part of the "Exploring Scrum" series

Patterns that Make Scrum Work

Understanding and Scaling Scrum

Dan Rawsthorne, PhD, PMP, CST

3Back

Exploring Scrum: Patterns that Make Scrum Work

Understanding and Scaling Scrum

Dan Rawsthorne, PhD, PMP, CST

This book is for sale at http://leanpub.com/PPSAD

This version was published on 2013-12-30

To Mel… you will be missed

Contents

Introduction

Scrum is the most common, and popular, **Agile Development** framework in the world. In this book I will describe some of the basic **Pattern**s that are used in, and with, **Scrum**. This will:

- Help you understand how **Scrum** works, and why it is what it is, and
- Introduce some basic **Scaling Patterns**, which allow us to extend **Scrum** to work in an **Organization**.

Scrum's People

Scrum is a very simple **Framework**, and it involves two *types* of people:

1. **Developer**s: People who build, produce, develop, or do something. Since the things we do are often complicated, these people are organized into **Teams**, and are called **Team Members**, **Developers**, or just plain worker-bees.
2. **Decider**s: People who decide what needs to built, produced, developed, or done *next*. These people are held **Accountable** for their decisions – they *own* their decisions – and I will refer to them as **Leader**s or **Owner**s.

Of course, **Scrum** is not that simple – this is not all of **Scrum**. There are **Schedule**s and **Cost**s to manage, advice and guidance on what to build to give, and a simple **Agile Process** that surrounds everything.

But the important stuff is about **Team**s and **Decider**s. What **Team**s do we need, and what do they do? What **Decider**s do we need, and who should they be? Who helps the **Decider**s get the information they need? And so on… these are the main questions in **Scrum**, and things the **Pattern**s involve.

To put it bluntly: when we are thinking about **Scrum** we shouldn't be thinking about **Process**, we should be thinking about Decision-Makers; we need to have the right people in the right place at the right time, making the right decisions – and being **Accountable** for those decisions.

Patterns and AntiPatterns

The advice in this book is largely presented in the form of **Pattern**s and **AntiPattern**, so I better explain them to you. :)

At its simplest, a **Pattern** is defined as a "solution to a problem in a context" and each **Pattern** I give in this book has the following parts:

- **Pattern Name**: a descriptive name…
- **Problem**: what needs to happen…
- **Context**: what is going on…
- **Solution**: short statement of solution…
- **Discussion and Examples**: often including non-software examples

Now, the **Pattern**s in this book *aren't* the **Design Pattern**s you might be used to. Most **Design Pattern**s are solutions to tactical problems or low-level strategic problems. For example, you might find many different **Design Pattern**s describing different ways to run **Daily Scrum**s, or you might find a **Design Pattern** telling you that you should have **Daily Scrum**s. People often try to use these **Design Pattern**s as recipes – as advice to follow more-or-less blindly.

The **Pattern**s in this book are more like **Architectural Patterns**; they describe the "big ideas" that form the "shape" of Scrum, whereas **Scrum**'s **Design Pattern**s often describe *how* **Scrum** works, or ways of *making* Scrum work… I hope you understand the difference… **Architectural Pattern**s describe *why* something is what it is – why Scrum is Scrum and not something else, for example – while **Design Pattern**s are used by a self-organizing **Team** to help them figure out how to solve a particular problem they may have.

I will also have a few **AntiPattern**s; things we see quite often, but are not a good idea. One way to think of **AntiPattern**s is that they are *"things that look like they should work, but don't."* when I present an **AntiPattern**, it will have the following parts:

- **AntiPattern Name**: a descriptive name…
- **Reason it Exists**: why people think it's a good idea; why it happens…
- **Discussion**: why it's not a good idea, additional caveats, etc…

So, it sounds like **Pattern**s and **AntiPattern** encapsulate good advice you *should just follow*, right?

It's not that simple, really. **Pattern**s and **AntiPattern**s can't be followed blindly and be expected to work. **Pattern**s and **AntiPattern**s contain some guidance and advice that will help you frame *your* questions and manage *your* thoughts in order to deal with *your* situation. **Pattern**s and **AntiPattern**s help you understand what is *important* about a situation, and allows you to focus and analyze, rather than thrash and panic. They allow you to reuse thoughts and analysis that other people have had, but should *not* be followed blindly.

In fact, let's think about that recipe thing…

Have you ever followed a recipe and the food was inedible? Who did you blame? – the yourself or the recipe? Think about that for a minute, talk to your friends about it… I'll wait.

* * *

Ahh, you're back. So, now you understand that *even* recipes *aren't* to be followed blindly. As I hope to realize, the cook is **Accountable** for the results, not merely following the recipe. Same with **Patterns**… just sayin'… it gets complicated :)

Accountability

I've already used the work "**Accountable**" a couple of times, so I better make sure we have a common understanding of what it means.

Accountability is a very important concept in **Scrum**. Most people don't really understand what **Accountability** is; they really don't understand the notion of a person being **Accountable**. In fact, there is not even a word for it in many languages. So, let me explain what it means for *us* when we're talking about it.

Here's the basic definition (see Wikipedia): "**Accountability** is answerability, blameworthiness, liability, and the expectation of account-giving… **Accountability** is the acknowledgment and assumption of responsibility for actions, products, decisions, and policies…" It is frequently described as a relationship between individuals; **Person-A** is **Accountable** to **Person-B** *for* **Thing-C** when **Person-A** may be "held to account" for **Thing-C** to **Person-B**. The phrases "held to account" or "make an accounting" both mean *being able to explain yourself.* Generally speaking, we usually hold people **Accountable** for something being done, or a decision being made. Let me discuss each of these cases:

- **Person-A** is "**Accountable** for *something being done*" means that **Person-A** needs to make sure that the something gets done – **Person-A** doesn't have to do it personally – and **Person-A** needs to be able to explain 'Why?' if it doesn't get done.
- **Person-A** is **Accountable** for *a decision being made*" means that **Person-A** makes sure the decision is made, and that **Person-A** *own*s the decision. In other words, **Person-A** needs to be able to explain the decision, and **Person-A** is also **Accountable** for *all* the consequences (both intended and unintended) of that decision.

Generally speaking, only individual people can be held **Accountable** for things. Many people speak of Team, Group, or Organizational **Accountability**, but this is a difficult concept – and we'll see this come up later.

Conventions

Most terms in **bold** will be defined in a glossary. Nouns (**Product**) are **bold**, Verbs (***Review***) are ***bold-italic***

In this book

Scrum has undergone almost constant change since the 1990's when it was first named, implemented, and described for software[1]. There are two (significantly) different versions of **Scrum** that have arisen: I will call them **Original Scrum** and **Modern Scrum**.

In the first chapter I will show how **Original Scrum** arose from a simple application of a few **Pattern**s, and in the next chapter I will show how this morphs into **Modern Scrum** by applying a few more. After that I will introduce **Scaling Pattern**s and analyze some popular **Agile Scaling Frameworks** by comparing them to these **Pattern**s.

With these definitions and explanations, we are now ready to explore **Scrum**.

[1]The first people to document **Scrum** for Software were Ken Schwaber and Jeff Sutherland. We owe them a big debt of thanks!

Original Scrum

I'm going to go through a few Patterns, and build **Original Scrum** one **Pattern** at a time.

I will describe these **Pattern**s as we find them "outside" the software context, in order to show that there is *nothing special* about these **Pattern**s – they are *not* software-specific. What *is interesting* is that we are (only recently) bringing them to mainstream Software Development.

After the **Pattern**s are described, I will assembly them into a **Software Development Scrum Team**…

Pattern: Well-Formed Team

The core of **Scrum** – its most basic idea – is the **Well-Formed Team**, so that's where we start…

Problem:

There is an **Item** of Work to be done.

Context:

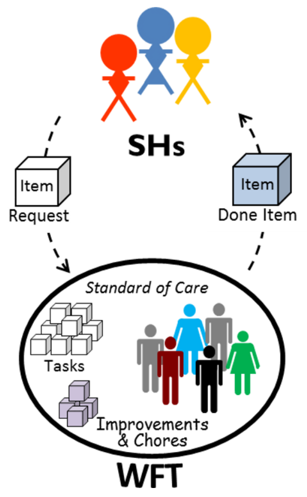
- The **Item** has one (or more) **Stakeholder**s who want the work to be done, and these **Stakeholder**s can't (or don't want to) do the work themselves.
- Each Item has **Acceptance Criteria**; the **Stakeholder**s have an *idea* about what the finished result should look like, be, or do – and these **Acceptance Criteria** will be shared as part of the **Item**.

Solution:

Use a **Well-Formed Team (WFT)** to do the work, which means:

- the **WFT** is **Self-Organized**: the **Team Member**s determine what **Tasks** are necessary for them to accomplish the work and complete the **Item**. The **Team Member**s are co-located (if not physically, then virtually) and have frequent **Coordination Meetings** to stay 'synced-up' while they are working. The **WFT** manages itself so that the **Stakeholders** don't have to.
- the **WFT** is **Self-Contained**: the **Team Member**s (collectively) have all the knowledge and skills they need in order to accomplish the work and complete the **Item**. The **Stakeholder**s will not have to worry about delays based on waiting for outsiders to complete *their part of the work*.
- the **Team Member**s are **Value-Driven**: they value working together; they are constantly working on **Improvements** of themselves, their **Team**, their environment, and their tools; and they do their **due diligence** to complete **Item**s to/with the **Standard of Care** they deserve. Doing their **due diligence** often requires them to do **Chores** that are not *directly* involved in working on the **Item**s. The **Stakeholder**s can trust that the **WFT** has **Integrity** – the **Team Member**s are **Professional**s – and will meet the **Acceptance Criteria** while meeting the appropriate **Standard of Care**[2].

[2]In Software, this notion of **Professionalism** is captured in the **Software Craftsmanship** movement.



The Well-Formed Team

Generally speaking, this **Pattern** specifies the following **Accountabilities**:

- The **Well-Formed Team** is **Accountable** to the **Stakeholder**s for using the appropriate **Standard of Care** and satisfying the **Item**'s **Acceptance Criteria**,
- The **Well-Formed Team** is **Accountable** to identify and carry out **Improvements** and **Chores** that are required for the **WFT**'s successful continuation, and
- The **Team Members** are **Accountable** to *each other* to be good **Team Member**s and live the **Values**.

Discussion and Examples:

We find **Well-Formed Teams** everywhere, in all walks of life, doing all sorts of work. The range from the Team of gardeners that maintains your yard and garden, to the team of mechanics that works on your car, to the team of doctors and nurses that works on you in the hospital.

Well-Formed Teams are usually thought of as specialists, professionals, or experts, in a particular domain. **Well-Formed Team**s can come in many sizes:

- A **Well-Formed Team** could be a 1-man Team, like my plumber Jerry.
- A **Well-Formed Team** could be a Small Team, like the collection of mechanics at a car repair center, the members of a brick-laying team, cooks and other workers in a (small) restaurant kitchen, or the truck-load of gardeners who come by to fix up your yard and garden.
- A **Well-Formed Team** could be Large, like a Construction Company, a Hospital, or the cooks and other workers in a (large) restaurant kitchen. However,
- The internal organization of this *large* **Well-Formed Team** *must* consist of **Well-Formed Team**s.

- In other words, it is a recursive definition: a Team made up of **Well-Formed Team**s whose governance is also done by **Well-Formed Team**s is a **Well-Formed Team**. This will be explored in the scaling patterns later in this book.

Well-Formed Teams are **Self-Contained** and **Self-Organized**, which means that the **Team Member**s figure out how to work together, combining their skills, in order to get the work done. This can take many different forms – here are just a few:

- The **WFT** could use an adaptable assembly line – like in a kitchen or brick-laying team. The work, itself, follows a certain flow – and it is adapted as necessary. There is often a member of the Team (like the sous-chef or Master Mason) who is orchestrating this flow.
- The **WFT** could normally work as individuals, but **Pair** and **Swarm** as needed. We see this pattern of behavior on most **WFT**s, but it is very obvious with mechanics and medical professionals.
- The **WFT**'s **Team Member**s *always* work in pairs, sharing expertise (and looking out for each other) while doing the work. This is often done when dangerous machinery is involved.

That being said, the third part of the definition (**Value-Driven**) is the most interesting, for a couple of reasons.

The first reason is because **Well-Formed Team**s are constantly improving. This has many facets: they need to increase their **Knowledge**, improve their **Teamwork**, and improve their **Tools** and **Environment**. This takes time (in the form of internally-generated **Items** or **Tasks**), and it is time that will *not* be spent working on the **Item**s coming from the **Stakeholder**s; it is just part of the *cost of doing work*.

The second reason is because of the notions of **Standard of Care** and **Acceptance Criteria**…

A **Well-Formed Team** must have a **Standard of Care** that it adheres to in its work. This **Standard of Care** is crucial and inviolate, because the **Stakeholder**s are relying on it. The **Stakeholder**s are relying on the **Professionalism** and **Integrity** of the **WFT**; the **Stakeholder**s want the work done *right*, and they are trusting the **WFT** to know what that means – and to do it *every* time. We would like **Team Member**s to be **Self-Motivated**, and do the work to the **Standard of Care** without being told, and this would be a very high level of **Professionalism**.

This is a very interesting concept. If we think of what it takes to get an **Item** ***Done***, we can see that a ***Done*** **Item** is one that meets the **Acceptance Criteria** *and* that the Team used the appropriate **Standard of Care** while doing the **Item**. To make it simple, we say that a ***Done*** **Item** met both the **Acceptance Criteria** *and* the **Standard of Care**.

On the one hand, the **Acceptance Criteria** come from the **Stakeholder**s (as part of the **Item**), and defines what the **Stakeholder**s want, need, expect, or are hoping for. The **Acceptance Criteria** may be either ambiguous or well-defined; they may be documented or undocumented; the idea is that the **Stakeholder**s have *some idea* about what the end result should be. Here are some examples:

- “Large sirloin, medium-rare, substitute broccoli for the asparagus, blue cheese dressing on the side”
- “I need a retaining wall set up right over here…”
- “My brakes don’t work! Fix them!”
- “My dad is sick… please help him…”
- “Just mow the grass and clean up those two garden beds today…”

Doing its **due diligence** to meet the appropriate **Standard of Care** often requires the **WFT** to do **Chores**, which (like **Improvements**) are internally-generated **Items** or **Tasks** that are *not* directly involved with completing an **Item**. For example, painters must clean their brushes, mechanics must clean their tools and dispose of old oil appropriately, workers in a kitchen must clean the dishes and equipment, and so on. In my book (*Exploring Scrum: the Fundamentals*), I refer to **Improvements** and **Chores**, collectively, as simply **Chores** – which are defined there as **Item**s that are needed, by not requested by the **Stakeholders**.

The fact that the **WFT** is **Self-Contained** implies that the **WFT** has the knowledge to understand and refine the **Acceptance Criteria** as needed. Part of the Team’s **Standard of Care** could be to ask questions and carry on a dialogue with the **Stakeholder**s in order to refine the **Acceptance Criteria** along the way.

On the other hand, the **Standard of Care** is *supplied* by the **WFT** – they are the experts on how to do the work… and they have the **Integrity** to do it the right way *every* time. The **Standard of Care** influences and defines how the **WFT** will do the work – what their internal process looks like. The **Standard of Care** could define what the end result has to *look like*, it could require certain process steps, whatever. A **WFT**’s **Standard of Care** could be highly personalized, or it could be determined by their Industry/Profession, or both.

In many Industries/Professions the notion of a minimum **Standard of Care** is enforced through laws, standards, inspections, or codes of ethics. For example:

- a Medical Professional will lose his/her license to practice medicine if it is determined that he or she did not meet the appropriate **Standard of Care** when practicing medicine;
- an electrician will lose his/her license if his/her work consistently fails inspections;
- a Restaurant Kitchen will be *shut down* if the staff does not meet the standards for cleanliness and food preparation;
- a mechanic will lose his/her certifications if the work is consistently done incorrectly;
- and so on…

As you can see, the **Well-Formed Team** is an powerful thing.

Pattern: Team Coach

The next thing to add to the **Scrum Team** we are building is the **ScrumMaster**, which is an example of a **Team Coach**.

Problem:

You want to develop and maintain a **Well-Formed Team**.

Context:

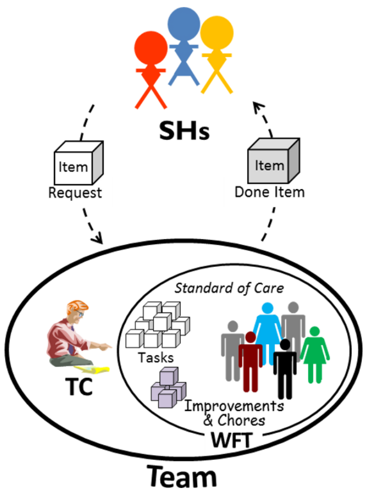
You have a group of people who either are, you think they could be, a **Well-Formed Team**.

Solution:

Have a **Team Member** acting as a **Team Coach** who will:

- Facilitate the **Well-Formed Team**'s **Self-Organization**, assuring that they improve their **Practices** and work together as a **Team**. This implies that the **Team Coach** should *not* have **Management Authority**, as the **Team** needs to *own* their own **Practices** – and it is inevitable that a **Team Coach** with **Management Authority** would wind up owning them… just sayin'…
- Make sure that the **WFT**'s **Team Member**s get whatever **Training** and **Coaching** they need to improve their knowledge and skills in order for the **WFT** to become more versatile and **Self-Contained**.
- Have frequent **Reflections** with the **WFT** to make sure it is constantly working on **Improvements** to its **Teamwork**, **Tools** and **Environment**.
- Work with the **WFT**'s **Team Member**s to help them understand the requirements of the **Standard of Care** they must use to do their work.
- Play a **Regulatory** role and make sure the Team has **Integrity** and does its **due diligence** constantly and consistently in order to meet the appropriate **Standard of Care** *every* time.

In this **Pattern** the **Team Coach** is **Accountable** for turning the **Team** into a **Well-Formed Team**. There may be more than one **Team Coach**, with **Accountability** for different facets of the **Well-Formed Team**. For example, one **Team Coach** may focus on **Knowledge** and **Tools**, while another focuses on **Teamwork** and **Self-Organization**.



Team Coach

At any given moment the **Team Coach** can be 'held to account' and must be able to discuss, enumerate, and explain what **Improvements** and **Chores** the **Team** is working on – what the **Team** is doing to improve itself. This is the **Accountability** The Team Coach has.

Discussion and Examples:

This **Pattern** is almost obvious; it is simply saying that a **Team** that is doing something complicated, or wants to change its behavior, needs a **Coach**. However, this pattern is evidenced in many different ways in different industries.

- In the Building Trades, where people go through the progression of **Apprentice** to **Journeyman** to **Master** as they mature and gain experience, all **Journeymen** and **Masters** are expected to mentor and train the **Apprentices**. Their intended goal is to produce a **Well-Formed Team** –- and it just happens organically as long as you have one or more **Masters** or **Journeymen** on your team.
- In the Military, it is part of a Non-Commissioned Officer's (NCO, Sergeant) fundamental set of responsibilities to train and mentor the Troops that he or she works with in order to turn them into a **Well-Formed Team**.
- In a restaurant kitchen, this is the sous-chef's job; it is the sous-chef who make sure that the people are trained and that the kitchen operates as a **Well-Formed Team**.

One thing that may not be obvious is that the **Team Coach** should be very wary of having **Skin in the Game**; that is, the **Team Coach** should be focused on the **Team** and its **Practices**, and not on its value-producing **Goal**s. Here are some *extreme* examples, just to illustrate the concept:

- A Team Doctor must be focused on the health of the Players, and not winning games. It the Team Doctor is a "Good Team Player" he/she can too easily put Players back in the game too soon… just sayin'…

- Having the Chief Mechanic as the owner of the shop can lead to problems, if the Chief Mechanic is also the **Team Coach** and starts concentrating on making money rather than **due diligence**. This is a *constant* challenge when it comes to this kind of work, and why it is recommended that *somebody else* do the coaching…

As you can see, adding a **Team Coach** to a **Well-Formed Team** is usually a good idea.

Pattern: Business Owner

So, we have the **Development Team** and the **ScrumMaster** taken care of; what we're missing from the **Scrum Team** is the **Product Owner**. The **Business Owner** I describe in this **Pattern** is the simplest version of a **Product Owner**.

Problem:

There is too much work competing for your **Well-Formed Team**'s time.

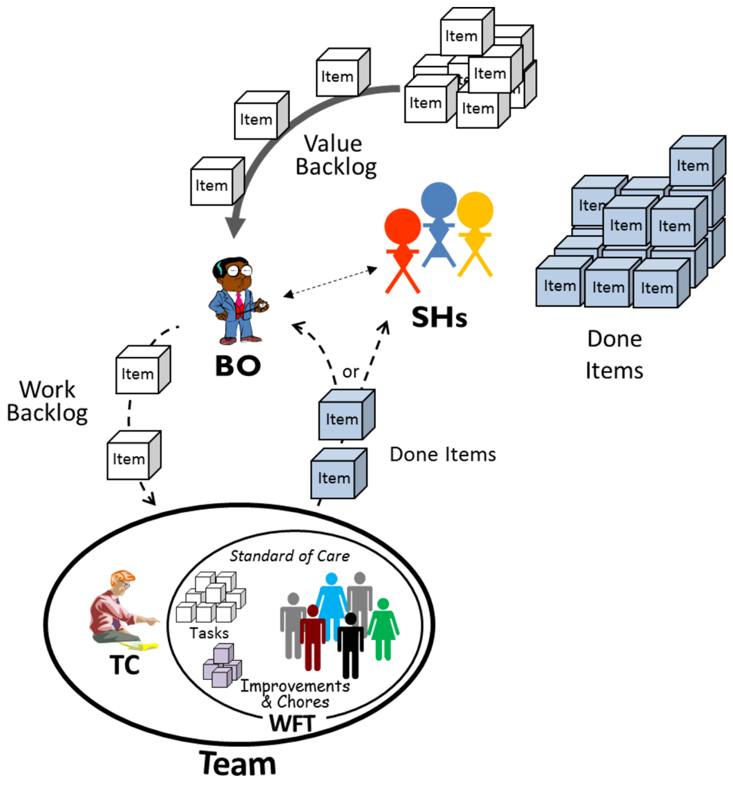
Context:

You have a **Team**, with multiple **Stakeholder**s (or one very conflicted **Stakeholder**) that have many **Items** they'd like your **Team** to work on. Your **Team** cannot do them all at once, and there is confusion about what to do next.

Solution:

Have a person, called the **Business Owner (BO)**, who sits *between* the **Stakeholder**s and the **Team**, who:

- Prioritizes/orders the work the **Stakeholder**s want into a single **Value Backlog**, and
- Moves the **Item**s from this **Stakeholder**'s **Value Backlog** to the **Team**'s **Work Backlog** at a rate that will not overload the **Team**.



The Business Owner

In addition to being **Accountable** for the above, this **Pattern** also requires the **Business Owner** to be **Accountable** for understanding the **Acceptance Criteria** for any **Item** that is put on the **Team**'s **Work Backlog**. We say that the **Business Owner** ***Represents*** the **Stakeholder**s and manages the **Value Backlog**.

Discussion, including examples:

Let me discuss this **Business Owner** concept with some of the **Well-Formed Teams** I introduced before:

- The **Business Owner** for my plumber Jerry is his Office Manager, who gets the calls from Clients describing what work needs to be done, and schedules them on Jerry's **Work Backlog** (his 'Job List'), *hoping* that Jerry is not overloaded on any given day. If Jerry can't get it all ***Done***, then the Office Manager rearranges the 'Job List' and works with the Clients to 'make it work'.
- The **Business Owner** for the collection of mechanics at a car repair center is the person at the front desk who asks the Car Owner's what is wrong with their cars and schedules their cars onto the Shop's **Work Backlog**. the **Business Owner** also works with the Car Owners to keep them notified of what is going on…

- The **Business Owner** for the kitchen personnel in a Restaurant is actually a shared role. All of the Waitstaff/Servers fill out meal tickets and deliver them to the kitchen, where they are put on the Team's **Work Backlog**, as shown here.

A Restaurant's Work Backlog

- The **Business Owner** for a Hospital Emergency Room has three parts:
- the Admitting Desk, who checks in new walk-in patients and puts them in order on the ER's **Work Backlog**,
- the Chief Resident, who meets Ambulances at the door, and immediately places emergency cases at (or near) the *front* of the ER's **Work Backlog**, and
- a medical professional performing triage in the waiting room, which allows some walk-ins to be *moved up* the ER's **Work Backlog** if their problems are urgent enough.

The second part of the **Business Owner**'s job: "Move the **Items** from this **Value Backlog** to the **Well-Formed Team**'s **Work Backlog** at a rate appropriate for the **WFT** to consume them" is very interesting.

- Sometimes the **Well-Formed Team** *asks* the **Business Owner** for a new **Item** when they are ready to start a new one,
- Sometimes the **Business Owner** *monitors* the **Team**'s **Work Backlog** and just replenishes it when it *looks a little low*,
- Sometimes the **Work Backlog** is populated at a rate *based on historical data* from the **Team** – like Jerry's daily 'Job List', and
- Nomatter what, the rate the **Well-Formed Team** completes **Items** from the **Work Backlog** is based on the difficulty of meeting the **Standard of Care** for the **Items** being worked on and how many **Improvements** and **Chores** the **Team** is undertaking.

The **Business Owner** is **Accountable** for the prioritization of the **Value Backlog** and the **Business Owner** is **Accountable** for knowing what **Item** is needed next. We say that the **Business Owner** *represents* the **Stakeholders** and *manages* the **Value Backlog**. These three things (the **Business Owner**, the **Value Backlog**, and the **Stakeholders**) go together. If I see a **Value Backlog**, I assume there is a **Business Owner** and **Stakeholders**, if I see a **Business Owner**, I know there is a **Value Backlog**. And there are always **Stakeholders**…

As you can see, the **Business Owner** is a good person for the **Well-Formed Team** to have around, because the **BO** both prioritizes and regulates the **Items** coming into the **Well-Formed Team**'s **Work Backlog** – thus allowing the **WFT** to focus on its necessary **Improvements**, **Chores**, and meeting the **Standard of Care** the **Items** deserve.

Pattern: Agile Product Development

The **Business Owner** I just discussed is used when we have **Value Backlogs** that are simply a *flow of work*, such as Backlogs of Bugs or service requests, for example. In these cases there is no need for **Agility** as we usually think of it.

However, we are often building **Products** that require some sort of **Feedback Loop** with the **Client**; in other words, we need some **Agile Development**. In this Pattern I will expand the **Business Owner**'s role to include *working* with a **Client** to develop a **Product** in an **Agile** way. When the **Business Owner** and **Client** have a **Project Plan** they are working with, I call the **Business Owner** a **Project Leader**.

Problem:

A **Client** wants a **Team** to ***Build*** and ***Deliver*** a **Product**.

Context:

- There is a **Client** that wants a **Product**
- There is a **Business Owner** working with this **Client**
- There is a **Well-Formed Team** that will be building the **Product**
- Building the **Product** will be complex, for any of a number of reasons, such as:
  - The **Client** does not have a firm grasp of the requirements, or
  - The **Well-Formed Team** doesn't know exactly what needs to be done, or
  - The requirements are expected to change; they are volatile.
- There may (or may not) be **Cost** or **Schedule** constraints that "need to be met". If there are, then this effort is called a **Project**…

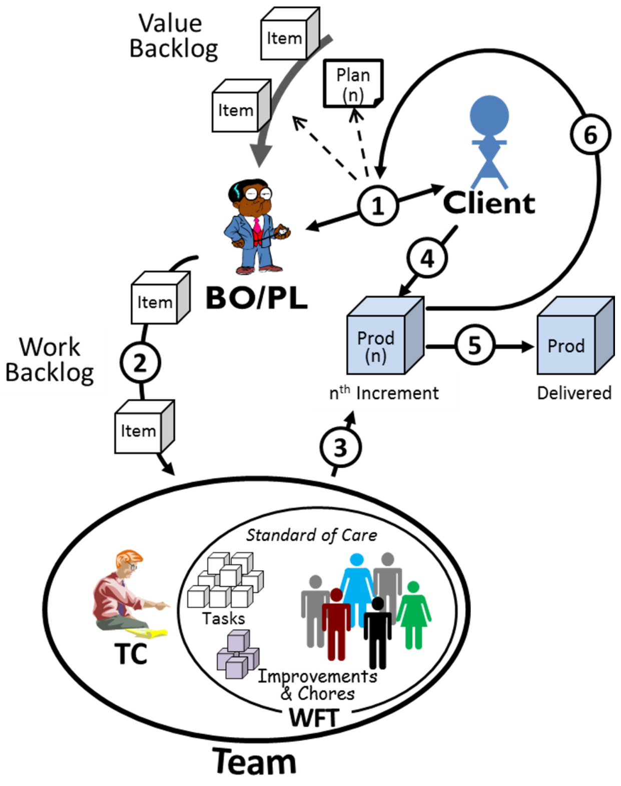
Solution:

There are two cases here: either there are **Cost** or **Schedule** constraints that "need to be met", or there are not. If there are, the **Business Owner** manages them by preparing and maintaining a **Project Plan**, and we call the **Business Owner** a **Project Leader**. Basically, a **Project Leader** is a **Business Owner** who is also maintaining a **Project Plan**. For shorthand purposes, I will refer to this **Role** as the **BO/PL**.

In either case, follow the following **Agile Process** steps:

1. The **Client** and the **BO/PL** discuss what should be built next, put appropriate **Item**s on the **Client**'s **Value Backlog**, and [if applicable] negotiate (or re-negotiate) the **Project Plan**.
2. The **BO/PL** moves some of these **Items** to the **Team**'s **Work Backlog** at a rate that will not overload the **Team**.

3. The **Team** builds a **Product Increment** based on the **Items** on the **Work Backlog**.
4. The **Client** ***Reviews*** the **Product Increment**.
5. If the **Product Increment** is acceptable by the **Client** for delivery, then the Team Cleans It Up and ***Deliver***s it.
6. If the **Product Increment** is *not* acceptable by the **Client** for delivery, then return to Step 1.



Agile Product Development

Discussion and Examples:

This type of process is **Iterative**, **Incremental**, and **Agile**:

- It is an **Iterative** process, since the overall process is repeated over and over – it is iterated,
- It is an **Incremental** process, since the **Product** is added to (incremented) every **Iteration**, and
- It is an **Agile** process, since the content of each **Iteration** is based on current knowledge.

This is a very simple **Agile Development Process**. During the **Development**, there is an ongoing creation of a flow of new **Items** being placed on the **Client**'s **Value Backlog**, based on the **Client**'s ***Review*** of the **Product Increments** and [if applicable] re-negotiations of the **Project Plan**. This **Value Backlog** is represented (to the **Team**) by the **BO/PO**, and is forwarded to them via their **Work Backlog**

Simple Business Owner - no Cost or Schedule Constraints

Let me give a simple example for the case that the **Agile Development** is *ad hoc*, there were no negotiated time or money constraints – we just got done when we got done. This example is when my son and I (the **Well-Formed Team**) built garden beds for my Wife (the **Client**) one vacation. Every morning my son and I would get up, eat some breakfast, and then spend a few hours working on the garden. We would go inside to clean up for lunch, and my wife would come outside and look at what had been done. After my wife had wandered around for a bit (***Reviewing*** the work), I (playing the role of **Business Owner**) would go out and talk with her to discuss what we should work on next. Then we'd all eat lunch and enjoy the rest of the day. We went through this process (**Iterated**) every day. After about 3-4 days, she liked what she saw, and the next day my son and I Cleaned Up the area, took tools back to the rental place, took the detritus to the dump, and so on. The day after that my wife took over the Garden (we ***Delivered***) and started planting flowers. And now we all have a garden.

Note that this simple **Business Owner** is *not* significantly different from the one we saw before, if there is no **Project Plan** involved. Because of the **Agile Process** in use here, the **Client** is constantly working with the **Business Owner** to figure out what to do next. As in the previous **Pattern**, the **Business Owner** is merely forwarding these new **Items** to the **Well-Formed Team**'s **Work Backlog** at an appropriate rate.

Business Owner as Project Leader

It is *very* common that the **Business Owner** is actually a **Project Leader**, and I'll give a few examples. The first one will be fleshed out in some detail, and the others are just suggestions for **Projects** – you fill in the blanks.

1. Let's say you want to have your kitchen re-modeled, so you hire a General Contractor to do it for you. Then you are the **Client** and the General Contractor is your **Project Leader**. The **Project** could go something like this:
    - You and the General Contractor draw up blueprints and sketches; pick out some floors, cabinets, counters, and appliances; and negotiate a price and schedule.
    - The Contractor's workmen (the **Team**) demolish the existing kitchen, dig into the walls, and start re-plumbing and rewiring the new kitchen. The workmen find mold in the walls, notify the Contractor, and the Contractor calls you in for a ***Review*** and potential **Re-Negotiation**.
    - It's not the **Team**'s fault that there's mold, so either the price goes up, or you cut some corners. You choose to use less expensive flooring, and the price and schedule remain unchanged.
    - The next day you go visit a friend and see some *real cool* appliances – not the ones you negotiated for – and you ask your Contractor about them. You do another ***Review*** of the kitchen and discuss how it could look. You both agree the new appliances would be great, and the **Cost** goes up $3000 and the **Schedule** pushes out 2 weeks because he needs to order the new Appliances from Germany.

    - A week later the Contractor calls and offers you a stunning deal on some granite counter-tops that he has left over from a Restaurant job. They're not exactly the color you want, so you go ***Review*** the kitchen and take a look at a sample of the counter-top. It's not what you *really* want, but you can't turn it down as it saves you $2000 and it's available to be installed *right now*.
    - Because of the new counter-tops, the tiles on the back-splash have to change as well as the overall paint scheme. These changes are no-cost, so you go for it…
    - Finally, you get a remodeled kitchen. It is about 20% different from the way you originally envisioned it, it cost $1000 more than you originally thought, and it's 2 weeks late. Believe it or not, though, you're really happy… you got the mold taken care of, and you were *in the loop* every step of the way.
2. Fixing your car that was in an accident, where the Insurance Company has simply given you a check for $8000 to fix it with.
3. Landscaping your new yard because all you got with the house is one little tree and some sod. You have a self-imposed budget of $20,000.

The **Project Leader (PL)** role represents the 'project management' part of a **Project Manager** - it does *not* include the 'people management' part of the **Project Manager**. The **Project Leader** role, as I describe it, is based on the appropriate parts of the PMI's PMBOK, which says that the **Project Leader** has two main responsibilities:

1. To ***Develop*** a **Project Plan**, and
2. To ***Update/Maintain*** the **Project Plan** throughout the **Project** so that it accurately reflects the **Reality** of the **Project** at all times.

And (to keep it simple) the **Project Plan** (the **Plan**) contains definitions and estimates for:

- **Cost**: The best, good-faith, estimate of what it will *cost* to ***Deliver*** an acceptable **Product**;
- **Scope**: The expected "content" of the **Product**'s **Value Backlog**; and
- **Schedule**: The best, good-faith, estimate of *when* the Team will ***Deliver*** an acceptable **Product**.

This often summarized by saying that the **Project Leader** makes sure that the **Plan** and **Reality** match. Using this formulation, we have:

- a *bad* **Project Leader** is one who tries to force **Reality** to match the **Plan**, and
- a *good* **Project Leader** is one who modifies the **Plan** to reflect **Reality**.

This **Pattern** requires *good* **Project Leaders**; that is, **Project Leaders** who update the **Project Plan** rather than try to change **Reality**. This is *not* easy, by any means. The following things are discussed and negotiated between the **Project Leader** and the **Client**:

- Contents of the **Project Plan**,
- Whether or not the current **Product Increment** is acceptable for ***Delivery***, and
- What it will take for a future **Product Increment** to be acceptable for ***Delivery***.

These negotiations often result in balancing trade-offs between the overall **Scope** versus the overall **Cost** and the overall **Schedule**, and these changed values are what is contained in the current **Project Plan**. If there is a contractual relationship between the **Project Leader** and the **Client** (with the **Project Leader** representing the **Contractor**), the *final* decision-maker about these trade-offs depends on the type of contract. Since the purpose of a contract is to *balance the risks* appropriately[3], this means:

- in a **Fixed-Price** contract, the decision-maker is the **Project Leader**, as it is the **Contractor** who is absorbing the risk, and
- in a **Time-and-Materials** contract, the decision-maker is the **Client**, as it is the **Client** who is absorbing the risk.

I will use the term **Project Leader (PL)** throughout this book to refer to the **Business Owner** who manages the Product's **Value Backlog** *and* is also accountable to develop and maintain the **Project Plan**. Remember that "accountable" means "can be held to account" or "must be able to explain the decisions involved" - it is *not* simply a blame-setting exercise.

Anyway, this **Project Leader** is a specific kind of **Business Owner** that we see…

[3]See the Federal Acquisition Regulation (FAR)

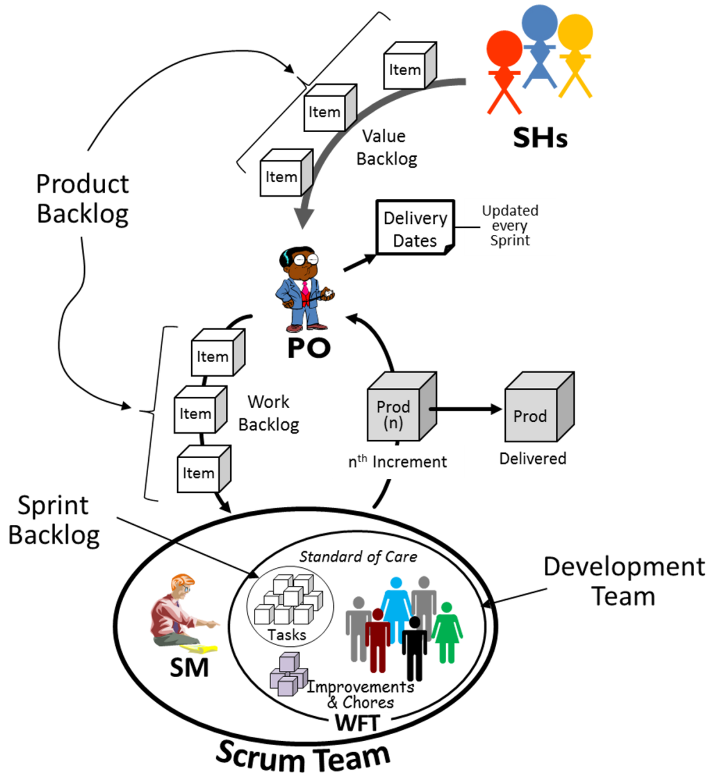
Original Scrum Discussion

We finally have all we need to discuss **Original Scrum**. As you can imagine, **Original Scrum** is a combination of all the Patterns we've seen so far, and is limited to a single, small, fixed-size, co-located (at least virtually) **Scrum Team** Developing and Maintaining a single **Product**.

Scrum has a very simple set of rules:

1) There are three Roles: the **Product Owner**, the **Scrum Master**, and the **Development Team**

- The **Product Owner** is a strange amalgam of the **Business Owner**, **Project Leader**, and **Client** (I'll discuss this later),
- The **Scrum Master** is the **Development Team**'s **Team Coach**, and
- The **Development Team** is a small (5-9 people), fixed-size, **Well-Formed Team** that will be doing the work.



Original Scrum

2) There are a small number of Practices:

- There is a **Sprint**, which is which is the length of the **Development Iteration**.
- There is a **Backlog** that consists of all **Item**s of work for the **Product**.

- The **Backlog** contains the **Stakeholder**s' **Value Backlog**, which is a list of **Item**s (in the **Stakeholder**'s language) that will provide value to the **Stakeholder**s, and that the **Stakeholder**s want the **Team** to do. Typically, these consist of **Feature**s (**Development** work) and **Defect**s (**Operations** work)
- The **Backlog** contains the **Team**'s **Work Backlog**, which consists of a list of the **Item**s that have been passed through (approved by) the **Product Owner** to be *actually* done by the **Team**. (*Just because the **Stakeholder**s want them does not mean the **Product Owner** wants the **Team** to do them.*)
- Each **Sprint** starts with **Sprint Planning**, which is when the **Development Team** and the **Product Owner** discuss and negotiate the content of the upcoming **Sprint**. There are two parts:
  - In Part 1 the **Product Owner** and the **Development Team** discuss what *should* be done, and the **Development Team** commits to which **Items** from the **Backlog** they will get **Done** in the **Sprint**
  - In Part 2 the **Development Team** decomposes these **Items** into **Tasks**, and these **Tasks** are collectively called the **Sprint Backlog**
- As the **Team** does work in the **Sprint**,
  - the **Development Team** has daily **Standup** meetings, which are a formalism of the **Well-Formed Team**'s frequent **Coordination Meetings**,
  - the **Development Team** produces a **Product Increment** by doing its **due diligence** and completing the committed-to **Items** to the appropriate **Standard of Care**, and
  - the **Development Team** does the **Improvements** and **Chores** they find necessary.
- At the completion of the **Sprint**,
  - the **Product Owner** ***Reviews*** the **Product Increment** and determines if it is acceptable for delivery. If the **Product Increment** *is* acceptable for delivery, then the **Team** cleans it up and delivers it. If it is *not* acceptable for delivery, the **Product Owner** may advertise predicted **Delivery Dates** to the **Stakeholder**s.
  - the **Scrum Master** has a **Retrospective** with the **Development Team**, which is a formalism of the **Reflections** the **Team Coach** has with the **Well-Formed Team**.

The Product Owner is Complicated

I mentioned that the **Product Owner** is an amalgam of the **Business Owner**, **Project Leader**, and **Client**. Here's why I said that:

- the **Product Owner** "owns and manages" the **Backlog**, and this makes him the **Business Owner**, as the **Product**'s **Value Backlog** is part of the **Backlog**, and the **Business Owner** manages the **Value Backlog**…
- the **Product Owner** ***Reviews*** the **Product Increment**. This makes him the **Client**, as it is the **Client** who ***Reviews*** the **Product Increment** to see if it is acceptable…

- the **Product Owner** predicts and advertises **Delivery Dates** and, because the **Team** is fixed-size, this determine both the **Schedule** and the **Cost**. Since the **PO** already manages the **Backlog** (which defines **Scope**), this means the **Product Owner** determines the **Project Plan** – which makes the **Product Owner** the **Project Leader**.

Here is the greatest quote I know about the **Product Owner**, from the *first* Scrum book, *Agile Software Development with Scrum*, Schwaber and Beedle, 2002, pg 34: "For the **Product Owner** to succeed, *everyone* in the organization has to respect his or her decisions. *No one* is allowed to tell the **Scrum Teams** to work from a different set of priorities, and **Scrum Teams** *aren't allowed* to listen to anyone who says otherwise." (emphases mine) This prioritization of the **Backlog** is the defining characteristic of the **Product Owner**, even though there are all the other responsibilities listed above.

Issues

The **Original Scrum** as defined here is a beautiful thing, and can be, and has been, used to great effect. However, there are *some* issues – see if you recognize them…

Value Backlog and Work Backlog "don't match"

A **Value Backlog** consists of **Items** that the **Stakeholder**s can discuss and understand, while the **Work Backlog** consists of **Items** that the **Development Team** understands and can implement. It is the **Product Owner**'s job to convert the **Stakeholder**s' **Value Backlog** into the **Team**'s **Work Backlog**, and they may be at different levels of abstraction, which can cause problems. For example, **Stakeholder**s may think about **Use Cases** or **Features**, while the **Development Team** needs to be able to focus on *small bits* of **Functionality**, like **Scenarios** or other **Items** defined by **Acceptance Tests**.

This issue will be addressed with the **Team Leader** Pattern.

Improvements and Chores

In a **Scrum Team** the **Development Team** is a **Well-Formed Team** that determines and works on its own **Improvements** and **Chores**. The time spent doing this is *not* managed from outside the **Team**. Many Organizations have problems with this, and want control over this work – and this can cause problems with the **Team**'s **Self-Organization**.

This issue will be addressed with the **Team Leader** Pattern.

Product Owner as Project Leader

The **Product Owner** (as **Business Owner**) Manages the **Backlog**, which defines the **Scope** of what the **Scrum Team** is working on. When the **Scrum Team** is doing Project work, the **Product Owner** (in the **Project Leader** role) can project **Delivery Dates**. Many **Product Owners** are *bad* **Project Leaders**, and try to force the **Development Team** to meet unrealistic **Delivery Dates**[4], rather than projecting **Delivery Dates** based on the **Reality** of the **Team**'s **Production Rate**. In fact, this is such a problem that many **Scrum Trainers** have said that one of the **Scrum Master**'s *primary* responsibilities is to *protect* the **Development Team** from these *bad* **Project Leaders** masquerading as **Product Owners**.

This issue will be addressed with the **Team Leader** Pattern.

Coercive Planning

In **Original Scrum** the **Development Team** *commits* to the work they will do in a **Sprint**, in spite of the fact they they don't know how much effort it will take to meet the appropriate **Standard of Care** or how much work they will need to spend on **Improvements** and **Chores**. How can this work? The **Development Team** is being set up to fail.

This issue is discussed in the **Forecast, don't Predict** Pattern.

Predictive Planning

In **Original Scrum** the contents of the whole **Sprint** are determined at the beginning, during **Sprint Planning**. This is a predictive process. How do the **Team** and **Product Owner** *know* what they will need to be doing 2 weeks from now? Maybe there will be a show-stopper bug that turns up… maybe the **Stakeholder**s will realize that they've changed their minds… I don't know. We would expect **Agile** methods to *avoid* making predictions, wouldn't we? We would expect the actual content of a **Sprint** to evolve throughout the **Sprint**, wouldn't we?

This issue is discussed in the **Forecast, don't Predict** Pattern.

The ScrumMaster often becomes a Manager

When the **Product Owner** is separated from the **Team**, the **Product Owner** often wants to hold *someone* on the **Team** accountable, or answerable, for what the **Team** is doing. If nothing else, the **Product Owner** often wants a **Point of Contact (POC)** on the **Team**, and this **POC** winds up with *de facto* **Management Authority**. So far, so good…

The problem is that this **POC** is often the **ScrumMaster**, who can be seen as "Second in Command" or as a *junior* (or *Proxy*) **Product Owner**. As we say in the **Team Coach** Pattern, this is dangerous, and could defeat the purpose of being a **Team Coach** - so don't do it!!

This issue will be addressed with the **Team Leader** Pattern.

[4]Ken Schwaber calls this the "Mother of all Problems" in his 2007 book: *The Enterprise and Scrum*, pg. 93.

Misunderstanding of the phrase “Potentially Shippable”

This issue doesn’t show up because of the **Pattern**s, but it has caused a lot of problems, so I’ll mention it. It has always been a rule of Scrum that the **Development Team** produce a **Potentially Shippable Increment** each Sprint, and some have read this to mean that they must ***Ship Product*** every Sprint. This has led to some seriously hacked-out **Code**, and a complete disregard of any reasonable **Standard of Care**. This is *exactly* the opposite of what is supposed to happen… the intention of of the **Potentially Shippable** idea is that it should *cause* the **Team** to follow an appropriate **Standard of Care**. Here is the quote[5]

“Scrum requires Teams to build an increment of product functionality every Sprint. This increment must be potentially shippable, because the Product Owner might choose to immediately implement the functionality. This requires that the increment consist of thoroughly tested, well-structured, and well-written code that has been built into an executable and that the user operation of the functionality is documented, either in Help files or in user documentation. This is the definition of a “done” increment.”

As you can see, this definition of a **Done Increment** (**Potentially Shippable**) is about Quality – the **Standard of Care** that must be followed. Unfortunately, many have seen it as required **Releasable** Code, which means Code that the User can actually use, which has led to hacking our *bad* Code that has a lot of features. The work that lives in the “Done but not yet actually Releasable” is often called **Undone** work; a simple example of this is when my son and I cleanup of the Garden and took the tools back to the Rental Place before actually “releasing” the Garden Beds to my wife.

Anyway, the entire point was missed by some people, and the idea was turned upside-down and inside-out. Such a shame…

This issue will be addressed with the **Definition of Done** Pattern.

Development Teams write Bad Code

There are two reasons **Development Team**s write Bad Code:

1. Most Developers don’t know how to write Good Code (or **Clean Code**) – they don’t understand their **Craft**.
2. A Team trying to meet unrealistic deadlines is often forced to write Bad Code – the “Mother of all Problems” I mentioned above.

What is Clean Code? In many (if not most) professions there is the notion of **due diligence** to meet a **Standard of Care**; not doing one’s **due diligence** is considered **malpractice** - and **malpractice** will get you kicked out of the profession.

Not so with software.

[5] *Agile Project Management with Scrum*, pg. 12.

There are no universal standards or requirements for software developers. We all want software to satisfy the '-ilities' (reliability, extensibility, maintainability, etc.), but there is no requirement for software developers to do their **due diligence** to make this happen. There are some standards that pertain to software in certain Industries or situations (FDA, FAA, Sarbanes-Oxley, etc.), but these standards are usually considered part of the **Acceptance Criteria** rather than as part of the **Standard of Care**. In software, the **Standard of Care** would be about how to write **Clean Code** (including analysis, design, code, test, etc.), not how to satisfy the requirements.

In software there are documented **Best Practices** about **Software Craftsmanship**, as we see in the eXtreme Programming (XP) practices, or in the books *Clean Code* by Bob Martin and *Working with Legacy Code* by Michael Feathers, and many others; but there is no *requirement* for software developers to either know, understand, or use them – there is no *requirement* to treat our vocation in a **Professional** way. This is unfortunate, and the reason that Scrum brought the **Well-Formed Team** pattern into software.

This issue will be addressed by the **Definition of Done** Pattern.

Project Management Pressures This is what Ken Schwaber calls the "Mother of All Problems"[6], where he talks about how **Legacy Code** *becomes* **Legacy Code**. The basic idea is simple:

- Trying to *make dates* causes the Team to take shortcuts,
- These shortcuts *must* compromise **Code Quality**, because there is no *wiggle room* in the other variables (**Cost**, **Scope**, and **Schedule**),
- As the **Quality** decreases, the shortcuts become more and more drastic to keep up with schedules, and
- The result is **Dead Code** (Code that can't be extended or maintained) in just a few **Sprints**…

This can *only* be handled by having a **Team** do its **due diligence** and having *good* **Project Leaders** – see the **Agile Project Leader** Pattern. If there are *bad* **Project Leaders** the only defense is the **Professionalism** of the **Team Members**.

This issue will be addressed by the **Definition of Done** Pattern and the separation of the **Project Leader** from the **Team**.

Basic Scaling Issues

And, finally, there are problems based on **Scaling**. There could be more than one **Product** involved, there could be more than one **Team** involved, or the **Team** could be in more than one (virtual) location. There's not a lot we can do about this last one – it seems to be intractable – but We'll address multiple **Team**s and multiple **Product**s issues later, when we talk about **Scaling**.

[6]*The Enterprise and Scrum*, pg. 93

Modern Scrum

Original Scrum works very well for mature **Teams** in good **Organization**s. Not surprisingly, less good otherwise. Since 2004 (or so) several improvements/changes have been made to **Scrum** to make it more useful. I will still assume there is a single **Team** working on a single **Product**, but I'll introduce the following **Pattern**s that improve things:

- Team Leader,
- Subject Matter Experts,
- Definition of Done, and
- Forecast, don't Commit (and others)

Pattern: Team Leader

Original Scrum is very powerful, but the fact that the **Product Owner** is *outside* the **Scrum Team** can cause problems. We solve these problems by having a **Team Leader**.

Problem:

One or more of the following Statements is true:

1. The **Business** wants to make sure the **Development Team**’s **Improvements** and **Chores** get prioritized “against” the **Items** coming from the **Business Owner**’s **Value Backlog** – the **Business** doesn’t want the **Development Team** making these decisions as part of their **Self Organization**.
2. The **Development Team** needs ongoing “tactical” advice – during the **Sprint** – about what is most important about the **Items** that are being worked on as part of the **Sprint Backlog**.
3. The **Items** being prioritized from the **Stakeholder**’s **Value Backlog** are not *well enough understood* for the **Team** to work on – they need to be ***Refined*** before they become appropriate for the **Team** to work on them.
4. The **Product Owner** needs a **Point of Contact** on the **Team** to hold **Accountable**, or answerable, for the **Team**’s actions.

Context:

There is a **Original Scrum Team** working on **Items** being prioritized by a **Product Owner** external to the **Team**.

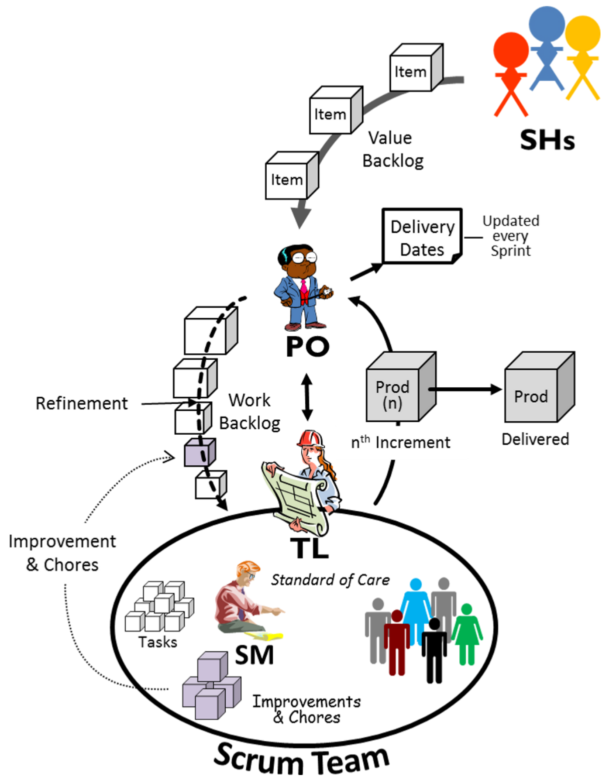
Solution:

Have a member of the **Scrum Team**, called the **Team Leader**, with the following **Accountabilities**:

- The **Team Leader** is **Accountable** to the **Product Owner** for the prioritization of *all* the work done by the **Scrum Team**, including **Improvements** and **Chores**,
- The **Team Leader** is **Accountable** for ***Refining*** the **Items** coming from the **Value Backlog** so that they are **Ready** for the **Team** to work on. Note that ***Refinement*** includes:
  - decomposing “big” **Items** into “smaller” **Items**,
  - extracting sub-**Items** from **Items**,
  - improving **Items** so they are better understood,
  - and so on…

You should be wary of making the **Team Leader** the same person as the **Team Coach**, because this could impinge upon the **Team Coach**'s "regulatory" responsibilities. This is not a universal truth, but certainly seems true for **Scrum Teams** doing **Software Development**.

Additionally, make sure it is understood that the whole **Scrum Team** is supposed to be a **Well-Formed Team** (not just the **Development Team**), so that **ScrumMastering** and **Product Ownership** responsibilities belong to the whole **Team** – the **Self-Organization** is not simply about **Development**.



The Team Leader

Discussion, including examples:

This is a common pattern, and I'll just give two simple examples:

1. In a Restaurant's Kitchen, we know that the Kitchen's **Value Backlog** consists of Food Orders provided by the WaitStaff. *Inside* the kitchen, however, the Sous Chef is running the show, and is accountable for "the kitchen's inventory, cleanliness, organization and the ongoing training of the entire staff"[7]. The Sous Chef prioritizes the work *inside* the kitchen; when to clean, train, and so on. These **Improvements** and **Chores** are added to the Kitchen's **Work Backlog** along with the orders coming in.

[7]See Wikipedia.

1. Workmen working on site have a Foreman who works as the **Team Leader**. The **Business Owner** assigns the Job to the **Team**, but the Foreman works with the **Team** to divide the Job into sub-Jobs and **Tasks**, as well as determine what **Chores** need to be done.

In many writings about **Scrum**, this **Team Leader** is referred to as the **Proxy Product Owner**.

Pattern: Subject Matter Experts

Scrum Teams are supposed to be **Self-Contained** – but many of them aren't.

Problem:

Many **Scrum Teams** aren't **Self-Contained**; they lack some skills they need.

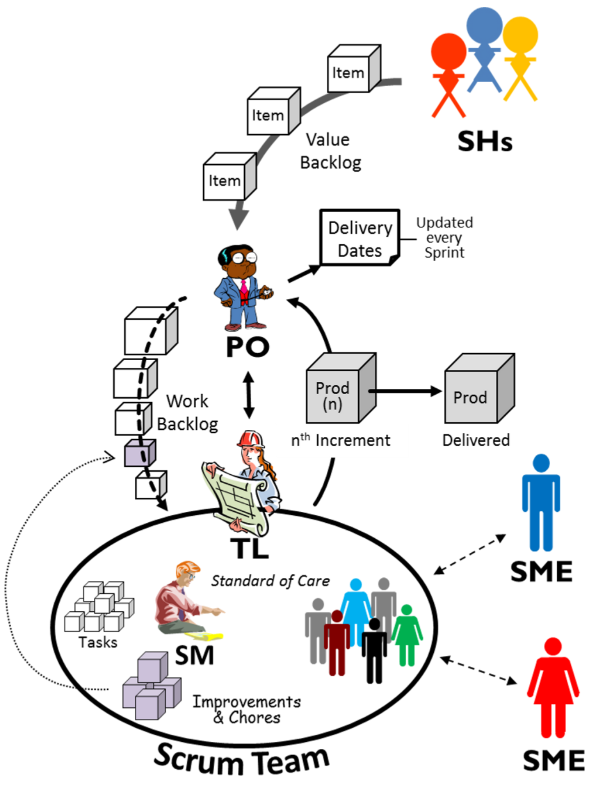
Context:

There is a **Scrum Team** working on **Item** from their **Work Backlog**. Some of the **Items** require expertise the **Team** does not have. This expertise could be:

- Technical Expertise the **Team** needs to complete its work,
- Business Expertise to better understand the **Item**'s **Acceptance Criteria**, or
- Expertise in ***Refining*** the **Value Backlog**'s **Item**s to be more appropriate for the **Team** to work on.

Solution:

Have **Subject Matter Experts (SMEs)** join the **Team**, as *honorary* **Team Members**, when the **Team** needs to use their expertise.



Subject Matter Experts

Discussion, including examples:

These **SMEs** come in many forms: they could be **Stakeholders** that are providing business-knowledge help on specific **Items**; they could be technical **SME**s, like Architects, Usability experts, Business Analysts, or Technical Writers; they could be helping the **Team** working on **Improvements** or **Chores**, they can be shared across several **Scrum Teams**, they can be just about anything.

Normally, the need for a **SME** is determined by the **Development Team** and the **ScrumMaster**, and the **ScrumMaster** works with the **Team Leader** and **Product Owner** to find and acquire the **SME**.

Subject Matter Experts are usually with the **Scrum Team** for a short period of time, and are often thought of as being extensions of the **Team**, I have heard of the **Scrum Team**, along with its **SME**s, referred to as the **Sprint Team** – indicating the fact that they are part of the **Team** for the duration of the **Sprint**.

Often, it is a good idea to use **SME**s as **Mentor**s, rather than having them do the work themselves. In this way, the **Scrum Team Members** become less reliant on them, which decreases the "lottery metric"[8] for the **Team**. My favorite pattern for using **SME**s is the **Buddy Up** Pattern, which identifies an existing **Team Member** to be the **SME**'s **Buddy**. This **Buddy** is **Accountable** to the **Team** for work the **Team** needs the **SME** to do, or help do. The **SME** and the **Buddy** work together, with the **SME** in

[8]The number of people who have to "win the lottery" before your **Organization** is totally screwed.

a mentor role, in order to do the work. This both gets the work done, and makes the **Buddy** smarter, decreasing future need for the **SME**.

Pattern: Definition of Done

Well-Formed Teams are supposed to have a **Standard of Care** that they follow, because they are **Professionals**. Unfortunately, in software, most don't seem to…

Problem:

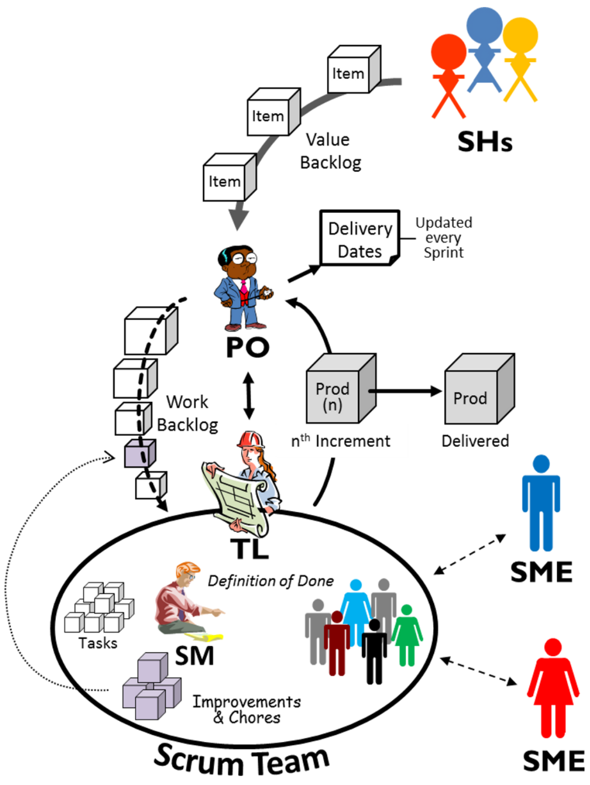
The **Scrum Team Members** don't use, or don't know, an appropriate **Standard of Care** in their work.

Context:

You have a **Scrum Team** doing work, and the work they are doing is not meeting an appropriate **Standard of Care**.

Solution:

Attach a **Definition of Done** to each **Item** (augmenting the **Item**'s **Acceptance Criteria**) that assures that the **Item** is developed with an appropriate **Standard of Care**. The **ScrumMaster** works with the **Team** to assure that the **Item** meets this **Definition of Done** as the **Item** is being worked on.

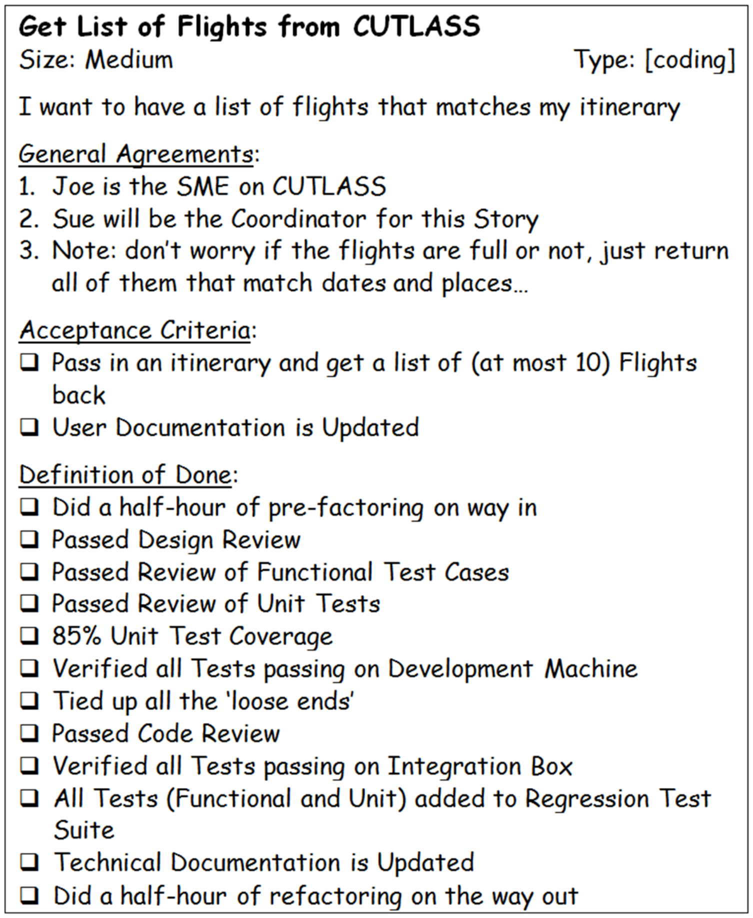


Definition of Done

Discussion, including examples:

Well-Formed Teams are supposed to be **Professional**s, and come prepared with their own **Standard of Care** that they do their **due diligence** to meet *every* time they do work. The **Stakeholders** are relying on this when they use a **Scrum Team**. In software, an example of this is a **Development Team** that diligently uses the **XP Technical Practices** and see themselves as **Professional Software Craftsmen** when they develop their Code.

Often, the **Team Member**s don't have an appropriate **Standard of Care** – they've never been taught, they've never learned, they've never used, they've never seen such a thing. So, we often have the **ScrumMaster** (in the role of **Technical Coach**) help the **Team** develop, and continuously improve, their own **Standard of Care**. In the Scrum Community, this is called "*having* a **Definition of Done**". Here is an example of an **Item** with both **Acceptance Criteria** and **Definition of Done** for an **Item** involving an Airline website.


Get List of Flights from CUTLASS
Size: Medium Type: [coding]

I want to have a list of flights that matches my itinerary

General Agreements:
1. Joe is the SME on CUTLASS
2. Sue will be the Coordinator for this Story
3. Note: don't worry if the flights are full or not, just return all of them that match dates and places...

Acceptance Criteria:
❑ Pass in an itinerary and get a list of (at most 10) Flights back
❑ User Documentation is Updated

Definition of Done:
❑ Did a half-hour of pre-factoring on way in
❑ Passed Design Review
❑ Passed Review of Functional Test Cases
❑ Passed Review of Unit Tests
❑ 85% Unit Test Coverage
❑ Verified all Tests passing on Development Machine
❑ Tied up all the 'loose ends'
❑ Passed Code Review
❑ Verified all Tests passing on Integration Box
❑ All Tests (Functional and Unit) added to Regression Test Suite
❑ Technical Documentation is Updated
❑ Did a half-hour of refactoring on the way out

Definition of Done Example

When there is more than one **Team** working in the same Codebase, they each *should* use the same **Definition of Done**, as they will be working with each others' Code. Each **Team** needs to trust the other **Team**s' Code as much as they trust their own – they need to know that the other **Team**s used their **due diligence** and met the same **Definition of Done** as they did.

Usually, each **Item** has its own **Definition of Done**, but some **Teams** have applied the concept to the **Product Increment**, instead. Some **Teams** have used a common **Definition of Done** for *all* **Items**, others have custom **Definitions of Done** for each **Item**. The **Team** could use different **Definitions of Done** for different kinds of **Items** (this is called **Storyotyping**[9]. It really doesn't matter, as long as everybody trusts that the Results are as "good" as they need to be, *all* the time.

Sometimes the **Team** can't meet its **Definition of Done**, either intentionally or unintentionally. When they do this we say that they have **Unfinished Work**. This work may need to be ***Delivered*** or ***Reviewed*** for perfectly valid business reasons (Trade Show, big Client, whatever), but the **Team** may *not* pretend that the Work was **Done** – the **Team Members** may *not* pretend that the Work met an appropriate **Standard of Care**. When this kind of stuff happens, they need to add a **Cleanup Item** – one that promises to clean up, or finish the work – to the **Backlog**, so that nobody will forget it; so that the **Team** will get the work **Done** as soon as possible.

Sometimes there is a delta between being **Done** and being **Releasable** – delta is called **Undone** work. This is not a bad thing, but deciding what to leave **Undone** is a decision that needs to be made. For example, deciding to leave off polishing the User Documentation until the end may be a good decision, but deciding to delay collecting the information for the user documentation until the end probably isn't.

The **Definition of Done** can be extended to be a robust Standard, including standardizing architectural and design patterns, providing necessary reviews and inspections, and so on. It is something *worth* standardizing, in my opinion, as it provides an anchor for the **Teams** to **Self-Organize** around – "this is what we've got to make it look like, how do we do that?"

Because not all **ScrumMasters** are technical, this could Lead to a **Technical Owner** or **Coach** role on a **Team**, who would be a **Team Member** who was **Accountable** for the existence of, training on, and use of, an appropriate **Definition of Done**.

[9]See chapter 3.10 of *Exploring Scrum: the Fundamentals*.

Pattern: Forecast, don't Commit (and others)

There have been many changes in the way a **Scrum Team** plans its **Sprints**

Problem:

The **Sprint Planning** meeting in **Original Scrum** is both Coercive and Predictive, and each of these is a bad thing, as is discussed in the discussion of **Original Scrum**.

Context:

You have a **Scrum Team** doing work, and you want to change the **Sprint Planning** so that it is less coercive and/or predictive.

Solutions:

Each of these problems, as well as other planning problems, have been addressed in **Modern Scrum**; in fact, there have been several improvements in **Sprint Planning** that we often see:

1. the notion of "committing" to a **Sprint Backlog** has been removed. Currently, **Sprint Planning** produces a **Forecast** of what *might* get done, and this is called the **Sprint Backlog**. This makes **Sprint Planning** less Coercive, but may not make it less Predictive – the **Team Members** are still predicting what they will be doing later on in the **Sprint**.
2. The original '2-pass' **Sprint Planning** has been largely replaced by '1-pass' **Sprint Planning**, which brings *current reality* into the planning, making the planning, itself, more agile and correct[10].

1. It is now recommended that **Sprint Planning** does not *fill up* the **Sprint** – planning is now somewhat *malleable*, making it less predictive. In the more recent **Scrum Guides**, Ken and Jeff have stated that the initial **Sprint Backlog** (the result of **Sprint Planning**) only fills up about 80% of the effort available in the **Sprint**, with remaining **Sprint Backlog Items** arising throughout the **Sprint**.
2. There is a move to doing a more kanban-ish version of **Sprint Planning**, where the **Sprint Planning** meeting only selects the **Sprint Backlog Item**s that will be started *right now*, and additional **Item**s will be brought into the **Sprint** as existing **Item**s get **Done**[11].

[10]See the "Agreement-Based Planning" chapter of *Exploring Scrum: the Fundamentals*.

[11]See the "Kanban(ish) Variant" chapter of *Exploring Scrum: the Fundamentals.*

Discussion, including examples:

Over the years, there was a realization that the **Sprint** is a feedback cycle, not a planning cycle or a work cycle. Along with this realization came the idea that **Sprint Planning** was not the big deal everybody thought it was. People realized that, logically, **Sprint Planning** is not actually necessary; if there is a prioritized **Work Backlog**, and each **Item** on the **Work Backlog** has a well-defined definition of **Done** (the combination of the **Acceptance Criteria** and the **Definition of Done**), then the **Scrum Team** just has to work its way through the **Work Backlog**, in order, and ***Review*** whatever is actually **Done** at the **Product Review**.

This is a tremendous refutation of the predictive planning we often see in non-**Agile** development. This realization led to *all sorts* of innovations in **Sprint Planning** – including what is listed above. My personal recommendation is that a **Team** should do kanban-ish planning once the **Team** has become *mentally* agile. This type of planning is neither Coercive nor Predictive, seems to be the *right way to go* in an Agile Development.

Modern Scrum Discussion

So, I finally have what I need to describe **Modern Scrum**, which is basically **Original Scrum** with the modifications described in this Chapter:

- The addition of a **Team Leader**, the **Team Member** who is accountable to the **Product Owner** for the value of the work performed;
- Making the whole **Team** (not just the **Development Team**) into a **Well-Formed Team**, and adding the responsibility to **Refine** the **Items** on the **Work Backlog** in order to make them **Ready** to work on;
- Inserting the **Scrum Team**'s **Improvements** and **Chores** directly into the **Work Backlog**, where they are **Refine**d and prioritized along with the other **Items** derived from the **Value Backlog**;
- The addition of **Subject Matter Experts** who provide knowledge and expertise that the **Team** needs, but does not have;
- Using the **Definition of Done** to impose the use of an appropriate **Standard of Care** by the **Team**; and
- Changes to **Sprint Planning** to make it less Predictive and Coercive.

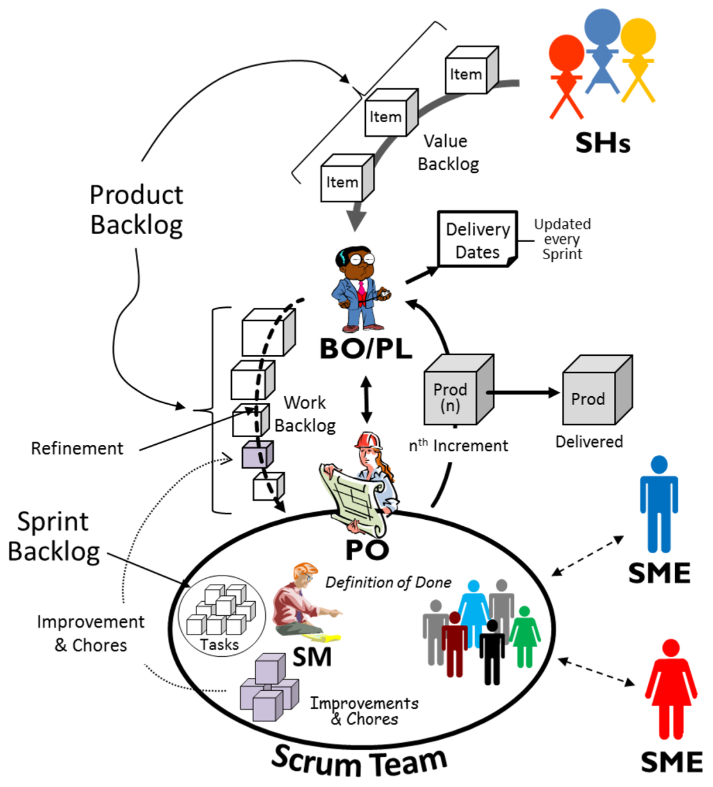
Product Owner Re-Definition

But, there's one more thing; and it's a *big* thing – we discussed (and argued) about this one a lot in the **Scrum Community**. This **Team Leader** role was called the **Proxy Product Owner** for a while, and then it was decided that, no, this **Team Leader** is *actually* the **Product Owner** – that the **Product Owner** had to be a **Scrum Team Member**; a part of the **Self-Organized Team** – and this decision was made in about 2005 or so.

Some are still fighting about this, but I agree with it whole-heartedly, and it is clearly assumed to be true in *The Enterprise and Scrum*, which was published in 2007. For example:

- on page 73: "The Product Owner and ScrumMaster are the first people on a **Scrum Team**."
- on pages 76-80 there are diagrams (figures 8-6 and 8-7) showing hierarchies of **Scrum Teams** with **Product Owner**s and **ScrumMaster**s at each level. In fact, the definition of the Product Owner in this book (pg. 114) is team-focused: The Product Owner is "the person who is responsible for what the Scrum Team builds and for optimizing the value of it."

So, it's a *done deal*, as far as the "official" definition of **Scrum** is concerned. But it causes us problems here, because the **Product Owner** role in **Original Scrum** is *not* the **Product Owner** role in **Modern Scrum**. Oops… so I'm going to go back to calling the original role either the **Business Owner** or the **Project Leader** depending on whether or not the **Stakeholder**'s **Value Backlog** is project-based – I tend to think of a **Project Leader** as the **Business Owner** of a **Project**. Anyway, here's my picture of **Modern Scrum**.



Modern Scrum

Now, one could argue, and many have, that the **Business Owner** and the **Product Owner** could be one-and-the-same person. I agree, in principle. However, I *don't* want the **Product Owner** to be the **Project Leader** – the **Development Team** *must* be separated from the potential of a *bad* **Project Leader** – a *bad* **Project Leader** can have a very coercive and damaging effect on the **Team**, and thus the **Product** itself.

Problems with Modern Scrum

As has been discussed throughout, **Modern Scrum** solves many of the problems that existed in **Original Scrum**. In fact, I see **Modern Scrum** as a well-oiled machine – as a balanced collection of **Patterns**. It is a *very good thing*. However, there are three issues that still remain:

1. the **Project Leader** (if there is one) must be a *good* **Project Leader**. There is *no way* we could guarantee this, but at least we have both the **Product Owner** and **ScrumMaster** protecting and shielding the **Team** from this potential *bad* **Project Leader** – as long as we keep the **Project Leader** *outside* the **Team**.
2. **Modern Scrum** still assumes a single **Product**; and

3. **Modern Scrum** still assumes a single **Scrum Team**.

These last two Issues are Scaling Issues, which I will discuss in the next section.