Extreme programming

XP

6

Programming

- Simple Design
- Design Improvement
- Test Driven Development
- Coding Standards

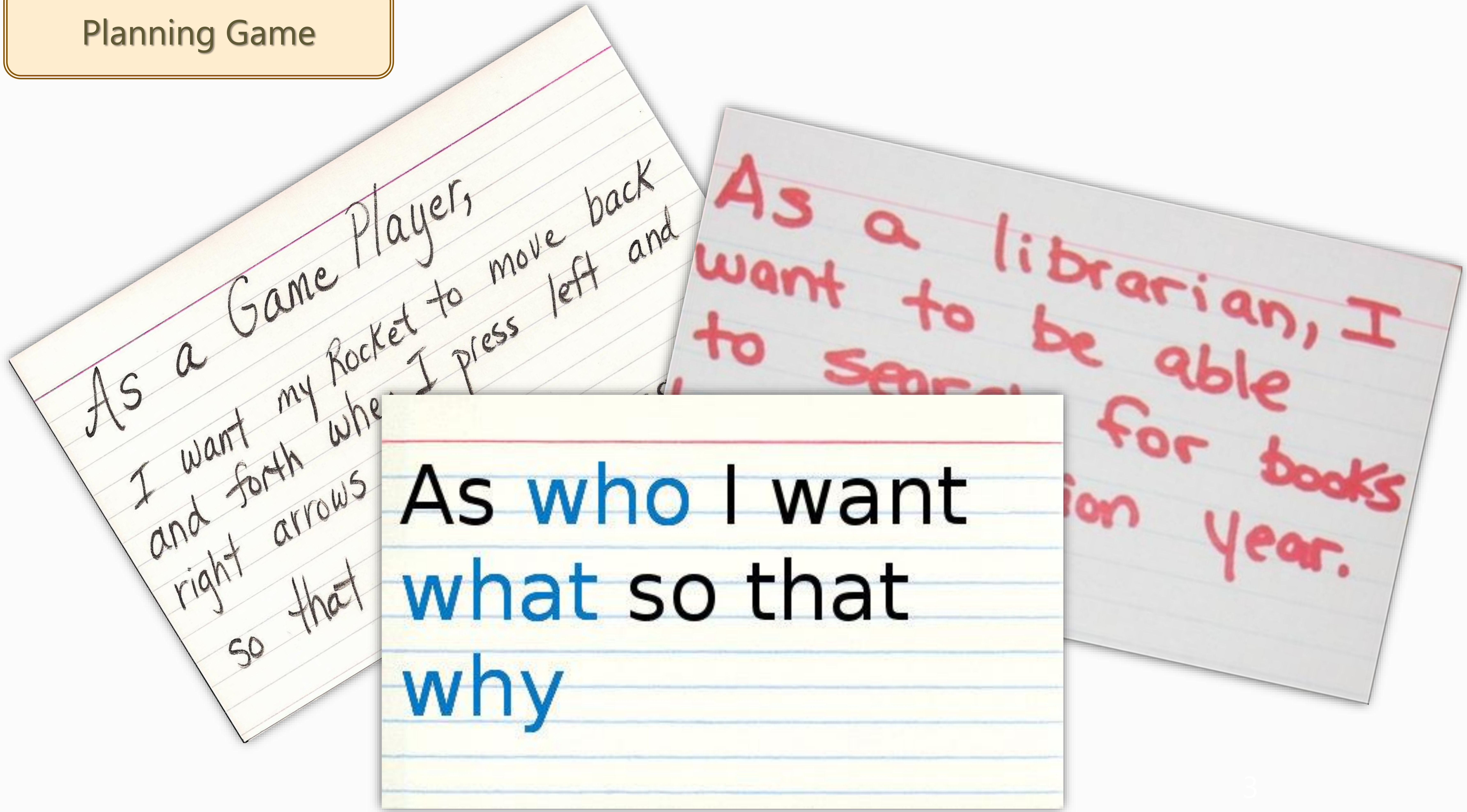
Team

- Sustainable Pace
- Collective Ownership
- Coding Standards
- Pair Programming
- Continuous Integration
- System Metaphor

Processes

- Whole Team
- Test Driven Development
- Small Releases
- Planning Game

Planning Game



As a Game Player,
I want my Rocket to move back
and forth whe I press left and
right arrows
so that

As a librarian, I
want to be able
to Sear for books
ion year.

As who I want
what so that
why

Planning Game

Independent: Stories should be as independent as possible. When thinking of independence it is often easier to think of “order independent.” In other words, stories can be worked on in any order. Why is this important? It allows for true prioritization of each and every story.
Negotiable: A story is not a contract. A story IS an invitation to a conversation. The story captures the essence of what is desired. The actual result needs to be the result of collaborative negotiation between the customer, developer and tester (at a minimum).
Valuable: If a story does not have discernable value it should not be done. Period. Hopefully user stories are being prioritized in the backlog according to business value, so this should be obvious.
Estimable: A story has to be able to be estimated or sized so it can be properly prioritized. A value with high value but extremely lengthy development time may not be the highest priority item because of the length of time to develop it. What happens if a story can’t be estimated? You can split the story and perhaps gain more clarity.
Small: Obviously stories are small chunks of work, but how small should they be? The answer depends on the team and the methodology being used. I teach agile and suggest two week iterations which allow for user stories to average 3-4 days of work – TOTAL! This includes all work to get the story to a “done” state.
Testable: Every story needs to be testable in order to be “done.” Thinking this way encourages more collaboration up front, builds quality in by moving QA up in the process, and allows for easy transformation to an acceptance test-driven development (ATDD) process.

Planning Game

Business	Technical
Define the scope of the release	Estimate how long each user story will take
Define the order of delivery (which stories are done first)	Communicate technical impacts of implementing requirements
Set dates and times of release	Break down user stories into tasks and allocate work

Small Releases

The only way to ensure that you are developing the software the customer expects!

Every release could be used as a checkpoint to measure the estimation accuracy.

Small Releases

Fail Fast

Fail Often

Small Releases

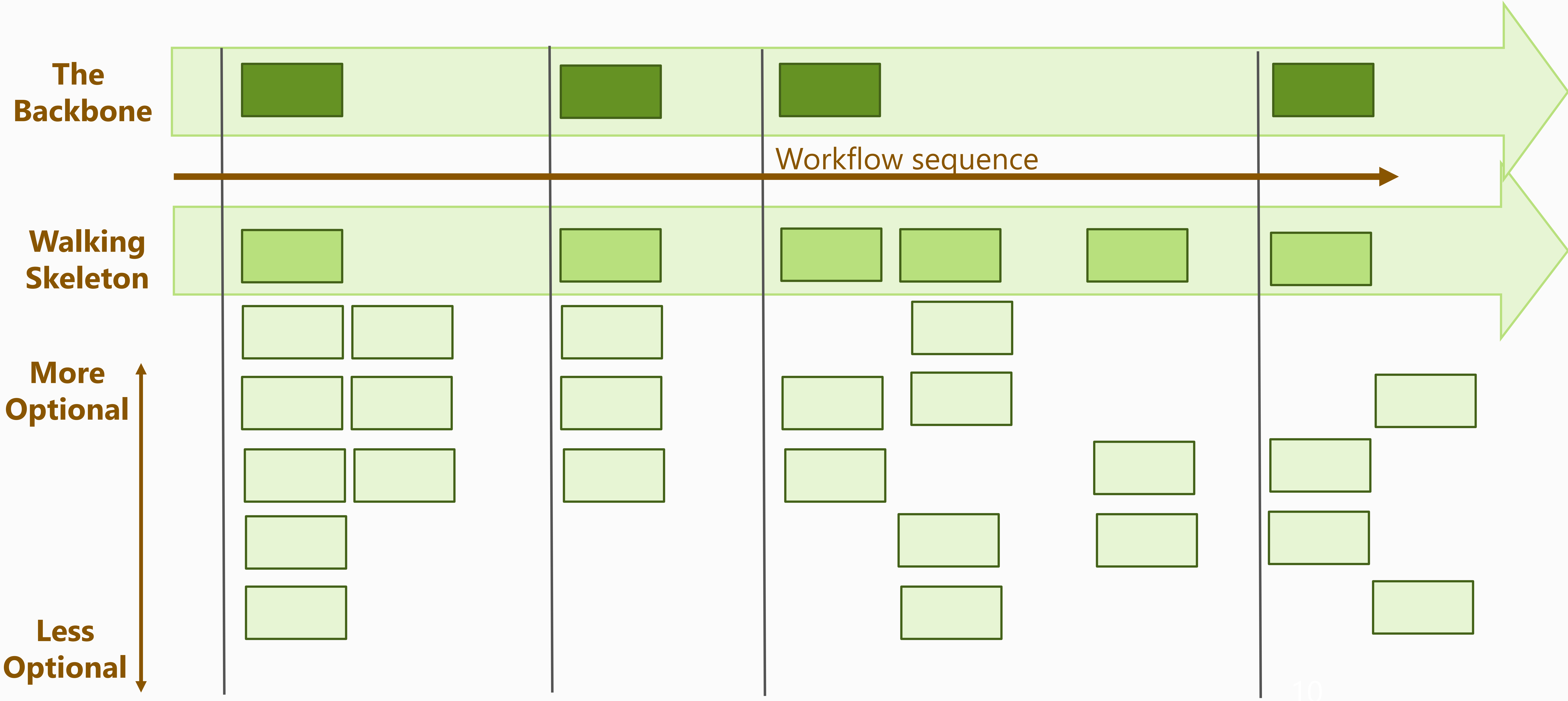
- Product Roadmap =

  visual overview of a product’s releases
  - Product roadmap = sequence of releases
  - Release = sequence of iterations
  - Iteration = set of user stories / features

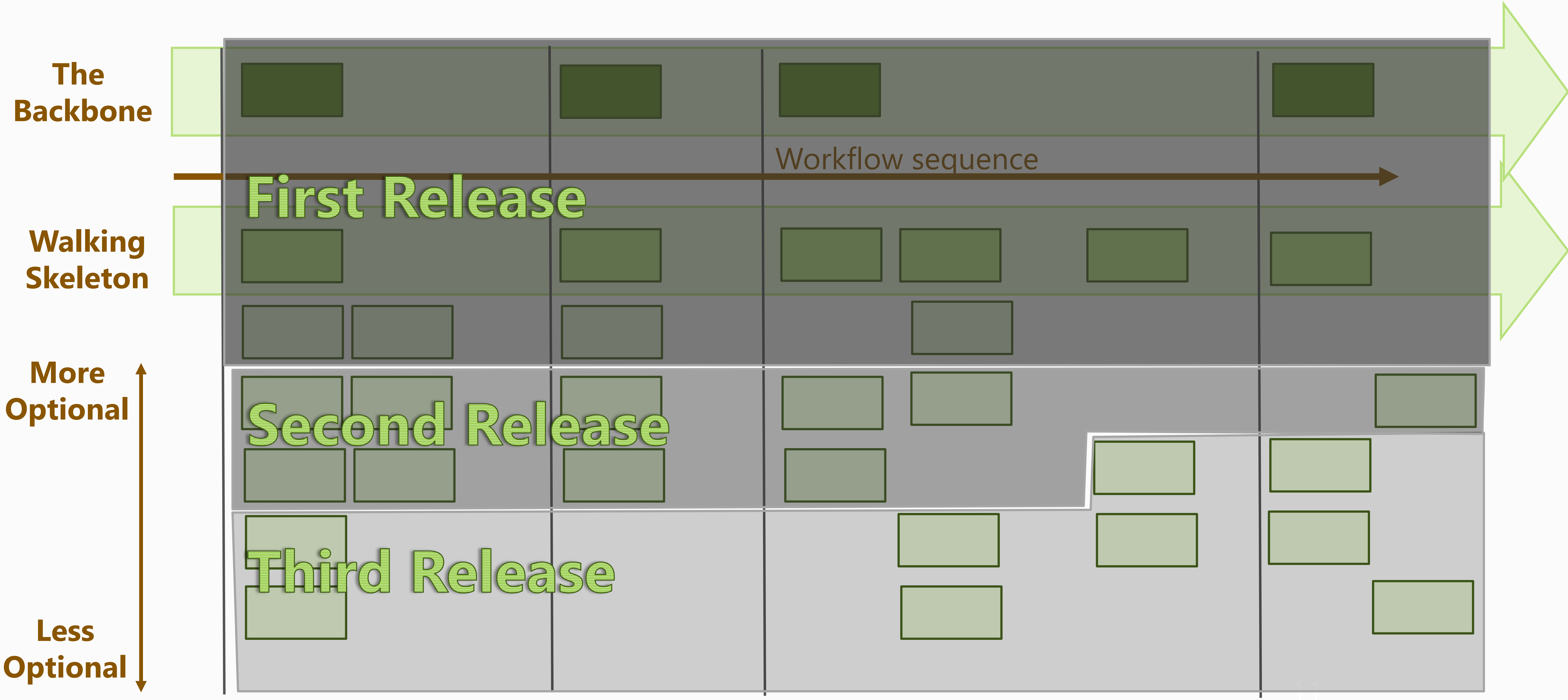
Small Releases

- Story Map (developed by Jeff Patton)
- Helps select and group features for a release
  - Backbone – essential functionality
  - Walking skeleton – smallest system that could possible work
  - Optional features

Small Releases



Small Releases



Metaphor

A good metaphor is a powerful aid in unifying the technical and business teams

The team evolves its own form of "*tribal language*", used to describe user stories and development

Metaphor

“. . . I still haven't got the hang of this metaphor thing. I saw it work, and work well, on the C3 project, but it doesn't mean I have any idea how to do it, let alone how to explain how to do it”

Martin Fowler

Metaphor

'Metaphor' seems to be one of the least understood precepts of XP although its supposed to be (one of) the most important.

Metaphor

Metaphor is something you start using when your mother asks what you are working on and you try to explain her the details.

How you find it is very project-specific.

Use your common sense or find the guy on your team who is good at explaining technical things to customers in a way that is easy to understand.

Simple Design

XP definition for *simplicity*:

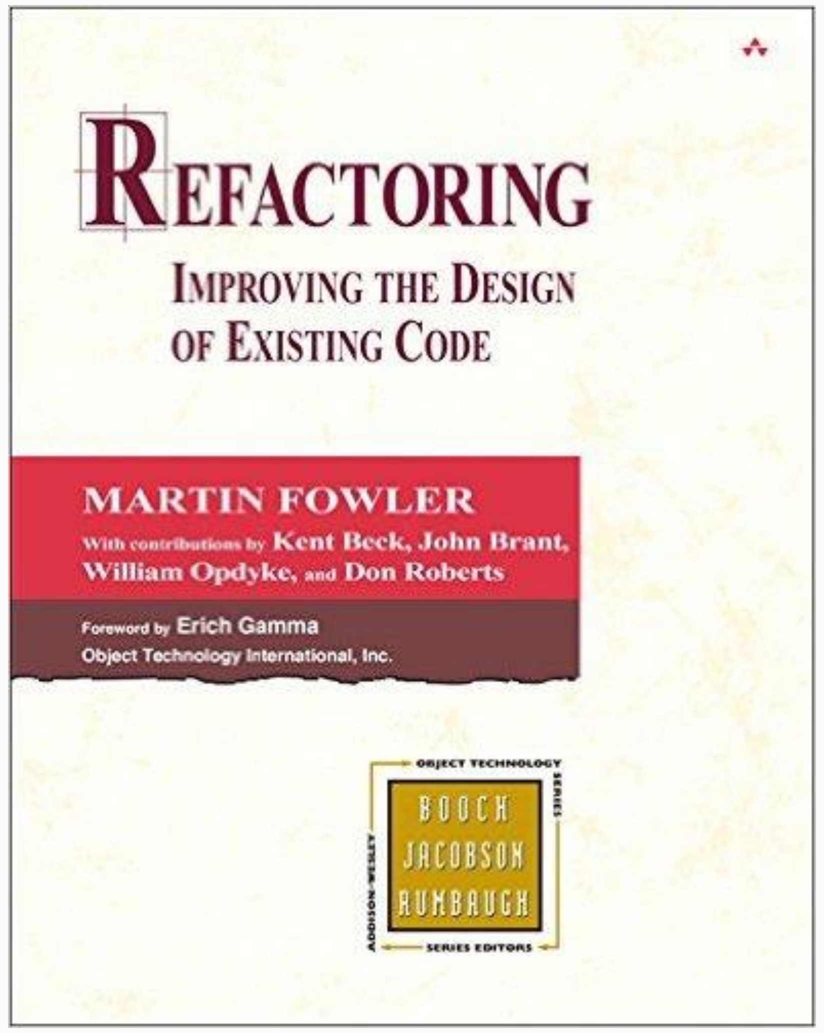
- Runs all the tests
- Contains no duplicate code
- States the programmer's intent for all the code clearly
- Contains the fewest possible classes and methods

No big design upfront!

1. Only do what you need to do now!
2. Don't add anything because you think you *might* need it!

Refactoring

Refactoring is the technique of improving code without changing functionality


REFACTORING
IMPROVING THE DESIGN OF EXISTING CODE
MARTIN FOWLER
With contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International, Inc.
OBJECT TECHNOLOGY SERIES
BOOCH JACOBSON RUMBAUGH
SERIES EDITORS
ADDISON-WESLEY

Why?

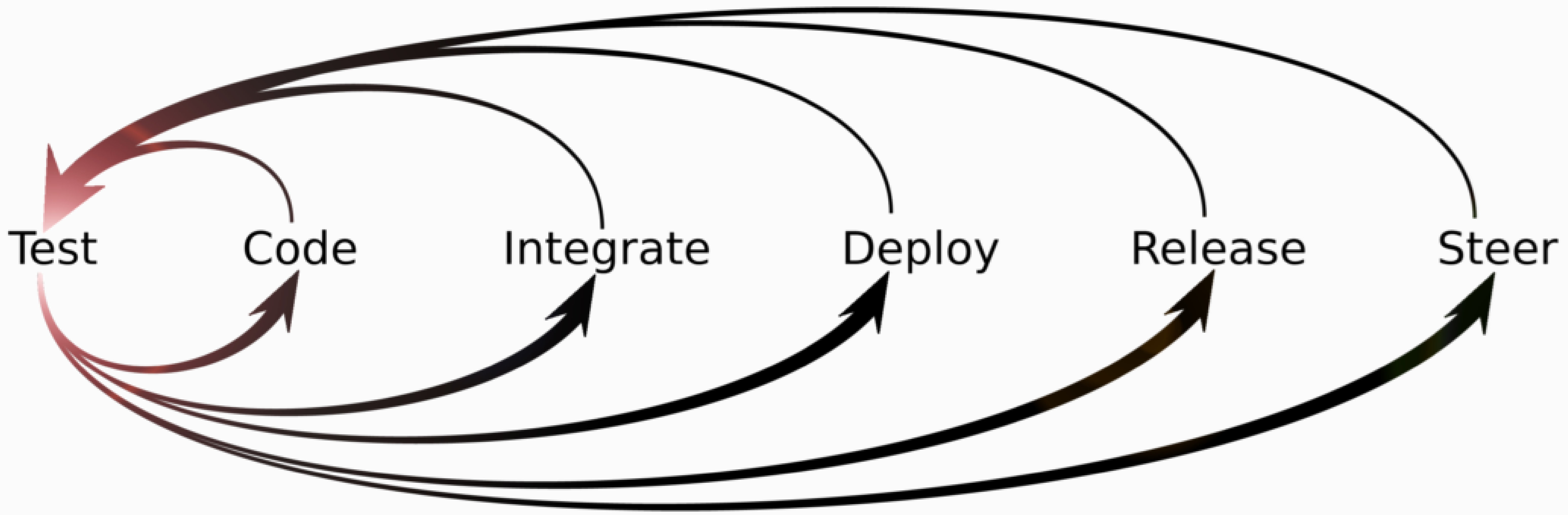
Because your code should be the simplest thing that could possibly work

Refactoring

1) Lack of tests
2) Name not from domain
3) Name not expressing intent
4) Unnecessary if
5) Unnecessary else
6) Duplication of constant
7) Method does more than one thing
8) Primitive obsession
9) Feature envy
10) Method too long (> 6 lines)
11) Too many parameters (> 3)
12) Test – not unitary
13) Test – setup too complex
14) Test – unclear Act
15) Test - more than one assert
16) Test – no assert
17) Test – too many paths

Source: Brutal Refactoring Game, Adi Bolboaca

Testing



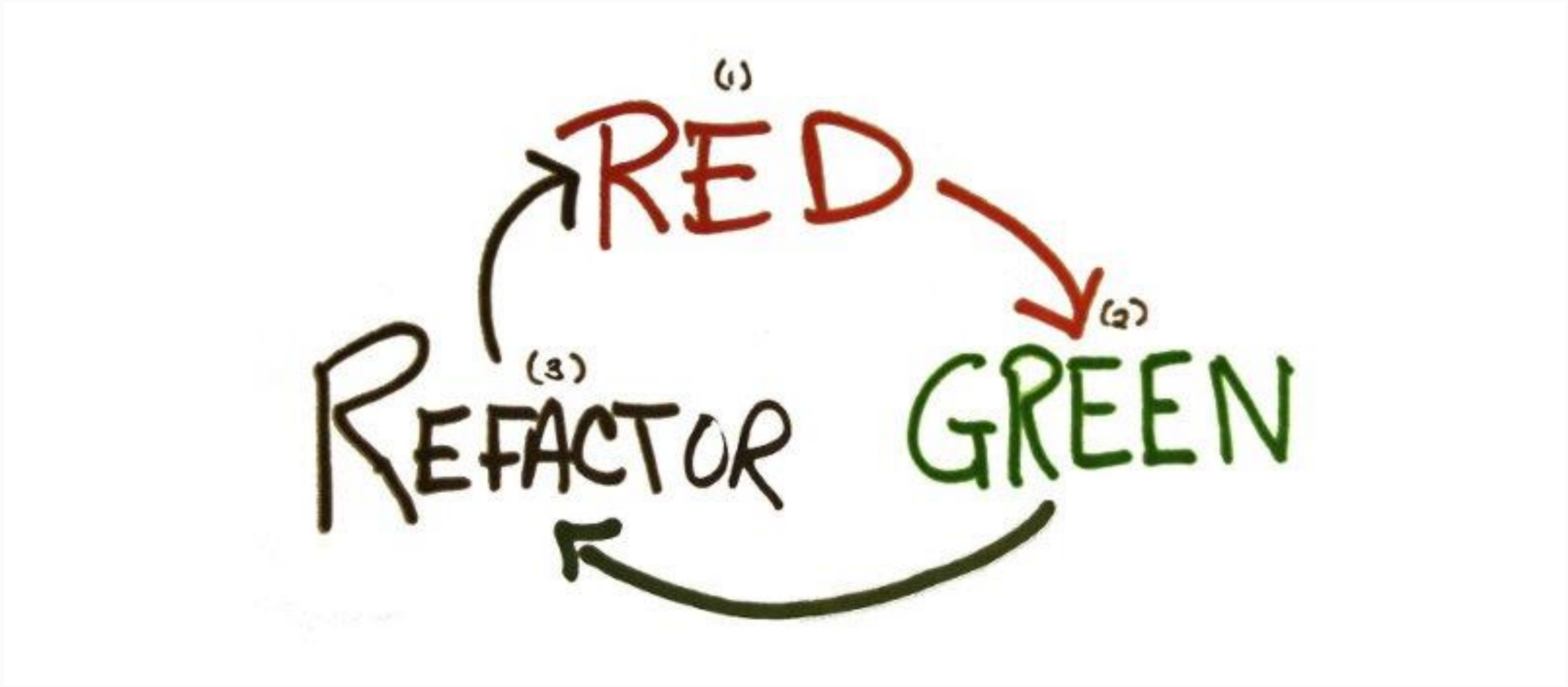
- Add a test, get it to fail, and write code to pass the test
- Remove duplication

Testing

if you can't do these, you probably shouldn't start writing any code

- Think about what you want to do.
- Think about how to test it.
- Write a small test. Think about the desired API.
- Write just enough code to fail the test.
- Run and watch the test fail. Now you know that your test is going to be executed.
- Write just enough code to pass the test (and pass all your previous tests).
- Run and watch all of the tests pass. If it doesn't pass, you did something wrong, fix it now since it's got to be something you just wrote.
- Refactor the code
- Run the tests again
- Repeat the steps above until you can't find any more tests that drive writing new code.

Testing



Collective Ownership

Any person can change
the application code
at anytime

The catch: If you own all the code, you are responsible for all the code as well

Continuous Integration

The longer the time period between integrations, the more conflicts you'll see, and the effort to integrate will increase.

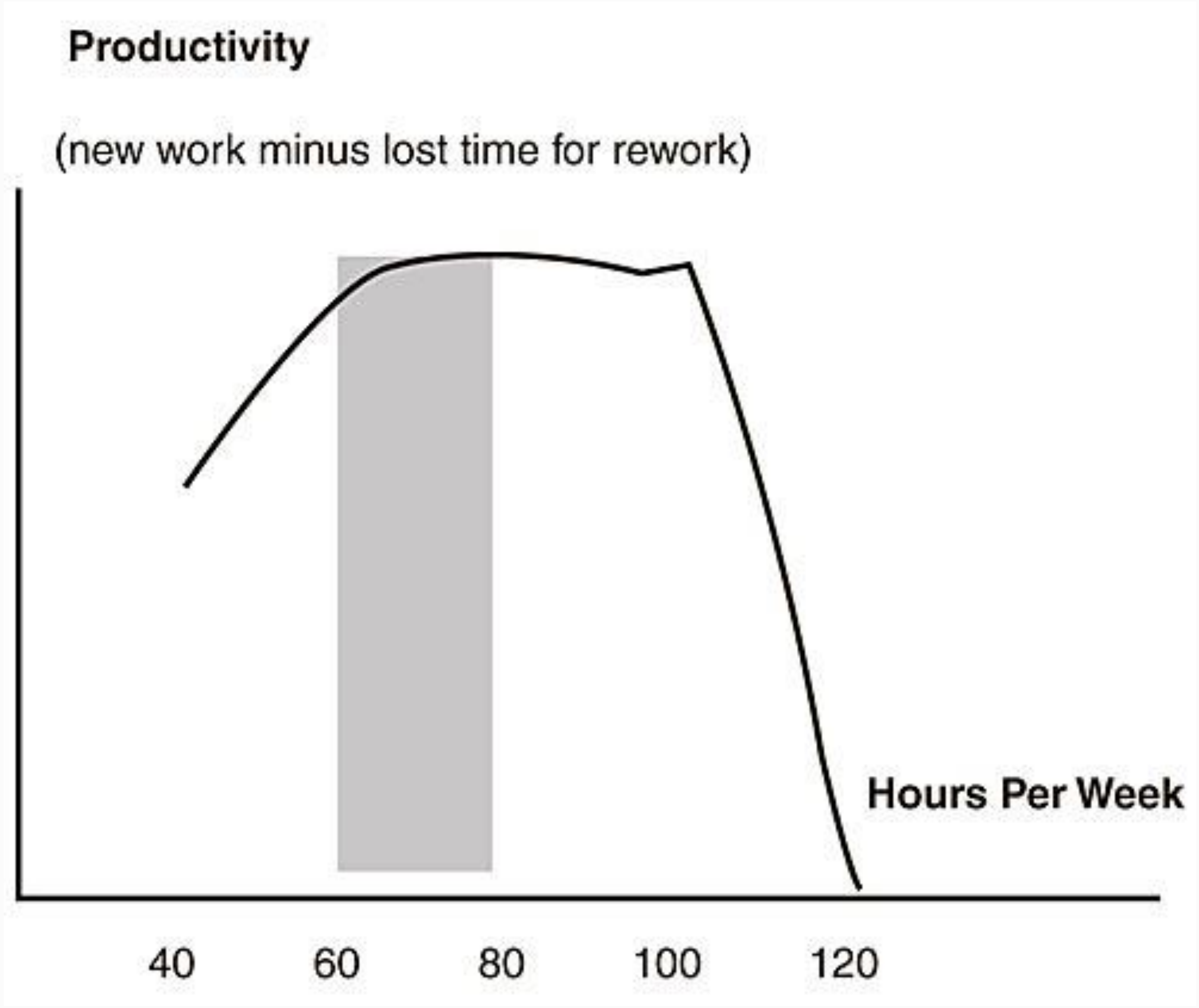
The process: developers work on tasks until complete, integrate them, run tests, fix problems

40-Hour Work Week

You cannot maintain quality with overtime-heavy teams!

Each country or culture has differing acceptance of reasonable working hours

40-Hour Work Week



On-Site Customer

The customer must be on the project full-time for the duration and be located on-site with the team

The *customer* could include users, business experts, and any other customer-side resource

Coding Standards

Mandatory — Those standards to be adhered to by all team members.

Guidelines — Those considered best or good practice and often describe the general approach toward development.

Recommendations — These rules are considered good practice and should be used at all times unless there are exceptional circumstances where valid justification can be given.

There are two types of people:

```
If (Condition)
{
          Statements
           /*
           …
           */
}
```

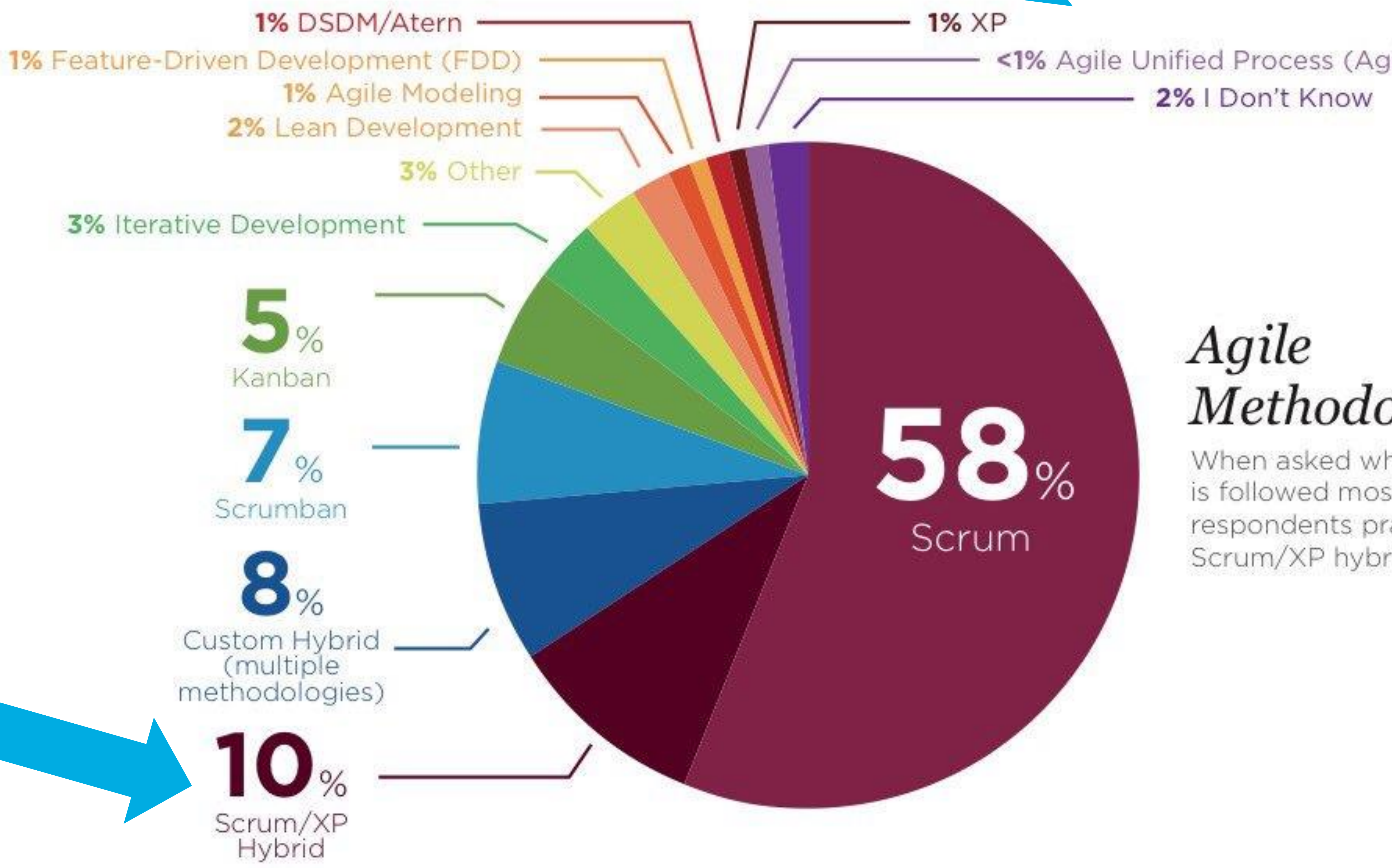
```
If (Condition){

      Statements
       /*
       …
       */
}
```

Types of coding standards:

- *Formatting*
- *Code structure*
- *Naming conventions*
- *Error handling*
- *Comments*

AGILE METHODS AND PRACTICES



Agile Methodologies Used

When asked what agile methodology is followed most closely, nearly 70% of respondents practice Scrum (58%) or Scrum/XP hybrid (10%).

Why XP is not popular?

- *It is software engineering centric*
- *It requires high investment*
  - *Rockstar developers*
  - *Trainings*
  - *Infrastructure (some automation solutions is*
  - *Culture*
- *It is irrational (to business people)*
  - *Unit tests, Test First Development, Story Points, Pair Programming*
- *It is too difficult*
  - *Test First Development, Refactoring, Simple & Emergent Design*