Computation and Social Science

2

2.1 Introduction and Motivation

Computation is a formal discipline used by scientists—in the social, physical, and biological disciplines—to uncover new insights and advance the frontiers of knowledge. It also informs the **Computational Paradigm of Social Science** introduced in Chap. 1. Social processes are *algorithmic*, and social systems are supported by *algorithms*, in the sense defined in this chapter. What are the elements of computation with the greatest significance for CSS? How is computation used to better understand social systems and processes? What are the core concepts and principles of social computation? Problem-solving, design, and programming are core elements of computation and the computational approach to social science. Similar activities are also foundational to understanding social systems.

The role of computation in CSS is comparable to that of mathematics in physics: it is used as a language to formalize theory and empirical research to express, study, and develop our understanding of social complexity in ways that are not accessible through other means. By contrast, pure computer scientists use computation to study computing, just as pure mathematicians use mathematics to study mathematics. This instrumental or utilitarian motivation does not prevent computational social scientists from developing deep interest in computation; there is much a computational social scientist can learn from the pattern of thinking of a computer scientist, a musician, a mathematician, or a historian. However, CSS is more like applied computer science or applied mathematics[1]: the formal approach (mathematical languages or programming languages) is used to gain substantive, domain-based knowledge about social complexity in all its rich forms.

[1]For example, applied computer scientists work on areas such as robotics, data analysis, and optimization, to name some of the major areas of research in computer science.

C. Cioffi-Revilla, *Introduction to Computational Social Science*,
Texts in Computer Science, DOI 10.1007/978-3-319-50131-4_2

This chapter uses the Python programming language for illustrative purposes, though not for providing tutorials. The notational graphic system known as the Unified Modeling Language (UML) is used for representing and better understanding social systems and processes—including those with significant theoretical or real-world complexity. Importantly, UML is also used in subsequent chapters to describe social systems and processes, such as decision-making by actors, polities and their institutions, socio-environmental dynamics, and other entities of social science research interest.[2]

2.2 History and First Pioneers

Computation has a long, interesting history in social science. Computational Social Science (CSS) began with the first applications of computation during the early 1960s, with pioneers such as Harold Guetzkow (1963), Herbert A. Simon (1969), Karl W. Deutsch (1963), John C. Loehlin (1968), and Samuel J. Messick (1963), roughly a decade after von Neumann's (1951) pioneering Theory of Automata. That was during the age of punched tape, 80-column IBM cards, and long hours spent at the university's computer center awaiting output results, often in vain due to some syntactical glitch in the program, which often caused another day's worth of work. In spite of such early difficulties, the advent of computation in social science came at an auspicious time, because theoretical and methodological advances were taking place along numerous frontiers across disciplines. Field Theory (Lewin 1952), Functionalist Theory (Radcliffe-Brown 1952), Conflict Theory (Richardson 1952a, 1952b), the Theory of Groups (Simon 1952), Political Systems Theory (Easton 1953), as well as Decision-making Theory (Allais 1953), among others, required new formalisms that could treat conceptual and theoretical complexity of human and social dynamics, beyond what could be accomplished through systems of mathematical equations solved in closed form.

Each of the social sciences (Anthropology, Economics, Political Science, Social Psychology, and Sociology) and related fields (Geography, History, Communication, Linguistics, Management Science) witnessed the introduction of computation into its own frontiers of theory and research within a few years. However, formal training in computation did not begin until decades later through high-level software packages for statistical applications (SPSS, SAS, Stata), followed by true programming languages (S and R), as well as computational applications to content analysis, network models, and social simulations. Many of these computational contributions will be examined in subsequent chapters of this book.

Those were the origins of CSS, a fledging field that has evolved from pioneering roots that began with primitive algorithms running on archaic computers with

[2]The material in this chapter assumes a level of computer science knowledge comparable to Eric Grimson and John Guttag's famous MIT course (Grimson and Guttag 2008) or Guttag (2013).

(mostly) historical interest, to today's object-oriented models running on modern and more powerful computers that would have seemed like science fiction even to Isaac Asimov's psychohistorian Hari ("The Raven") Seldon in *Foundations*. What about the future? The future of CSS will be written in the language of advanced distributed computing, graphic processing units (GPU), quantum computing, and other information technologies still at the frontiers of computational science.

2.3 Computers and Programs

2.3.1 Structure and Functioning of a Computer

All computers are information-processing systems: they compute, as the term indicates, based on a set of instructions called a **program**. Programs are written as a series of instructions, not unlike a recipe, in computer code. The code must be written so that it conforms to the format of the programing language, or syntax.

All computation can be seen as a **problem-solving system** consisting of subsystems of **hardware** and **software** components. While hardware provides the physical means for information processing (i.e., computing machines, or computers, in a narrow sense), software provides the algorithmic instructions that tell the hardware what to do (i.e., what to do with the information being processed) in some programming language. In computer science, software is also known as **code**, not to be confused with the same term as used in social science measurement and empirical research to represent the value of some variable (usually a nominal variable). Computationally speaking, code is distinct from **data**, which are processed by code.[3]

These initial ideas have resonance in social science, where information-processing systems are ubiquitous, significant, and highly consequential: individuals, groups, and institutions ranging from local neighborhoods to the global system of international organizations process information following procedures, engage in problem-solving, and use institutions (akin to hardware?) as well as established and adaptive systematic processes (software?) to address and solve problems pertaining to the full spectrum of societal issues. The mapping between computers and social systems is not exact, nor is it necessary for computation to be useful in social science, but it can be insightful in pointing out significant features of social complexity that extant social theories have neglected or simply been unable to explain. Metaphors are often useful in science, but for computation to be a powerful paradigm and methodology in CSS it is necessary to look deeper into its concepts and principles.

[3] In social science, data and information denote different concepts. Data (the lower level concept) normally refers to raw observations, such as field data or census data, whereas information (higher level) is based on, or is derived from, data and provides a basis for knowledge. Data is the plural of *datum* (or fact, in Latin), so the correct phrases are "one *datum*" and "several *data*".

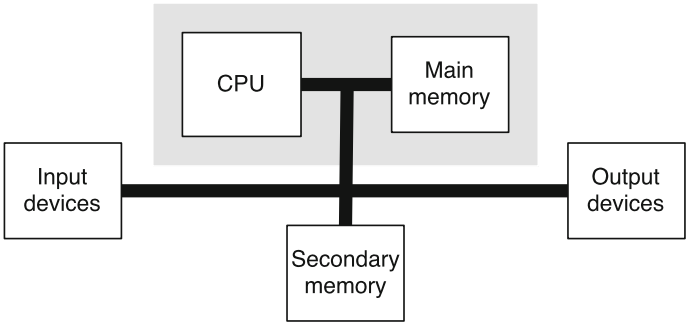

Fig. 2.1 A computer with its functional components (the five boxes) based on a bus architecture (fast-speed data connections)

As illustrated in Fig. 2.1, in its most fundamental structure, a computer is a machine (hardware system) composed of five types of components, each designed to perform a specific function. There are two core components: a **central processing unit** (CPU) and **main memory**. The CPU carries out the most basic computations, such as arithmetic operations, comparisons, or Boolean true/false operations. Data and programs are stored in main memory (or RAM, random access memory), which has a tightly coupled interactive relationship with the CPU for performing computations (i.e., executing instructions).

Secondary memory (Fig. 2.1, lower center), in larger and typically slower form than main memory (e.g., a disk), is used to store information more permanently (as programs and data files) when a computer is turned off. When a computer is turned on, secondary memory is accessed to retrieve data and programs that are executed by the CPU using main memory.

Input and **output devices** are for us humans to interact with the three components just described, as human–machine interfaces (Fig. 2.1, left and right). Input devices include keyboard, mouse, microphones, cameras, joysticks, and many kinds of sensors ranging from relatively simple (e.g., a thermostat fixed to a wall) to highly complex (biohazard sensors mounted on an unmanned autonomous vehicle or UAV). Output devices include printers, speakers, electromechanical devices (e.g., robots), and other devices. The earliest monitors were output devices, whereas today some monitors serve a dual function as input devices as well (e.g., a touch-sensitive video screen).

Fast data connections (called "internal buses") link core components (CPU and main memory) between themselves and, via other connections (also called "expansion buses"), with external components (I/O devices). The overall architecture of internal components with relations among them, versus external devices in the environment of a computer, bears resemblance to Herbert A. Simon's model of a complex adaptive system consisting of an inner system and an external environment—a paradigmatic model and theory that we will examine in much closer detail later, given its significance for explaining and understanding social complexity. This important approach is still mostly unknown in the social sciences, 40 years after Simon's

pioneering work, and Simon is remembered mostly for his work on bureaucracy and incrementalism.

When a computer is turned on and a program (of any kind) is asked to run, the operating system handles what is called the *load-fetch-decode-execute cycle*, or **fetch-execute cycle** for short—and it is something a CSS researcher needs to know. Understanding this is helpful for deciding, for instance, whether a model can be implemented to run on a single processor, or whether some form of parallel processing is necessary. First, program instructions are loaded ("fetched") from secondary memory, where they reside (almost) permanently, onto the main memory (RAM). Second, the CPU accesses the first instruction from RAM, decodes that instruction, and executes it. When finished executing the first instruction, the same fetch–execute cycle is repeated as many times as there are instructions in the program. A well-written program will organize this cycling process in such a way as to take advantage of the fast cycling time of the CPU, subject to available RAM capacity. Knowledge of this cycling process is not necessary for most programing tasks, but becomes increasingly important with parallelization, especially GPU programming. Most multithreading is handled at a higher level of abstraction.

A significant feature of the fetch–execute cycle is that it consists of discrete events that are (i) critically necessary (i.e., conjunctive), (ii) sequential in a strict order (sequential conjunction), and (iii) each event takes time that cumulatively determines total cycle duration. While this is common knowledge for computer scientists, few social scientists have paid close attention at the deep properties and principles of such systems and processes of sequential conjunction for explaining and understanding social complexity. On time scales that are many orders of magnitude slower than computers, human cognition, decision-making by individuals and groups, policy-making, and numerous other social processes examined in this book—especially in Chaps. 6 and 7—share significant isomorphic features with fundamental patterns in computation, such as the sequential conjunction of the fetch–execute cycle and others.

DID YOU KNOW THAT …? Comparing the time-scales of computers with that of individual humans and human institutions adds perspective to information processing under different architectures of complexity. A MacBook Pro laptop computer has a 2.66 GHz Intel Core i7 CPU and four GB 1067 MHz DDR3 RAM chips. CPU speed is measured in cycles per second (or hertz), so this means that the CPU of the MacBook Pro laptop can execute $2{,}660{,}000{,}000 = 2.66 \times 10^9$ instructions per second. High speeds such as these allow a modern computer to execute many instructions in background mode while a relatively idle program, such as a word processor, is in use.
Suppose we compare an instruction execution by a CPU to a policy decision by a national legislative body. No one has yet estimated the number of decisions made *each year* by such institutions, but it is clearly many orders of magnitude slower. By contrast, human individual decision-making takes place on a scale of tens of milliseconds.

2.3.2 Compilers and Interpreters

A CPU understands only its own **machine language**, whereas most computer programs are written in a high-level language. In order for a computer to run a program written in a given high-level language (i.e., a program in other than low-level machine language), the program must first be either **compiled** or **interpreted**. The difference between these two processes is fundamental, subtle, consequential, and important for CSS researchers to understand. A **compiler** is a program that literally translates source code written in a high-level programming language (e.g., Fortran, C++, Pascal, Python) into machine code that is specific to and executed by the computer's CPU. Once compiled, a program can then be run many times without having to recompile the source code. Compiled code is machine-specific binary code, ready for execution by the CPU; it provides a complete translation of all instructions, line by line.

By contrast, other languages that are not compiled use an **interpreter** that is specific to the high-level language for communicating the program's instructions to a computer. An interpreter is a specialized, low-level program that enables hardware to execute the high-level software. A language requiring an interpreter must use its associated interpreter every time the program is executed; otherwise the CPU will not understand the program's instructions.

In sum, a compiler translates a program into machine code, line by line, to execute; an interpreter reads all the source code and directly communicates its instructions in machine code to the CPU without compiling a new program (as the compiler does). The main difference is similar to knowing a foreign language (compiling) versus translating one line at a time (interpreting). Comparing the two types of high-level languages, compiled programs run relatively faster but have drawbacks, whereas interpreted programs run somewhat slower but they can run interactively. The difference is important for a CSS researcher to understand, because it can mean choosing one programming language over another, depending on what model or algorithm is being implemented.[4]

2.4 Computer Languages

Social science uses mathematics as a language to formalize theory and investigate features of social complexity that are exclusively accessible through the medium of mathematical structures, such as sets, probability, game theory, or dynamical systems. The same is true when using computer languages in CSS. A **computer language** is a structured, formal grammar for communicating with and controlling

[4]The case of Java is somewhat hybrid: Java is technically compiled into Java byte code, and then just-in-time compiled into machine code by the Java Virtual Machine (JVM)—which can be viewed as a byte code interpreter.

Table 2.1 Comparison of computer programming languages. Paradigm types are explained in the text. *Source:* Wikipedia, "Comparison of programming languages: General comparison"

Assembly language	Imperative
BASIC	Imperative, procedural
C	Imperative, procedural
C++	Imperative, object-oriented, procedural
Fortran	Imperative, object-oriented, procedural
Java	Imperative, object-oriented, reflective
Lisp	Imperative, functional
Mathematica	Imperative, functional, procedural
MATLAB	Imperative, object-oriented, procedural
Pascal	Imperative, procedural
Python	Aspect-oriented, functional, imperative, object-oriented, reflective
S and R	Functional, imperative, object-oriented, procedural

what a computer does. Like all languages, including mathematical structures used by social scientists, computer languages consist of **syntax**, **semantics**, and **pragmatics**.[5] Syntax refers to the proper rules for writing instructions, the correct sentences of a properly written program. Semantics refers to the meaning of symbols; i.e., what various code elements stand for. Pragmatics refers to the primary purpose, function, or paradigmatic orientation of a language. Computer languages differ by intent, just like different symbolic systems or mathematical structures are created for various purposes (e.g., music notation or game theory).

Social science has used a significant array of mathematical structures over the past two hundred years, but formal instruction in mathematics has lagged behind statistics. Now, in addition to statistics and mathematics, social scientists require training in programming languages, which is essential for CSS theory and research on social complexity.

Every computer language has features that make it more or less effective in implementing human and social dynamics, just as is true for different modeling languages used in mathematical social science. Specifically, each programing language has its own syntax, semantics, and pragmatics, which results in features such as those listed in Table 2.1.

Python is a programming language with several desirable features for learning Computational Social Science: it is easy to learn *and* can be used to learn some of the best **computer programming habits**, such as consistent style, modularity,

[5]Linguists would also add **genetics**, the origin of a specific language. For example, the Python programming language was created by Guido van Rossum in the late 1980s and has since evolved into version 3 (as of this writing), supported by a global community.

defensive coding, unit testing, and commenting.[6] A drawback of Python is that it can slow down considerably with increasing program complexity, such as when used for social simulations such as those examined later in this book. A recommended strategy is to learn how to program using Python, then learn a more advanced language, such as Java or C++.

Several other specifically technical features of Python include:

- **Object-orientation:** Python is a language that supports the object-orientation to programming (OOP), meaning that the basic building blocks of a Python program can represent social entities (e.g., actors, relations, groups), similar to the building blocks of many social theories. In turn, social entities (objects and associations) contain within them ("encapsulate") variables and dynamics that determine the state of the overall social entity or phenomenon being modeled. By contrast, earlier programming languages required direct modeling of variables and equations, which is sometimes too difficult, cumbersome, or impractical for many social theories.[7]
- **Interpreted code:** Python code is interpreted, not compiled, so it can be run interactively. This is helpful for several purposes: developing a program as it grows from simple to more complicated; verification and debugging; running simulation experiments. Python code runs from a command line terminal or from a shell editor, as well as interactively or as an executable file.
- **Imperative style:** As an imperative language, a program written in Python can contain statements that change the state of the program. This means that a Python program can implement a series of commands or instructions that the computer can execute to change the state of social objects, constructs, or entities represented in the program.[8] Assignment statements, looping statements, and conditional branching are important features of imperative programing.
- **Function libraries:** As with other popular programing languages, Python supports the use of functions that are evaluated in terms of specific arguments. Given a function `f(x)` with argument `x`, the evaluation of `f` always returns the same result as long as `x` does not change. Functions are used to implement many kinds of social processes, such as utility functions in decision-making, interaction dynamics, and other behavioral features. Functions need not always be mathematical equations. For example, they can be table functions.

[6] By contrast, *bad* programming habits include lack of modularity, hazardous loops that can easily spin forever, "stringy" code, and comments that are unclear, unhelpful, quirky, or plain absent. Good coders avoid these and other bad habits and strive to develop an excellent, "tight" style, as discussed later in this chapter.

[7] This is a significant advantage of OOP that will arise again in various chapters. The main idea of the object-orientation to programming is that basic social *entities* and *relations* are identified first; all the rest (variables, data, parameters, equations) come later.

[8] By contrast, a so-called **declarative** style of programming emphasizes the desired result of a program, not the instructions necessary to produce results.

Python can be used for many scientific purposes in CSS, running in both *interactive* and *batch* modes. As a *calculator*, Python can be used to compute results, such as a probability value or an expected value, just like a hand calculator. More complicated functions are best analyzed in batch mode.

Example 2.1 (Interaction Between Human Communities) In human and social geography, the potential for many modes of human interactions between two communities (marriages, migrations, and phone calls, among others) is approximated by the so-called *gravity model:*

$$I \approx \frac{P_1 P_2}{D^{\alpha}}, \tag{2.1}$$

where I is the interaction potential, and P_1 and P_2 are the populations of the two communities separated by distance D. The exponent α denotes the difficulty involved in realizing interactions (costs, terrain, transportation opportunities, and similar), such that I decays rapidly with increasing α. Suppose two communities with 20,000 and 30,000 inhabitants are 120 miles away from each other. To appreciate the effect of difficulty α on the interaction potential I, we can compute the potential with $\alpha = 2$ (standard assumption) and 3 (greater difficulty), respectively:

```
>>> print((30000)*(20000)/(120**2))
41666.666666666664
>>> print((30000)*(20000)/(120**3))
347.22222222222223
```

We see immediately how a single unit difference in difficulty α (2 vs. 3) causes a drop in interaction potential of two orders of magnitude (10^4 vs. 10^2).

Example 2.2 (Terrorist Attacks) Terrorists face a daunting challenge when planning an attack, mainly because the probability of success in carrying out an attack (technically called a *compound event*, as we will examine later in greater detail) is contingent on many things going well: planning, recruitment of confederates (e.g., scouts, suppliers, operatives, etc.), training in weapons and tactics, proper target selection, execution, and overcoming target passive and active defenses, among other requisites. Assuming $N = 10$ critical requirements for a successful attack, each being solved with probability $q = 0.99$, we get:

```
>>> print(.99**10)
0.9043820750088044
```

Under somewhat more realistic (but still generous) assumptions, with a lower 0.90 probability of requirement-level success:

```
>>> print(.90**10)
0.3486784401000001
```

In fact, as demonstrated in subsequent chapters, the partial derivative $\partial(q^N)/\partial q$ is highly sensitive to the probability of individual task success q (more than to N). This explains why counterterrorism strategies aimed at hindering individual tasks are quite effective, without having to target every single stage of a potential attack process.

Python can be used as a simple calculator for exploring, analyzing, and learning more about social models as in these and other examples. Note that typing `print()` is not necessary, strictly speaking, but it is a good habit because when running a batch script it is always necessary to use print to output results.

Alternatively, and more interestingly from a research perspective, Python can be used for running *programs* for investigating a large spectrum of social models—from individual decision-making to global dynamics in the international system—that have been analyzed only through closed form solutions, without using the power of simulation and other CSS approaches. However, due to speed limitations mentioned earlier, running social simulations or network models in Python can be problematic in terms of speed, so a better methodological approach in some cases is to implement models in other, faster languages, such as Java or C++. This is why Java is a common language for multi-agent simulation systems or "toolkits," such as MASON and Repast, and why C++ is often used in parallel, distributed computing. For example, Repast-HPC is based on C++.

The following are other features or types of programming languages mentioned in Table 2.1:

- **Procedural programming:** This refers to the programming paradigm based on procedure calls (in high-level languages) or subroutines (low-level). Routines and methods are procedure calls containing some sequence of computations to be executed.
- **Reflective programming:** The ability of a programming language to read and alter the entire structure of the object at compile time is called reflection.

Why should a CSS researcher know about different features (or *paradigms*, as they are called in computer science) of programming languages? The reasons are similar to why a mathematical social scientist needs to know about what each formal language is capable of modeling. For example, classic dynamical systems can model deterministic interactions, whereas a Markov chain can model probabilistic change, game-theoretic models capture strategic interdependence, and so on for other mathematical languages. Reliance on the same mathematical structure every time (e.g., game theory, as an example), for every research problem, is unfortunately a somewhat common methodological pathology that leads to theoretical decline and a sort of inbreeding visible in some areas of social science research. Dimensional empirical features of social phenomena—such as discreteness–continuity, deterministic–stochastic, finite–infinite, contiguous–isolated, local–global, long-term versus short-term, independence–interdependence, synchronic–diachronic, among others—should determine the choice of mathematical structure(s). Similarly, different programming languages provide different features, so they should be selected in accordance with the nature of the social phenomena to be modeled. The same is true of using programming languages in CSS, for the very same reason: not all problems can (or should!) be solved with the same scientific tool.

2.5 Operators, Statements, and Control Flow

The examples in the previous section used the **interactive mode** in Python, which works well for simple calculations or short code snippets that are brief and are used just once or a small number of times. When the calculations are more complex, when instructions need to be executed several times, or when the sequence of instructions is longer than just a few lines, it makes more sense to create a separate file containing a **program** consisting of statements. Then the program can be written, edited, and saved, just like any text file. The program is then executed any number of times by running (or *calling*) it from the command line or from Python's own shell (e.g., IDLE).

The following program illustrates a number of ideas concerning operators, statements, and control flow.

Example (Chaotic Process (Zelle 2010: 13)) This example is taken from a leading textbook on Python, illustrating the nature of chaotic processes. Write the following simple program in a text file (say, `chaos.py`) and run it from the Python shell.

```
>>> def main():
 print("This program illustrates a chaotic function")
 x = eval(input("Enter a number between 0 and 1:"))
 for i in range(10):
    x = 3.9 * x * (1 - x)
    print(x)

main()
```

When `main()` runs, it should return the following result:

```
This program illustrates a chaotic function
Enter a number between 0 and 1:
```

Next, enter a number between 0 and 1, and the program should return a sequence of 10 values. Change the range from 10 to N, call `main()` again, and now N values will be returned. The coefficient can also be changed to a value different from 3.9, which will generate a different chaotic series.

The example just discussed contains a number of points worth noting from a CSS perspective. First, it takes relatively little in terms of program sophistication to opt for a program, rather than using the interactive mode. Or we may wish to run a program with variations for conducting **computational experiments**. Most social models require some statements that warrant a program, even when the number of **lines of code (LOC)** is relatively small (i.e., less than a dozen), as in the example. Copying and pasting in interactive mode helps, but calling a program (e.g., as in `>>> import filename`) is even easier, and that is what most researchers would do.[9]

Second, the structure of a program is always a function of "The Question" (or set of questions) being asked in a given investigation. In this case, the question concerned the behavior of a chaotic process; specifically, which series would be generated by a given initial value, assuming a specific coefficient. A different program would be necessary to address closely related but different questions, such as:

- What happens when noise is introduced?

[9]Computer programs are artifacts—in the sense of Simon—which sometimes, in turn, provide support to other artifacts. An example of this is a spacecraft. As of early 2012 the International Space Station orbiting Earth—one of the world's most complex adaptive artifacts—was supported by computer programs with approximately 2.3 million LOC, a figure always increasing with growing project complexity until the ISS mission is completed. Unfortunately, however, LOC per se are not a good proxy measure for algorithmic or software complexity: high LOC may reflect mere lack of expertise, whereas low LOC may result from overly complicated implementations, instead of simpler, maintainable versions that would require more LOC.

- What if the coefficient varies as a function of some other parameter affecting the process?
- What is the correlation between series of values generated by different initial conditions (or different coefficients)?
- How can we graph the process, as in a time-series plot, rather than observe a list of numbers?

None of these questions can be addressed by the same program, especially the last, which requires calling additional facilities, such as Python's `graphics` library. Each program is designed to address a specific question.

Third, note that each statement in a program is intended to control some aspect of the information being processed. In this case the program began by defining a new **function**, called `main`. *Knowing how to define new functions is a basic programming skill* and an easy task in Python. Next, the program states that something is to be printed exactly as specified by the `print` function. This is optional, but *good practice*, since it tells the user what is going on without having to look into details. The program then contains a core statement about evaluating another function, this time an `input` function in response to a query. Next, the program uses a series of related statements to control the computation of `x` by means of a **loop**: `for i in range(10):...`. Loops are essential **control flow** statements along with others, such as `if` and `while` statements.

2.6 Coding Style

Computer programming is a form of formal writing, so style matters and developing a good style for writing programs is important for a computational social scientist—just as it is for a computer scientist. General principles of **good coding style** apply to all programming, while specific principles or guidelines apply to particular programming languages, similar to mathematics in this respect.

The need for **general principles of good coding style** is motivated by many factors that operate in any field of modern science, including Computational Social Science:

- Code is a formal system of writing, so its syntax and semantics are governed by both technical and esthetic principles, not just the former. The same is true of mathematics: well-written mathematical papers are also based on technical and esthetic principles.
- Code is sometimes used by programmers long after it was first written by the original programmer(s). If it was not well-written to begin with, subsequent programmers (or even the initial programmer) may have a difficult time understanding it.

- Many multi-disciplinary projects in CSS contain researchers from diverse backgrounds (social science, computer science, environmental science, or other disciplines), which increases the communications requirements.

The following are important general principles of good coding style:

1. **Readability**: Always write code in such a way that others can easily read and understand it. Code should not be written using short variable names or function names, such as is common practice in mathematics. "`numberOfRefugees-intheCamp`" is good; "`N`" or even "`NORIC`" are not. Incomprehensible code is not a sign of genius; it is a sign of disrespect toward collaborators, current or future.
2. **Commenting**: Writing informative comments is an important way to implement readability. Uncommented code needs to deciphered, or it may be useless. The main consumer of comments is often the original programmer, since even a few days later it is easy to forget what a code segment was intended to do.
3. **Modularity**: Write in modules, such that the overall program is akin to a **nearly decomposable system,** in the sense of Simon. Object-oriented design patterns can be useful when separating components that are not so obviously decomposable. Functions and their embedding property provide a viable strategy for modularization.
4. **Defensive coding:** Writing defensive code means to try to ensure that code does not malfunction, ending up doing something different from the intended purpose. An example would be being careful in avoiding loops that can cycle infinitely. This is achieved by careful coding and by inserting proper tests that will prevent infinite loops.

These basic principles of good coding style are intended not just for beginners; they are also practiced by good modelers and software engineers.

2.7 Abstraction, Representation, and Notation

How does science (any science) make fruitful inquiry feasible and tractable, given the complexity of the real world? The viability of doing science in any field depends on making the subject matter tractable in terms of research that is systematic, reproducible, and cumulative. Social, physical, and biological scientists render their substantive fields tractable through **simplifications** that are sufficient to ensure the growth of a viable science, but not so simple as to preclude deep understanding of phenomena. **Tractability** is therefore a sophisticated strategy of scientific inquiry that seeks to simultaneously maximize **parsimony** and **realism**—as in a *Pareto frontier*. Parsimony ensures causal explanations (theories) and empirical descriptions (laws) that contain a minimal number of factors deemed essential for explanation, understanding, and sometimes prediction. Realism ensures that the science remains

empirically relevant and sufficiently rich in terms of capturing real-world features. Science seeks to make real-world complexity tractable.

Social complexity in the real world of people, their thoughts, decisions, social relations, and institutions, is intricate and far more complex than the simple world of two-body mechanics and equilibrium systems.[10] It consists of individual actors with bounded rationality, interactions that are often hard to predict (even when they are just dyadic), and the emergent social results generate networks, organizations, systems, and processes that challenge all areas of social science theory and research—transcending individual disciplines. To solve this challenge, social science has learned to rely on **abstractions**, **representations**, and specialized **notations** to advance our understanding of the social universe through concepts, theories, and models.

For hundreds of years, since the rise of modern social science in the Age of Enlightenment and the Scientific Revolution, social scientists have used statistical and mathematical representations based on abstractions of real human and social dynamics. All such models—and the social theories they involve—are formal linguistic inventions based on systems of specialized notations.[11] Just as social scientists have learned to use abstractions to formulate statistical and mathematical models of the social world in many domains, today computational social scientists use computer programs and computational models to abstract, represent, analyze, and understand human and social complexity. What do abstraction, representation, and notation require in CSS? How do they work in a coordinated way to produce viable code for modeling and analyzing complex social systems and processes?

Abstraction

In computer science, abstraction means hiding information. In CSS, abstracting from the world "reality"—whether directly experienced (observing a riot downtown) or indirectly learning about it (reading history)—is a process involving stimulus signals, perceptions, interpretation, and cognition. CSS relies on several **sources for abstracting** key entities, ideas, and processes from raw stimulus signals from the real world. These sources span a hierarchy in terms of their social scientific status. At the very top of the hierarchy are **social theories** with demonstrable validity in terms of formal structure (internal validity) and empirical observation (external validity). Not all existing social theories meet these stringent requirements, although an increasing number of them do as research progresses. Examples of social theories that meet

[10] A little-known fact among many social scientists is that the theory of mechanics in physics is built around the abstraction of single- and two-body problems. Already three-body problems are hugely difficult by comparison; and, most interesting, N-body problems defy mathematical solution in closed form.

[11] Interestingly, humanistic fields such as music and ballet also use systems of specialized notation, far beyond what is used in traditional social science. In music, Guido d'Arezzo [b. A.D. 991 (or 992), d. 1050] is considered the founder of the modern music staff; in ballet, Rudolf von Laban [b. 1879, d. 1958] invented the symbolic system known as "labanotation" (Morasso and Tagliasco 1986).

internal and external validity standards include Heider's Theory of Cognitive Balance in psychology, Ricardo's Theory of Comparative Advantage in economics, and Downs's Median Voter Theory in political science, among others. Social theories are abstractions that point to relevant social entities, variables, and dynamics that matter in understanding and explaining social phenomena.

A second source of abstraction consists of **social laws**. Examples of social laws include the Weber–Fechner Law in psychometrics, the Pareto Law in economics, and Duverger's Law in political science. Theories explain; laws describe (Stephen Toulmin 1967).[12] Some of the most scientifically usefully social laws can be stated mathematically, as in these examples. Social laws also contain relevant entities, variables, and functional relations for describing social phenomena.

A third source of abstraction consists of **observations** that can range from formal (e.g., ethnography, content analysis, automated information extraction, text mining, among others) to informal (historical narratives, media, and other sources about social phenomena). Observations of social phenomena can describe actors, their beliefs, social relations, and other features ranging from individual to collective.

Finally, a fourth source of abstraction consists of **computational algorithms** capable of emulating social phenomena, as in artificial intelligence (AI). Artificial (i.e., not really human) algorithms do not claim to be causal in the same sense as social theories. They "work," but without causal claims in the same sense as social theories. They are efficient, in the sense that they (sometimes) can closely replicate social phenomena. AI algorithms are typically (and intentionally) efficient and preferably simple; extreme parsimony in this case comes at the expense of realism. Examples of AI algorithms include Heatbugs (Swarm, NetLogo, MASON), Boids (Reynolds 1987), and Conway's (1970) Game of Life. In spite of their lack of social realism, AI algorithms can be useful sources for abstracting social entities, ideas, or processes because they can highlight features that either elude theories or are hard to observe. An example would be the agglomeration patterns generated in a Heatbugs model, as a function of varying parameters of "social" interaction among the set of agents, or the role of apparent "leadership" in a flock of boids.

Representation

Abstraction is a necessary early step in scientific inquiry, whether in the context of empirical observation, theoretical construction, or model-building. A second step requires representation of abstractions. In CSS this means representing abstracted social entities (e.g., actors, relations, institutions) in a way that a computer can understand sufficiently well to be able to execute a program about such entities.

Why does representation matter? The short answer is: because a computer can only understand sequences of the binary digits 0 and 1. In computer science, Donald E. Knuth is credited with playing an influential role in conceptually separating abstraction from representation (Shaw 2004: 68).

[12] The late international relations theoretician Glenn H. Snyder [1924–2013] spoke often about this dichotomy, which he attributed to the philosopher of science, S. Toulmin.

Table 2.2 Main data types in Python

Type	Description	Examples
`str`	Alphanumeric text	United Nations, climate change, Leviathan
`list`	Mutable sequence	[7.4, 'stress', False]
`tuple`	Immutable sequence	(7.4, 'stress', False)
`set`	Group of unordered elements without duplicates	{7.4, 'stress', False}
`dict`	List of key-value pairs	{'key1': 2.57, 7: True}
`int`	Integer number	7
`float`	Floating point number	2.71828182845904523536
`bool`	Boolean binary values	True, False

The more complete answer—to the question of why representation matters—warrants close attention. Earlier in Sect. 2.3.1, we distinguished between code (instructions) and data. In turn, data can be either numeric or alphanumeric, and numeric data can be either integer or real. Therefore, the information that needs to be represented to the computer (i.e., to both CPU and RAM) consists of four basic types: **real numbers**, **integer numbers** (positive or negative whole numbers, which include *ordinal variables*), **alphanumeric data** (including *nominal* or *categorical variables*), and **instructions**. Numbers, letters, and instructions are all represented in **bits** of information, consisting of sequences of the binary digits 0 and 1. More bits are necessary for representing more information.

Each programming language defines a set of data types as a semantic feature. The main data types defined in Python are summarized in Table 2.2. From a representation perspective, the Python interpreter translates each data type into binary code; i.e., every symbol in the syntax of a program (number, letter, or symbol) is represented as a sequence of the binary digits 0 and 1. The most commonly used data types are `str`, `int`, `float`, and `bool`.[13]

Representation can be seen as having two aspects. **Effective representation** refers to the choice of data types that helps answering the desired research question(s). **Efficient representation,** on the other hand, refers to the choice of data types that minimize computational cost in terms of CPU cycles or RAM size. Achieving both effectiveness and efficiency is challenging.

[13] A boolean variable is called an "indicator variable" in probability and a "dummy variable" in social statistics and econometrics. (Dummy? As supposed to what? A strange phrase, don't you think?).

Notation

Notation is necessary to express representations derived from abstraction. While in statistical and mathematical models, "notation" refers to equations and other formal structures (e.g., matrices, trees, graphs), in computational science the term refers to **programming languages** used to write software code. In 2004 it was observed that "hundreds of billions of lines of software code are currently in use, with many more billions added annually" (Aho and Larus 2004: 74). High-level programming languages (Python, Java, and many others) serve as bridges that span the "semantic gap" (Aho and Larus 2004: 75) between (a) the abstractions that we wish to investigate from the real world, and (b) binary notation understandable to computers. Without high-level programming languages a computational scientist would have no choice but to write software programs in binary code.

Several notational features of modern high-level programming languages (such as Python) are noteworthy:

Specificity: A programming language can be specifically dedicated to solving a narrow range of scientific problems, such as numerical computation, data visualization, or network dynamics.

Portability: A high-level programing language can be used to write code that executes in different computers, even those running different operating systems.

Reliability: Errors are difficult to avoid when writing low-level assembly language code, whereas they are more preventable with higher level programming languages.

Optimization: While binary code executes at astonishing speeds (recall the earlier example of the MacBook Pro CPU cycling at many MHz), "a program written in a high-level language often runs faster" (Aho and Larus 2004: 75) because compiled code is highly optimized. Speed, memory, and energy are the most common goals of optimization.

Multiple approaches: High-level programming languages provide alternative and sometimes multiple approaches to programming, with emphasis on features such as imperative, declarative, and others.

Automated memory management: Information must be stored in main memory (recall Fig. 2.1), which is a major programming task when not automated. Automated memory management is a major useful feature of any high-level programming language.

Other features of modern high-level programming languages include procedures, patterns, constructs, advances in modularity, type checking, and other developments that are constantly being added to facilitate improvements in effectiveness of representation and efficiency of computation.

2.8 Objects, Classes, and Dynamics in Unified Modeling Language (UML)

A fascinating feature of social science is that the subjects of inquiry in the real-world social universe span a remarkable spectrum of ideas, entities, phenomena, systems, and processes, and have many ties to numerous other disciplines across the sciences and humanities. The variety is so great that it is difficult to parse the entire landscape.[14] Not surprisingly, social science encompasses not one, but several disciplines (the Big Five: anthropology, economics, political science, social psychology, sociology) and related fields (communication, education, geography, history, law, linguistics, management), the totality of which is necessary to investigate the social world to understand it.

The vast landscape of all these disciplines includes an extraordinary variety of simple, complicated, and complex subject matter, much of which remains unknown and is poorly understood. So, these exciting scientific opportunities are innumerable! How does CSS handle such rich complexity to advance scientific understanding?

2.8.1 Ontology

Ontology refers to "what exists," or "the landscape of entities of interest," so to speak. It can be said that, from a high-level ontological perspective, the entire social world consists of **social systems** (which can be simple or complex; adaptive or not) and their **environments**, an idea introduced in Chap. 1 as a cornerstone of computational thinking about society and illustrated in Fig. 2.2. All entities in the social world (systems and environments) have a key **ontological** feature in common: they constitute **objects** and **classes** related by **associations** among them. An object belongs to a class, similar to the set-theoretic idea that an element is a member of a set. "Person" and "John Q. Smith," or "Country" and "Spain," are class and object, respectively, from an object-oriented (OO) computational perspective.[15]

The phrases "object-oriented modeling" (OOM) and "object-oriented programming" (OOP) denote, as the terms suggest, an approach to modeling (abstracting and representing) that uses objects as the fundamental ontological entities. Note that the building blocks of computational methodology consist of social entities, not variables. (Variables come later, "encapsulated" in objects.)

Figure 2.3 illustrates people in four different social ontologies or "worlds." Let us consider each in some detail, from an "OO" perspective.

[14] Winston Churchill (1948) said: "History is simply one damned thing after another.".

[15] The idea of a tightly coupled relation between system and environment is also well-captured by the Spanish maxim, "*Yo soy yo y mi circunstancia*" (I am I and my circumstance), by José Ortega y Gasset (1914).

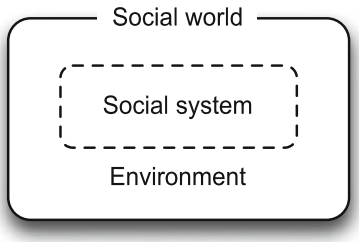

Fig. 2.2 A "social world" consists of a social system situated in its environment. This ontology is foundational for many social theories examined through formal and empirical analysis, including Simon's Theory of Artifacts, the Canonical Theory, and others based on the Complex Adaptive Systems Paradigm. Unfortunately, this graphic representation is useless although common throughout social science. Later in this section we introduce UML as a helpful graphic notation system for representing social worlds

Fig. 2.3 Ontology across scales of human and social systems complexity: The family is the smallest kin-based social system (*upper left*). Teams of people provide assistance in humanitarian crises and disasters (*upper right*). Polities are complex social aggregates capable of producing historical milestones (*lower left*). Humans in space constitute complex, coupled, socio-technical systems operating in extreme environments (*lower right*)

Upper left: **A family**. The first image shows a family consisting of a man, a woman, and a child as three distinct human entities that constitute a class we may call "people" or "family members." The basic association among them is defined by kinship. The environment is a professional photography studio that shows a white wall behind the family. From an OOM computational perspective, people and photo studio are objects with attributes.[16]

Upper right: **Disaster victims**. The second image shows a team of humanitarian crisis workers and a victim being carried on a stretcher. The environment is a rural setting in the aftermath of a hurricane in Indonesia. The associations here are somewhat more complicated, involving collaboration among the aid workers and assistance provided by aid workers to the victim. Here the objects consist of people, artifacts, and natural environment.

Lower left: **Leadership summit**. The third image shows a political gathering of heads of states and governments. Here the associations are even more complex, involving relations among people, polities, symbols, and historical events. The environment is urban, in 1980s Berlin, Germany. Still, the objects are the same: people and artifacts situated in some environment. In this case the environment is built (urban), not natural.

Lower right: **Orbiting astronauts.** The fourth image shows a contemporary space scene consisting of astronauts and a spacecraft (the International Space Station). This is arguably the most complex ontology of the four—a scene that would have been pure fiction just a few years ago. The environment is low Earth orbit (LEO) between 320 km (199 mi) and 400 km (249 mi) above the Earth's surface, orbiting at an average speed of 7,706.6 m/s (27,743.8 km/h, 17,239.2 mph). The objects are still people, artifacts (spacesuits, spacecraft), and nature ("empty" space and planet Earth).

The main purpose of OOM is to facilitate the abstraction of the most relevant set of classes, objects, and relations (associations among classes and objects) that we are interested in. After all, we can not represent the whole world, nor do we want or need to. We call the abstracted set a **model** or abstracted system, whereas the system in the real world is called the **referent system**, **focal system**, or **target system**.[17]

In spite of their diversity along numerous dimensions, from a computational perspective the four human situations or "social worlds" in Fig. 2.3 share a **common ontology in terms of entities and relations.** The entities, relations, and environments in Fig. 2.3 can be summarized as in Table 2.3 in terms of a socio-environmental perspective (Sect. 1.5.2). Note that this table is based on the process of *abstraction*, discussed earlier in Sect. 2.7. Obviously, each of the four social situations contains (much!) more detail than is abstracted in the table. But what *really* matters is that three

[16] For now, we do not care about the various features of entities. We will explore that in the next section.

[17] The three terms are synonymous. Target system is more common in simulation research, as we will see later. All three terms mean the same: the system-of-interest in the real, empirical world.

Table 2.3 Human entities and selected associations in socio-technical systems. Environments are named, not detailed

"World"	Classes	Objects	Associations	Environments
Family	Wife, daughter, husband	Sally J. Smith, Mary Smith, John Q. Smith	Mother-child, child-father, mother-father	Photo studio
Disaster situation	Aid workers, victim	J. Eno, T. Abij, unknown	Co-workers, assisted	Rural road
Leadership summit	Political leaders, aides	R. Reagan, M. Gorbachov, others	Speaker-audience	Urban location
Orbiting astronauts	Astronauts	J. Uko, K. Oli	Collaboration	Low earth orbit

abstract categories alone (classes, objects, and associations) are universal across all social worlds. This fundamental ontology of CSS is consistent with classical social theory from ancient (Aristotle, Socrates, Plato) to modern (Parsons, Easton, Moore) and contemporary perspectives (including "constructivists").

The idea that objects of the same class share all common class-level features is called **inheritance** in object-oriented modeling. Thus, all wives are female, all husbands are male, all daughters have a mother, all disaster victims experience some level of stress, all political leaders govern through some base of support, all astronauts undergo many years of specialized training, and so on. Each object may also possess idiosyncratic features, but in order for it to belong to a class they must all share or "inherit" one or more features. Inheritance links classes and objects as a fundamental form of association.

Table 2.3 highlighted humans and associations among them, with only a coarse identification of the environments in which humans (social systems) are situated. A more complete abstraction, one based on the earlier socio-artifactual-natural perspective, is shown in Table 2.4. Now each type of "world" is decomposed (parsed) into three main components: the social (sub-)world is composed of the set of people and the set of social relations among them; the artificial component consists of built or engineered systems; and the natural component consists of the biophysical environment where the first two components (social and artificial) are embedded.

The buffering, adaptive, or interface character of artificial systems is highlighted by the ontological abstraction: Artifacts mediate between humans and nature, as the former adapt to the latter, following Simon's theory. In reference to Table 2.4 we see that:

1. The family is in a photographic studio room and only the white wall in the back is visible. Such an artificial room situation with highly controlled lighting conditions is necessary to ensure a high-quality portrait, as opposed to a more natural setting that cannot be as easily controlled.

Table 2.4 Social, artifactual, and natural components of coupled systems

"World"	Social	Artifactual	Natural
Family	Family members	White wall in back	Indoors
Disaster situation	Relief workers and victims	Road, stretcher	Countryside Indonesia
Leadership summit	Leaders, staff	Monuments, flags, public address systems	Outdoors in Berlin
Orbiting astronauts	Astronauts	International Space Station	Near Earth orbit

2. The disaster situation is mitigated by the use of artifacts such as a reopened road and medical equipment, in this case a special field stretcher. The uniforms of the relief workers are also functional artifacts, to protect the workers, to carry additional items, and to distinguish them from other members of the population.
3. The monuments, flags, and other stimulating symbols are used by leaders as artifacts to convey significance and power. Other artifacts consist of equipment for broadcasting and other communications infrastructure.
4. Astronauts use hugely complex artifacts such as spacesuits and the ISS to be able to function in the natural environment of orbital space, which would instantly kill them without such adaptive infrastructure.

In general: Artifact A is created for humans in social system S to perform in natural environment N. Symbolically, we might summarize this tripartite functional ontology as $A : S \leftrightarrows N$.

2.8.2 The Unified Modeling Language (UML)

Pictures and narratives, such as those used thus far, and other sources such as documents and data, can be informative, but they are usually insufficient for scientific purposes. They may tell us something about the focal world we are attempting to analyze, but are not very helpful in specifying the exact entities in terms of classes, objects, and their associations. Tables and other data can help, but can also be cumbersome for representing some features, such as complex relationships. The **Unified Modeling Language** (UML) is a standardized notational system for graphically representing complex systems consisting of classes, objects, associations among them, dynamic interactions, and other scientifically important features. Unlike most diagrams that appear in the social science literature, UML diagrams are rigorous, specialized graphics with specific scientific meaning—similar to a flowchart or a Gantt chart, where each symbol has specific meaning (semantics) and the arrangement of symbols is dictated by rules (syntax).

Although UML was created for representing systems of any kind, it is a valuable system for representing social systems and processes, given the lack of a standardized graphical notation system in the social sciences. There are different kinds of UML

diagrams, because complex systems require alternative, complementary ways of modeling them—as is the case for any multi-faceted problem. There are three most useful UML diagrams for modeling social systems and processes: **class diagrams**, **sequence diagrams**, and **state diagrams**.[18] The first is used for representing *statics* while the other two represent *dynamics*.

Why, when, and how? UML was invented in the 1990 s by a group of computer scientists and engineers that included James E. Rumbaugh, Grady Booch, and Ivar Jacobson. The Object Management Group (OMG) is the UML governance body that meets periodically to review and set standards. The original (and arguably still most prevalent) use of UML diagrams was to ensure that a diverse community of computer programmers and software engineers working on complicated code projects in large organizations (e.g., NASA, IBM, Boeing, Google) could work with a common understanding of a given programming project and collaborate effectively. Multidisciplinarity, personnel turnover, multi-lingual requirements, and other complicating factors conspire against producing and maintaining excellent and sustainable code. UML diagrams help a modeler and programmer by providing graphic representations, of key aspects of a complex computational project that are more inter-subjective than, for instance, narratives. The current UML standard is version 2.0, which is found at http://www.uml.org/.

2.8.2.1 Static Diagrams: UML Class Diagrams

A **class diagram** in UML is a graphic representation of the main entities and relations in a given social world or situation of interest. Figure 2.4 shows a simple class diagram of the general kind of social worlds we have been analyzing in the four instances discussed in this section (family, disaster, summit, astronauts): all four "worlds" consisted of a social system of some scale (small scale, as in the family, or large, as in the summit and space cases) and an environment of varying levels of complexity where the system was situated or embedded.

A UML class diagram consists of two main parts in terms of notation: *rectangles*, representing classes or objects, and *links* between them, representing associations, the labels and annotations of both are important. Rectangles are labeled by the name of each class or object (e.g., "world," "system," "environment"). Each association between entities (classes and objects) is also labeled by three elements: (1) an *arrowhead* symbol, (2) a descriptive *verb* describing the association (i.e., the role or function that one entity plays in terms of another), and (3) the *multiplicity* of the association, as defined below. In Fig. 2.4 the association between a system and its

[18] The "state diagram" is also known as a "state machine diagram." We will use the simpler term "state diagram," without loss of meaning.

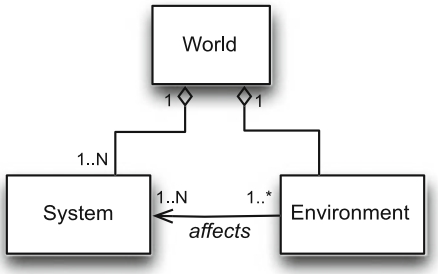

Fig. 2.4 UML class diagram of a basic world ontology consisting of a social system and its environment. Note that this graph is intended to represent the same as Fig. 2.2, but it conveys much more inforamtion

environment is denoted by the active but very general verb "affects;" the model does not include a reverse specification of anthropogenic effects (i.e., system feedback) on the environment, although in principle it could.

Dual graphic representations in UML. As with any graphic notation system in science, UML diagrams can be used to represent either the abstracted system (i.e., the model of reality) or a real-world system in greater detail than the abstracted model—for instance, as a reference of what is being omitted, if it is to added later. For example, there might be a UML diagram of a coalition being modeled, as well as a more detailed UML diagram of a real-world cabinet system with details on support from the multi-party system. The *most common use* of a UML diagram is for representing a model in terms of its abstracted components, not the real world. However, nothing prevents the use of a UML diagram for describing a real target system of interest if that is helpful. This may happen for a number of reasons, as when we wish to highlight the difference between a target system and a simulation model of such a system. The difference between a model diagram (abstract) and realistic diagram (empirical) would highlight all those elements omitted by the abstraction.

The concept of **multiplicity** is fundamental in computational modeling, although it is often neglected or left mostly undefined or it is implicit in more traditional social science theory and research. Multiplicity refers to the precise number of instances of a class or object. The notation in Table 2.5 is standard for specifying the multiplicity of entities in UML diagrams. We will be using this notation throughout this textbook, so it is important to master it, although it may be omitted in a summary diagram. Note that the symbol ".." (two periods) is used in computational UML notation to signify a range of values, rather than the more traditional mathematical notation "..." (called ellipsis).

For example, in Fig. 2.4 there is one World entity (a class) consisting of one or more (up to N) System entities and one or many (up to an indefinite number, represented by the asterisk symbol "*") of Environment entities affecting the System.

Table 2.5 Multiplicity values in UML class diagrams

Value	Meaning	Example	Mathematical notation
0..1	A range between no instances and one, meaning none or just one object	Number of prime ministers in a government	[0, 1]
1	One and only one instance	Each system has one environment	1
0..* or *	Range between 0 and unspecified many	Number of children in a family	$[0, +\infty]$
1..*	Range between 1 and unspecified many	Number of cities in a country	$[1, +\infty]$
0..*N* or *N*	Range between 0 and exactly *N*	Number of midlevel managers in a firm	[0, *N*]
1..*N*	Range between 1 and exactly *N*	Number of provinces in a polity	[1, *N*]

The multiplicity of World is implicitly one in this case, so the value of 1 is normally omitted because it is redundant (unnecessary). Later we will examine other examples.

Social Science dedicates a great deal of effort attempting to describe and understand *social relations* among various *entities* (actors, their beliefs, institutions, and their environments, among others). In UML the type of association that is assumed to exist between entities is denoted by the special form of the link's **arrowhead**. (As we will see, this is not arbitrary or esthetic, as in most traditional social diagrams! The form of an arrowhead has precise meaning in UML.) In the case of Fig. 2.4 the association between World and Environment is one of aggregation (hence the white diamond-head, as explained below), because the World class is being modeled or specified as consisting of two component classes: the social System of interest and the Environment in which such a system is situated, with the latter "affecting" the former. (For now we need not worry about the meaning of the term "affects"; the common meaning will suffice.)

The four most common **types of association** are called "inheritance," "aggregation," "composition," and "generic," which are distinct types of social relations denoted by the symbols illustrated in Fig. 2.5.

Earlier we encountered **inheritance** (empty arrowhead symbol) when discussing the association between classes and objects, in the sense that an object is an instance of a class, such that all objects belonging to the same class are said to share or "inherit" a common set of characteristics. The inheritance association is also called the "is a" relation. It is denoted by an arrow with a blank arrowhead. In Fig. 2.4 we saw an example of aggregation and a generic association relationship ("affects").

The following are examples of the inheritance association (one from each of the Big Five social science disciplines):

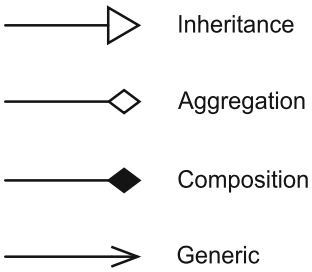

Fig. 2.5 Associations among classes or objects are drawn in UML using arrows with different arrowheads that denote different types of relations (e.g., social relations, socio-environmental interactions, or others). Unlike the informal and widespread use of arrows in many social science illustrations, the notation for social relations modeled with a class diagram is formal and strictly defined, making meanings inter-subjective and reliable from a conceptual and terminological perspective. Examples of each type of social relation are provided in the main text

- **Politics: Political regimes**. Consider the class "Political Regimes." It contains classes such as "democracies" and "autocracies," both of which represent particular forms of political regimes but also share in common many features having to do with the relationship between society and government. Thus, both democracies and autocracies are said to inherit the properties of the class "Political Regimes."
- **Anthropology: Social complexity**. The classes "band," "tribe," "chiefdom," and "state," from anthropological archaeology represent ordinal forms of social complexity. All these forms inherit the features of a broader class that may be called "Polity." All polities—and therefore all chiefdoms, states, and empires—include a "society" (population, community) and a "system of government." In turn, all systems of government share some common features, such as constitutional regime (defining the society–government relationship), bureaucratic structure, support mechanism, public finance (resource base), policy-making process, and other constituent or defining features.
- **Psychology: Cognitive balancing (Abelson 1959)**. The objects "Differentiation," "Bolstering," "Denial," and "Transcendence," are instances of the broader class of "Cognitive Balancing Mechanisms." All four mechanisms serve the purpose (have the function) of resolving or mitigating cognitive inconsistencies that arise in human complex belief systems.
- **Economics: Goods**. The classes "commodity goods" and "luxury goods" inherit the features of the broader class of "private goods." An instance of the private good-class, such as a 2012 Ferrari racing car, is an object, due to its concretely empirical specificity. All private goods share some common features, such as, for example, quantity produced, price, provenance, production method, and useful life, among others. In turn, "private goods" belong to the superclass of "economic goods," which also comprises "public goods," such as "clean air" and "public security."

- **Sociology: Organizations**. The class "organizations" comprises "private organizations" and "public organizations." Both types (whether private or public) inherit all the features of the former, such as mission, size, structural features, age, and domain of activity, among others. In addition to class-level features, both private and public organizations have other features, such as membership characteristics for private organizations or public finance for public organizations.
- **Aristotle's Classification of Governments**. Aristotle [384–322 B.C.] was the first comparative social scientist of whom we have a surviving record. The Aristotelian classification of governments distinguishes between normal and degenerate forms of government. The three normal forms are monarchies, aristocracies, and democracies, while the degenerate forms are tyrannies, oligarchies, and ochlocracies, respectively. Thus, an abusive monarchic ruler yields a tyranny; degenerative rule by an elite produces an oligarchy; and extreme democracy yields an ochlocracy (literally, "mob rule"). Representative government (e.g., as in a parliamentary system) is a regime that attempts to implement democracy to avoid ochlocracy (as occurred during the Reign of Terror, A.D. 1793–1794, in France). All six types inherit all the features of the class Government, with each type having additional characteristics.

These examples illustrate the inheritance association, which is represented by the empty arrowhead in Fig. 2.5. A UML class diagram of each example would include the main entities and the inheritance association link annotated with the multiplicity of each entity (class or object).

The next two types of association—called "aggregation" and "composition"—apply to *compound social entities.*[19] Committees, belief systems, organizations, and whole polities, economies, and societies are prominent examples of compound social entities. CSS examines these compound entities by distinguishing between those that are structured by aggregation versus those that are structured by composition.

The second type of association is called **aggregation** (empty diamond arrowhead), which has the conceptual meaning of "consists of" in natural language. Aggregation is also called the "has a" relation. This is a loose type of collection, which in some cases may just be ad hoc (as opposed to the stronger form of membership rule implied by the composition association, discussed below). The following are examples of aggregation in compound social entities:

- A human belief system consists of concepts (represented as nodes) and associations among them (valued links).
- A family is a social aggregate consisting of parents and children.
- A society is comprised of individuals that share a set of commonly held attributes.

[19] A compound social entity C may be thought of in a similar way as a compound event in probability theory. Accordingly, C consists of several smaller parts or subsystems "smaller" than C, similar to the way in which a compound event is defined as a function of its conjunctive elementary events (sample points).

- An economy is composed of producers, consumers, and lenders.
- A coupled socio-techno-natural system consists of interacting social, artifactual, and biophysical components in interaction with one another.

A key feature of aggregation is that the members can survive without the aggregate, as in the examples above. Parents and children do not cease to be such when a divorce occurs. Concepts that are part of a belief system can endure after a belief system is no longer accepted. Producers, consumers, and lenders can endure even after an economy disintegrates.

Aggregation is denoted by the empty diamond arrowhead in a UML class diagram (Fig. 2.5), with the arrowhead pointing to the higher order class (superclass).

The third type of association is called **composition** (solid diamond head symbol), which is a stronger form of aggregation. Composition is used instead of aggregation when member classes have a constituent relationship with respect to the superclass; i.e., when the set of member classes cannot exist without the superclass. Accordingly, composition can also be called the "is constituted by" relation, similar to "is a" and "has a" for inheritance and aggregation. Under composition the superclass compound is said to "own" the member classes of the compound entity, in the sense that if the superclass dies—or somehow is destroyed—so do the classes under it.

The following are examples of association by composition in compound social entities:

- A bureaucracy is an organization composed of bureaus or administrative units. The units exist by virtue of their contribution to the overall organization.
- The provinces, counties, and other administrative units of a country are associated to the larger country by composition.
- As a compound social entity or "body," a given committee with members playing various functional roles, as in the case of a ministerial cabinet, is linked to its members by composition. A cabinet minister does not exist without there being a cabinet. This is normally defined by a constitution.
- The institutions of international governmental organizations, such as the General Assembly and the Security Council of the United Nations Organization, or the Commission and Parliament of the European Union, are associated to the organization by composition, not just aggregation.

Many aggregate entities of common interest in social science are in fact compositions, not mere aggregations, because component classes are defined as a function of some superclass (compound social entity), such that parts are meaningless without the whole. When social scientists speak of "the importance of context," they often have in mind the composition association of compound social entities, rather than mere aggregation. Context can matter, precisely because some constituent social entities are fundamentally (constitutionally) dependent on larger compound entities only through association by composition.

The key *difference between aggregation and composition* is conceptually subtle, significant (theoretically and empirically), and unfortunately quite often left implicit

in social science theory and research on compound entities of all kinds, which consist of actors, events, systems, and processes. The multiplicity of aggregation can assume any value (i.e., the natural numbers or positive integers $1..N$), whereas the multiplicity of composition is zero or one on the compound, higher order class (the superclass). Testing this idea with examples is good for understanding the difference. Whether associations or relationships in a compound social entity are either by aggregation or by composition is something that should be decided and denoted accordingly in a well-specified UML class diagrams, or the unresolved ambiguity can result in confusion leading to modeling errors in implementation. Formally, composition spans a *tree*, whereas aggregation forms a *net* (Eriksson et al. 2004: 113).

Some compound social systems have **hybrid associations**, as shown in Fig. 2.6. An example is a polity P, which consists of a given society S and a system of government G for addressing issues I that affect members of S. Whereas G is "owned" by P (hence composition specifies the polity–government association), in the sense that it makes no sense to think of a governmental system except within the context of *some* polity, society S is an aggregate that has autonomy regardless of whether or not P exists (aggregation specifies the polity-society association), since S is an association among people in terms of identity and other features (whether members of some elite or the mass public). The class of issues I is also related to P and G by association, because issues affecting S can persist regardless of P and G. The compound social system P is therefore a hybrid of compositions (in G) and associations (in S and I).

Inheritance, aggregation, and composition have their own special ad hoc symbols because they are so common. Finally, a fourth type of association is called **generic** (plain arrow symbol), which is a category intended to represent any association in terms of a verb connecting any two entities. Generic association is symbolized by a simple arrowhead and the verb that best describes the association. For example, the association between Environment and System in Fig. 2.4 is represented by the simple arrow from E to S and the verb "affects" describing the association. Similarly, in Fig. 2.6 there are three generic associations represented: Public Issues affect Society, causing stress; Society places demands on Government to deal with issues; and Government deals with issues by issuing (i.e., formulating and implementing) policies that mitigate stress on Society.

2.8.2.2 Dynamic Diagrams: UML Sequence and State Diagrams

In addition to the static diagrams introduced so far, the Unified Modeling Language also provides standardized graphics for representing dynamical aspects of social entities; i.e., social *processes*. Two of the most common dynamic diagrams are those called "sequence diagram" and the "state machine diagram." Other dynamic diagrams include "activity diagrams" and "communications diagrams" (Eriksson et al. 2004).

A **sequence diagram** portrays dynamic interactions that take place in a social process among entity components. Figure 2.7 shows a UML sequence diagram for the standard model of a polity represented earlier in Fig. 2.6. There are three main components in a sequence diagram: (1) a set of separate vertical "lanes," each representing

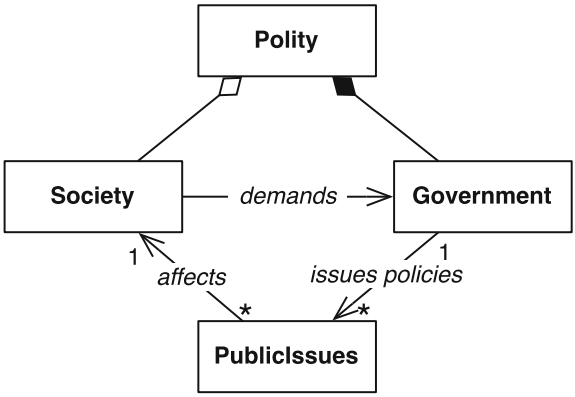

Fig. 2.6 UML class diagram of the standard model of a polity in political science. The diagram consists of four entities and three types of associations that denote different kinds of social relations, as explained in the main text. Diagrams such as these, and subsequent versions with more details, are valuable for communicating between social science modelers and computer programmers in charge of code implementation. Adapted from Cioffi-Revilla (2008)

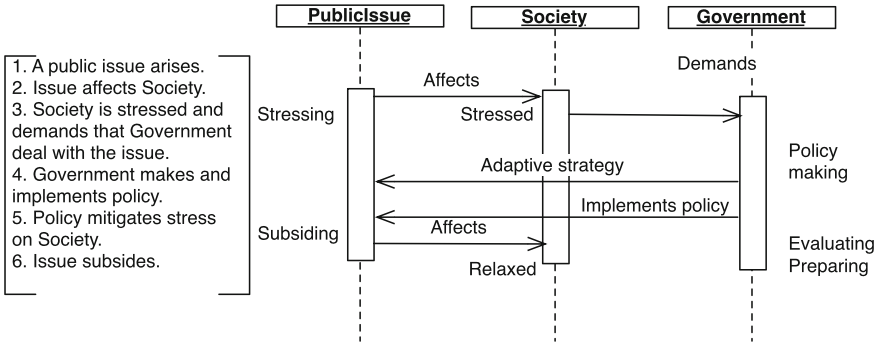

Fig. 2.7 UML sequence diagram of basic dynamic processes in a simple polity

the main interacting entities in the compound superclass (e.g., in this case PublicIssues, Society, and Government); (2) arrows indicating various activities among entities; and (3) a summary natural language chronology of main events of interest (left), which should say in plain English what the sequence diagram is intended to graphically represent.

Several features of the UML sequence diagram are noteworthy from the example in Fig. 2.7:

1. UML symbolic notation is standardized and systematically developed, not arbitrary. This enables researchers to communicate using a common set of universal symbols that have been agreed upon. By contrast, most traditional social science

diagrams are drawn using ad hoc symbols often invented by an author and used by no one else.
2. A diagram like this cannot be drawn without fairly precise understanding of the social process being represented. At a minimum, a researcher needs to stipulate or hypothesize some parts of the process where theoretical explanation or empirical descriptions are missing in the basic relevant science.
3. The basic space-time ontology of the social process is discretized—in terms of classes and objects (social space) and events (time)—not continuous. This enables the specification of precise interactions and their sequence within an overall framework.
4. The diagram is ordered by time, flowing from top to bottom, as in an historical timetable or a flowchart. Thus, addition or deletion of events requires shifting down or shrinking everything downstream.[20]
5. The information dimensionality of the basic notation is simple, so much room is available for increasing the information content of the diagram by use of color, tones, patterns, and additional shapes.
6. A potentially significant drawback of the sequence diagram is its tendency to become too cluttered when more than a few objects or classes ("lanes") must be represented. Having to represent a process with many objects or classes almost guarantees an unreadable or messy diagram, so the sequence diagram does not scale well with respect to the cardinality of the ontology being modeled.

Another type of dynamic UML diagram is the **state diagram**, which represents transitions between macroscopic states of a system during its typical operating or life cycle. Figure 2.8 shows the state diagram for a polity, based on the standard model that we introduced previously. This diagram consists of three components: (1) a set of start-end states, represented by large black dots; (2) a set of labeled possible contingencies represented as transitions between states; and (3) a set of labeled states representing the condition of the system as a result of each transition.

A UML state diagram depicts various states of the system and possible transitions between states. In this case the polity starts out in an unstressed state, which we may call a **ground state**, since Society S is not initially affected by any issues—everything is fine. When S is in an unstressed state, two things can happen: either some issue arises or it does not. If no issue arises, then the state of S remains unstressed (shown by the loop arrow labeled "No issue"). However, if an issue arises, then the state of S transitions to being stressed (by the issue). When S is stressed, two things can happen: either Government G pays attention and produces policy, or G fails and policy is not produced (the "No policy loop arrow"). When G produces policy, two things can happen: either policy fails, or it works. If policy works, then S is again unstressed. The process continues or ends after policy is produced.

[20] By contrast, archaeologists draw timelines from bottom to top, consistent with stratigraphic analysis, such that the oldest date is at the base.

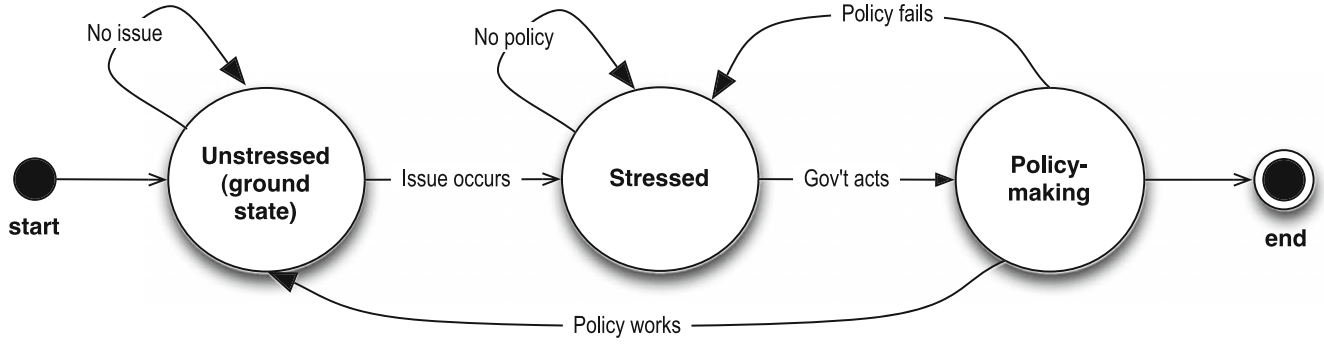

Fig. 2.8 UML state (or "state machine") diagram of macro system-level dynamics in a simple polity consisting of a Society stressed by issues and a Government that formulates policies to address public issues and lower or eliminate stress. A state diagram provides a more dynamic model of a polity than a class diagram, but entities (classes, objects) are not represented. *Source*: This and other UML diagrams of a polity are adapted from Cioffi-Revilla (2008)

This single-issue account of the standard polity and policy process is intentionally simplified (abstracted) for illustrative purposes. One simplifying assumption is that public issues affect S without any anticipation on behalf of G, which is sometimes (not always!) unrealistic. For example, in the case of many domestic policies, G often prepares by producing anticipatory mitigating policies. Another simplifying assumption is the direct, unmediated pressure of S on G, without intermediaries. In fact, interest groups and other intermediary groups (e.g., lobbies, unions) act between S and G, producing more transitions and additional intermediary states before policies are produced. Finally, another assumption in Fig. 2.8 is that the policy-making process is finite, rather than going on forever.

The UML state diagram is characterized by a set of features, as seen from Fig. 2.8:

1. The diagram is read from left to right, with various possible transitions and loops as the state of the system evolves to the end of each cycle.
2. The diagram is reminiscent of a Markov model representing the states of a system and possible transitions, minus the start and end states. Unlike a Markov model, however, transition probabilities are not generally represented.
3. The diagram is also reminiscent of a flowchart, but with exclusive emphasis on the state or condition of the system.
4. Classes or objects (entities) do not appear in a state diagram. Instead, this type of diagram focuses on the state or condition of the system, given the possible interactions among entities.
5. Each transition is specified by asking "what can happen next?" given some state.
6. The state diagram is formally a graph. Specifically, it is a directed graph. It can also be weighted, if transition probabilities (or other measures associated with the links between states) are known.

State diagrams such as the one in Fig. 2.8 are sometimes difficult to specify in a way that is sufficiently complete or precise, in part because the detailed dynamics of a social system and process may not be clearly understood. In that case, resort to other sources such as narratives or other diagrams may prove useful. For example, a classical flowchart may be helpful for uncovering the information needed for a state diagram.

When attempting to specify a detailed state diagram, it is always good practice to begin with a simple version with the fewest possible number of states and transitions.

Other UML diagrams include activity diagrams and use case diagrams. We will use UML diagrams throughout this book for two main purposes: increasing scientific clarity and enabling computational specificity. Both uses are new in social science.

2.8.3 Attributes

Now that we have covered the basics of classes, objects, and associations, we must take a closer look at them by focusing on two key computational aspects: their

defining attributes and operations. We will do this assisted by some further UML notation created precisely for dealing with attributes and their operations, as in Fig. 2.9.

We will approach the parts of Fig. 2.9 in sequence, with the last part (e) being the most complete in terms of specification. To begin, note that the following notational details in Fig. 2.9 are standard and important, not arbitrary:

1. Class and object names are written in the center of the first compartment of each diagram, with initial capital letter (as in a proper noun), preferably in boldface (e.g., Class, Object, Polity, Switzerland).
2. The name of an object is underlined (Object, Province, County, City).
3. Attributes are written with the first letter in lowercase, followed by additional words without spacing (e.g., classAttribute1, classAttribute2, popSize, capitalCity, numberOfiPhones, inflationRate), always left-justified.
4. The data type of each attribute is written after the attributeName.
5. Operations are written in a similar way, followed by left and right parentheses.
6. The so-called "visibility" or "accessibility" of attributes and operations is denoted by plus and minus signs, representing their public or private status, respectively, as explained further below.

A feature, variable, or parameter that characterizes a social entity is called an **attribute** in CSS. Attributes are familiar concepts in Social Science, often under the name of "variables" or "parameters." The following are some illustrative attributes, based on earlier examples in this chapter. In the case of a coupled socio-techno-natural system, we may model the natural environment as consisting of ecosystems with biophysical attributes such as biomass distribution, climate variables, topography, and others. Similarly, social attributes are often used to characterize various actors and groups abstracted in a model, such as economic, political, and social variables. In the case of a polity, commonly specified attributes include population size, size of its economy, territorial extent, cultural indicators, military capabilities, and other numerous features. Each social object or class is always defined in terms of some set of attributes.

In Figs. 2.9(b)–(e) we saw how attributes are annotated in the second compartment of a UML class diagram. Figure 2.10 shows how this is done in a more complete UML class diagram, as part of each class or object, in this case using the Polity model discussed earlier. In this case we have chosen to abstract the following class attributes: the name, continent in which it is located, territorial size, and name of the polity's capital city; the population size and amount of resources of the society of that polity; the government's gross capacity and net capacity for policy-making and implementation; and the type, salience, cost, and onset and resolution dates of public issues that arise in the polity.

Figures 2.9(d) and (e) also show the **visibility** or **accessibility** of each attribute by using plus and minus signs. This is a feature of attributes and operations that defines the status of information in relation to other classes. Specifically:

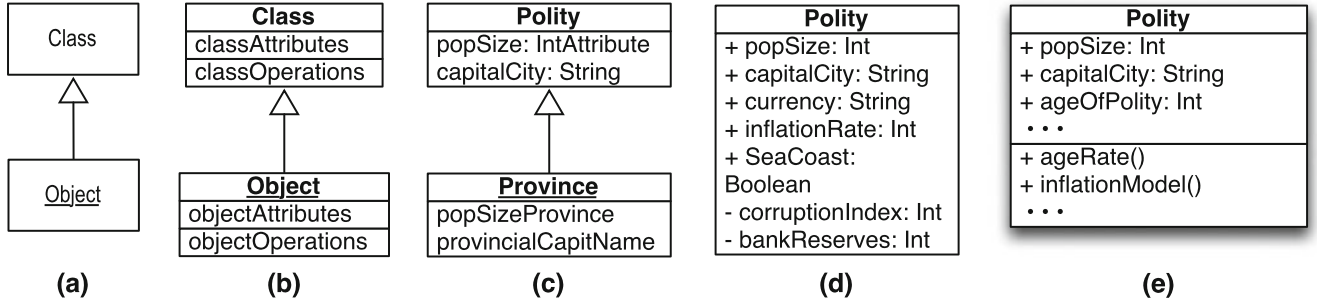

Fig. 2.9 UML class and object diagrams with various specifications of attributes and operations: (**a**) Class and object associated by inheritance, without specific attributes or operations, as in earlier class diagrams. (**b**) Class and object notation containing encapsulated attributes and operations shown by convention in the second and third compartments, respectively. (**c**) Example of class and object with some specific attributes. (**d**) Visibility of attributes denoted by public (*plus sign*) and private (*minus*) attribute notation. (**e**) Complete specification of a class with encapsulated attributes, operations, and visibilities

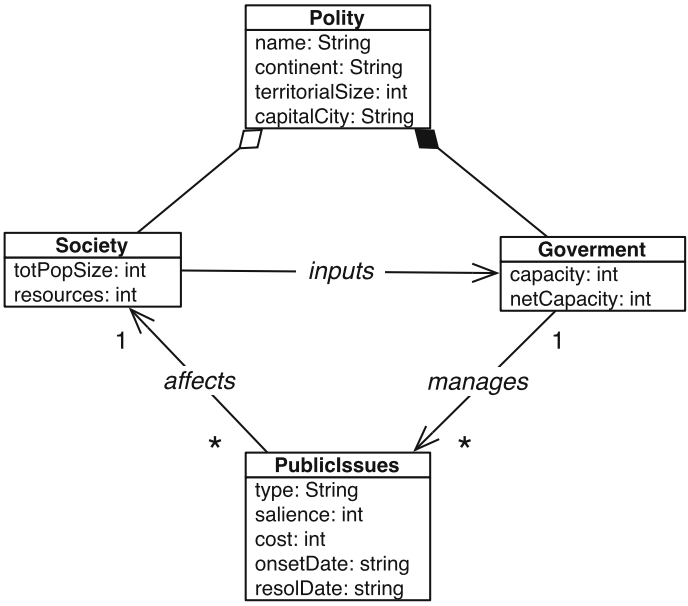

Fig. 2.10 UML class diagram of the standard polity model, with specified attributes (variables). Note that each attribute is denoted by a uniquely designated name and corresponding data type

1. An attribute is **private** when it can be accessed only from its own class, denoted by the minus sign −.
2. An attribute is **public** when it can be used and viewed by any other class, denoted by the plus sign +.
3. An attribute is **protected** when it can be accessed only by its class or subclasses, denoted by the pound sign #.

The attribute of an object is called a **object variable**, to distinguish it from the class-level feature. This nomenclature is consistent with the earlier idea that an object belongs to a class. Similarly, the attribute of a class is also called a **class variable**.

2.8.4 Operations

We saw earlier (Fig. 2.9(b)) how attributes and operations define a class. An **operation** changes the value of one or more attributes, and consequently the state of objects and classes. At the object level, an operation is called a **method.** Operations and methods are implemented by functions in Python. A common example of an operation is a function that would specify how the population of a polity changes each year. Operations specify dynamics, whereas attributes define statics; both determine the state of classes and objects.

Figure 2.11 shows how operations are added to the third compartment of a classes's box to complete the model in greater detail, extending the earlier model in Fig. 2.10. This is the same familiar model of Polity, only now we have added some operations that tell us how attributes are supposed to change in each class. For example, in the Polity class, the attribute (or class variable in this case) called ageOfPolity will change as specified by a function called agingRate(), which is defined in the third compartment of Polity. This is presumably a simple function that returns an annual increment of 1.0. Similarly, the attribute called corruptionRate in the Government class is driven by corruptionChange(), which is a more complicated operation defined in the third compartment of Government. For example, corruptionChange() might be specified or modeled as a function of other attributes, such as levels of foreign investment, literacy, rule of law, and other variables (i.e., the known determinants or drivers of governmental corruption reported in the empirical literature) that are located in the same or other classes.

In any social system some associations are more important than others. For example, note the "manages" association between Government and PublicIssues in Fig. 2.11(left). This is a very significant relation between two major entities of a polity, which in this case abstracts the notion of a policy. It is through policies that governments address public issues. Thus, the seemingly simple association between Government and PublicIssues should be elevated to the higher status of having a class by itself, as **association class** named Policy. As shown in Fig. 2.11(right), an association class is denoted by the same class notation, joined to the association link by a dashed link. To decide whether a given association warrants the status of being modeled as an association class, rather than a mere association, the following heuristic questions are helpful:

1. Does the association in question have significant attributes that can be specified?
2. If so, what are they?
3. Moreover, do such attributes have operations that can be similarly specified?

If the answer is yes to questions 1 and 3, then the association in question is a candidate for promotion to association class status. For example, in the previous case it is certainly, true that a policy has attributes, such as type (economic, social, political, environmental, or other), effectiveness (degree to which it is likely to solve the issue), efficiency (cost/benefit), and other features. However, whether promotion to the status of association class, rather than mere association, is warranted is a different question, which depends on research questions and not just our ability to identify relevant attributes.

Both attributes and operations are said to be "encapsulated" within a class or object. "This process of packaging some data along with the set of operations that can be performed on the data is called **encapsulation**" (Zelle 2010: 418). Encapsulation is a powerful, defining feature of all OOM and OOP. In UML modeling terms this means that all attributes and operations must always appear contained within the second or third compartments, respectively, of *some* class or object entity—never unassociated, by themselves. More importantly in OOP, encapsulation means that classes and

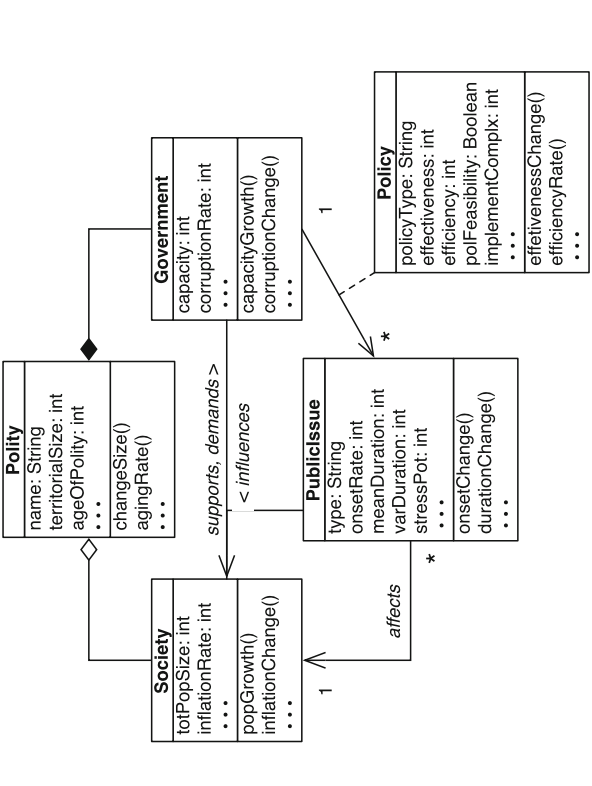

Fig. 2.11 UML class diagrams of a polity with class attributes and operations. The model on the *left* shows operations in the third vertical compartment of each class. The model on the *right* makes explicit the "manages" association between Government and PublicIssues, elevating the association to the higher status of a class by itself, named Policy

objects can interact without having to access inner components or computations that can be hidden within entities. All fully OOP languages implement encapsulation, whereas most procedural languages do not. Hence, implementing social models comprised of entities that encapsulate attributes is best accomplished in an OOP language so as not to risk breaking encapsulation.

Python implements encapsulation as a convention, as part of proper programming style, and is not an absolute requirement of the language. By contrast, encapsulation is a required feature of abstraction in Java.

Encapsulation implies that variables and methods/operations are always defined as belonging to some object or class, never by themselves. Other common language phrases used to signify encapsulation are "in the context of," "with respect to," and "in relation to," among others. For example, the context for the variable inflation is an economy, the context for corruption is government (or business), voting behavior is associated with a polity, and so forth. From an OO perspective in Computational Social Science, variables or parameters make no sense by themselves, in isolation. Hence, they are always encapsulated within some class or object. *Understanding this idea also provides a powerful principle for turning a variable-based model into a potentially more powerful object-based model.*

2.9 Data Structures

Classes and objects represent one form of data that encapsulates a set of attributes/variables and operations/methods for changing the value of attributes/variables. However, data come in many forms—not a surprise in social science! We have already seen various value types for variables, such as **integer**, **string**, and **boolean**. The term **data structures** refers to the various ways in which data are organized for purposes of computation. Sometimes the same information is organized in different ways, so it will be structured differently, depending on computational need. When it comes to data structures, remember the famous design principle from architecture: "Form follows function" (Louis Sullivan, American architect, 1896).

The following are the most common data structures, listed in order of generalization[21]:

Tuple: A tuple is similar to a record structure, the main difference being that individual records need not be arranged as in the 2-dimensional structure typical of a spreadsheet. Elements of a tuple must all have the same type. Examples: calendar dates expressed by year, month, and day; N-dimensional Cartesian (or other coordinate system) n-tuple of coordinate values for a point $(x_1, x_2, x_3, \ldots, x_N)$;

[21] There are as many kinds of data structures as there are ways in which information can be organized. The US National Institute of Standards and Technology (NIST) provides a comprehensive, encyclopedic online survey (Black 2004).

payoff values (u, v) in a 2×2 normal form game Γ, where u and v are the payoffs for each player. The elements of a tuple are ordered.

Array: An array has elements of the same type accessible by some index. Examples: all vectors and matrices; input–output table of sectors in an economy; adjacency matrix of a network. A vector is a one-dimensional array, whereas a matrix is a 2-dimensional array. A vector is a datum with both scalar value and direction (whereas a scalar lacks direction). A "data cube" is a 3-dim array (e.g., countries $\times$ attributes $\times$ years). A *sparse array* is one where many entries are zero or missing, which may be better structured as a list.

List, or **sequence:** A list is a mutable tuple of variable length, with the first element called the *head* or *header*, and the ones that follow are called the *tail*. Examples: Cities ranked by population size, the head being the largest; conflicts or disasters ordered by severity, the head being the worst case; network nodes arranged by the number of links with other nodes (called *degree*), the head being the node with highest degree.

Queue: A list of items where the head is accessed first. Examples: legislative bills in a calendar for voting; items on a formal agenda; refugees arriving at a camp site; military units being deployed. Operations defined on a queue include addition (new value is added to the tail), deletion (from the head), as well as others. A queue is also called a FIFO (first-in-first-out) list, or pushup list. Queues are also a significant social process, so half of Chap. 9 is dedicated to them.

Stack: A stack is a data structure consisting of an ordered list of data such that the datum inserted last gets drawn first. Examples: location visited most recently; the most recent acquaintance; the most recent course taken by a student or taught by an instructor, from among a complete list of courses taken or taught, respectively. Chronological order of entry into the data structure is a key idea in a stack.

Bag: A bag is a set of values that can contain duplicates. Examples: the set of all countries that have experienced civil war during the past τ years; a list of individuals who have voted in the past N elections; the set of terrorist organizations that have launched suicidal bombing attacks since 9/11. The term *multi-set* is synonymous with bag.

Set: A collection of elements in no particular order with each element occurring only once. Examples: The set of cities in a given country; coalition members; candidates in an election; budget priorities; major powers in the international system (*polarity*); nodes in a network; legislative bill proposals in the "hopper." This is a general and powerful mathematical concept with broad applicability across the social sciences.

Hash table: Also known as a **dictionary**, a hash table is a data structure in which values and keys are assigned by a function, called the hash function. A hash table is an array of 2-tuples consisting of values and associated keys, such that there is a one-to-one mapping between values and keys (binary relation). The list of values is also called a hash table. Examples: a telephone directory; a list of voters and their voting precincts; administrative units (counties, provinces, states, countries) and abbreviations or codes; items and barcodes; geographic gazetteers; organizational charts; course catalogues. A hash table provides a fast way to lookup data.

Tree: A tree is a data structure consisting of a *root* element with *subtrees* branching out to terminal *nodes* called *leaves*. Nodes located between the root and leaves (i.e., "crotches," in common language) are called *internal nodes*. A taxonomy has the structure of a tree. Examples: classification of social entities; extensive form games; tree of phone calls for emergencies; hierarchal organization in business and public administration; star network; population settlement pattern (capital [root], provincial centers, town, villages, hamlets [leaves]). Tree-like data structures are ubiquitous in social systems and processes, but they are rarely analyzed as such.

Graph: A graph is a generalization or extension of a tree, in which nodes and links (also called arcs or edges) can be arranged in any way, as we discuss in detail in Chap. 4.

Note that data structures do not contain any code; they just contain data organized in various ways.

A **record** is like a composite data type rather than a true data structure, in a strict sense. It consists of information fields or members comprising a set. Examples: a person with contact information (address, telephone, email, Skype address); a polity profile (country name, capital city, total population, and other attributes); a bibliographic entry (author, title, place, and date of publication); events data (actor, target, date, descriptive verb, other event attributes). A spreadsheet entry is often like a set of records, with columns representing various fields, as is a common in social science datasets.

All of these data structures can be used in the Python and Java programming languages (and many more). Python can handle lists of many types, including stacks, queues, matrices (a list of lists), tuples, and sets, among others. A set of operations (functions and methods) is defined for various data structures in each programming language.

2.10 Modules and Modularization

In all but the simplest cases, a computer program usually requires "parsing" into main components and subcomponents. This is because writing a long, "monolithic" program is impractical as soon as the program requires more than just a few lines of code (LOC). **Modularization** is not just a programming style; it matters greatly in terms of overall program performance.

One way to think of modularity is in terms of performance: how should a given computer program be written in order to maximize its speed? Intuitively, there may be many ways in which a computer program could be modularized. For example, computation and visualization could be separated; but so could various stages of execution, in sequential fashion, as derived from a flowchart. The way in which a given program should be modularized into parts is not necessarily obvious. David Parnas (1972), a famous computer scientist, introduced the influential **Principle of Decomposition by Information Hiding**. Given a program P, the **Parnas Principle** states that P should be structured in nearly decomposable modules, such that each module

encapsulates a nearly self-contained (encapsulated) cluster of instructions *and* the interface between modules is such that it minimizes "communication overhead."

> Direct quote: "… one begins with a list of difficult design decisions or design decisions which are likely to change. Each module is then designed to hide such a decision from the others. Since, in most cases, design decisions transcend time of execution, modules will not correspond to steps in the processing. To achieve an efficient implementation we must abandon the assumption that a module is one or more subroutines, and instead allow subroutines and programs to be assembled collections of code from various modules" (Parnas 1972: Conclusions).

The following are significant advantages of Parnas-modularity:

- Modules are easier to understand.
- Independent programmers can work on different modules.
- The program can be more easily changed.
- Sensitive information may be more easily protected.

The overall structure of a modular program is that of a network composed of any number of communicating clusters, as in a cellular network (similar to the Horton or Tutte graphs), such that most of the communication takes place within clusters and minimal communication across them.[22]

2.11 Computability and Complexity

Consider the following questions:

1. A leader needs to form a coalition in order to ensure security against a powerful adversary. Given a set of potential allies, what are the possible combinations that might produce successful, winning coalitions?
2. A person involved in a disaster faces a set of competing priorities (safety, family, shelter, neighbors, supplies), which can induce severe frustration, compounded by fear and uncertainty. Which course of action is best, or at least satisfactory?
3. A country affected by climate change must choose from among a set of competing policies, finite resources, and imperfect information. How can policy analysts arrive at defensible recommendations for policy-makers?

[22] Interestingly, the structure of a terrorist organization is also that of a cellular network, as we shall see later on. What does Parnas' Principle suggest in the context of terrorist organizations, terrorism in general, or counterterrorism policy analysis? Which of those insights derived from a CSS approach are also available from traditional social science perspectives?.

Questions such as these require complex social computations, not just in terms of crude costs and benefits, but also in probabilistic assessments, alternative combinatorial arrangements, fitness assessments with respect to known empirical patterns, and other computational features. The necessary science (social or natural) may also be incomplete, so allowance must be made for deep uncertainty—not just risk with known probability distributions. And yet, as scientists we wish to obtain computable answers to questions such as the three listed above.

Computation is feasible over an immense and expanding problem-space, but it is not universal. **Computability** has to do with the effective ability to compute an algorithm, given some functions/methods/operations and data. More precisely, effective computability requires two conditions:

1. The algorithm must consist of a finite and relatively simple set of functions arranged in some proper way; and
2. Each function must execute in finite time.

Given these two requirements, a problem is not computable if either condition is not met.

Informally, **computational complexity** refers to the degree of difficulty involved in solving a computational problem of size N, in terms of space or time resources required. Formally, let $T(n)$ and $M(n)$ denote separate measures of computational complexity with respect to time and memory, respectively, where $n \in N$ denotes the size of the problem. For example, N may refer to the number of possible alliances in Problem 1, the number of alternatives in Problem 2, or similar features that measure size. In general, computational complexity has to do with how computability scales with respect to a given size. A problem that scales as a **polynomial** is said to be computationally tractable, whereas one that scales **exponentially** is not. A problem is said to be **intractable** when it cannot be solved in polynomial time.

2.12 Algorithms

So far we have used the term **algorithm** more or less as synonymous with "code" or "program." Stated more precisely, a program is a *formalization* of an algorithm, similar to the way in which an equation specifies a function. According to the *Dictionary of Algorithms and Data Structures* published by the National Institute of Standards and Technology (NIST), an algorithm is defined as follows:

Definition 2.1 (*Algorithm;* Black 2007) An algorithm is a computable set of steps to achieve a desired result.

In this chapter we have already seen several initial examples of algorithms, ranging from chaos to elections. We should now be able to have a better appreciation of

how the concept of algorithm relates to the *Computational Paradigm* of CSS discussed earlier in Chap. 1. Such a perspective views a social system (on any scale) as an information-processing entity; i.e., as algorithmically structured. How is this possible? The information processed by social systems is structured in many ways, as discussed in Sect. 2.9. Information can be in the form of records, arrays, trees, or other data structures. Algorithms involve search, comparisons, maximization, sorting, and other fundamental and compound forms of processing information.

Algorithms are implemented in social systems using many different real-world processes. The following are some examples of significant social processes viewed in terms of "desired results" and "sets of computational steps," consistent with Definition 2.1:

Cognitive balancing (Psychology): As humans, we maintain overall cognitive coherence in our belief systems *by* adjusting beliefs through Abelsonian mechanisms (discussed in Chap. 4).

Census (Sociology): Every complex society (chiefdoms, states, empires) counts the size of its population *by* conducting surveys and other procedures for gathering data on individuals and households.

Economic transaction (Economics): Economic agents conduct a sale *by* exchanging information and agreeing on terms.

Election (Politics): A democratic polity determines a leader *by* counting votes according to some set of rules.

Legislate (Politics): Policymakers enact laws *by* aggregating preferences following constitutionally established procedures.

CSS requires us to examine social processes from an algorithmic perspective and social systems as supported by functionally significant algorithms, following the Computational Paradigm. Obviously, each of these complex processes has far more real-world complexity than can be reasonably stated in a single sentence. However, the fact that each descriptive sentence has the same algorithmic form as in Definition 2.1 is interesting and insightful. Formally, this kind of similarity is called an **isomorphism.**[23] The Computational Paradigm discussed earlier in Chap. 1 is about a general isomorphic perspective, whereby social systems are designed as adaptations (Simon's Principle) to perform complex algorithms of many kinds.

Algorithms matter greatly in CSS because through improved design of algorithms we can develop better models of social complexity and—in doing so—advance our

[23]The term isomorphism comes from mathematics, where it means having the same formalism or equation in different domains. For example, a cannonball shot (physics) and a parabolic demand function (economics) are said to be isomorphic since both are described by a second degree polynomial, $y(x) = a + bx + cx^2$. Similarly, social transactions between two populations (human geography) and gravitational attraction between two masses (physics) follow an isomorphic inverse-square law, $y = kS_1S_2/D^2$, where S and D denote sizes (for populations and masses) and distance between them, respectively. Two systems are said to be isomorphic if the relevant equations obey the same mathematical form.

understanding of human and social dynamics. Learning how to design and implement efficient algorithms requires both technical skill and experience through practice. Key steps involve understanding **search**, **sort**, and **recursive** algorithmic structures. For example, there are significant differences in the efficiency of various search routines (e.g., linear versus binary) depending on input size and other considerations. **Binary search**—an example of what are called **divide-and-conquer algorithms**—is often desirable as an algorithm because it only requires time in logarithmic (i.e., less than linear) proportion to the size of a list. By contrast, **linear search** is much more time consuming (hence less computationally efficient) for relatively long lists, but is usually better for searching items in short lists. The exact tradeoff between the two strategies depends on data structures, code used, and hardware, but, in general, linear and binary search strategies are best for short and long lists, respectively.

Unfortunately, a binary search usually requires a presorted list, which can be a problem for sorting. **Recursive functions** for **sorting** come to the rescue! Different sorting algorithms include **select sort** and **merge sort**. Select sorting requires time that is proportional to the square of collection size (cardinality). By contrast, merge sorting is a divide-and-conquer algorithm that sorts in $n \log n$ time.

Problems

2.1 The fundamental structure of a computer consists of

(a) code and programming languages.
(b) hardware, software, and data.
(c) hardware and software.
(d) central processing unit (CPU) and peripheral units.
(e) CPU and main memory.

2.2 Data and computer programs are stored in a computer's

(a) main memory.
(b) CPU and main memory, respectively.
(c) CPU and RAM.
(d) all of the above.
(e) none of the above.

2.3 The main function of input and output devices is to

(a) enhance computer speed.
(b) improve workflow and coordinate data and software.
(c) interact with humans.
(d) interact with the physical internet.
(e) both a and c.

2.4 The two most tightly coupled components in the basic structure of a computer are

(a) input and output devices.
(b) CPU and main memory.
(c) main memory and secondary memory.
(d) CPU and secondary memory.
(e) none of the above.

2.5 Internal buses connect

(a) input and output devices.
(b) CPU and main memory.
(c) main memory and secondary memory.
(d) CPU and secondary memory.
(e) all of the above.

2.6 Understanding the fetch-execute cycle of a computer is important for issues such as

(a) deciding on the single-processor or distributed architecture of a simulation.
(b) improving internal buses and their speed.
(c) increasing I/O speed.
(d) all of the above.
(e) none of the above.

2.7 The fetch-execute cycle is best characterized by

(a) distributed computation.
(b) serial disjunction.
(c) parallel GPUs.
(d) concurrent conjunction.
(e) sequential conjunction.

2.8 Comparing current CPU and human decision-making speeds, approximately how many orders of magnitude separate the two?

2.9 The language that a CPU understands is called

(a) compiled language.
(b) object-oriented language.
(c) machine language.
(d) interpreted language.
(e) high-level language.

2.10 True or false? Compiled programs run relatively slower, but have advantages, whereas interpreted programs run faster but have more drawbacks.

2.11 The following are object-oriented languages

(a) R, Pascal, and Fortran.
(b) C++, Lisp, and Java.
(c) Java, Python, and R.
(d) C, C++, and Java.
(e) R, Lisp, and Python.

2.12 Most social theories and processes are expressed in terms of entities with attributes. This feature of the logic of social explanation makes which feature of programming languages very useful: Imperative, procedural, object-orientation, reflective, or functional?

2.13 Some programming languages are more difficult to learn than others; for example, Java has a steep learning curve relative to other languages. Which object-oriented language is well-known for its ease of learning *and* for development of good programming habits?

2.14 A mathematical equation and a table are two forms of which computational object?

(a) a looping statement.
(b) an assignment statement.
(c) a conditional branching statement.
(d) a function.
(e) none of the above.

2.15 Recall the earlier equation for the probability of a compound event $\mathbb{E}$ with N conjunctive component events, $\Pr(\mathbb{E}) = p^N$, where p denotes the probability across the N component events. (See also Example 2.2, on the probability of terrorist attacks.) Let $N = 7$ (Miller's number).

(1) Write the simplest imaginable Python program to calculate values of this function.
(2) Add some code to plot the function.
(3) Comment your code. (Commenting simple programs like this is good training practice for the good habit of writing commented code in more complex programs where comments are indispensable.).

2.16 Repeat Problem 2.15 using Eq. 2.1 ("gravity model" or law of interaction between human communities). This time, assume populations are of the same size

(say, 20,000 inhabitants), and both D and α are independent variables, so your plot should show a three-dimensional graph of the function. What happens to $I(D, \alpha)$ in the range $0 < \alpha \leq 1.0$? Validate your computational results using classical multivariate calculus, assuming D and α are strictly continuous.

2.17 What is the main question addressed by, or the main purpose of the `chaos.py` program discussed in this chapter?

(a) introduce the Python programming language
(b) demonstrate how a program can print and not just calculate.
(c) demonstrate that the function is chaotic.
(d) show the value of a looping statement.
(e) compute values of a chaotic function.

2.18 Loops are

(a) statements that define a function.
(b) control flow statements.
(c) assignment statements.
(d) all of the above, depending on where they are located in a program.
(e) conditional branching statements.

2.19 Another parallel between mathematical models and computer programs is the importance of style based on principles. True or false?

2.20 Code is sometimes used long after it was first written by the original programmer(s). An important way to mitigate the risk of code being incomprehensible to other programmers is

(a) relying on well-commented code.
(b) implementing a program in different languages.
(c) debugging.
(d) archiving code, making it publicly available.
(e) none of the above because code naturally decays due to new versions of lower level software.

2.21 Readability, commenting, modularity, and defensive coding (RCMD) are

(a) desirable features of Department of Defense software that has been certified.
(b) fundamental principles of good coding.
(c) good advice for beginners only, because advanced coders rely on stringy code that makes commenting unnecessary.
(d) all of the above.
(e) none of the above.

2.22 Answer true or false: Although abstraction, parsimony, and tractability are critical in mathematical models of social complexity, these features are less desirable in computational models, because computer programs can be as complex as the hardware will support.

2.23 Does the term "abstraction" mean the same in CSS as in computer science?

2.24 The following are distinctive sources of abstracting in CSS:

(a) social theories.
(b) empirical laws of human and social behavior.
(c) only a.
(d) only b.
(e) both a and b.

2.25 Answer true or false: In CSS the term "representation" means rendering abstracted social entities (e.g., actors, relations, institutions) in a way that a computer can understand and be able to execute a program about such entities.

2.26 The conceptual separation between abstraction and representation is due to

(a) Donald E. Knuth.
(b) Herbert A. Simon.
(c) John von Neumann.
(d) the advent of UML diagrams.
(e) the invention of object-oriented programming (OOP) languages.

2.27 Specificity, portability, reliability, optimization, and automated memory management are features of

(a) all programing languages.
(b) modern low-level programming languages.
(c) modern meso-level programming languages.
(d) modern high-level programming languages.
(e) none of the above.

2.28 As defined in this chapter, the entity "social world"

(a) is too abstract to be represented.
(b) consists of a social system situated in a given environment.
(c) cannot be represented in UML.
(d) can be represented by a UML class diagram but not by a sequence diagram.
(e) is a commonly used term in traditional social research but useless for CSS.

2.29 Ontology refers to

(a) the entities and relationships in a given area or problem of interest.
(b) only the dynamics of interest.
(c) the structure of a procedural program.
(d) the fractal dimension of an algorithm.
(e) the science of optimization in complex programs.

2.30 Which of the following object-oriented features of social science most readily facilitate ontological abstraction and representation?

(a) entities.
(b) variables.
(c) interdependence.
(d) concurrency.
(e) nonstationarity.
Hint: what are the most common subjects of social science theories and explanations of human and social behavior?

2.31 Which of the following is true?

(a) attributes belong to objects.
(b) objects belong to classes.
(c) classes belong to attributes.
(d) a and b.
(e) b and c.

2.32 The four pictures in Fig. 2.3 are increasingly complex, ranging from a small family to astronauts working on the International Space Station (ISS). How many and which of the four have an ontology consisting of human, artificial, and natural (HAN) components?

2.33 Where is the natural environment in the picture in Fig. 2.3(c)?

2.34 In object-oriented modeling, the idea that objects of the same class share all common class-level features is called

(a) ontology.
(b) inheritance.
(c) an object.
(d) a referent system.
(e) encapsulation.

2.35 Which of the following is the most fundamental function of an artifact, according to Simon's paradigm of social complexity?

(a) as a tool for information processing.
(b) as a buffer between humans and environments.
(c) as a resource for implementing collective action.
(d) as a cultural symbol of social integration.
(e) Simon's theory concerns adaptation, not artifacts.

2.36 Which of the following is not a reason for using UML diagrams?

(a) to clarify the meaning of classes and objects in models.
(b) to ensure comparability with a flowchart, an earlier form of graphic diagram.
(c) to facilitate multidisciplinary collaboration.
(d) to standardize notation concerning classes, objects, associations between them, encapsulation, and other features of every object-based model.
(e) all of the above.

2.37 The three main categories of UML diagrams discussed in this chapter are.

(a) flowchart diagrams, class diagrams, and sequence diagrams.
(b) class diagrams, flowchart diagrams, and data type diagrams.
(c) class diagrams, sequence diagrams, and state diagrams.
(d) data type diagrams, sequence diagrams, and state diagrams.
(e) class diagrams, sequence diagrams, and ontology diagrams.

2.38 In a UML class diagram the types of associations between classes is represented by

(a) the form of each association link.
(b) the multiplicity of each association.
(c) the direction of each association.
(d) the length of each association link.
(e) the type of arrowheads.

2.39 In UML and OOM terminology, the term multiplicity refers to the number of

(a) classes in a whole model.
(b) attributes or variables in a class or object.
(c) instances of a class or an object.
(d) association links in a whole model.
(e) the number of methods encapsulated in an object.

2.40 Which of the following is *not* provided as an example of the inheritance association

(a) political regimes.
(b) public goods.
(c) cognitive balance mechanisms.
(d) organizations.
(e) political revolutions.

2.41 What is the formal association type for the conceptual meaning of the phrase "consists of" in natural language?

(a) aggregation.
(b) composition.
(c) inheritance.
(d) adaptation.
(e) multiplicity.

2.42 Which is the opposite of inheritance?

(a) aggregation.
(b) composition.
(c) negation.
(d) generalization.
(e) multiplicity.

2.43 The standard model of a polity (SMP) applies to

(a) nation–state polities.
(b) all levels of analysis and types of polities.
(c) some local levels of governance, such as provinces.
(d) stable regimes with uncontested authorities.
(e) failing states.

2.44 Which type of association is absent in the high-level version of the SMP?

(a) composition.
(b) generic.
(c) aggregation.
(d) inheritance.
(e) all are present.

2.45 The main dynamic diagrams are

(a) state and sequence diagrams.
(b) flowcharts and state diagrams.
(c) sequence diagrams and flowcharts.
(d) aggregation and composition diagrams.
(e) none of the above.

2.46 In computational object-oriented (OO) terminology, the following are synonyms

(a) objects and classes.
(b) objects and associations.
(c) aggregation and composition.
(d) data and variables.
(e) model and code.

2.47 Which is the best answer to the following question: a computational social object is defined by and encapsulates

(a) attributes and operations.
(b) classes and associations.
(c) compositions and aggregations.
(d) data and variables.
(e) past and present states.

2.48 Plus and minus signs in the attributes and operations of a class or object denote

(a) aggregation or composition.
(b) objects or classes.
(c) small or large objects.
(d) visibility or accessibility.
(e) lower or higher levels of aggregation.

2.49 In UML class diagrams, the first, second, and third compartments of a class or object denote

(a) name, attributes, and operations.
(b) name, operations, and attributes.
(c) associations, variables, and operations.
(d) data, variables, and aggregation.
(e) data, variables, and parameters.

2.50 Which figure in this chapter shows public and private attributes?

2.51 Which symbol is used to denote a protected attribute?

2.52 What is the main difference between a private and protected attribute?

2.53 An object always belongs to some

(a) composition.
(b) model.
(c) attribute.
(d) theory.
(e) class.

2.54 The terms operation and method

(a) are synonymous terms.
(b) apply to classes and objects.
(c) apply to composition and aggregation.
(d) are complementary.
(e) are used for different data types.

2.55 The state of an object is defined by

(a) its attributes.
(b) its operations.
(c) both a and b.
(d) the state of its class.
(e) both c and d.

2.56 When an association between classes or objects in a model becomes more important, it can encapsulate its own attributes and operations, thereby becoming

(a) an aggregate class.
(b) a public class.
(c) a private class.
(d) an association class.
(e) all of the above.

2.57 The various ways in which data are organized for purposes of computation is called

(a) data structure.
(b) data matrix.
(c) data array.
(d) data file.
(e) data directory.

2.58 Answer true or false: Data types and data structures mean the same.

2.59 A no-fly list of individuals containing data on name, nationality, place and date of birth, and affiliations is

(a) an array.
(b) a bag.
(c) a hash table.
(d) a tuple.
(e) a tree.

2.60 A matrix of diplomatic relations between countries is

(a) a tuple.
(b) a bag.
(c) a hash table.
(d) an array.
(e) a tree.

2.61 A multi-set is

(a) a tuple.
(b) a bag.
(c) a hash table.
(d) an array.
(e) a tree.

2.62 A course catalog is

(a) a tuple.
(b) a bag.
(c) a hash table.
(d) an array.
(e) a tree.

2.63 As a data structure, a graph is a generalization of

(a) a tuple.
(b) a bag.
(c) a hash table.
(d) an array.
(e) a tree.

2.64 Given a program P, the Parnas Principle states that P should be structured in _____ modules, such that each module encapsulates as much as possible a self-contained (encapsulated) cluster of instructions and the interface between modules is such that it minimizes "communication overhead."

(a) class-object
(b) embedded
(c) nearly decomposable
(d) tightly coupled
(e) encapsulated

2.65 The TeX program used to write and produced this book consists of a root file and separate files for frontmatter, main content (chapters), and backmatter. This is an example of

(a) modularity.
(b) nearly decomposable program.
(c) optimization.
(d) all of the above.
(e) only b and c.

2.66 Which of the following social entities most resembles a modular program?

(a) a road transportation network
(b) a terrorist network
(c) a queue at an airport
(d) a group of friends
(e) a family

2.67 State the necessary conditions for effective computability.

2.68 A problem is said to be _____ when it cannot be solved in polynomial time.

(a) intractable
(b) exponentially tractable
(c) linearly tractable

(d) nearly tractable
(e) none of the above

2.69 A computable set of steps to achieve a desired result is called

(a) a computer program.
(b) an algorithm.
(c) a tractable problem.
(d) a tractable problem in polynomial time.
(e) all of the above.

2.70 Search, comparisons, maximization, sorting, communications, decision-making, movement, and other fundamental and compound forms of processing information are defining features of

(a) computable data structures.
(b) tractable programs.
(c) algorithms.
(d) object-oriented models.
(e) none of the above.

2.71 The CSS paradigmatic idea that social systems and processes are algorithmic relies on a(n)

(a) isomorphism.
(b) homomorphism.
(c) sociomorphism.
(d) conjunction.
(e) disjunction.

2.72 Key steps in understanding and using algorithmic structures effectively involve understanding

(a) scheduling, optimization, and parallelization.
(b) recursion, sort, and scheduling.
(c) search, sequencing, and recursion.
(d) search, sort, and state machines.
(e) search, sort, and recursion.

2.73 Answer true or false. The exact tradeoff between binary and linear search strategies depends on data structures, code used, and hardware, but, in general, linear and binary search strategies are best for long and short lists, respectively.

2.74 Divide-and-conquer algorithms are an example of

(a) binary search.
(b) linear search.
(c) bubble search.
(d) Google search.
(e) exponential search.

Exercises

2.75 If you are beginning to learn about computing for the first time, or your programming skills are a bit rusty or you wish you review key ideas, watch and study the series of excellent lectures by MIT's Eric Grimson and John Guttag on *Introduction to Computation and Programming Using Python* (Guttag 2014), at https://www.youtube.com/watch?v=k6U-i4gXkLM.

2.76 Read Isaac Asimov's *Foundation* trilogy, on and off, while you study this textbook. Compare and contrast psychohistory with CSS.

2.77 Discuss similarities and differences in the use of computers by computer scientists and by computational social scientists. Why does the textbook claim that CSS uses computing in ways analogous to the use of computing in the physical, biological, or engineering sciences?

2.78 The term "code" has different meanings in CSS and in social science. Explain this.

2.79 Think about the analogy between social institutions and computer hardware, and social processes and computer software. To what extent is such an analogy valid? Is it insightful? What are some pitfalls?

2.80 On page 26, there is a reference to a computer as an artifact, in the sense of Simon. Discuss this in terms of the five basic components of a computer.

2.81 Recall the analogy in Exercise 2.79. How would you extend the fetch–execute cycle analogy in the context of institutions and their internal processes (i.e., decision-making, bureaucratic procedures, implementation, and the like).

2.82 Kline (1985) and Saaty (1968) are classic methodological essays on mathematical languages as expressive of diverse qualitative (and of course also quantitative) aspects of real-world phenomena, which is why many formalisms have been invented.

Think of this idea in the context of programming languages. Do you see an analogy? If so, how valid is it? Is it insightful?

2.83 Tally and discuss features of good versus bad programming habits, based on those mentioned in this chapter, plus additional readings and online research. List your top 20 Do's and Don'ts, along with a brief explanation of each.

2.84 Identify, define, and discuss three examples of encapsulation in social entities.

2.85 Compare and contrast features of declarative versus imperative programming styles. Use Python code and a social use case of your choosing as an example.

2.86 Unlike Python, the programing language R is more functional, imperative, and procedural, although it too is object-oriented (cf. Table 2.1).

(1) Repeat Problems 2.15 and 2.16 using R.
(2) Compare results obtained from the two programs (Python vs. R).
(3) Scale up the range of the function to see if you are able to detect differences in speed between the two programs running identical computations.
(4) Discuss your results.

2.87 At the end of Sect. 2.4 there is reference again to the methodological principle that *different formal languages (in this case computer programming languages) map onto different qualitative aspects of empirical (in our case social) entities, such that formalism F should be effective and efficient in modeling entity E*—the so-called Saaty-Kline principle. Discuss this principle in terms of the coding problems and exercises in this chapter. How does Python perform in terms of modeling the various entities or domains? Do you understand the analogy between different programming languages and different mathematical languages?

2.88 Discuss a computer program as an artifact, in the sense of Simon. Cover multiple aspects of a program, such as purpose, environment, and structure, among others. Simon's theory of artifacts and complexity also includes concepts such as hierarchy and near-decomposability. How would you think about these concepts in the context of computer programs or code?

2.89 Recall the Richardson magnitude μ_R introduced in Chap. 1. Consider the magnitude of lines of code μ_R(LOC). Discuss some advantages and disadvantages of μ_R(LOC) as a measure of program complexity? Can you think of alternative ways to assess the complexity of a program?

2.90 Understanding why good coding is difficult, albeit always possible, is important. Recall the RCDM standards of good coding covered in this chapter.

(1) Explain how the code you have written so far in previous exercises in this chapter meets the RCMD standards.
(2) Since each component of the RCDM standard is required for good coding, their joint occurrence constitutes a compound conjunctive event, in the sense previously defined. What is the marginal gain in overall program quality for each increment in one of the standards?
(3) What new insights may be provided by viewing the RCDM standard as a compound event with associated probability?
(4) What would it mean to parallelize each component of the standard in order to achieve greater quality?

2.91 Discuss the problem of uncommented code in terms of Shannon's theory of the communication channel. Hint: assume the uncommented code or program corresponds to the signal.

2.92 Discuss the meaning of parsimony in a CSS context. Compare your answer with the meaning of parsimony in a mathematical social science context.

2.93 Compare the UML sequence diagram with labanotation in ballet. To what extent are the two systems of notation equivalent? Can one translate onto the other?

2.94 A hallmark of CSS is its reliance on empirically validated social theory to inform algorithms.

(1) Discuss this proposition. What does it mean?
(2) Can you think of some examples?
(3) How does this CSS principle compare with, say, artificial intelligence?

2.95 The discussion of abstraction in terms of social theory mentions several examples, such as Down's theory, and Heider's theory. How many other examples can you think of?

2.96 Look up each of the social laws covered under sources of abstraction and

(1) state each with a corresponding mathematical equation,
(2) discuss whether each law is explained by a theory, and
(3) write a Python program that demonstrates how each law works.

2.97 The Human Relations Area Files (HRAF) at Yale University is a major source of ethnographic (and archaeological) information concerning human and social features and behaviors in all human cultures. Look up its website and explore its content. Propose ways in which you could use HRAF data as a source for abstracting a computational model.

2.98 Understand similarities and differences among the following categories of data types: string, list, tuple, set, and dictionary. Explain which levels of measurement (nominal, ordinal, interval, and ratio) correspond to each of these data types.

2.99 Compare and contrast effectiveness versus efficiency in the context of representation. Do you understand why achieving both is challenging?

2.100 Without high-level programming languages a computational scientist would have no choice but to write software programs in binary code. Is this true or false? If true, why is it so. If false, why?

2.101 Explain the idea that "variables come later, 'encapsulated' in objects," in the context of object-oriented modeling and programming. Can you provide some examples different from those given in this chapter?

2.102 Table 2.3 (on human entities and selected associations) identifies a few associations. Provide ten more associations contained in each of the four social worlds referenced in this table and the source figure in the previous page.

2.103 Draw UML class diagrams corresponding to each of the four social worlds in Fig. 2.3, based only on what is observable in these pictures (i.e., refrain from modeling what may be under the stretcher being carried by humanitarian workers, or inside the space station module). Rank the class diagrams by their social complexity, taking into account classes and associations. Hint: use a word processing outliner to generate an ontology tree of each social world, rooted in the three HAN classes, each of which contains additional lower level classes down to the lowest level of resolution visible in the picture. (An ontology consisting just of pixels is not an acceptable answer :-)

2.104 Draw a UML sequence diagram of a process that could have led to the picture showing the humanitarian workers. Hint: think about the implicit disaster that preceded this scene.

2.105 Constructing a chronology of events is a critical first step in building a UML sequence diagram. Do this for the previous exercise.

2.106 Ontologically, what are some additional entities (classes and objects) that are not observable in the four pictures in Fig. 2.3, but that are necessary for each picture to be real? For example, ISS operations are in (large) part dependent on Earth-based operations.

2.107 Ontological analysis, abstraction, and representation are useful for identifying various spatial, temporal, or organizational scales in a given social world. Select two

of the four worlds in Fig. 2.3 and prepare a table comparing and contrasting them with respect to the three scales.

2.108 Tables 2.3 and 2.4 are both about the four social worlds portrayed in Fig. 2.3. Compare and contrast these two tables. Discuss their similarities and differences. In terms of classes and associations, how would you express the primary purpose of each table?

2.109 Compare and contrast Figs. 2.2 and 2.4. Identify and discuss five similarities and five differences beyond those provided in the text.

2.110 Provide two additional social examples for each of the six types of multiplicity values in Table 2.5. Understand and explain why each of your examples is valid. If not, revise your choice of examples.

2.111 Discuss the difference between the association of aggregation and that of composition. Provide three examples of each.

2.112 What is the fundamental difference between aggregation and composition in the association between classes or objects? Do you find the concept of "ownership" useful in this context? Provide three examples of each type of these two associations, different from the examples provided in this chapter. Provide two additional examples of each kind, drawn from the content of the previous chapter. Test each example to verify that it meets the definition of each form of association. Create a simple table containing your examples and a brief explanation of why the example belongs in one category or the other. For the examples from the previous chapter, make sure to cite the section and pagination of origin.

2.113 Draw a UML class diagram of this book. Discuss its content, structure, main classes, and various forms of association. In other words, use this exercise to demonstrate the extent of your understanding of the UML class diagram material covered in this chapter.

2.114 Draw a set of related UML diagrams (class, sequence, and state) that illustrate your university life as you study CSS.

2.115 Discuss the duality of inheritance and generalization. Provide three examples to illustrate this idea, different from any other examples provided thus far.

2.116 Draw UML class, sequence, and state diagrams of the topic or main subject of your most recent research paper. How many different classes and associations did you need to build these UML models? How complex or simple are your diagrams? Are they sufficient and effective in conveying the main ideas in your paper? Could

you have used these diagrams to improve communication? Do the diagrams suggest any new insights?

2.117 Develop UML class, sequence, and state diagrams for the production of legislation (enactment of laws) by a legislative body. You may choose any local, regional, national, or international example with which you are familiar.

2.118 Draw UML class, sequence, and state diagrams for describing Simon's theory of artifacts and social complexity, as described in the previous chapter and in his seminal essay on *The Sciences of the Artificial*.

2.119 UML sequence diagrams and state diagrams are both for describing dynamics. Discuss the similarities and differences between these two categories of diagrams. Given some social theory that you are familiar with, which would you use first? State some reasons for your preference. Select the social theory you know best and model it with both diagrams. Hint: start by constructing a class diagram first, since that way you will have an initial idea of key relationships, some of which may be dynamic in nature, while others may be just organizational.

2.120 If you know what a Markov chain model is, discuss the similarities and differences between it and a UML state diagram. Include some illustrative examples.

2.121 Earlier in Chap. 1 (and later in Chap. 7) it was pointed out that a viable scientific theory always contains an explanatory process consisting of one or more causal mechanisms. Provide UML diagrams for one or more of the following classical social theories:

(a) the theory of cognitive balance
(b) deterrence theory
(c) Ricardo's theory of international trade
(d) the theory of collective goods
(e) the theory of complex adaptive systems
(f) balance of power theory
(g) collective action theory
(h) social control theory

2.122 Attribute, variable, parameter, indicator, data, dimension, feature, aspect, and similar terms form a conceptual cluster about the characteristics or categories that describe an object. Discuss pros and cons of this "tower of Babel" situation. Is it necessary to rely on so many terms to express roughly the same meaning? Which of these terms are closer in meaning, and which are more distant? Can you propose some kind of map of these terms?

2.123 The coverage of objects, operations, and UML in this chapter relies heavily on the standard model of a polity as a running example. Provide two other examples with comparable applicability and usefulness for learning the same ideas.

2.124 Consider the following episodes from ancient to contemporary history. Select two of these and draw UML class, sequence, and state diagram models of the entities and dynamics involved in them. Compare, contrast, and discuss your results. Make a list of new insights provided by this analysis, as well as possible research questions for further investigation.

1. the Arab Spring
2. the 9/11 terrorist catastrophe
3. the 2008 financial crisis
4. the end of the Cold War
5. the Industrial Revolution
6. the European conquest of the Americas
7. the decline and fall of the Roman Empire
8. the onset of the Great Peloponnesian War
9. the Neolithic Revolution

2.125 Use UML diagrams to model one or more of the following public policy issues:

1. climate change
2. poverty
3. mass migrations and humanitarian crises
4. cybersecurity
5. economic development
6. proliferation of weapons of mass destruction
7. health epidemics
8. technological innovation
9. elections in a democracy

2.126 Select a major social science data set, such as the National Election Study, Eurobarometer, or the Yearbook of the United Nations, and a major "big data" set, such as ICEWS, GDELT, daily Wikipedia edits, or other in any domain and compare similarities and differences in terms of data structures. Which data structures are most common in big data analytics? (Questions like this also appear in the next chapter.)

2.127 Think about the data required to answer 2.124 on historic episodes and assess pros and cons of various types of data structures for creating UML diagrams of events and processes. Which data structures are more/less useful in this context.

2.128 Provide a formal statement of Parnas' Principle, or aspects related to the principle, in mathematical form.

2.129 Consider the two conditions for effective computability. Discuss the following: Are they necessary? Sufficient? Necessary but not sufficient? Necessary and sufficient? Formalize the idea that computability is a compound event and list some inferences and insights from formal analysis.

2.130 Extend the social science examples of computability provided in Sect. 2.11. Think of other social situations in which intractability may arise. Which kind of social phenomena are most tractable? Can you think of some features of social phenomena that generate intractability?

2.131 If you like to cook, discuss the recipe of your favorite dish as an algorithm. If you do not cook, discuss the recipe for chocolate fudge in Lofti Zadeh's seminal 1973 paper on the application of fuzzy sets. Identify similarities and differences between computer algorithms and cooking recipes.

2.132 Discuss the following social processes as algorithms:

1. policy-making
2. a national census
3. signing a contract
4. humanitarian assistance
5. economic development
6. proliferation of weapons of mass destruction
7. life cycle of a health epidemic
8. technological innovation
9. elections in a democracy

2.133 The application of algorithms and data structures is the essence of computational science. Discuss the idea that social systems and processes are algorithmically structured. Use a specific domain of social science for this exercise, such as the one you know best. Include aspects of data structures as well. Identify a set of inferences and insights. Compare this paradigmatic view of CSS with traditional disciplines.

2.134 Two milestone algorithms in computer science are von Neumann's merge sort algorithm and Google's PageRank algorithm.

(a) Look them up and examine their respective algorithmic structure.
(b) Provide two examples of social processes that resemble these algorithms.
(c) Merge sorting is a divide-and-conquer algorithm that sorts in $nlogn$ time.
(d) What about the PageRank algorithm? Explain how it works.
(e) How do the two computabilities compare?

2.135 Amdahl's law states that the gain in speedup S in a parallelized program is proportional to the number of processors N and inversely proportional to the proportion of code P that cannot be parallelized.

(1) Find the mathematical equation for Amdahl's law.
(2) Analyze it using multivariate calculus.
(3) Think of social examples of distributed systems and discuss them in terms of Amdahl's law.

Recommended Readings

H. Abelson, G.J. Sussman, J. Sussman, *Structure and Interpretation of Computer Programs*, 2nd edn. (MIT Press/McGraw-Hill, Cambridge/London, 1996)

S.W. Ambler, *The Elements of UML 2.0 Style* (Cambridge University Press, Cambridge, 2005)

I. Asimov, *Foundation Trilogy: Foundation (1951), Foundation and Empire (1952), and Second Foundation (1953)* (Gnome Press, New York, 1953)

J. Barker, An innovative single-semester approach to teaching object modeling and java Programming, in *3rd International Conference on the Principles and Practice of Programming in Java, PPPJ 2004*, (2004)

J. Barker, *Begining Java Objects: From Concepts to Code*, 2nd edn. (Apress, Berkeley, 2005)

P.E. Black, Data structure, in *Dictionary of Algorithms and Data Structures*, ed. by P.E. Black (US National Institute of Standards and Technology, Washington, 2007)

C. Cioffi-Revilla, Simplicity and reality in computational modeling of politics. Comput. Math. Organ. Theory **15**(1), 26–46 (2008)

A.C. Clarke, *2001: A Space Odyssey* (Penguin, New York, 1968)

H.-E. Eriksson, M. Penker, B. Lyons, D. Fado, *UML 2 Toolkit* (Wiley, New York, 2004)

M. Felleisen, R.B. Findler, M. Flatt, S. Krishnamurthi, *How to Design Programs: An Introduction to Programming and Computing* (MIT Press, Cambridge, 2001)

M. Flynn, *In the Country of the Blind* (Tor Science Fiction, New York, 2003)

E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software* (Addison-Wesley, Reading, 2012)

R.L. Graham, E. Knuth Donald, O. Patashnik, *Concrete Mathematics: A Foundation for Computer Science*, 2nd edn. (Addison-Wesley, Reading, 1994)

E. Grimson, J. Guttag, *MIT 6.00 Introduction to Computer Science and Programming*. Online course: http://videolectures.net/mit600f08_intro_computer_science_programming/

J.V. Guttag, *Introduction to Computation and Programming Using Python* (MIT Press, Cambridge, 2013)

Y.-T. Lau, *The Art of Objects: Object-Oriented Design and Architecture* (Addison-Wesley, Boston, 2001)

J.D. Murray, After Turing: mathematical modelling in the biomedical and social sciences, in *How the World Computes*, ed. by B.S. Cooper, A. Dawar, B. Löwe (Springer, Berlin, 2012), pp. 517–527

D.L. Parnas, On the criteria to be used in decomposing systems into modules. Commun. ACM **15**(12), 1053–1058 (1972)

E. Regis, *Who Got Einstein's Office?* (Basic Books, New York, 1988)

H.A. Simon, *The Sciences of the Artificial*, 3rd edn. (MIT Press, Cambridge, 1996)

M. Weisfeld, *The Object-Oriented Thought Process*, 2nd edn. (Developer's Library, Indianapolis, 2004)

J. Zelle, *Python Programming: An Introduction to Computer Science*, 2nd edn. (Franklin Beedle & Associates, Sherwood, 2010)

http://www.springer.com/978-3-319-50130-7

Introduction to Computational Social Science
Principles and Applications
Cioffi-Revilla, C.
2017, XXXVI, 607 p. 59 illus., 21 illus. in color.,
Hardcover
ISBN: 978-3-319-50130-7