Computer – Máy Tính

Nguyễn Quản Bá Hồng*

Ngày 1 tháng 11 năm 2024

Tóm tắt nội dung

This text is a part of the series *Some Topics in Advanced STEM & Beyond*:
URL: https://nqbh.github.io/advanced_STEM/.
Latest version:

- *Computer – Máy Tính.*
  PDF: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer/NQBH_computer.pdf.
  TEX: URL: https://github.com/NQBH/advanced_STEM_beyond/blob/main/computer/NQBH_computer.tex.

Mục lục

1 Wikipedia

1.1 Wikipedia/abstraction (computer science)

"In software engineering & computer science, *abstraction* is the process of generalizing concrete details, e.g. attributes, away from the study of objects & systems to focus attention on details of greater importance. Abstraction is a fundamental concept in computer science & software engineering, especially within the object-oriented programming paradigm. E.g.:

*A Scientist & Creative Artist Wannabe. E-mail: nguyenquanbahong@gmail.com. Bến Tre City, Việt Nam.

- the usage of abstract data types to separate usage from working representations of data within programs.
- the concept of functions or subroutines which represent a specific way of implementing control flow;
- the process of reorganizing common behavior from groups of non-abstract classes into abstract classes using inheritance & subclasses, as seen in object-oriented programming languages.

1.1.1 Rationale

> "The essence of abstraction is preserving information that is relevant in a given context, & forgetting information that is irrelevant in that context." – John V. Guttag

Computing mostly operates independently of the concrete world. The hardware implements a model of computation that is interchangeable with others. The software is structured in architectures to enable humans to create the enormous systems by concentrating on a few issues at a time. These architectures are made of specific choices of abstractions. Greenspun's 10th rule is an aphorism on how such an architecture is both inevitable & complex.

Language abstraction is a central form of abstraction in computing: new artificial languages are developed to express specific aspects of a system. *Modeling languages* help in planning. *Computer language* from the machine language to the assembly language & the high-level language. Each stage can be used as a stepping stone for the next stage. The language abstraction continues e.g. in scritping languages & domain-specific programming languages.

Within a programming language, some features let the programmer create new abstractions. These include subroutines, modules, polymorphism, & software components. Some other abstractions such as software design pattens & architectural styles remain invisible to a translator & operate only in the design of a system.

Some abstractions try to limit the range of concepts a programmer needs to be aware of, by completely hiding the abstractions they are built on. The software engineer & writer Joel Spolsky has criticized these efforts by claiming that all abstractions are *leaky* – that they can never completely hide the details below; however, this does not negate the usefulness of abstraction.

Some abstractions are designed to inter-operate with other abstractions – e.g., a programming language may contain a foreign function interface for making calls to the lower-level language.

1.1.2 Abstraction features

Programming languages. Main article: Wikipedia/programming language. Different programming languages provide different types of abstraction, depending on the intended applications for the language. E.g.:

- In OOP languages e.g. C++, Object Pascal, or Java, the concept of *abstraction* has itself become a declarative statement – using the syntax `function(parameters) = 0;` (in C++) or the keywords `abstract` & `interface` (in Java). After such a declaration, it is the responsibility of the programmer to implement a class to instantiate the object of the declaration.
- Functional programming languages commonly exhibit abstractions related to functions, e.g. lambda abstractions (making a term into a function of some variable) & higher-order functions (parameters are functions).
- Modern members of the Lisp programming language family e.g. Clojure, Scheme, & Common Lisp support macro systems to allow syntactic abstraction. Other programming languages such as Scala also have macros, or very similar metaprogramming features (e.g., Haskell has Template Haskell, & OCaml has MetaOCaml). These can allow a programmer to eliminate boilerplate code, abstract away tedious function call sequences, implement new control flow structures, & implement Domain Specific Languages (DSLs), which allow domain-specific concepts to be expressed in concise & elegant ways. All of these, when used correctly, improve both the programmer's efficiency & the clarity of the code by making the intended purpose more explicit. A consequence of syntactic abstraction is also that any Lisp dialect & in fact almost any programming language can, in principle, be implemented in any modern Lisp with significantly reduced (but still nontrivial in most cases) effort when compared to "more traditional" programming languages such as Python, C, or Java.

Specification methods. Main article: Wikipedia/formal specification. Analysts have developed various methods to formally specify software systems. Some known methods include:

- Abstract-model based method (VDM, Z);
- Algebraic techniques (Larch, CLEAR, OBJ, ACT ONE, CASL);
- Process-based techniques (LOTOS, SDL, Estelle);
- Trace-based techniques (SPECIAL, TAM);
- Knowledge-based techniques (Refine, Gist).

Specification languages Main article: Wikipedia/specification language. Specification languages generally rely on abstractions of 1 kind or another, since specifications are typically defined earlier in a project, (& at a more abstract level) than an eventual implementation. The UML specification language, e.g., allows the definition of *abstract* classes, which in a waterfall project, remain abstract during the architecture & specification phrase of the project.

1.1.3 Control abstraction

Main article: Wikipedia/control flow. Programming languages offer control abstraction as 1 of the main purposes of their use. Computer machines understand operations at the very low level such as moving some bits from 1 location of the memory to another location & producing the sum of 2 sequences of bits. Programming languages allow this to be done in the higher level. E.g., consider this statement written in a Pascal-like fashion: `a := (1 + 2) * 5`. To a human, this seems a fairly simple & obvious calculation. However, the lower-level steps necessary to carry out this evaluation, & return the value 15, & then assign that value to the variable `a`, are actually quite subtle & complex. The values need to be converted to binary representation (often a much more complicated task than one would think) & the calculations decomposed (by the compiler or interpreter) into assembly instructions (again, which are much less intuitive to the programmer: operations such as shifting a binary register left, or adding the binary complement of the contents of 1 register to another, are simply not how humans think about the abstract arithmetical operations of addition or multiplication). Finally, assigning the resulting value of 15 to the variable labeled `a`, so that `a` can be used later, involves additional 'behind-the-scenes' steps of looking up a variable's label & the resultant location in physical or virtual memory, storing the binary representation of 15 to that memory location, etc.

Without control abstraction, a programmer would need to specify *all* the register/binary-level steps each time they simply wanted to add or multiply a couple of numbers & assign the result to a variable. Such duplication of effort has 2 serious negative consequences:

1. it forces the programmer to constantly repeat fairly common tasks every time a similar operation is needed.

2. it forces the programmer to program for the particular hardware & instruction set.

Structured programming. Main article: Wikipedia/structured programming. Structured programming involves the splitting of complex program tasks into smaller pieces with clear flow-control & interfaces between components, with a reduction of the complexity potential for side-effects.

In a simple program, this may aim to ensure that loops have single or obvious exit points & (where possible) to have single exit points from functions & procedures.

In a larger system, it may involve breaking down complex tasks into many different modules. Consider a system which handles payroll on ships & at shore offices:

- The uppermost level may feature a menu of typical end-user operations.
- Within that could be standalone executables or libraries for tasks such as signing on & off employees or printing checks.
- Within each of those standalone components there could be many different source files, each containing the program code to handle a part of the problem, with only selected interfaces available to other parts of the program. A sign on program could have source files for each data entry screen & the database interface (which may itself be a standalone 3rd party library or a statically linked set of library routines).
- Either the database or the payroll application also has to initiate the process of exchanging data with between ship & shore, & that data transfer task will often contain many other components.

These layers produce the effect of isolating the implementation details of 1 component & its assorted internal methods from the others. Object-oriented programming embraces & extends this concept.

1.1.4 Data abstraction

Main article: Wikipedia/abstract data type. Data abstraction enforces a clear separation between the *abstract* properties of a data type & the *concrete* details of its implementation. The abstract properties are those that are visible to client code that makes use of the data type – the *interface* to the data type – while the concrete implementation is kept entirely private, & indeed can change, e.g. to incorporate efficiency improvements over time. The idea is that such chances are not supposed to have any impact on client code, since they involve no difference in the abstract behavior.

E.g., one could define an abstract data type called *lookup table* which uniquely associates *keys* with values, & in which values may be retrieved by specifying their corresponding keys. Such a lookup table may be implemented in various ways: as a hash table, a binary search tree, or even a simple linear list of `(key:value)` pairs. As far as client code is concerned, the abstract properties of the type are the same in each case.

Of course, this all relies on getting the details of the interface right in the 1st place, since any changes there can have major impacts on client code. As 1 way to look at this: the interface forms a *contract* on agreed behavior between the data type & client code; anything not spelled out in the contract is subject to change without notice.

1.1.5 Manual data abstraction

While much of data abstraction occurs through computer science & automation, there are times when this process is done manually & without programming intervention. 1 way this can be understood is through data abstraction within the process of conducting a systematic review of the literature. In this methodology, data is abstracted by 1 or several abstractors when conducting a meta-analysis, with errors reduced through dual data abstraction followed by independent checking, known as adjudication.

1.1.6 Abstraction in OOP

Main article: Wikipedia/object (computer science). In OOP theory, *abstraction* involves the facility to define objects that represent abstract "actors" that can perform work, report on & change their state, & "communicate" with other objects in the system. The term encapsulation refers to the hiding of state details, but extending the concept of *data type* from earlier programming languages to associate *behavior* most strongly with the data, & standardizing the way that different data types interact, is the beginning of *abstraction*. When abstraction proceeds into the operations defined, enabling objects of different types to be substituted, it is called polymorphism. When it proceeds in the opposite direction, inside the types or classes, structuring them to simplify a complex set of relationships, it is called delegation or inheritance.

Various OOP languages offer similar facilities for abstraction, all to support a general strategy of polymorphism in object-oriented programming, which includes the substitution of 1 type for another in the same or similar role. Although not as generally supported, a configuration or image or package may predetermine a great many of these bindings at compile-time, link-time, or loadtime. This would leave only a minimum of such bindings to change at run-time.

Commmon Lisp Object System or Self, e.g., feature less of a class-instance distinction & more use of delegation for polymorphism. Individual objects & functions are abstracted more flexibly to better fit with a shared functional heritage from Lisp.

C++ exemplifies another extreme: it relies heavily on templates & overloading & other static bindings at compile-time, which in turn has certain flexibility problems.

Although these examples offer alternate strategies for achieving the same abstraction, they do not fundamentally alter th need to support abstract nouns in code – all programming relies on an ability to abstract verbs as functions, nouns as data structures, & either as processes.

Consider e.g. a sample Java fragment to represent some common farm "animals" to a level of abstraction suitable to model simple aspects of their hunger & feeding. It defines an `Animal` class to represent both the state of the animal & its functions:

```
public class Animal extends LivingThing
{
    private Location loc;
    private double energyReserves;

    public boolean isHungry() {
        return energyReserves < 2.5;
    }

    public void eat(Food food) {
        // Consume food
        energyReserves += food.getCalories();
    }

    public void moveTo(Location location) {
        // Move to new location
        this.loc = location;
    }
}
```

With the above definition, one could create objects of type `Animal` & call their methods like this:

```
thePig = new Animal();
theCow = new Animal();
if (thePig.isHungry()) {
    thePig.eat(tableScraps);
}
if (theCow.isHungry()) {
    theCow.eat(grass);
}
theCow.moveTo(theBarn);
```

In the above example, the class `Animal` is an abstraction used in place of an actual animal, `LivingThing` is a further abstraction (in this case a generalization) of `Animal`.

If one requires a more differentiated hierarchy of animals – to differentiate, say, those who provide milk from those ho provide nothing except meat at the end of their lives – that is an intermediary level of abstraction, probably `DairyAnimal(cows,goats)` who would eat foods suitable to giving good milk, & `MeatAnimal(pigs,steers)` who would eat foods to give the best meat-quality.

Such an abstraction could remove the need for the application coder to specify the type of food, so they could concentrate instead on the feeding schedule. The 2 classes could be related using inheritance or stand alone, & the programmer could define varying degrees of polymorphism between the 2 types. These facilities tend to vary drastically between languages, but in general each can achieve anything that is possible with any of the others. A great many operation overloads, data type by data type, can

have the same effect at compile-time as any degree of inheritance or other means to achieve polymorphism. The class notation is simply a coder's convenience.

Object-oriented design. Main article: Wikipedia/object-oriented design. Decisions regarding what to abstract & what to keep under the control of the coder become the major concern of object-oriented design & domain analysis – actually determining the relevant relationships in the real world is the concern of object-oriented analysis or legacy analysis.

In general, to determine appropriate abstraction, one must make many small decisions about scope (domain analysis), determine what other systems one must cooperate with (legacy analysis), then perform a detailed object-oriented analysis which is expressed within project time & budget constraints as an object-oriented design. In our simple example, the domain is the barnyard, the live pigs & cows & their eating habits are the legacy constraints, the detailed analysis is that coders must have the flexibility to feed the animals what is available & thus there is no reason to code the type of food into the class itself, & the design is a single simple `Animal` class of which pigs & cows are instances with the same functions. A decision to differentiate `DairyAnimal` would change the detailed analysis but the domain & legacy analysis would be unchanged – thus it is entirely under the control of the programmer, & it is called an abstraction in object-oriented programming as distinct from abstraction in domain or legacy analysis.

1.1.7 Considerations

When discussing formal semantics of programming languages, formal methods or abstract interpretation, *abstraction* refers to the act of considering a less detailed, but safe, definition of the observed program behaviors. E.g., one may observe only the final result of program executions instead of considering all the intermediate steps of executions. Abstraction is defined to a *concrete* (more precise) model of execution.

Abstraction may be *exact* or *faithful* w.r.t. a property if one can answer a question about the property equally well on the concrete or abstract model. E.g., if one wishes to know what the result of the evaluation of a mathematical expression involving only integers $+, -, \cdot$, is worth module n, then one needs only perform all operations module n (a familiar form of this abstraction is casting out nines).

Abstractions, however, though not necessarily *exact*, should be *sound*. I.e., it should be possible to get sound answers from them – even though the abstraction may simply yield a result of undecidability. E.g., students in a class may be abstracted by their minimal & maximal ages; if one asks whenever a certain person belongs to that class, one may simply compare that person's age with the minimal & maximal ages; if his age lies outside the range, one may safely answer that the person does not belong to the class; if it does not, one may only answer "I don't know".

The level of abstraction included in a programming language can influence its overall usability. The Cognitive dimensions framework includes the concept of *abstraction gradient* in a formalism. This framework allows the designer of a programming language to study the trade-offs between abstraction & other characteristics of the design, & how changes in abstraction influence the language usability.

Abstractions can prove useful when dealing with computer programs, because nontrivial properties of computer programs are essentially undecidable (Rice's theorem). As a consequence, automatic methods for deriving information on the behavior of computer programs either have to drop termination (on some occasions, they may fail, crash or never yield out a result), soundness (they may provide false information), or precision (they may answer "I don't know" to some questions).

Abstraction is the core concept of abstract interpretation. Model checking generally takes place on abstract versions of the studied systems.

1.1.8 Levels of abstraction

Main article: Wikipedia/abstraction layer. Computer science commonly presents *levels* (or, less commonly, *layers*) of abstraction, wherein each level represents a different model of the same information & processes, but with varying amounts of detail. Each level uses a system of expression involving a unique set of objects & compositions that apply only to a particular domain. Each relatively abstract, "higher" level builds on a relatively concrete, "lower" level, which tends to provide an increasingly "granular" representation. E.g., gates build on electronic circuits, binary on gates, machine language on binary, programming language on machine language, applications & operating systems on programming languages. Each level is embodied, but not determined, by the level beneath it, making it a language of description that is somewhat self-contained.

Database systems. Main article: Wikipedia/database management system. Data abstraction levels of a database system. Since many users of database systems lack in-depth familiarity with computer data-structures, database developers often hide complexity through the following levels"

- **Physical level.** The lowest level of abstraction describes *how* a system actually stores data. The physical level describes complex low-level data structures in detail.
- **Logical level.** The next higher level of abstraction describes *what* data the database stores, & what relationships exist among those data. The logical level thus describes an entire database in terms of a small number of relatively simple structures. Although implementation of the simple structures at the logical level may involve complex physical level structures, the user of the logical level does not need to be aware of this complexity. This is referred to as physical data independence. Database administrators, who must decide what information to keep in a database, use the logical level of abstraction.

- **View level.** The highest level of abstraction describes only part of the entire database. Even though the logical level uses simpler structures, complexity remains because of the variety of information stored in a large database. Many users of a database system do not need all this information; instead, they need to access only a part of the database. The view level of abstraction exists to simplify their interaction with the system. The system may provide may views for the same database.

Layered architecture. Main article: Abstraction layer. The ability to provide a design of different levels of abstraction can

- simplify the design considerably
- enable different role players to effectively work at various levels of abstraction
- support the portability of software artifacts (model-based ideally)

System designs & business process design can both use this. Some design processes specifically generate designs that contain various levels of abstraction.

Layered architecture partitions the concerns of the application into stacked groups (layers). It is a technique used in designing computer software, hardware, & communications in which system or network components are isolated in layers so that changes can be made in 1 layer without affecting the others." – Wikipedia/abstraction (computer science)

2 Linux

Resources – Tài nguyên.

1. [Sho19]. William Shotts. *The Linux Command Line: A Complete Introduction.*

I used SUSE & OpenSUSE in WIAS Berlin but I do not like it so I go back to Ubuntu.

3 Programming

3.1 C/C++

Resources – Tài nguyên.

1. [Ngọ02]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C.*
2. [Ngọ03]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C++.*
3. [Str13]. Bjarne Stroustrup. *The C++ Programming Language.*
4. [Str18]. Bjarne Stroustrup. *A Tour of C++.*

3.2 Pascal

Resources – Tài nguyên.

1. [Ngọ08]. Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình Pascal.*
2. [Ngọ09]. Quách Tuấn Ngọc. *Bài Tập Ngôn Ngữ Lập Trình Pascal.*
3. [DT06]. Lê Văn Doanh, Trần Khắc Tuấn. *101 Thuật Toán & Chương Trình Bài Toán Khoa Học Kỹ Thuật & Kinh Tế Bằng Ngôn Ngữ Turbo-Pascal.*

3.3 Python

Resources – Tài nguyên.

1. [Đức22]. Nguyễn Tiến Đức. *Tuyển Tập 200 Bài Tập Lập Trình Bằng Ngôn Ngữ Python.*
2. [Huy24]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 1.*
3. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_2**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 2.*
4. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_3**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 3.*
5. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_4**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 4.*

6. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_5**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 5.*

7. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_6**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 6.*

8. [**Huy_sang_tao_thuat_toan_lap_trinh_tap_7**]. Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 7.*

4 Software

4.1 FeNiCS

Resources – Tài nguyên.

1. [Dok20]. Jørgen S. Dokken. *Automatic shape derivatives for transient PDEs in FEniCS & Firedrake.*

2. [LL16]. Hans Petter Langtangen, Anders Logg. *Solving PDEs in Python.*

4.2 Firedrake

4.3 Fireshape

Resources – Tài nguyên.

1. [PW20]. Alberto Paganini, Florian Wechsung. *Fireshape Documentation, Release 0.0.1.*

2. [PW21]. Alberto Paganini, Florian Wechsung. *Fireshape: a shape optimization toolbox for Firedrake.*

4.4 Git

Resources – Tài nguyên.

1. [CS14]. Scott Chacon, Ben Straub. *Pro Git.*

4.5 Gmsh

Resources – Tài nguyên.

1. [GR09]. Christophe Geuzaine, Jean-François Remacle. *Gmsh: A 3D finite element mesh generator with built-in pre- & post-processing facilities.*

4.6 OpenFOAM

Resources – Tài nguyên.

1. There are 3 variants of OpenFOAM:
   (a) OpenFOAM.com: Commercial.
   (b) OpenFOAM.org: Open-source with a large community.
   (c) Extended OpenFOAM.

2. [GW22]. Christopher Greenshields, Henry Weller. *Notes on Computational Fluid Dynamics: General Principles.*

3. [TN13]. M. Towara, U. Naumann. *A Discrete Adjoint Model for OpenFOAM.*

4.7 ParMooN

Resources – Tài nguyên.

1. [Wil+17]. Ulrich Wilbrandt, Clemens Bartsch, Naveed Ahmed, Volker John. *ParMooN – a modernized program package based on mapped finite elements.*

4.8 SU2

4.9 Sublime Text

Resources – Tài nguyên.

1. [Bos14]. Wes Bos. *Sublime Text Power User: A Complete Guide.*

2. [Pel13]. Dan Peleg. *Mastering Sublime Text*

5 Miscellaneous

Tài liệu

[Bos14] Wes Bos. *Sublime Text Power User: A Complete Guide*. 2014, p. 202.

[CS14] Scott Chacon and Ben Straub. *Pro Git*. 2nd. Apress, 2014, p. 458.

[Dok20] Jørgen S. Dokken. "Automatic shape derivatives for transient PDEs in FEniCS and Firedrake". In: (2020). URL: https://arxiv.org/abs/2001.10058.

[DT06] Lê Văn Doanh and Trần Khắc Tuấn. *101 Thuật Toán & Chương Trình Bài Toán Khoa Học Kỹ Thuật & Kinh Tế Bằng Ngôn Ngữ Turbo-Pascal*. In lần thứ 10. Nhà Xuất Bản Khoa Học & Kỹ Thuật, 2006, p. 268.

[Dức22] Nguyễn Tiến Dức. *Tuyển Tập 200 Bài Tập Lập Trình Bằng Ngôn Ngữ Python*. Nhà Xuất Bản Đại Học Thái Nguyên, 2022, p. 327.

[GR09] Christophe Geuzaine and Jean-François Remacle. "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *Internat. J. Numer. Methods Engrg.* 79.11 (2009), pp. 1309–1331. ISSN: 0029-5981. DOI: 10.1002/nme.2579. URL: https://doi.org/10.1002/nme.2579.

[GW22] Christopher Greenshields and Henry Weller. *Notes on Computational Fluid Dynamics: General Principles*. Reading, UK: CFD Direct Ltd, 2022.

[Huy24] Nguyễn Xuân Huy. *Sáng Tạo Trong Thuật Toán & Lập Trình. Tập 1*. Tái bản lần 10. Nhà Xuất Bản Thông Tin & Truyền Thông, 2024, p. 371.

[LL16] Hans Petter Langtangen and Anders Logg. *Solving PDEs in Python*. Vol. 3. Simula SpringerBriefs on Computing. The FEniCS tutorial I. Springer, Cham, 2016, pp. ix+148. ISBN: 978-3-319-52461-0; 978-3-319-52462-7. DOI: 10.1007/978-3-319-52462-7. URL: https://doi.org/10.1007/978-3-319-52462-7.

[Ngọ02] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C*. Nhà Xuất Bản Thống Kê, 2002, p. 425.

[Ngọ03] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình C++*. Nhà Xuất Bản Thống Kê, 2003, p. 476.

[Ngọ08] Quách Tuấn Ngọc. *Ngôn Ngữ Lập Trình Pascal*. Nhà Xuất Bản Thống Kê, 2008, p. 338.

[Ngọ09] Quách Tuấn Ngọc. *Bài Tập Ngôn Ngữ Lập Trình Pascal*. Nhà Xuất Bản Giáo Dục, 2009, p. 187.

[Pel13] Dan Peleg. *Mastering Sublime Text*. Packt Publishing, Birmingham - Mumbai, 2013, pp. iv+94.

[PW20] Alberto Paganini and Florian Wechsung. "Fireshape Documentation, Release 0.0.1". In: (2020), pp. ii+31. URL: https://fireshape.readthedocs.io/en/latest/index.html.

[PW21] Alberto Paganini and Florian Wechsung. "Fireshape: a shape optimization toolbox for Firedrake". In: *Struct. Multidiscip. Optim.* 63.5 (2021), pp. 2553–2569. ISSN: 1615-147X. DOI: 10.1007/s00158-020-02813-y. URL: https://doi.org/10.1007/s00158-020-02813-y.

[Sho19] William Shotts. "The Linux Command Line: A Complete Introduction". In: (2019), p. 640.

[Str13] Bjarne Stroustrup. *The C++ Programming Language*. 4th edition. Pearson Addison-Wesley, 2013, pp. xiv+1346.

[Str18] Bjarne Stroustrup. *A Tour of C++*. 2nd edition. Pearson Addison-Wesley, 2018, pp. xii+240.

[TN13] M. Towara and U. Naumann. "A Discrete Adjoint Model for OpenFOAM". In: *Procedia Computer Science* 18 (2013), pp. 429–438. DOI: 10.1016/j.procs.2013.05.206. URL: https://doi.org/10.1016/j.procs.2013.05.206.

[Wil+17] Ulrich Wilbrandt, Clemens Bartsch, Naveed Ahmed, and et al. "ParMooN—a modernized program package based on mapped finite elements". In: *Comput. Math. Appl.* 74.1 (2017), pp. 74–88. ISSN: 0898-1221. DOI: 10.1016/j.camwa.2016.12.020. URL: https://doi.org/10.1016/j.camwa.2016.12.020.