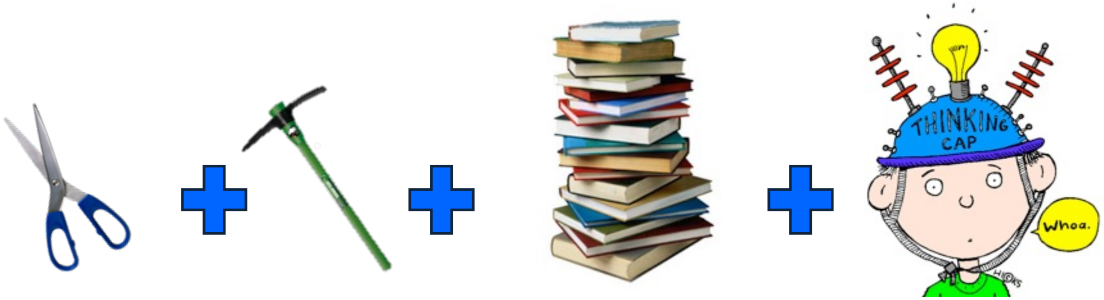
3246 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 58, NO. 6, DECEMBER 2011

Evaluation of Atomic Electron Binding Energies for Monte Carlo Particle Transport

Maria Grazia Pia, Hee Seo, Matej Batic, Marcia Begalli, Chan Hyeong Kim, Lina Quintieri, and Paolo Saracco







Reengineered code

Benefits

Transparency

Ease of **maintenance**

Simplicity of **testing** for V&V

Basic physics V&V can be performed by means of **lightweight unit tests**

Exploring new physics models is made easier

Quantification of accuracy and performance is facilitated

An example of reengineering + new functionality

Photon elastic scattering simulation

1636 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 59, NO. 4, AUGUST 2012

Photon Elastic Scattering Simulation: Validation and Improvements to Geant4

Matej Batič, Gabriela Hoff, Maria Grazia Pia, and Paolo Saracco

State of the art

Form factor approximation:
Non-relativistic, relativistic, modified + anomalous scattering factors

2nd order S-matrix calculations
recent calculations, not yet used in Monte Carlo codes

Quantification

Statistical analysis, GoF + categorical

in Geant4 **Differential cross sections** new

	Penelope 2001	Penelope 2008	EPDL	Relativ. FF	Non-Rel. FF	Modified FF	MFF ASF	RFF ASF	SM NT
ε	0.27	0.38	**0.38**	0.25	0.35	0.49	0.52	0.48	**0.77**
error	±0.05	±0.06	±0.06	±0.05	±0.06	±0.06	±0.06	±0.06	±0.05

ε = fraction of test cases not incompatible with experiment, 0.01 significance

Computational performance

Popular belief

Physics model X is intrinsically slow

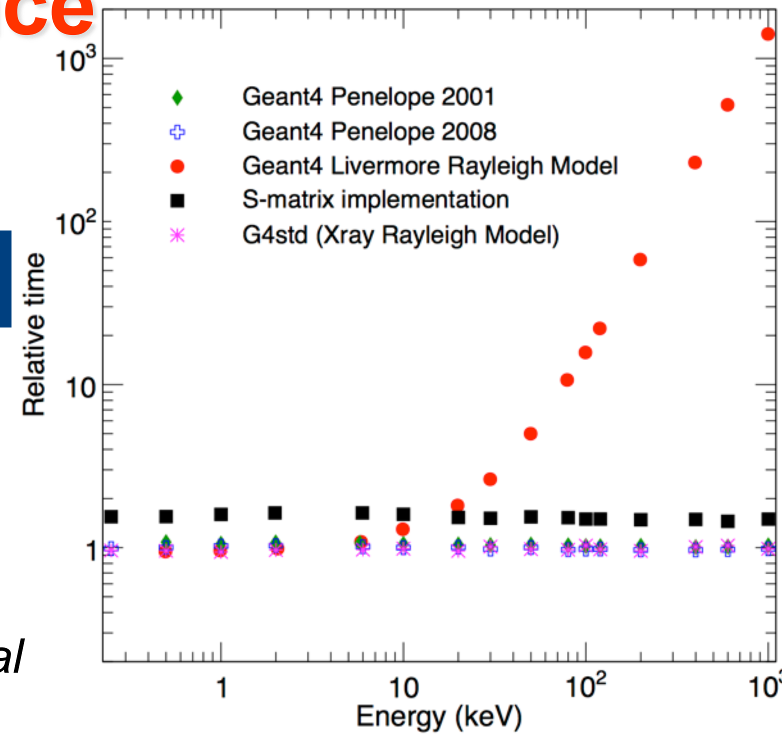
Baroque methods to combine it with “faster” lower precision models and limit its use to cases where one is willing to pay for higher precision

This design introduces an additional computational burden due to the effects of inheritance and the combination algorithms themselves

Truth

Physics model X is intrinsically fast

But its computationally fast physics functionality is spoiled by an inefficient sampling algorithm



- **No code smell**
- Spotted through in-depth **code review** in the course of **software validation**

Change the sampling algorithm!

It works
It doesn't hurt… so long as you don't want to change it

Most of the fear involved in making changes to large software systems is **fear of introducing bugs**

The legacy code dilemma

When we change code, we should have tests in place	To put tests in place, we often need to change code

With tests,
you can change (and improve) your code

Without tests, you just don't know whether things are getting better or worse

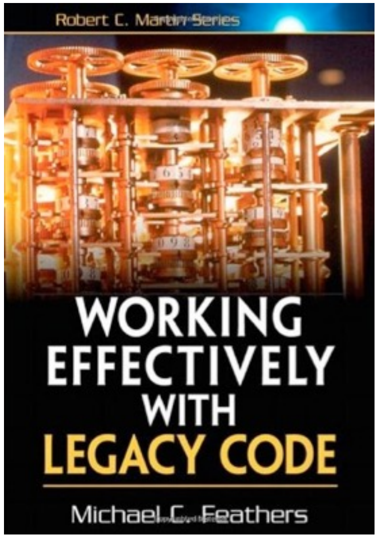
Dealing with legacy code

Legacy code often lacks tests

Techniques to make existing code testable

Strategy

1. Identify change points
2. Find an inflection point
3. Cover the inflection point
    a. Break external dependencies
    b. Break internal dependencies
    c. Write tests
4. Make changes
5. Refactor the covered code



Prevention

- Most of these problems can be easily solved if we simply write tests as we develop our code *(and we maintain them…)*
- If a test is hard to write, that means that we have to **devise a different design** which **is testable**
- It is always possible

Provocative thought…

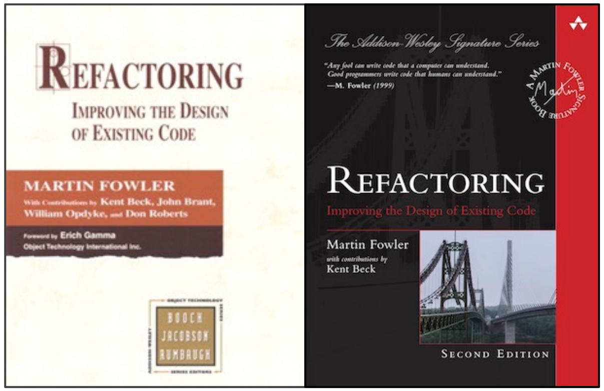
Need to refactor legacy code due to:

- Requirements change
- Computing environment changes *(compilers, language standards…)*
- New technology becomes available

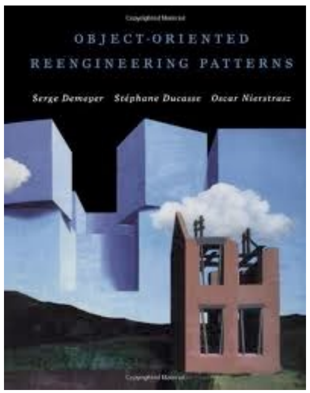
Need to refactor legacy code due to:

- Laziness to adopt a sound software development process
- Sloppiness
- Refuse to invest in learning technology
- Contempt for good practices
- Lack of design and code reviews
- Lack of adequate mentoring
- …

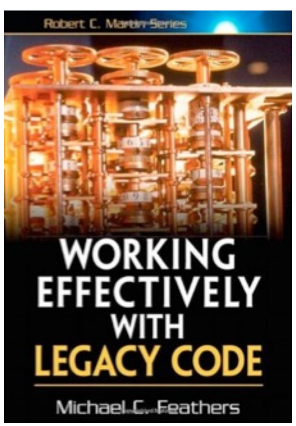
Books



Martin Fowler et al.
Refactoring: Improving the Design of Existing Code
Addison-Wesley, 1999/2018
http://www.refactoring.com/

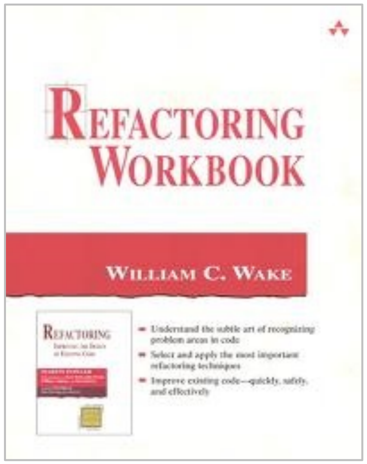


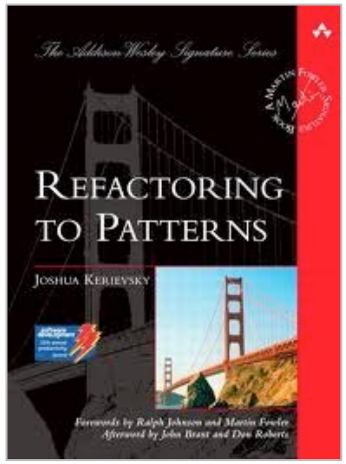
Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Object Oriented Reengineering Patterns
Morgan-Kaufmann, 2003
Revised 2008: http://www.iam.unibe.ch/~scg/OORP

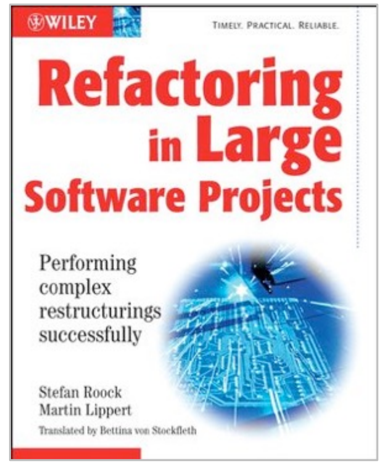


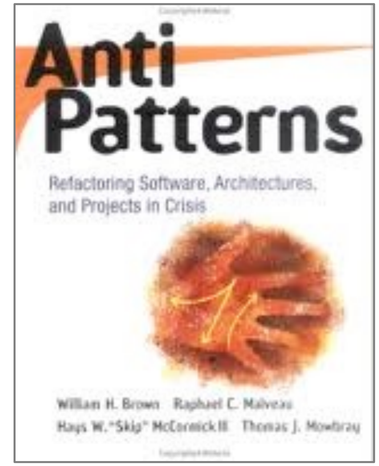
Micheal Feathers
Working Effectively with Legacy Code
Prentice Hall, 2005

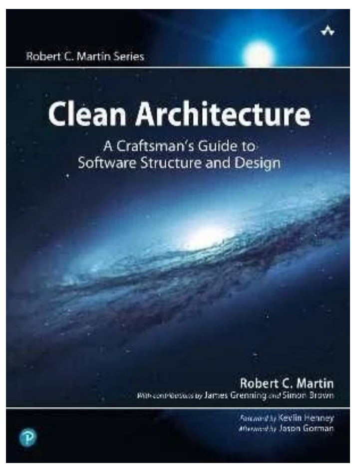
Other useful books











Get a mentor!

Conclusions

Dealing with legacy code in a disciplined, effective way

Methods
Techniques
Sources for further learning

Keep in mind peculiarities of physics software!

Refactoring techniques and reengineering patterns support software **maintenance**, **evolution** and **validation**
They may also contribute to improving **computational performance**

Thorough testing is the key

…but also sound background in OO methods,
healthy software engineering practices and physics insight

Hands-on exercises

- **“Video store” example**
  - M. Fowler, Refactoring, chapter 1
- Simple problem domain
- Practice a few refactoring techniques
- Learn the method
  - Write tests, change in small steps, test, change, test…
- *We’ll figure out together how many steps we do as an exercise and how many we’ll just review*