Engineering excellence: the art of doing small things really well

Abhishek Tiwari

Citation: *A. Tiwari*, ”Engineering excellence: the art of doing small things really well”, Abhishek Tiwari, 2024. doi:10.59350/tv006-nyg03

Published on: September 15, 2024

Recently I have been reading Art of Leadership, The: Small Things, Done Well by Michael Lopp. This book is an excellent read and it covers small things that you can do to build trust and to become an authentic and true leader to your team at different stages of your leadership journey. Michael use stories from his tenure as manager at Netscape , as director at Apple , and as executive at Slack to capture and distill a list of wisdom to help anyone be effective at management and leadership. Book advocates that by focusing on the small things and executing them well, leaders can create a lasting impact on their teams and organisations. It argues that great leaders understand that success is built on a foundation of countless small, well-executed actions.

While reading the book, I had an interesting realisation that doing small things really well can also lead to engineering excellence. Look around and you will find 100s of opportunities to make small changes with lasting impact. In this post we will look at some example of small well-executed actions.

What is engineering excellence?

Engineering excellence, in the context of software development, is the relentless pursuit of quality, efficiency, and effectiveness in every aspect of our work. It's not just about writing code that works; it's about crafting solutions that are elegant, maintainable, and scalable. It's about paying attention to the minutiae that, when combined, create a robust and reliable system.

Examples of small things that matter

Here are a few good examples which I can think off, small yet highly impactful:

Keeping dependencies up-to-date

Regularly updating libraries and dependencies is a crucial practice that can prevent security vulnerabilities and ensure you're benefiting from the latest improvements. This seemingly small task can have a significant impact on the overall health and security of your project.

Outdated dependencies can expose your application to known security vulnerabilities, potentially compromising your system's integrity. They may also lack performance improvements or bug fixes that could benefit your project. By staying up-to-date, you not only enhance security but also take advantage of new features and optimizations that could improve your application's performance and functionality.

However, manually tracking and updating dependencies can be time-consuming and error-prone. This is where automated tools can play a crucial role in maintaining engineering excellence with minimal effort.

Automating dependency updates using GitHub Dependabot

GitHub's Dependabot is an excellent example of how small, automated processes can contribute to engineering excellence. Dependabot can automatically create pull requests to update your dependencies when new versions are available. When combined with GitHub's auto-merge feature, it can streamline the process of keeping your dependencies up-to-date with minimal human intervention.

Listing 1: Dependabot config to check for updates to npm packages and Docker images weekly, create pull requests for updates, and apply specific labels to these pull requests.

```yaml
version: 2
updates:
  - package-ecosystem: "npm" # For JavaScript projects using npm
    directory: "/"
    schedule:
      interval: "weekly"
    open-pull-requests-limit: 10
    target-branch: "develop"
    labels:
      - "dependencies"
      - "automerge"
    versioning-strategy: auto

  - package-ecosystem: "docker"
    directory: "/"
    schedule:
      interval: "weekly"
```

By implementing this system, you ensure that your project is always using the latest secure versions of its dependencies, with minimal manual intervention. This small, automated process can save countless hours of manual work and significantly reduce the risk of security vulnerabilities sneaking into your project through outdated dependencies.

Listing 2: Github Actions workflow to automatically merge Dependabot pull requests that only include minor or patch version updates, assuming they pass all required checks.

```yaml
name: Auto-merge Dependabot PRs

on:
  pull_request:

jobs:
  auto-merge:
    runs-on: ubuntu-latest
    if: github.actor == 'dependabot[bot]'
    steps:
      - uses: actions/checkout@v2
      - uses: ahmadnassri/action-dependabot-auto-merge@v2
```

```
        with:
          target: minor
          github-token: ${{ secrets.GITHUB_TOKEN }}
```

Remember, while automation is powerful, it's still important to review significant updates, especially major version changes that might include breaking changes. The goal is to strike a balance between staying up-to-date and maintaining stability in your project.

Documenting decisions

Keeping a record of why certain technical decisions were made can provide crucial context for future development, ideally through Architecture Decision Records (ADRs) or Request for Comments (RFCs). For these documents, the "why" is more important than the "how".

ADRs and RFCs serve as a historical record of significant decisions in a project. They capture the context around a decision, which is often lost over time as team members change or memories fade. This context is crucial for understanding the rationale behind the current state of a system, which can inform future decisions and prevent the repetition of past mistakes.

When creating these documents, focus on answering questions like: What was the problem we were trying to solve? What constraints were we operating under? What alternatives did we consider? Why did we choose this particular solution over others?

The "how" of implementation is important, but it's typically well-documented in the code itself and associated technical documentation. The "why", on the other hand, is often not captured anywhere else.

Listing 3: An example ADR explaining why a team chose to use a particular database technology.

```
# ADR 1: Choice of Database Technology

## Context
Our application needs to handle large volumes of unstructured data with
   high write throughput.

## Decision
We will use MongoDB as our primary database.

## Rationale
- MongoDB's document model suits our unstructured data better than a
   relational model.
- It offers high write performance, which meets our throughput
   requirements.
- Our team has experience with MongoDB, reducing the learning curve.
- It provides good scalability options for our future growth plans.
```

```
## Consequences
- We'll need to carefully design our data model to avoid performance
   issues with large documents.
- We'll lose some of the strong consistency guarantees offered by
   relational databases.
- We may need to implement our own solutions for complex queries that
   would be simple in SQL.
```

Above ADR clearly explains why MongoDB was chosen, providing future developers with the context they need to understand and work with this decision. It doesn't go into the details of how MongoDB will be implemented – that information belongs in technical specifications and code documentation.

By consistently creating and maintaining these records, teams build a valuable knowledge base that supports long-term maintainability and informed decision-making. This practice, while small and often overlooked, can have a significant impact on the overall excellence of a software engineering project.

Feature branching and semantic versioning

A prime example of how small, well-executed practices can lead to engineering excellence is the use of feature branching with semantic versioning, integrated into a Continuous Integration (CI) system. This approach, when implemented consistently, can significantly improve code quality, release management, and team collaboration.

Teams often use a branching strategy where each new feature or bug fix is developed in its own branch. These branches are named following a specific convention, often including the type of change (`feature`, `bugfix`, `hotfix`) and a brief description. For example: `feature/user-authentication` or `bugfix/login-timeout`.

Semantic versioning is then used to version the software. This typically follows the format `MAJOR.MINOR.PATCH` (e.g., `2.3.1`), where each number is incremented based on the nature of the changes.

When these practices are combined with a CI system, it creates a powerful workflow. As developers push their feature branches, the CI system can automatically build the code, run tests, and even generate versioned build artifacts. For instance, a feature branch might produce a pre-release version like `2.4.0-feature.user-authentication.1`.

To enforce these practices, teams often employ Git hooks. These are scripts that run automatically on certain Git events. For example, a pre-commit hook can check that the branch name follows the agreed-upon convention. A commit-msg hook can ensure that commit messages adhere to a specific format, perhaps including a reference to a ticket number or following a conventional commit format.

These may seem like small details, but they contribute significantly to the overall quality and manageability of the software development process. They ensure consistency, improve traceability, and make it easier to understand the history and state of the project at any given time.

Git Hook: Enforcing branch naming convention

To illustrate how these small practices can be implemented, let's look at a concrete example of a Git hook that enforces a branch naming convention. This pre-push hook, written in Bash, ensures that all branch names follow a specific pattern before allowing a push to the remote repository.

Listing 4: Git hook allows only branch names starting with 'feature/', 'bugfix/', or 'hotfix/', followed by lowercase letters, numbers, and hyphens. If the branch name doesn't match, it prints an error explaining the convention and exits with a non-zero status, which prevents the push.

```
#!/bin/bash

# File: .git/hooks/pre-push
# Make this file executable: chmod +x .git/hooks/pre-push

# Define the allowed branch name pattern
# This pattern allows: feature/, bugfix/, hotfix/ followed by lowercase
   letters, numbers, and hyphens
allowed_pattern="^(feature|bugfix|hotfix)/[a-z0-9-]+$"

# Get the current branch name
branch=$(git symbolic-ref --short HEAD)

if [[ ! $branch =~ $allowed_pattern ]]; then
    echo "Error: Branch name '$branch' does not follow the naming
       convention."
    echo "Branch names should start with 'feature/', 'bugfix/', or 'hotfix
       /' followed by lowercase letters, numbers, and hyphens."
    echo "Example: feature/user-authentication"
    exit 1
fi

# If we've made it here, the branch name is valid
exit 0
```

Above script, barely 20 lines long, can have a significant impact on maintaining consistency across an organisation. It ensures that all team members follow the same branching convention, which in turn makes it easier to understand the purpose of each branch at a glance, automate processes based on branch names, and maintain a clean and organised repository.

Git Hook: Preventing accidental commit of secrets

Another crucial use of Git hooks is to prevent the accidental commitment of sensitive information, such as API keys or passwords. Here's an example of a pre-commit hook that checks for potential secrets in the files being committed:

Listing 5: An example Git hook to prevent accidental commit of the secrets. For each staged file, it checks the content against each secret pattern. If any potential secrets are detected, the hook exits with a non-zero status, preventing the commit from proceeding.

```
#!/bin/bash

# File: .git/hooks/pre-commit
# Make this file executable: chmod +x .git/hooks/pre-commit

# Define patterns for potential secrets
secret_patterns=(
    'password\s*=\s*['"'"'"]\w+['"'"'"]'
    'api[_-]key\s*=\s*['"'"'"]\w+['"'"'"]'
    'secret\s*=\s*['"'"'"]\w+['"'"'"]'
    'aws_access_key_id\s*=\s*['"'"'"]\w+['"'"'"]'
    'aws_secret_access_key\s*=\s*['"'"'"]\w+['"'"'"]'
)

# Get list of staged files
staged_files=$(git diff --cached --name-only)

# Flag to track if any secrets are found
secrets_found=false

# Check each staged file
for file in $staged_files; do
    # Skip binary files
    if [[ -z $(grep -Ia . < "$file") ]]; then
        continue
    fi

    # Check file content against secret patterns
    for pattern in "${secret_patterns[@]}"; do
        if grep -Eq "$pattern" "$file"; then
            echo "Potential secret found in $file"
            grep -En "$pattern" "$file"
            secrets_found=true
        fi
    done
done

# If secrets are found, abort the commit
if $secrets_found; then
```

```
    echo "Error: Potential secrets found in commit. Please remove them and
        try again."
    exit 1
fi

# If we've made it here, no secrets were found
exit 0
```

Above hook can prevent a catastrophic security breach caused by accidentally committing sensitive information to a repository. It's a prime example of how a simple, proactive measure can significantly enhance the security practices of a development team.

By implementing practices like these, teams can build a foundation of small, well-executed actions that contribute to overall engineering excellence. Each individual action might seem minor, but together they create a robust, secure, and efficient development process.

Automated linting in development workflows

One of the most impactful small practices in modern software development is the use of automated linting tools integrated directly into the development workflow. This can be achieved through two primary methods: IDE integration and pre-commit hooks.

When integrated into an Integrated Development Environment (IDE), linters can provide real-time feedback as developers write code. These tools analyze the code for potential errors, style violations, and adherence to best practices, offering immediate suggestions for improvements. This instant feedback loop allows developers to correct issues on the fly, leading to higher quality code from the outset. Popular IDEs like Visual Studio Code, IntelliJ IDEA, and others offer robust support for various linting tools across different programming languages.

Listing 6: An example of how ESLint automatically fixed several issues in code: (1) Removed unnecessary spaces in the function declaration (2) Changed var to const (assuming you're using ES6+ (3) Added a semicolon at the end of each statement (4) Fixed indentation.

```
// Before saving:
function badlyFormattedFunction ( ) {
    var x = 'This is a badly formatted string'
    return x
}

// After saving:
function badlyFormattedFunction() {
  const x = 'This is a badly formatted string';
  return x;
}
```

Pre-commit hooks, on the other hand, act as a final checkpoint before code is committed to version control. These hooks can be configured to run linting checks automatically whenever a developer attempts to make a commit. If any linting rules are violated, the commit can be blocked, ensuring that only code meeting the project's standards makes it into the repository. This approach helps maintain consistent code quality across the entire project and prevents style-related issues from cluttering code reviews.

By implementing automated linting through both IDE integration and pre-commit hooks, development teams can ensure consistent code style across the project and catch and correct potential errors early in the development process. This approach significantly reduces time spent on style-related discussions in code reviews, allowing team members to focus on more substantive aspects of the code. Furthermore, it serves as an effective mechanism for enforcing best practices and project-specific conventions automatically.

Automated linting also provides an invaluable learning mechanism for developers, especially those new to the project or language. As they write code, they receive immediate feedback on best practices and project conventions, accelerating their learning curve and helping them align with team standards more quickly.

The power of automated linting lies in its ability to make code quality a natural part of the development process rather than an afterthought. These small, frequent checks compound over time, leading to significantly improved code quality, reduced technical debt, and increased developer productivity. By baking these practices into the daily workflow, teams can achieve a higher standard of engineering excellence with minimal additional effort. The cumulative effect of these small, automated checks is a codebase that is more consistent, more maintainable, and less prone to errors, ultimately contributing to the overall success of the software project.

The compounding effect

Just as in leadership, these small actions in engineering compound over time. A codebase that consistently applies these practices becomes increasingly robust, maintainable, and efficient. Team members who habitually focus on these details often find that their productivity increases, bugs decrease, and the overall quality of their work improves dramatically.

As engineering organizations grow larger, the payoff from these small actions becomes massive. Let's consider a few scenarios:

1. In an organization with 1,000 engineers, if 10% of them accidentally commit a secret once a month, and preventing each incident saves 1 hour of remediation time, that's 100 hours saved per month. Over a year, this amounts to 1,200 hours or 150 working days saved.

2. If you have 100 repositories, and using Dependabot saves 2 hours per repository per month that would have been spent manually updating dependencies, that's 200 hours saved per month. This translates to 2,400 hours or 300 working days saved per year.
3. If each of the 1,000 engineers saves just 5 minutes a day due to consistent, automatically enforced code formatting, that's 83 hours saved daily. Over a year, this could save over 20,000 hours of developer time.

These examples illustrate how small, seemingly insignificant actions can lead to enormous time and resource savings when implemented across a large organization. Moreover, these practices not only save time but also prevent potential security breaches, reduce technical debt, and improve overall code quality.

The true power of these small actions lies not just in the immediate time saved, but in the cumulative effect they have on the organization's engineering culture. They foster an environment of conscientiousness and attention to detail, where excellence becomes the norm rather than the exception. This cultural shift can lead to innovations, improved product quality, and ultimately, better outcomes for the business and its customers.

Cultivating a culture of excellence

Engineering excellence isn't just about individual actions; it's about fostering a team culture that values and prioritizes these small but important details. This culture can be built by leading by example, recognizing and rewarding attention to detail, providing time and resources for continuous improvement, and encouraging knowledge sharing and mentorship.

Conclusion

As software engineers and leaders, it's easy to focus solely on the big picture and overlook the small details. However, true engineering excellence lies in the art of doing small things really well. By paying attention to these details and consistently executing them with care, we can create software that not only works but excels in quality, maintainability, and user satisfaction.

Remember, excellence is not an act, but a habit. Start small, be consistent, and watch as these tiny improvements compound into significant results. The path to engineering excellence is paved with small, well-executed actions. What small thing will you improve today?