Pragmatic Programmers

Agile in a Flash

Speed-Learning Agile Software Development

Card 1—Why Agile?

The Idea

Get us all on the same page Imagine the ...

1 Why Agile?

- Get us all on the same page
- Get product out the door
- Drive down risk
- Learn, adapt, deliver!
- Take pride in your craft
- True transparency
- The joy of building

Agile Cards for Agile Teams

Jeff Langr and Tim Ottinger

edited by Susannah Pfalzer

What Readers Are Saying About *Agile in a Flash*

I have only one major issue with your cards, which is that I didn't think of them and do them first. That wouldn't be so bad if you were screwing them up, but unfortunately they're great.

► **Ron Jeffries**
Coauthor, *The Agile Manifesto*, www.XProgramming.com

Agile in a Flash is the only place to find a concise summary of all things agile. I recommend my customers buy it for all their programmers.

► **Mike Cohn**
Author of *Succeeding with Agile*, *Agile Estimating and Planning*, and *User Stories Applied*

Pragmatic Bookshelf

Printed in Canada.

ISBN-13: 978-1-934356-71-5 P1.0 printing, January 2011 Version: 2011-1-18 Printed on acid-free paper.

In agile development, the values are axiomatic, and all the rest is derivative and changeable.
► Tim Ottinger

Using This Deck

Leaders, team members, problem solvers, coaches, coders: this *Agile in a Flash* deck is for you. Use the cards as conversation icebreakers, as reminders or references, and as a source of practical tips and hard-won wisdom.

Don't let the *Agile in a Flash* deck be shy. The cards are flexible! Share with your teammates. Slip one into your manager's pocket as a hint. Choose a card at random to stimulate discussion. Tack useful cards onto walls or monitors. To get the point across, fold a card into a paper airplane and airmail it. Customize your deck by removing or reordering cards. Tear a card up to make a statement—reorders are easy and cheap.

Just learning agile? The deck provides concise summaries of important agile concepts and practices. It also provides a wealth of experience to help you avoid pitfalls going forward. Use it as a reference or an "agile tip of the day" stack. Coaching an agile team? The deck goes with you everywhere in your back pocket and is there to support you when you need to make an important point to the team. Experienced with agile? Take your team to the next level—we've captured some advanced and nuanced thoughts on how to succeed with agile.

Our one caveat is that the existence of this deck suggests you can be agile by simply checking off all items on the cards. Instead, treat these cards as collected wisdom, reminders, and guidelines, not absolute prescriptions and rules.

Each *Agile in a Flash* card features a short list or simple diagram with further explanation on the back. The cards are categorized into four sections:

The Idea	💡	Core agile principles, concepts, and values
The Plan	✦	Planning, estimation, stories, acceptance tests
The Team	🚩	Team and collaborative practices
The Code	🔄	Coding, design, Test-Driven Development (TDD), refactoring

Visit our companion website at http://AgileInAFlash.com to read the corresponding blog entry for each card in the deck, as well as for those that didn't make it into print. Also visit http://www.pragprog.com/titles/olag, where you'll find a discussion forum and errata.

1 Why Agile?

- Get us all on the same page
- Get product out the door
- Drive down risk
- Learn, adapt, deliver!
- Take pride in your craft
- True transparency
- The joy of building

Get us all on the same page Imagine the enmity disappearing as business and development team up to deliver ongoing value to customers!

Get product out the door Quarterly releases? That's for sissies! With iterative development, you can deliver every few weeks or even every few days.

Drive down risk Short release cycles allow you to get real, end-user feedback long before you've invested too much in a risky feature.

Learn, adapt, deliver! Feedback from real users keeps you in tune with the changing marketplace. Internally, you can continually improve your team with retrospection.

Take pride in your craft Using agile technical practices such as TDD, refactoring, and automated tests, you can be proud of the low-defect product you send out the door.

True transparency You can't miss the charts on the walls and the ongoing conversations in an agile team room. It's obvious that team members have and share more information than in nonagile projects.

The joy of building In agile, everyone can experience the excitement of working in a true team that delivers cool, working stuff all the time.

2 The Agile Values, aka the Agile Manifesto

We[1] have come to value:

Individuals and interactions	*over*	Processes and tools
Working software	*over*	Comprehensive documentation
Customer collaboration	*over*	Contract negotiation
Responding to change	*over*	Following a plan

That is, while there is value in the items on the right, we value the items on the left more.

1. See the list of seventeen signatories at http://agilemanifesto.org.

Agile software development teams collaborate and adapt with minimal ceremony and overhead to deliver quality software.

You may need “the things on the right” to succeed with agile, but it is effective to bypass paperwork and **just talk to people**. Translating needs into written form is wasteful compared to having your customer simply tell you what they want and remain available to answer questions.

Similarly, **working software matters**. Documents? Not as much. Capture specifications in the product and its tests through agile practices such as TDD (see Card 44, *A Rhythm for Success: The TDD Cycle*), metaphor (shared understanding of the system), and acceptance testing (see Card 21, *Acceptable Acceptance Tests*).

You can diminish the importance of contracts if you **negotiate on a continual basis**. This involves an increase in transparency and trust, supported greatly by the openness and heavy collaboration that agile promotes.

Plans are valuable, but your customer and the marketplace care less about your plans than about you **delivering software that fits their ever-changing needs**.

3 Principles Behind the Agile Manifesto

- Satisfy the customer through early, continuous delivery
- Welcome changing requirements, even late
- Deliver working software frequently
- Businesspeople and developers collaborate daily
- Build projects around motivated individuals
- Convey info via face-to-face conversation
- Primary progress measure: working software
- Maintain a constant pace indefinitely
- Continuously demonstrate technical excellence
- Simplify: maximize amount of work not done
- Self-organize
- Retrospect and tune behavior

The Agile Manifesto *values* can sound “warm and fuzzy,” but you will find that its dozen *principles* (paraphrased here from the originals at http://agilemanifesto.org) provide a much meatier description of agile.

Agile is foremost about **continually and incrementally delivering quality software** to a customer who must **constantly juggle changing requirements** in order to compete in the marketplace. Your job: **make your customer happy**.

You’ll best succeed if your team is **highly motivated**. They must **communicate and collaborate at least daily**, which is easiest if **everyone is in the same room, talking face-to-face.**

The team must embrace **technical excellence**, an essential part of **maintaining a constant delivery pace indefinitely**. By self-organizing, the team derives the best possible architectures, requirements, and designs.

To maintain a consistent rhythm of delivery, the team must **adapt through retrospection**. An incremental mindset helps sustain this rhythm: **keep it simple** by introducing features and complexity only when demanded, no sooner.

Always remember it’s about delivering the good software, baby. **Measure progress and success by your ability to continue to deliver**.

4 Role-Playing in Agile

- **Customer**: Helps define the product
- **Programmer**: Helps construct the product
- **Tester**: Helps verify the product works as defined
- **Tracker**: Helps gather and present useful metrics
- **Coach**: Helps guide the team to success
- **Coordinator** (optional): Helps manage external communication

In an agile team, everyone helps out, doing whatever it takes to deliver a useful, high-quality product. You are not bound by your job title. A tester might track team metrics, a programmer might help define acceptance criteria, and so on.

The **customer** has special responsibility and authority, because they are responsible for the product's functionality and user-facing design. Their supporting cast may include business analysts, product owners, and others who help define the product (including **testers**), but everyone on the team is responsible for advising the customer.

Programmers (and other technical folks, such as architects and support technicians) are responsible for the internal design, construction, and maintenance of the product.

A **coach** helps educate and guide your team, shunning command-and-control approaches. They help your team devise their own rules and protocols. The best coaches help teams mature to the point where the team no longer needs them.

We substitute **team coordinator** for roles like *manager*, *project manager*, and *Scrum Master*. The coordinator buffers the team from outside interference and distraction. They might communicate schedules, handle incoming requests, and smooth interpersonal problems.

5 Agile Success Factors

- Freedom to change
- Energized team
- Communication with customer
- Collaboration
- Attention to quality
- Incrementalism
- Automation

Taking the Agile Manifesto's values and principles one step further, these are the make-or-break factors for a team wanting to be agile:

Freedom to change A team must be allowed to own and change its process. Tampering diverts from shipping quality software.

Energized team A winning agile team is eager to deliver value, collaborates freely, and never bypasses quality controls even under pressure.

Communication with customer The best agile teams are in constant dialogue with an enthusiastic, individual, dedicated customer who understands and communicates the product vision.

Collaboration Get past cube mentality! Meetings aren't collaboration; working together in your code base is.

Attention to quality Lack of attention slows you down until you can no longer meet customer demand. Make quality part of everything you do.

Incrementalism Yet-smaller steps to everything (including retrospection) allow you to verify whether each step got you closer to the end goal.

Automation Agile cannot work unless you automate as many menial, tedious, and error-prone tasks as possible. There's just not enough time!

6 Courage

- ➤ To always deliver quality work
- ➤ To simplify code at every turn
- ➤ To attack code the team fears most
- ➤ To make architectural corrections
- ➤ To throw away unneeded code and tests
- ➤ To be transparent, whether favorable or not
- ➤ To take credit only for complete work

Cowardly Lion: *As long as I know myself to be a coward, I shall be unhappy.*

To always deliver quality work Even when pressure mounts, never discard the practices needed to deliver quality code. See Card 13, *Don't Get Too Deep in Technical Debt*.

To simplify code at every turn Simple code reads faster and more clearly, which leads to fewer defects and more fluid development. Simplifying is investing in speed of development.

To attack code the team fears most Fear of breaking ugly code makes a team dread changes instead of embracing them. Empower your team by getting that code under control.

To make architectural corrections Systems may outgrow early architectural decisions. It takes courage to undertake changes that will affect existing code, even if the change is clearly for the better.

To throw away tests and code Often the best results are produced by discarding a poor solution and reworking it. It takes far less time than you think.

To be transparent, whether favorable or not If you "color" your reports about status, progress, or quality, you contribute to ill-informed decision making.

To take credit only for complete work Incomplete work provides no business value! Don't rationalize calling it "done" for status tracking purposes.

7 Redefining Discipline

- Work attentively
- Work on one thing at a time
- Shorten feedback loops
- Continuously reflect and adapt
- Know when to call it a day
- Push back when it matters

In a pre-agile world, *discipline* was defined mostly as the inclination to stick with a plan. An agile workflow is very disciplined but with a different definition of discipline.

Work attentively Try to keep up but also keep track. Note which aspects of your work system are hard, clunky, troublesome, or slow.

Work on one thing at a time Do not dilute your effort and your attention by taking on multiple tasks at once. Take one tiny, verifiable step at a time.

Shorten feedback loops How long until you detect defects or reap benefits? Aggressively simplify your process to shorten that time.

Continuously reflect and adapt Invest in the team's ability to easily produce a quality product. Always build a better "next week."

Know when to call it a day A few more minutes of digging might crack a tough problem, but so could a night's sleep. Learn when to stop.

Push back when it matters Nobody wants a contentious employee. So, choose your battles carefully, but always protect the product, team, and company by advising against changes that oppose the values stated in the manifesto.

8 Pillars of Software Craftsmanship

- Care
- Learn
- Practice
- Share

Via the Craftsmanship Google Group,[2] Jason Gorman offered this nicely concise statement of Craftsman ethics, abbreviated and quoted here:

Care We care about the quality of the work we do. It should be as good as we're capable of making it.

Learn We learn from our mistakes and from examples, books, blogs, webinars, magic beans, and electric parsnips. When we're exposed to good examples, we learn better, so it's important to have good influences.

Practice We learn by doing, so we *practice*. Continuously. We internalize our knowledge and build our "software development muscles" and the necessary reflexes and muscle memory needed to be a productive programmer. You can no more master a skill like refactoring by reading a book on it than you can master riding a bicycle by reading the manual. It takes practice—good, focused, measured practice.

Share We share what we've learned. We build the good examples others learn from. We are all simultaneously masters and apprentices, teachers and students, mentors and mentored, coaches and coachees.

2. http://groups.google.com/group/software_craftsmanship

9

Toyota Production System (TPS) Principles

- Continuous improvement
- Respect for people
- Long-term philosophy
- Develop the right process
- Develop your people and partners
- Continuously solve root problems

Your agile team—scratch that, *any* team—can learn plenty from Toyota Production System principles (see http://en.wikipedia.org/wiki/Toyota_Production_System).

Continuous improvement Not only must we continuously reflect and adapt, but we must base such decisions on facts that derive from the true source.

Respect for people Even if a process fosters the delivery of quality software, it is a failure if it does so at the expense of the workers who employ it.

Long-term philosophy Building a process around short-term goals leads to a sloppy product. Teams must take a larger product focus.

Develop the right process The software development process is not about controlling a project with voluminous documentation and mandates. Allow each team to devise and continuously tweak their own mechanisms for success.

Develop your people and partners The best agile teams seek not only to continually learn more and challenge themselves but to promote this attitude in others with whom they interact.

Continuously solve root problems All involved should go to the problem to see it firsthand and work with the whole team to reach a consensus on how to solve it.

10 The Right Process

- Continuous process flow
- Pull work to avoid overproduction
- Work like the tortoise, not the hare
- Stop to fix problems
- Standardize tasks
- Expose problems with visual control
- Use only reliable technology that serves people and process

The Toyota Production System says “the right process will produce the right results.”[3]

Continuous process flow A process stream that requires continual delivery creates transparency; any blips in the process are sorely felt downstream.

Pull work to avoid overproduction An overproducing group produces waste by creating backlog. Produce only when the downstream group requests it.

Work like the tortoise, not the hare Rushing repeatedly to short-term deadlines can produce poor quality and burnout that destroys products and teams.

Stop to fix problems Small problems quickly pile up into progress-killing waste. Engage the whole team, and correct the problems before moving on.

Standardize tasks You cannot hope to continually improve without some foundational level of standardization.

Expose problems with visual control You cannot fix a problem easily until everyone admits it’s a problem. Don’t bury your mistakes.

Use only reliable technology that serves people and processes Your team has to be masters of the tools they use.

3. Source: http://en.wikipedia.org/wiki/Toyota_Production_System. See also Card 9, *Toyota Production System (TPS) Principles*.

11 Got Organizational Obstinance?

- **It can’t work here** “Our company is uniquely complex.”
- **They won’t let us** “Our culture doesn’t support it.”
- **Guilt by association** “Agile is like something else that failed.”
- **Means/ends juxtaposition** “It doesn’t support our management style.”
- **Inferiority complex** “We’re too afraid to improve the code.”
- **Superiority complex** “We’ve been shipping on time just fine.”
- **Rejection of insufficient miracle** “But it won’t solve all our problems.”

How will your team respond to the typical excuses when they want to try agile?

It can't work here Agile isn't a rigid set of practices. You need only a team willing to start with the core values and incrementally grow together.

They won't let us Start small by tackling a few agile practices, and win over management with success and the real data behind it.

Guilt by association A nonagile, semi-agile, or other nonwaterfall method may have failed for you in the past. That's no reason to avoid a proper agile project.

Means/ends juxtaposition Software management structures, ceremonies, and documents support developing software, not the other way around. Help your organization adopt new structures to support agile development.

Inferiority complex Agile improves developers via teamwork and doesn't leave people behind in their cubes while hoping the superstars deliver.

Superiority complex If you're perfect, why are you even considering agile? :-) If not, welcome to a world where we know we can always do better.

Rejection of insufficient miracle "Nothing will ever be attempted if all possible objections must first be overcome." —Samuel Johnson.

12 Got Individual Obstinance?

- **Personal bubble**: “I don’t want to share my space.”
- **Lone ranger**: “I work best alone.”
- **Old dog**: “I already know how to do my work.”
- **Zero-sum game**: “Why should I share credit?”
- **Inferiority complex**: “They won’t want to work with me.”
- **Superiority complex**: “Co-workers just slow me down.”
- **Rejection of insufficient miracle**: “I’ll wait for a panacea.”

Before you can overcome personal resistance to agile, you must understand it.

Personal bubble/social dysfunction Agile demands interpersonal interaction. Self-esteem issues, social dysfunctions, jealousy, and grudges can make it hard for members to collaborate.

Lone ranger Developers who seek to be the team's guru or hero, by staying late or mastering arcane code and skills, may fear losing their esteemed status.

Old dog It's hard to abandon productive old habits without certainty that new skills will prove to be even more productive.

Zero-sum game A classic dysfunction is viewing a teammate's failure as a personal success, and vice versa.

Inferiority complex Some developers fear that collaboration will expose their self-perceived weaknesses and devalue them to the company.

Superiority complex Some developers will object to agile because working with their "inferior" teammates will "drag them down."

Rejection of insufficient miracle Though agile software development may help solve many problems, it may leave others unsolved.

13 Don't Get Too Deep in Technical Debt

Technical debt is technical work deferred as a business decision, but it quickly becomes a serious business problem.

- Development slows down
- Interest-only payments don't help much
- Paying early and often is wise
- Bankruptcy is a dire option
- Have a workable plan to pay it off
- Those deep in debt can't imagine life without it

Development slows down Soon even simple changes take too long and are accompanied by defects that slow the release of new features.

Interest-only payments won't help Checking in new code with high quality isn't enough. Developers must test and clean up the existing code they touch to avoid entrenching bad code.

Paying early and often is wise The best way to combat it is via TDD-supported refactoring (see Card 44, *A Rhythm for Success: The TDD Cycle*). Unit tests build confidence for continual, incremental code cleanup.

Bankruptcy is a dire option Debt can get so bad that the only option seems to be to rewrite the entire system. Rewrites tend to be unexpectedly slow and expensive. They seldom reach functional parity with the original.

Have a workable plan to pay it off Rework will require time and effort from developers, which has an opportunity cost. Plan payment timing and get buy-in from stakeholders before accruing technical debt.

Those deep in debt can't imagine life without it If you think software is always full of hacks and unneeded features, you may be an addict. Beat the mind-set, and eliminate your debt through collaborative refactoring (see Card 47, *Prevent Code Rot Through Refactoring*).

14 Incremental Everything

- Build/groom a prioritized feature backlog
- Select stories for iteration
- Develop in a timebox (one week to one month)
- Certify release candidate
- Release to production
- Retrospect
- Repeat

For an agile team, incremental development applies both to product and to workflow. Most teams begin with a project plan that follows the outline presented here.

The customer first puts together **a prioritized list**, or backlog, of desired features for upcoming product releases. The list is neither final nor comprehensive. Acceptance tests are written for the most immediately needed features.

Each iteration is a **fixed-length period** of development. At the outset of each iteration—no sooner—the **team and customer agree on what will be delivered.** During the iteration, the customer clarifies decisions and answers questions. At the end of each iteration, the team **certifies the release candidate** by demonstrating it passes all acceptance tests so far. The (partial) product may then be **released to production**. Also at iteration's end, a team holds a **retrospective** to determine how to improve the work system or the product so that future iterations will have higher quality and functionality.

The next iteration then **repeats** the whole process but includes the retrospective-inspired changes. These changes will evolve the software development practice even as the product grows. Agile teams are always trying to build a better future, both for the customer and for themselves.

15

Embrace Change

- Abandon
- Switch direction
- Defer before investing
- Grow

An agile project is not an all-or-nothing proposition and is not heavily vested in a planned future. It is planned incrementally as the product is being built and released. In *Extreme Programming Explained* [Bec00], Kent Beck described four options this style provides:

Abandon Skip further development or even abandon the product itself. Working software is provided in every iteration, and users gain early experience using it. It is easier to stop when it is "done enough" or if it doesn't seem profitable to continue.

Switch direction Change direction if the original plan doesn't match current needs. An agile project may accommodate change even late in the development process.

Defer before investing Do this so that the features with more immediate payback may be built first. Incremental design allows the organization to choose where, when, and whether to spend their software development money.

Grow Do this to take advantage of a market that is taking off. This includes building scalability, extending popular features, or driving the product into new problem domains. The business can continue development as long as new features will provide value to them.

16

Reach Consensus on Story Priority

- Simple Triage Rule
- Covey's quadrants
- Value first
- Risk first
- Greatest good to greatest number
- Fixed-length queue
- Bargain
- Alphabetical

Compose a **simple triage rule**, such as *bugs before features* or *system integrity before functionality.*

In *First Things First* [Cov94], Covey et al. suggest mapping significance vs. urgency in **quadrants**. Do significant urgent work first, followed by significant nonurgent work.

Maximize return by doing items of highest **value first** or minimize risk by doing tasks with greatest **risk first**. Alternatively, prefer tasks that offer the **greatest good to the greatest number**.

Work in a small **fixed-length queue** of tasks (based on velocity). If only three things can be done, which are the best three to do? What if it is only one?

Create consensus with a **bargaining** system. Give stakeholders each three votes per iteration. Allow multiple votes per story, vote trading, and deal making.

Implement stories **alphabetically**. It is arbitrary, even silly, but is still better than being blocked by indecision.

Remind all stakeholders that this is an agile work system, so prioritization is never final. It can be revisited and revised before each iteration.

INVEST in Your Stories

- **I**ndependent
- **N**egotiable
- **V**aluable
- **E**stimable
- **S**mall
- **T**estable

Customers describe their needs as briefly stated *stories*, best captured in a few words on index cards. Vet these candidate stories against Bill Wake's INVEST mnemonic (see http://xp123.com/xplor/xp0308). Fix them if they're not up to snuff.

Independent Your customer wants cool features now, not boring data input screens. Can you deliver? Yes! Each story is an independent, incremental need. You don't need the "add" story yet. Life isn't neat and orderly.

Negotiable A story is not a contract! It's a promise for more communication. Don't fill your story cards with every last detail.

Valuable Stories represent small bits of business value for your customer. Each implemented bit lets them see what you can deliver and provide feedback. Stories promising implementation of technical details (such as "build a database layer") provide no visible business value—never create them!

Estimable A reasonable estimate might not exist if a story is too big or if you don't know what's involved. Go back to the drawing board.

Small Stories are small, many fitting into an iteration and none approaching the full iteration's length. An ideal story would take your team a day to deliver.

Testable If you can't verify a story in some manner, you'll never know when it's done! Tests are often the best way to flesh out your understanding of a story.

Categorize Requirements with FURPS

- **F**unctionality
- **U**sability
- **R**eliability
- **P**erformance
- **S**upportability

As you mature in agile (see Card 27, *Shu-Ha-Ri*), consider ideas outside its realm, such as HP's FURPS requirements categorization scheme.[4]

Stories technically are not requirements; they are informal expressions of customer need. You can still categorize such needs using FURPS. Most significantly, FURPS reminds you that stories can represent more than just the immediate goals a user wants to accomplish by interacting with the system—the functionality.

More than with **F**unctionality stories, you must vet **-URPS** (excuse us!) candidates against INVEST characteristics of a good story (see Card 17, *INVEST in Your Stories*). For example, it's hard to define acceptance criteria for usability, but otherwise you have no idea when the story is done.

Functionality User features your customer wants, of course!

Usability Product effectiveness, aesthetics, documentation, and so on.

Reliability Failover, downtime, recovery, system accuracy, and so on.

Performance Max response time, throughput, memory consumption, and so on.

Supportability Testability, serviceability, monitorability, extensibility, and so on.

4. http://en.wikipedia.org/wiki/FURPS

19 Sail on the Three C's

- Card
- Conversation
- Confirmation

Cards are superb for story derivation and planning, but focus on the dialogue they create—not so much on their format or wording. Sail to success on Ron Jeffries' three C's (see http://xprogramming.com/articles/expcardconversationconfirmation).

Card Story cards are little physical things with purposefully insufficient space for every detail about what should be built. They are promises for subsequent communication that must occur. Ron says cards are tokens; the card isn't the real thing—it's a placeholder for the real thing.

Conversation So, what *is* the real thing? Collaborating to build and deliver software that meets customer needs! To succeed, you must converse continually. You must continue to negotiate supporting specifics for a story until all parties are in agreement and the software is delivered.

Confirmation A story's specifics must ultimately be clear to the customer and your team. The customer uses acceptance tests (ATs) to define this criteria. These ATs exercise the system to demonstrate that the implemented story really works. When all ATs for a story pass, the customer knows that the story is truly done—the software does what they asked.

Think of stories in a more ephemeral sense. Once you build the customer's need (represented by the story) in code, the story has been heard. Only the working feature and the tests that document how it behaves remain. Rip up that card!

20 Shrink XL Stories to Fit

- Defer alternate paths, edge cases, or error cases
- Defer supporting fields
- Defer side effects
- Stub dependencies
- Split operationally (for example, CRUD)
- Defer nonfunctional aspects
- Verify against audit trail
- Defer variant data cases
- Inject dummy data
- Ask the customer

Deliver completed stories in each iteration. Avoid XL (extra large) stories that "go dark," requiring many iterations before you can show completed product. Consider splitting off **alternate paths, edge cases, or error cases,** using acceptance tests to delineate each. (But remember, they must still show business value.)

In many cases, you can split off secondary details: **supporting data fields** (distill to a few core fields), **input validation** (assume the input data is perfect), or **side effects** (worry later about, for example, impacts to downstream feeds). You might similarly defer **nonfunctional constraints** (such as logging, graceful error handling, performance, or auditing) and **variant data cases** (for example, have a shipping system send all packages to the same address).

Here are some other options for putting those XL stories on a diet:

- "Fake it until you make it," by **stubbing dependencies** on third-party systems or by **injecting dummy data** and doing data collection stories later.
- **Split along CRUD (Create, Read, Update, Delete) boundaries**. You don't have to build "create" stories first!
- Use **audit trails (or logs) to verify** progress not visible to the user.

Finally, **ask your customer**—don't be surprised if they come up with an even better idea for breaking down your bloated stories!

21 Acceptable Acceptance Tests

Acceptance tests are used to verify that the team has built what the customer requested in a story.

- Are defined by the customer
- Define “done done” for stories
- Are automated
- Document uses of the system
- Don’t just cover happy paths
- Do not replace exploratory tests
- Run in a near-production environment

Are defined by the customer Acceptance tests (ATs) are an expression of customer need. All parties can contribute, but ultimately, a single customer voice defines their interests as an unambiguous set of tests.

Define "done done" for stories ATs are written before development as a contract for completion. Passing ATs tell programmers their work is done and tell customers that they can accept it.

Are automated You can script, and thus automate, all tests that define expected system capabilities. Manually executed scripts are a form of abuse.

Document uses of the system Design readable tests so they demonstrate, by example, valid uses of the system. Such documentation never becomes stale!

Don't just cover happy paths It's hard to capture all alternate and exceptional conditions. Inevitably you'll miss one. Add it to your AT suite.

Do not replace exploratory tests Exploratory testing highlights the more creative aspects of how a user might choose to interact with a new feature. It also helps teach testers how to improve their test design skills.

Run in a near-production environment ATs execute in an environment that emulates production as closely as possible. They hit a real database and external API calls as much as possible. ATs are slow by definition.

Acceptance Test Design Principles

Well-designed acceptance tests are

- **A**bstract
- **B**ona fide
- **C**ohesive
- **D**ecoupled
- **E**xpressive
- **F**ree of duplication
- **G**reen

Acceptance tests (ATs) are as enduring and important an artifact as your code. Their proper design will minimize maintenance efforts.

Abstract A test is readable as a document describing system behavior. Amplify your test's essential elements, and bury its irrelevant details so that non-technical staff can understand why it should pass.

Bona fide To ensure continual customer trust, a test must always truly exercise the system in an environment as close to production as possible.

Cohesive A test expresses one goal accomplished by interacting with the system. Don't prematurely optimize by combining multiple cases into one test.

Decoupled Each test stands on its own, not impacted by or dependent upon results of other tests.

Expressive A test is highly readable as documentation, requiring no research or analysis from its readers. Name it according to the goal it achieves.

Free of duplication Duplication increases risk and cost, particularly when changes to frequently copied behavior ripple through dozens or more tests.

Green Once a story is complete, its associated ATs must always pass. A failing production AT should trigger a stop-the-production-line mentality.

Story Estimation Fundamentals

- **A**ll contributors estimate
- **B**reak story into tasks to verify scope
- **C**ome to **C**onsensus with Planning Poker
- **D**ecrease granularity as story size increases
- **E**stimate with relative sizes

Estimates are always wrong—it's just a matter of degree! But dates are important to the business, and agile provides many chances to improve estimates over time.

All contributors estimate Those who build the software are the only ones who get to say how long they think it will take.

Break story into tasks to verify scope In agile, design and planning go together. Designing raises many questions that dramatically impact the plan.

Come to Consensus with Planning Poker This simple technique for deriving consensus on estimates keeps your planning meetings focused, lively, engaging, and on track. See Card 24, *A Winning Hand for Planning Poker*.

Decrease estimation granularity as story size increases Estimation accuracy rapidly diminishes once stories grow beyond a few hours or a day. Don't deceive yourself by attaching precise estimates to larger stories.

Estimate using relative sizes, not calendar days Once a bread box, always a bread box—a story's size doesn't change over time. Its size relative to other stories is also constant, regardless of developer capability and availability.

Remember: all contributors estimate, even in the most mature teams.[5]

5. See Card 27, *Shu-Ha-Ri*.

24

A Winning Hand for Planning Poker

1. Agree on a short size scale
2. Team briefly discusses a story
3. Everyone silently selects a point card
4. Team reveals all cards in unison
5. If outliers exist, discuss and re-vote

James Grenning's Planning Poker[6] is a fun, consensus-based technique to expedite your story estimation sessions. It requires **a short scale of story sizes**, such as 1, 2, 3, 5, 8, or XS, S, M, L, XL. The units are not days or hours but purely relative sizes: 1 being tiny, 2 being twice as large. The larger gap between higher-point values reflects the reality of diminished accuracy for larger estimates. Stories larger than the scale must be broken into smaller stories.

Each Planning Poker participant brings an estimation deck (perhaps hand-drawn) containing one card for each value in the scale. The customer and team **discuss and design a story** until all understand and agree on its meaning. Each team member who will help build the story then **secretly selects a card** representing their estimate of the story size. **All cards are revealed simultaneously**. The private selection and reveal prevent otherwise overly influential folks from dominating the estimates!

If everyone agrees precisely, move on. If not, **discuss the outliers** to find out the rationale behind lower or higher estimates. Timebox the discussions. If you can't agree upon a final estimate because of insufficient information, have a quick follow-up estimation session after any needed investigation. Otherwise, quickly select the more conservative estimate. Cheers! Move on.

6. See http://renaissancesoftware.net/papers/14-papers/44-planing-poker.html.

Iterate with Principle

- Team commits to stories they can complete
- Team works stories from prioritized card wall
- Team minimizes stories in process
- Team collaborates daily
- Customer accepts only completed stories
- Team reflects and commits to improvements

Team commits to stories they can complete Start each iteration with a short planning session to understand the customer's stories and determine what the team is confident they can deliver.

Team works stories from prioritized card wall The customer posts prioritized stories; your team selects the highest-priority story and moves it to an in-progress bucket. The card wall, real or virtual, is visible to everyone.

Team minimizes stories in process Your team works on the smallest sensible number of stories at once, maximizing completion rate so that many stories are truly done well before iteration's end.

Team collaborates daily Having few stories in process means you must collaborate frequently throughout the day. A sole stand-up meeting isn't enough.

Customer accepts only completed stories Stories must pass all programmer and acceptance tests. Anything less than 100 percent gets your team zero credit; incomplete work provides no business value. Don't over-commit.

Team reflects and commits to improvements Iterations are opportunities for incremental change, whether to improve quality, throughput, morale, or any other process and team concerns. Run a retrospective!

Everything else is implementation details, which are up to your team to determine.

Communication-SMITH with Information Radiators

Be **C**ommunication-**SMITHS** with information radiators that are

- **C**urrent
- **S**imple
- **M**inimal in number
- **I**nfluential
- **T**ransient
- **H**ighly visible
- **S**tark

Information radiators, aka big visible charts (BVCs), are wall hangings you design to broadcast important information. In yo' face! Your team won't have to proactively seek out their relevant info. (Unhip folks just call 'em posters.)

Current Stale information is soon ignored. Ensure it's current, or take it down.

Simple Dense or complex BVCs fail to communicate. Present information so people can understand and digest it in a few seconds.

Minimal in number Killing too many trees creates a thick, impenetrable forest on your wall. Also, exposing too many problems at once can be demoralizing.

Influential Worthwhile BVCs influence the team (and perhaps managers, customers, or other stakeholders) to improve how they work.

Transient BVCs that expose problems should be short-lived; otherwise, it's clear you're not solving your problems. Highlight challenges for which you can demonstrate progress in a few days or iterations.

Highly visible Radiating information requires your BVCs to be readable by everyone who enters the team area.

Stark BVCs don't exist to convince others of your team's excellence. Don't mask problems with them—use BVCs to display progress and expose problems.

27 Shu-Ha-Ri

守	**shu**	Embracing the kata
破	**ha**	Diverging from the kata
離	**ri**	Discarding the kata

Kata are movement patterns practiced by martial arts students *and* instructors. Shu-ha-ri are the three mastery levels for these movements. Embracing shu-ha-ri can help us understand why agile is much more than a set of simple practices.

shu You replicate the instructor's basic moves as precisely as possible, ingraining them until they are second nature. Wax on, wax off. You follow rigid rules for stand-ups to learn how to quickly communicate daily with your whole team.

ha You have some leeway to diverge a bit from strict application. Lightbulbs turn on as you begin to recognize the true usefulness of the skill. You begin to see how important it is to hear in the stand-up meeting what you are doing as a whole team. But you also realize that there's no reason for the team to wait for the next formal stand-up to quickly share information.

ri Rules are not constrictions; they are stepping stones to learning and freedom. You no longer *think* about rules—you simply apply the ingrained movements and can also specialize them based on prior experience. You may even forego formal stand-ups, for even greater success, if the whole team has learned to communicate frequently throughout each day.

Rules are meant to be broken but only after you've truly experienced them. Having mastered the rules, you are free to ignore them—particularly now that you know and accept the risks of ignoring them!

28

The Only Agile Tools You'll Ever Need

- White boards, flip charts, and markers
- Pairing and integration stations
- Index cards
- Toys and food
- Stickies and sharpies

What, not a single piece of software? If you and your whole team work together in a single room, you don't need high-end agile project management software.

White boards, flip charts, and markers Sketch designs and useful notes on the white board to broadcast them to all who enter. You'll find that very little of this information needs to survive beyond an iteration. Instead of archiving it on a computer, summarize on a flip chart page, and tape it to the wall.

Pairing and integration stations Developer comfort and a screamin' machine on which to pair are of utmost importance—with, of course, the best developer environment available to build and test the product.

Index cards Cards are our ubiquitous idea bulbs. They stack, tack, and sort quite well and are just small enough to prevent us from writing too much down.

Toys and food Distractions help us relax and focus; snacks keep us energized. Concerned about the cost? Contrast it with a license for a high-end agile tool that you don't need. There is no requirement for the team to be unhappy.

Stickies and sharpies Little reminders are quicker and more visible than emails or other software-based solutions. And you can never have enough pens at your disposal.

If you have a good coach, you won't have to ask for refills for these items.

Successful Stand-up Meetings

- All team members stand for the duration
- Each team member shares with the team
  - What did I accomplish yesterday?
  - What will I accomplish today?
  - What obstacles do I have?
- Team coordinator notes obstacles (to be cleared before the next stand-up meeting)
- Break into follow-up meetings as needed

A stand-up meeting is the first opportunity for the team to get together and begin collaboration for the day. Successful stand-ups are short and to the point and generally follow the rules shown on this card. You actually do stand to help constrain the meetings to five to ten minutes.

Beyond the rules we show here, create your own rules to help keep stand-ups lively but to the point. For example, some teams toss around a token indicating who has the floor. One team we met required newcomers to sing for the group (Jeff sang *I Feel Pretty*).

Stand-ups are designed for the day-to-day team. It's important that the team prevent others from derailing the focus of the meeting. Scrum dogmatically requires outsiders to be silent, for example, but you could simply ask offenders to stifle it.

Having tedious stand-ups? Your team is likely avoiding collaborative work. If we each work a story alone, there's little reason to converse or even listen to one another. You may as well just email a status report and save the time of a stand-up. Better yet, work as a team and collaborate on delivering a new story every day or so if possible. Your stand-ups will be far more interesting!

Remember that stand-ups are only a starting point for daily team conversation. The meaty dialogue must occur afterward, throughout the day.

ABCs of Pair Programming

- **A**ll production code

  …must be developed by a pair
- **B**oth parties contribute to the solution

  …switching roles between “driver” and “navigator” frequently
- **C**hange pairs frequently

  …once to three times per day
- **D**evelop at a comfortable workstation

  …that accommodates two people side by side
- **E**nd pairing when you get tired

  Constrain to no more than three-fourths of your workday

Pair programming—two programmers jointly developing code[7]—can be an enjoyable practice that results in better-quality solutions. Follow our ABCs of pairing to avoid applying it as a practice that's neither effective nor fun.

Pairing on production code is a must—it's a form of review—but you can still work solo on other tasks. See Card 32, *When Not Pairing*.

Pairing is not one person watching over another programmer's shoulder. That seems like it would be a waste of time! **Pairing instead is "two heads" collaborating on a better solution** than either alone would produce.

The least obvious and most abused rule is that **pairs should *not* remain joined at the hip** for days at a time or more. That'd just result in more workplace violence! Instead, ensure that at least three people, not just two, contribute to and review a solution. This creates some context switching overhead, but adherence to agile quality practices (particularly TDD and refactoring) will minimize the overhead.

Pairing must be enjoyable and comfortable to be sustainable. Make sure your environment isn't causing you pain, and **don't overdo it**. We advise also keeping refreshing mints or gum on hand.

7. Lisa Crispin reminds us that pairing also works well for other efforts, including designing tests.

31 Retrospectives

- **Set the stage**: Get everyone to speak. Agree on rules. Use a *safety exercise*.
- **Gather data**: Feelings are legitimate data!
- **Generate insights**: "Why?" Begin discussing how to do things differently.
- **Decide what to do**: Commit to one to two action items or experiments.
- **Close the retrospective**: Review the retrospective itself. Capture info. Thank all.

Retrospectives are regular team opportunities to reflect on our past performance and then commit to adapt. The retrospective meeting is a great tool for managing continuous improvement, an essential quality of "being agile." These steps will help keep your meetings running smoothly.

Start a retrospective meeting by using a safety exercise that anonymously determines how comfortable everyone is with speaking openly. Next gather data about what occurred over the last relevant period (usually an iteration or release). Just the facts, ma'am—prevent participants from predetermining solutions. The team can then analyze the facts to uncover the underlying problem they must tackle. Only then should they discuss solutions and plan a course of action.

By meeting end, the team must *commit to* one or more concrete changes to how they work. These "change stories" require acceptance criteria–specific goals to be met by a certain time. Consider using SMART (Specific-Measurable-Attainable-Relevant-Time bound) goals to vet the quality of a change story.

Accept a change story only if measurements show it meets the acceptance criteria. In other words, track and test it.

Absolutely read *Agile Retrospectives* [DL06], which includes the meeting structure outlined here and exercises to keep your meetings lively, sane, and useful.

32 When Not Pairing

- Build nonproduction code[8]
- Create meaningful and lasting documentation
- Work on spikes for future stories
- Learn a new tool or technique
- Identify production code needing refactoring
- Refactor tests
- Improve existing test coverage

8. Test frameworks, tools, the build, and so on

It's not always possible to pair, even in shops where pairing is standard practice. Realities such as variant working hours and odd numbers of programmers make full-time pairing difficult. For these reasons alone, it's probably a bad idea for your team (or worse, someone outside the team) to mandate 100 percent pairing on everything.

Card 30, *ABCs of Pair Programming*, clearly defines when you *must* pair: when developing any production code. Refer to the other side of *this* card for a list of what you might do, other than twiddle your thumbs, when no pair partner is available.

Your team should derive ground rules for work done during pair-less times. Start with this list. Sometimes, unfortunately, you'll need to change production code, but let that happen only rarely. Your team should derive ground rules for this circumstance. Since the primary goal of pairing is review, some form of follow-up review is the most sensible place to start.

As much as we enjoy pairing, we view our occasional pair-less times as opportunities to do something better about our lot in the code's life. We relish the chance to finally add those missing tests we wish we'd had or to clean up some of the ugliness in the build.

How to Be a Team Player

- **Cooperation**: Focus on shared goals
- **Information**: Feedback loops closed
- **Humanity**: Respect for struggles of others
- **Equality**: Recognition of peers
- **Energy**: Common effort toward common problems

Agile teams require close collaboration, but transitioning agile teams often struggle because of poor working relationships among team members. With these five shared values, you will find that you can usually put aside personal issues and past history to work successfully together.

Cooperation Focus on the work you do together, and try to make the team successful instead of jockeying for position

Information Look for ways to determine whether the new work system is producing good results. Prepare to give and receive honest feedback. Be increasingly transparent.

Humanity Teach and learn. Be humane to teammates by being patient when your strengths are not equal or their circumstances and training differ from your own.

Equality Neither a take-credit nor a blame-shoveler be.

Energy Do not exhaust the people you work with. Keep up. Work hard when working, and then go home after seven to nine hours.

Collective Code Ownership

- Anyone can modify any code at any time
- The team adheres to a single style guide
- Original authorship is immaterial
- Abundant automated tests create confidence
- Version control provides insurance

Anyone can modify any code at any time You may improve any section of the code without wasting time seeking permission. You're not a cubed individual piling up a code fiefdom, but part of a team collaborating on a deeply connected system.

The team adheres to a single style guide The points in Card 35, *Coding Standards*, reduce the wasteful friction of learning a new standard when working elsewhere in the system. A common style guide lets you spend time solving problems that matter. Your IDE might be able to help here.

Original authorship is immaterial Personal attachment to "your" code provides no additional value when your team all strives toward the same goal of high-quality code. Lose the pride over code you just created, substituting enthusiasm for figuring out how to make it even better.

Abundant automated tests create confidence TDD provides abundant tests that both declare and protect the original programmer's intent, giving you the freedom to refactor with impunity.

Version control provides insurance. The use of a good version control system means you can return to a previous (working, tested) version of the system, making code experiments affordable.

Coding Standards

- Standardize to avoid waste
- Start with an accepted standard
- Timebox the debate
- Fit on one page
- Revisit regularly until no one cares
- The code becomes the standard

"We don't need no stinkin' standards!" Yes you do, unless you want your team to waste time with careless, inconsistent effort.

Standardize to avoid waste Inconsistent code generates silly waste: slower comprehension time, arguments, and rework by annoyed programmers. Worse waste comes when someone misinterprets code and introduces a defect.

Start with an accepted standard Surprise, others have debated coding standards for countless hours, long before you. Take advantage of existing community or even IDE standards (and save the angst of going against their grain).

Timebox the debate An initial debate should take no longer than an hour. The goal is to obtain consensus, not to determine the One True Way. Don't allow wavering from your teammates on standards, only tightening.

Fit on one page Do not try to build a comprehensive guide to programming. Keep it small, simple, and unambiguous.

Revisit regularly until no one cares Increment and improve the standard as needed using retrospectives, just as you would any agile product. Eventually the topic will bore everyone.

The code becomes the standard You will know the style guide is unnecessary when all code looks alike and it all looks good!

36

Is Your Team Circling the Drain?

- Individual work assignments
- Piles of unfinished work
- Work assignments given under the table
- Empty ceremonies
- Neglecting quality practices
- Guarded speech

If a team has had trouble completing tasks well and on time in the past, it will have increased pressure from above. The urge to hold individuals accountable results in **individual work assignments**. To motivate fervent effort, work is piled on, resulting in **piles of unfinished work**.

Since outside parties have trouble getting work done through official channels, they find ways to pass **work assignments under the table**. This makes it less likely that official tasks will be completed (Gerald Weinberg's Law of Raspberry Jam is "The more you spread it, the thinner it gets").

Since the team is not truly collaborating, meetings become **empty ceremonies**. Overloaded team members **neglect quality practices** (such as TDD, testing, or CI) in a desperate bid to get tasks off their plates. As team members become bitter and disappointed, their public **speech becomes increasingly guarded**.

The development team needs to be rebooted so it can build a new reputation using an agile work system (we recommend hiring a capable agile coach). It needs a "whole-team" approach, with limited work in progress and no under-the-table assignments. Those in these circumstances are reminded of these wise words: *change your organization, or change your organization.*

Pair Programming Smells

- Unequal access
- Keyboard domination
- Unhealthy relationships
- Worker/rester
- “Everyone does their own work”
- Endless debate

Sniff out and correct these smells that prevent successful pairing.

Unequal access Ensure tight cube setups don't require one partner to sit behind another. Pairing is two people collaborating, not one watching another.

Keyboard domination Domination can occur when one partner dismisses the other, verbally or otherwise. A coach (perhaps Cesar Millan) must intervene if the dominated party remains silent.

Unhealthy relationships Pairs should change frequently enough to avoid stagnation ("pair marriage"). Pair enemies are a more serious problem, requiring remediation or even reorganization.

Worker/rester Watch for disengaged partners (a second computer is an obvious indicator). Take breaks, switch partners, and coach habitual resters.

"Everyone does their own work" The whole team is responsible for collaborating (and not abandoning each other) to complete the work. A manager discourages pairing when they hold *individuals* accountable for tasks.

Endless debate People unskilled at pair programming will over-analyze how to derive a solution. Any debate longer than ten minutes should be resolved via code. Arguing in code is better than arguing about code.

38 Stop the Bad Test Death Spiral

- Only integration tests are written → Learn TDD.
- Overall suite time slows → Break into “slow/fast” suites.
- Tests are run less often → Report test timeouts as build failures.
- Tests are disabled → Monitor coverage.
- Bugs become commonplace → Always write tests to “cover” a bug.
- Value of automated testing is questioned → Commit to TDD, acceptance tests (ATs), refactoring.
- Team quits testing in disgust → Don’t wait until it’s too late!

Each misstep with TDD may lead down a bad spiral[9] toward its abandonment.

Learn TDD Violating cohesion and coupling principles impedes unit testing and promotes integration tests instead. TDD drives a unit-testable, SOLID[10] design.

Break into slow/fast suites A fast suite runs in ten seconds. Keep the slow suite small. Use a continuous test tool such as Infinitest. Keep coupling low.

Report test timeouts as build failures Continually monitor the health of your test suite. If the suite slows dramatically, developers soon skimp on testing.

Monitor coverage Seek coverage above 90% on new code and stable/increasing coverage on existing code. Recast integration tests as unit tests or ATs.

Always write tests to cover a bug Test first, of course. Defects indicate inadequate test coverage. Track and understand each defect's root cause!

Commit to TDD, ATs, refactoring Do TDD *diligently*. Many bugs are rooted in duplication that you must factor out. Quality problems slow production!

Don't wait until it's too late! If you admit defeat, it may be too late—managers rarely tolerate second attempts at what they think is the same thing.

9. SCUMmy Cycle, Agile2009 presentation by B. Rady and R. Coffin
10. See http://en.wikipedia.org/wiki/Solid_(object-oriented_design).

39 How to Stay Valuable

- Stay positive
- Stay engaged
- Stay professional

Be valuable to your employer, whether times are good or bad and whether you're talking about your current employer or your next one.

Stay positive A positive attitude is crucial to building goodwill, inspiring others, and having fun. Staying positive when things are difficult is not just good citizenship; it is leadership.

Stay engaged You are not effective when you are not participating in the work of the team. Focusing every day is hard, so use pair programming and perhaps the Pomodoro Technique (see *Pomodoro Technique Illustrated* [NÖ9]) to help keep your head in the game.

Stay professional As long as you are on the job, you are an ambassador of your profession. Learn and model those attitudes, skills, and techniques that improve the quality of work for the team (including customers and managers). As much as possible, be helpful and diligent in your work. Remember that you are building a reputation, and make it a good one.

Eight Crucial Practices of Agile Programmers

- Continuous integration (CI)
- Test-Driven Development (TDD)
- Constant design improvement
- Coding standard
- Collective code ownership
- Simple design
- System metaphor
- Pair programming

Agile programmers use these eight practices that derive from Extreme Programming (see *Extreme Programming Explained, First Edition* [Bec00]) to improve their software development practice. The practices can work individually but have a profound synergy when used together.

A **coding standard** (see Card 35, *Coding Standards*) is foundational for working as a team that **collectively owns the code**. **Continuous integration** ensures the system is always in working order. You know the system works because it has been built by **pairs of programmers** reviewing the other person's work and using **TDD** (plus automated acceptance tests) to verify that it works. You know the system has a quality design, because the programmers use **TDD** to **continually improve the design (refactor)** using **simple design** concepts.

You communicate these system ideas with your whole team—which includes the business and other nontechnical folks—through use of a **metaphor**. A metaphor is a shared system vision and language that simplifies decision making and reduces the need for deep-code spelunking. The shopping cart is a common metaphor; you talk about "carts, items, and checkout." You might end up with what's known as the *naïve metaphor*: "This is a purchasing system, with orders, inventory items, and submissions." Boring but effective enough!

41

Build Superior Systems with Simple Design

- All tests must pass
- No code is duplicated
- Code is self-explanatory
- No superfluous parts exist

As far as a prescriptive set of design principles go, these four rules[11] of simple design, listed in order of importance, are the simplest thing that might possibly work.

- Code must work and be demonstrated by passing tests. A great paper model of a design is useless if the software doesn't work. Use TDD so that you know the code is working at all times.
- Removing duplication is the next most important design activity, because it directly lowers the cost of system maintenance and drives you toward an appropriate level of abstraction.
- Contrary to popular belief, it's almost always possible to create code that others can understand and maintain—particularly if your TDD-created tests clearly document intended behavior. Maintenance costs rise dramatically with unclear code.
- Finally, YAGNI (You Aren't Going To Need It) tells us to remove all superfluous features and complexity—anything not needed at this precise moment.

11. *Extreme Programming Explained, First Edition* [Bec00]

The Seven Code Virtues

- Working, as opposed to incomplete
- Unique, as opposed to duplicated
- Simple, as opposed to complicated
- Clear, as opposed to puzzling
- Easy, as opposed to difficult
- Developed, as opposed to primitive
- Brief, as opposed to chatty

Working, as opposed to incomplete A program that works is superior to one that might work later. TDD requires code to work early and often.

Unique, as opposed to duplicated Many regression errors are the result of improvements to only one version of copied code. Eliminate duplication in TDD's refactoring step.

Simple, as opposed to complicated Simple code hides fewer bugs and tends to optimize well.

Clear, as opposed to puzzling Code misunderstandings generate errors. Use Card 43, *Really Meaningful Names*, and simple structures to make such errors unlikely.

Easy, as opposed to difficult Structure and organize code so that it is easy to use. Don't hesitate to add convenience methods and classes.

Developed, as opposed to primitive Prefer the clarification and encapsulation of abstract data types to built-in variable types. Primitive obsession muddles the code and can create maintenance nightmares.

Brief, as opposed to chatty Code should be brief but never cryptic. All other virtues still observed, less code is always better than more.

Really Meaningful Names

- Are accurate
- Are purposeful
- Are pronounceable
- Begin well
- Are simple
- Depend on context
- Match name length to scope

Are accurate Do not mislead the reader. Naming a user's postal code birthDate might infuriate co-workers! Characters like *O* and *l* can fool readers into seeing the digits 0 and 1.

Are purposeful Describe the intended use of a thing, not its origin or composition. Variable names like text and psz lack intention.

Are pronounceable Pronounceable names are easily recognized and discussed. We want to talk about the code without sounding crazy.

Begin well Leading differences stand out, but differences in the middle or end of names can be hard to spot.

Are simple Long names are more descriptive but less distinct than short ones. Encoded prefixes and suffixes make differences between names subtle. Don't try to cram too much meaning into a name.

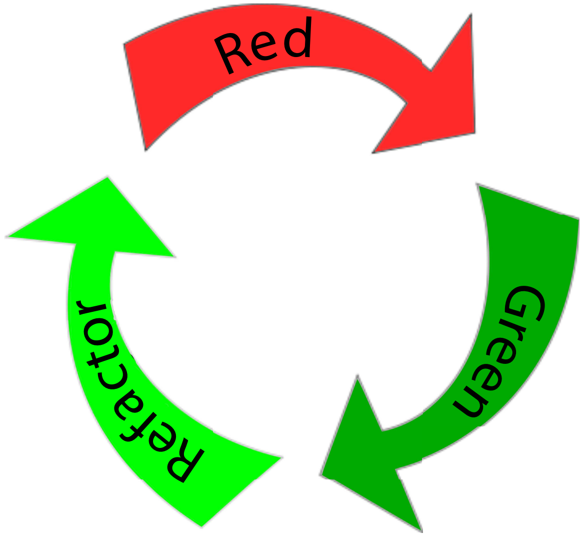
Depend on context The fully qualified name is the real name. The local name is just the nickname.

Relate name length to scope Names in very small scope (short loops, closures) can be small. Names visible from many places must be extra descriptive.

See *Clean Code* [Mar08] for Tim's exhaustive guide to naming.

44

A Rhythm for Success: The TDD Cycle



Of the agile developer practices, TDD will change your coding approach most fundamentally. TDD is a rhythm that helps you continually keep steady, forward progress. In the TDD cycle, you produce enough test code to fail, produce enough code to make the test pass, and then refactor before writing the next microtest. Red and green are the colors shown by GUI test tools for failing and passing tests.

Red Newbies commonly make the mistake of not watching tests fail first. The red is essential feedback that the tests are written correctly and recognized by the test runner.

Green Making the test pass is the part that is obvious; the hard part is that you write only enough code to make the test pass. No more! By limiting the amount of coding, you force the tests to be more complete, you avoid the cost of an over-designed system, and you learn how to incrementally grow behavior.

Refactor While the test is passing, before anyone can depend on its current implementation, clean it. Rename a test. Improve readability. Extract an interface or a class. Unlike the Green step, there are no limits on the amount of refactoring you can do per step, as long as the tests always pass. Refactoring is critical to survival; it is your main opportunity to ensure your system's design stays clean.

FIRST Properties of Unit Tests

- **F**ast
- **I**solated
- **R**epeatable
- **S**elf-verifying
- **T**imely

Fast Tests should be *really* fast. If the entire unit test suite takes a minute, people will be reluctant to run it. Break dependencies to make tests profoundly fast and small.

Isolated When a test fails, it should be for a single, obvious reason. A long, complex test may fail in many places for many reasons. Isolated tests can run alone or in a suite, in any order.

Repeatable Tests can be run in a loop, without intervention. They will continue to fail or succeed until code is changed that breaks them. They do not leave the system in a state that will not allow them to run again.

Self-verifying Unit tests are pass/fail. No interpretation is needed to determine success. If the test fails, it states why it failed.

Timely Unit tests are written with the code, not after the code is completed. In TDD style, tests are written first. Your best results will always come from following the Red/Green/Refactor cycle.

Source: Brett Shuchert, Tim Ottinger

Triple A for Tight Tests

- **Arrange** all the things needed to exercise the code
- **Act** on the code you want to verify
- **Assert** that the code worked as expected

Arrange-Act-Assert (AAA), a simple concept devised by Bill Wake of xp123.com, provides subtle power by helping test readers quickly understand the intent and flow of a test. No more “What the heck is this test even testing?”

In a unit test, you first create, or **arrange**, a context; then execute, or **act** on, the target code; and finally **assert** that the System Under Test (SUT) behaved as expected. Thinking in terms of AAA has direct impact on the code’s visual layout.

The simplest (perhaps ideal) test has three lines, one for each *A*. Here’s an example where we must verify that the SUT applies late fines to a library patron:

```
@Test public void applyFine() {
  Patron patron = new Patron();           // arrange the context
  patron.setBalance(0);

  patron.applyFine(10);                   // act

  assertEquals(10, patron.fineBalance()); // assert
}
```

You don’t need the comments—the blank lines alone are sufficient.

AAA (also known as *given-when-then*) isn’t an absolute rule. You might not need an Arrange, and it’s OK to combine an Act and an Assertion into a single-line test.

47 Prevent Code Rot Through Refactoring

- Refactor only while green
- Test constantly
- Work in tiny steps
- Add before removing
- Test extracted functionality
- Do not add functionality

Start with all tests passing (green), because code that does not have tests cannot be safely refactored. If necessary, add passing characterization tests so that you have a reasonable green starting point.

By working in tiny steps and running tests constantly, you will always know whether your last change has broken anything. Refactoring is much easier if you insist on always having only one reason to fail.

By adding new code before removing old code (while testing constantly), you ensure you are not creating blocks of untested code as you work. For a while, your code will be half-refactored, with the code of old and new implementations of a behavior present. Each act of refactoring takes several edit/test cycles to reach a clean end state. Work deliberately and incrementally.

When extracting classes or methods, remember that they may need unit tests too, especially if they expose behaviors that were previously private or hidden.

Be watchful of introducing changes in functionality. When tempted to add some bit of functionality, complete the current refactoring and move on to the next iteration of the Red/Green/Refactor cycle.

Refactoring Inhibitors

- Insufficient tests
- Long-lived branches
- Implementation-specific tests
- Crushing technical debt
- No know-how
- Premature performance infatuation
- Management metric mandates

Agile's demand for continual change can rapidly degrade a system's design. Refactoring is essential to keeping maintenance costs low in an agile environment.

Insufficient tests TDD gives you the confidence to refactor and do the right thing, when you would not otherwise for fear of breaking existing code.

Long-lived branches Working on a branch, you'll plead for minimal trunk refactoring in order to avoid merge hell. Most branches should be short-lived.

Implementation-specific tests You may want to discard tests, ultimately killing refactoring, if small refactorings breaks many tests at once. Minimize test-to-SUT encapsulation violations, which mocks can create.

Crushing technical debt Insufficient refactoring creates overwhelming rampant duplication and difficult code. Refactor on every green to minimize debt.

No know-how You can't refactor if you don't know which direction to take the code. Learn all you can about design, starting with simple design.

Premature performance infatuation Don't let baseless performance fear stifle cohesive designs. Make it run, make it right, and only then make it fast.

Management metric mandates Governance by often one-dimensional and insufficient metrics (such as *increase coverage*) can discourage refactoring.

Field Guide to Mocks

- **Test double**: Any object that emulates another
- **Stub**: Returns a fixed value to the SUT
- **Fake**: Emulates a production object
- **Mock**: Self-verifies
- **Partial mock**: Combines production and mock methods
- **Spy**: Records messages for later verification

Test doubles are emulation objects you build to simplify testing. Surrounding the System Under Test (SUT) with test doubles allows you to exercise and observe it in isolation. Otherwise, accessing database, network, external hardware, or other subsystems can slow or break the test run and cause false positives. **Stubs**, **fakes**, **mocks** (partial or not), and **spies** are nuanced kinds of test doubles. This field guide, based on *XUnit Test Patterns* [Mes07], will help you sort out standard terminology (used in many mock tools).

Test doubles have changed the art of test-driving by allowing tests to be smaller, simpler, and faster running. They can allow you to have a suite of thousands of tests that runs in a minute or less. Such rapid feedback gives you the confidence to dramatically change your code every few minutes. But take care; you should do the following:

- Learn to use test doubles, but employ them only when you *need* the isolation.
- Use a mock tool (instead of hand-coding them) if it improves test quality.
- Learn the various types of mocks summarized in this field guide.[12]
- Read Card 51, *Test Double Troubles*, to avoid their many pitfalls.

12. If only to avoid being embarrassed by your peers. Or just keep this card handy.

Break Unit Test Writer's Block

- Test that you can call the method at all
- Pick the most interesting functionality
- Pick the easiest functionality
- Write an assertion first
- Rename or refactor something
- Switch programming partners
- Reread the tests and code

Software authors can also get writers block. A good trick to beating it is to do some useful programming activity to get the creative juices flowing.

In legacy code, **calling a method at all** can require considerable setup, mocking, and dependency breaking. These activities get you entrenched in the code and activate your urge to clean up the code base.

Let your curiosity guide you. Test the **most interesting** bit of the functionality you're writing. You'll have more energy to do work that interests you most.

Alternatively, **pick the easiest** functionality so that you experience some success and build momentum. "Simple is as simple does."

When writing a method normally (begin-to-end) fails, try **writing the assertion first** and working backward from there.

If you **rename or refactor something**, you'll have your head wrapped around the problem more and end up with a code improvement to boot!

If the flow of ideas is getting stale, "a change is as good as a break," so **switch programming partners.**

You may be stuck because you don't really understand the code you need to change. Help your brain: **reread the tests and code** that already exist.

Test Double Troubles

- Inhibited refactoring
- Tool complexity
- Passing tests that indicate nothing
- Mocking the SUT
- Low readability
- Ambitious mock implementations
- Vendor dependency

Inhibited refactoring Test doubles exploit the linkages between classes, so refactoring class relationships may cause mock-based tests to fail.

Tool complexity Mock tools have extensive APIs and can be idiosyncratic. It's costly to understand both the tool and the code it produces.

Passing tests that indicate nothing Fakes and stubs may not act quite like the classes they replace, leading to tests that pass and code that fails.

Mocking the SUT Complex setup can bury the embarrassing fact that code we need to be real has been replaced with mock code, invalidating the test.

Low readability Mock setup can be dense and idiomatic, making it difficult to see what's really being tested.

Ambitious mock implementations A fake is an alternative implementation of the class it replaces and can have bugs of its own. Keep your test doubles small and purposed to a single test or test fixture.

Vendor dependency All third-party tools eventually fall out of favor, replaced with something newer and better. Don't get stuck with yesterday's tool.

To remedy most of these challenges, keep your use of mocks isolated and minimal. Refactor tests to emphasize abstraction and eliminate redundant mock detail.

TDD Process Smells

- Not failing first
- No green bar in last ten minutes
- Skipping the refactoring step
- Skipping easy tests
- Skipping hard tests
- Not writing the test first
- Setting code coverage targets

Not failing first It's tempting to go right for green (passing), but an initial red (failing) test tells you that the *test* works.

No green bar in last ten minutes You're not relying on your tests anymore if it's been a while since you've seen green. Is it ever a good time to *not* know whether you've broken something? Take smaller steps.

Skipping the refactoring step It's a truly bad idea to make a mess "for now" and promise to clean it up in the mythical "later" (Robert Martin has addressed this fallacy in his many keynote addresses).

Skipping easy tests You won't save much time by not testing simple code, and you'll possibly create silly mistakes that go undetected in the code.

Skipping hard tests Difficulty in testing drives us to reconsider and improve our design (to something that is also easier to test!).

Not writing the test first TDD requires you to drive your code changes from the outside. Writing tests later makes code harder to test and hard to refactor for testing.

Setting code coverage targets This is a well-understood failure because of the Hawthorne Effect (roughly, "you get what you measure"). It is better to discourage artificial inflation (gaming) of metrics.

Resources

[Bec00] Kent Beck. *Extreme Programming Explained: Embrace Change.* Addison-Wesley, Reading, MA, 2000.

[Cov94] Stephen R. Covey. *First Things First.* Simon and Schuster, New York, 1994.

[DL06] Esther Derby and Diana Larsen. *Agile Retrospectives: Making Good Teams Great.* The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2006.

[Mar08] Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship.* Prentice Hall, Englewood Cliffs, NJ, 2008.

[Mes07] Gerard Meszaros. *xUnit Test Patterns: Refactoring Test Code.* Addison-Wesley, Reading, MA, 2007.

[NÖ9] Staffan Nöteberg. *Pomodoro Technique Illustrated.* The Pragmatic Programmers, LLC, Raleigh, NC, and Dallas, TX, 2009.

About the Authors

Jeff Langr has been building software for more than a quarter century. He is the author of *Agile Java* and *Essential Java Style*, and more than 80 articles on software development and a couple of chapters in Uncle Bob's *Clean Code*. Jeff enjoys helping others learn how to build software well, but is happiest when he gets to code.

Tim Ottinger has more than 30 years of software development experience, including time as an agile coach, OO trainer, contractor, in-house developer, and even a little team leadership and management. He is also a contributing author to *Clean Code*. He writes code. He likes it.