Advanced Programming Concepts

8 - 12 October 2012

DESY, Hamburg

Refactoring

Maria Grazia Pia

INFN Genova, Italy

Maria.Grazia.Pia@cern.ch

http://www.ge.infn.it/geant4/training/DESY2012/

"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

- 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**

Warnings you are heading into trouble

usually do not occur isolated

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

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.

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

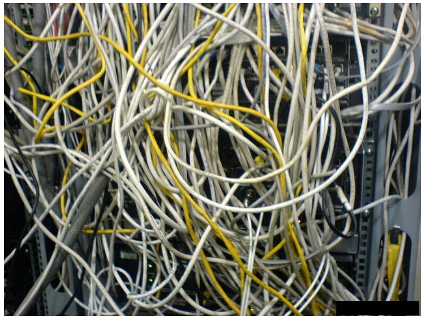
S. Demeyer, S. Ducasse, O. Nierstrasz,
Object Oriented Reengineering Patterns

software evolution ↔ **legacy software**

Methods and techniques

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

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



Refactoring

Reengineering

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**

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 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*

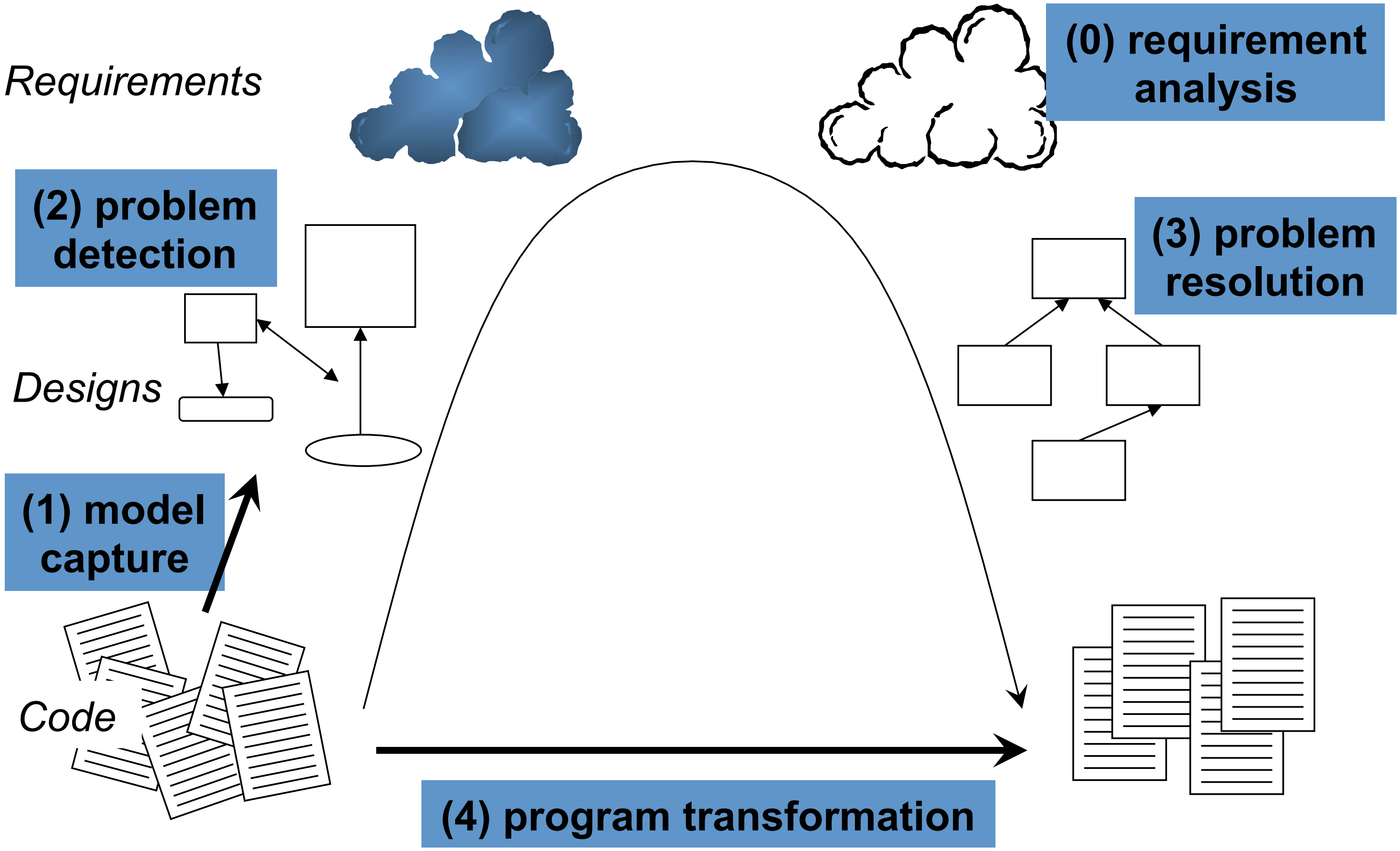
- Accommodate changes in the software environment
- Incorporate new user requirements
- Fix errors
- Prevent future problems

OO techniques promise better

- flexibility,
- reusability,
- maintainability
- …

but they do not come for free!

Reengineering



Requirements

(0) requirement analysis

(2) problem detection

(3) problem resolution

Designs

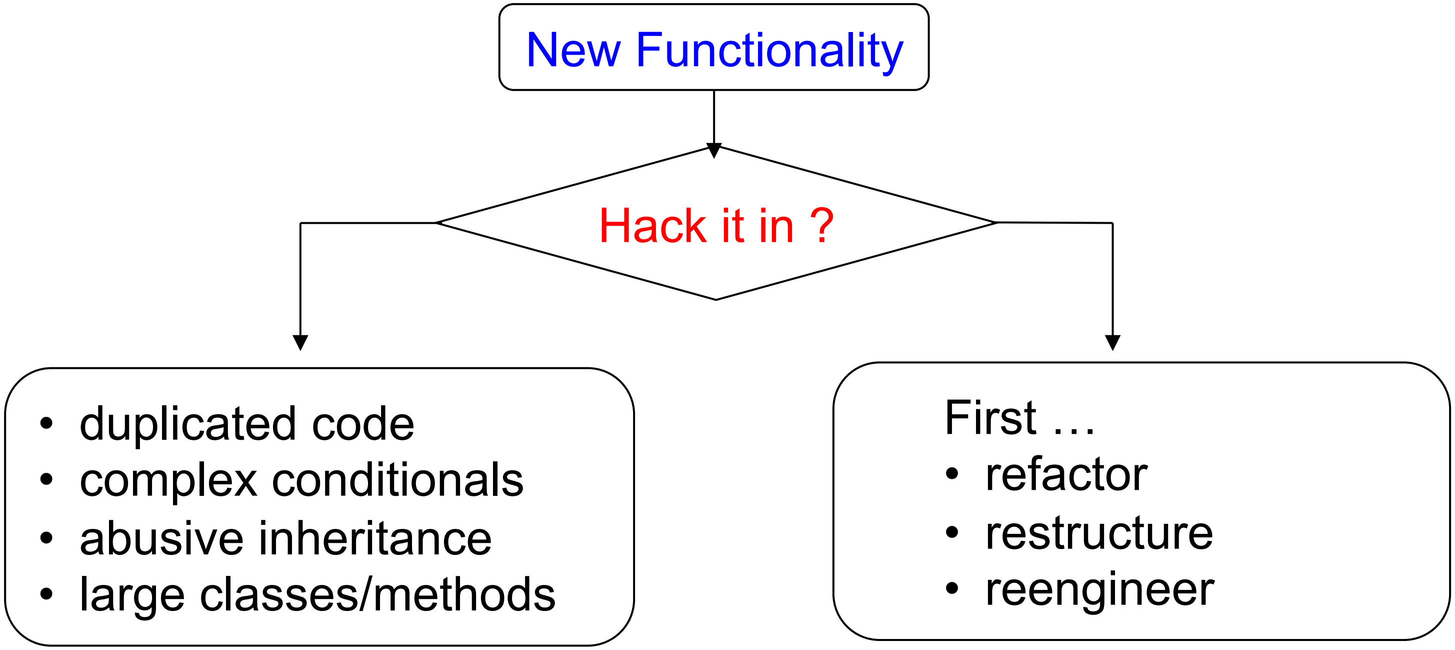
(1) model capture

Code

(4) program transformation

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

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

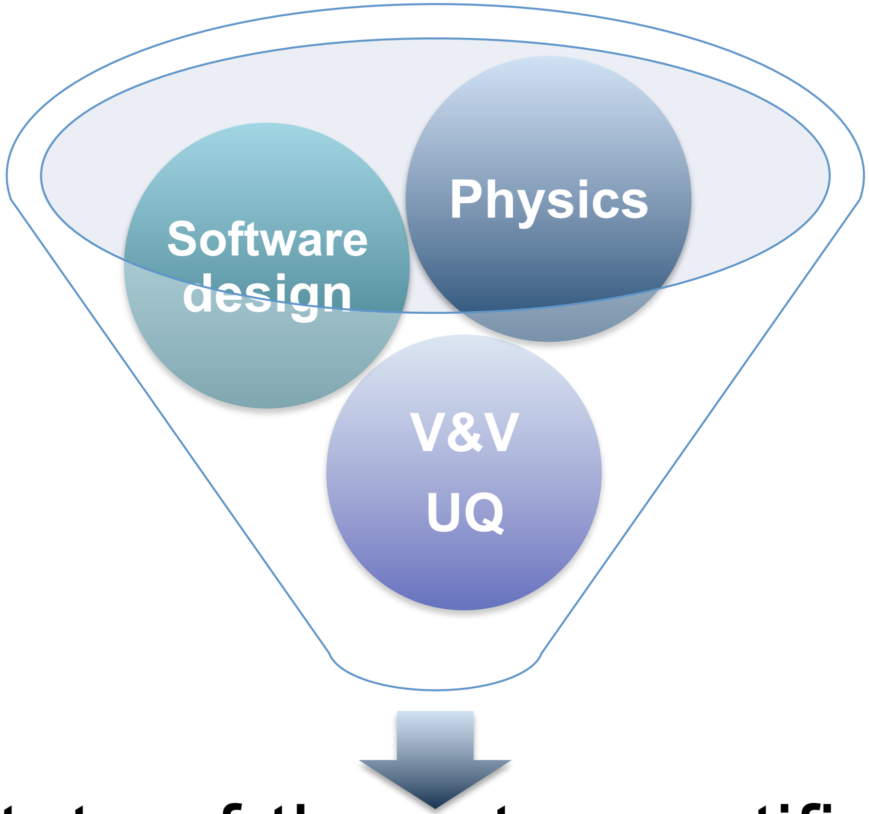
Conclusion

Does it pay back?

M. Batic, M. Begalli, M. Han, S. Hauf, G. Hoff, C. H. Kim, M. Kuster, M. G. Pia, P. Saracco, H. Seo, G. Weidenspointner, A. Zoglauer
Refactoring, reengineering and evolution: paths to Geant4 uncertainty quantification and performance improvement
To be published in the Proc. CHEP (Computing in High Energy Physics) 2012
http://arxiv.org/abs/1209.5989

Geant 4

S. Agostinelli et al., **Geant4: a simulation toolkit,**
NIM A, vol. 506, pp. 250-303, 2003
>3000 citations, most cited CERN paper



State-of-the-art, quantified simulation

R&D Project

Series of **pilot projects** going on since 2008: refactoring, reengineering, evolution

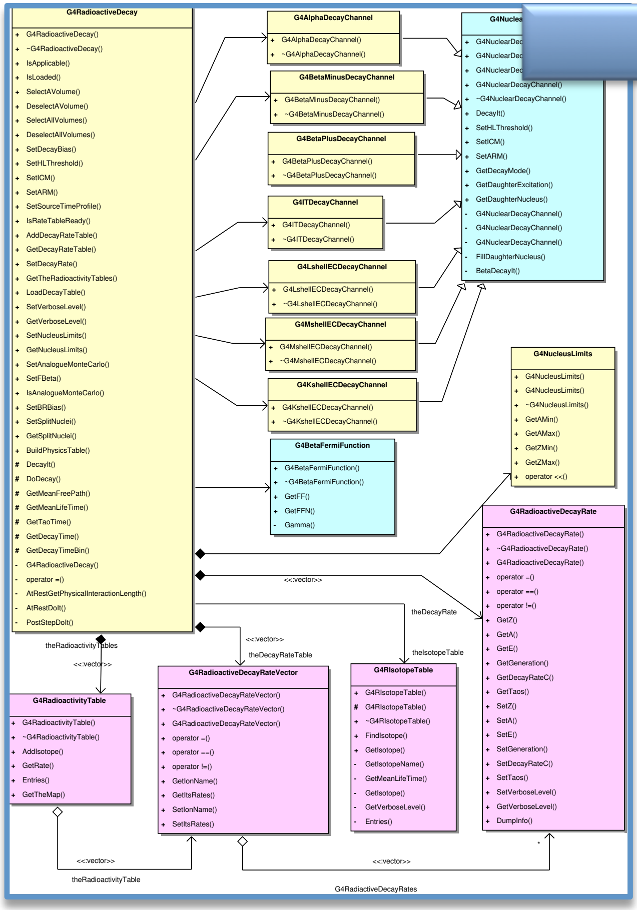
Archival literature

http://www.ge.infn.it/geant4/papers

Refactoring Geant4 Radioactive Decay

Steffen Hauf
PhD Thesis

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



G4RadioactiveDecay
+ G4RadioactiveDecay()
+ ~G4RadioactiveDecay()
+ IsApplicable()
+ IsLoaded()
+ SelectAVolume()
+ DeselectAVolume()
+ SelectAllVolumes()
+ DeselectAllVolumes()
+ SetDecayBias()
+ SetHLThreshold()
+ SetICM()
+ SetARM()
+ SetSourceTimeProfile()
+ IsRateTableReady()
+ AddDecayRateTable()
+ GetDecayRateTable()
+ SetDecayRate()
+ GetTheRadioactivityTables()
+ LoadDecayTable()
+ SetVerboseLevel()
+ GetVerboseLevel()
+ SetNucleusLimits()
+ GetNucleusLimits()
+ SetAnalogueMonteCarlo()
+ SetFBeta()
+ IsAnalogueMonteCarlo()
+ SetBRBias()
+ SetSplitNuclei()
+ GetSplitNuclei()
+ BuildPhysicsTable()
# DecayIt()
# DoDecay()
# GetMeanFreePath()
# GetMeanLifeTime()
# GetTaoTime()
# GetDecayTime()
# GetDecayTimeBin()
- G4RadioactiveDecay()
- operator =()
- AtRestGetPhysicalInteractionLength()
- AtRestDoIt()
- PostStepDoIt()

G4AlphaDecayChannel
+ G4AlphaDecayChannel()
+ ~G4AlphaDecayChannel()

G4BetaMinusDecayChannel
+ G4BetaMinusDecayChannel()
+ ~G4BetaMinusDecayChannel()

G4BetaPlusDecayChannel
+ G4BetaPlusDecayChannel()
+ ~G4BetaPlusDecayChannel()

G4ITDecayChannel
+ G4ITDecayChannel()
+ ~G4ITDecayChannel()

G4LshellECDecayChannel
+ G4LshellECDecayChannel()
+ ~G4LshellECDecayChannel()

G4MshellECDecayChannel
+ G4MshellECDecayChannel()
+ ~G4MshellECDecayChannel()

G4KshellECDecayChannel
+ G4KshellECDecayChannel()
+ ~G4KshellECDecayChannel()

G4BetaFermiFunction
+ G4BetaFermiFunction()
+ ~G4BetaFermiFunction()
+ GetFF()
+ GetFFN()
- Gamma()

G4Nuclea
+ G4NuclearDec
+ G4NuclearDec
+ G4NuclearDec
+ G4NuclearDecayChannel()
+ ~G4NuclearDecayChannel()
+ DecayIt()
+ SetHLThreshold()
+ SetICM()
+ SetARM()
+ GetDecayMode()
+ GetDaughterExcitation()
+ GetDaughterNucleus()
- G4NuclearDecayChannel()
- G4NuclearDecayChannel()
- G4NuclearDecayChannel()
- FillDaughterNucleus()
- BetaDecayIt()

G4NucleusLimits
+ G4NucleusLimits()
+ G4NucleusLimits()
+ ~G4NucleusLimits()
+ GetAMin()
+ GetAMax()
+ GetZMin()
+ GetZMax()
+ operator <<()

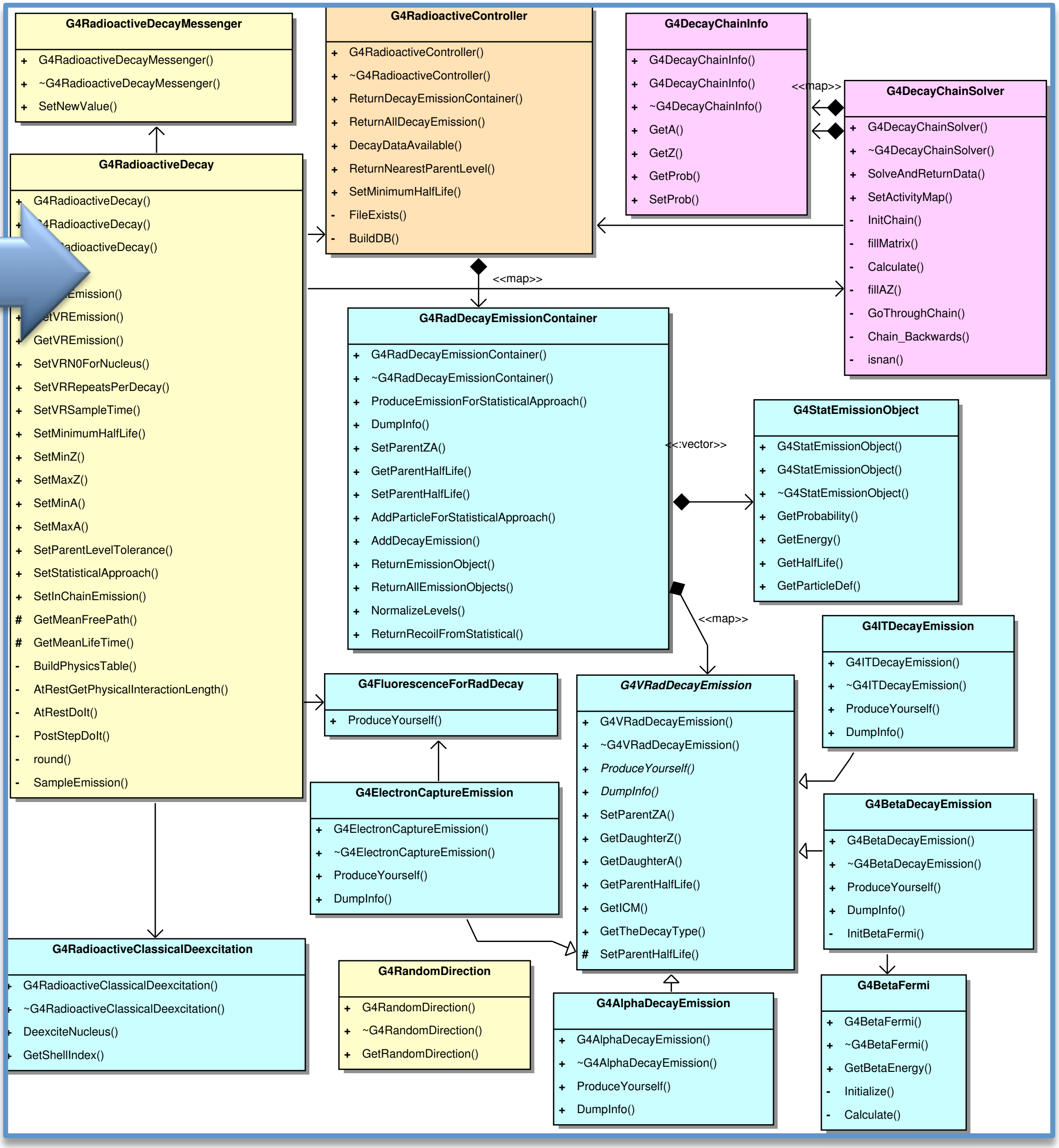
G4RadioactiveDecayRate
+ G4RadioactiveDecayRate()
+ ~G4RadioactiveDecayRate()
+ G4RadioactiveDecayRate()
+ operator =()
+ operator ==()
+ operator !=()
+ GetZ()
+ GetA()
+ GetE()
+ GetGeneration()
+ GetDecayRateC()
+ GetTaos()
+ SetZ()
+ SetA()
+ SetE()
+ SetGeneration()
+ SetDecayRateC()
+ SetTaos()
+ SetVerboseLevel()
+ GetVerboseLevel()
+ DumpInfo()

G4RadioactivityTable
+ G4RadioactivityTable()
+ ~G4RadioactivityTable()
+ AddIsotope()
+ GetRate()
+ Entries()
+ GetTheMap()

G4RadioactiveDecayRateVector
+ G4RadioactiveDecayRateVector()
+ ~G4RadioactiveDecayRateVector()
+ G4RadioactiveDecayRateVector()
+ operator =()
+ operator ==()
+ operator !=()
+ GetIonName()
+ GetItsRates()
+ SetIonName()
+ SetItsRates()

G4RIsotopeTable
+ G4RIsotopeTable()
# G4RIsotopeTable()
+ ~G4RIsotopeTable()
+ FindIsotope()
+ GetIsotope()
- GetIsotopeName()
- GetMeanLifeTime()
- GetIsotope()
- GetVerboseLevel()
- Entries()

Maria Grazia Pia, *INFN Genova*



G4RadioactiveDecayMessenger
+ G4RadioactiveDecayMessenger()
+ ~G4RadioactiveDecayMessenger()
+ SetNewValue()

G4RadioactiveController
+ G4RadioactiveController()
+ ~G4RadioactiveController()
+ ReturnDecayEmissionContainer()
+ ReturnAllDecayEmission()
+ DecayDataAvailable()
+ ReturnNearestParentLevel()
+ SetMinimumHalfLife()
- FileExists()
- BuildDB()

G4DecayChainInfo
+ G4DecayChainInfo()
+ G4DecayChainInfo()
+ ~G4DecayChainInfo()
+ GetA()
+ GetZ()
+ GetProb()
+ SetProb()

G4DecayChainSolver
+ G4DecayChainSolver()
+ ~G4DecayChainSolver()
+ SolveAndReturnData()
+ SetActivityMap()
- InitChain()
- fillMatrix()
- Calculate()
- fillAZ()
- GoThroughChain()
- Chain_Backwards()
- isnan()

G4RadioactiveDecay
+ G4RadioactiveDecay()
+ 4RadioactiveDecay()
adioactiveDecay()
Emission()
+ etVREmission()
+ GetVREmission()
+ SetVRN0ForNucleus()
+ SetVRRepeatsPerDecay()
+ SetVRSampleTime()
+ SetMinimumHalfLife()
+ SetMinZ()
+ SetMaxZ()
+ SetMinA()
+ SetMaxA()
+ SetParentLevelTolerance()
+ SetStatisticalApproach()
+ SetInChainEmission()
# GetMeanFreePath()
# GetMeanLifeTime()
- BuildPhysicsTable()
- AtRestGetPhysicalInteractionLength()
- AtRestDoIt()
- PostStepDoIt()
- round()
- SampleEmission()

G4RadDecayEmissionContainer
+ G4RadDecayEmissionContainer()
+ ~G4RadDecayEmissionContainer()
+ ProduceEmissionForStatisticalApproach()
+ DumpInfo()
+ SetParentZA()
+ GetParentHalfLife()
+ SetParentHalfLife()
+ AddParticleForStatisticalApproach()
+ AddDecayEmission()
+ ReturnEmissionObject()
+ ReturnAllEmissionObjects()
+ NormalizeLevels()
+ ReturnRecoilFromStatistical()

G4StatEmissionObject
+ G4StatEmissionObject()
+ G4StatEmissionObject()
+ ~G4StatEmissionObject()
+ GetProbability()
+ GetEnergy()
+ GetHalfLife()
+ GetParticleDef()

G4FluorescenceForRadDecay
+ ProduceYourself()

G4VRadDecayEmission
+ G4VRadDecayEmission()
+ ~G4VRadDecayEmission()
+ *ProduceYourself()*
+ *DumpInfo()*
+ SetParentZA()
+ GetDaughterZ()
+ GetDaughterA()
+ GetParentHalfLife()
+ GetICM()
+ GetTheDecayType()
# SetParentHalfLife()

G4ITDecayEmission
+ G4ITDecayEmission()
+ ~G4ITDecayEmission()
+ ProduceYourself()
+ DumpInfo()

G4ElectronCaptureEmission
+ G4ElectronCaptureEmission()
+ ~G4ElectronCaptureEmission()
+ ProduceYourself()
+ DumpInfo()

G4BetaDecayEmission
+ G4BetaDecayEmission()
+ ~G4BetaDecayEmission()
+ ProduceYourself()
+ DumpInfo()
- InitBetaFermi()

G4RadioactiveClassicalDeexcitation
+ G4RadioactiveClassicalDeexcitation()
+ ~G4RadioactiveClassicalDeexcitation()
+ DeexciteNucleus()
+ GetShellIndex()

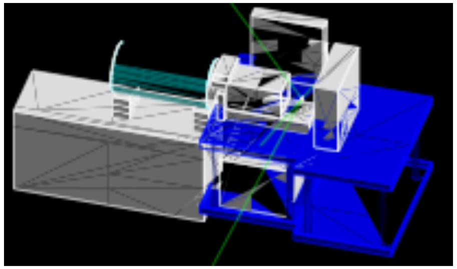
G4RandomDirection
+ G4RandomDirection()
+ ~G4RandomDirection()
+ GetRandomDirection()

G4AlphaDecayEmission
+ G4AlphaDecayEmission()
+ ~G4AlphaDecayEmission()
+ ProduceYourself()
+ DumpInfo()

G4BetaFermi
+ G4BetaFermi()
+ ~G4BetaFermi()
+ GetBetaEnergy()
- Initialize()
- Calculate()

Motivations

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

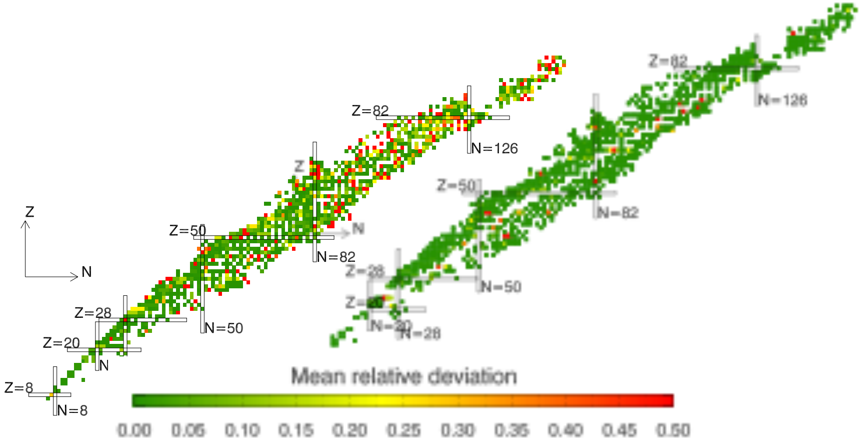


Experiment: Z. W. Bell *(ORNL)*

Enabled by refactoring

New algorithm

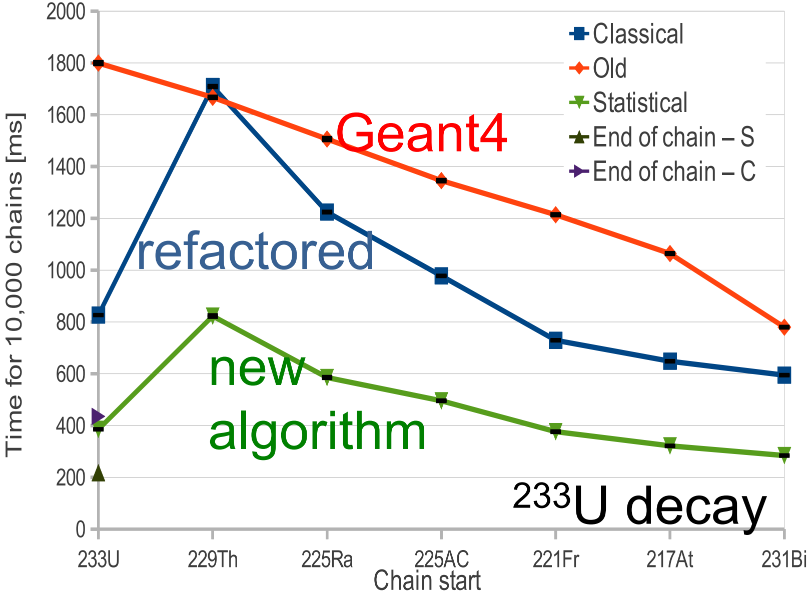
Improved physical accuracy



Maria Grazia Pia, *INFN Genova*

Performance

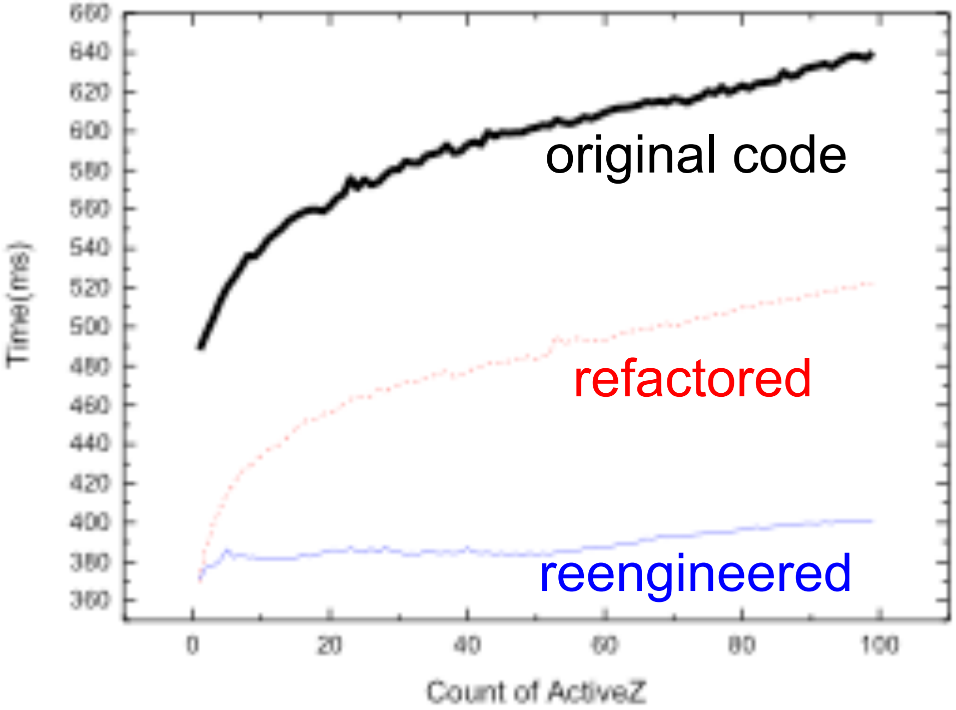
Faster computation



Refactoring Geant4 physics data management

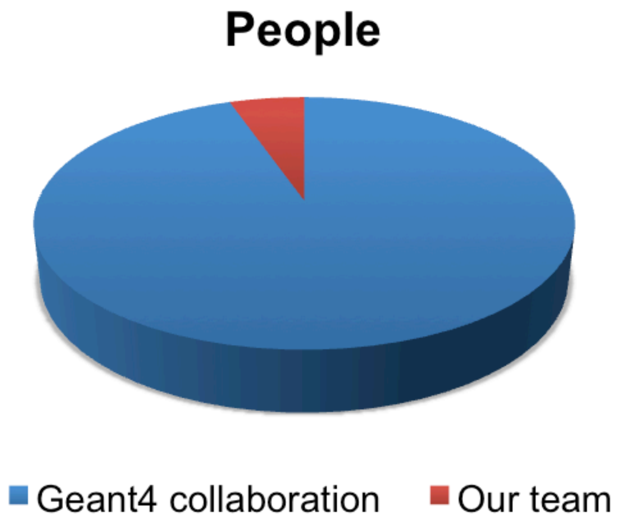
- Today’s technology
  - …keeping an eye on the new C++ Standard

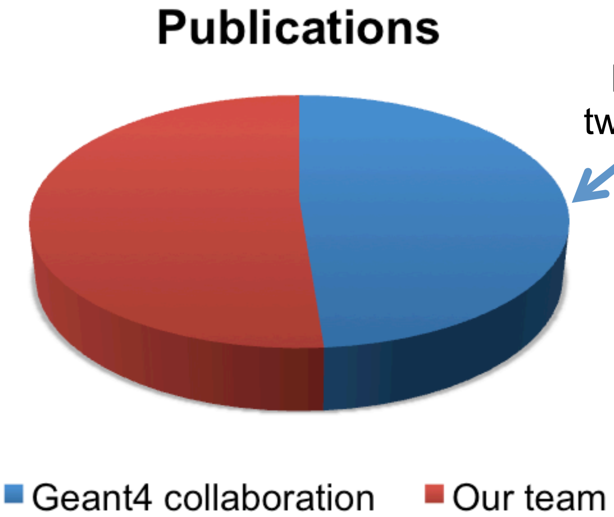
- Optimal container
- Pruning data
- Splitting files
- Software design



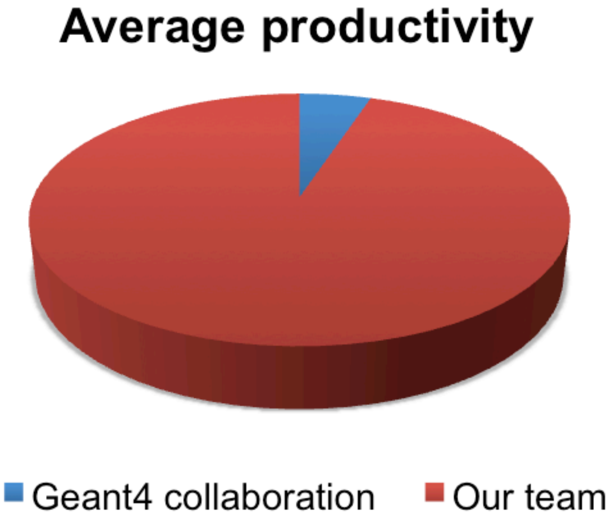
Mincheol Han
Hanyang Univ., Seoul, Korea
Undergraduate student project

Producing results





Including
two flawed
papers



2003-2012

Geant4 core subjects
(no applications)
2 Geant4 general papers excluded

Sources:
http://geant4.web.cern.ch/geant4/results/publications.shtml
http://www.ge.infn.it/geant4/papers/

Outline

Software technology

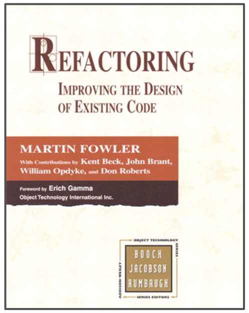
- Problems
- Methods
- Techniques

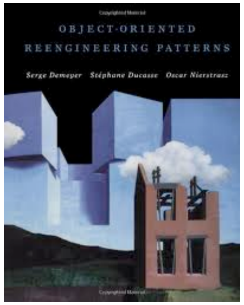
to deal with evolving/legacy software

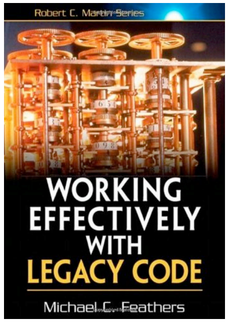
Overview

Focus on basic **concepts**

Guidance for further personal study


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH JACOBSON RUMBAUGH


OBJECT-ORIENTED REENGINEERING PATTERNS
Serge Demeyer Stéphane Ducasse Oscar Nierstrasz



no time to enter into details in this lecture

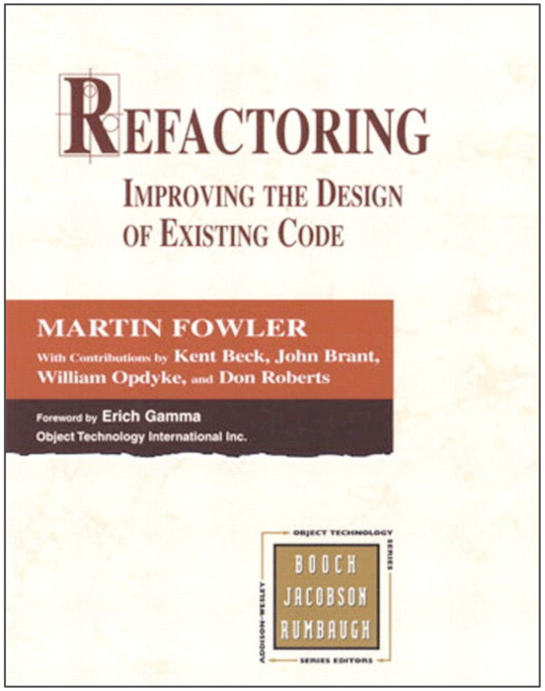
Peculiarities of refactoring physics software

Opportunities for practice & mentoring after the school

Common problems of legacy codes

- 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

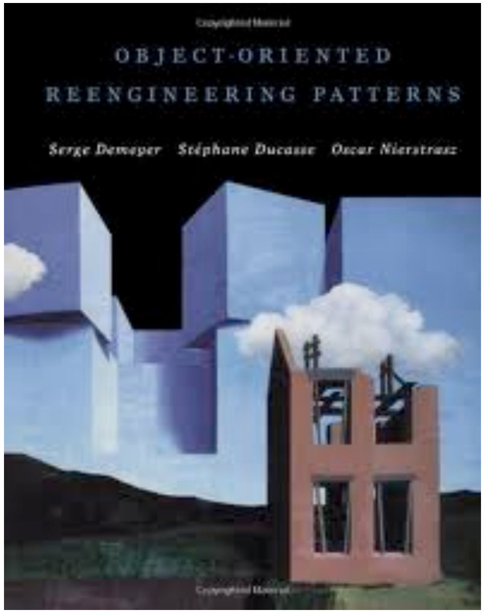


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

Refactoring

"**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

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

Interplay with other processes: **testing**

> 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

Identifying code to be refactored

Code Smells

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

#1 in the stink parade

- **Duplicated Code**
  - 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*
  - or 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 form the code…

Motivation: understanding other people's code

Testing

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

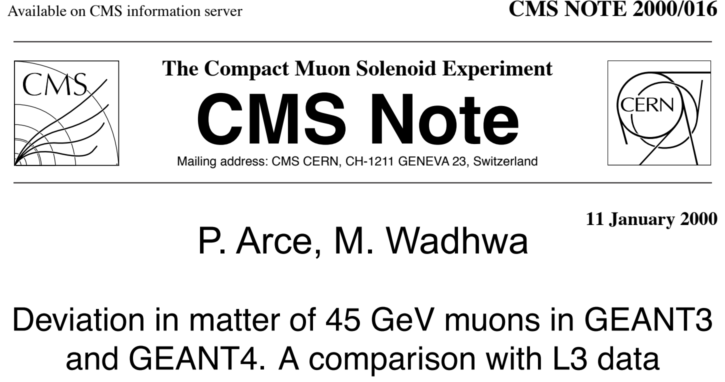
Refactoring occurs as a **series of small changes**

Test original code

Apply one action at a time

Test

…



Available on CMS information server

CMS NOTE 2000/016

The Compact Muon Solenoid Experiment

CMS Note

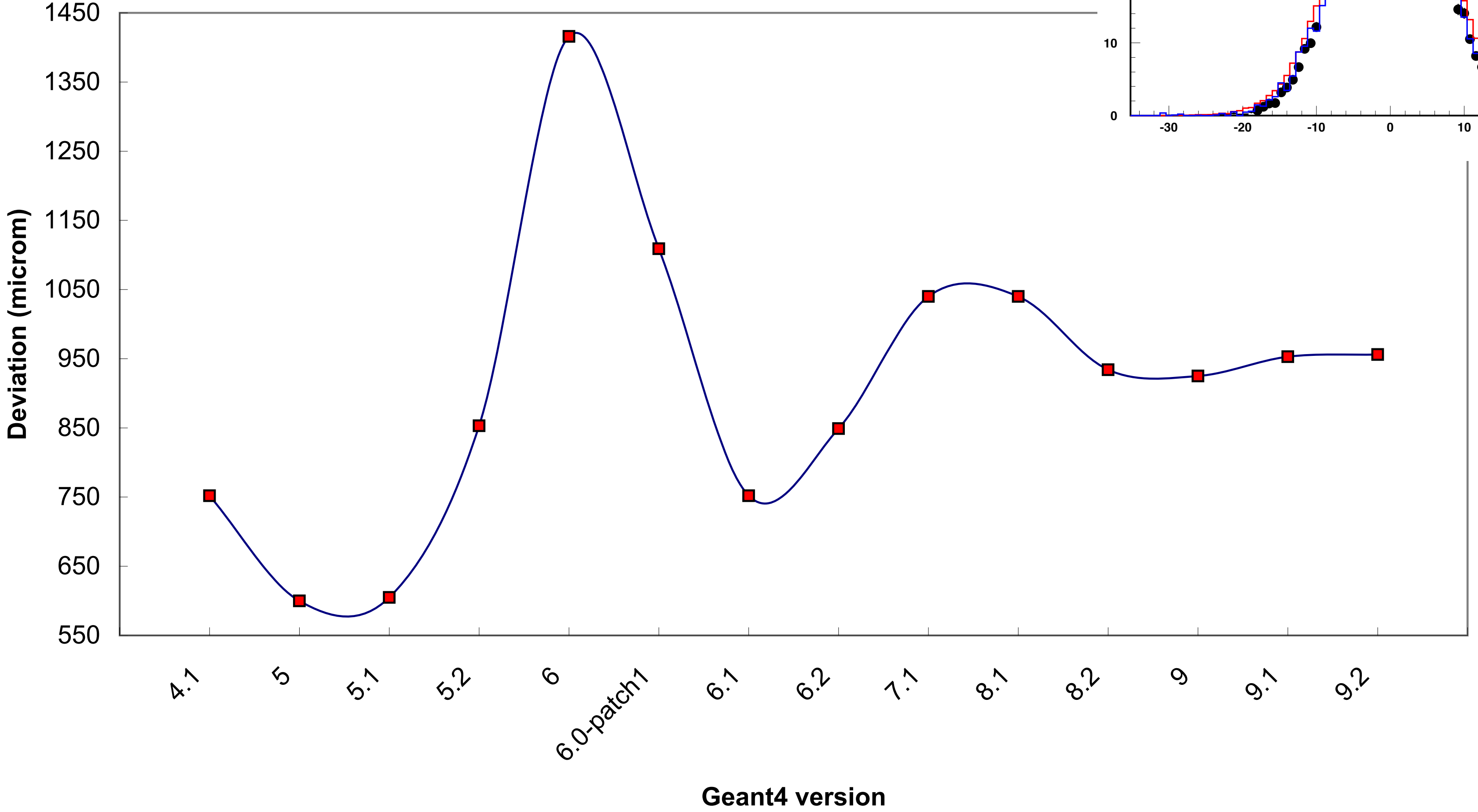
Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

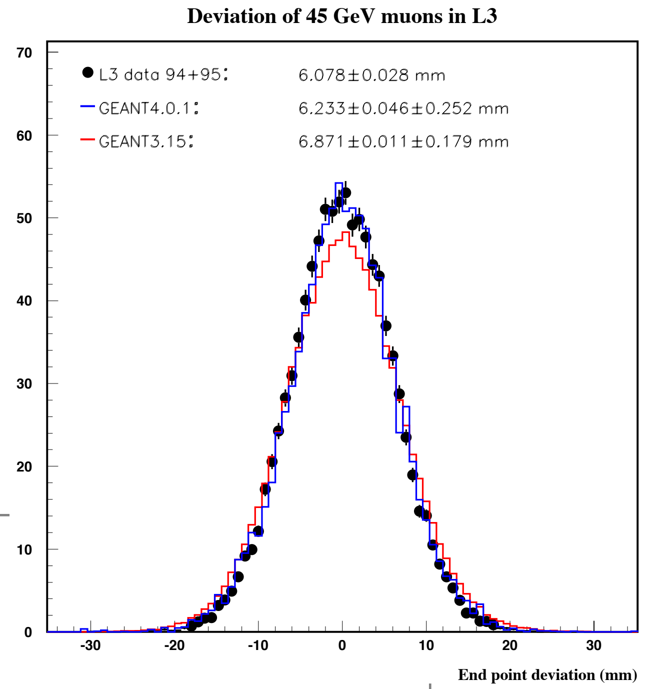
11 January 2000

P. Arce, M. Wadhwa

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

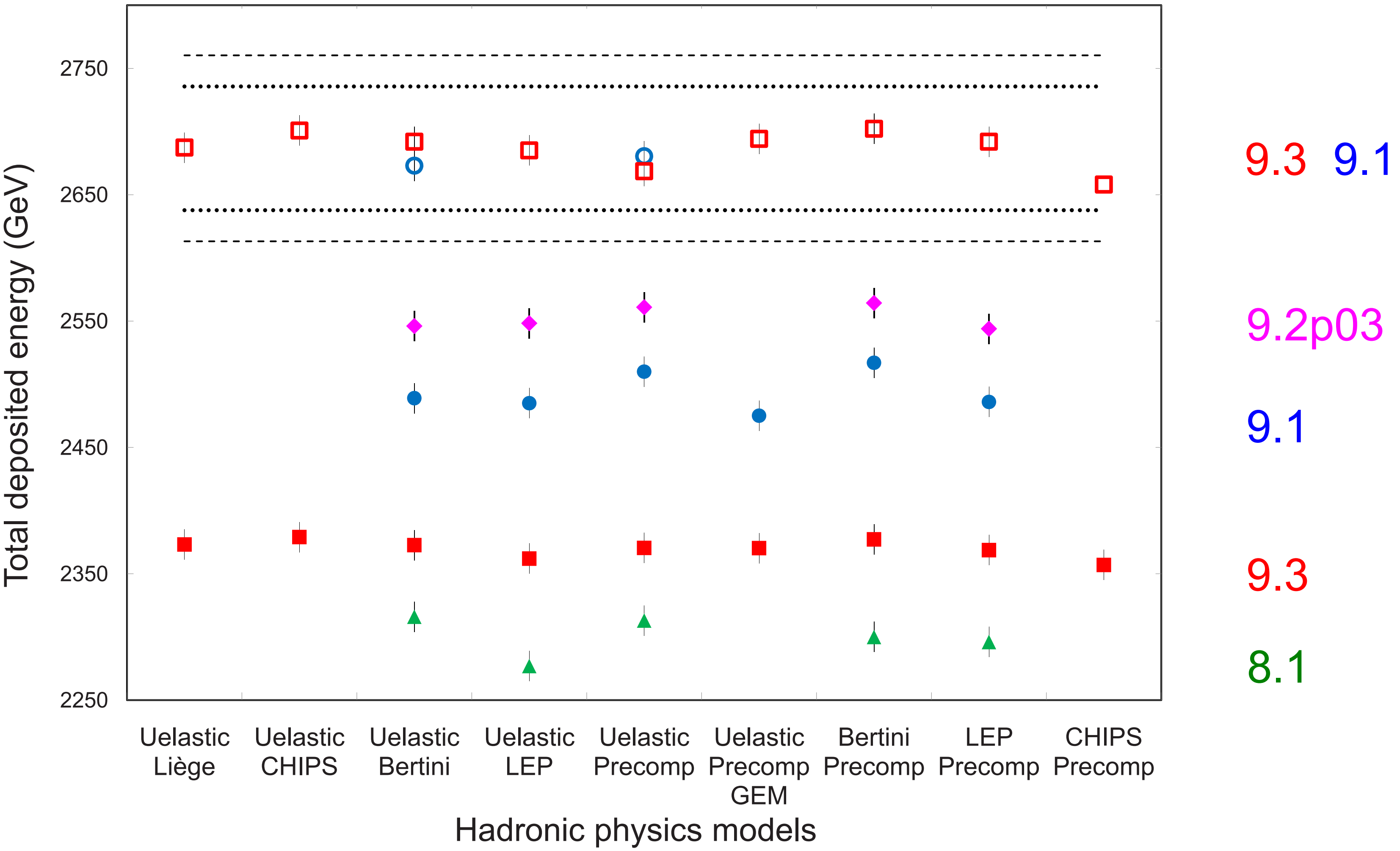
100 GeV mu+, 1 m Fe, lateral deviation at end-point



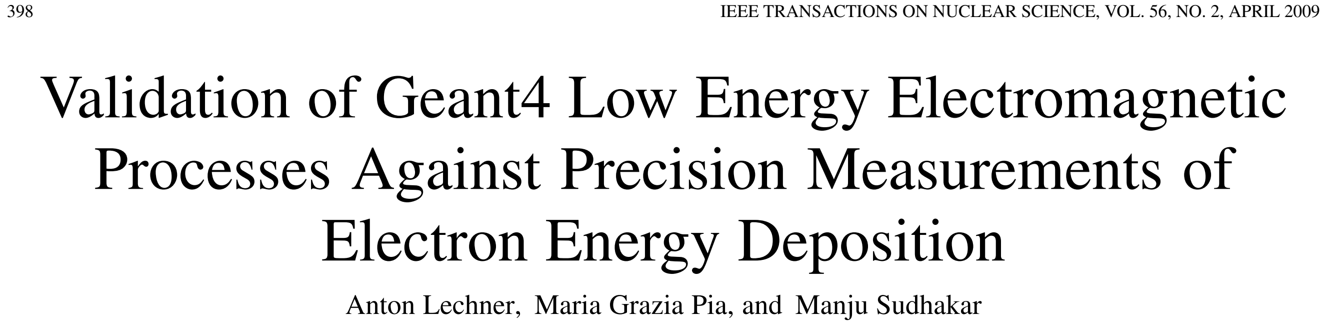


Physics-Related Epistemic Uncertainties in Proton Depth Dose Simulation

Maria Grazia Pia, Marcia Begalli, Anton Lechner, Lina Quintieri, and Paolo Saracco



IEEE NSS Best Student Paper, 2007

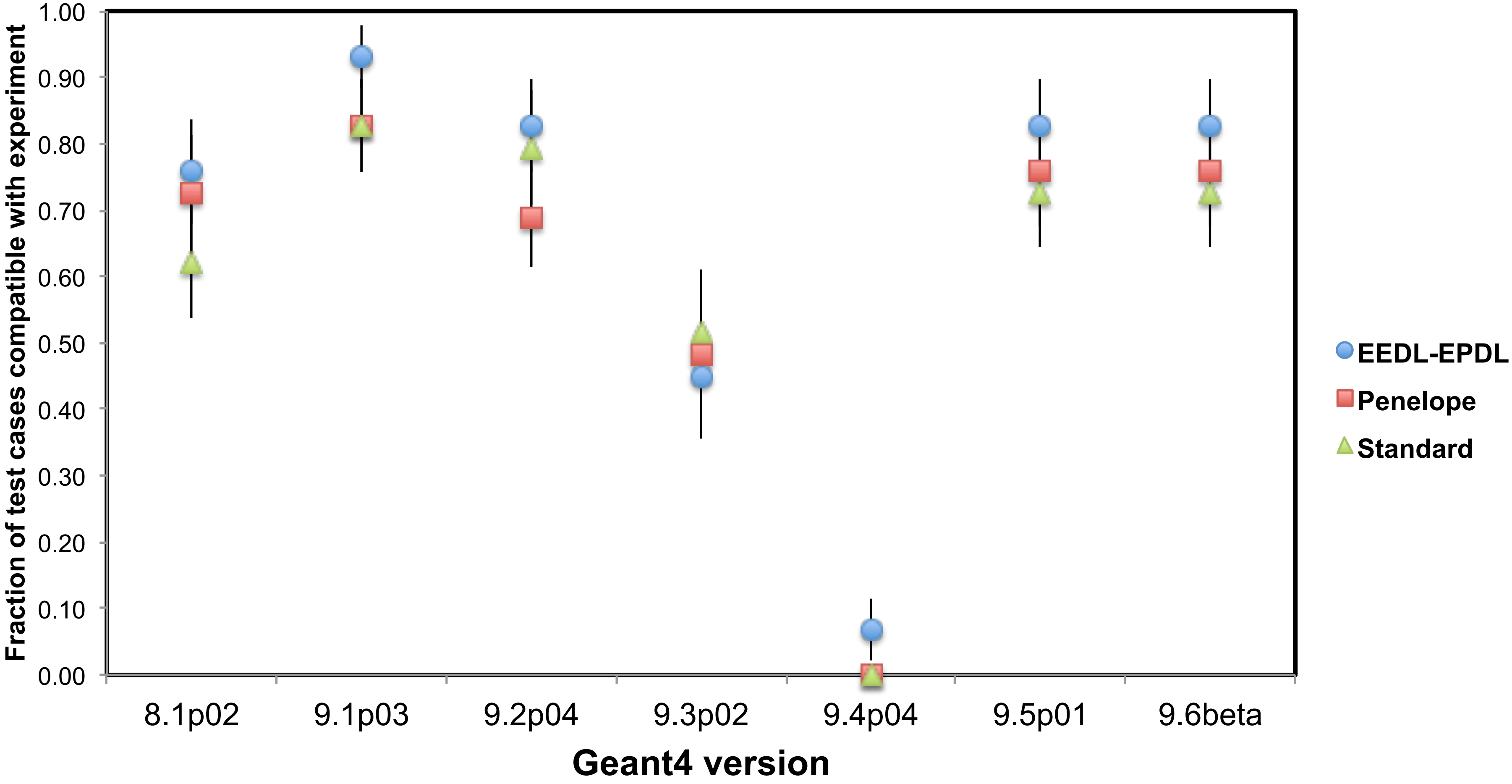


398 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 56, NO. 2, APRIL 2009

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

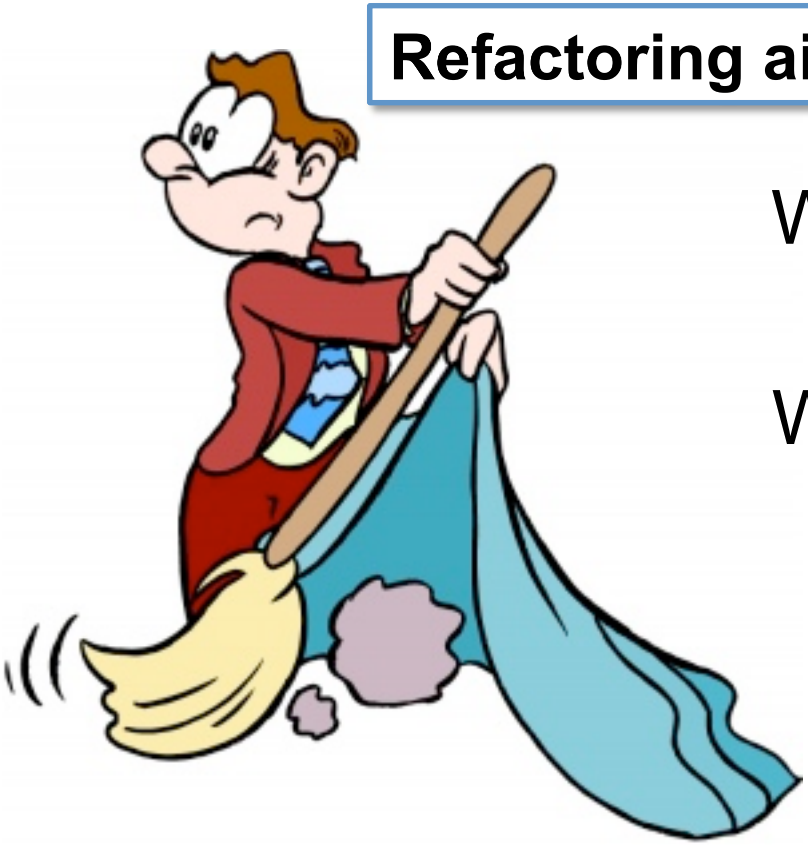
Anton Lechner, Maria Grazia Pia, and Manju Sudhakar

Total energy deposition validation (Sandia 80) - preliminary 17/9/2012



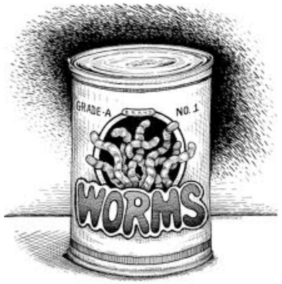
Sweeping under the carpet?

Refactoring aims to preserve correctness



Was the original code verified?

Was the original code **validated**?



IEEE Standard 1012
Software Verification & Validation
ISO 12207

What was the test coverage?

Were the test process and the test results documented?

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**
- …

Basic actions for common smells

- **Method invocation**
  - Consolidate recurring code into a single method
  - OK when the recurring code doesn’t 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

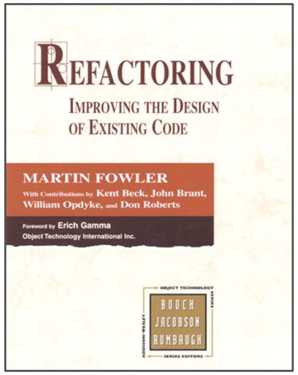
Composing Methods — Extract Method, Inline Method,…

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



REFACTORING
IMPROVING THE DESIGN
OF EXISTING CODE

MARTIN FOWLER
With Contributions by Kent Beck, John Brant,
William Opdyke, and Don Roberts

Foreword by Erich Gamma
Object Technology International Inc.

OBJECT TECHNOLOGY SERIES
BOOCH
JACOBSON
RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

Duplicated code #1 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

How to find duplicated code?

- Automated tools
  - Some exist
- By hand
  - Still the most common way
  - Not necessarily the most efficient (the most inefficient)
- **Reverse engineering**
  - Gain understanding of the code

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

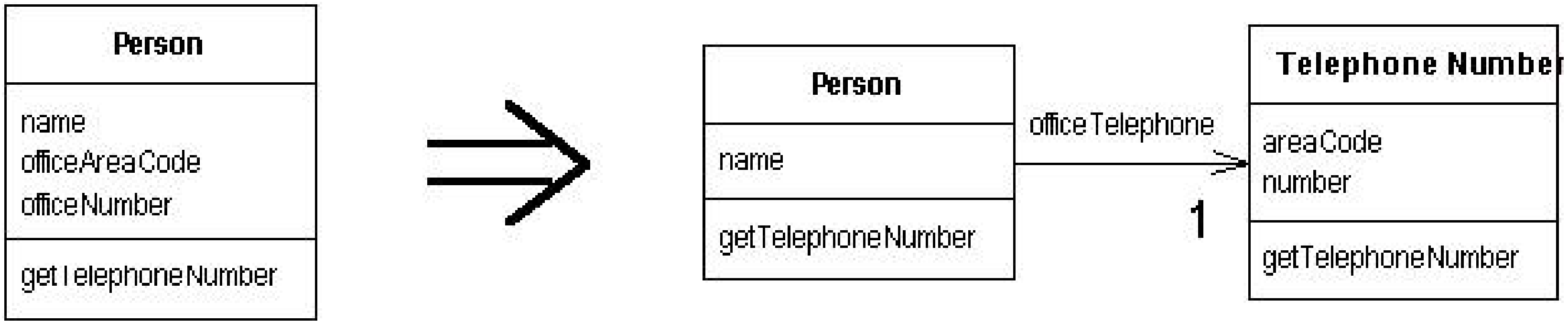
- A class that tries to do too much
- Often has too many instance variables
- Prone to duplicated code

- **Extract Class**
- **Extract SubClass**
- **Extract Interface**

Extract class

One class does work that should be done by two

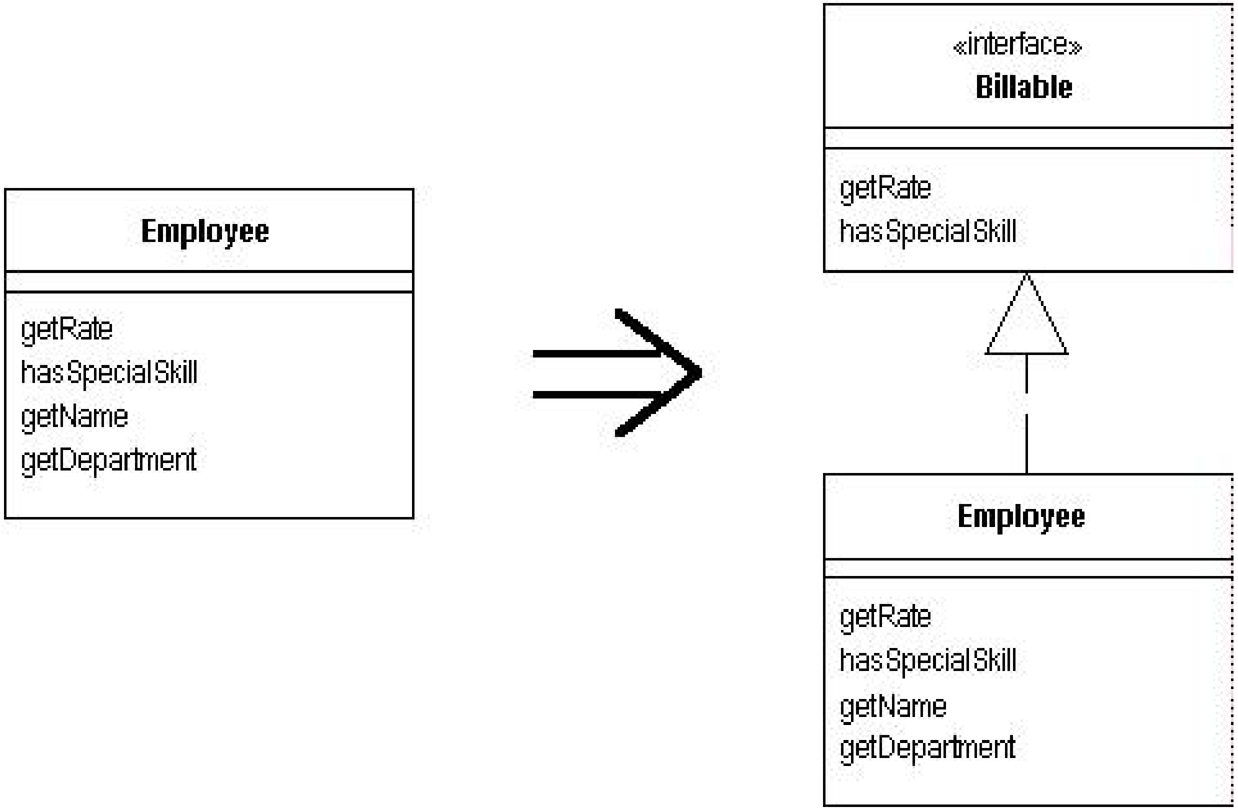
Create a new class and move the relevant fields and methods from the old class into the new class



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

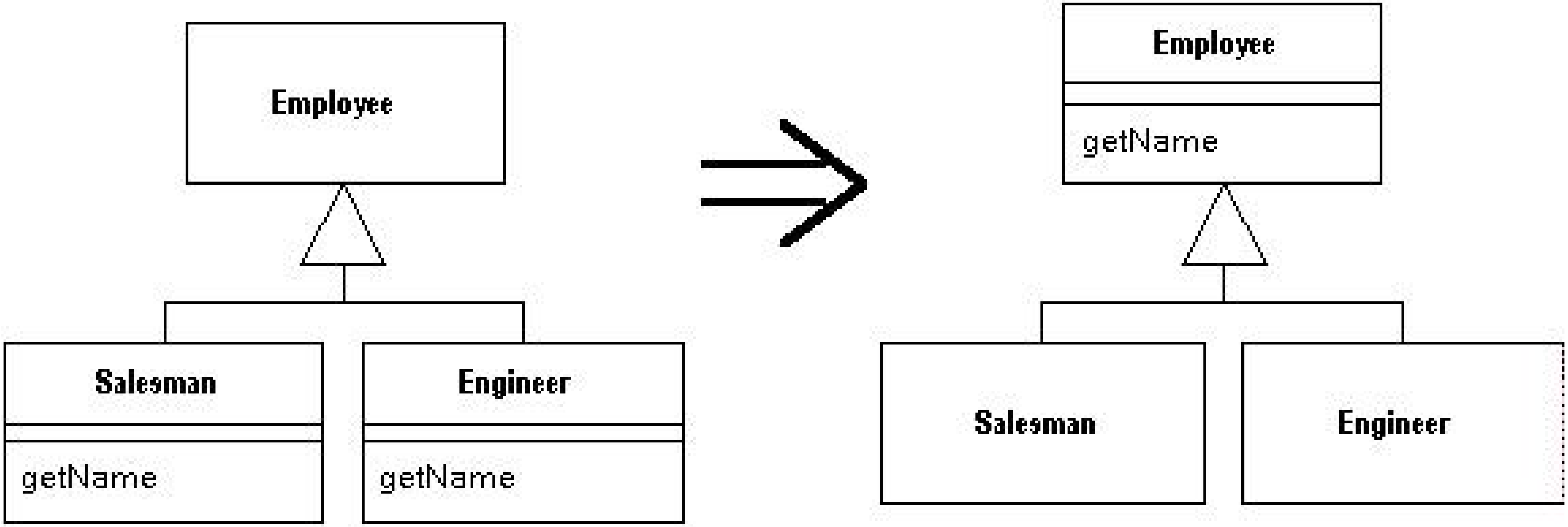


Maria Grazia Pia,

Pull up method

Methods with identical results on subclasses

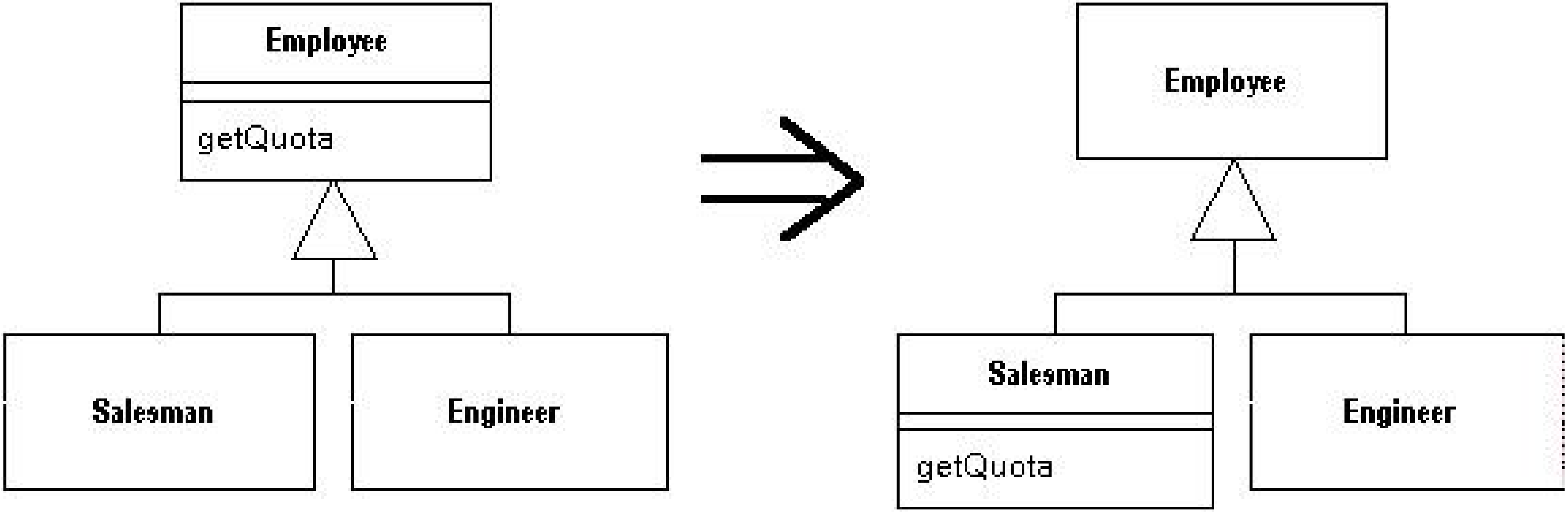
Move them to the superclass



Push down method

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

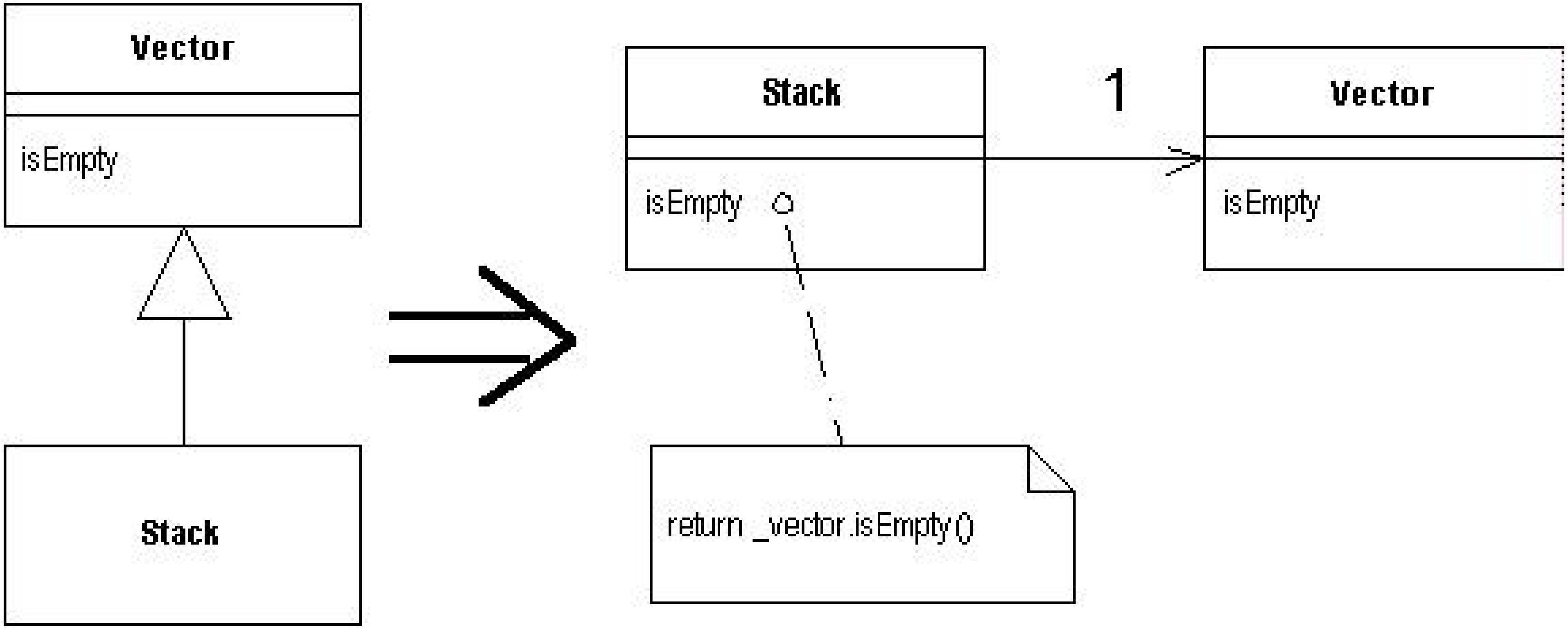
Move it to those subclasses



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** 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

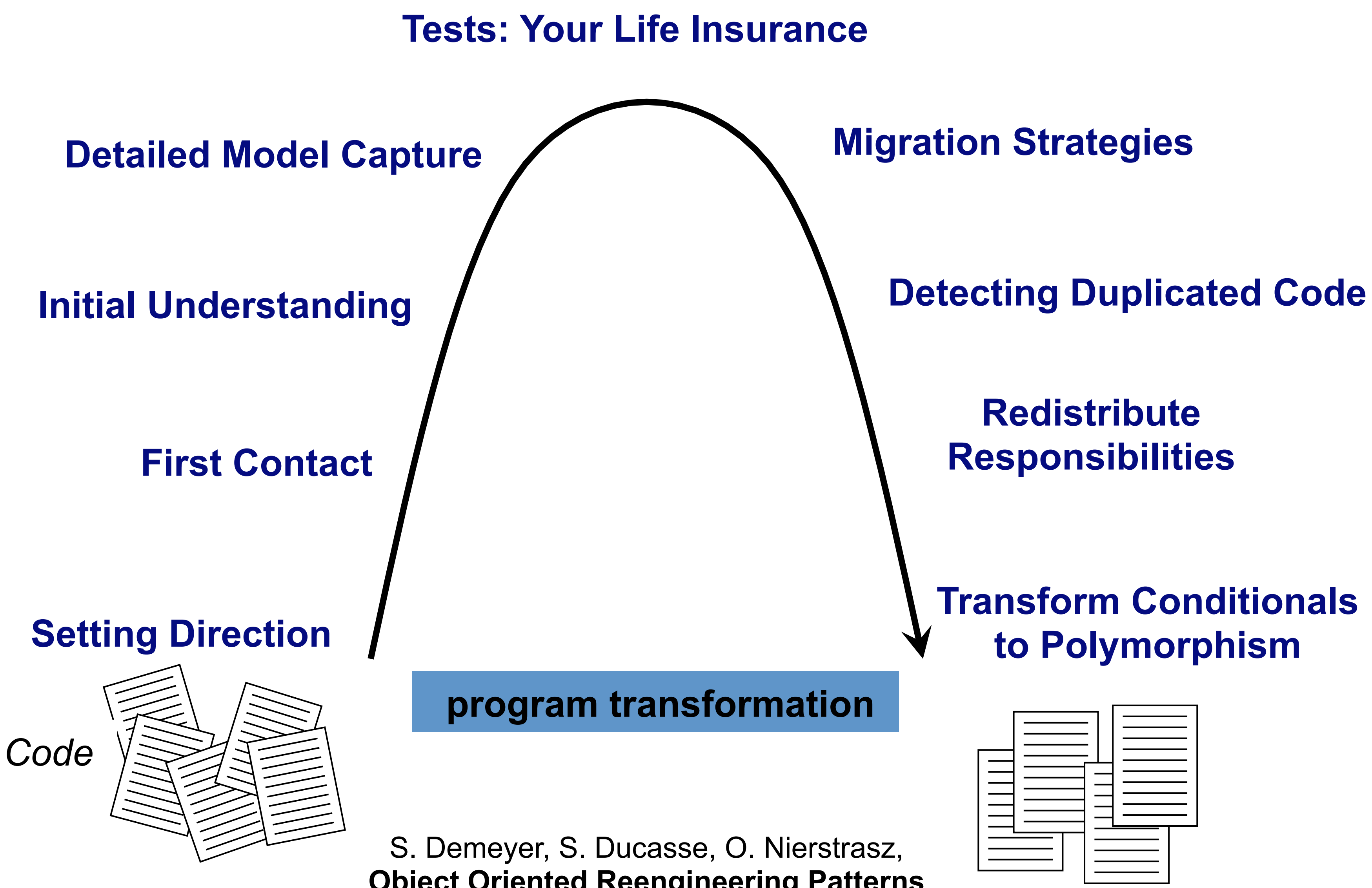
Dealing with Generalization

- **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

A Map of Reengineering Patterns



Tests: Your Life Insurance

Detailed Model Capture

Initial Understanding

First Contact

Setting Direction

Migration Strategies

Detecting Duplicated Code

Redistribute Responsibilities

Transform Conditionals to Polymorphism

program transformation

Code

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

Computational performance

Refactoring may make the code slower

conventional wisdom

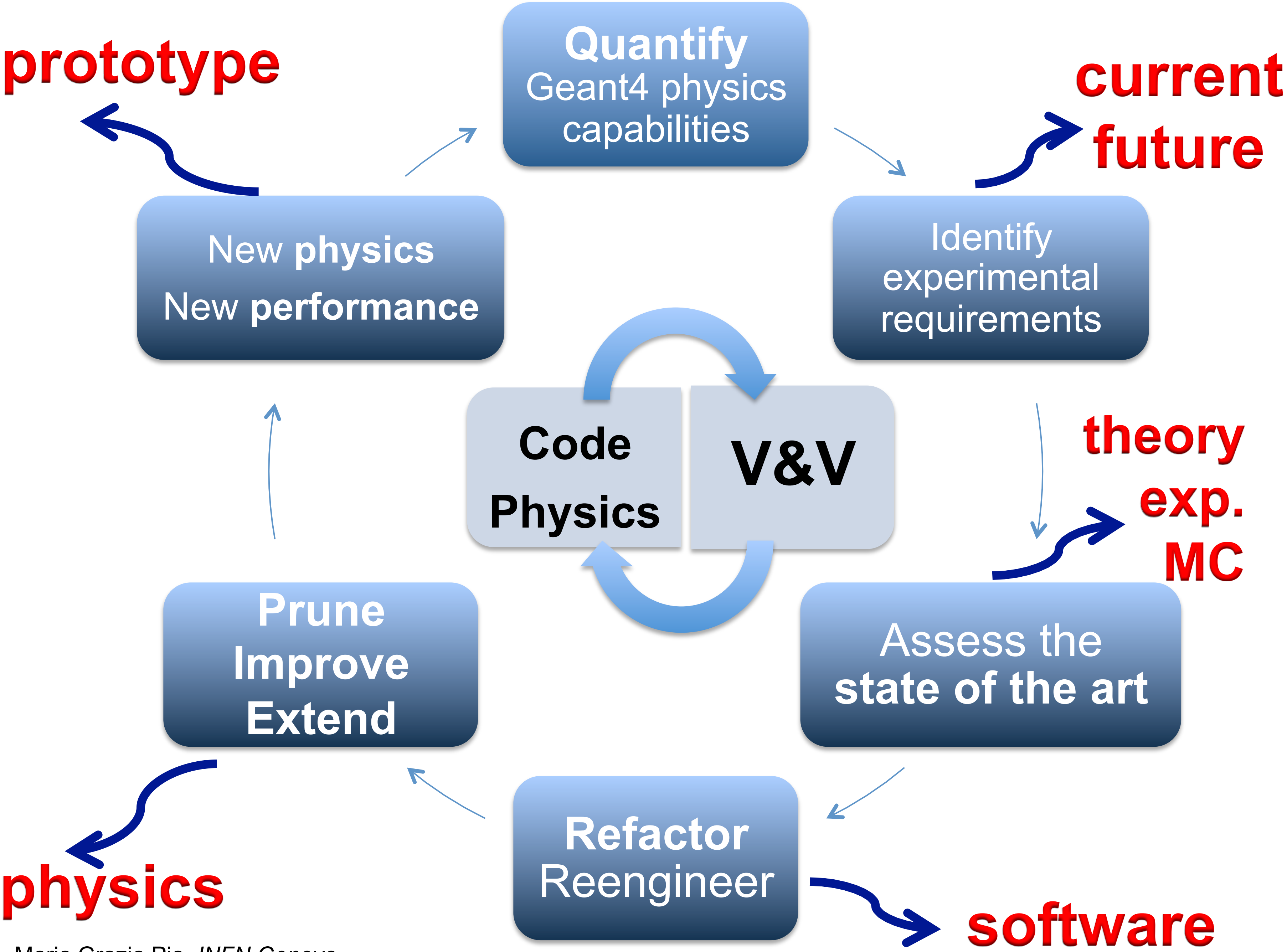
Yes, sometimes…

"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…



Quantify
Geant4 physics capabilities

current
future

Identify experimental requirements

theory
exp.
MC

Assess the **state of the art**

Refactor
Reengineer

software

Prune
Improve
Extend

physics

New **physics**
New **performance**

prototype

Code
Physics

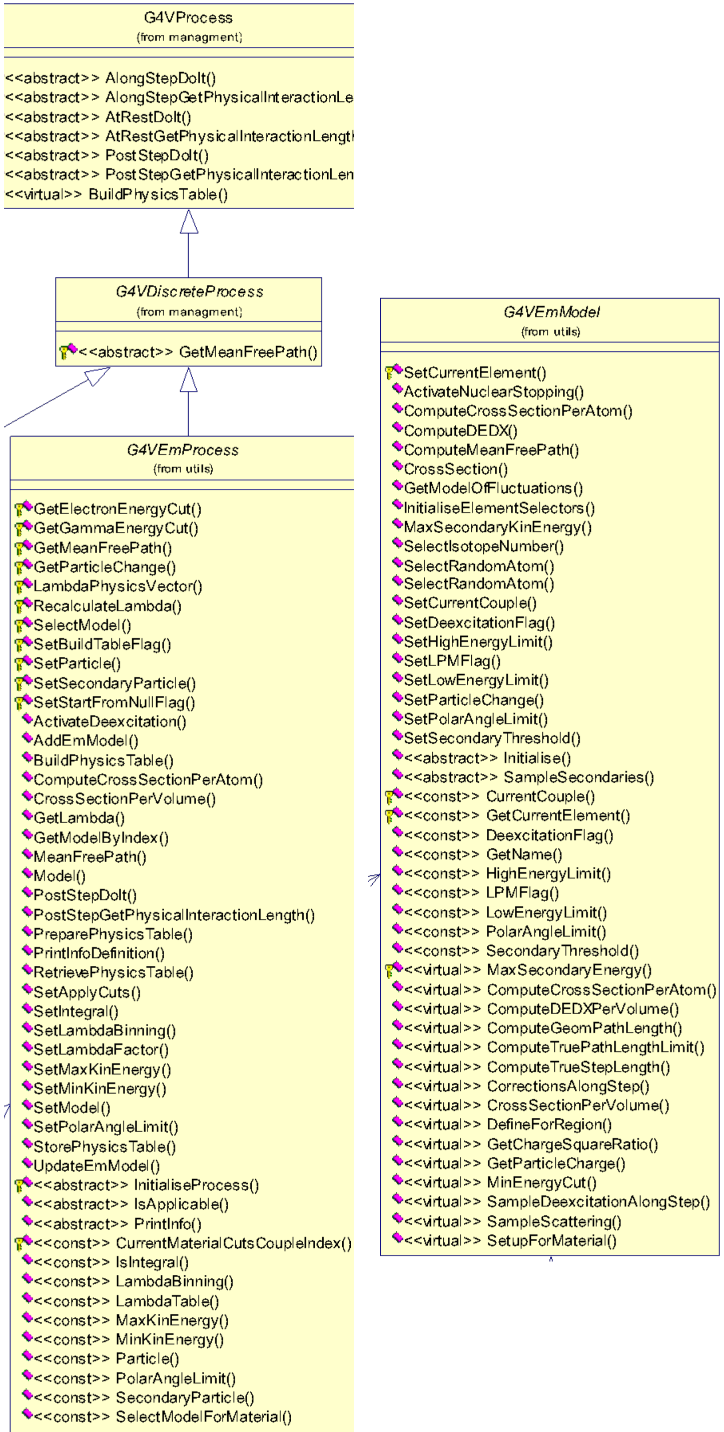
V&V

Maria Grazia Pia, *INFN Genova*

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
…

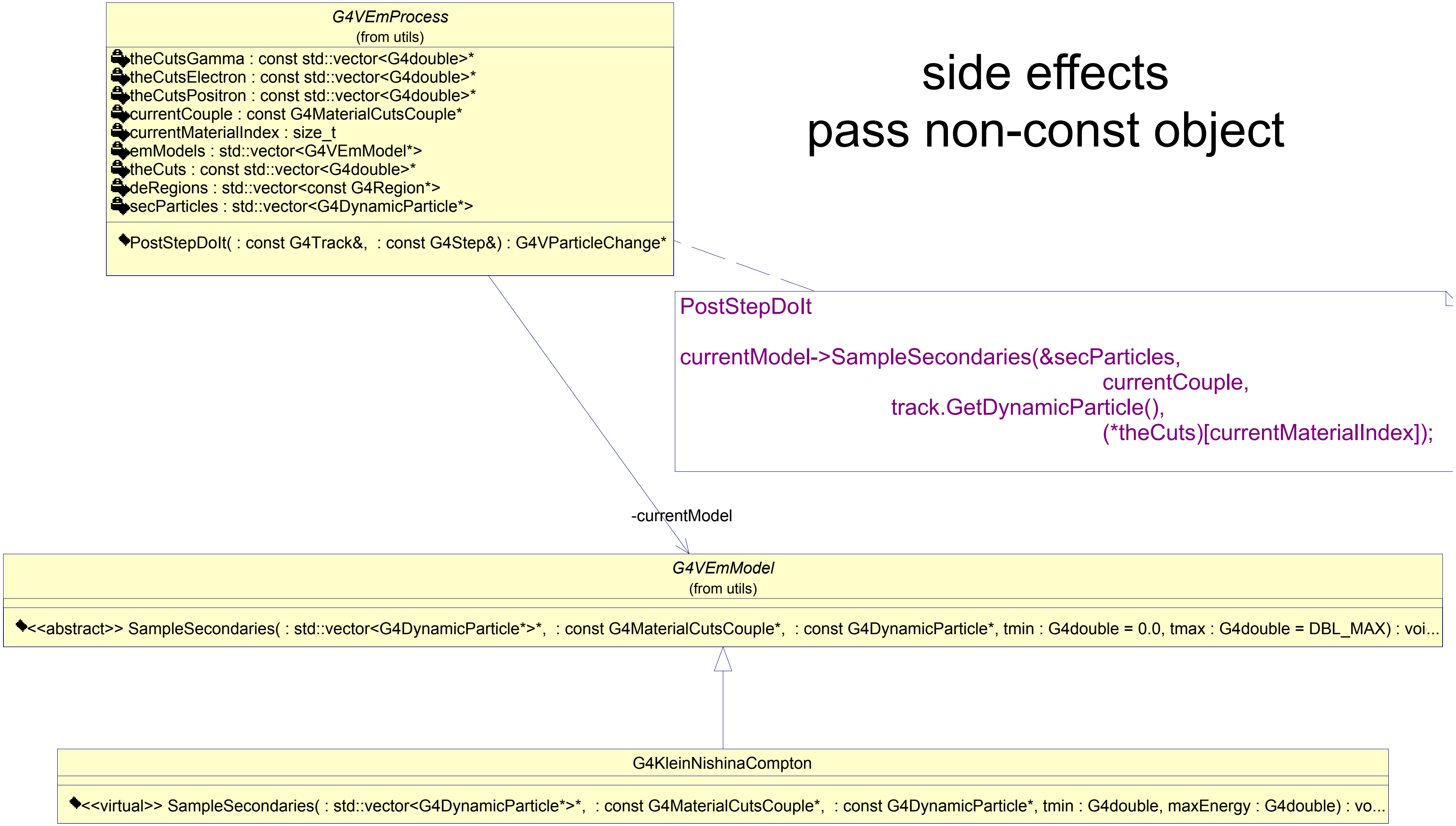
Evolution away from RD44 discipline


G4VProcess
(from managment)
<<abstract>> AlongStepDoIt()
<<abstract>> AlongStepGetPhysicalInteractionLe
<<abstract>> AtRestDoIt()
<<abstract>> AtRestGetPhysicalInteractionLengt
<<abstract>> PostStepDoIt()
<<abstract>> PostStepGetPhysicalInteractionLen
<<virtual>> BuildPhysicsTable()
G4VDiscreteProcess
(from managment)
<<abstract>> GetMeanFreePath()
G4VEmProcess
(from utils)
GetElectronEnergyCut()
GetGammaEnergyCut()
GetMeanFreePath()
GetParticleChange()
LambdaPhysicsVector()
RecalculateLambda()
SelectModel()
SetBuildTableFlag()
SetParticle()
SetSecondaryParticle()
SetStartFromNullFlag()
ActivateDeexcitation()
AddEmModel()
BuildPhysicsTable()
ComputeCrossSectionPerAtom()
CrossSectionPerVolume()
GetLambda()
GetModelByIndex()
MeanFreePath()
Model()
PostStepDoIt()
PostStepGetPhysicalInteractionLength()
PreparePhysicsTable()
PrintInfoDefinition()
RetrievePhysicsTable()
SetApplyCuts()
SetIntegral()
SetLambdaBinning()
SetLambdaFactor()
SetMaxKinEnergy()
SetMinKinEnergy()
SetModel()
SetPolarAngleLimit()
StorePhysicsTable()
UpdateEmModel()
<<abstract>> InitialiseProcess()
<<abstract>> IsApplicable()
<<abstract>> PrintInfo()
<<const>> CurrentMaterialCutsCoupleIndex()
<<const>> IsIntegral()
<<const>> LambdaBinning()
<<const>> LambdaTable()
<<const>> MaxKinEnergy()
<<const>> MinKinEnergy()
<<const>> Particle()
<<const>> PolarAngleLimit()
<<const>> SecondaryParticle()
<<const>> SelectModelForMaterial()
G4VEmModel
(from utils)
SetCurrentElement()
ActivateNuclearStopping()
ComputeCrossSectionPerAtom()
ComputeDEDX()
ComputeMeanFreePath()
CrossSection()
GetModelOfFluctuations()
InitialiseElementSelectors()
MaxSecondaryKinEnergy()
SelectIsotopeNumber()
SelectRandomAtom()
SelectRandomAtom()
SetCurrentCouple()
SetDeexcitationFlag()
SetHighEnergyLimit()
SetLPMFlag()
SetLowEnergyLimit()
SetParticleChange()
SetPolarAngleLimit()
SetSecondaryThreshold()
<<abstract>> Initialise()
<<abstract>> SampleSecondaries()
<<const>> CurrentCouple()
<<const>> GetCurrentElement()
<<const>> DeexcitationFlag()
<<const>> GetName()
<<const>> HighEnergyLimit()
<<const>> LPMFlag()
<<const>> LowEnergyLimit()
<<const>> PolarAngleLimit()
<<const>> SecondaryThreshold()
<<virtual>> MaxSecondaryEnergy()
<<virtual>> ComputeCrossSectionPerAtom()
<<virtual>> ComputeDEDXPerVolume()
<<virtual>> ComputeGeomPathLength()
<<virtual>> ComputeTruePathLengthLimit()
<<virtual>> ComputeTrueStepLength()
<<virtual>> CorrectionsAlongStep()
<<virtual>> CrossSectionPerVolume()
<<virtual>> DefineForRegion()
<<virtual>> GetChargeSquareRatio()
<<virtual>> GetParticleCharge()
<<virtual>> MinEnergyCut()
<<virtual>> SampleDeexcitationAlongStep()
<<virtual>> SampleScattering()
<<virtual>> SetupForMaterial()

Maria Grazia Pia, *INFN Genova*

Smells

side effects
pass non-const object



G4VEmProcess
(from utils)

- theCutsGamma : const std::vector<G4double>*
- theCutsElectron : const std::vector<G4double>*
- theCutsPositron : const std::vector<G4double>*
- currentCouple : const G4MaterialCutsCouple*
- currentMaterialIndex : size_t
- emModels : std::vector<G4VEmModel*>
- theCuts : const std::vector<G4double>*
- deRegions : std::vector<const G4Region*>
- secParticles : std::vector<G4DynamicParticle*>

PostStepDoIt(: const G4Track&, : const G4Step&) : G4VParticleChange*

```
PostStepDoIt

currentModel->SampleSecondaries(&secParticles,
                                currentCouple,
                track.GetDynamicParticle(),
                                (*theCuts)[currentMaterialIndex]);
```

G4VEmModel
(from utils)

<<abstract>> SampleSecondaries(: std::vector<G4DynamicParticle*>*, : const G4MaterialCutsCouple*, : const G4DynamicParticle*, tmin : G4double = 0.0, tmax : G4double = DBL_MAX) : voi...

G4KleinNishinaCompton

<<virtual>> SampleSecondaries(: std::vector<G4DynamicParticle*>*, : const G4MaterialCutsCouple*, : const G4DynamicParticle*, tmin : G4double, maxEnergy : G4double) : vo...

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;
```

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

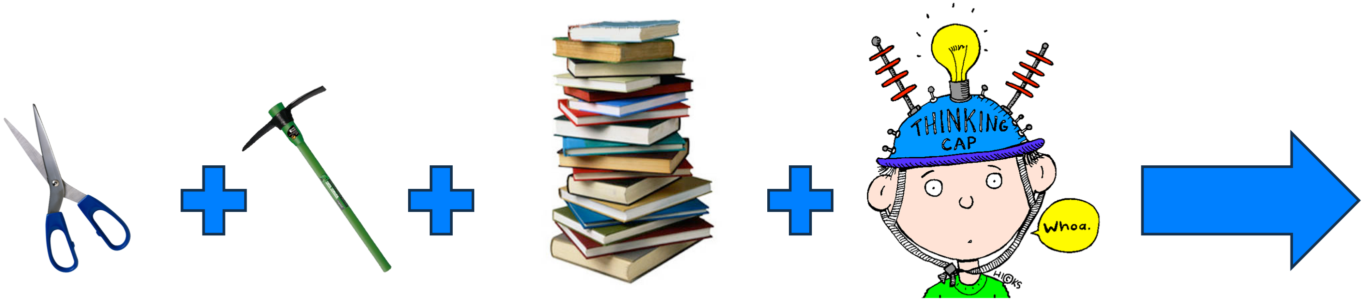
Dependencies

on other parts of the software

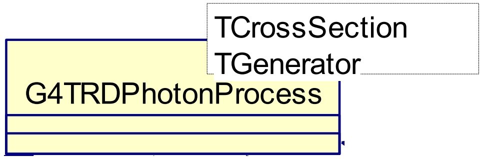
One needs a geometry
(and a full scale application)
to test (verify) a cross section

Difficult to test → no testing
often

Problem domain analysis
Improve domain decomposition



Reengineered code


TCrossSection
TGenerator
G4TRDPhotonProcess

Benefits

Transparency

Ease of **maintenance**

Simplicity of **testing** for V&V

> *Numera ciò che è numerabile, misura ciò che è misurabile,*
> *e ciò che non è misurabile rendilo misurabile.*
>
> Galileo Galilei (1564-1642)

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

Exploring new models (calculations) is made easier

Quantification of accuracy is facilitated

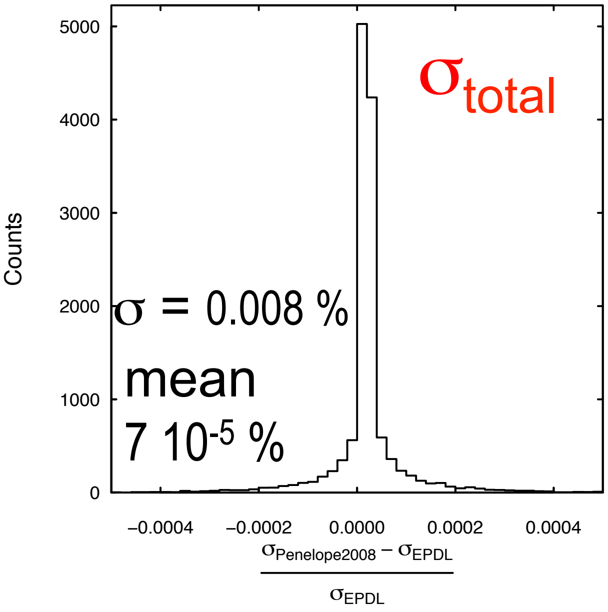
Prune

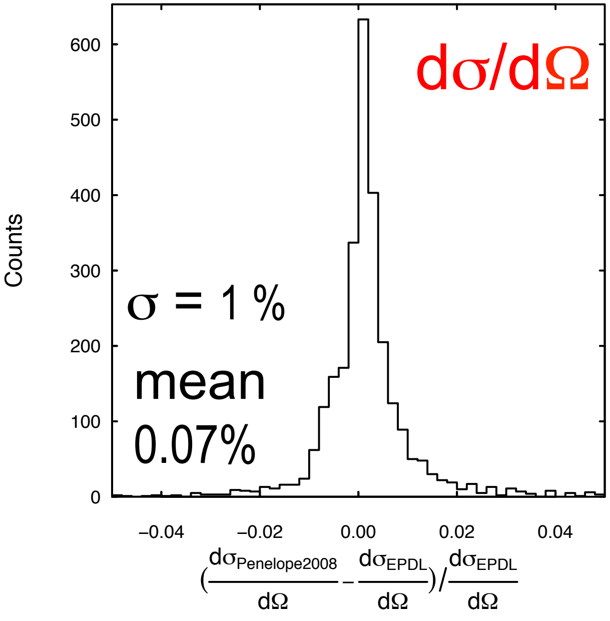
Number one in the stink parade is duplicated ~~code~~ physics

M. Fowler, *Refactoring*

Objective quantification of smell

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





Efficiency w.r.t. experiment

“Livermore”	Penelope
EPDL97	EPDL97
0.38±0.06	0.38±0.06

Code bloat

Burden on

- Software design
- Maintenance
- User support

Unnecessary complexity

Bremsstrahlung, evaporation, proton elastic scattering etc.

Trash

and redo

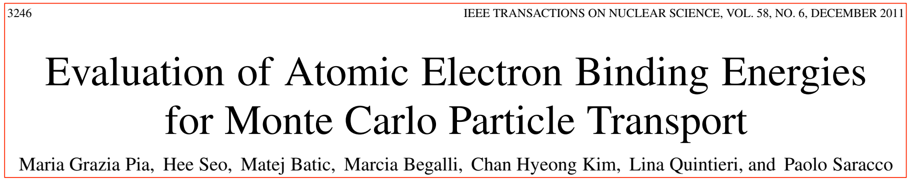
Number one in the stink parade is duplicated ~~code~~ numbers

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

Atomic binding energies

Geant 4 $\begin{cases}\text{Carlson + Williams}\\ \text{EADL}\end{cases}$ $\left(\begin{matrix}\textit{Carlson}\\ \textit{Shirley}\end{matrix}\right)$

Source of epistemic uncertainties?

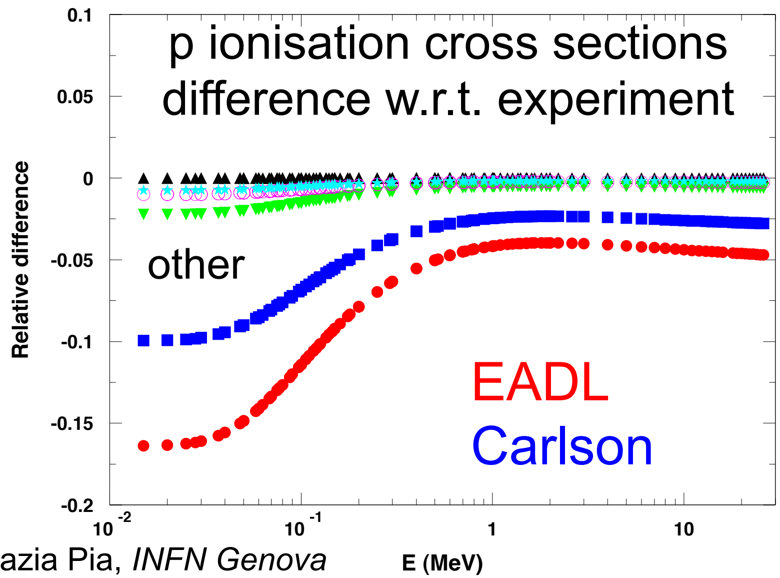


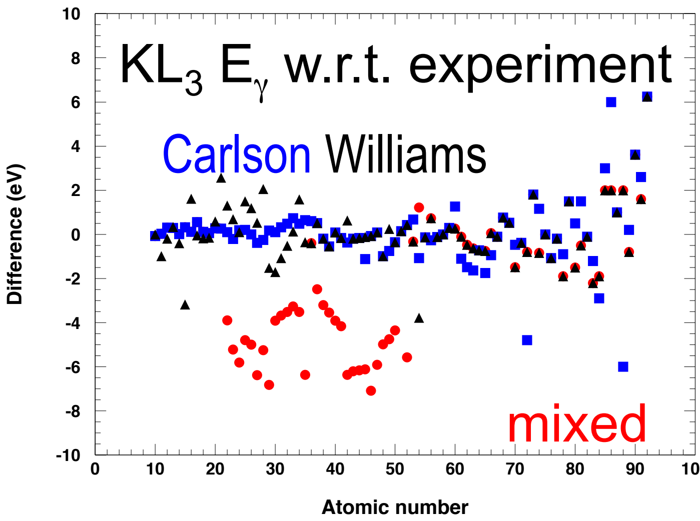
3246 IEEE TRANSACTIONS ON NUCLEAR SCIENCE, VOL. 58, NO. 6, DECEMBER 2011

Evaluation of Atomic Electron Binding Energies for Monte Carlo Particle Transport

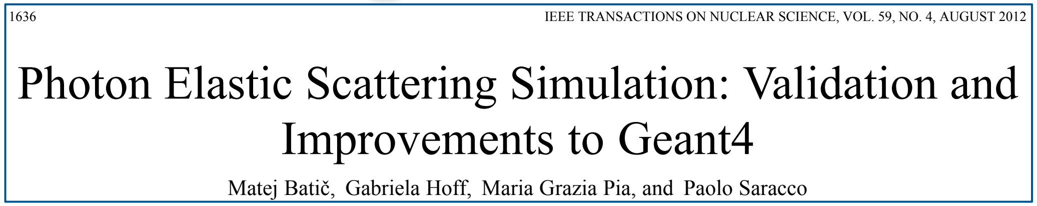
Maria Grazia Pia, Hee Seo, Matej Batic, Marcia Begalli, Chan Hyeong Kim, Lina Quintieri, and Paolo Saracco

23 pages





An example of reengineering:
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

current Geant4 **Differential cross sections** new

	Penelope	Penelope	EPDL	Relativ.	Non-Rel.	Modified	MFF	RFF	SM
	2001	2008		FF	FF	FF	ASF	ASF	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 compatible with experiment, 0.01 significance

Computational performance

Popular belief

Physics model X is intrinsically slow

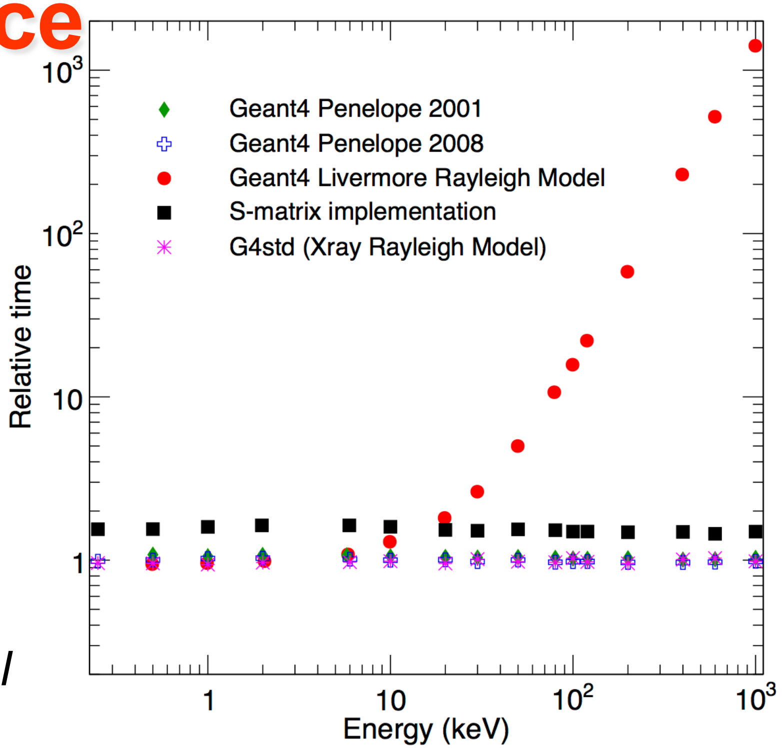
Baroque methods to combine it with “faster” lower precision models and limit its use to cases where one is willing to pay for higher precision

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

Truth

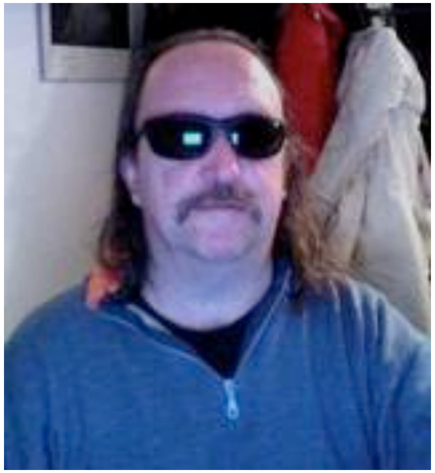
Physics model X is intrinsically fast

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!



The legacy code dilemma

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

When we change code, we should have tests in place

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

Lack of tests distinguishes legacy code from non-legacy code

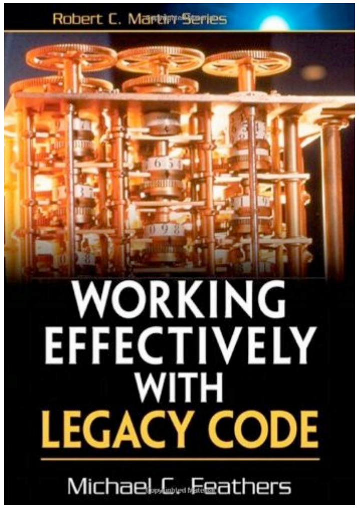
Most of the fear involved in making changes to large software systems is **fear of introducing bugs**

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

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

Legacy management 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



Test Covering

A set of tests used to introduce an invariant on a code base

- Usually cover a set of classes
- Provide some “invariant” that let us know when we have changed the behaviour of our system
  - get that invariant before refactoring or adding new behavior
- Correctness is defined by original behaviour of the code base

Identify change point

can’t get this class in a test harness

Inflection Point

A narrow interface to a set of classes

If anyone changes any of the classes behind an inflection point, the change is either detectable at the inflection point, or inconsequential in the application

Cover an inflection point

Write tests for it

Hard point:
make it compile in a test harness

Usually requires breaking dependencies

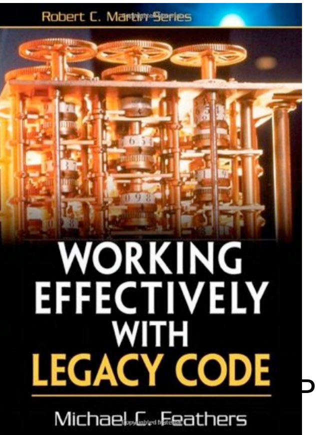
External dependencies

objects which we have to provide to setup the object we are creating *(e.g. in a constructor)*

Internal dependencies

the class we want to cover creates its own objects internally

Write tests
Make changes
Refactor



Techniques for breaking dependencies

I can’t get this class in a test harness

- Objects of the class can’t be created easily
- The test harness won’t easily build with the class in it
- The constructor we need to use has bad side effects
- Significant work happens in the constructor, and we need to sense it

“Tricks” to make the class testable

Irritating parameter

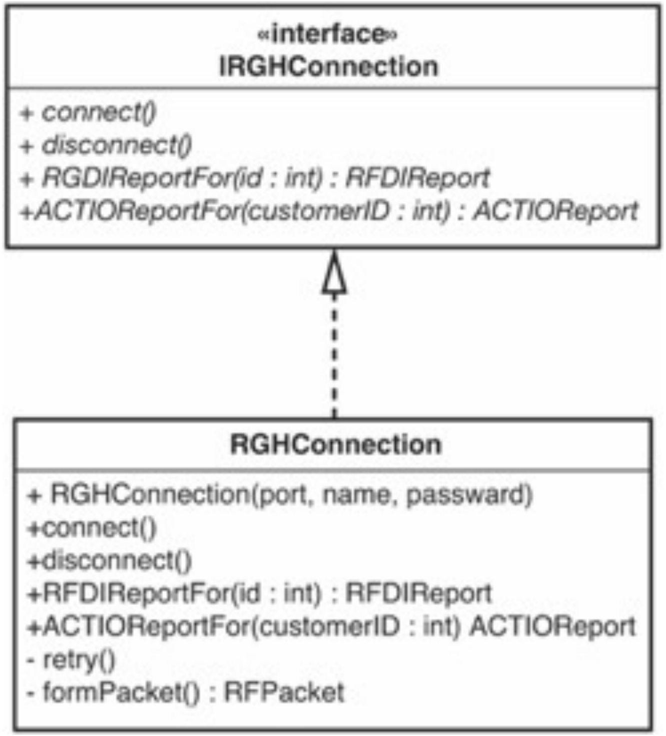
How am I going to construct these parameters for the test?

```
public class CreditValidator {
public CreditValidator(RGHConnection connection,
                       CreditMaster master,
                       String validatorID) {
 ...
}
Certificate validateCustomer(Customer customer)
     throws InvalidCredit {
   ...
}
... }
```

Setting up network connection is not possible

Solution to irritating parameter

Extract interface + create FakeConnection class


«interface»
IRGHConnection
+ connect()
+ disconnect()
+ RGDIReportFor(id : int) : RFDIReport
+ACTIOReportFor(customerID : int) : ACTIOReport
RGHConnection
+ RGHConnection(port, name, passward)
+connect()
+disconnect()
+RFDIReportFor(id : int) : RFDIReport
+ACTIOReportFor(customerID : int) ACTIOReport
- retry()
- formPacket() : RFPacket

```
public class FakeConnection
        implements IRGHConnection {
  public RFDIReport report;
  public void connect() {}
  public void disconnect() {}
  ...
}
```

Solutions for irritating parameter

- **Pass null**
  - If an object requires a parameter that is hard to construct
  - If the parameter is used during your test execution an exception will be thrown
  - You can then still reconsider to construct a real object

- Variant solution: **“null object”**
  - A sibling to the original class with no real functionality
  - Returns default values

Hidden dependency

```
mailing_list_dispatcher::mailing_list_dispatcher()
  : service(new mail_service), status(MAIL_OKAY)
{
const int client_type = 12;
service->connect();
status = MAIL_OFFLINE;
…}
```

The constructor relies on the class mail_service

We don't want to initialize the mail_service, because then we connect to the network and start sending actual mails...

Solution to hidden dependency

Parameterize constructor

```
mailing_list_dispatcher::mailing_list_dispatcher
  (mail_service* service) : status(MAIL_OKAY)
```

Big improvement

Allows for introducing a fake mail service

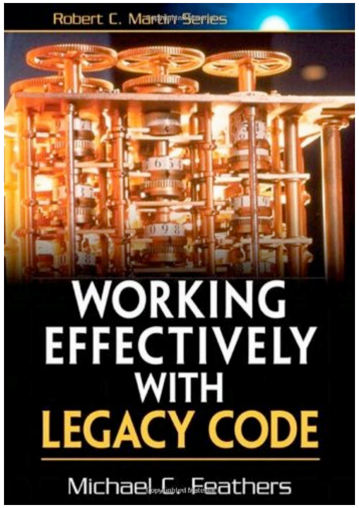
- Extract interface for mail_service
- Introduce fake class that senses the things we do

Hidden method

- How do we write a **test for a private method**?

Two obvious solutions:

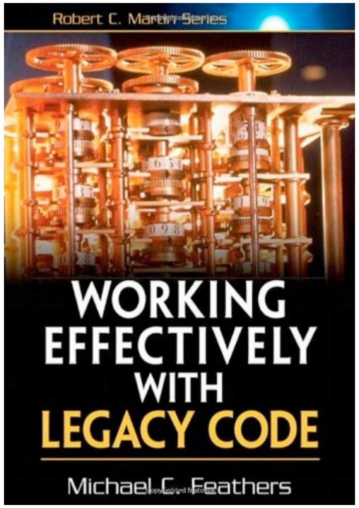
- Make the method **public**
- Change the private method to **protected** and then **subclass** it



… and more
No time to go into details

Tip on software development

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



Provocative thought…

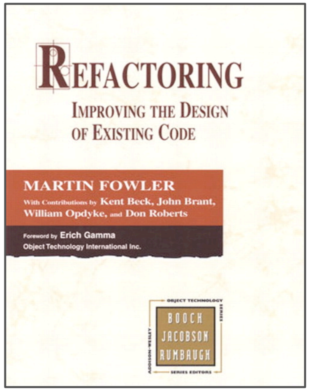
Need to refactor legacy code due to:

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

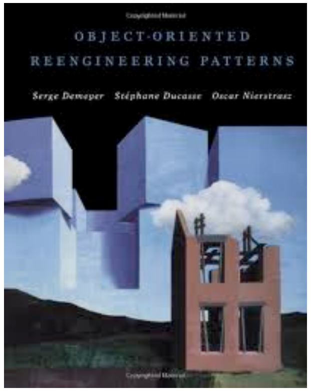
Need to refactor legacy code due to:

- 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
- …

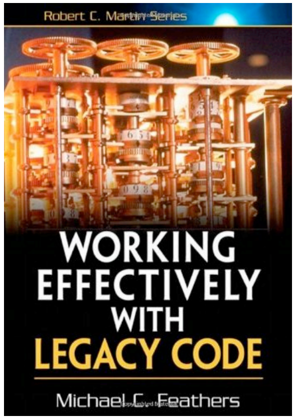
Books and other resources


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH JACOBSON RUMBAUGH

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

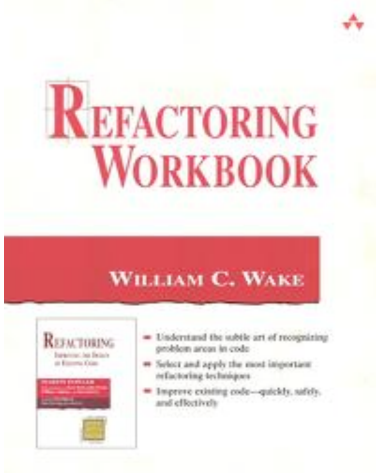


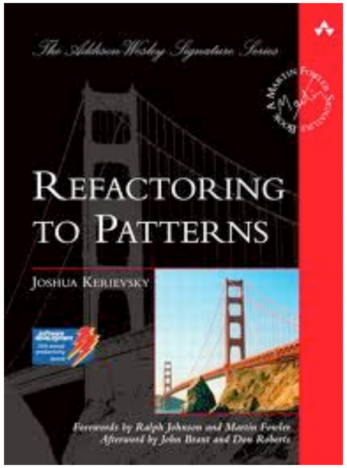
Serge Demeyer, Stéphane Ducasse, Oscar Nierstrasz
Object Oriented Reengineering Patterns
Morgan-Kaufmann, 2003
Revised 2008: http://www.iam.unibe.ch/~scg/OORP

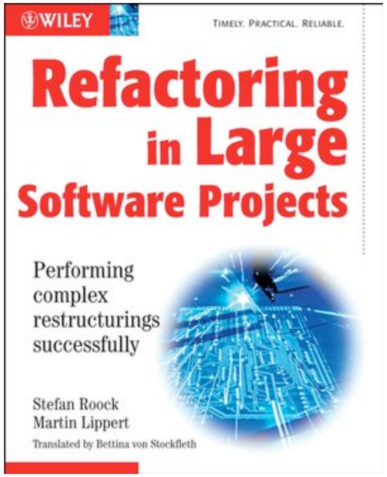


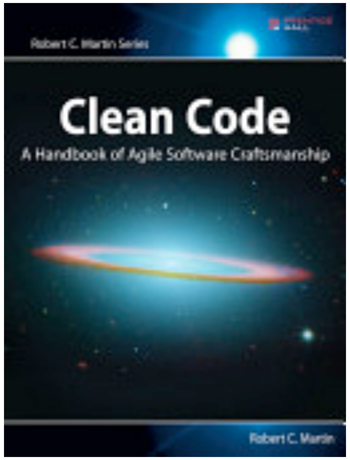
Micheal Feathers
Working Effectively with Legacy Code
Prentice Hall, 2005

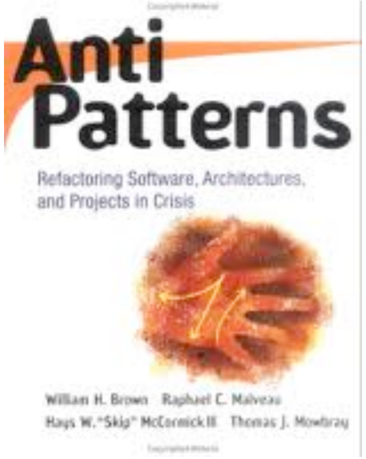
Other useful books


Refactoring Workbook
William C. Wake
Understand the subtle art of recognizing problem areas in code
Select and apply the most important refactoring techniques
Improve existing code—quickly, safely, and effectively


The Addison-Wesley Signature Series
A Martin Fowler Signature Book
Refactoring to Patterns
Joshua Kerievsky
Forewords by Ralph Johnson and Martin Fowler
Afterword by John Brant and Don Roberts


Wiley
Timely. Practical. Reliable.
Refactoring in Large Software Projects
Performing complex restructurings successfully
Stefan Roock
Martin Lippert
Translated by Bettina von Stockfleth




Anti Patterns
Refactoring Software, Architectures, and Projects in Crisis
William H. Brown
Raphael C. Malveau
Hays W. "Skip" McCormick III
Thomas J. Mowbray

Get a mentor!

Conclusions

Dealing with legacy code in a disciplined, effective way

Methods
Techniques
Sources for further learning

But don't forget peculiarities of physics software!

Refactoring techniques and reengineering patterns contribute to improve **computational performance** and facilitate **software validation** (not only maintenance and evolution…)

Thorough testing is the key

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

Post-conclusion

Opportunities for real-life refactoring/reengineering projects after this school

with expert guidance, mentoring

Acquire “good habits”
Contribute to a widely used HEP tool
Contribute to concrete scientific advancement
Work will end up in journal publications

…beware that it would be real work