XP & jUnit
TDD & BDD

The Values of Extreme Programming

Extreme Programming (XP) is based on values. The rules we just examined are the natural extension and consequence of maximizing our values. XP isn't really a set of rules but rather a way to work in harmony with your personal and corporate values. Start with XP's values listed here then add your own by reflecting them in the changes you make to the rules.

Simplicity: We will do what is needed and asked for, but no more. This will maximize the value created for the investment made to date. We will take small simple steps to our goal and mitigate failures as they happen. We will create something we are proud of and maintain it long term for reasonable costs.

Communication: Everyone is part of the team and we communicate face to face daily. We will work together on everything from requirements to code. We will create the best solution to our problem that we can together.

Feedback: We will take every iteration commitment seriously by delivering working software. We demonstrate our software early and often then listen carefully and make any changes needed. We will talk about the project and adapt our process to it, not the other way around.

Respect: Everyone gives and feels the respect they deserve as a valued team member. Everyone contributes value even if it's simply enthusiasm. Developers respect the expertise of the customers and vice versa. Management respects our right to accept responsibility and receive authority over our own work.

Courage: We will tell the truth about progress and estimates. We don't document excuses for failure because we plan to succeed. We don't fear anything because no one ever works alone. We will adapt to changes when ever they happen.

What lessons have we learned about implementing XP so far.

ExtremeProgramming.org home | XP Rules | XP Map | Lessons Learned | About the Author

The Rules of Extreme Programming

Planning

- User stories are written.
- Release planning creates the release schedule.
- Make frequent small releases.
- The project is divided into iterations.
- Iteration planning starts each iteration.

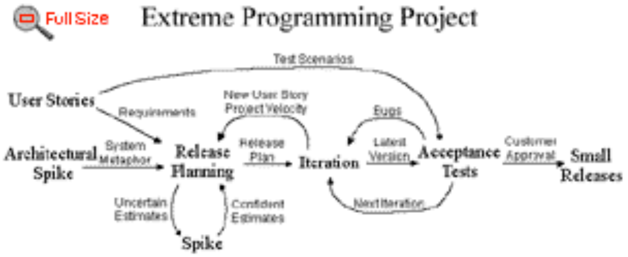
Managing

- Give the team a dedicated open work space.
- Set a sustainable pace.
- A stand up meeting starts each day.
- The Project Velocity is measured.
- Move people around.
- Fix XP when it breaks.

Designing

- Simplicity.
- Choose a system metaphor.
- Use CRC cards for design sessions.
- Create spike solutions to reduce risk.
- No functionality is added early.
- Refactor whenever and wherever possible.

Full Size



Coding

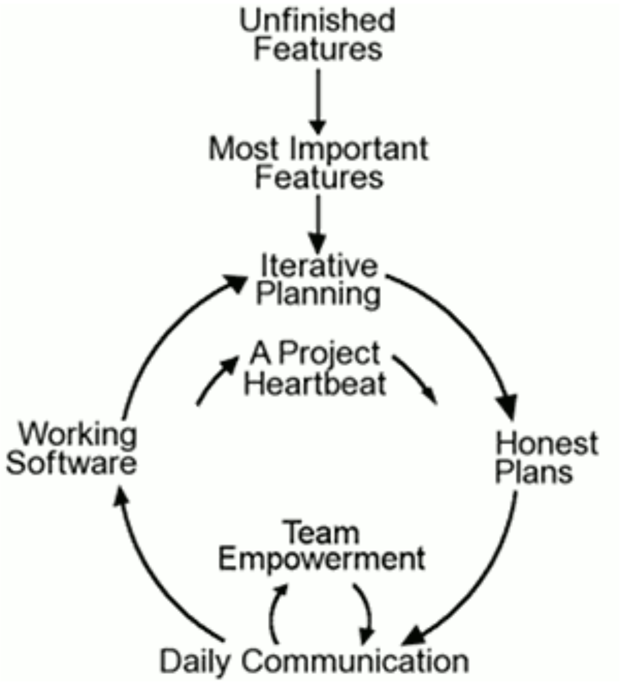
- The customer is always available.
- Code must be written to agreed standards.
- Code the unit test first.
- All production code is pair programmed.
- Only one pair integrates code at a time.
- Integrate often.
- Set up a dedicated integration computer.
- Use collective ownership.

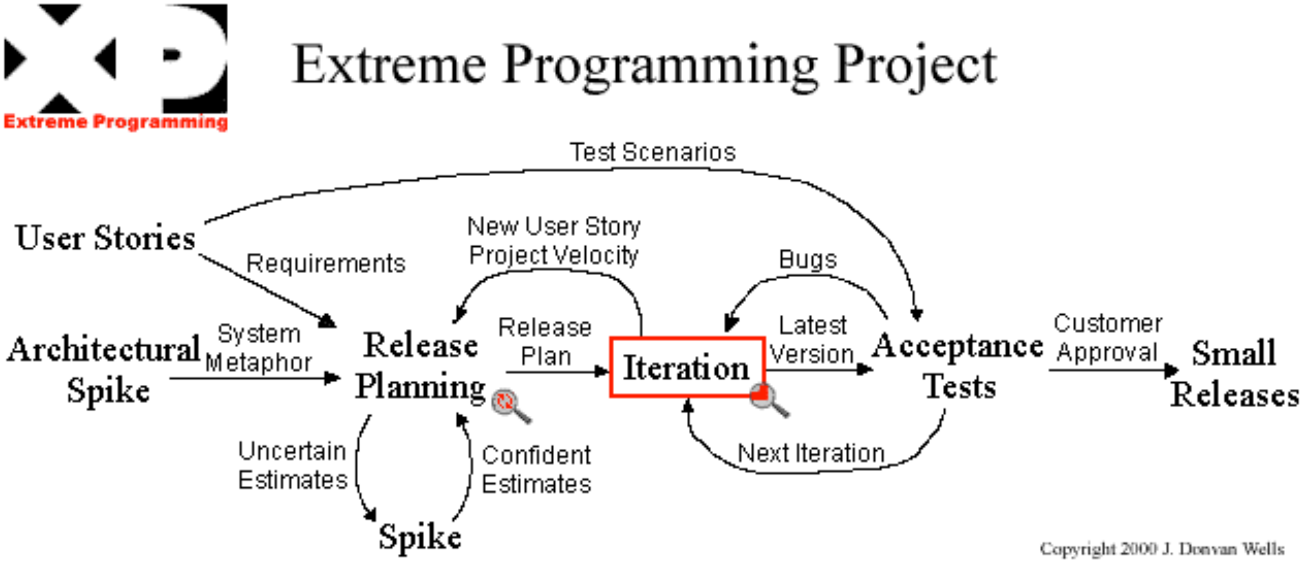
Testing

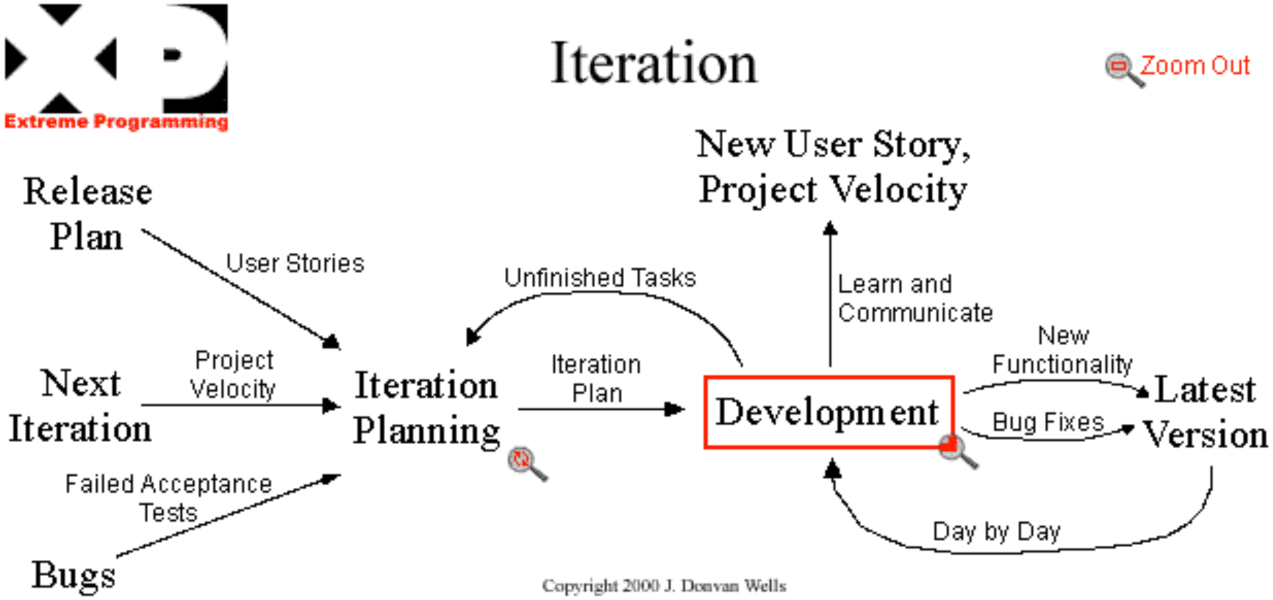
- All code must have unit tests.
- All code must pass all unit tests before it can be released.
- When a bug is found tests are created.
- Acceptance tests are run often and the score is published.

Let's review the values of Extreme Programming (XP) next.

ExtremeProgramming.org home | XP Map | XP Values | Test framework | About the Author


Unfinished Features
Most Important Features
Iterative Planning
A Project Heartbeat
Honest Plans
Working Software
Team Empowerment
Daily Communication

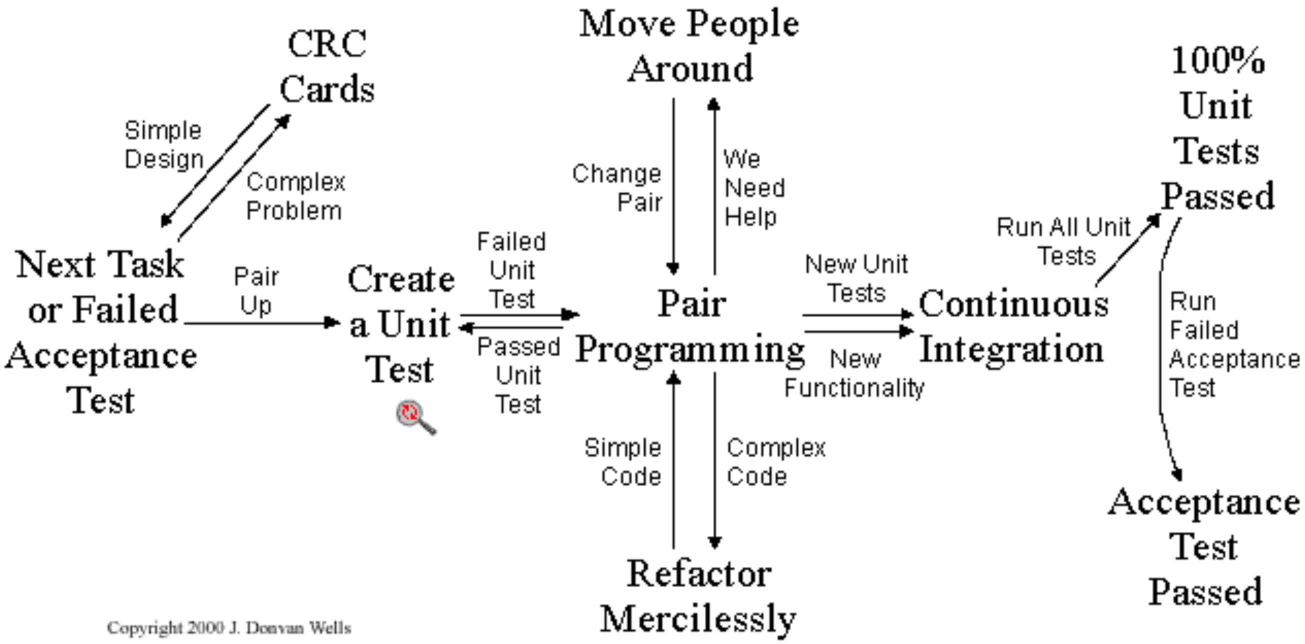

Extreme Programming
Extreme Programming Project
Test Scenarios
User Stories
Requirements
New User Story
Project Velocity
Bugs
Architectural Spike
System Metaphor
Release Planning
Release Plan
Iteration
Latest Version
Acceptance Tests
Customer Approval
Small Releases
Uncertain Estimates
Confident Estimates
Next Iteration
Spike
Copyright 2000 J. Donvan Wells


Extreme Programming
Iteration
Zoom Out
New User Story, Project Velocity
Release Plan
User Stories
Unfinished Tasks
Learn and Communicate
Next Iteration
Project Velocity
Iteration Planning
Iteration Plan
Development
New Functionality
Bug Fixes
Latest Version
Failed Acceptance Tests
Bugs
Day by Day
Copyright 2000 J. Donvan Wells

Extreme Programming

Collective Code Ownership

Zoom Out



CRC Cards

Simple Design

Complex Problem

Next Task or Failed Acceptance Test

Pair Up

Create a Unit Test

Failed Unit Test

Passed Unit Test

Move People Around

Change Pair

We Need Help

Pair Programming

Simple Code

Complex Code

Refactor Mercilessly

New Unit Tests

New Functionality

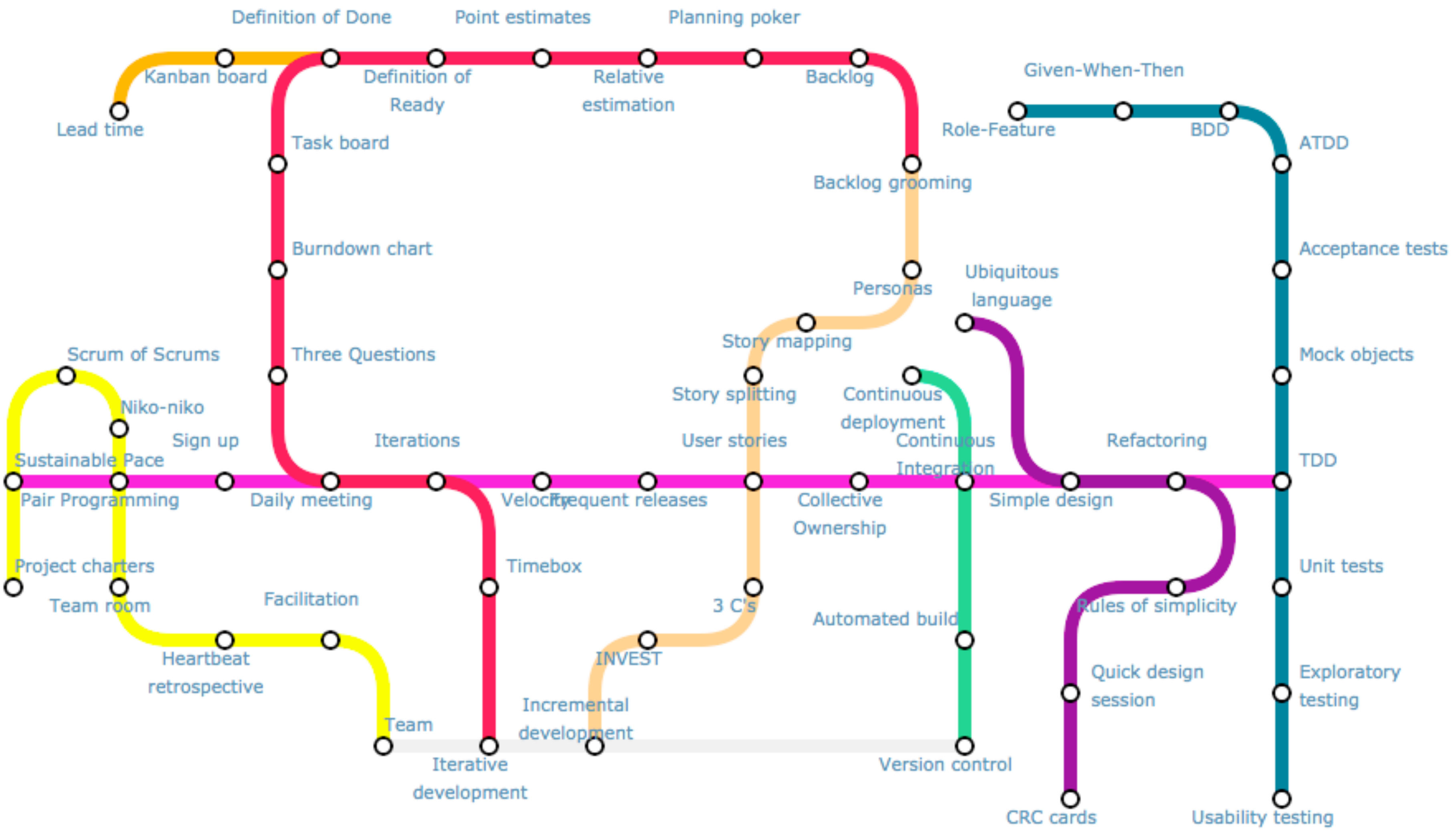
Continuous Integration

Run All Unit Tests

100% Unit Tests Passed

Run Failed Acceptance Test

Acceptance Test Passed

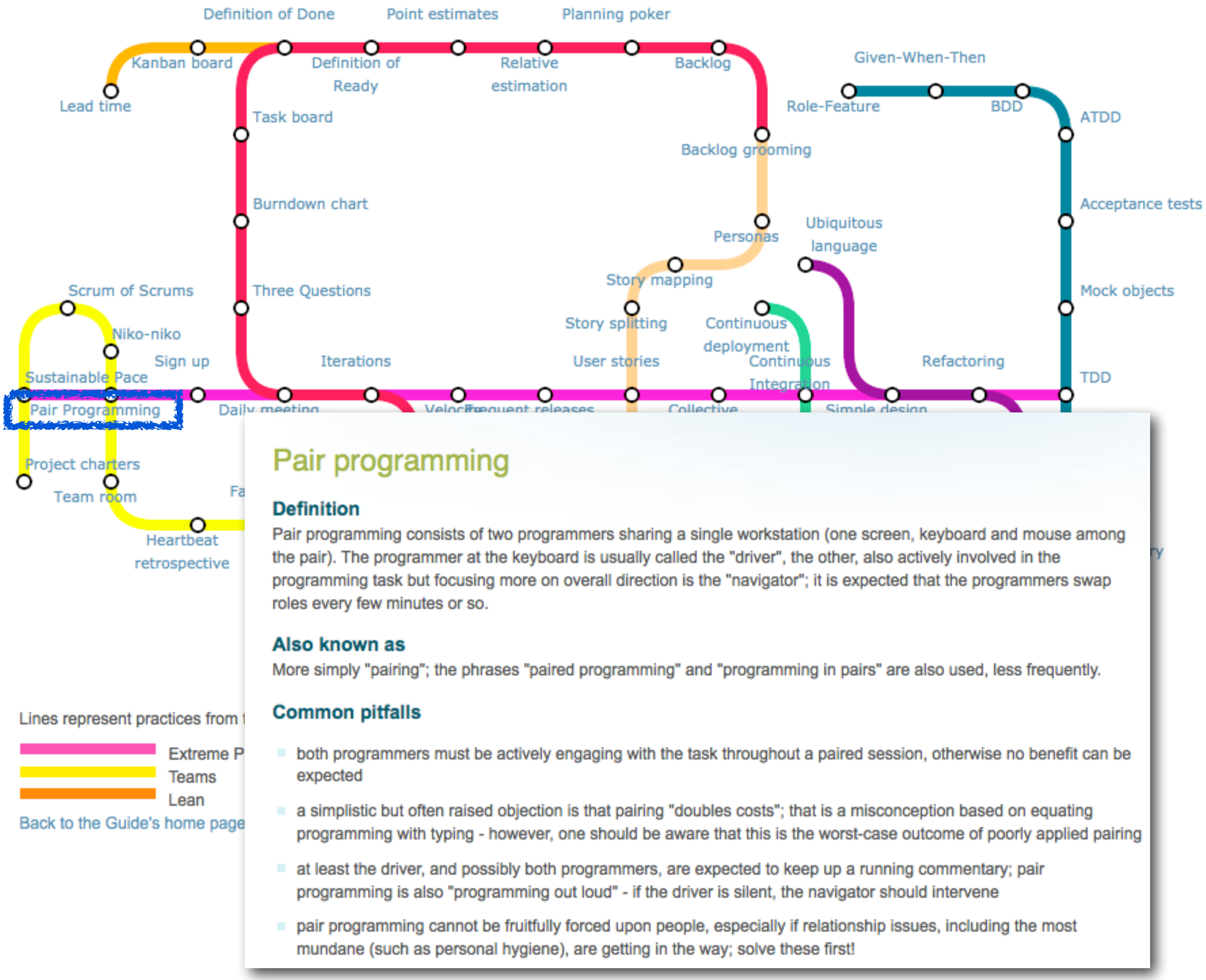


Lines represent practices from the various Agile "tribes" or areas of concern:

🟪	Extreme Programming	🟥	Scrum	🟪	Design
🟨	Teams	🟧	Product management	🟦	Testing
🟧	Lean	🟩	Devops	⬜	Fundamentals

Back to the Guide's home page

http://guide.agilealliance.org/subway.html



Pair programming

Definition

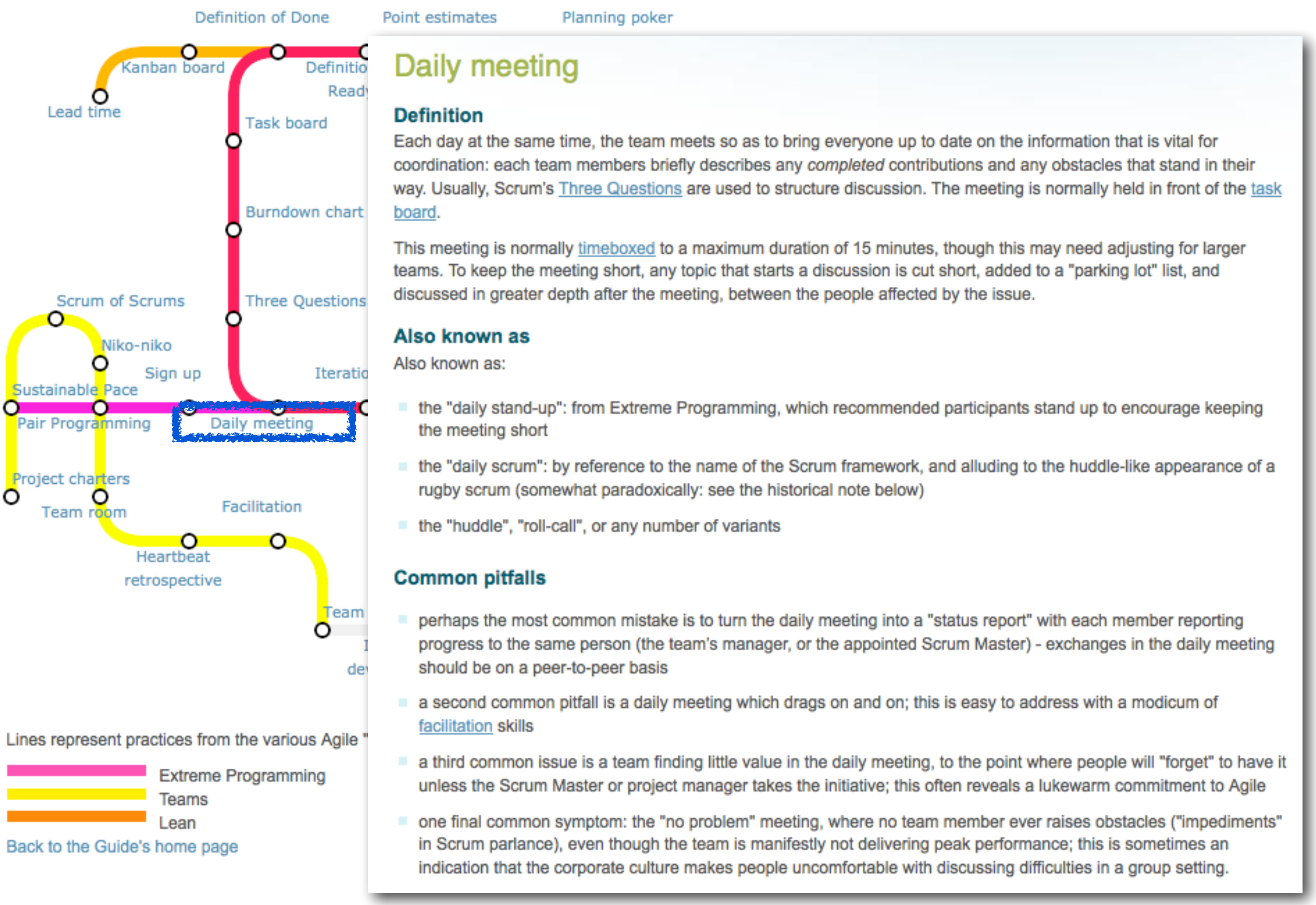
Pair programming consists of two programmers sharing a single workstation (one screen, keyboard and mouse among the pair). The programmer at the keyboard is usually called the "driver", the other, also actively involved in the programming task but focusing more on overall direction is the "navigator"; it is expected that the programmers swap roles every few minutes or so.

Also known as

More simply "pairing"; the phrases "paired programming" and "programming in pairs" are also used, less frequently.

Common pitfalls

- both programmers must be actively engaging with the task throughout a paired session, otherwise no benefit can be expected
- a simplistic but often raised objection is that pairing "doubles costs"; that is a misconception based on equating programming with typing - however, one should be aware that this is the worst-case outcome of poorly applied pairing
- at least the driver, and possibly both programmers, are expected to keep up a running commentary; pair programming is also "programming out loud" - if the driver is silent, the navigator should intervene
- pair programming cannot be fruitfully forced upon people, especially if relationship issues, including the most mundane (such as personal hygiene), are getting in the way; solve these first!



Lines represent practices from the various Agile "

- 🟪 Extreme Programming
- 🟨 Teams
- 🟧 Lean

Back to the Guide's home page

Daily meeting

Definition

Each day at the same time, the team meets so as to bring everyone up to date on the information that is vital for coordination: each team members briefly describes any *completed* contributions and any obstacles that stand in their way. Usually, Scrum's Three Questions are used to structure discussion. The meeting is normally held in front of the task board.

This meeting is normally timeboxed to a maximum duration of 15 minutes, though this may need adjusting for larger teams. To keep the meeting short, any topic that starts a discussion is cut short, added to a "parking lot" list, and discussed in greater depth after the meeting, between the people affected by the issue.

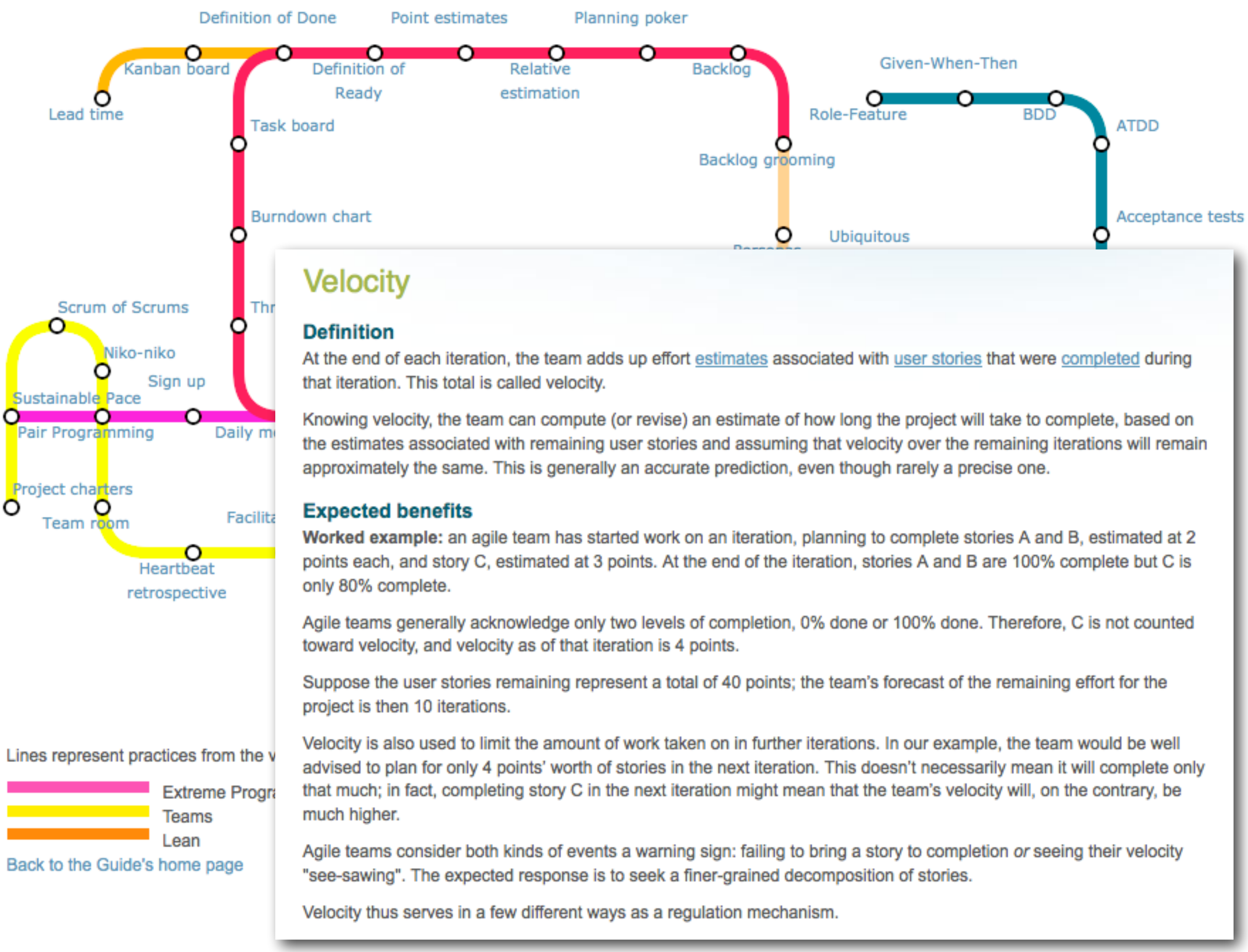
Also known as

Also known as:

- the "daily stand-up": from Extreme Programming, which recommended participants stand up to encourage keeping the meeting short
- the "daily scrum": by reference to the name of the Scrum framework, and alluding to the huddle-like appearance of a rugby scrum (somewhat paradoxically: see the historical note below)
- the "huddle", "roll-call", or any number of variants

Common pitfalls

- perhaps the most common mistake is to turn the daily meeting into a "status report" with each member reporting progress to the same person (the team's manager, or the appointed Scrum Master) - exchanges in the daily meeting should be on a peer-to-peer basis
- a second common pitfall is a daily meeting which drags on and on; this is easy to address with a modicum of facilitation skills
- a third common issue is a team finding little value in the daily meeting, to the point where people will "forget" to have it unless the Scrum Master or project manager takes the initiative; this often reveals a lukewarm commitment to Agile
- one final common symptom: the "no problem" meeting, where no team member ever raises obstacles ("impediments" in Scrum parlance), even though the team is manifestly not delivering peak performance; this is sometimes an indication that the corporate culture makes people uncomfortable with discussing difficulties in a group setting.



Lines represent practices from the v

🟪 Extreme Progr

🟨 Teams

🟧 Lean

Back to the Guide's home page

Velocity

Definition

At the end of each iteration, the team adds up effort estimates associated with user stories that were completed during that iteration. This total is called velocity.

Knowing velocity, the team can compute (or revise) an estimate of how long the project will take to complete, based on the estimates associated with remaining user stories and assuming that velocity over the remaining iterations will remain approximately the same. This is generally an accurate prediction, even though rarely a precise one.

Expected benefits

Worked example: an agile team has started work on an iteration, planning to complete stories A and B, estimated at 2 points each, and story C, estimated at 3 points. At the end of the iteration, stories A and B are 100% complete but C is only 80% complete.

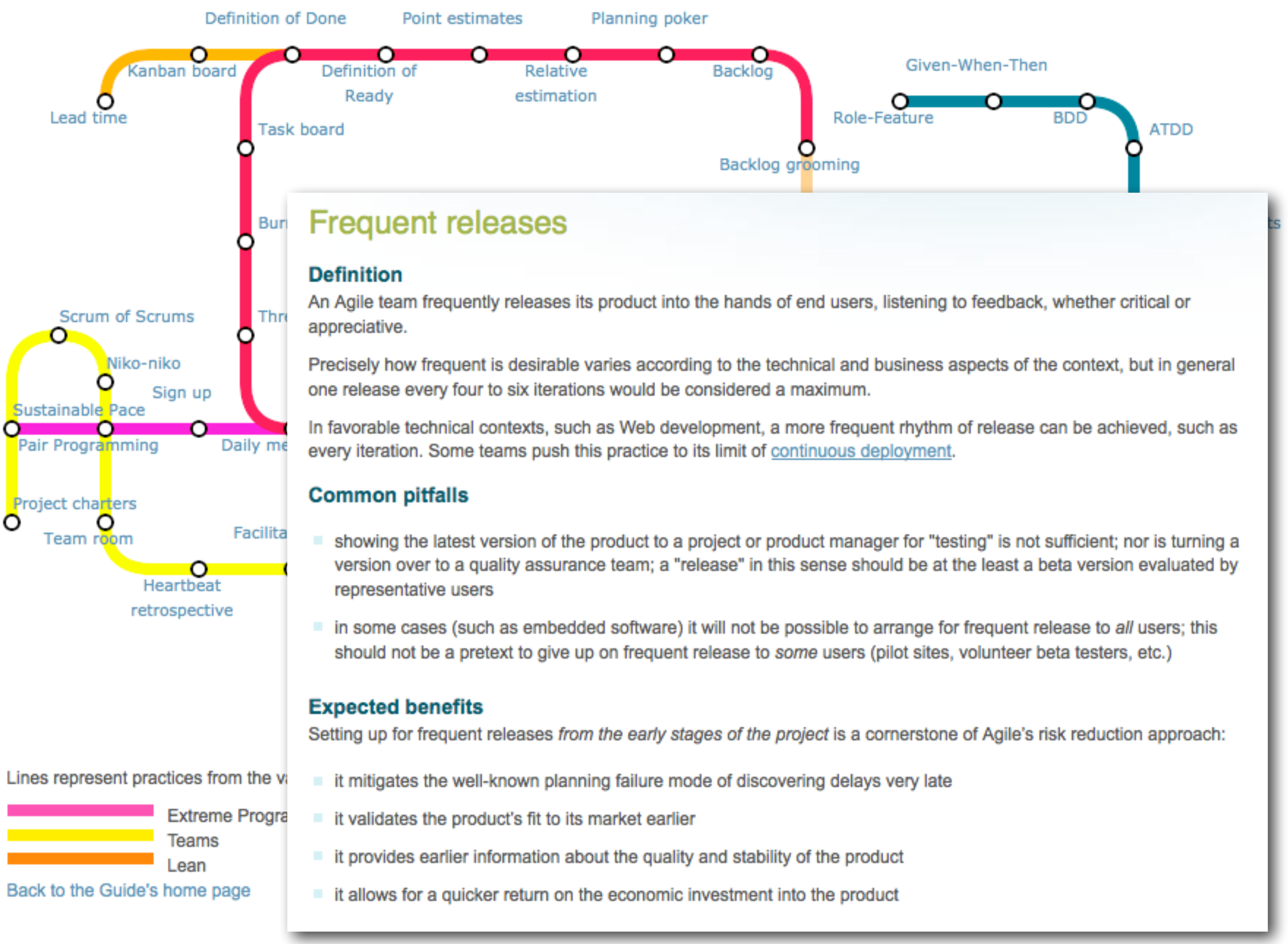
Agile teams generally acknowledge only two levels of completion, 0% done or 100% done. Therefore, C is not counted toward velocity, and velocity as of that iteration is 4 points.

Suppose the user stories remaining represent a total of 40 points; the team's forecast of the remaining effort for the project is then 10 iterations.

Velocity is also used to limit the amount of work taken on in further iterations. In our example, the team would be well advised to plan for only 4 points' worth of stories in the next iteration. This doesn't necessarily mean it will complete only that much; in fact, completing story C in the next iteration might mean that the team's velocity will, on the contrary, be much higher.

Agile teams consider both kinds of events a warning sign: failing to bring a story to completion *or* seeing their velocity "see-sawing". The expected response is to seek a finer-grained decomposition of stories.

Velocity thus serves in a few different ways as a regulation mechanism.



Frequent releases

Definition

An Agile team frequently releases its product into the hands of end users, listening to feedback, whether critical or appreciative.

Precisely how frequent is desirable varies according to the technical and business aspects of the context, but in general one release every four to six iterations would be considered a maximum.

In favorable technical contexts, such as Web development, a more frequent rhythm of release can be achieved, such as every iteration. Some teams push this practice to its limit of continuous deployment.

Common pitfalls

- showing the latest version of the product to a project or product manager for "testing" is not sufficient; nor is turning a version over to a quality assurance team; a "release" in this sense should be at the least a beta version evaluated by representative users
- in some cases (such as embedded software) it will not be possible to arrange for frequent release to *all* users; this should not be a pretext to give up on frequent release to *some* users (pilot sites, volunteer beta testers, etc.)

Expected benefits

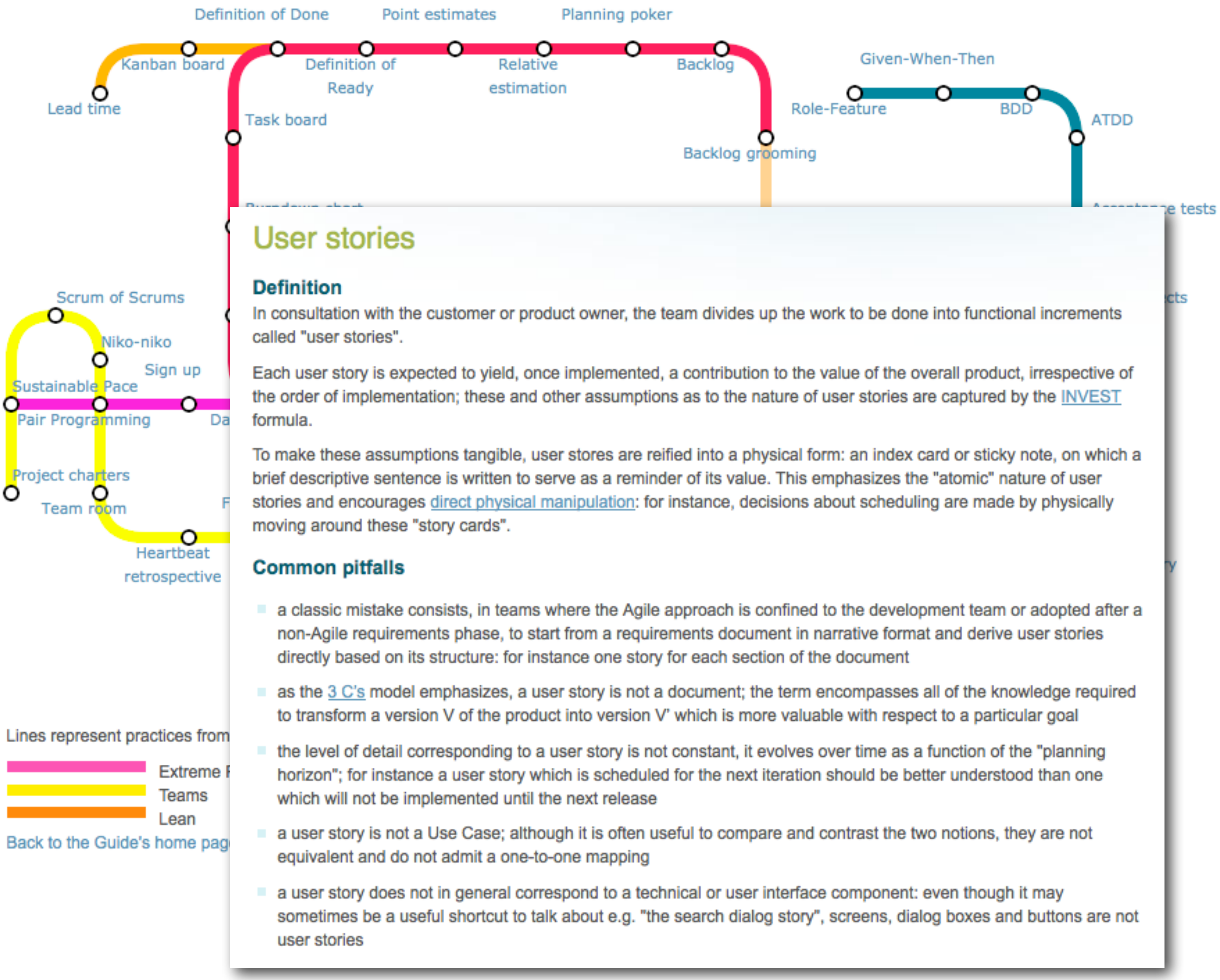
Setting up for frequent releases *from the early stages of the project* is a cornerstone of Agile's risk reduction approach:

- it mitigates the well-known planning failure mode of discovering delays very late
- it validates the product's fit to its market earlier
- it provides earlier information about the quality and stability of the product
- it allows for a quicker return on the economic investment into the product

Lines represent practices from the v

🟪 Extreme Progra
🟨 Teams
🟧 Lean

Back to the Guide's home page



User stories

Definition

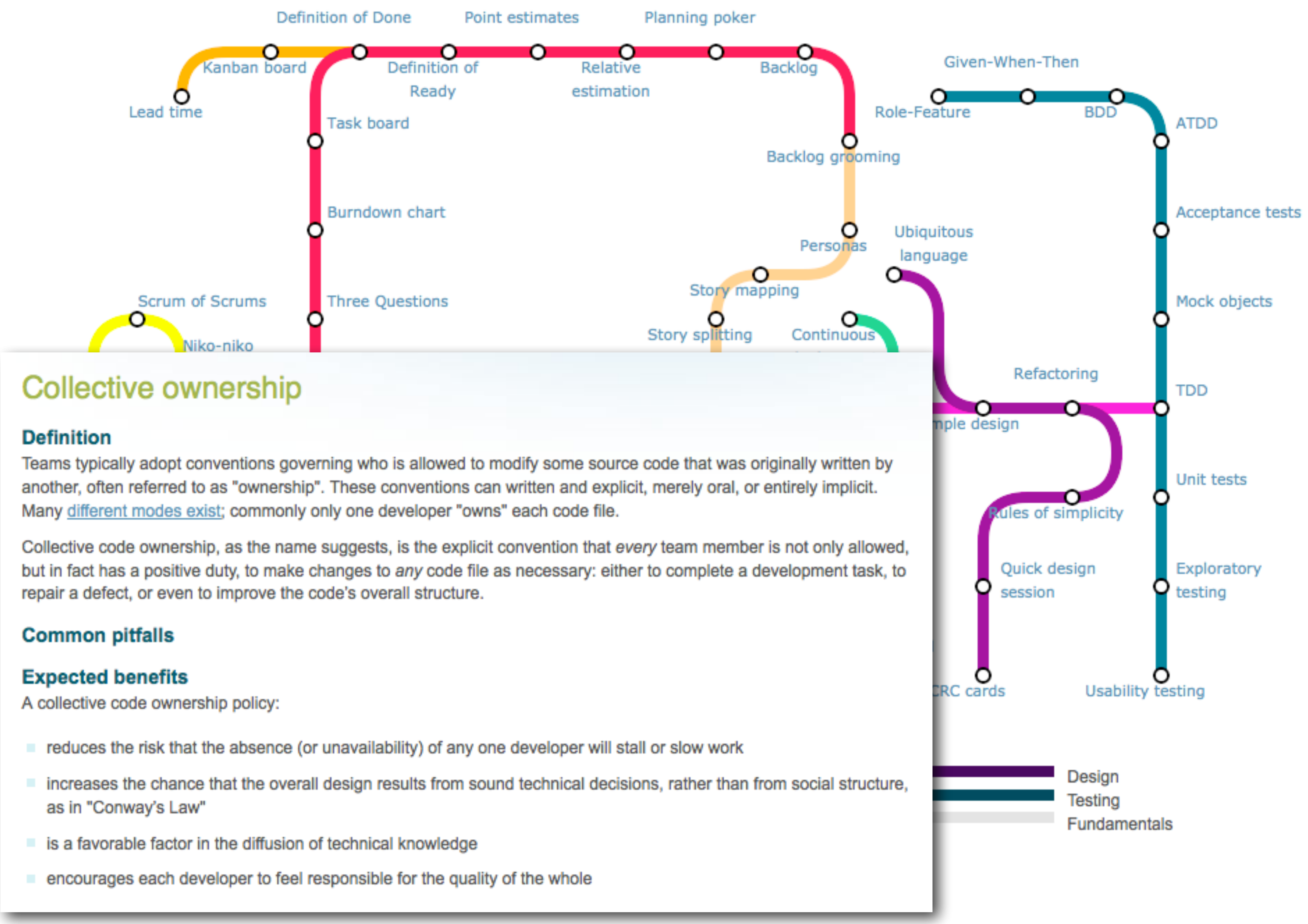
In consultation with the customer or product owner, the team divides up the work to be done into functional increments called "user stories".

Each user story is expected to yield, once implemented, a contribution to the value of the overall product, irrespective of the order of implementation; these and other assumptions as to the nature of user stories are captured by the INVEST formula.

To make these assumptions tangible, user stores are reified into a physical form: an index card or sticky note, on which a brief descriptive sentence is written to serve as a reminder of its value. This emphasizes the "atomic" nature of user stories and encourages direct physical manipulation: for instance, decisions about scheduling are made by physically moving around these "story cards".

Common pitfalls

- a classic mistake consists, in teams where the Agile approach is confined to the development team or adopted after a non-Agile requirements phase, to start from a requirements document in narrative format and derive user stories directly based on its structure: for instance one story for each section of the document
- as the 3 C's model emphasizes, a user story is not a document; the term encompasses all of the knowledge required to transform a version V of the product into version V' which is more valuable with respect to a particular goal
- the level of detail corresponding to a user story is not constant, it evolves over time as a function of the "planning horizon"; for instance a user story which is scheduled for the next iteration should be better understood than one which will not be implemented until the next release
- a user story is not a Use Case; although it is often useful to compare and contrast the two notions, they are not equivalent and do not admit a one-to-one mapping
- a user story does not in general correspond to a technical or user interface component: even though it may sometimes be a useful shortcut to talk about e.g. "the search dialog story", screens, dialog boxes and buttons are not user stories



Collective ownership

Definition

Teams typically adopt conventions governing who is allowed to modify some source code that was originally written by another, often referred to as "ownership". These conventions can written and explicit, merely oral, or entirely implicit. Many different modes exist; commonly only one developer "owns" each code file.

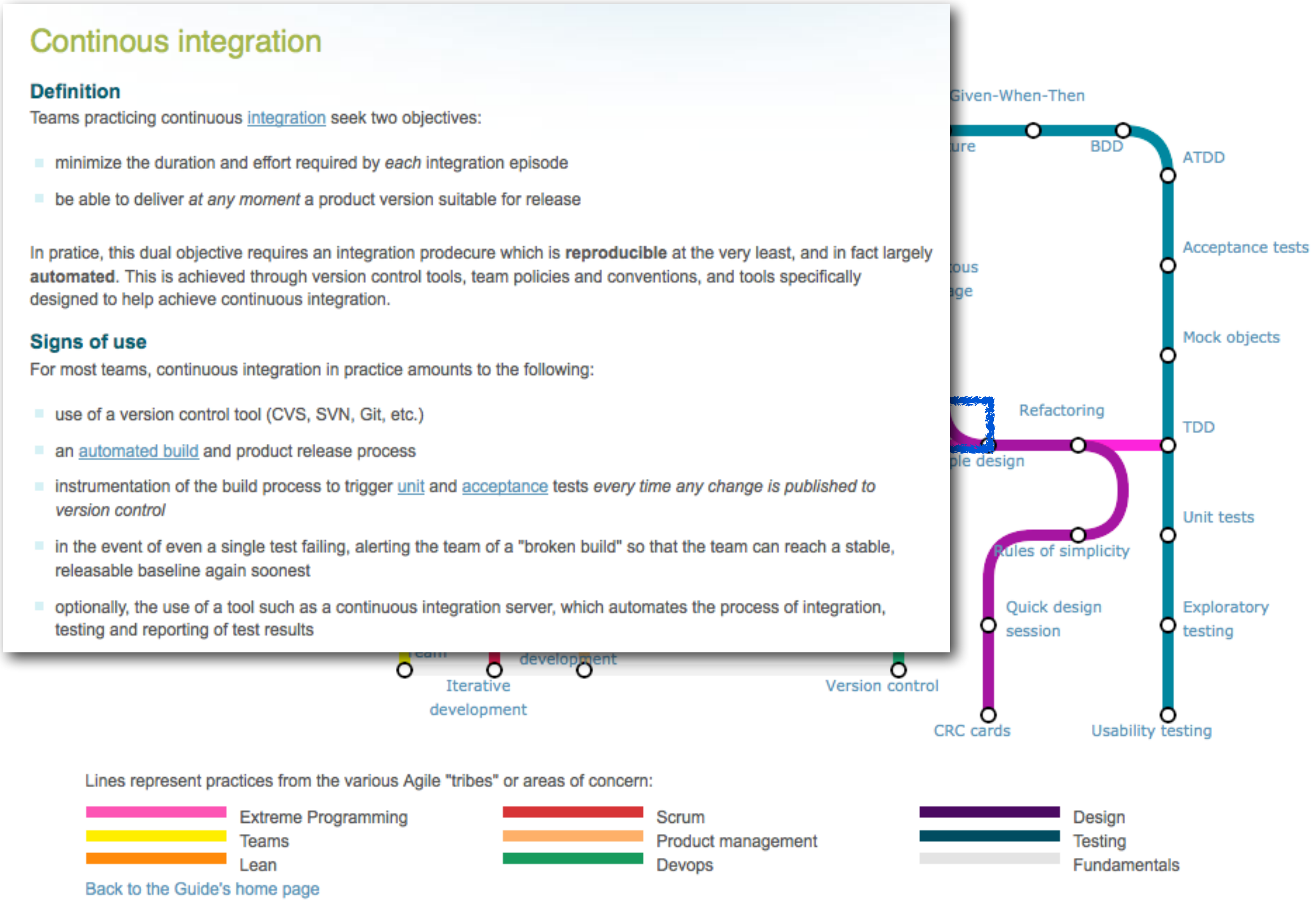
Collective code ownership, as the name suggests, is the explicit convention that *every* team member is not only allowed, but in fact has a positive duty, to make changes to *any* code file as necessary: either to complete a development task, to repair a defect, or even to improve the code's overall structure.

Common pitfalls

Expected benefits

A collective code ownership policy:

- reduces the risk that the absence (or unavailability) of any one developer will stall or slow work
- increases the chance that the overall design results from sound technical decisions, rather than from social structure, as in "Conway's Law"
- is a favorable factor in the diffusion of technical knowledge
- encourages each developer to feel responsible for the quality of the whole



Continous integration

Definition

Teams practicing continuous integration seek two objectives:

- minimize the duration and effort required by *each* integration episode
- be able to deliver *at any moment* a product version suitable for release

In pratice, this dual objective requires an integration prodecure which is **reproducible** at the very least, and in fact largely **automated**. This is achieved through version control tools, team policies and conventions, and tools specifically designed to help achieve continuous integration.

Signs of use

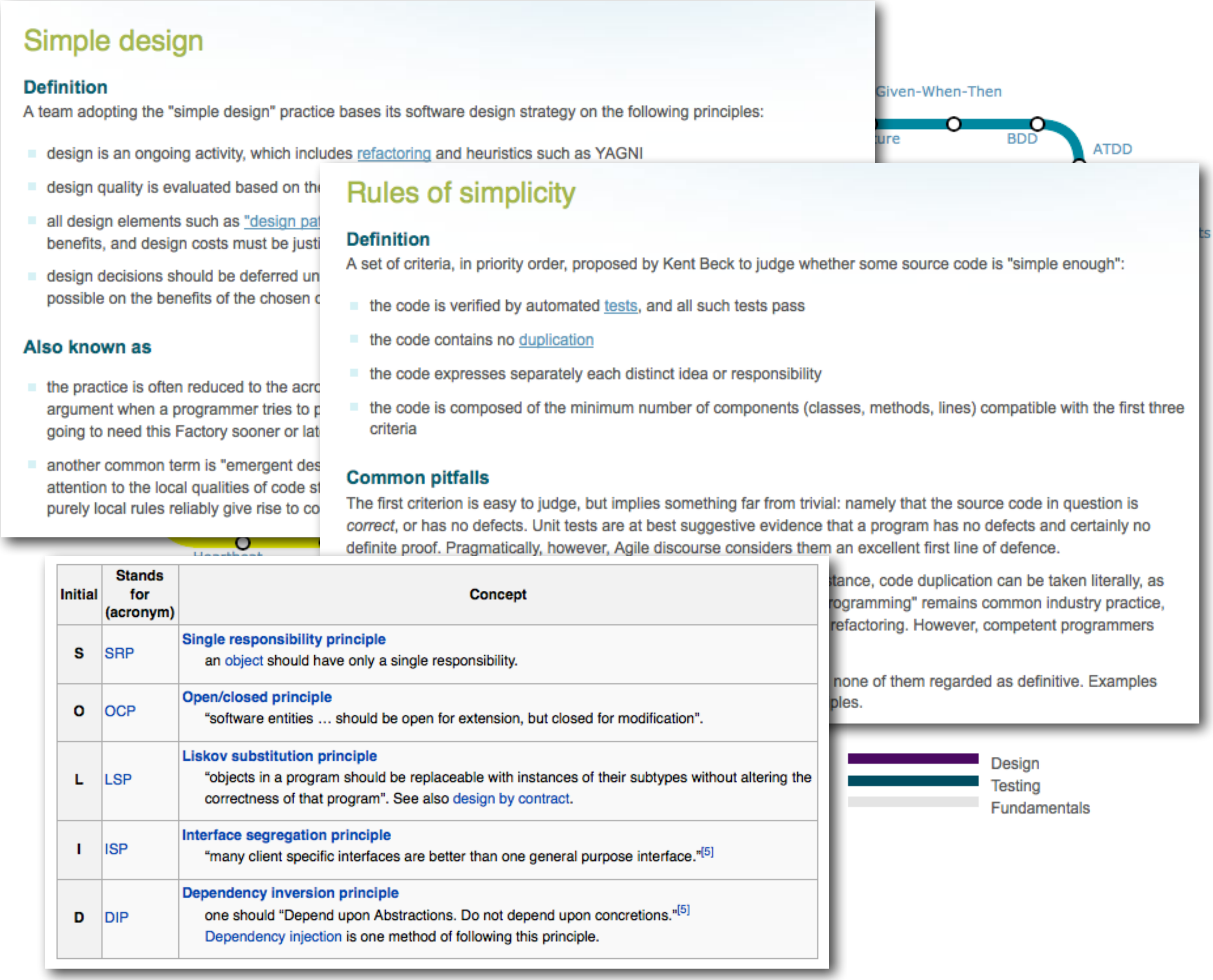
For most teams, continuous integration in practice amounts to the following:

- use of a version control tool (CVS, SVN, Git, etc.)
- an automated build and product release process
- instrumentation of the build process to trigger unit and acceptance tests *every time any change is published to version control*
- in the event of even a single test failing, alerting the team of a "broken build" so that the team can reach a stable, releasable baseline again soonest
- optionally, the use of a tool such as a continuous integration server, which automates the process of integration, testing and reporting of test results

Lines represent practices from the various Agile "tribes" or areas of concern:

🟪	Extreme Programming	🟥	Scrum	🟪	Design
🟨	Teams	🟧	Product management	🟦	Testing
🟧	Lean	🟩	Devops	⬜	Fundamentals

Back to the Guide's home page



Simple design

Definition

A team adopting the "simple design" practice bases its software design strategy on the following principles:

- design is an ongoing activity, which includes refactoring and heuristics such as YAGNI
- design quality is evaluated based on th
- all design elements such as "design pat benefits, and design costs must be justi
- design decisions should be deferred un possible on the benefits of the chosen c

Also known as

- the practice is often reduced to the acro argument when a programmer tries to p going to need this Factory sooner or lat
- another common term is "emergent des attention to the local qualities of code st purely local rules reliably give rise to co

Rules of simplicity

Definition

A set of criteria, in priority order, proposed by Kent Beck to judge whether some source code is "simple enough":

- the code is verified by automated tests, and all such tests pass
- the code contains no duplication
- the code expresses separately each distinct idea or responsibility
- the code is composed of the minimum number of components (classes, methods, lines) compatible with the first three criteria

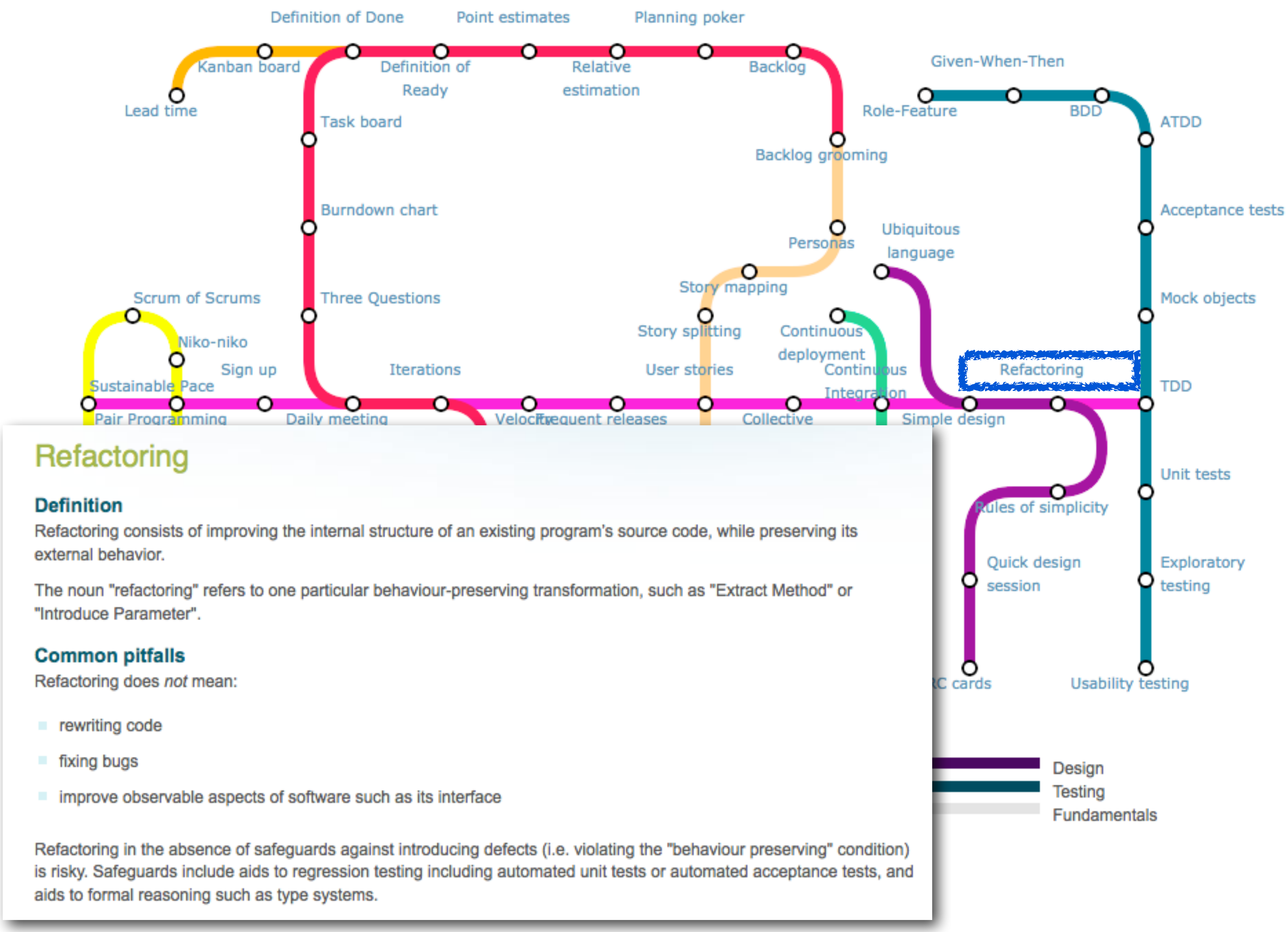
Common pitfalls

The first criterion is easy to judge, but implies something far from trivial: namely that the source code in question is *correct*, or has no defects. Unit tests are at best suggestive evidence that a program has no defects and certainly no definite proof. Pragmatically, however, Agile discourse considers them an excellent first line of defence.

tance, code duplication can be taken literally, as ogramming" remains common industry practice, refactoring. However, competent programmers

none of them regarded as definitive. Examples ples.

Initial	Stands for (acronym)	Concept
S	SRP	**Single responsibility principle** an object should have only a single responsibility.
O	OCP	**Open/closed principle** "software entities … should be open for extension, but closed for modification".
L	LSP	**Liskov substitution principle** "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program". See also design by contract.
I	ISP	**Interface segregation principle** "many client specific interfaces are better than one general purpose interface."[5]
D	DIP	**Dependency inversion principle** one should "Depend upon Abstractions. Do not depend upon concretions."[5] Dependency injection is one method of following this principle.



Refactoring

Definition

Refactoring consists of improving the internal structure of an existing program's source code, while preserving its external behavior.

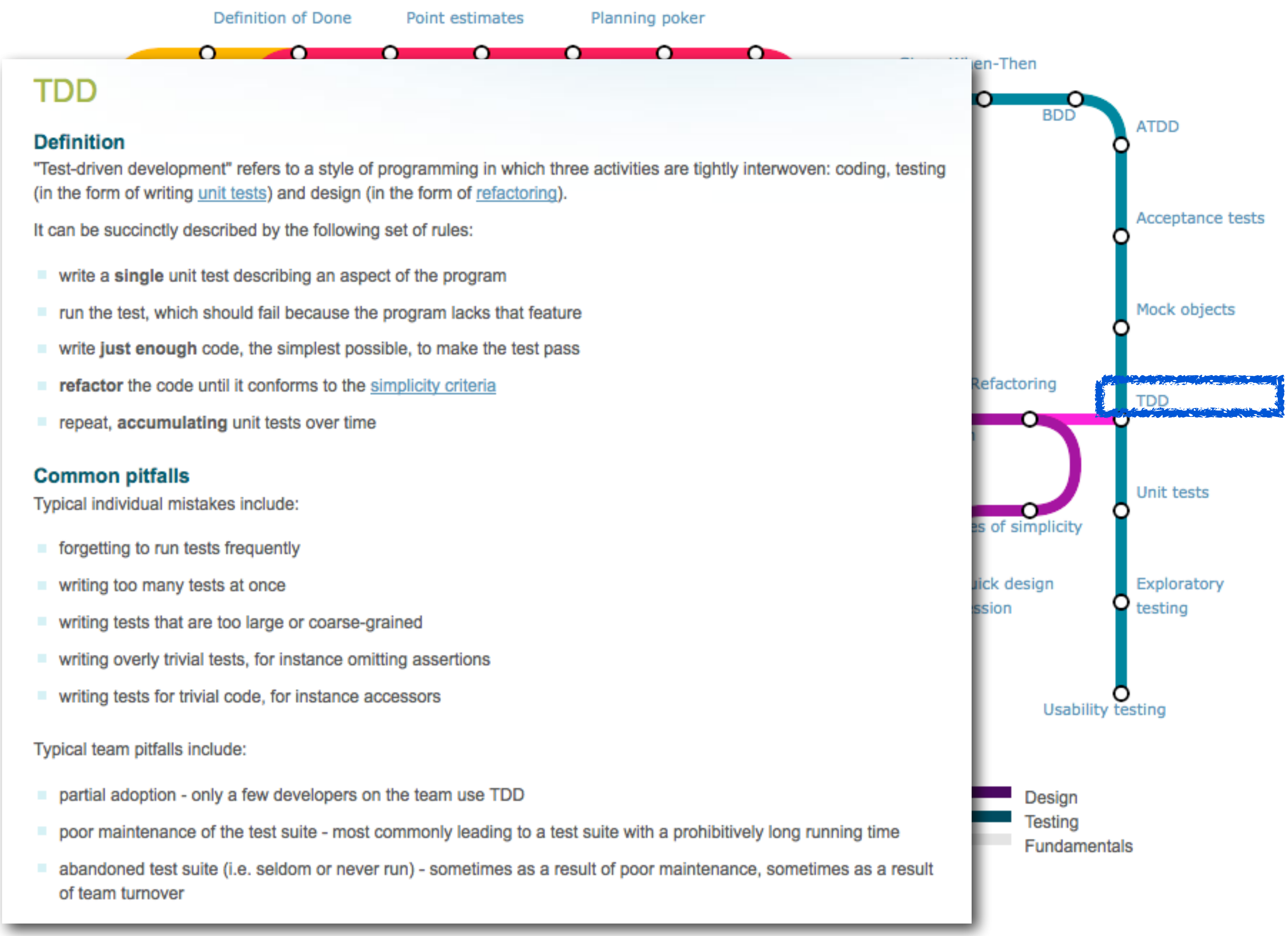
The noun "refactoring" refers to one particular behaviour-preserving transformation, such as "Extract Method" or "Introduce Parameter".

Common pitfalls

Refactoring does *not* mean:

- rewriting code
- fixing bugs
- improve observable aspects of software such as its interface

Refactoring in the absence of safeguards against introducing defects (i.e. violating the "behaviour preserving" condition) is risky. Safeguards include aids to regression testing including automated unit tests or automated acceptance tests, and aids to formal reasoning such as type systems.



TDD

Definition

"Test-driven development" refers to a style of programming in which three activities are tightly interwoven: coding, testing (in the form of writing unit tests) and design (in the form of refactoring).

It can be succinctly described by the following set of rules:

- write a **single** unit test describing an aspect of the program
- run the test, which should fail because the program lacks that feature
- write **just enough** code, the simplest possible, to make the test pass
- **refactor** the code until it conforms to the simplicity criteria
- repeat, **accumulating** unit tests over time

Common pitfalls

Typical individual mistakes include:

- forgetting to run tests frequently
- writing too many tests at once
- writing tests that are too large or coarse-grained
- writing overly trivial tests, for instance omitting assertions
- writing tests for trivial code, for instance accessors

Typical team pitfalls include:

- partial adoption - only a few developers on the team use TDD
- poor maintenance of the test suite - most commonly leading to a test suite with a prohibitively long running time
- abandoned test suite (i.e. seldom or never run) - sometimes as a result of poor maintenance, sometimes as a result of team turnover

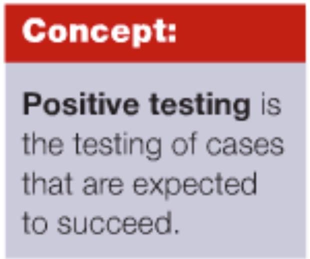
Unit Testing

Each unit of an application may be tested.

1. Method, class, module (package in Java).

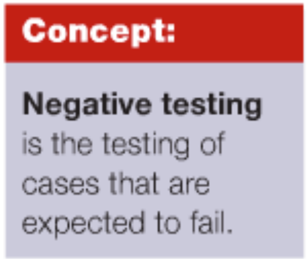
Can (should) be done during development.

1. Finding and fixing early lowers development costs (e.g. programmer time).
2. A test suite is built up.



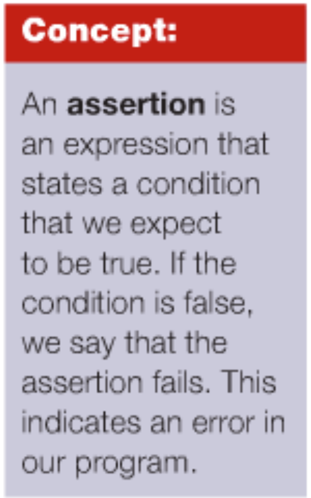
Concept:

Positive testing is the testing of cases that are expected to succeed.



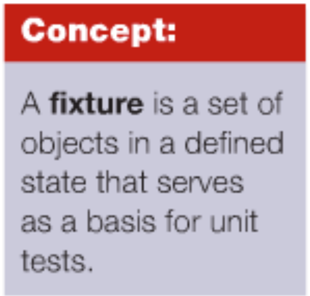
Concept:

Negative testing is the testing of cases that are expected to fail.



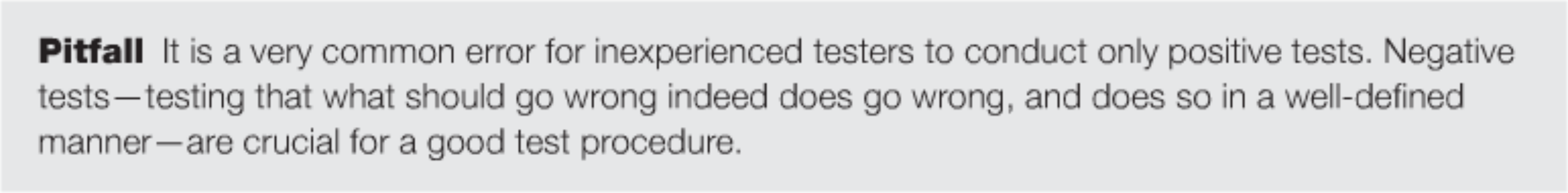
Concept:

An **assertion** is an expression that states a condition that we expect to be true. If the condition is false, we say that the assertion fails. This indicates an error in our program.



Concept:

A **fixture** is a set of objects in a defined state that serves as a basis for unit tests.



Pitfall It is a very common error for inexperienced testers to conduct only positive tests. Negative tests—testing that what should go wrong indeed does go wrong, and does so in a well-defined manner—are crucial for a good test procedure.

Testing Fundamentals

Understand what the unit should do – its contract.

- You will be looking for violations.
- Use positive tests and negative tests.

Test boundaries.

- Zero, One, Full.

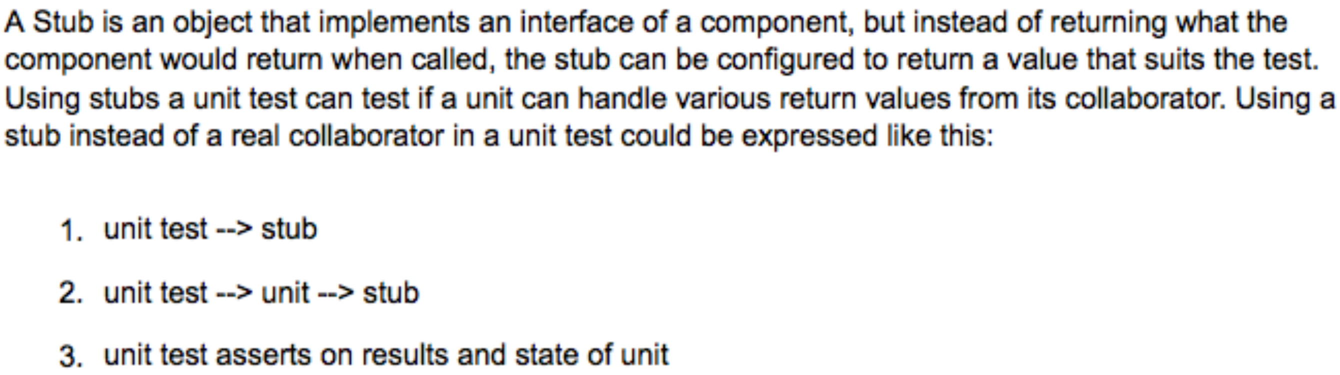
  Search an empty collection.

  Add to a full collection.

Test harness

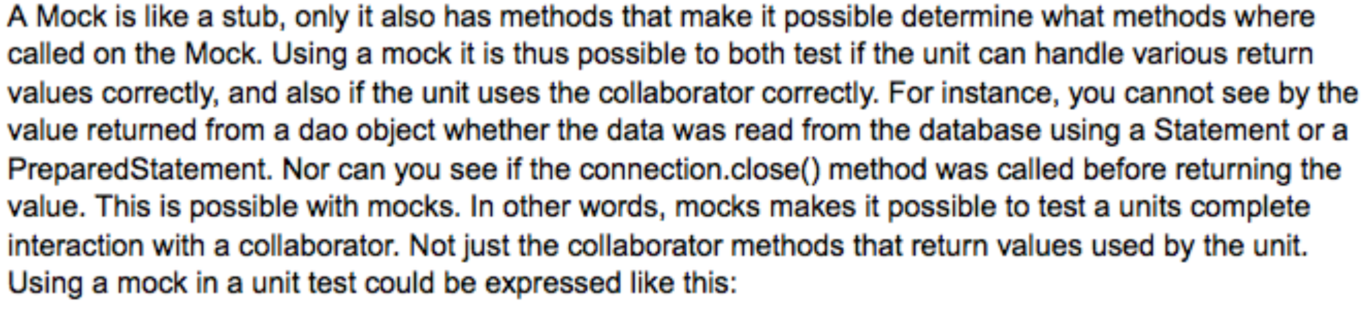
- Additional test classes are written to automate the testing.
- Objects of the harness classes replace human interactivity.
- Creativity and imagination required to create these test classes.
- Test classes must be kept up to date as functionality is added.

Stubs, Mocks & Proxies

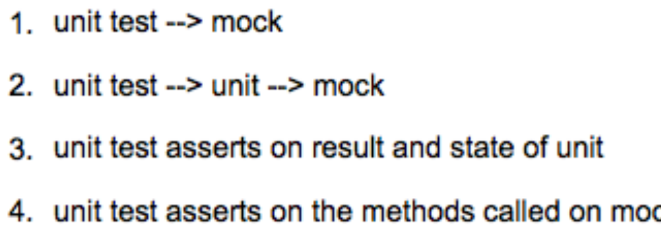


A Stub is an object that implements an interface of a component, but instead of returning what the component would return when called, the stub can be configured to return a value that suits the test. Using stubs a unit test can test if a unit can handle various return values from its collaborator. Using a stub instead of a real collaborator in a unit test could be expressed like this:

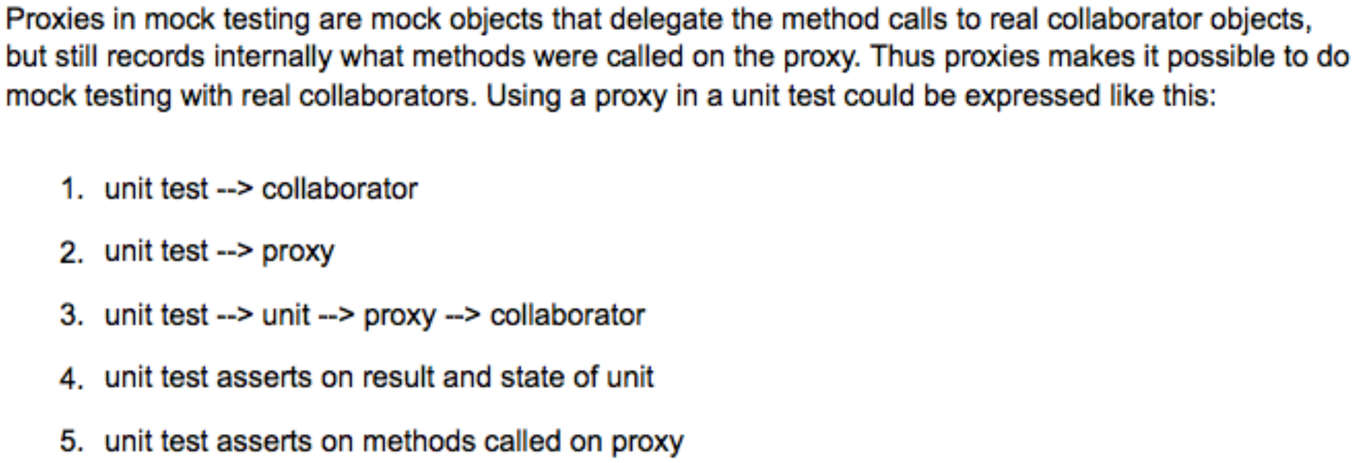
1. unit test --> stub
2. unit test --> unit --> stub
3. unit test asserts on results and state of unit



A Mock is like a stub, only it also has methods that make it possible determine what methods where called on the Mock. Using a mock it is thus possible to both test if the unit can handle various return values correctly, and also if the unit uses the collaborator correctly. For instance, you cannot see by the value returned from a dao object whether the data was read from the database using a Statement or a PreparedStatement. Nor can you see if the connection.close() method was called before returning the value. This is possible with mocks. In other words, mocks makes it possible to test a units complete interaction with a collaborator. Not just the collaborator methods that return values used by the unit. Using a mock in a unit test could be expressed like this:



1. unit test --> mock
2. unit test --> unit --> mock
3. unit test asserts on result and state of unit
4. unit test asserts on the methods called on moc



Proxies in mock testing are mock objects that delegate the method calls to real collaborator objects, but still records internally what methods were called on the proxy. Thus proxies makes it possible to do mock testing with real collaborators. Using a proxy in a unit test could be expressed like this:

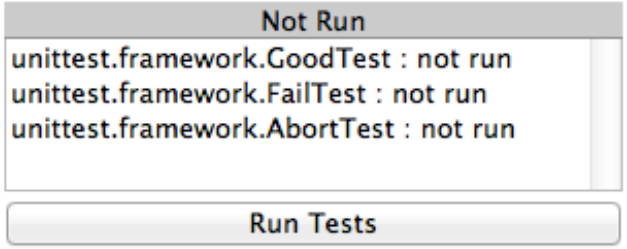
1. unit test --> collaborator
2. unit test --> proxy
3. unit test --> unit --> proxy --> collaborator
4. unit test asserts on result and state of unit
5. unit test asserts on methods called on proxy

http://tutorials.jenkov.com/java-unit-testing/stub-mock-and-proxy-testing.html

Unit Testing Framework

The most common misconception about unit testing frameworks is that they are only testing tools. They are development tools same as your editor and compiler. Don't keep this powerful development tool in reserve until the last month of the project, use it throughout. Your unit testing framework can help you formalize requirements, clarify architecture, write code, debug code, integrate code, release, optimize, and of course test.

Unit testing frameworks are not hard to create from scratch. It is worth the effort to create your own because you will understand it better and be able to tailor it to your own needs. A simple change to the unit testing framework can often save you large amounts of development time. But to realize this savings you must feel comfortable and confident about extending your framework.



```
unittest.framework.GoodTest : not run
unittest.framework.FailTest : not run
unittest.framework.AbortTest : not run
```

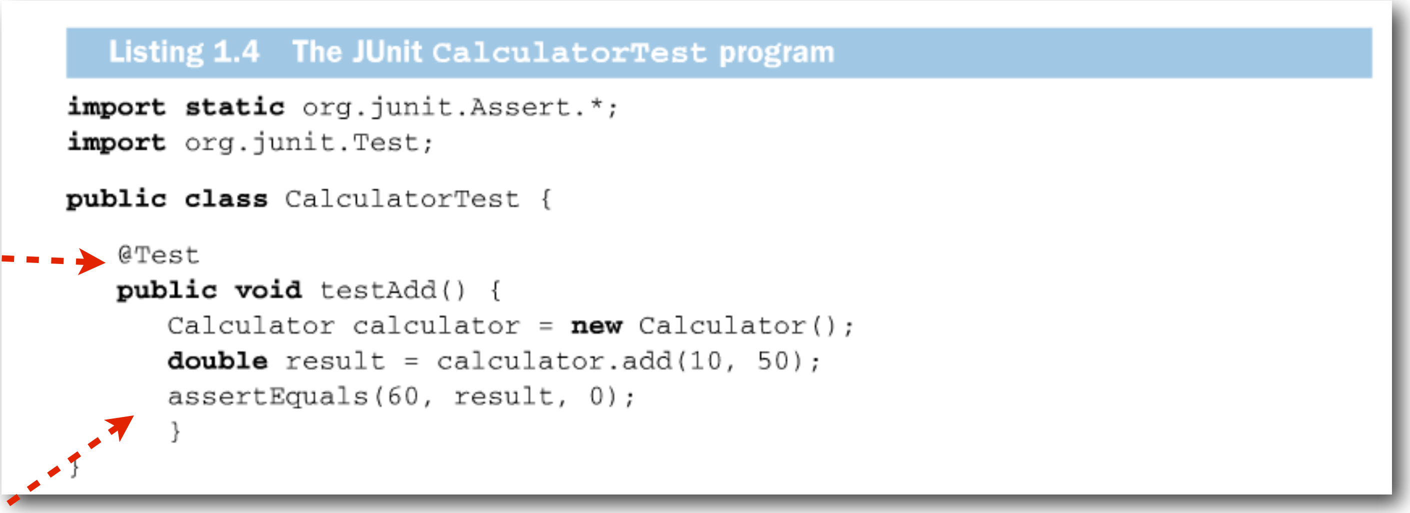
Most languages already have a unit testing framework available for download from XProgramming.com. Use this free version as a starting point. See how it works, then create your own. The team must claim ownership of the unit testing framework and be able to change any part of it. JUnit is quickly becoming the standard for unit testing in Java. If you download a unit test framework refactor it and make it your own so you understand how to extend it.

ExtremeProgramming.org home | XP Rules | A Bug is Found | About the Author

JUnit

Annotation "marks" method for Test Framework

Assert statements "passes" or "fails" a test.



Listing 1.4 The JUnit `CalculatorTest` program

```
import static org.junit.Assert.*;
import org.junit.Test;

public class CalculatorTest {

    @Test
    public void testAdd() {
        Calculator calculator = new Calculator();
        double result = calculator.add(10, 50);
        assertEquals(60, result, 0);
    }
}
```

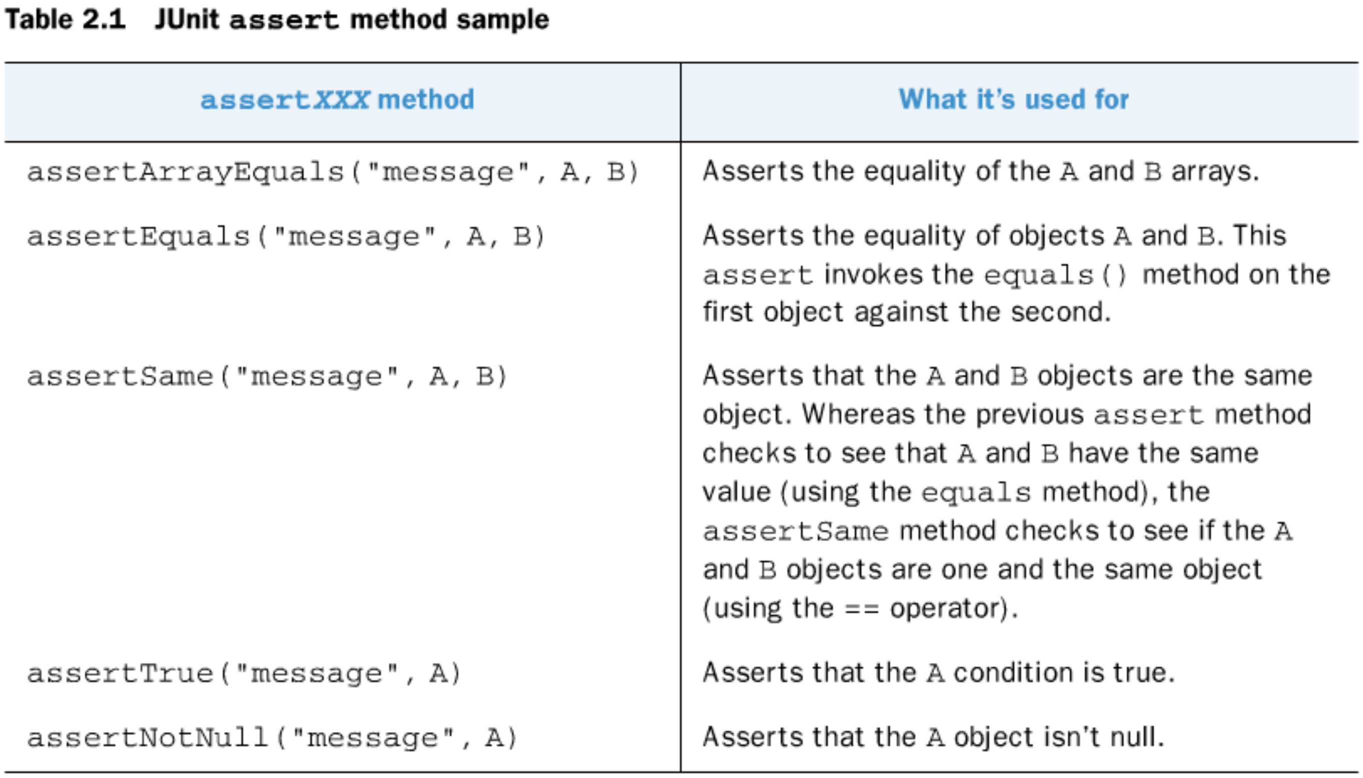
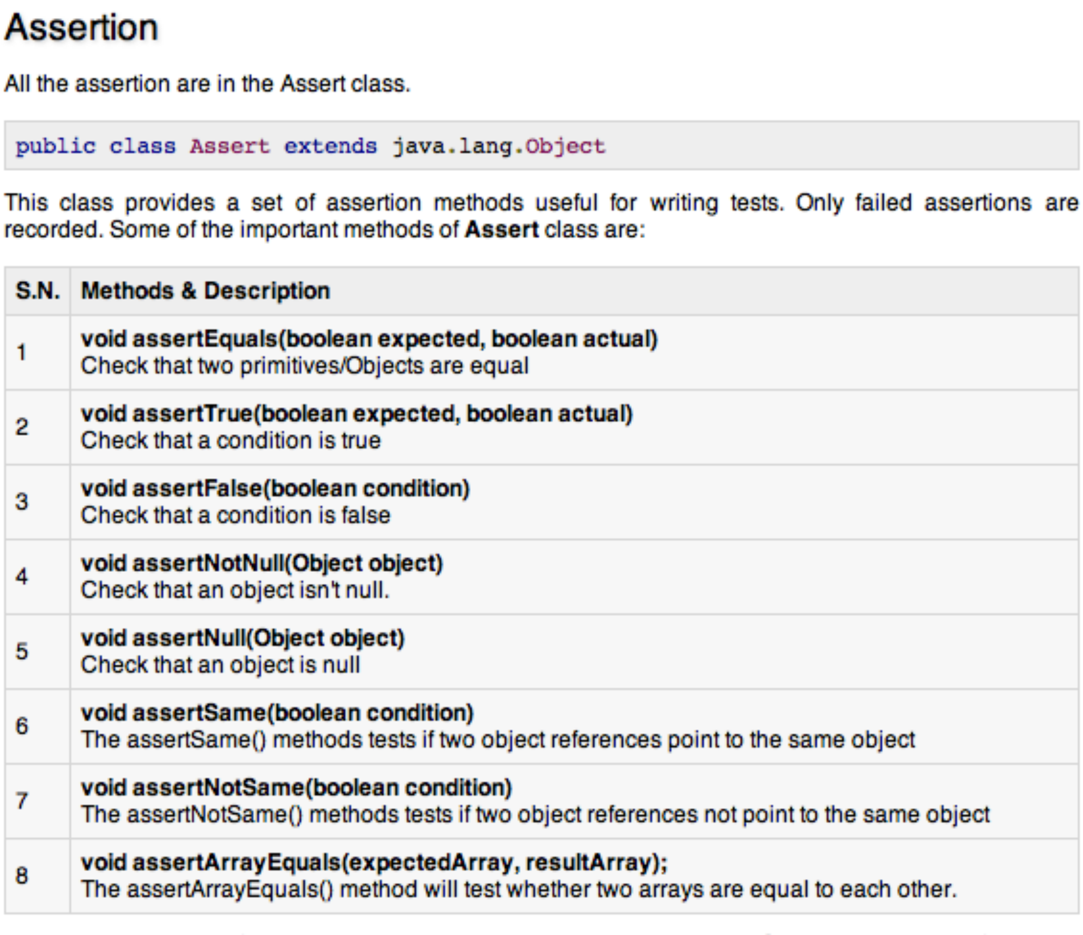


Table 2.1 JUnit `assert` method sample

`assertXXX` method	What it's used for
`assertArrayEquals("message", A, B)`	Asserts the equality of the `A` and `B` arrays.
`assertEquals("message", A, B)`	Asserts the equality of objects `A` and `B`. This `assert` invokes the `equals()` method on the first object against the second.
`assertSame("message", A, B)`	Asserts that the `A` and `B` objects are the same object. Whereas the previous `assert` method checks to see that `A` and `B` have the same value (using the `equals` method), the `assertSame` method checks to see if the `A` and `B` objects are one and the same object (using the `==` operator).
`assertTrue("message", A)`	Asserts that the `A` condition is true.
`assertNotNull("message", A)`	Asserts that the `A` object isn't null.

JUnit



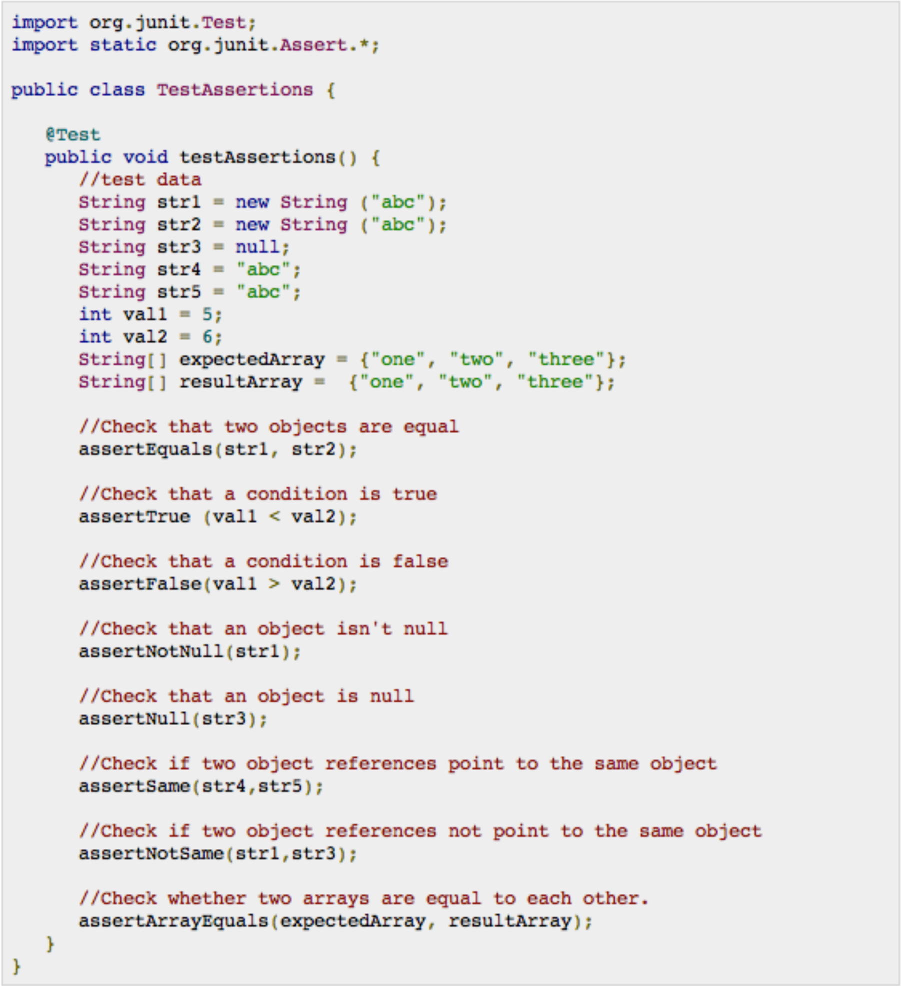
Assertion

All the assertion are in the Assert class.

```
public class Assert extends java.lang.Object
```

This class provides a set of assertion methods useful for writing tests. Only failed assertions are recorded. Some of the important methods of **Assert** class are:

S.N.	Methods & Description
1	**void assertEquals(boolean expected, boolean actual)** Check that two primitives/Objects are equal
2	**void assertTrue(boolean expected, boolean actual)** Check that a condition is true
3	**void assertFalse(boolean condition)** Check that a condition is false
4	**void assertNotNull(Object object)** Check that an object isn't null.
5	**void assertNull(Object object)** Check that an object is null
6	**void assertSame(boolean condition)** The assertSame() methods tests if two object references point to the same object
7	**void assertNotSame(boolean condition)** The assertNotSame() methods tests if two object references not point to the same object
8	**void assertArrayEquals(expectedArray, resultArray);** The assertArrayEquals() method will test whether two arrays are equal to each other.



```
import org.junit.Test;
import static org.junit.Assert.*;

public class TestAssertions {

   @Test
   public void testAssertions() {
      //test data
      String str1 = new String ("abc");
      String str2 = new String ("abc");
      String str3 = null;
      String str4 = "abc";
      String str5 = "abc";
      int val1 = 5;
      int val2 = 6;
      String[] expectedArray = {"one", "two", "three"};
      String[] resultArray =  {"one", "two", "three"};

      //Check that two objects are equal
      assertEquals(str1, str2);

      //Check that a condition is true
      assertTrue (val1 < val2);

      //Check that a condition is false
      assertFalse(val1 > val2);

      //Check that an object isn't null
      assertNotNull(str1);

      //Check that an object is null
      assertNull(str3);

      //Check if two object references point to the same object
      assertSame(str4,str5);

      //Check if two object references not point to the same object
      assertNotSame(str1,str3);

      //Check whether two arrays are equal to each other.
      assertArrayEquals(expectedArray, resultArray);
   }
}
```

http://www.tutorialspoint.com/junit/junit_using_assertion.htm

Behavior Driven Development

Title
The story should have a clear, explicit title.

Narrative

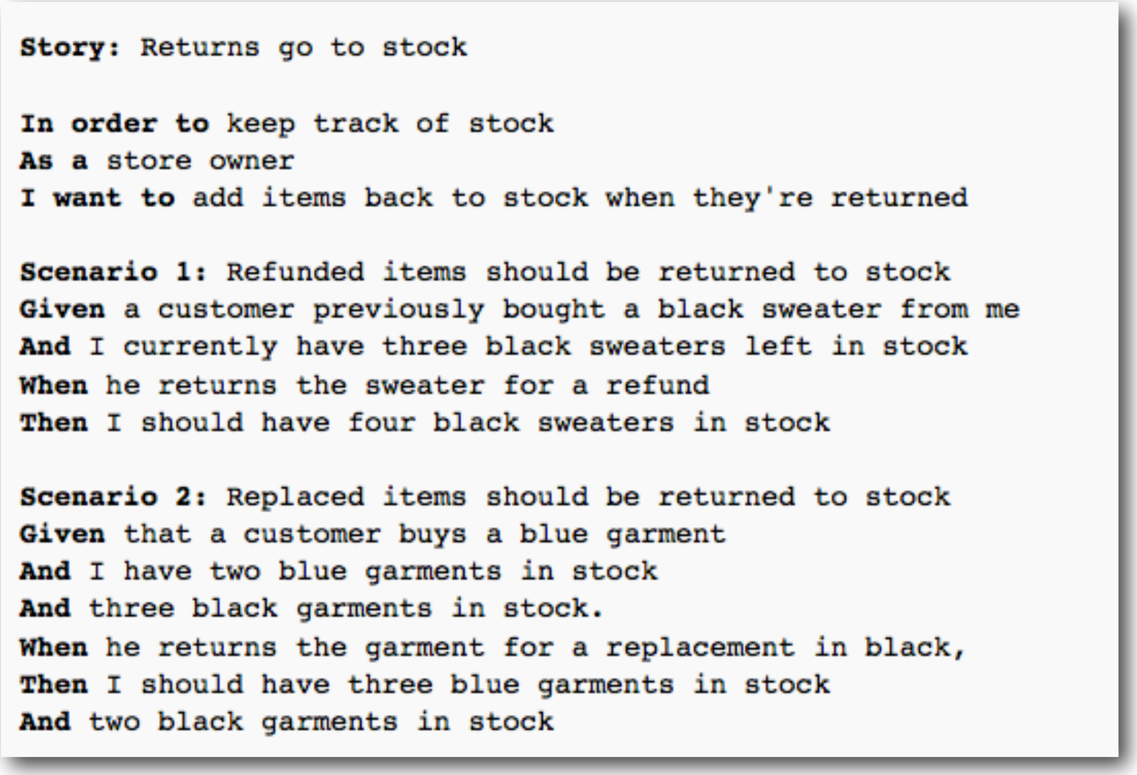
A short, introductory section that specifies

- who (which business or project role) is the driver or primary stakeholder of the story (the actor who derives business benefit from the story)
- which effect the stakeholder wants the story to have
- what business value the stakeholder will derive from this effect

Acceptance criteria or scenarios

a description of each specific case of the narrative. Such a scenario has the following structure:

- It starts by specifying the initial condition that is assumed to be true at the beginning of the scenario. This may consist of a single clause, or several.
- It then states which event triggers the start of the scenario.
- Finally, it states the expected outcome, in one or more clauses.



```
Story: Returns go to stock

In order to keep track of stock
As a store owner
I want to add items back to stock when they're returned

Scenario 1: Refunded items should be returned to stock
Given a customer previously bought a black sweater from me
And I currently have three black sweaters left in stock
When he returns the sweater for a refund
Then I should have four black sweaters in stock

Scenario 2: Replaced items should be returned to stock
Given that a customer buys a blue garment
And I have two blue garments in stock
And three black garments in stock.
When he returns the garment for a replacement in black,
Then I should have three blue garments in stock
And two black garments in stock
```

http://en.wikipedia.org/wiki/Behavior-driven_development

http://dannorth.net/introducing-bdd/