Chapter 24. Making Teams Effective

In addition to creating technical architectures and making architecture decisions, software architects are also responsible for leading the development team and guiding it through implementing the architecture. Architects who do this well create effective development teams that work together closely to solve problems and create winning solutions. While this may sound obvious, we've seen too many architects ignore their development teams and create an architecture alone, in a siloed environment. When this architecture is handed off to the development team, the developers often struggle to implement it correctly.

Making teams productive is one of the ways successful software architects differentiate themselves. In this chapter, we introduce some basic techniques for improving development teams' effectiveness.

Collaboration

All too often, the software industry treats architecture and development as entirely separate activities. Consider Figure 24-1, which compares the traditional responsibilities of architects to those of developers. Architects are responsible for activities like analyzing business requirements to extract and define architectural characteristics, selecting architecture patterns and styles to solve the problem domain, and creating logical components. The development team uses the artifacts the architect creates during these activities to create class diagrams for components, build UI screens, and write and test source code.

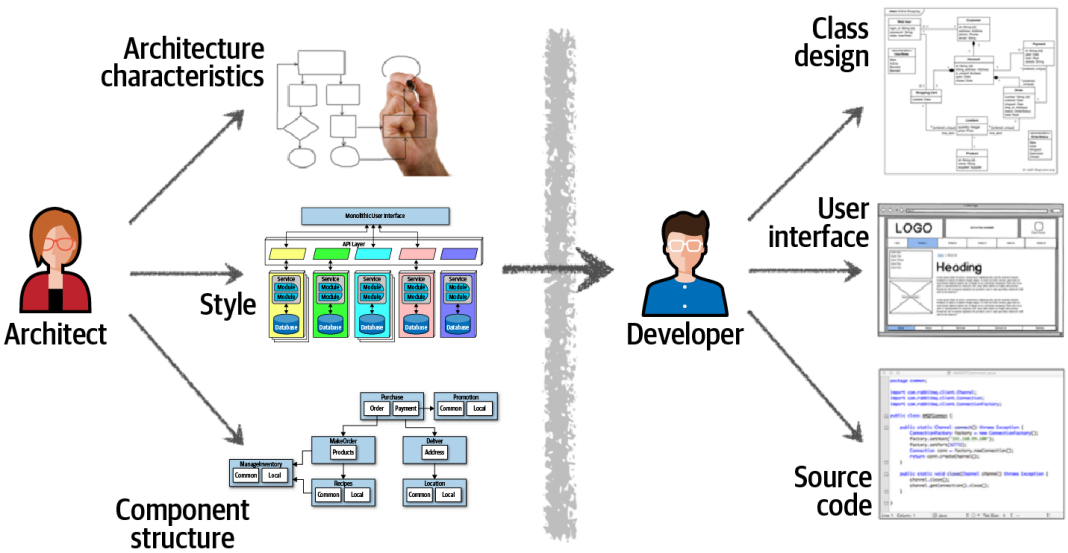

Figure 24-1. Traditional roles of architects versus developers

The image in Figure 24-1 illustrates why this traditional approach to architecture rarely works. Look at the unidirectional arrow that passes through the virtual and physical barriers separating the architect from the developer: that's what causes all of the problems. Architects' decisions don't always make it to the development team, and when development teams change the architecture, it rarely gets back to

the architect. In this model, because the architect is so disconnected from the development team, the architecture rarely accomplishes its goals.

The key to making architecture work is breaking down the barriers, both physical and virtual, between architects and developers, instead forming a strong bidirectional collaborative relationship between them. The architect and development team must be on the same virtual team, as in the collaborative model depicted in Figure 24-2. Not only does this model facilitate strong bidirectional communication and collaboration, it also allows the architect to mentor and coach developers.

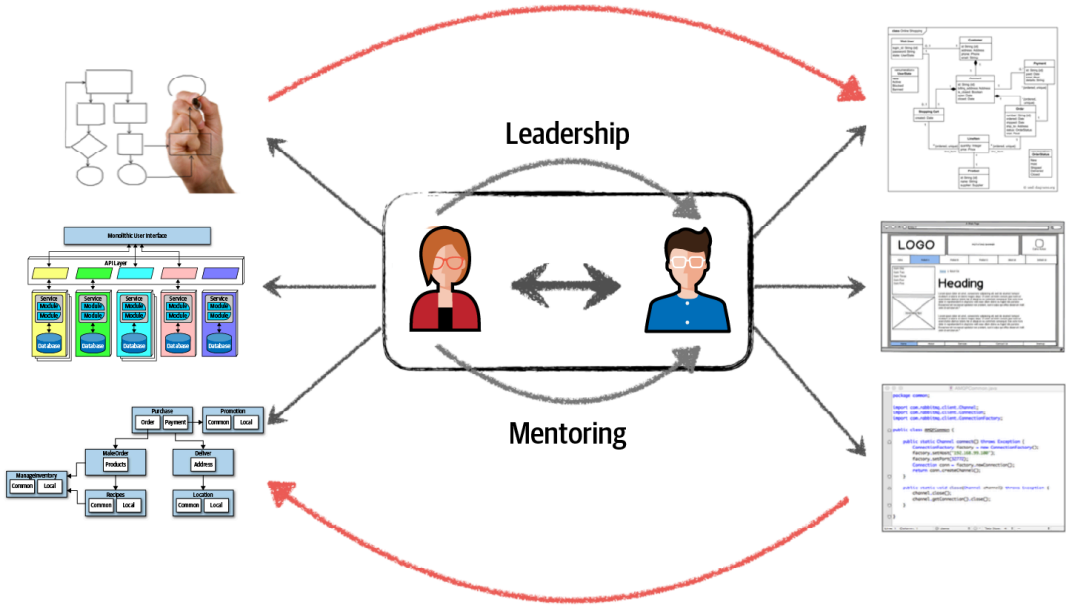

Figure 24-2. Making architecture work through collaboration

Unlike with the static, rigid old-school waterfall approaches, today's software architectures change and evolve with nearly every iteration or phase of a product effort. Tight collaboration between the architect and the development team is essential for success.

In the rest of this chapter and in Chapter 25, we'll show you techniques for forming the healthy, bidirectional, collaborative relationships that not only make development teams more effective, but create more robust and successful architectures.

Constraints and Boundaries

It's been our experience that a software architect can significantly influence the success or failure of a development team. Teams that feel left out of the loop or estranged from their architects often lack knowledge about various constraints on the system; without the right level of guidance, they struggle to implement the architecture correctly.

One of the roles of a software architect is to create and communicate the constraints within which developers must implement the architecture. Think of these constraints as forming a "room" in which the development team works to implement the architecture. As Figure 24-3 demonstrates, having boundaries that are too tight or too loose directly hampers teams' ability to successfully implement the architecture.

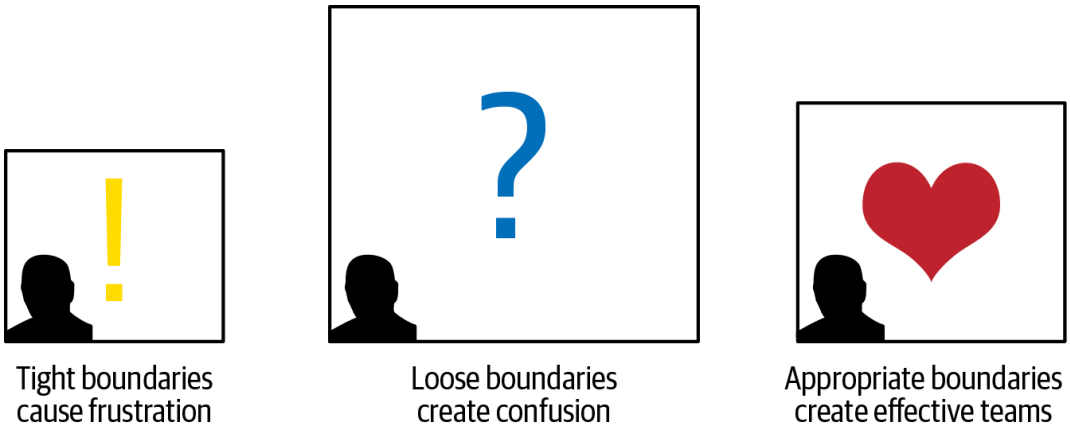

Figure 24-3. The boundaries a software architect creates affect the team's ability to implement the architecture

Too many constraints make the room too small for the development team, preventing them from accessing many of the tools, libraries, and practices they need to implement the system. This causes frustration, and usually results in developers leaving the project for happier and healthier environments.

The converse can also happen. Constraints that are too loose (or having no constraints at all) make the room too big. In this case there are too many choices available, forcing the development team to essentially take on the role of the architect to make all of the important architectural decisions. Lacking proper guidance, the developers find themselves performing too many proofs of concept, struggling over design decisions, and becoming unproductive, confused, and frustrated.

Effective software architects strive to provide the right level of guidance and appropriate constraints so that the team has everything it needs. The rest of this chapter is devoted to showing how to create these appropriate boundaries.

Architect Personalities

We're going to generalize wildly for the sake of conceptual clarity here, and tell you that architects' personalities come in three basic types: the *control-freak architect*, the *armchair architect*, and the *effective architect*. Control-freak architects tend to produce tight boundaries, armchair architects tend to produce loose boundaries, and effective architects produce appropriate boundaries. The following sections describe the details of each architecture personality.

The Control-Freak Architect

The control-freak architect tries to control every detail of the software development process. Every decision they make is usually too fine-grained and too low-level, resulting in tight boundaries and too many constraints on the development team.

For example, a control-freak architect might restrict the development team from downloading useful or even necessary open source or third-party libraries, or place tight restrictions on naming conventions, class designs, method lengths, and so on. They might go so far as to write pseudocode for the development teams to implement, essentially stealing the art of programming away from the developers. Developers find this frustrating and often lose respect for the architect.

Unfortunately, it's very easy to become a control-freak architect, particularly when you're transitioning into architecture from a software developer role. Architects' role is to create the building blocks of the application (the logical components) and determine how they all interact. The developers' corresponding role is to determine how best to implement those logical components, using class diagrams and design patterns. New architects, being used to creating the class diagrams and selecting the design patterns themselves as developers, often find this temptation difficult to resist.

For example, suppose an architect creates a logical component that manages the reference data within the system: things like the static name-value pair data used on the website, product codes, and warehouse codes. The architect's role is to identify the logical component (in this case, a `Reference Manager`), determine its core set of operations (for example, `GetData`, `SetData`, `ReloadCache`, and `NotifyOnUpdate`), and identify which other components need to interact with the `ReferenceManager`. A control-freak architect might think that the best way to implement this component is through a parallel loader pattern, leveraging an internal cache with a particular data structure. This might be an effective design, but it's not the only design, and, more importantly, it's not the architect's job to come up with an internal design for the `Reference Manager`—that's the developer's job.

As we'll talk about in this chapter, sometimes architects need to play the role of control freak, depending on the complexity of the project and the team's skill level. However, most of the time, a control-freak architect disrupts the development team, doesn't provide the right level of guidance, gets in the way, and is generally ineffective as a leader.

The Armchair Architect

The armchair architect is an architect who hasn't coded in a very long time (if ever) and doesn't account for implementation details when creating an architecture. They are typically disconnected from the development team and are rarely around, simply moving on to the next project after completing the initial architecture diagrams.

Some armchair architects are simply in way over their heads: they just don't know the technology or business domain well enough to provide leadership or guidance. Think about it: what do developers do? They write source code, of course. Writing source code is really hard to fake; either you can write source code or you can't. What does an architect do? No one knows! Draw lots of lines and boxes? It's all too easy to fake it as an architect.

For example, suppose an armchair architect designing a stock-trading system is in way over their head. Their architecture diagram might only contain two boxes: one representing the trading system, and one representing the trade compliance engine it communicates with. There's nothing *wrong* with this architecture—it's just too high-level to be of any use to anyone.

Armchair architects create loose boundaries around their development teams, so those teams end up doing the work the architect is supposed to be doing. Their velocity and productivity suffer as a result, and everyone gets confused about how the system should work.

It's as easy to become an armchair architect as it is to become a control freak. When an architect finds that they don't have time for the development teams that are implementing the architecture (or simply choosing not to spend time with them), that's an indicator that they might be falling into the armchair-architect

personality. Development teams need an architect's support and guidance, and they need the architect to be available to answer questions. Other indicators of an armchair architect are the following:

- Not fully understanding the business domain, business problem, or technology being used
- Not enough hands-on experience developing software
- Not considering the implications associated with a certain implementation of the architecture solution (such as complexity, maintenance, and testing)

Few architects *intend* to become armchair architects; it just "happens" when they get spread too thin between projects or teams and lose touch with the technology or the business domain. To avoid this, we recommend getting more involved in the project's technologies and building a stronger understanding of the business problem and domain.

The Effective Architect

An *effective* software architect creates appropriate constraints and boundaries, ensures that team members are working well together, and provides them with the right level of guidance. The effective architect also makes sure that the team has the correct tools and technologies in place and removes any other roadblocks between the development team and its goals.

While this sounds obvious and easy, it's not. There is an art to becoming an effective *leader* as well as an effective software architect. It requires collaborating closely with the development team and gaining their respect. In the following sections, we'll show you some techniques for determining how involved an architect should be with a development team.

How Much Involvement?

Becoming an effective architect is about knowing how much to be involved with a given development team—and when to stay out of their way. This concept, known as Elastic Leadership, is widely evangelized by author and consultant Roy Osherove. We're going to deviate a bit from Osherove's work in this area to focus on factors that are specific to software-architecture leadership.

Knowing how much to be involved with a development team can be challenging, and so is determining how many teams or projects you can manage at once. Here are five key factors to consider:

Team familiarity

> How well do the team members know each other? Have they worked together before? Generally, the better team members know each other, the more they can self-organize and the less they need the architect involved. Conversely, the newer the team members are, the more they'll need the architect to facilitate collaboration and reduce cliques.

Team size

> We consider more than a team with 12 developers to be a big team, and one with 5 or fewer to be a small team. The larger the team, the more the

architect is needed. We discuss this topic in more detail in "Team Warning Signs".

Overall experience

What is the team's mix of senior and junior (inexperienced) developers? How well do its members know the technology and the business domain? (If the business domain is particularly complex, consider assessing team members' overall level of technological experience separately from their business-domain experience.) Teams with lots of junior developers require more involvement and mentoring from the architect and more mentoring. In teams with more senior developers, the architect can become more of a facilitator than a mentor.

Project complexity

Highly complex projects require the architect to be more available to assist with issues, while relatively simple, straightforward projects require less involvement.

Project duration

Is the project short (say, two months), long (two years), or average duration (around six months)? The architect involvement needed grows with the length of the project.

While most of these factors might seem obvious, project duration is sometimes confusing. As we indicated, the shorter the project's duration, the less involvement is needed; the longer the project, the more involvement is needed. Does that seem counterintuitive? Consider a quick two-month project. Two months is not a lot of time to qualify requirements, experiment, develop code, test every scenario, and release into production. In this case, the architect should act more like an armchair architect; the development team already has a keen sense of urgency, and a control-freak architect would just get in the way and delay the project. Now think about a two-year project: the developers are more relaxed, not feeling a sense of urgency. They're likely to be planning vacations and taking long lunches. Thus, for longer duration projects the architect is needed to ensure that the project moves along on schedule and that the team accomplishes the most complex tasks first.

To illustrate how to use these factors to determine an appropriate level of involvement, assume a fixed scale of 20 points for each factor, as shown in Figure 24-4. The side of the scale with negative values indicates less involvement, ending in the extreme of the armchair architect. The positive values indicate more involvement, ending in the extreme of the control-freak architect.

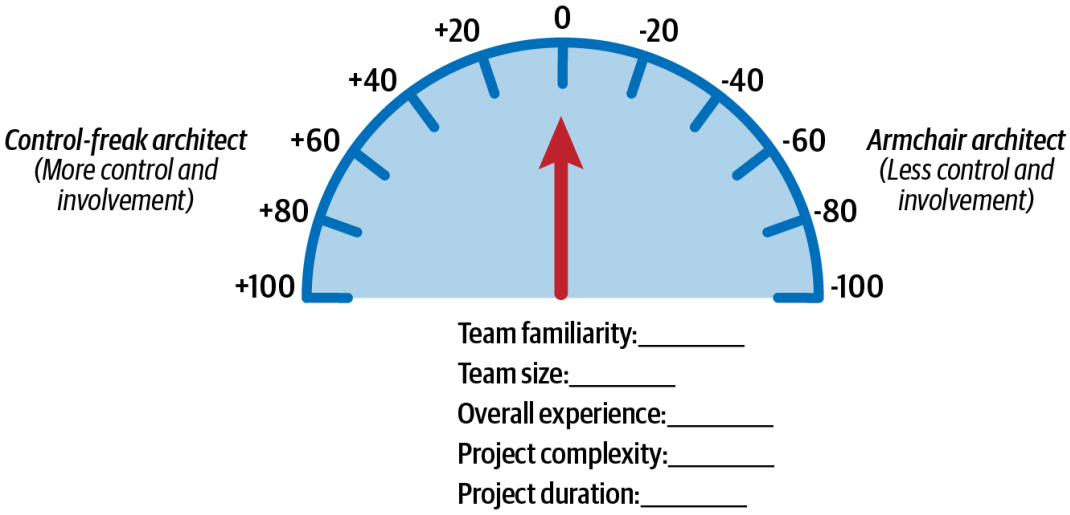

Figure 24-4. Scale for measuring architects' amount of involvement with development teams

This scale is not exact, of course, but it does help in determining the amount of involvement the architect should expect to have with a development team. For example, consider the project scenario, Scenario 1, shown in Table 24-1 and Figure 24-5. The values for each factor in the table move the "needle" toward either extreme: +20 for a factor that indicates more involvement, -20 for one that indicates less involvement. The factor scores rated here for Scenario 1 total -60, indicating that the architect should limit their involvement in daily interactions, facilitating but staying out of the team's way. They will be needed to answer questions and make sure the team is on track, but for the most part the architect should be largely hands-off and let the experienced team do what they do best—develop software quickly.

Table 24-1. Scenario 1 example for amount of involvement

Factor	Value	Rating	Personality
Team familiarity	New team members	+20	Control freak
Team size	Small (4 members)	-20	Armchair architect
Overall experience	All experienced	-20	Armchair architect
Project complexity	Relatively simple	-20	Armchair architect
Project duration	2 months	-20	Armchair architect
Accumulated score		-60	Armchair architect

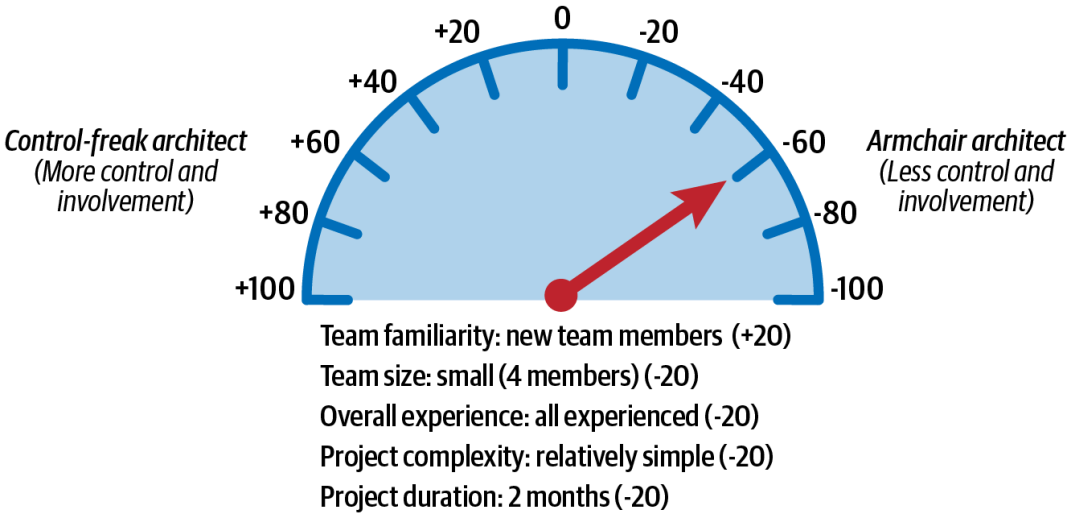

Figure 24-5. Amount of involvement for Scenario 1

Now consider Scenario 2, described in Table 24-2 and illustrated in Figure 24-6, where the team members know each other well, but the team is large (12 team members) and consists mostly of junior developers. The project is relatively complex, with a duration of six months. In this case, the accumulated score comes out to +20, indicating that the effective architect should take on a mentoring and coaching role and be fairly involved in day-to-day activities, but not so much as to disrupt the team.

Table 24-2. Scenario 2 example for amount of involvement

Factor	Value	Rating	Personality
Team familiarity	Know each other well	-20	Armchair architect
Team size	Large (12 members)	+20	Control freak
Overall experience	Mostly junior	+20	Control freak
Project complexity	High complexity	+20	Control freak
Project duration	6 months	-20	Armchair architect
Accumulated score		+20	Control freak

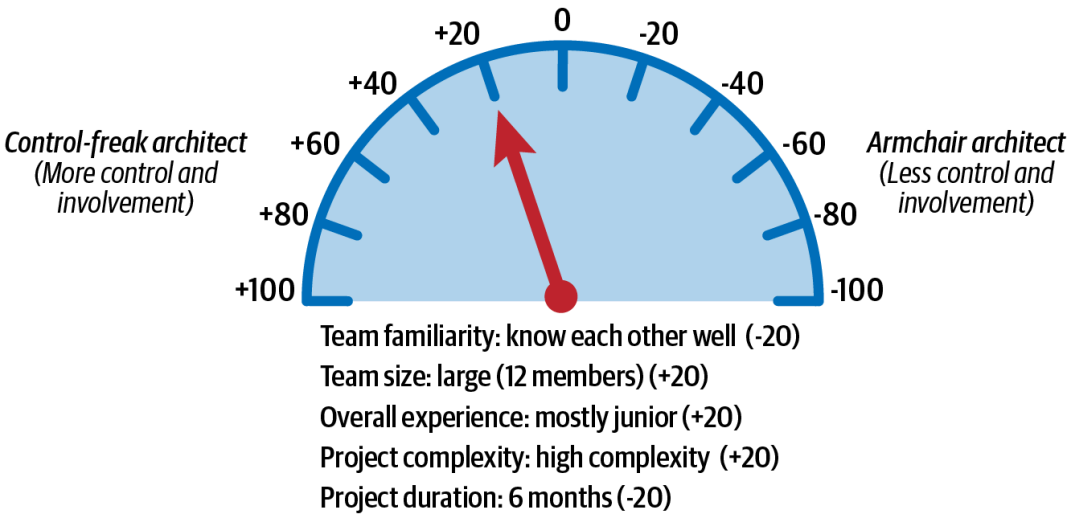

Figure 24-6. Amount of involvement for Scenario 2

Architects use these factors at the start of a project to determine how involved they plan to be, but as the project progresses, their level of involvement usually changes. We thus recommend analyzing these factors continually throughout the project's lifecycle.

It is difficult to make these factors objective, since some (such as the team's overall experience level) might carry more significance than others. In these cases the metrics can easily be weighted or modified to suit a particular situation.

The primary message here is that the appropriate amount of architect involvement on a development team varies according to these five factors. By using these factors to gauge their level of involvement, architects can draw the appropriate boundaries and create the right size "room" for the team.

Team Warning Signs

We mentioned team size as one of the five factors that help an architect determine the level of involvement on a development team: the larger a team, the more architect involvement is needed; the smaller the team, the less involvement is needed. However, what constitutes a "large team"? In this section, we'll look at

three factors that can help an architect determine if the team is too large, and hence not as effective as it can be.

Process Loss

The term *process loss* was coined by Fred Brooks in his book *The Mythical Man-Month* (Addison-Wesley, 1995). The basic idea of process loss, otherwise known as Brooks's Law, is that *the more people you add to a project, the more time the project will take*. As Figure 24-7 shows, *group potential* is defined by the collective efforts of everyone on the team. However, Brooks states that any team's *actual* productivity will always be less than its *potential* productivity; the difference is called the team's *process loss*.

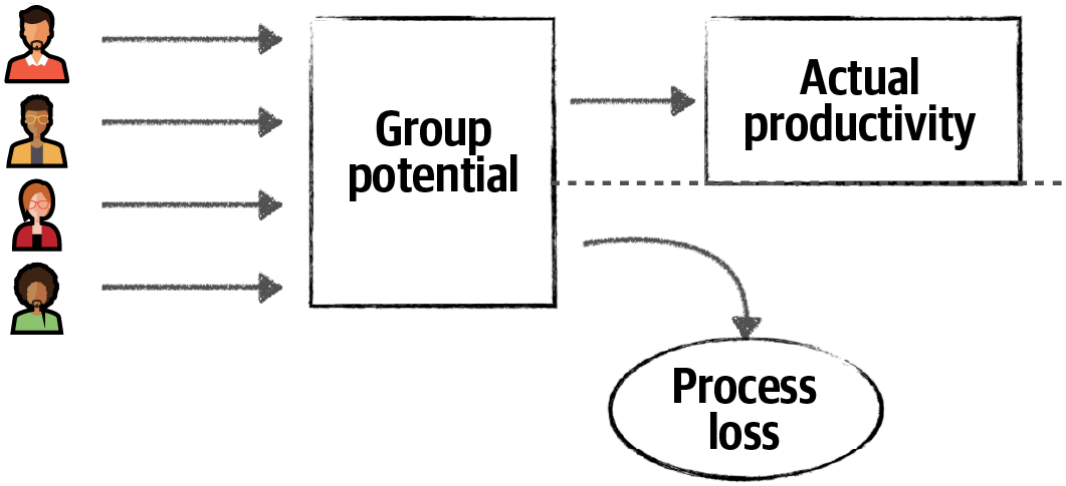

Figure 24-7. Brooks's Law holds that a team's size affects its actual productivity

Process loss is a good factor in determining the correct team size for a particular project, and an effective software architect observes the development team to look for indications of process loss. For instance, if team members frequently run into merge conflicts when pushing code to a repository, this is an indication that they are working on the same code and possibly getting in each other's way.

To avoid process loss, we recommend looking for areas of parallelism and having team members working on separate services or areas of the application. Anytime a project manager proposes adding a new team member to a project, an effective architect will look for opportunities to create parallel work streams. If they find none, they notify the project manager that a new addition could have a negative impact on the team.

Pluralistic Ignorance

Pluralistic ignorance is when everyone privately rejects a norm, but agrees to it because they think they are missing something obvious. For example, suppose the majority of people on a large team agree that using messaging between two remote services is the best solution. One person thinks this is a silly idea, because of a secure firewall between the two services. However, that person publicly agrees along with everyone else to use messaging. Although they privately reject the idea, they are afraid that they might be missing something obvious if they speak up. The larger the group, the less willing people are to confront others. On a smaller team, they might have spoken up to challenge the original solution, prompting the team to land on another protocol (such as REST) for a better solution.

The concept of pluralistic ignorance was made famous by the Danish children's story "The Emperor's New Clothes", by Hans Christian Andersen. In the story, two con artists, posing as tailors, convince the king that the new clothes they've "made" for him are invisible to anyone who is unworthy to see them. The king, who can't see the clothes but is unwilling to admit that he himself must be unworthy, struts around totally nude, asking all of his subjects how they like his new clothes. The subjects, afraid of being considered unworthy, assure the king that his new clothes are the best thing ever. This folly continues until a child finally calls out to the king that he isn't wearing any clothes at all.

During meetings, an effective software architect observes people's facial expressions and body language, alert for the possibility that some might mask their skepticism because of pluralistic ignorance. If the architect senses this happening, they act as a facilitator—perhaps interrupting to ask a skeptic what they think about the proposed solution and supporting them when they speak up, even if the person is wrong. The point here is for the architect, as a facilitator, to make sure everyone feels it is a safe enough environment to speak up.

The third factor that indicates appropriate team size is called *diffusion of responsibility*. As teams get bigger, their growth has a negative impact on communication. If team members are confused about who is responsible for what and things are getting dropped, those are good signs that the team is too large.

Figure 24-8 shows someone standing next to a broken-down car on the side of a country road. In this scenario, how many people might stop and ask the motorist if everything is OK? Because it's a small road, it's probably in a small community, so maybe everyone who passes by will stop. But what if that same motorist is stranded on the side of a busy highway in a large city? Thousands of cars might simply drive by without anyone stopping to ask if everything is OK. This is a good example of the diffusion of responsibility. As cities get busier and more crowded, people assume the motorist has already called for help, or that some other member of the crowd witnessing the event will help. However, in most of these cases, help is not on the way, and the motorist is stuck with a dead or forgotten cell phone.

Figure 24-8. Diffusion of responsibility

An effective architect not only guides the development team through implementing the architecture, but ensures that its members are healthy, happy, and working together to achieve a common goal. Looking for these three warning

signs and helping to correct the problems they indicate is a good way to ensure an effective development team.

Leveraging Checklists

Airline pilots use checklists on every flight. Even the most experienced, seasoned veteran pilots have checklists for takeoff, landing, and thousands of other situations—both common ones and unusual edge cases. They use checklists because one missed aircraft setting or procedure (such as forgetting to set the flaps to 10 degrees before takeoff) can mean the difference between a safe flight and a disaster.

In Dr. Atul Gawande's excellent book *The Checklist Manifesto* (Picador, 2011), he describes the power of checklists to make surgical procedures safer. Alarmed at the high rate of staph infections in hospitals, Dr. Gawande created surgical checklists. Infection rates in hospitals using the checklists fell to near zero, while rates in control hospitals not using the checklists continued to rise.

Checklists work. They're excellent vehicles for making sure every task is covered and addressed. So why doesn't the software development industry leverage them too? Having worked for many years in this industry, we firmly believe that checklists make a big difference in the effectiveness of development teams. Of course, most software developers aren't handling life-and-death matters like flying airliners or performing open-heart surgery. In other words, software developers don't require checklists for everything. The key is knowing when to leverage them and when not to.

Figure 24-9 is not a checklist—it's a set of procedural steps for creating a new database table, so it should not be in checklist form. Some of its tasks have dependencies; for example, the database table cannot be verified if the form has not yet been submitted. Any process with a procedural flow of dependent tasks should not be in a checklist. Nor should simple, familiar processes that are executed frequently without error.

Done	Task description
☐	Determine database column field names and types
☐	Fill out database table request form
☐	Obtain permission for new database table
☐	Submit request form to database group
☐	Verify table once created

Figure 24-9. Example of a bad checklist

Good candidates for checklists include processes that don't have a set procedural order or dependent tasks, as well as those where people frequently skip steps or make errors. Don't go overboard and start making everything a checklist. Architects often do this once they find that checklists do, in fact, make development teams more effective. They risk invoking what is known as the *Law of Diminishing Returns*. The more checklists the architect creates, the less likely developers are to use them. It's also wise to make checklists as small as possible while still capturing all the necessary steps. Developers generally will not follow overly long checklists. If any of the tasks listed can be automated, automate them and remove them from the checklist.

Note

Don’t worry about stating the obvious in a checklist. The obvious stuff is what usually gets missed.

Three of the checklists we’ve found most helpful cover developer code completion, unit and functional testing, and software releases. Each checklist is discussed in the following sections.

The Hawthorne Effect

The hardest part of introducing checklists to a development team is getting developers to actually use them. It’s all too common for some developers to run out of time and simply check off all the items in a checklist without actually performing the tasks.

One way to address this issue is by talking with the team about the difference using checklists can make and having them read *The Checklist Manifesto* by Atul Gawande. Make sure each team member understands the reasoning behind each checklist. Consider having them decide collaboratively what procedures should and shouldn’t be on a checklist—creating a sense of ownership also helps.

When all else fails, there’s the *Hawthorne effect*: the tendency for people who know they are being observed or monitored to change their behavior, generally to do the right thing. This effect doesn’t require actual monitoring so much as a perception; for example, many employers mount non-functioning cameras in highly visible areas, while others install website monitoring software that they rarely check. (How many of those reports are actually viewed by managers?)

To use the Hawthorne effect to govern checklists, let the team know that because using checklists is critical to the team’s productivity, all checklists will be verified to make sure the task was actually performed. In reality, occasional spot-checks are all that’s needed; developers will be much less likely to skip items or falsely mark them as completed.

Developer Code-Completion Checklist

The developer code-completion checklist is a useful tool, particularly when a developer states that they are “done” with the code. It also is useful for arriving at a “definition of done”: if everything in the checklist is complete, the developer can say they are actually done with the code they were working on.

Here are some things to include in a developer code-completion checklist:

- Coding and formatting standards not included in automated tools
- Frequently overlooked items (such as absorbed exceptions)
- Project-specific standards
- Special team instructions or procedures

Figure 24-10 shows an example developer code-completion checklist. There are some obvious tasks here, like “Run code cleanup and code formatting” and “Make sure there are no absorbed exceptions.” How often do developers in a hurry forget to run code cleanup and formatting from the IDE? Plenty often. In *The Checklist*

Manifesto, Gawande finds the same phenomenon with respect to surgical procedures—the obvious tasks are often the ones missed.

Done	Task description
☐	Run code cleanup and code formatting
☐	Execute custom source validation tool
☐	Verify the audit log is written for all updates
☐	Make sure there are no absorbed exceptions
☐	Check for hardcoded values and convert to constants
☐	Verify that only public methods are calling setFailure()
☐	Include @ServiceEntrypoint on service API class

Figure 24-10. Example of a developer code-completion checklist

The architect should always review the checklist to see if any items can be automated or written as plug-ins for a code-validation checker. While the project-specific tasks in the checklist (such as executing the custom validator, verifying the audit log is written, calling the `setFailure()` method, and including the `@ServiceEntrypoint` annotation) are good to have in a checklist, some of these could be automated. For example, while an automated check might not be feasible for "Include `@ServiceEntrypoint` on service API class," it would work for "Verify that only public methods are calling `setFailure()`"—a straightforward check to automate with any code-crawling tool. Checking for areas of automation shortens the checklist's size and improves its ratio of signal to noise.

Unit and Functional Testing Checklist

Perhaps one of the best checklists is for unit and functional testing. This checklist contains some of the more unusual and edge cases that software developers tend to forget to test. Whenever someone from QA finds a code issue based on a particular test case, add that test case to this checklist.

This particular checklist is usually one of the longest, since it encompasses all the types of tests that can be run against code. Its purpose is to ensure the most complete testing possible so that when the developer is done with the checklist, the code is essentially production ready.

Here are some of the items found in a typical unit and functional testing checklist:

- Special characters in text and numeric fields
- Minimum and maximum value ranges
- Unusual and extreme test cases
- Missing fields

As with the developer code-completion checklist, if any items can be written as automated tests or are already included in the automated test suite, they should be removed from the checklist and automated.

Developers sometimes don't know where to start when writing unit tests or how many they should write. This checklist provides a way to make sure general and

specific test scenarios are included in the development process. In organizations where testing and development are performed by separate teams, this checklist helps to bridge the gap. The more development teams perform complete testing, the easier they make the testing teams' jobs, freeing them up to focus on business scenarios not covered in the checklists.

Software-Release Checklist

Releasing software into production is perhaps one of the most error-prone points in the software development lifecycle, so it makes for a great checklist. This checklist helps avoid failed builds and deployments, and significantly reduces the risk associated with releasing software.

The software-release checklist is usually the most volatile of the checklists presented here, because it changes each time a deployment fails or has problems, to address new errors and shifting circumstances.

The software-release checklist typically includes:

- Configuration changes in servers or external configuration servers
- Third-party libraries added to the project (JAR, DLL, etc.)
- Database updates and corresponding database-migration scripts

Anytime a build or deployment fails, the architect should analyze the root cause of the failure and add a corresponding entry to the software-release checklist. This way, the item will be verified during the next build or deployment, preventing the problem from happening again.

Providing Guidance

Another way software architects can make teams effective is by using design principles to provide guidance. This also helps form the constraint "room" in which developers work to implement the architecture. Communicating design principles is one of the keys to creating a successful team.

To illustrate this point, imagine you're guiding a development team in using what is typically called the *layered stack*—the collection of third-party libraries that make up the application. Development teams usually have lots of questions about the layered stack, including which libraries are OK, which are not, and whether or when they can make their own decisions about libraries.

You might begin by having the developers answer the following questions about a library they want to use:

- Are there any overlaps between the proposed library and the system's existing functionality?
- What is the justification for using the proposed library?

The first question guides developers to check if the functionality provided by the new library can be satisfied through an existing library or existing functionality. When developers ignore this activity (which sometimes happens), they can end up creating lots of duplicate functionality, particularly in large projects and teams.

The second question prompts the developer to ask if the new library or functionality is truly needed. We recommend asking for both a technical justification and a business justification, in part because this technique helps make developers aware of the need to provide business justifications.

The Impact of Business Justifications

One of your authors was the lead architect on a particularly complex Java-based project with a large development team. One team member was obsessed with the Scala programming language and desperately wanted to use it on the project. Their desire to use Scala ended up becoming so disruptive that two key team members declared their intention to leave the project for other, “less toxic” environments. Your author convinced them to hold off. Then he told the Scala enthusiast that he would support using Scala within the project *if* the enthusiast provided a business justification for the costs of the training and rewriting involved. The Scala enthusiast was ecstatic and said they would get right on it. They left the meeting yelling, “Thank you—you’re the best!”

The next day, the Scala enthusiast came into the office completely transformed and asked to speak with your author. They began by immediately (and humbly) saying, “Thank you.” The Scala enthusiast explained that they had come up with all the technical reasons in the world to use Scala, but none of those technical advantages had any business value in terms of the cost, budget, and timeline. In fact, the Scala enthusiast had realized two things: first, that the increase in cost, budget, and timeline would provide no benefit whatsoever; and, second, that they’d been disrupting the team. Before long, the Scala enthusiast had transformed into one of its best and most helpful members. Being asked to provide a business justification for something they wanted increased their awareness of the business’s needs, making them a better software developer and making the team stronger and healthier. The two key developers who’d been planning on leaving stayed on the team.

Another good way to communicate design principles is through graphical explanations about which decisions the development team can make and which they can’t. The graphic in Figure 24-11 shows what this might look like for controlling the layered stack.

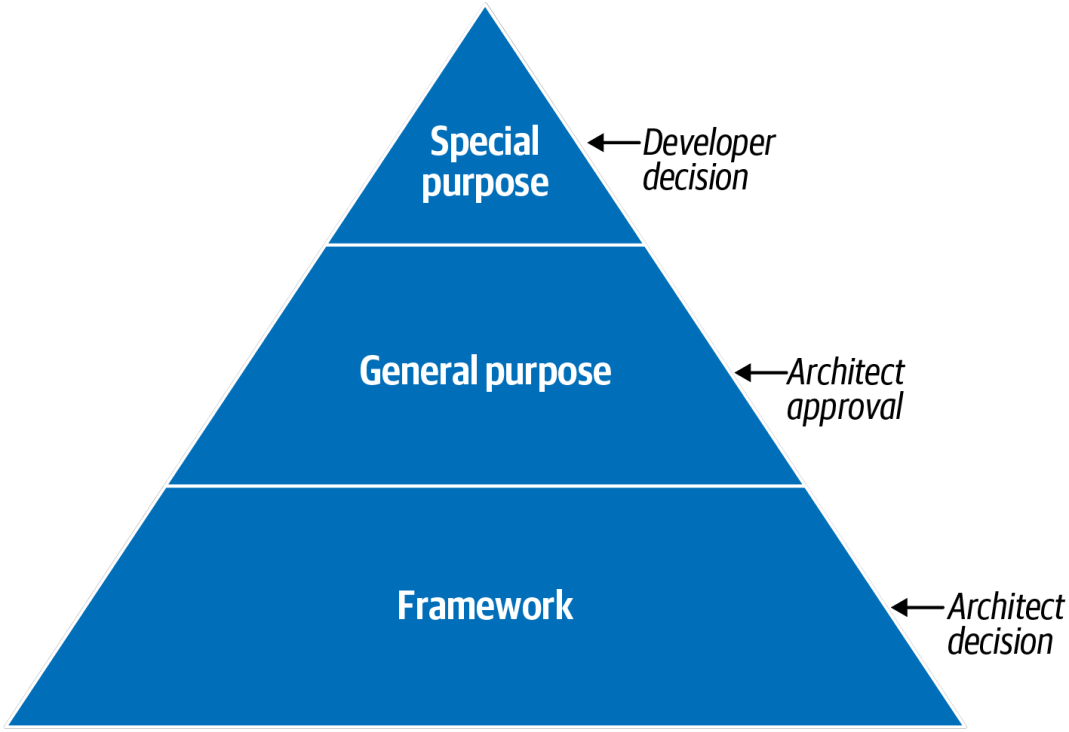

Figure 24-11. Providing guidance for the layered stack

These categories are only examples, but many more can be defined. In presenting Figure 24-11 to the team, the architect should apply each category to the third-party library proposed for use:

Special purpose

> These are specific libraries used for things like rendering PDFs, scanning barcodes, and other circumstances that do not warrant writing custom software.

General purpose

> These libraries are wrappers on top of the language API and include things like Apache Commons and Guava for Java.

Framework

> These libraries are used for things like persistence (such as Hibernate) and inversion of control (such as Spring). In other words, they make up an entire layer or structure of the application and are highly invasive.

Next, the architect creates the “room” around this design principle. Notice, in Figure 24-11, that the architect has authorized developers to make decisions about special-purpose libraries without consulting the architect. For general-purpose libraries, however, while developers can analyze overlap analysis, provide justifications, and make a recommendation, this category of library requires architect approval. Finally, framework libraries are entirely the architect’s responsibility—the development teams shouldn’t even perform analysis for these types of libraries.

Summary

Making development teams effective is hard work. It requires lots of experience and practice, as well as strong people skills (which we will discuss in subsequent chapters). That said, the simple techniques in this chapter for elastic leadership, leveraging checklists, and providing guidance by communicating design principles do in fact work. We've seen how useful they prove to be in making development teams work more intelligently.

Some question architects' role in such activities, insisting that this work should be assigned to the development manager or project manager. We strongly disagree. Software architects guide the team on technical matters and lead them through implementing the architecture. Building a close collaborative relationship with a development team allows the architect to observe team dynamics and facilitate changes to make the team more productive.