Journal of Physics: Conference Series **664** (2015) 062048 doi:10.1088/1742-6596/664/6/062048

How do particle physicists learn the programming concepts they need?

S Kluth[1], M G Pia[2], T Schoerner-Sadenius[3], P Steinbach[4]

[1] Max Planck Institute for Physics, Föhringer Ring 6, 80805 Munich, Germany
[2]INFN Sezione di Genova, Via Dodecaneso 33, 16146 Genova, Italy
[3] DESY-FH/CMS, Notkestr. 85, D-22607 Hamburg, Germany
[4] Scientific Computing Facility, Max Planck Institute of Molecular Cell Biology and Genetics, Pfotenhauerstr. 108, 01307 Dresden, Germany

E-mail: Maria.Grazia.Pia@cern.ch, skluth@mpp.mpg.de, thomas.schoerner@desy.de, steinbac@mpi-cbg.de

Abstract. The ability to read, use and develop code efficiently and successfully is a key ingredient in modern particle physics. We report the experience of a training program, identified as "Advanced Programming Concepts", that introduces software concepts, methods and techniques to work effectively on a daily basis in a HEP experiment or other programming intensive fields. This paper illustrates the principles, motivations and methods that shape the "Advanced Computing Concepts" training program, the knowledge base that it conveys, an analysis of the feedback received so far, and the integration of these concepts in the software development process of the experiments as well as its applicability to a wider audience.

1. Introduction

Large software systems play a fundamental role in detector simulation, detector operation, data read-out and analysis [1, 2, 3, 4, 5] in modern particle and nuclear physics. The software systems used hereby involve object-oriented (OO) frameworks of typically more than 10^5 lines of source code. As the task they try to tackle exposes a high complexity, the software that maps a solution of the task is complex as well.

Scientists that would like to record data, build new detectors or analyse data are required to use these systems to extract knowledge, improve detectors designs and test scientific hypotheses in order to eventually answer scientific questions. This applies to all stages of the academic career: it concerns students, post-docs and senior physicists. However, often physicists are not adequately trained by means of the standard university curriculum to object-oriented programming (OOP).

This has lead to the situation, that a high proportion of the daily academic work is dedicated to learn how to program, to understand and potentially fix source code and to layout and implement new functionality that is either missing or present in an unusable fashion. As this fact has to be acknowledged as a reality, there was a need for suitable training on the topic for physicists.

We report the experience of a training program, identified as "Advanced Programming Concepts" (APC), that introduces software concepts, methods and techniques to work effectively on a daily basis in a HEP experiment or other programming intensive fields. The program is


Content from this work may be used under the terms of the Creative Commons Attribution 3.0 licence. Any further distribution of this work must maintain attribution to the author(s) and the title of the work, journal citation and DOI.
Published under licence by IOP Publishing Ltd

targeted at students and young researchers involved in physics analysis and detector development or related software heavy activities, not only at core software developers of relevant scientific code bases. The APC workshop introduces basic and advanced programming techniques as well as elements of the software development process and project management skills. Emphasis is given to methods on how to work effectively with existing code, to improve code and to build a basis for further self-improvement in the field.

The paper is laid out as follows: section 2 identifies the key motivations behind the APC curriculum, section 3 illustrates the contents of the school and provides insight in the pedagogical ideas applied, section 4 highlights key findings of the participant survey that was conducted and finally section 5 concludes the discussion.

2. Ideas behind the School

Object-orientation is currently the prevalent programming paradigm adopted in scientific codes in High Energy Physics (HEP). However, OOP is hard to learn and even harder to put to efficient and effective use.

As the generation of scientific results is of utmost priority to most scientific stakeholders and group leaders, the common approach of handling the complexity of software for newcomers in research groups is to read-up on internet based tutorials or arbitrary monographs in order to learn a minimal set of programming language features. Then scientists turn to examples or peer produced source code and alter parameters and individual lines of code to produce the application behavior aspired. At the cost of considerable inefficiency, the described mindset is retained and carried out throughout entire PhD and post-doc careers. Not only does the task complexity in scientific careers as well as deadline pressure increase, but also many scientists are expected to contribute back to experimental code bases which makes these codes subject to software quality and performance regressions if not approached with care.

Having the above in mind, the APC set out in 2010 [6] to convey central aspects of OO software design to scientists whose programming skills are moderate on average. The core motivation was not to teach the academic depths of multi-facetted requirement analysis, user story identification and software modeling as commonly taught in computer science based courses of the like. We aspired more to extract carpentry-style aspects of software design and map them to exercises and examples that the participants can relate to.

The idea condensed in teaching recipes to solve re-occurring problems in OOP which go by the names of "class design principles" [7] and "design patterns" [8] and practice these within well defined exercises together with peers. We intended to help beginners grasp the OO methods by means of practise, apply and train proven solutions in a collaborative fashion and most of all take concrete code snippets back to their home institute.

The APC workshop program is largely different from that of most computing schools addressed to young high energy physicists, which focus on teaching elementary C++ programming or are addressed to core developers of software frameworks. The APC program focuses on the needs of "normal" young physicists in their everyday work in high energy physics experiments: by far, this category represents the vast majority of young physicists. It is worthwhile to note that not only individuals, but also the experiments as a whole, would benefit from improving the knowledge base, software development skills and methodological awareness of this large category of the experiments' manpower.

Four APC workshops were held between 2010 and 2014, which were attended by 138 participants in total. As the curriculum has evolved over the years, the core ideas of communicating essential concepts of OOP, training on software design and maintenance remained at the heart of the school. Thus, the next section will discuss these in detail with a focus on the workshop as given in 2014 [9].

3. School Layout and Content

3.1. Test-Driven Development and class design principles

Many of our participants work in a code-centric environment. This means, their programming skills have not yet reached a level of maturity where they could easily concentrate on the design of the software rather than the implementation.

Therefore, we started in 2011 to introduce them to unit tests and test-driven development (TDD, [10]). The idea behind this was to provide a method to assure the programmer that behavior already implemented does not change due to redesigns or extensions of the code base. At the heart of test-driven development, this idea is taken even further whereas the canonical test-aided programming work-flow (write tests on implemented functionality) is inverted: the tests of functionality are provided first, then the implementation satisfying these tests are put in place and eventually the software design is re-evaluated and/or updated before new tests are provided, i.e. the TDD cycle starts over.

These concepts are first demonstrated live while implementing a simple vector class in C++ or python - the choice of language depends on where the majority of the audience feels most at home in. Simple member functions such as `get`, `set` methods or `add` functionality are used as an example to demonstrate the TDD work-flow based on a xUnit style unit test framework (see SUnit as the first of this kind, [11]) - we used `Boost.Test` [12] here.

The students are then asked to practice TDD on their own by extending the vector to offer a `magnitude` member function which ultimately should even take different policies on how to calculate the magnitude (Minkowski metric, Euclidean metric, etc.). Already during the live coding, the students are encouraged to code along in order to have everyone see the lecture contents first hand running on his/her laptop.

Even though, the students are not expected to pick up TDD as their style of programming, this lecture provides an entrance to contemplating programming from a higher level. TDD not only touches on software quality and reproducibility, it also makes participants think about independent feature sets of their classes and a defensive mindset when implementing new features.

The latter discussion of independent feature sets of an object-oriented class, imminently leads to the question, if there are best practices on how to design a class. The class design lecture in APC that tries to answer this follows immediately and is loosely based on [7]. The Single-Responsibility principle, Open/Closed principle, the Liskov Substitution principle, Interface Segregation and the Dependency Inversion principle are discussed in depth. Also, package design principles are covered if time allows it.

In this part of the workshop, a more abstract level of thinking about OOP is conveyed and a solid understanding of inheritance and its relation to a hierarchy of feature sets is achieved. The Unified Modeling Language (UML) is put to use in order to visualize source code that complies or refuse the principles states above. Even though small pen-and-paper exercises to fortify the contents are provided, the lecture is laid out in an open fashion so as to adapt to the speed of the participants to a high degree [13].

3.2. Object Oriented methods, Unified Modeling Language and Design Patterns

Essentially all of our students, and in fact most of the lecturers, did not have significant formal education during their university studies on computer science in general or OOP in particular. Therefore students often have difficulties to appreciate the benefits of object oriented programming for creating large software systems, since they never had the opportunity to understand the scientific arguments leading to OOP. In a lecture based on classic books on OOP, the fundamental principles are presented and contrasted with procedural programming in order to highlight the differences.

The Unified Modelling Language UML is used to present class relations, e.g. to discuss the

class design principles. The APC has a lecture combined with paper-and-pencil exercises to explain the UML for classes, relations between classes, for objects, and for sequences of events between objects. The UML has a detailed formal definition and in the lecture we only present what is needed to discuss design issues with pencil and paper or on a blackboard. The main idea is to establish a common language for discussing software design issues which goes beyond writing down code or pseudo-code. The UML concepts are discussed as analogue to developing formulae or Feynman diagrams to simplify understanding a physics problem and writing down the correct solution. UML diagrams are then consistently used to present more advanced topics in the course of the school. With UML a more structured discussion about architecture and design of software systems becomes possible.

With OOP, class design principles, the UML language and the structured code development practice of TDD discussed, many students ask with justification how for a real project they should start. At the heart of this question is that, of course, a single class cannot solve even a small software project. At this point the "Design patterns" come into play [8]. The design patterns collected in the well-known book were inspired by the idea of collecting common solutions for common architecure design problems as initiated by C. Alexander [14]. In the lecture and exercises the most important and common object oriented programming class design patterns are presented and discussed with examples drawn from HEP software whenever possible. Going beyond the GoF (Gang of Four) design patterns some more HEP software specific design patterns are presented and discussed as well. This helps students to grasp the concepts behind many of the large software systems in HEP.

3.3. Software Engineering and Refactoring

In a lecture combined with a hands-on session all aspects presented so far are brought together and combined with the topic of refactoring [15].

Refactoring refers to the process of improving the design of existing code without changing its behaviour and functionality.

Dealing with existing code is the situation students and post-doctoral associates commonly face in their project assignments in high energy physics experiments, which are characterized by a long life-cycle. Since in a young physicist's project either already existing code has to be modified or new code has to be added and made functional, it is clear that almost always the work is done inside a body of code which already has some function or behaviour. The room assigned to refactoring in the APC workshop program recognises this common situation. Refactoring is presented in the APC lectures not only as a technique to improve the design of existing code - often as the a necessary step to modify or add functionality, but also as a set of practical software design guidelines to develop the students' new code according to good quality standards.

Emphasis is placed on formalising the development process into a sequence of small steps, supported by a solid set of tests, where the changes between steps are small and the tests verify that no errors where introduced and the behaviour of the original code was unaltered.

The hands-on session is based on a prepared small C++ project, which is modified by the students in several small steps to either change the internal structure to prepare for adding a new feature, or adding the new feature together with its unit test.

Only with all three ingredients, namely unit testing, OOP (including for our purposes class design principles and design patterns) and refactoring, a complete practice of software engineering applicable to the daily basic needs of programming in HEP emerges. The test discipline guides in building up a working body of code with built-in verification, OOP (as defined above) helps in finding an appropriate structure for the code, and refactoring is the method to systematically make changes to existing code.

The conceptual framework of the school drives the students to understand these relationships

and to appreciate how following this systematic approach can make their activity as programmers in HEP much more productive.

3.4. Performance and modern programming techniques

As section 3.3 covered essential methods and tools to re-engineer and modernize code, techniques of state-of-the-art C++ and performance improvements to exploit modern computer architectures have been added to the curriculum.

HEP based codes yield a characteristic performance footprint: embarassingly parallel single node applications heavy on I/O, which exploit data-parallelism on multiple levels. For this reason, we introduced one entire half-day lesson that focusses on performance measurements and programming techniques to use multi-core systems as well as SIMD instructions in a approachable way.

We put the focus on performance evaluation first by discussing modern CPU architectures from a very high level and then turning to open-source and free tools to evaluate application performance (`iperf`, `valgrind`, etc.). We emphasize the importance of measurements first over the common trend to exploit parallelism and related low-level CPU features at will where it might not be needed. The latter is a common culprit that many beginners invest too much time in without achieving significant gains, and at the same time producing overly complex code. For introducing multi-threading into applications, we chose to teach OpenMP [16]. For SIMD instructions, we teach compiler based auto-vectorisation [17] and SSE intrinsics [18], if time permits.

In recent editions of the APC workshops, we concluded the programme by returning to a more source code based discussion. As the majority of HEP codes apply runtime feature selection by virtual inheritance, we introduced a discussion on an alternative way in C++ to structure code. For this, we perform a live-coding session without any slides entirely (as opposed the lessons discussed in section 3.1, where live-coding and slide presentation is mixed). Here, we start by introducing the `template` keyword to functions in simple C++ applications, going further to templated classes, compile-time interfaces as realized by the curiously-recurring-template pattern [19] and finalizing this session by unrolling a loop at compile time.

4. Evaluation

Various methods of evaluation are applied to collect feedback from the participants of the APC workshops. Questionnaires circulated at the end of the workshop provide an immediate appraisal of the lectures. For purposes of this paper, we have conducted an evaluation to collect information on long-term effects, based on an online survey embedded in each workshops indico web page. Given the relatively small number of students involved, a significant role is also played by informal, direct communication between students and lecturers, who remain reachable after the formal completion of the school programme.

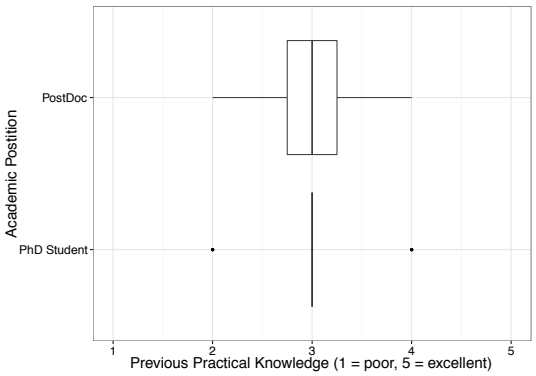
The evaluation of long-term effects comprised a question catalog of 22 items (see Appendix A). Participants were asked to submit their answers anonymously. The evaluation was sent to participants of all workshops. In total 23 individuals have replied within the deadline for the submission of this paper; thereof 4 participants from the 2010, 4 from 2011, 5 individuals from 2012 and 11 entries from 2014 were recorded. From this sample, a bias towards the workshop program of 2014 is to be expected. 19 out of 23 participant were PhD students at the time of their course; 4 were Post-Docs.

As the statistics is very limited at the time of writing this paper, we will only highlight certain aspects of the evaluation. Further, all figures listed below were obtained from the full data sets irrespective of the participants' workshop year.

Figure 1 summarizes the answers of participants of APC on how they estimated their programming and software design proficiency before the course. The evaluation inquired on the



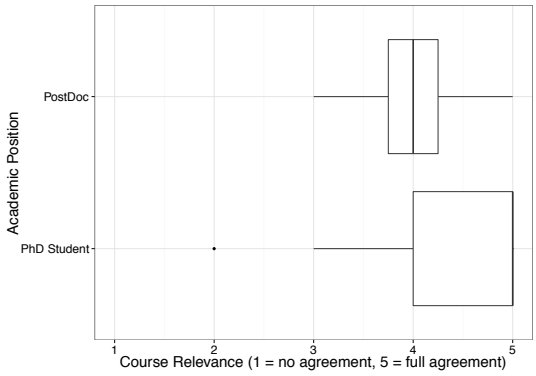
(a) Candidates estimate of their prior theoretical knowledge before the course.



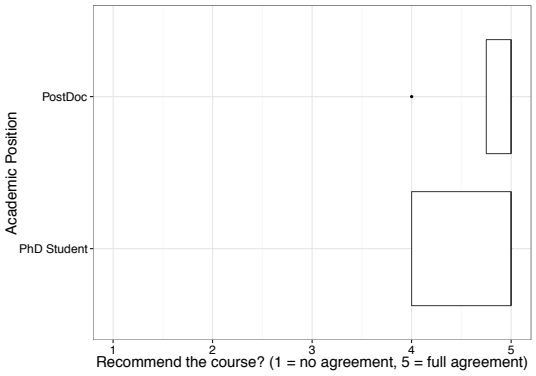
(b) Candidates estimate of their prior practical knowledge before the course.

Figure 1: Box plots of the candidates estimates on their prior knowledge (for sub-figure 1a referring to theory and for sub-figure 1b to the practical aspects) on advanced programming concepts prior to the school. The vertical box outer limits denoted the 25% and 75% quantile limits of the answer. The bold vertical line inside the box marks the arithmetic mean of the sample.

theoretical knowledge they had (design patterns, programming methodology, etc.) and on the practical aspects of it (implementation details, language specifics, etc.). In both fields, candidates consider themselves average (figure 1b) with a slight tendency towards good knowledge (figure 1a).



(a) Candidates estimate of the workshops relevance for their work.



(b) Candidates reply if they would recommend the workshop to others.

Figure 2: Box plots of the candidates estimates on whether they felt the APC contents to be relevant for their work (sub-figure 2a and whether participants would recommend the workshop to their peers (sub-figure 2b). The vertical box outer limits denoted the 25% and 75% quantile limits of the answer. The bold vertical line inside the box marks the arithmetic mean of the sample.

Figure 2 illustrates the effect that the APC school had on its participants. Figure 2a indicates that the material conveyed is fully relevant for the work of the PhD students that came. The Post-Docs appear to consider it to be relevant, but not to 100%. This might be due to several factors: first, Post-Docs are not expected to spent all of their time dedicated to code. Second,

PhD students are more likely to be in the situation of adding new feature sets to existing source code motivated by a physics question.

Figure 2b emphasizes that the course focus, teaching and overall layout of the APC was well received by all participants as both Post-Docs and PhD students would fully recommend the course to others.

Although limited by the small data sample size at the time of submitting this paper, the statistical data analysis appears consistent with the immediate evaluation questionnaires and the informal feedback collected by some of the lecturers in their direct interactions with APC workshop participants. It confirms that the need for training, mentoring and collaborative exercise of advanced software design concepts is very high and that the community would profit greatly from an extension or continuation of the efforts established in APC.

5. Summary

This report illustrates the principles and methods that shape the "Advanced Computing Concepts" training program, the knowledge base that it conveys, an analysis of the feedback received so far, and the integration of these concepts in the software development process of the experiments as well as its applicability to a wider audience.

It intends to promote a discussion in the software-oriented particle physics community on the responsibility of better preparing our young people for their work in the experiments, and on how the experiments could profit from a wider knowledge of advanced software methods and techniques.

Acknowledgments

We would like to thank the Helmholtz Alliance "Physics at the Terascale" for financially supporting the workshop over four years. We also would like to thank past contributors: Thomas Velz (University of Bonn, now industry) was a participant once in the workshop and contributed as a teacher in 2014 ([9], Benedikt Hegner (CERN, [20]) and Eckhardt von Toerne (University of Bonn, [6]) both made substantial contributions to past workshops.

References

[1] Antcheva I, Ballintijn M, Bellenot B, Biskup M, Brun R, Buncic N, Canal P, Casadei D, Couet O, Fine V, Franco L, Ganis G, Gheata A, Maline D G, Goto M, Iwaszkiewicz J, Kreshuk A, Segura D M, Maunder R, Moneta L, Naumann A, Offermann E, Onuchin V, Panacek S, Rademakers F, Russo P and Tadel M 2009 *Computer Physics Communications* **180** 2499 – 2512 ISSN 0010-4655 40 {YEARS} {OF} CPC: A celebratory issue focused on quality software for high performance, grid and novel computing architectures URL `http://www.sciencedirect.com/science/article/pii/S0010465509002550`

[2] Bockelman B 2014 *J.Phys.Conf.Ser.* **513** 042006

[3] Clemencic M, Degaudenzi H, Mato P, Binet S, Lavrijsen W *et al.* 2010 *J.Phys.Conf.Ser.* **219** 042006

[4] Chauhan N *et al.* (ATLAS) 2014 *J.Phys.Conf.Ser.* **513** 052022

[5] Agostinelli S *et al.* 2003 *Nucl. Instr. Meth. A* **506** 250–303

[6] 2010 Advanced programming concepts workshop `https://indico.desy.de/confRegistrantsDisplay.py/list?confId=3155`

[7] Martin R 2003 *Agile Software Development: Principles, Patterns, and Practices* Alan Apt series (Pearson Education) ISBN 9780135974445 URL `https://books.google.de/books?id=0HYhAQAAIAAJ`

[8] Gamma E, Helm R, Johnson R and Vlissides J 1994 *Design Patterns: Elements of Reusable Object-oriented Software* (Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc.) ISBN 0-201-63361-2

[9] 2014 Advanced programming concepts workshop `https://indico.desy.de/conferenceDisplay.py?confId=9932`

[10] Beck K 2002 *Test Driven Development: By Example* 1st ed (Addison-Wesley Professional) ISBN 0321146530

[11] Beck K 1999 *Kent Beck's Guide to Better Smalltalk: A Sorted Collection* SIGS Reference Library (SIGS) ISBN 9780521644372 URL `https://books.google.de/books?id=Y7FwNB4GV4EC`

[12] *Boost Test Library* `http://www.boost.org/doc/libs/1_55_0/libs/test/doc/html/index.html`

[13] Ambrose S, Bridges M, DiPietro M, Lovett M, Norman M and Mayer R 2010 *How Learning Works: Seven Research-Based Principles for Smart Teaching* Jossey-Bass higher and adult education series (Wiley) ISBN 9780470617601 URL `https://books.google.de/books?id=gu5qpi5aFDkC`
[14] C Alexander S Ishikawa M S 1977 *A Pattern Language: Towns, Buildings, Construction* (Oxford University Press)
[15] Fowler M, Beck K, Brant J, Opdyke W and Roberts D 2012 *Refactoring: Improving the Design of Existing Code* (Addison-Wesley) google eBook
[16] OpenMP Architecture Review Board 2013 *OpenMP Application Program Interface* `http://www.openmp.org/mp-documents/OpenMP4.0.0.pdf`
[17] *Auto-vectorization in GCC* `https://gcc.gnu.org/projects/tree-ssa/vectorization.html`
[18] *Intel Intrinsics Guide* `https://software.intel.com/sites/landingpage/IntrinsicsGuide/`
[19] Coplien J O 1995 Curiously recurring template patterns Tech. rep. Software Production Research Dept., Bell Laboratories c++ Report
[20] 2011 Advanced programming concepts workshop `https://indico.desy.de/conferenceDisplay.py?confId=4542`

Appendix A. Evaluation Catalog

(i) The year you were born (open text field)
(ii) Your gender (choice: male or female)
(iii) Academic position at the time of the workshop (choice: Student, PhD Student, Post-Doc, PI/Group Leader, Technician, Other)
(iv) My theoretical knowledge on the subject prior to the course (choice: "1 = Poor" to "5 = Excellent")
(v) My practical experience on the subject prior to the course (choice: "1 = Poor" to "5 = Excellent")
(vi) Course content (choice: "1 = Poor" to "5 = Excellent")
(vii) Course structure (choice: "1 = Poor" to "5 = Excellent")
(viii) Preparation of the course (choice: "1 = Poor" to "5 = Excellent")
(ix) Focus of the course on relevant points (choice: "1 = Poor" to "5 = Excellent")
(x) Illustration of possible applications (choice: "1 = Poor" to "5 = Excellent")
(xi) Course duration appropriate to the content (choice: "1 = Poor" to "5 = Excellent")
(xii) Course materials (choice: "1 = Poor" to "5 = Excellent")
(xiii) Encouragement of the learning process (choice: "1 = Poor" to "5 = Excellent")
(xiv) Enthusiasm of lecturer with the subject matter (choice: "1 = Poor" to "5 = Excellent")
(xv) Availability of lecturer for questions during / after course (choice: "1 = Strongly disagree" to "5 = Strongly Agree")
(xvi) The course is relevant for my current work (choice: "1 = Strongly disagree" to "5 = Strongly Agree")
(xvii) The course broadened my general comprehension (choice: "1 = Strongly disagree" to "5 = Strongly Agree")
(xviii) I benefited from the course (choice: "1 = Strongly disagree" to "5 = Strongly Agree")
(xix) I enjoyed attending the course (choice: "1 = Strongly disagree" to "5 = Strongly Agree")
(xx) I would recommend this course to others (choice: "1 = Strongly disagree" to "5 = Strongly Agree")
(xxi) General Comments (positive) (open multi-line text field)
(xxii) General Comments (negative) (open multi-line text field)