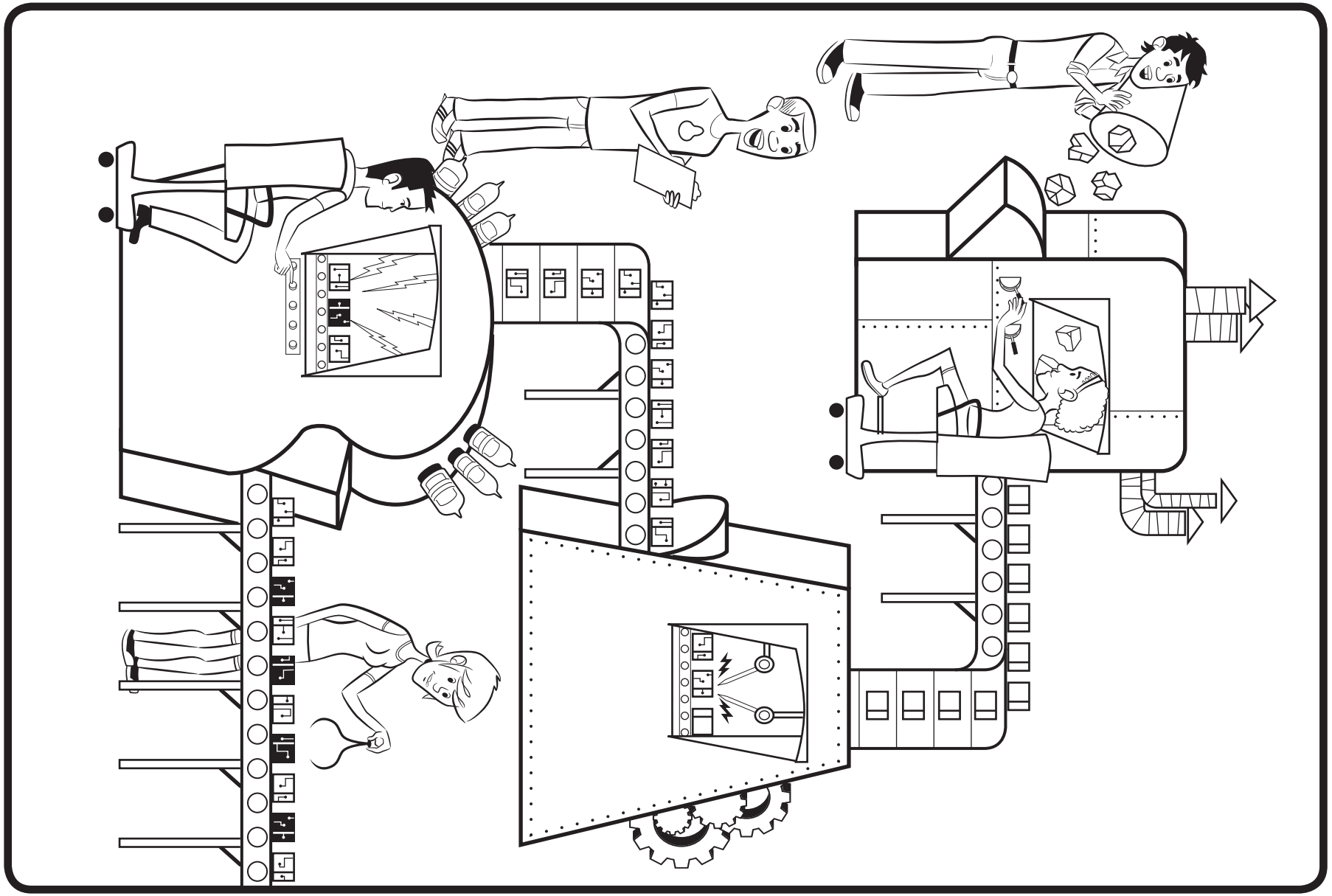


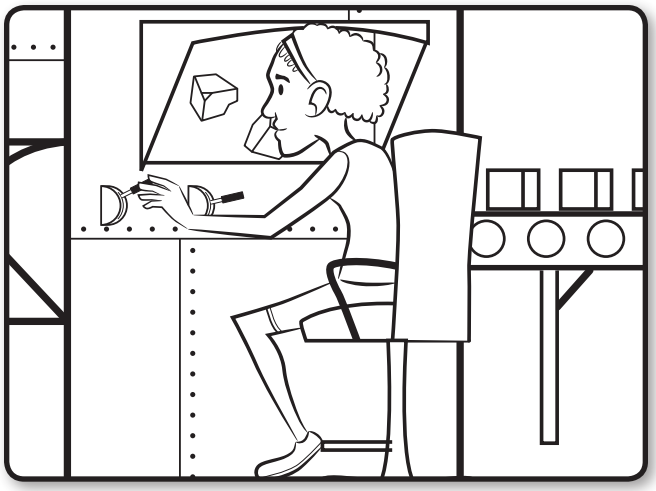
Analyzing effects of computation

Computational Thinking Practices #1:

Analyzing effects of computation

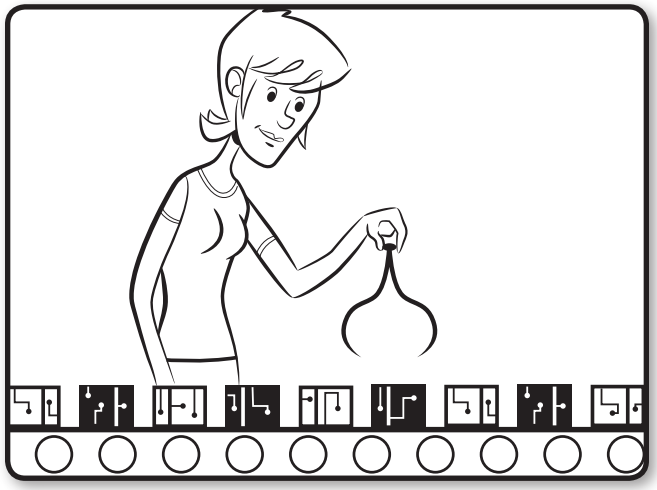
Computation is everywhere. From search engines that help us find information, to cash registers in stores, to software used for designing bridges, we live in a world built on the effects of computation.

Computation is not just another word for technology. For example, a cellular phone contains many different technologies: a radio transmitter and receiver, a processor, memory, and electro-mechnical parts like buttons and touch screens. When studying the effects of computation, we aren't trying to learn how physics governs these technologies. Analyzing the effects of computation means specifically looking at what happens when we collect, store, and process data.



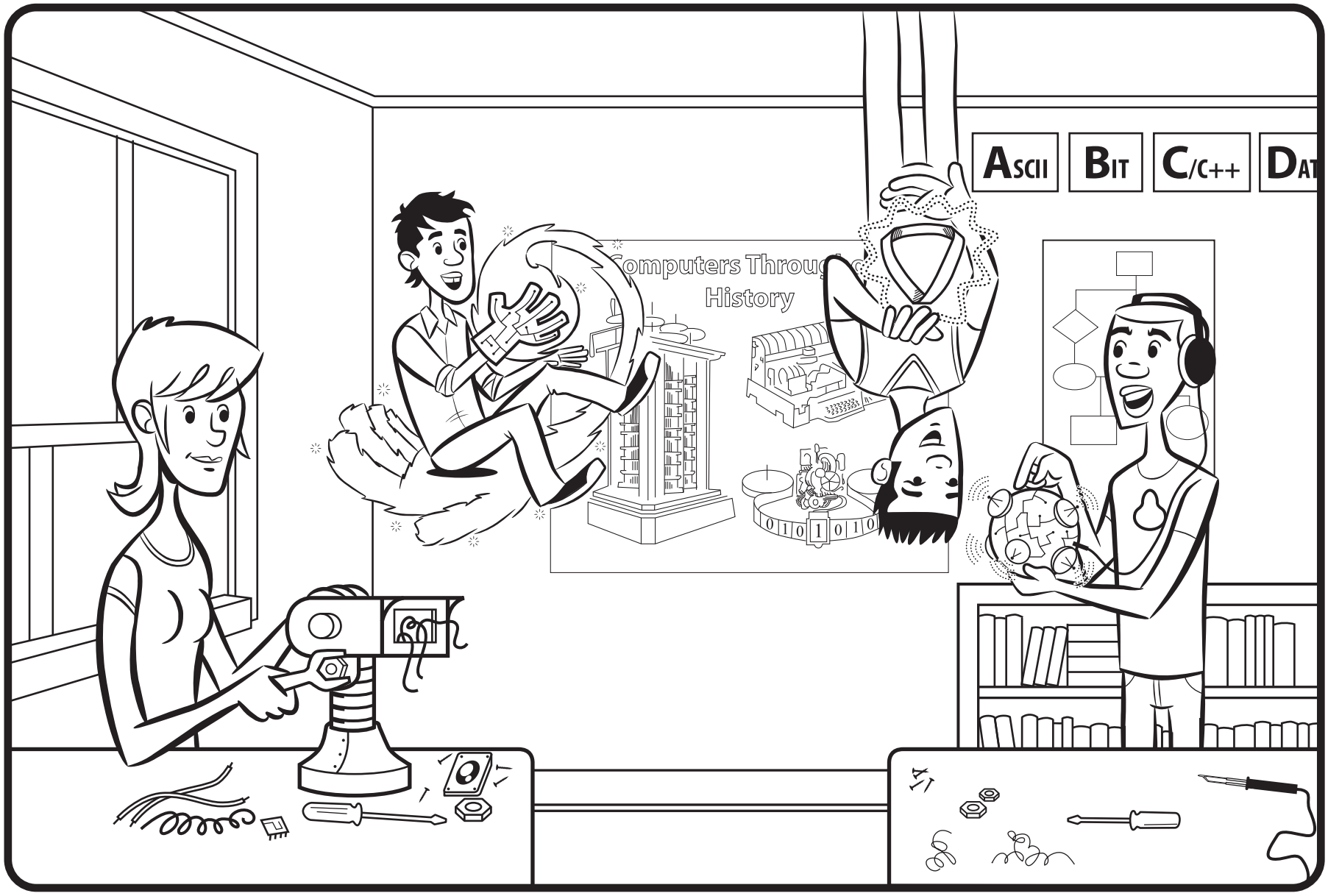
The computation done by a cell phone involves recording your voice as data, compressing and transmitting that data, and interacting with a larger system that routes your call's data to its destination. This same computational process is done in reverse so your conversation partner can talk back to you. That sounds like a lot of computational work for your cell phone to do, but that's only part of what happens when you make a call.

All the sending and receiving of data happens via radio waves. When the technology for radio communication was first invented by Nikola Tesla, it could only be used for mass communication in the form of broadcasts. It takes computation to transform that raw technological capacity into the more refined form we use today in our cell phones. One effect of computation is that radio can now be used for person-to-person communication, with many simultaneous conversations happening in the same physical area.



When we analyze the effects of computation, we take note and measure how data is transformed. We look at how information is processed and what is accomplished by that processing. We can think about what we might do if such computational power wasn't available. That can also help us start to imagine new things we can strive to accomplish using computation.

A major part of the work in analyzing the effects of computation is careful observation, as **Blaze, Ada, Charles, Alan,** and **Grace** are doing in this illustration. In their world, as in ours, computation is everywhere. By looking closely, we can start to see what computation -- not just raw technology -- does for us.



Producing computational artifacts

Computational Thinking Practices #2:

Producing computational artifacts

Creating computational artifacts is all about making things. Programming is one of the most visible ways we make computational artifacts. In that case, the artifacts are both the programs we made and their outputs. But the term *computational artifact* is not limited to just computer programs. It can refer to a whole range of things from microprocessors to bar codes to an airplane's navigation system.

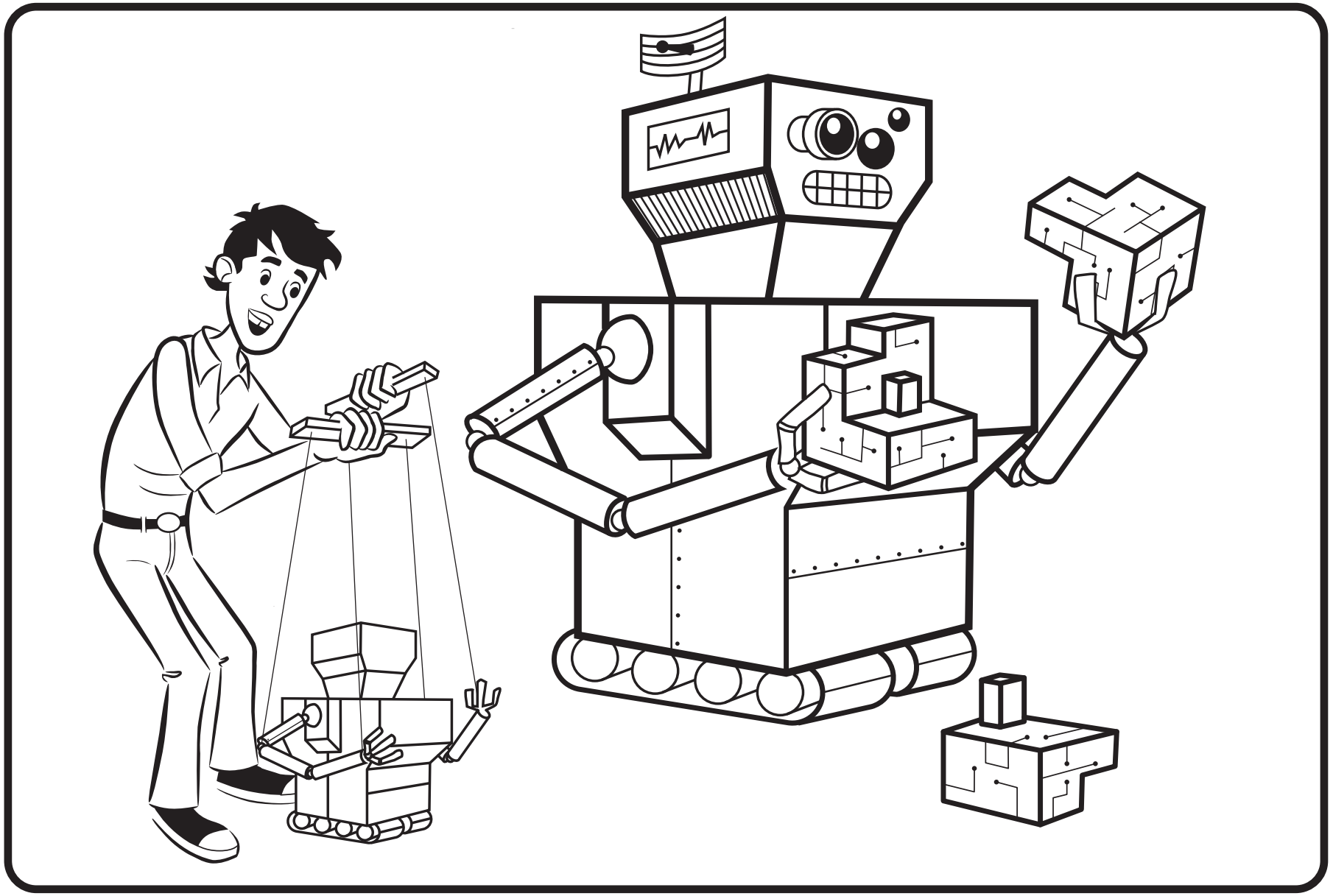
In this illustration, the characters are building, testing, and exploring computational artifacts. The process of creating is not limited to only thinking of ideas, or just assembling parts. The machines you see in these cartoons are symbolic, designed to be open to interpretation and imagination. Here are some ways of looking at them to help you get started:

Grace is building something new. At the moment, she's using a wrench because it's the right tool for the work she's doing. She's not just using a machine built by someone else; she's actually making something new herself. Sometimes, creating things is a time-consuming and difficult process, but it gets easier with experience.

Alan is walking on the ceiling. He's holding a Möbius strip, a topological surface with only one side. When twisted and attached back to itself, a regular flat rectangle can be transformed into a Möbius strip. Using computational thinking, we can change our perspective to solve a complex problem -- like Alan, who is upside down! Many computational concepts, like the idea of the Möbius strip, can challenge our assumptions about what's possible and reveal deeper truths about the properties of the systems we are using or creating. At first this can seem as difficult as walking on the ceiling, but after a while you'll probably find it fun.

Blaze is wearing a glove that controls a much larger and stronger hand. This hand can do many things, including lift up Blaze himself. Blaze's glove is a metaphor for computational artifacts that allow us to harness the power of machines to carry out massive calculations. When we turn that power back upon itself as we do when we use recursion, higher-order functions, or write a compiler for a language in that same language, things can get very exciting.

Charles is holding an orb covered in what look like small radio dishes. Computational artifacts need not be designed to work in isolation. They can work together and communicate to accomplish a task, like we see in multi-core processors or parallel computing. Perhaps the radio dishes are helping Charles to hear things that other characters can not. Similarly, algorithms for pattern recognition, signal processing, error correction, and noise reduction enhance our ability to extract information from data. With the help of computational artifacts, we gain new powers.



Using abstraction and models

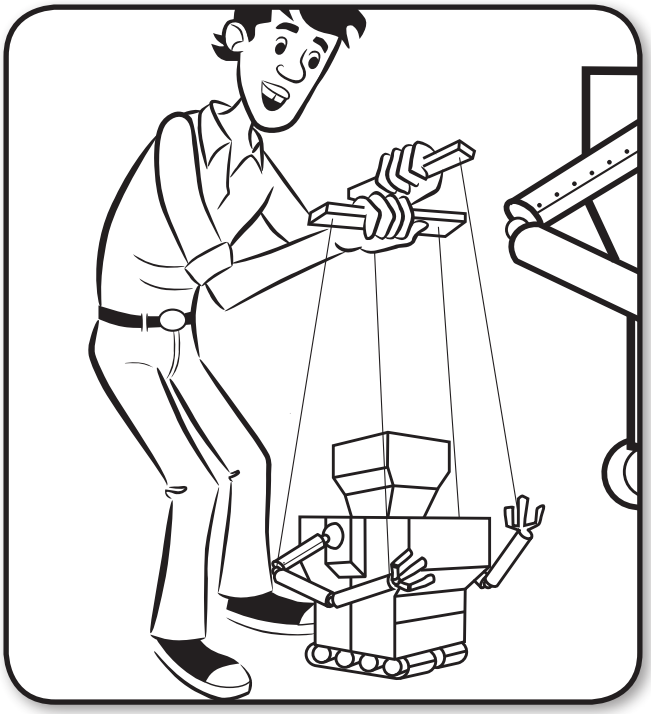
Computational Thinking Practices #3:

Using abstractions and models

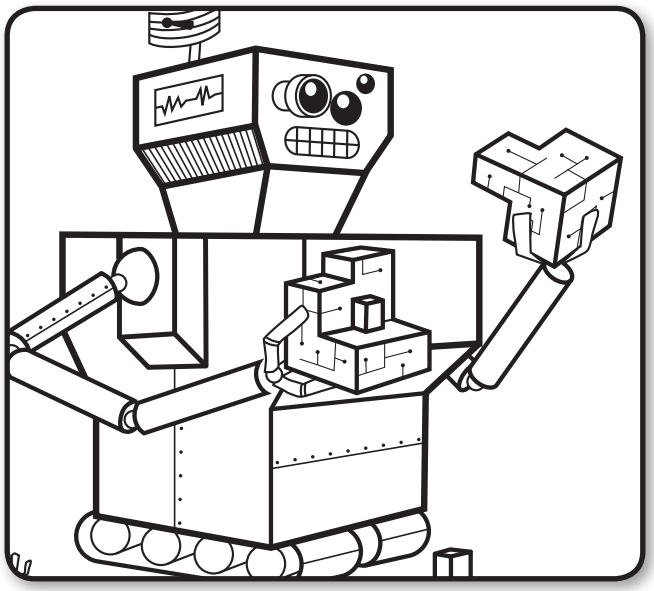
"All models are wrong, but some are useful."

– George E. P. Box

One meaning of the word model is: A smaller or simpler version of the original item. The model could be a physical object like the small robot in this illustration. Notice that Blaze isn't trying to move the arms of the huge robot, nor trying to move the heavy blocks himself. Instead, he is working with a model robot small enough for him to literally put his own hands around. This is simplifies the physical work he needs to do, just like a simplified model of an idea makes thinking easier.

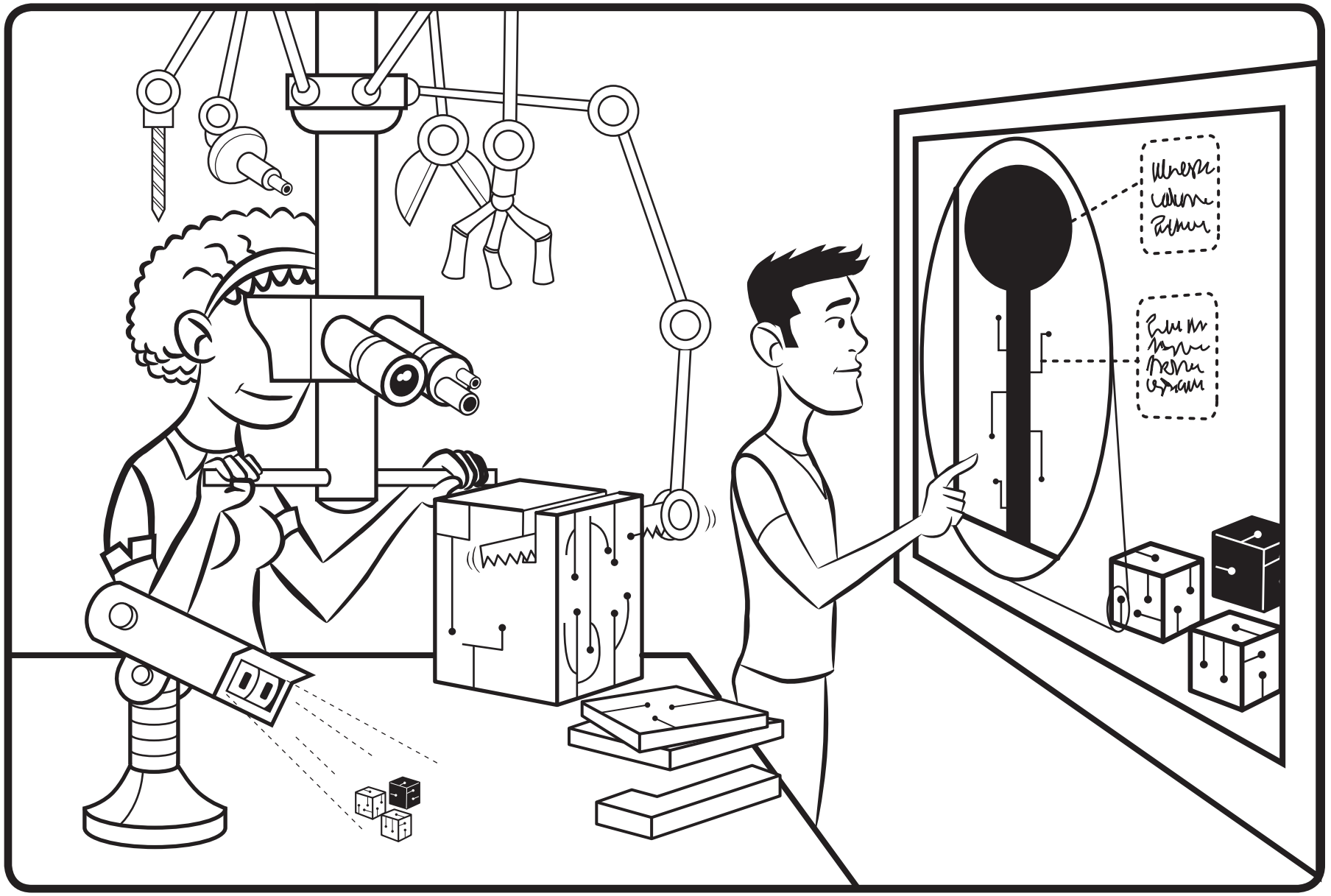


For example, classical mechanics is a model: It's Newton's easily-computed approximation of the more complex reality of motion. In computer science, we make a model every time we write a program. We must choose the information and level of detail represented in our program. Some details must be left out. If we tried to include everything in a model or program, we would end up simulating the whole world!



In a complex system, we might use many different models and make them work together. We might not even care if one part of the system was switched out for something else that can accomplish the same goal. We could say we've abstracted that part of the system. Carefully selecting the qualities we care about and ignoring the rest of the details is the key to abstraction. When we deliberately separate our system into parts that can be individually understood, tested, reused, and substituted, then we are creating new abstractions.

See also: Modularity, map–territory relation, marionette.

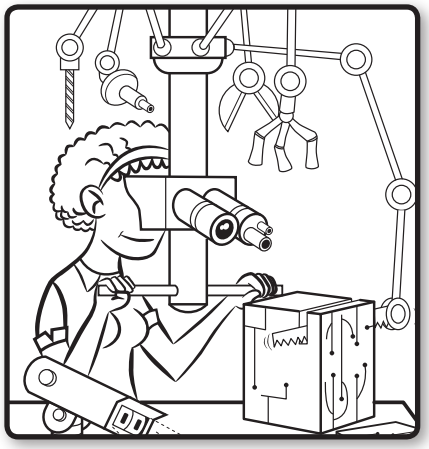


Analyzing problems and artifacts

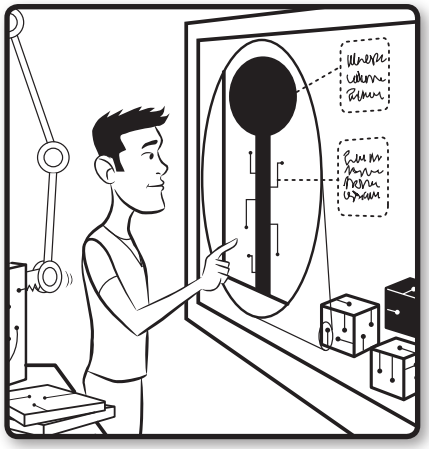
Computational Thinking Practices #4:

Analyzing problems and artifacts

Wikipedia says, "Analysis is the process of breaking a complex topic or substance into smaller parts to gain a better understanding of it."



In this illustration, **Ada** is using a tool with many attachments, representing the idea that we often need to try multiple approaches and many different tools before we can "crack" a problem. Different problems and different approaches to these problems have different weaknesses. Often, we can't solve a problem until we try a number of different ways to break it down. That's why it's so valuable to have a variety of conceptual tools available when working on a problem.



Over to the right, **Alan** is controlling a zoomed-in view of the cubes on the table. This allows him to see and understand not only how a cube looks and acts from the outside, but to how its internal workings contribute to its overall behavior. Programmers engage in this kind of analysis when they use a debugger; so do electrical engineers when they use an oscilloscope to visualize signals.

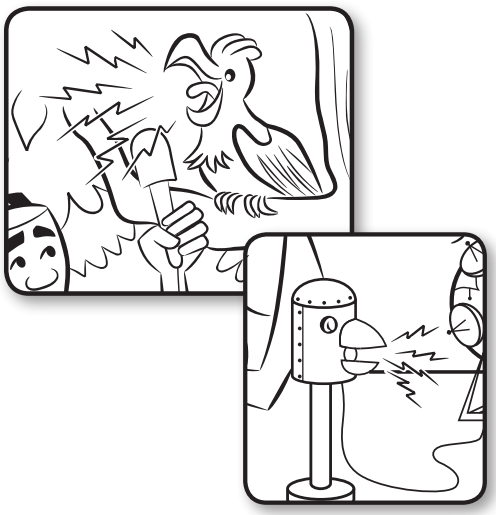

MIGRATION PATTERNS
PARROT

Computational Thinking Practices #5:

Communicating processes and results

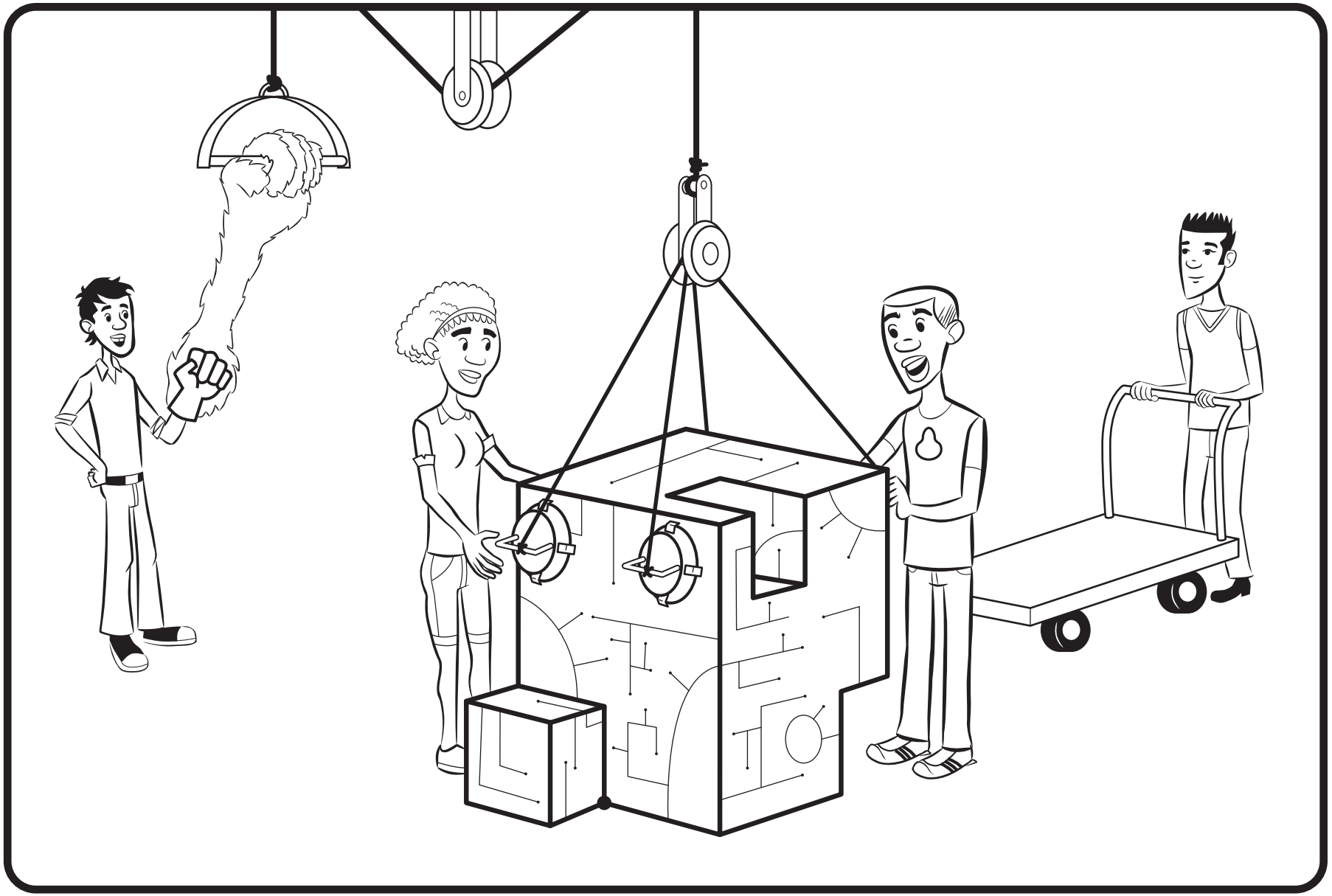
Very rarely is a computational artifact self-explanatory. A CPU made of microscopic transistors on silicon or a compiled binary program of 1's and 0's are both quite difficult to understand. Their forms are optimized for computational performance, not human comprehension. The design plan for the CPU or the source code of the program are more easily understood. But even these precursors don't necessarily explain how they were made or why they work.

Computational thinking requires us to discuss processes for both people and machines to follow, and how these processes are intended to lead to specific results. For example, when a programmer is learning how to write programs, they need to be taught to debug by printing the value of a variable. When you discover a new mathematical technique for manipulating 3d shapes efficiently, you have to write it up so other people can understand and use it. Communication is the way we bring the things we know into the world.



When we use computation to solve a problem, the answer we get isn't automatically meaningful to others. We have to communicate this result in a way that reveals both its importance and its origin.

In the illustration, **Charles** is capturing the sounds of a parrot in the wild and transmitting them to **Grace** at another location. We can interpret this literally as communication of audio data, like someone's voice on a phone call. However, to another parrot perhaps the parrot's song represents a process (like a technique for finding fruits and seeds, or a plan for timing seasonal migration) or some important news (such as the winner of the annual parrot speechmaking contest). Communicating about processes and results allows us to benefit from insights gained by other computational thinkers.



Working effectively in teams

Computational Thinking Practices #6:

Working effectively in teams

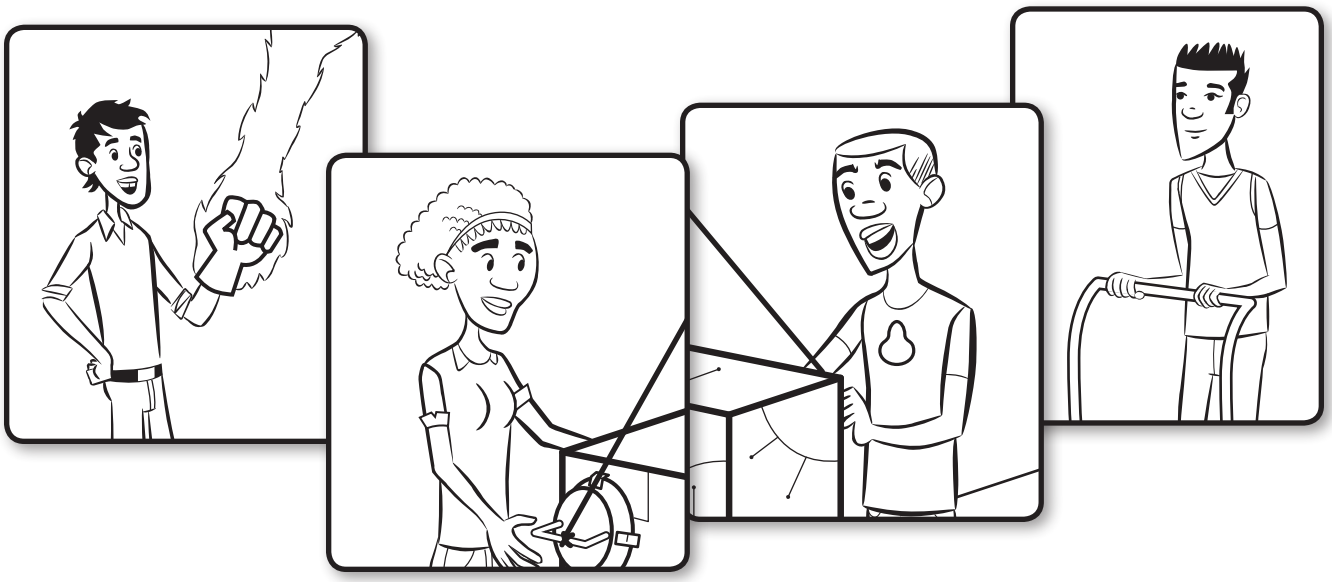
The ability to work in a team can mean the difference between success and failure. Building any complex system, software or hardware, requires more work be done in less time than any single person can accomplish. But adding more people doesn't necessarily mean that the job will get done sooner.

To make teamwork effective, individuals need interpersonal and communication skills as well as knowledge of different team methodologies and processes. As teams grow in size, the role of culture and management becomes increasingly important. Teamwork, like any other skill, takes practice.

Various strategies for dividing up work have different strengths and weaknesses. Figuring out the best way to work together isn't always easy, but it's important for computational thinking.

As multi-core processors and distributed computing become more common, we see computers themselves working in teams. Most web sites that you visit are served from data centers, where hundreds or thousands of individual computers work together to accomplish amazing tasks. We humans can do the same!

See also: Brooks's law, pair programming, revision control system.



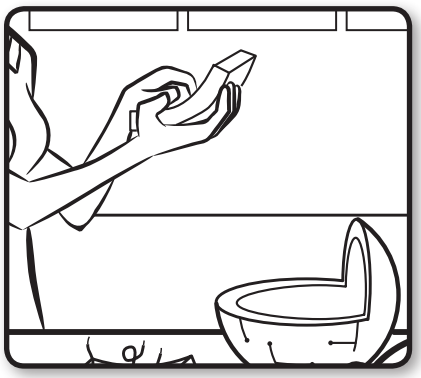
Decomposition

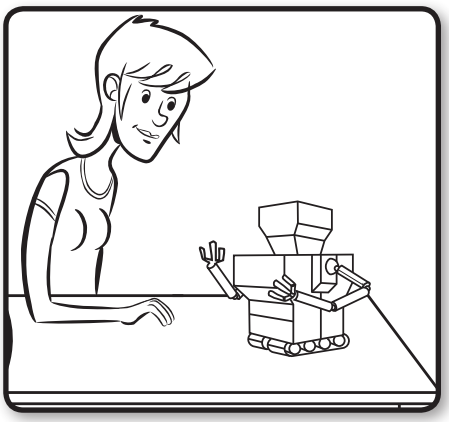
Exploring Computational Thinking #1:

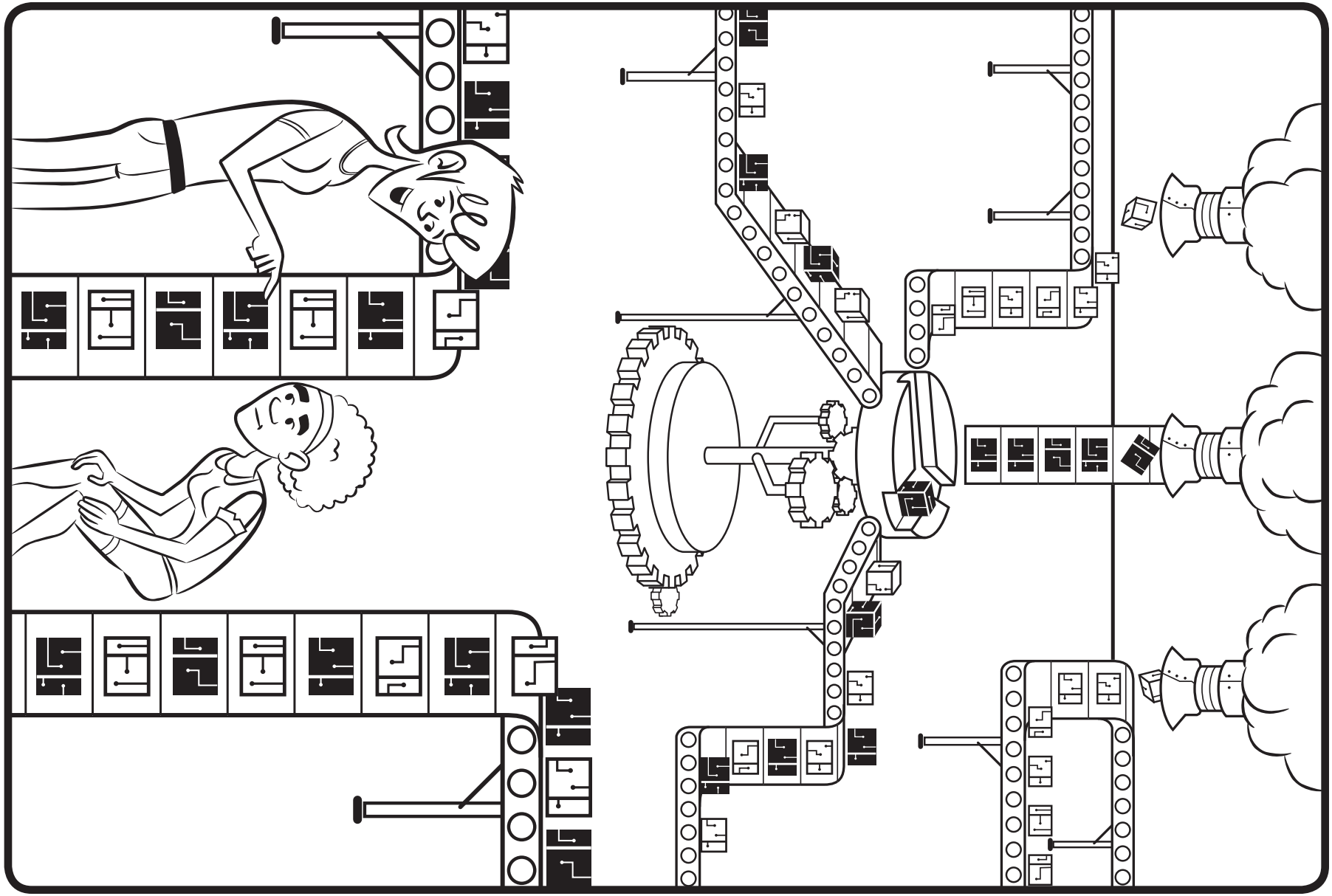
Decomposition

In this illustration, **Ada, Alan,** and **Grace** are each taking apart some of the machines we have seen in other scenes. But decomposition isn't only about disassembling objects. It's also about pulling apart the steps of a process. Some things that we think of as a single action are actually a composite of many smaller actions. For example we may say that we are going to make dinner. But when we apply decomposition, we find that making dinner actually means to opening the refrigerator, getting out the broccoli, cutting up an onion, turning on the stove, and many other small steps.

A difficult computational problem can sometimes be solved by thinking of the overall task as being made up of many smaller, simpler tasks. Decomposition involves identifying those smaller tasks and how they fit together. The more times you do this, the easier it gets. Just ask **Ada**, who is taking apart an orb. Even though each of the orbs is a little different, she has a pretty good idea of what pieces she's going to find when she takes one apart.





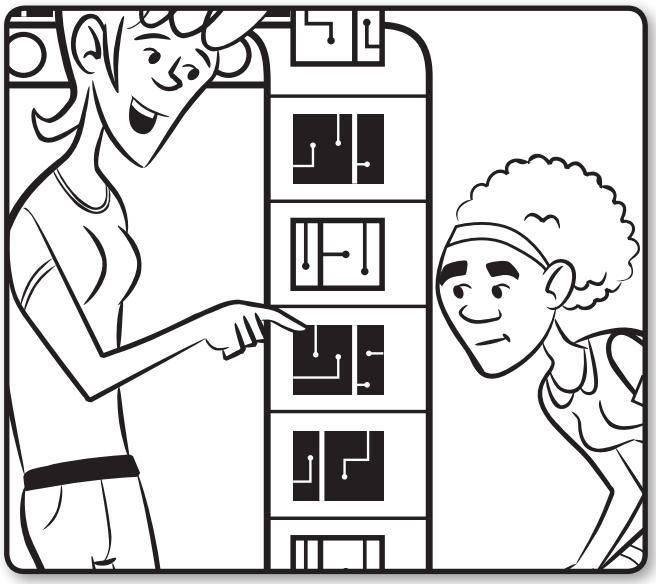


Decomposition

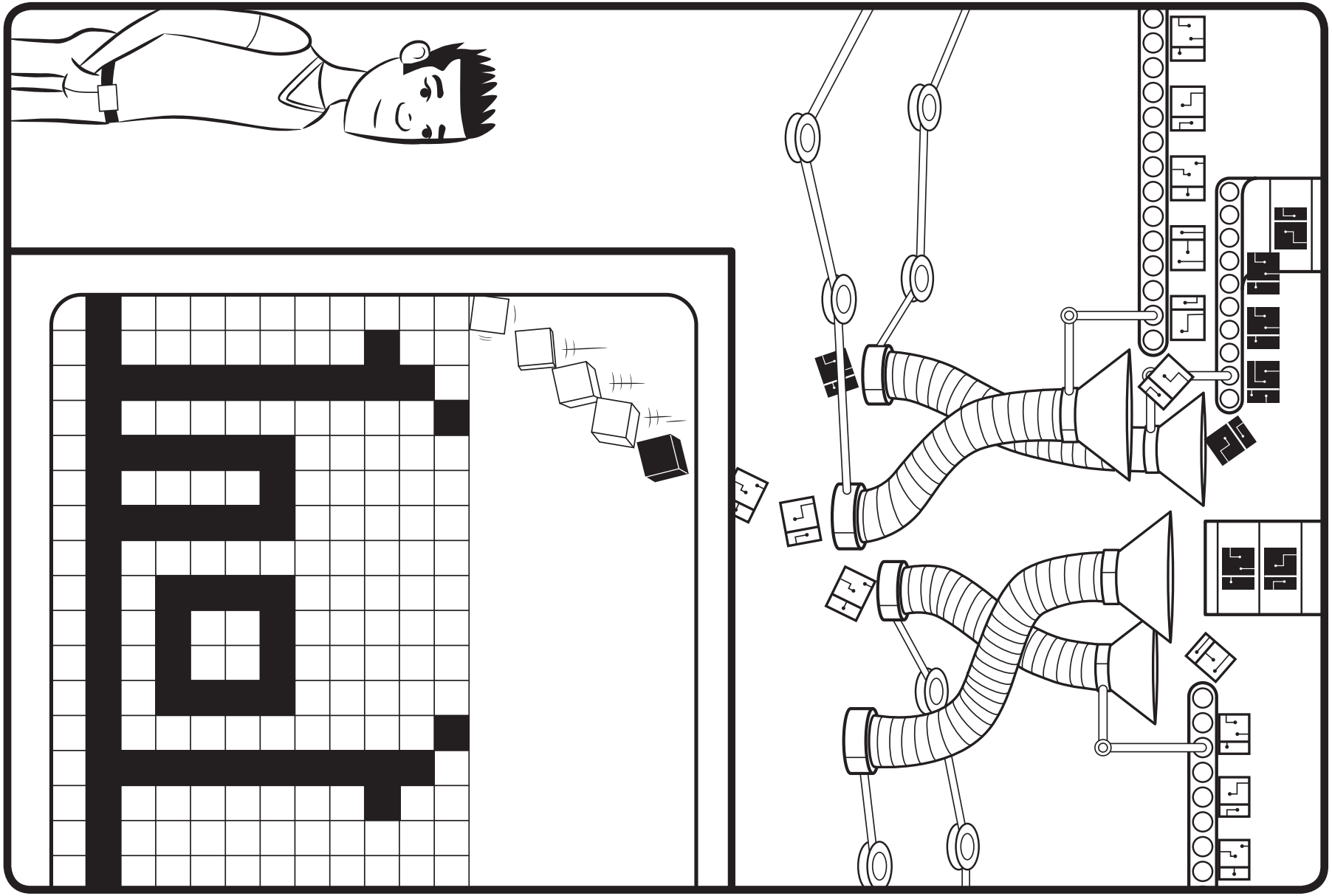
Exploring Computational Thinking #2:

Pattern Recognition

There's something strange about the pattern of blocks, and **Grace** is pointing it out to **Ada**. Although they aren't looking at the whole complicated machine that produces this pattern of blocks, they can still identify what is unusual. This doesn't mean anything is wrong, but it tells them that there might be more going on than they first thought.



Forming an idea of what you expect is one way to find patterns. The more you look, the more patterns you will find in nature, in computational artifacts, and in processes. When we recognize a pattern, we can use our other computational thinking skills to help us understand its importance.



Pattern Generalization and Abstraction

Exploring Computational Thinking #3:

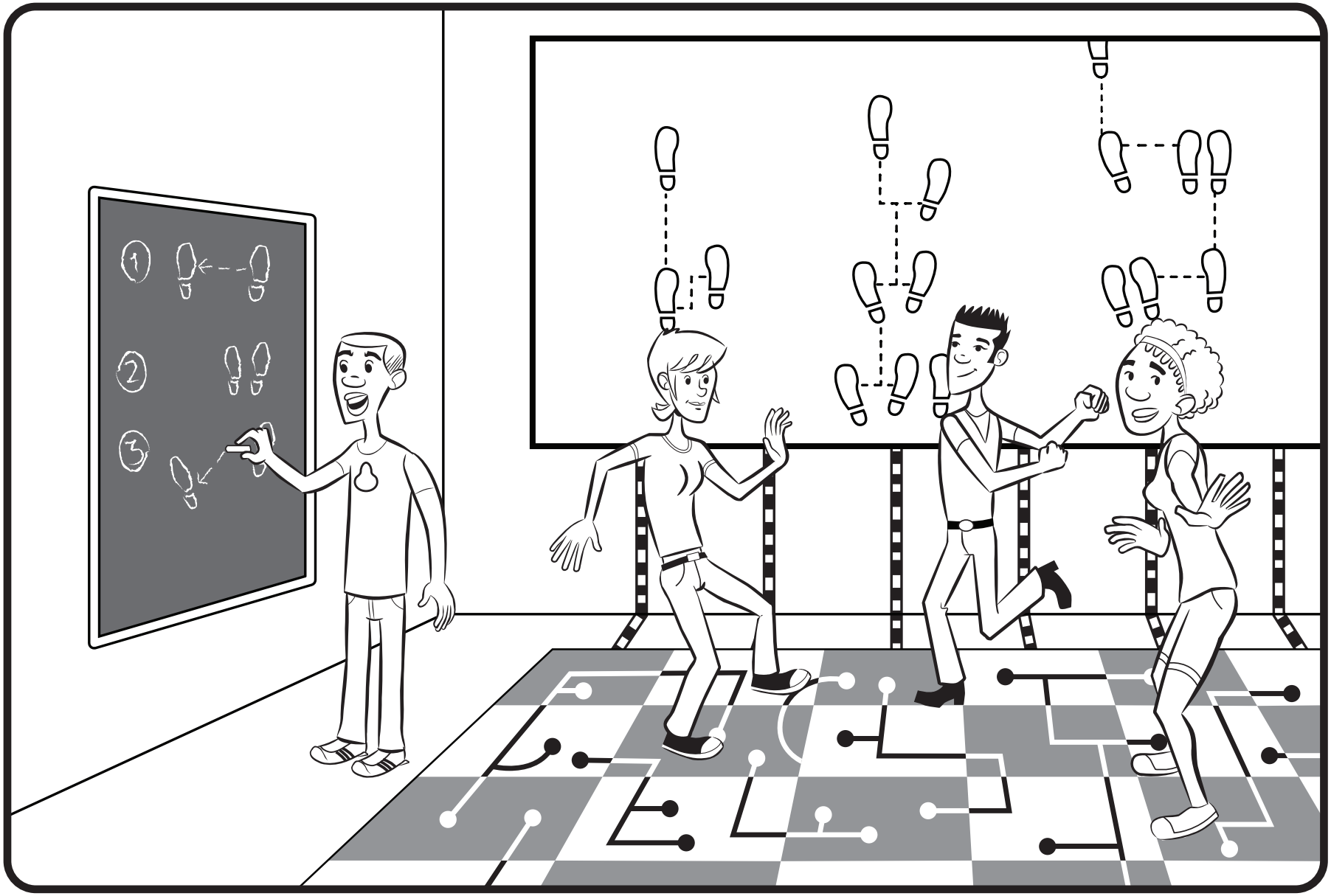
Pattern Generalization and Abstraction

After you've seen the same pattern a few times, you might start thinking of different ways to describe it. **Alan** is watching some blocks fall into place to form a picture. If the machines drop the same pattern of blocks again, they'll make the same picture.

There's a lot for **Alan** to think about here, watching the blocks fall. There lots of possible patterns -- see if you can calculate the number. There are also a lot of ways to describe these patterns.

If we wanted the machines to make a picture of a house with the door on the right side instead of the left side, the instructions would be almost the same. What if instead of giving the machines new instructions every time, we simply told them what to change about some other instructions? We would need instructions that describe how to change other instructions.

Thinking this way is some of the work we do when we try to generalize patterns. We look for what's the same about a group of patterns and try to describe it them a way that's both clear and efficient. If we can describe the group of patterns all at once, a pattern of patterns, then we have an abstraction.



Algorith Design

Exploring Computational Thinking #4:

Algorithm Design

It's a computational thinking dance party! The special dance floor in this illustration might be recording their steps, or it might be lighting up with dance instructions. But while **Grace, Alan,** and **Ada** are dancing away, **Charles** is actually designing a new dance. Like an algorithm, a dance is a set of steps that can be followed by others to get the same result.

Sometimes we think of algorithms as being written down like a computer program, but an algorithm is more like an idea. The same algorithm can be written in many different computer languages. It's the steps in the process that make an algorithm what it is.

In order to design an algorithm, or a dance, you need to understand your goal. You also need to understand the constraints of the system. Humans only have two feet, so a dance designed for humans has to work with that limitation. Computational systems have different kinds of limitations, such as the speed of the processors or the size of the memory or the amount of electricity they consume. Designing an algorithm that accomplishes specific goal within the constraints of the system is like creating an elegant dance that everyone else wants to learn.