F O U R

Rhythm: Assuring Beat, Process, and Movement

Rhythm is one of the principal translators between dream and reality. Rhythm might be described as, to the world of sound, what light is to the world of sight.
—Edith Sitwell in Taken Care Of *[Sitwell65]*

Overview

Grady Booch points out in *Object Solutions* that the iterative nature of object-oriented development makes rhythm a critical element for success. He writes that having a rhythm forces closure at periodic intervals, coordinates supporting activities, and helps organizations react better when problems arise [Booch96]. Rhythm is important to any development process, but especially for architecture-based development. Rhythm can battle complexity, keep competition off-balance, and maintain sanity and predictability for architecture and development teams.

The sharing of an architecture is like an improvisational jazz ensemble. Each player in an ensemble is autonomous, but each musician's performance is coordinated by cues exchanged with the other musicians as well as the tempo, key, and style of the performance. While the basic elements of the performance may be written down or planned, many elements are performed by the musicians relying on their instinct, training, and talent.

Just as a jazz combo must share a common tempo, phrasing, and progression to have a rhythm, an architecture team must share work products with predictable timing, content, and quality. Software architectures are developed and used in many different organizations. Since many of these groups are autonomous, it is not possible to fully coordinate all of them from the top down. Without rhythm, sharing an architecture can befuddle even the best-designed schemes for communicating across teams.

Rhythm provides a temporal framework that allows groups sharing an architecture to synchronize activities and expectations. With rhythm, stakeholders know when and on which activities to focus. Not only can organizations with rhythm coordinate planned activities, but they can also coordinate those tasks that do not show up on plans because they are performed by other organizations or are not visible enough to be included in the planning process. When rhythm is weak, dissonance between organizations emerges, paving the road to architecture breakdown.

RHYTHM DEFINITION

Rhythm is the recurring, predictable exchange of work products within an architecture group and across their customers and suppliers.

There are three elements of rhythm: tempo, content, and quality (see Figure 4.1). As in music, architecture rhythm is not just the repetition of a beat. Effective rhythm enables teams throughout the organization to coordinate explicit and complex activities without the corresponding load of communication and coordination. If tempo, content, or quality is lacking, these benefits will not be realized, and progress will not be made.

Figure 4.1
There are three elements of rhythm: Tempo, content, and quality.

Tempo

Tempo is the frequency with which the same type of handoff occurs between one group and another—for example, between the architecture team and product development engineers. The more predictable the timing of each handoff becomes, the easier each transition is to manage. As illustrated by Figure 4.2, there may be many different tempos. Some organizations have different intervals for major releases, minor releases, and bug fixes. DAILY BUILD AND SMOKE TEST is one example of this notion of tempo [Cabrera99]. Microsoft has popularized this practice [Cusumano95][McCarthy95][McConnell96]. Regular release schedules are another example of tempo.

Content

It is not enough to maintain a beat of daily builds if the builds are not used.

Content is the delivery of value from one group to another. An example of the delivery of content is when a group develops a new or modified feature that another group uses to fill a need. Moving completed builds from development to testing is another example of the delivery of content. Content requires that the receiving group derives value from the delivery. For example, it is not enough to maintain a beat of daily builds if the builds are not used. Figure 4.3 illustrates an organization that maintains a regular tempo without delivering enough content with each beat. In this situation, stakeholders tune out the rhythm because so little progress is made with each beat. Iterative development, when working effectively, exemplifies content delivery as illustrated in Figure 4.4. Each iteration builds on the previous cycle. Figure 4.5 describes the situation in which value is added from each handoff, but because there is

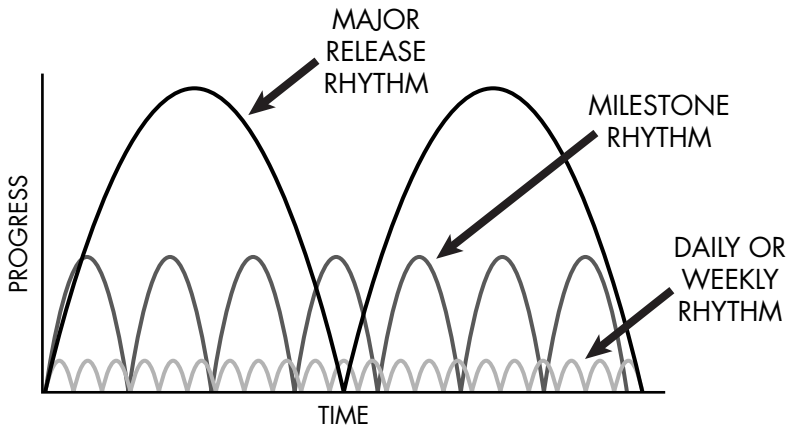


Figure 4.2 There may be rhythms of many different tempos at once.

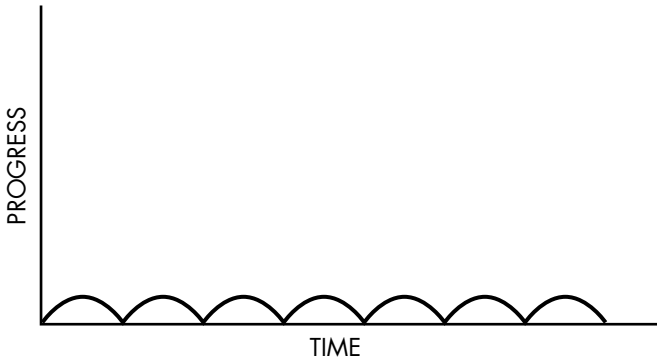


Figure 4.3 Regular tempo with little content.

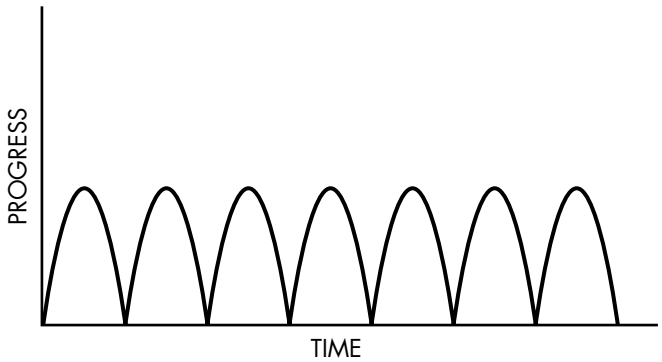


Figure 4.4 The tempo of effective iterative development.

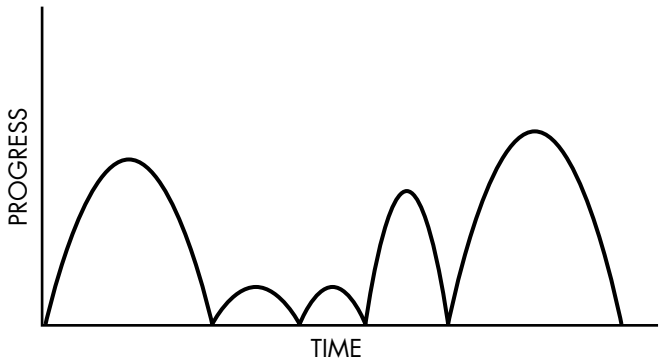


Figure 4.5 Content without a regular tempo.

no regular tempo, the timing of these handoffs is irregular. In this situation, rhythm is lacking because participants cannot anticipate when handoffs are going to occur.

Quality

Quality means that processes are followed to ensure that the architecture is free of deficiencies.[1] Organizations sometimes try to sustain their tempo by compromising on quality, for example, by skimping on testing, or by redefining what is required by a handoff. This situation is illustrated in Figure 4.6. Organizations may be able to accelerate their tempo by eliminating steps that do not add value, but if essential processes are truncated, rhythm will break down.

Consider the following example of the deterioration of quality. A development group was trying to achieve a goal of reaching an established milestone every three months. It became clear that the group was not going to reach a particular milestone, so they redefined the criteria for passing a milestone to maintain their schedule. They reclassified the severity of a number of outstanding defects. In this case, the process was abbreviated and the product was able to pass the milestone with lower quality. Additional effort was needed to improve the quality before the product reached the next milestone. Even though beat was maintained for the initial milestone, it just postponed the breakdown until a later milestone.

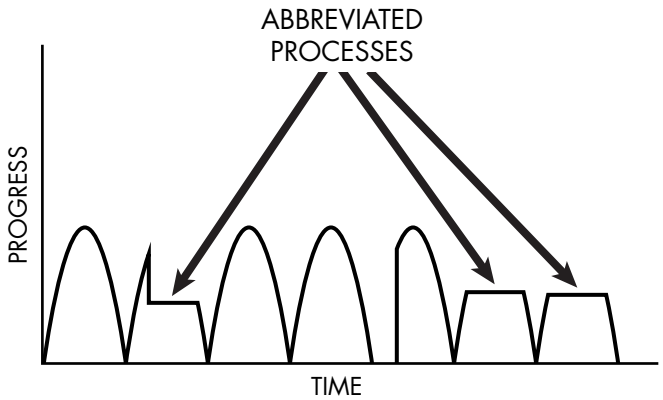


Figure 4.6 Incomplete rhythm: Truncated process to maintain a tempo

[1] This definition is based on the American Society for Quality's definition of the term quality. [ASQ00]

MOTIVATION

Why is rhythm so important to software architecture? Since software architectures are developed and used across many organizational boundaries, rhythm provides a stabilizing force that coordinates activities within and across teams and organizations. Like an improvisational jazz ensemble, architecture stakeholders need to be able to anticipate the activities of the other stakeholders. If stakeholders cannot plan activities and budget resources, then each transition requires a great deal of effort and time for communication and coordination among the involved parties. Maintaining a rhythm over multiple releases also strengthens the credibility of the architecture supplier.

Our interviews with Allaire Corporation demonstrated how this ability to anticipate enabled the organization to act quickly and efficiently. When Allaire experienced growth, managers knew when in their release cycle to hire testers, when to hire developers, and when to hire customer support personnel. Managers also knew what training to provide and when. Shortages or oversupply of a skill set were identified as a sure sign that rhythm was breaking down.

Rhythm Aids Transition Management

When rhythm is strong, stakeholders build strong skills that enable them to anticipate and execute transitions and handoffs. Organizations are then able to treat transitions as a recurring, regularly planned activity. When rhythm is not strong, transitions and handoffs often come as a surprise. Kathleen Eisenhardt and Shona Brown point out that "because major transitions are periods when companies are likely to stumble, we expected to find that managers would devote extra attention to them. The surprise is that they don't"[2] [Eisenhardt98].

Rhythm Drives Closure

Rhythm also helps an organization bring activities to closure. "Iterative and incremental releases," writes Booch, "serve as a forcing function that drives the development team to closure at regular intervals" [Booch96]. A study by Connie Gersick illustrates this notion. She observed project groups from six organizations. She found that even though the studied projects ranged from several days to several months, every group exhibited a distinctive approach to its task when it commenced and remained with that approach "until precisely halfway through the group's allotted duration." At the

[2] K. Eisenhardt, S. Brown, "Time Pacing: Competing in Markets that Won't Stand Still." *Harvard Business Review* (March–April, 1998).

Figure 4.7
Halfway to a deadline, teams typically adjust their approach and make dramatic progress

halfway point, she observed that the groups "dropped old patterns, reengaged with their outside supervisors, adopted new perspectives on their work, and made dramatic progress" [Gersick89] (see Figure 4.7). An organization with a good rhythm establishes regular intervals and halfway points to motivate this reassessment and progress.

USING RHYTHM TO TAKE CHARGE

Developers can use rhythm to take charge of their fates, even when their parent organization is bureaucratic. In one large, hierarchical information technology organization, a team built a component. Unlike most of the components owned by the parent organization, both the parent organization and customers in other chains-of-command used the team's component. The parent organization controlled the schedule. Component users could not count on timely delivery because release dates were tied to releases of the rest of the architecture, whose tempo was generally chaotic, as illustrated in Figure 4.8.

To resolve the situation, the component team began a regular release cycle, decoupling the release of the component from the release of the rest of the architecture. The component team was capable of delivering multiple releases for each release of the parent architecture. As a result, the component team was able to release more frequently because they cut the time required to deliver a release in half, as illustrated in Figure 4.9. Not only were customers outside the parent's chain-of-command pleased by the predictable schedules, but customers inside the parent organization found it easier to coordinate their releases. Testing became more predictable, quality

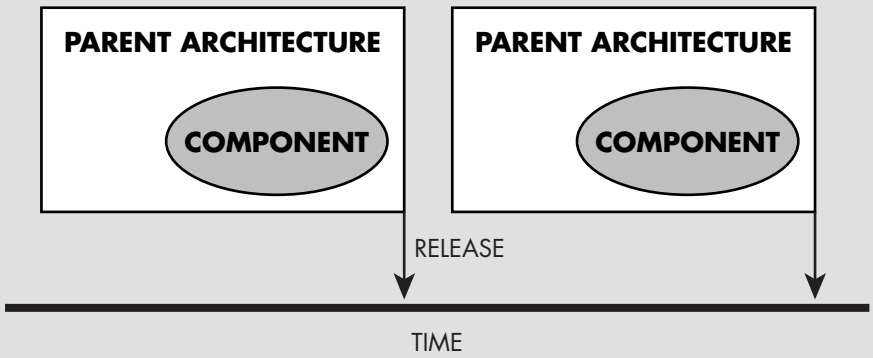


Figure 4.8 Before—The release cycle of the component is tied to the parent's release cycle.

improved, and customer satisfaction increased. In addition, the parent architecture release cycle became shorter and more regular.

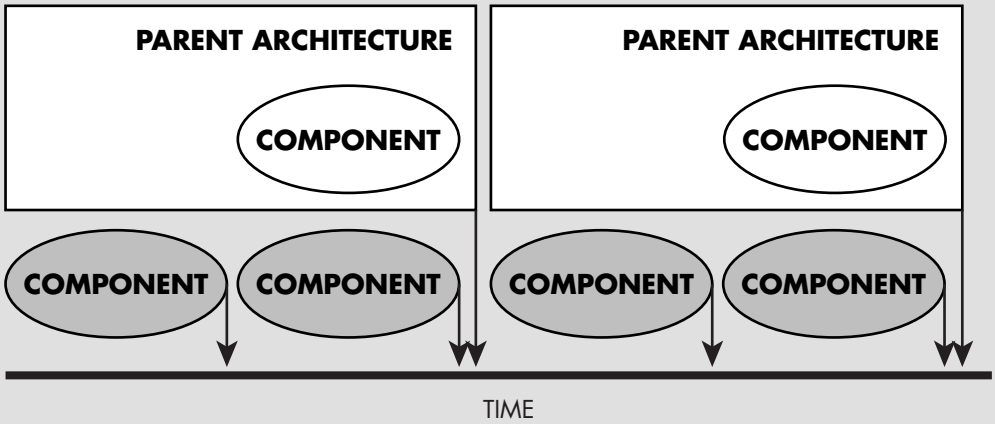
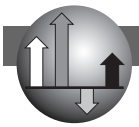


Figure 4.9 After—When the release cycle was decoupled, the component could be released more frequently.

PUTTING RHYTHM INTO PRACTICE: CRITERIA, ANTIPATTERNS, AND PATTERNS

The previous sections describe how the principle of rhythm is important for coordinating the activities of architecture stakeholders. The consequences of failing to establish a rhythm can lead to dissatisfied customers, unexpected defects, and components that do not work well together.

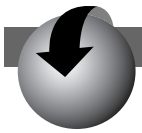


CRITERIA

The following criteria, patterns, and antipatterns provide guidelines and techniques that organizations can use to determine how well they establish a predictable execution of process, movement, and beat within an architecture group and across its customers and suppliers.

When rhythm is working…

1. Managers periodically reevaluate, synchronize, and adapt the architecture.
2. Architecture users have a high level of confidence in the timing and content of architecture releases.
3. Explicit activities are coordinated via rhythm.



ANTIPATTERNS

KILLER FEATURE is what happens when an organization becomes so focused on getting one feature to market that the internal rhythm is disrupted. Even if the feature is delivered, the organization may be blind-sided by competition because of the single-minded focus on getting the feature to market. It is an example of what happens when the management does not regularly reevaluate and adapt the architecture.

SHORT CUT can happen when the organization tries to maintain a regular beat of releases by taking short cuts in the organization's process. This antipattern compromises the quality and content that users expect from the architecture.

BROKEN LOADS can happen when an organization has tried to implement regular builds, but the builds frequently fail to compile or pass automated tests. This represents a breakdown of coordination.

PATTERNS

RELEASE COMMITTEE describes an approach for coordinating the parties involved in releasing a new architecture. The pattern illustrates a way for managers to reevaluate, synchronize, and adapt an architecture during the final stretch of an architecture's release.

DROP PASS examines how organizations can maintain a beat by moving less critical features to later release cycles. By maintaining the rhythm, this pattern gives users more confidence on the timing of architecture releases.

SYNCHRONIZE RELEASES is a technique for extending the notion of rhythm beyond an organization's boundaries. This pattern provides a way to synchronize the activities of the architecture team and their users.

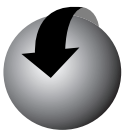
Table 4.1 illustrates how the remainder of this chapter is organized.

TABLE 4.1 *Mapping Criteria to Antipatterns and Patterns*

CRITERION— HOW YOU MEASURE	ANTIPATTERN— WHAT NOT TO DO	PATTERN— WHAT YOU CAN DO
1. Managers periodically reevaluate, synchronize, and adapt the architecture.	KILLER FEATURE	RELEASE COMMITTEE
2. Architecture users have a high level of confidence in the timing and content of architecture releases.	SHORT CUT	DROP PASS
3. Explicit activities are coordinated via rhythm.	BROKEN LOADS	SYNCHRONIZE RELEASES

Criterion 1: Managers periodically reevaluate, synchronize, and adapt the architecture.

A good rhythm needs a regular beat. From the perspective of management, this means shifting from *event-pacing* to *time-pacing*. Rather than making decisions when there is a change in competition, technology, or customers, decisions are made on a regular, calendar-driven schedule [BrownS98]. For managers of architecture organizations, this means that they must reevaluate, synchronize, and adapt their plans for the architecture at regular intervals. Time-pacing is effective as a strategy "because it forces managers to look up from their business on a regular basis, survey the situation, adapt, if necessary, and get back to work" [BrownS98].[3] The beat of the rhythm can also provide a framework for the planning process. Instead of asking "How long will it take to implement a feature?" an organization with a good rhythm can ask "How much of the feature can we do in a single beat?" or "How many beats will it take us to fully implement the feature?"



Antipattern: KILLER FEATURE

Alias: CLOSE TO EVEREST

General Form. You are an executive or architect and you are planning a new release that will be driven by a major, cutting edge feature. While drop-dead great features provide a serious competitive edge, focus on the impossible dream of one feature can literally kill a product or product line by

[3] S. Brown, K. Eisenhardt, "Competing on the Edge." Harvard Business School Press (1998).

destroying internal rhythm. You sincerely believe that if the team can just get this feature out, everything else, such as increased sales and marketshare, will fall into place. The team is driven to deliver this feature to the exclusion of others, often cutting corners on quality. Managers do not look up until the KILLER FEATURE is delivered or until they are blind-sided by their competitors or customers. When the release containing the feature is finally delivered, the team is exhausted. Worse, since everyone had been so focused on the feature, no one knows what to do next.

Forces. Customers and potential customers are not shy about tying leading-edge features to their commitment of continued business. These statements can lead entrepreneurs to conclude that if they could provide one single feature, they would gain a prized customer's business. While some key customers make decisions solely based on the presence or absence of one feature, other potential customers may be discouraged if there is no definition of or commitment to a broad array of features, or if the future direction of the architecture is not articulated clearly. A successful architecture is never finished, but is continually evolving to keep pace with changing customer needs and environments.

Solution. The key feature should be implemented as part of the team's rhythm, not in spite of it. A particular release may be focused around a particular theme, which may help the release to take advantage of opportunities in the marketplace. If the key feature is particularly complex, it may need to be implemented across several iterations. If it becomes difficult to maintain the rhythm while implementing the key feature, it is most likely a warning sign that the risk and complexity of the feature is greater than anticipated, and that replanning is necessary.

Rationale. A KILLER FEATURE can be like mountain climbers scaling Mount Everest. The climbers see the summit, and it appears easy to attain, but in reality the distance is an illusion—a hard and treacherous journey to the peak remains, and the oxygen becomes thinner with each step. Similarly, the drive for a KILLER FEATURE may seem like a short diversion for the team, but may expose the organization to a number of risks. Organizations with an effective rhythm benefit from the tacit coordination of many activities and groups. When an organization trades this rhythm for a drive to implement a KILLER FEATURE, this coordination will suffer. Further, successful software architectures can remain in use for a very long time, and it is impossible to forecast all of the turns in the marketplace over that period of time. While it is true that a killer feature may increase revenue or marketshare, it is also true that another brand-new technology could be around the corner that will completely reshape the marketplace. Without regular cycles to reevaluate and adapt the architecture these changes in environment might be missed. The

sponsors and senior management for a software architecture must periodically reevaluate and adapt the architecture to respond to these challenges.

Example. A major vendor of a very successful accounting and constituent management software product line hired a CEO whose primary mission was to take the company public. The CEO recognized that going public as an Internet company would result in a much higher valuation. However, changing the research and development (R&D) to focus on a KILLER FEATURE, a Web front-end integrated with the legacy product line, would pose a massive disruption of the company's long-established rhythm and release cycle. The company would have to establish a new division, manage and evolve new skills, and establish new interfaces to integrate technologies that did not exist when the disparate legacy technology was developed. It was a massive challenge.

What did the CEO do? He punted. Instead of investing in R&D for the required technology and setting an impossible schedule for delivery, he shopped to acquire an Internet company and settled on a seven-figure transaction with a proven Web-application development firm. The CEO said, "Time-to-market is everything." The company brought to market a credible Internet offering and stayed on track for its IPO.

Another leading software and hardware vendor applies a number of strategies to avoid this antipattern. One group uses an approach in which they split the time in the release into four roughly equal parts for requirements, design, implementation, and delivery. If the features for a release cannot fit into the time allotted for requirements, they re-plan the release based on the logic that if the requirements couldn't be completed in the allotted window, then the other planned windows would not be sufficient either. Another group in the same company takes this management of features in each release a step further. They plan more releases, some of which are developed in parallel, and partition the features between the releases. Each release is implemented more quickly, so the important features get to market more quickly. Candidate and Upcoming features are always in the pipeline, so the direction is clear to both the team and the customers.

When is This Antipattern a Pattern? Sometimes it makes sense to organize a release around a particular theme. This is particularly true when the feature enables users' products to move in a strategic direction. Some organizations use this theme as a way to prioritize which features are essential for a release and which are optional. In these cases, the essential features do receive more attention and resources, but they are still approached within the overall vision of the architecture.

Variation: LATE KILLER FEATURE. The worst case of this antipattern is the LATE KILLER FEATURE. In this case, the feature surfaces very late in the development cycle. Just when things are settling down and a release seems imminent, it suddenly becomes necessary to add something new to address "market conditions." These late additions are uniformly regrettable. Since they were not part of the vision for the cycle they do not fit into the architecture and are implemented as kludges. Even though the new feature seems critical, the long-term impact on the architecture and on the customers is harmful. Correcting mistakes added at these late stages leaves hanging the customers who bought into them.

Related Antipatterns and Patterns. This antipattern is similar to TUNNEL VISION (Chapter 5, Anticipation); in both cases, the organization is driven to a particular goal. Unlike TUNNEL VISION, where the direction is maintained in the face of contrary evidence, there may be no such evidence in KILLER FEATURE. The KILLER FEATURE antipattern may result even if the feature in question does indeed turn out to be very important to the marketplace.

Pattern: RELEASE COMMITTEE

Problem Statement. How do you get teams with conflicting perspectives and agendas in sync to meet a planned release date?

Context. A product organization with a dozen to several hundred people, including product support, marketing, architecture, testing, and design, is developing and supporting an architecture. These groups do not all report to the same manager. A configuration control board, or similar process, is in place and the organization has committed to a set of features for a particular release. The product involves several components that must be coordinated for customers to get the most value from it.

Forces. Many different groups within an organization are involved when releasing a version of an architecture. Participating groups may include development, marketing, testing, quality assurance, configuration management, and subcontractors. Groups that report to different managers may have conflicting, and often hidden, priorities and agendas. It can be particularly difficult, for example, for a manager to state the obvious when a positive change could threaten the size or viability of his or her group. Even when everything is arranged perfectly, group dynamics produce unexpected results [Smith87]. Larger groups take more time to make decisions than smaller groups. When one group is clearly behind, other groups may try to play SCHEDULE CHICKEN and take advantage of the delay and avoid blame. That is, groups may believe they can get away with risking a delay if they think another group will be even further delayed [Olson98].

Solution. Hold regular and formal meetings that include each critical stakeholder in the organization to guide the progress of the release. During the meetings, review changes in product features and priorities, so that product documentation, marketing promises, public relations, testing, and development are in agreement. Where appropriate, metrics should be used to measure the progress of the release. At these meetings, commitments and dependencies are shared, and decisions are made about how to proceed. Document and distribute the decisions that are made at the meetings. The members participating in the committee should be stable. There should be consistent membership at the meeting, and the participants also need to have enough authority to make decisions. The RELEASE COMMITTEE differs from a configuration control board in that it focuses on the execution of a release, whereas a configuration control board is typically concerned with the content of a release.

Result. As a result of the pattern, surprises are avoided, as are unnecessary delays. When issues arise, they can be fairly and adequately represented, and then quickly resolved. Because all stakeholders are represented, there is a better understanding of the context in which decisions are made. The use of metrics can help focus the discussion and make it easier for the participants to come to a common understanding on the progress of a release. The committee improves the timing of releases by enhancing coordination of the groups involved. The committee's decision-making process also improves the content and quality of releases, and it helps ensure that consistent information about the release is given to customers.

Consequences. The practice is time-consuming, especially if the meetings do not have clear agendas established and enforced. The REPRESENTATIVE RELEASE COMMITTEE variation addresses some of these concerns for large organizations. The meeting may provide a forum for some groups to air pet peeves and introduce other obstacles—including participants who are not essential for a successful release, but who can make the RELEASE COMMITTEE less efficient. Because the RELEASE COMMITTEE can exert a strong influence on the execution of a release plan, a CORNCOB (a curmudgeon of the unwelcome kind) may cause havoc as a member of the committee [BrownW98].

Rationale. Releases of large, complex software products typically involve a variety of groups that report to different managers. This makes lateral integration difficult, which can lead to risks, poor decisions, and delays. A RELEASE COMMITTEE provides a structure in which stakeholder groups work together to deliver an architecture or other major software component. This coordination can improve the delivery of content between the groups responsible for delivery. By avoiding missteps, the committee can also help maintain the tempo of a release. Stable membership is needed so that issues resolved

during one meeting will not resurface at later meetings. If participants do not have enough authority to make decisions, then decisions cannot be made at the meeting, or the decisions made will have no influence on the organization.

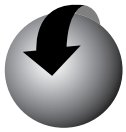
Example. This practice is used by a number of companies including Allaire, the maker of the ColdFusion Web application development environment. Allaire kicks off a new release with the stakeholders at an off-site meeting. A release plan is created at this meeting, and it ensures that all of the departments are in agreement regarding the priorities of the release. Then, as the release progresses, the group meets every week. These weekly meetings make commitments and dependencies visible to all of the participants. Changes are recorded in a release plan.

Variation: REPRESENTATIVE RELEASE COMMITTEE. If the organization is very large, it may not be feasible to have direct participation in the RELEASE COMMITTEE. A REPRESENTATIVE RELEASE COMMITTEE can be used instead. A telecommunications firm had a very large product that involved more than 20 distinct organizations. Release committee meetings to determine whether a checkpoint had been passed were bogged down when participants began to air pet peeves and push pet features and often surprised everyone by blocking releases with issues that had not been previously raised. They adopted a three- (and later four-) member REPRESENTATIVE RELEASE COMMITTEE. While each stakeholder was not a direct participant in the committee, each member of the committee explicitly represented a stakeholder. If a stakeholder had an issue that needed to be addressed by the committee, the stakeholder raised the issue through this committee representative. The group used metrics, such as the number of outstanding defects, to assess the progress of a release. After adopting the smaller release committee, issues were resolved more quickly and meetings were more focused and productive.

Related Patterns and Antipatterns. The RELEASE COMMITTEE needs participants who have sufficient authority to approve changes. If one group is delayed, SCHEDULE CHICKEN may ensue [Olson98]. A CORNCOB may wreak havoc if placed on a RELEASE COMMITTEE [BrownW98]. Although they focus on the software development process, some of the patterns in Ward Cunningham's EPISODES pattern language could be adapted for use by a release committee. For example, TECHNICAL MEMO could be used to document the committee's decisions [Cunningham96].

Criterion 2: Architecture users have a high level of confidence in the timing and content of architecture releases

When no overriding rhythm is set for an organization, particularly a large one, executives set the pace by reacting to internal or external forces by imposing panic deadlines on their staff. These panic actions are rarely coordinated with one another and create dysfunctional rhythms that wreak havoc on the timing and content of architecture releases, in the same way that a mixer with three beaters running in one bowl would quickly drain a bowl and splatter a room. If architecture users do not trust the timing and content of architecture releases, then the users may not plan to adopt new architecture releases, or they may choose another architecture altogether. Therefore, a lack of user confidence in the timing and content of architecture releases is a warning sign that a good rhythm has not been established.



Antipattern: SHORTCUT

General Form. You are the leader of an architecture team that has established a good rhythm. There are regular builds and releases. However, it is becoming difficult to maintain your tempo, schedules have slipped, and you are now facing intense pressure to get back on track. In an attempt to sustain the tempo, you have decided to skip a number of process steps (see Figure 4.10). Peer reviews might be omitted, or a late change might be introduced without going through the entire configuration management process. While the immediate tempo is retained by the move, it sows the seeds for later disruption. Defects that might have been detected in the skipped steps can surface later when they will be more expensive to correct.

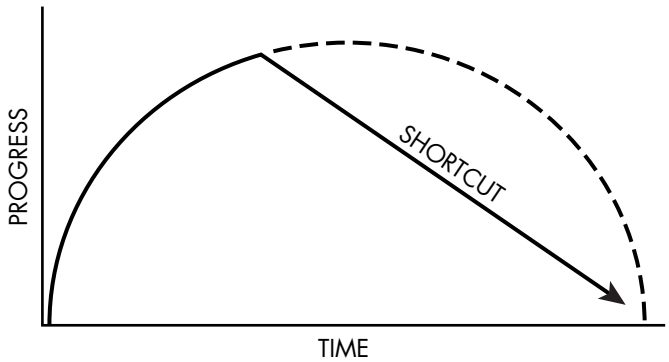


Figure 4.10 The SHORTCUT antipattern occurs when processes are skipped to maintain tempo.

Forces. There is often intense schedule pressure in software development, and this is no less true for organizations trying to establish and maintain a rhythm. Customers, stakeholders, and managers can all bring significant pressure to complete a release. When deadlines loom, there is a tendency to make decisions with a short-term benefit in exchange for consequences that are longer-term. The benefits of such decisions are immediately visible, while the consequences are less tangible and may not be immediately apparent. Defects are typically less expensive to correct the sooner in the life-cycle they are detected.

Solution. The appropriate way for processes to be enforced depends on the culture of the organization. In some organizations, periodic software audits may serve to ensure that the processes are followed. However, in many organizations, audits are not an effective way to change or enforce behavior. The actions of senior management can alter behavior more directly. Are there sufficient resources allocated to perform all the planned steps? Do managers create clear expectations so that processes are followed? Do they inquire if these expectations are met? Management actions can have a dramatic impact on whether the organization takes shortcuts to maintain the appearance of rhythm.

Rationale. In these days of shorter and shorter product cycles, there is a great deal of pressure to take shortcuts. Ironically, organizations that try to establish a rhythm may be even more susceptible. A good rhythm is composed of tempo, content, and quality, but of these, tempo is by far the most visible. Action is needed to counter-balance the tendency to maintain tempo at the expense of content and quality. In the long run, such sacrifices undermine what makes rhythm valuable in the first place.

Example. An architecture component developer for a large command and control system had well-established processes for creating and delivering new component releases. The component team was reorganized and received new managers. Just after this management change, the primary customer for the component requested some minor changes. In an attempt to please their customers, the new managers abbreviated the established delivery process, but in doing so several defects were introduced. Rather than pleasing the customer with their responsiveness, a defective component was delivered, followed by a delay as the component team redelivered the component using the established process.

When is This Antipattern a Pattern? Sometimes it does make sense to trim steps out of a process or tailor a process for a particular situation if it does not add value in that context (Chapter 7, Simplification). SHORTCUT might also make sense if the project is in a do-or-die situation where long-term consequences are not as important because the company might not be around long enough to experience them. Skipping a step that both internal and

external stakeholders are expecting should be approached with great caution and validated by a seasoned manager. In these situations, the organization needs enough discipline to record the shortcuts and mitigate the consequences once the immediate crisis has passed.

Variation: REDEFINE THE RULES. Instead of explicitly skipping a process step, this antipattern sometimes emerges when groups REDEFINE THE RULES. For example, a beat might be defined as completing a major milestone every month, and one of the criteria for completing a milestone is having no outstanding high-impact defects. In this variation of the antipattern, a group redefines critical defects to lower-impact classifications in order to pass the milestone. It appears that the organization has maintained a regular beat but at the price of burying a potentially significant risk. Additional effort and discipline are needed to manage the risk because it has been hidden from the usual processes.

Related Antipatterns and Patterns. SHORTCUT can result when there is too much emphasis on the timing of DAILY BUILD AND SMOKE TESTS but not enough emphasis on content of the builds or whether they are used [Cabrera99].

Pattern: DROP PASS

Problem Statement. How do you maintain the tempo of architecture releases when components are delayed?

Context. The architecture is released in regular intervals; for example, the architecture could have annual major releases and quarterly minor releases. Further, there are a number of independently developed components in the architecture. The release in question involves changes to a number of these components. The enhancements made to delayed components are not among the most essential features of the architecture release. Even though an effort has been made to get a component back on schedule, it appears that it will still be delayed.

Forces. The developers using the architecture could be adversely affected if the component release date is delayed. If the delay is due to one component, the other component owners might be tempted to try to play SCHEDULE CHICKEN [Olson98]. Not all components are equally important to the developers using the architecture, but all customers may be affected if an architecture release is delayed. Many users of an architecture do not read the release notes in detail and may not see notices about delayed or dropped features. It is very hard for suppliers to fully understand the consequences of a delayed component.

Solution. When it appears that revisions to a component will not be completed in time, alert stakeholders as soon as you think the feature might not

be included and verify that it is not critical. Go ahead with the architecture release without changes to the delayed component. To avoid the problem of users who do not read or see announcements about the change in features, make sure to drop the feature from the preliminary releases, so they experience the changes in alpha or beta, and not in the production release. Incorporate the changes to that component in a later release of the architecture.

Result. The tempo of the release is maintained. Activities planned after the release of the architecture can proceed as scheduled. The practice may also motivate component owners to finish their revisions on schedule. Developer trust grows because the architecture release occurs when promised. The developer also has more confidence that the next release will occur as planned.

Consequences. The release will have less functionality than if the release was held up for the component. Developers with plans to use the new features in the delayed component will need to make alternate plans. If customers do not trust the architecture provider, the announcement of a DROP PASS may prompt them to find an alternate provider. Unless other stakeholders are involved in the decision to drop a feature, problems could surface. For example, the product could fail tests because the test cases were not updated to reflect the change, or marketing could provide incorrect information to customers.

Rationale. Like a hockey player using a drop pass to give the puck to a teammate following from behind, this pattern moves a revised component to a following release. The pattern calls for an explicit tradeoff between the tempo and content of a release. Because there are many activities tied to the release of an architecture, delaying the release of the entire architecture could disrupt many organizations.

The pattern only works if the delayed component is not essential for the new architecture to be valuable to the developers who plan to use it. If the delayed component is critical, it may make more sense to delay the release instead. The onus is on the dependent parties here. They are the ones who can best judge the impact of postponing a component. The consumers have the obligation to push back on the suppliers if a component will be delayed too long.

Example. Microsoft uses a common product architecture so that new features can be written once and shared across products. They also release their applications over a number of time-paced intervals. They can drop pass low-priority features to the next planned release to make sure that critical features are included in the current release [Eisenhardt98].

Related Patterns and Antipatterns. A RELEASE COMMITTEE can be used to help mitigate the consequences of DROP PASS by ensuring that all the

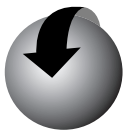
stakeholders are in agreement and aware of the changes. Many of the patterns from Linda Rising's Customer Interaction Patterns can be used to communicate the impact of a DROP PASS with customers [Rising99].

Criterion 3: Explicit activities are coordinated via rhythm

Software architectures have stakeholders in many different organizations. A shared architecture rhythm helps these autonomous groups to work together across organizational boundaries because it helps establish shared assumptions about when and how key events will occur. For example, if a product developer knows that there is a major architecture release annually and quarterly maintenance releases, the developer could time product releases to follow the expected architecture releases and take advantage of new architecture features.

Rhythm also coordinates activities across different groups in a firm. A daily or weekly build can coordinate activities across a development group. A human resources apartment can coordinate hiring to acquire developers early in a release cycle.

When rhythm is not a coordinating presence among stakeholders, decisions and transitions become more cumbersome. Human resources could hire staff the development group is not prepared to integrate into its team. Additional effort must be spent when rhythm is lacking to coordinate activities and to recover from transitions that occur at inopportune times. Developers might get slowed down because a shared component does not integrate cleanly.



Antipattern: BROKEN LOADS

Alias: BROKEN BUILDS

General Form. You are the leader of a team that has established regular builds. However, even though the code is checked in, compiled, and tested regularly, it does not always do so cleanly. Broken compiles or failed test cases are not considered to be a big deal. The team may believe it will be possible to straighten out the inconsistencies later in the process, or perhaps the team is using an advanced configuration management system to continue working and still keep track of the code versions in conflict. However, when it comes time to get a release out the door, it takes much longer to get all the pieces to work together than was anticipated.

Forces. There are many benefits from regular builds; however, they require significant effort to implement, especially for very large applications. In large systems it may not be feasible to rerun all test cases after each build, and managing test case dependencies can be especially challenging. If a build

breaks, it may be tempting to put off resolving the defect if there is other more interesting work to do. If the build breaks, the software is in an unknown state, and so larger problems may remain hidden. An advanced configuration management system can give the developers a false sense of security to manage multiple codebases when a build breaks. Such a system could allow developers to continue programming without stopping for the build to be repaired and without addressing the causes of the broken load.

Solution. Establish a commitment to regular builds. Management must clearly communicate to the developers that the regular builds are expected to be successful. The builds should include not only compiling the product, but also some form of automated testing. The process for regular builds may also be modified to prevent new work from moving forward until a broken build is corrected. Similarly, failed test cases that had been previously successful should be addressed immediately. Some organizations use social pressure on developers who break a load; that is, developers who break a load may be required to wear a proverbial dunce cap. Caution—this particular strategy has the potential to be counterproductive, depending on the culture of the organization.

Rationale. Builds must be used to be valuable. If they are not used as baselines, and if they are not used to keep the system in a known state, then they cannot help the organization maintain a rhythm. While a regular build process does introduce a regular tempo, if the build is always breaking, then the benefit of this tempo and the content delivered by the build is lost (see Figure 4.11). An effective regular build process can promote communication among developers and other stakeholders. For example, in organizations with such a build process, there is more cooperation among developers to ensure that their code works together so they can avoid breaking a load. When builds start breaking, it is a sign that this informal coordination among the component owners has deteriorated as well.

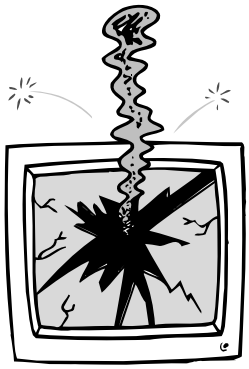


Figure 4.11
BROKEN LOADS are a sign that RHYTHM is breaking down.

Example. A group at a telecommunications firm had a process of completing weekly builds, but managers had not placed a priority on maintaining this schedule. When a build was not successful, development on new features continued as a "managed exception." An internal review identified this as a problem, and corrective action was taken. An engineer submitted new code that caused the build to break. The engineer had failed to coordinate the changes in his component with the other component owners before submitting the change. The engineer received a call in the middle of the night and received a visit from a vice-president first thing the next morning. The expectation was communicated to the entire team that both the schedule and the quality of the weekly builds needed to be maintained. After this new emphasis on rhythm was established, the team improved its ability to deliver on schedule.

The need to address BROKEN LOADS is often balanced with the need to move forward. At one leading software firm, when a build breaks, a series of hot bugs are entered and dispatched to developers as quickly as possible. The developer must drop everything to fix the bug, and a fix is typically expected within minutes, or hours at worst. Many daily builds will generate no significant bugs, while others may generate five or ten (for a product with several million lines of code). The build lab waits for fixes to come in and then iterates the build. At a certain point in the day they will abandon the current day's build and start preparing for the next.

When is This Antipattern a Pattern? Builds may occasionally break, and this may not be a big deal. But when this becomes a recurring event, then it is a problem that must to be resolved.

Variation: CONTINUOUS BUILDS. Some groups have taken regular builds a step further by using CONTINUOUS BUILDS. Every time code is checked in, it must pass through an automated system which forces a complete rebuild and "sniff test" of all affected code before the check-in is committed. Teams that use continuous builds will have a daily build available for the developers as well.

This approach is similar to that of eXtreme Programming. In XP, no code goes more than a couple of hours without integration. When code is integrated, the latest release is used, and it is expected to pass all of the test cases [Beck00].

Related Antipatterns and Patterns. This antipattern is an example of DAILY BUILD AND SMOKE TEST gone awry [Cabrera99].

Builds must be used to be valuable. Rhythm synchronizes content; content delivers value.

Pattern: SYNCHRONIZE RELEASES

Problem Statement. How can you accelerate the release of products built on your architecture?

Context. There have been a number of regular releases of your architecture, and there are a number of trusted developers who use the architecture to create products that add value to it. These developers are partners who receive early access to releases of the architecture so that they can provide feedback and accelerate development of their own products. The planned features for the release have already been determined.

Forces. Increasingly, companies are partnering with other vendors to provide complete solutions for their customers. This means that when a new or revised architecture comes to market, other complementary products are also needed for the architecture to be successful. It takes time for these complementary products to come to market. Customers may hold off on adopting an architecture release or upgrade until these complementary products become available. If the complementary products are available more quickly, then architecture adoption can be accelerated.

Solution. Work with your partners to determine the order in which the features of the architecture should be delivered for them to develop products using the architecture. To the extent possible, include these features in the early access releases of the architecture. If there are changes to the architecture that require substantial changes in complementary products, then these changes should be visible in the first preliminary releases. Communicate with the partners about when to expect which features. In exchange for adjusting the early releases in this manner, establish agreements with partners to quickly bring to market their products that include or require your architecture.

Result. When this pattern is successfully applied, there is little or no delay between the release of the architecture and the release of value-added components built on the architecture. With more components available that work with the architecture, architecture adoption will be quicker and more widespread.

Consequences. Not all developers will have the same needs, and so it may not be possible to coordinate the features incorporated in the releases in a way that satisfies everyone. The optimal release plan for the architecture team may differ from the optimal release plan for the partners. Some firms may be accused of "playing favorites" if they give early access to some firms but not others. Some partners may not be interested in making the commitment associated with synchronizing their releases; they may wish to wait until the architecture has gained market acceptance.

Rationale. Synchronizing releases accelerates the availability of complementary products because the needs of the partners are incorporated into the release plans and communicated back to the partners. The tempo and content of the architecture releases can be coordinated with that of the partners using the architecture. Partners are better able to prioritize their own development processes. Because the needs of the partners are met, they can bring complementary products to market more quickly.

Example. An operating systems development group noticed that it would usually take several months after the release of a new version of the operating system before compatible products and tools from third parties reached the market. The group partnered with the third-party developers to find out which features were needed for the early developer releases of the operating system. By allocating features across the developer releases in this fashion, the third-party developers were able to get more work done more quickly. Once the practice was adopted, new releases of the operating system were simultaneous with the release of compatible third-party products.

There are also examples of this pattern in other industries. A large household goods manufacturer adjusted the timing of its product launches to synchronize with the shelf-planning cycles of large retailers such as Wal-Mart and Target. The practice increased the shelf space for the manufacturer's products, and it helped the retailers stock the newest products [Eisenhardt98].

Related Patterns and Antipatterns. Many of the patterns from Linda Rising's Customer Interaction Patterns can be used to work with partners to establish SYNCHRONIZED RELEASES. IT'S A RELATIONSHIP, NOT A SALE is particularly relevant [Rising99].

SUMMARY

Rhythm coordinates the activities of the architecture stakeholders, and it helps spur progress of the architecture team. There are three elements to rhythm—tempo, content, and quality. Tempo is the frequency with which the same type of handoff occurs between groups. Content is the delivery of value from one group to another. Quality is the set of activities needed to develop and maintain an architecture without deficiencies. When these three elements are present, an organization has a good rhythm.

Rhythm is important for architecture-based development because so many different organizations are involved in the development and use of an

architecture. It is not possible to manage all of these groups from the top down, so rhythm is needed to coordinate these autonomous groups.

Rhythm has other benefits. First, it enables the architecture stakeholders to focus on transitions. Transitions are critical for success, but they frequently do not receive the attention they warrant. Rhythm also enables the architecture stakeholders to create an urgency that drives progress forward.

While there are many practices that support rhythm, such as regular builds popularized by Microsoft, there are a number of pitfalls. This chapter explored some of these pitfalls and described solutions to some of the problems encountered by organizations seeking to establish and maintain a rhythm.

OTHER APPLICABLE PATTERNS AND ANTIPATTERNS

There are other patterns that can be used to put the principle of Rhythm into practice, as well as antipatterns to avoid along the way. Table 4.2 lists organizational patterns and antipatterns cataloged in the 2000 edition of *The Patterns Almanac* [Rising00]. Table 4.3 lists antipatterns from *Antipatterns: Refactoring Software, Architectures, and Projects in Crisis* [BrownW98].

TABLE 4.2 *Organizational Patterns and Antipatterns That Can Shape Rhythm [Rising00]*

BACKLOG [Beedle99]	PHASING IT IN [Coplien95]
CASUAL DUTY [Olson98]	PRODUCTION POTENTIAL [Taylor99]
COMPLETION HEADROOM [Cunningham96]	PROGRAMMING EPISODE [Cunningham96]
COUPLING DECREASES LATENCY [Coplien95]	PULSE [Taylor99]
DECOUPLE STAGES [Coplien95]	SACRIFICE ONE PERSON [Cockburn98]
DELIVERABLES TO GO [Taylor99]	SCHEDULE CHICKEN [Olson98]
DIVIDE AND CONQUER [Coplien95]	SCRUM MEETINGS [Beedle99]
DON'T INTERRUPT AN INTERRUPT [Coplien95]	SOMEONE ALWAYS MAKES PROGRESS [Cockburn98]
EARLY AND REGULAR DELIVERY [Cockburn98]	SPRINT [Beedle99]
EFFECTIVE HANDOVER [Taylor99]	TAKE NO SMALL SLIPS [Coplien95]
GET INVOLVED EARLY [DeLano98]	TAKE TIME [DeLano98]
GOLD RUSH [Cockburn98]	TEAM PER TASK [Cockburn98]
HUB, SPOKE, AND RIM [Coplien95]	TIME TO TEST [DeLano98]

TABLE 4.2 *(Continued)*

INTERRUPTS UNJAM BLOCKING [Coplien95]	WORK GROUP [Cunningham96]
KEEP IT WORKING [Foote99]	WORK QUEUE [Cunningham96]
LONG POLE IN THE TENT [Olson98]	WORK QUEUE REPORT [Cunningham96]
MICROCOSM [Cockburn98]	WORK SPLIT [Cunningham96]
NAMED STABLE BASES [Coplien95]	

TABLE 4.3 *Antipatterns That Can Derail Rhythm [BrownW98]*

ARCHITECTURE BY IMPLICATION
FEAR OF SUCCESS
FIRE DRILL
IRRATIONAL MANAGEMENT
PROJECT MISMANAGEMENT
SMOKE AND MIRRORS
THE GRAND OLD DUKE OF YORK
THROW IT OVER THE WALL
WALKING THROUGH A MINEFIELD
WARM BODIES