# Managing the Work in an Agile Project

Dan Rawsthorne, PhD, CSM
Senior Consultant, Net Objectives

**Abstract**: This article introduces the ideas of a functional Work Breakdown Structure (WBS), Business Value (BV), Earned Business Value (EBV), and Earned Business Value Index (EBVI), as they pertain to agile software development projects. In order to contrast their use with "standard" agile project management, the well-known Work Backlog (WB) is also defined and discussed. This article is written for project managers, analysts and others who need to manage an agile project. It shows how introducing the functional WBS and EBVI into an agile project can provide visibility and control for the project manager and customer team.

**Contents**:

## *Introduction*

In the last decade it has become apparent that agility works, whatever that may mean. However, most software projects remain artifact-driven and waterfallish. Why is this? The most common reasons seem to be that agility is perceived as too developer-centric, too light-weight, or that feedback to business is hard to understand. In other words, there is not enough formality, especially concerning communication to management.

The essence of agility is validation and feedback, and the overall object of an agile project is to incrementally develop software with attention to changing priorities, requirements, and other realities of development[1].

This incrementalism allows for success, but also provides challenges. At the personal level, agility requires discipline, honesty, focus, and the ability to communicate; which are things we don't discuss in this article. At the team level, one of the major problems is communication of progress to management, which is the focus of this article.

In order to discuss progress we must discuss the work that the team does in order to develop a product. By discussing work I can touch on most of the management issues I'm interested in for this article – and the discussion is easily understandable (I hope).

[1] Two popular methods that accomplish this incrementalism in a low-ceremony way are XP and Scrum. In this article we borrow heavily from both; many of the terms are from XP, while much of the philosophy is from Scrum.

## *Two Views of Work*

There are two fundamental perspectives of the work being done on a project: from the developer's point of view, and from the business' point of view.

The developer is looking at the project from the inside, and is interested in the work as it is being done. The developer actually does the work, needs to decompose and estimate the work, and is fundamentally interested in the questions: "What am I working on now?" and "How much work can I get done?"

The business (management) is looking at the project from the outside. It supplies the overall requirements and prioritizations, and is interested in the questions: "How much have we got done?" and "How much more do we need to do?"

I will discuss these two ways of looking at the work in the next few sections, but first we need the fundamental units of work: stories and tasks.

## *Stories and Tasks*

A *story* is the fundamental unit of work that is visible from both the inside and the outside of the project. Stories have value, and are used as the interface between the business and developer sides of the project. They need only have three things:

- A description, usually in business terms;
- A size, for rough estimation purposes (I like to use the values Tiny, Small, Medium, Large, Big, and Unknown); and
- A short description of how the story will be validated.

A *task* is smaller than a story, and is the unit of work that is actually worked on by a developer. They are manageable, doable, and trackable units of work. Stories are decomposed into tasks[2], which have the following attributes:

- A description of the work to be performed, in either technical or business terms;
- An estimate of how much time the work will take (I like to use hours or half-days);
- An owner, who may or may not be pre-assigned
- Exit criteria and verification method (test or inspection), and an indication of who will be responsible for the verification. This may be omitted in some cases, such as for development tasks that are developed in pairs using TDD. The reason for this is that the verification is implicit in the process for such tasks. However, the concept is sound – all tasks must be verified, not just "done".

There is a special type of story or task, called the *orphan*. An orphan is a story or task whose value is not business value; the business wouldn't pay for it to be done unless it is required for completion of a story that does have business value. These are called orphans because they need to be adopted by a *parent* with business value in order to get done.

In the following sections these fundamental units of work are organized and discussed in two different ways, the Work Backlog and the Work Breakdown Structure.

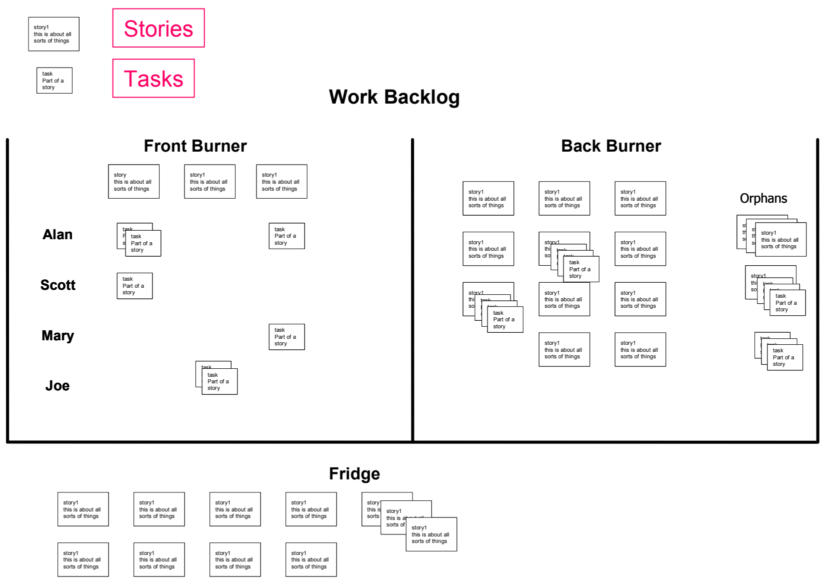
[2] If a story is already "right sized" it can be converted to a task by simply adding the appropriate attributes.

## *Work Backlog*

The first way to look at work is common to both XP and Scrum. We call this the *work backlog*[3], which is defined (in Scrum) as "an evolving, prioritized queue of business and technical functionality that needs to be developed into a system." In practice, the work backlog is a collection of stories and tasks organized by development priorities. I like to use the following three priorities:

- *Front Burner* – those Stories and Tasks that I am actually working on **right now**; they are in the current iteration.
- *Back Burner* – Stories and Tasks that I intend to work on **soon**. I have probably tentatively assigned them to the next iteration, and I allow myself to think about them.
- *Fridge* – Stories and Tasks that I might do **someday**. They're on the list, but I'm not actively thinking about them right now.

I assume that the readers of this article are familiar with this concept, but here is a depiction of a work backlog as an XP-like storyboard, with stories and tasks displayed on a wall.



The work backlog is a common concept, so rather than belabor it; here is a short list of its attributes:

- It is developer focused, as it shows the stories and tasks that the developer is working on, or will be working on;
- It is focused on the order of development, emphasizing the stories and tasks in the current iteration;
- The primary metric that is used with the work backlog is *velocity*, which measures (using a variety of metrics) how many stories or tasks the team develops per iteration.

[3] Called the "product backlog" in Scrum, it is also represented by the project's "stack of cards" in XP.

In both XP and Scrum, the work backlog is the primary tool used by the development team to present and manage the work. There are a variety of metrics and information radiators[4] that are used, but none of them really satisfy all of the needs of management. To make up the difference we use a functional Work Breakdown Structure.

## *Work Breakdown Structure*

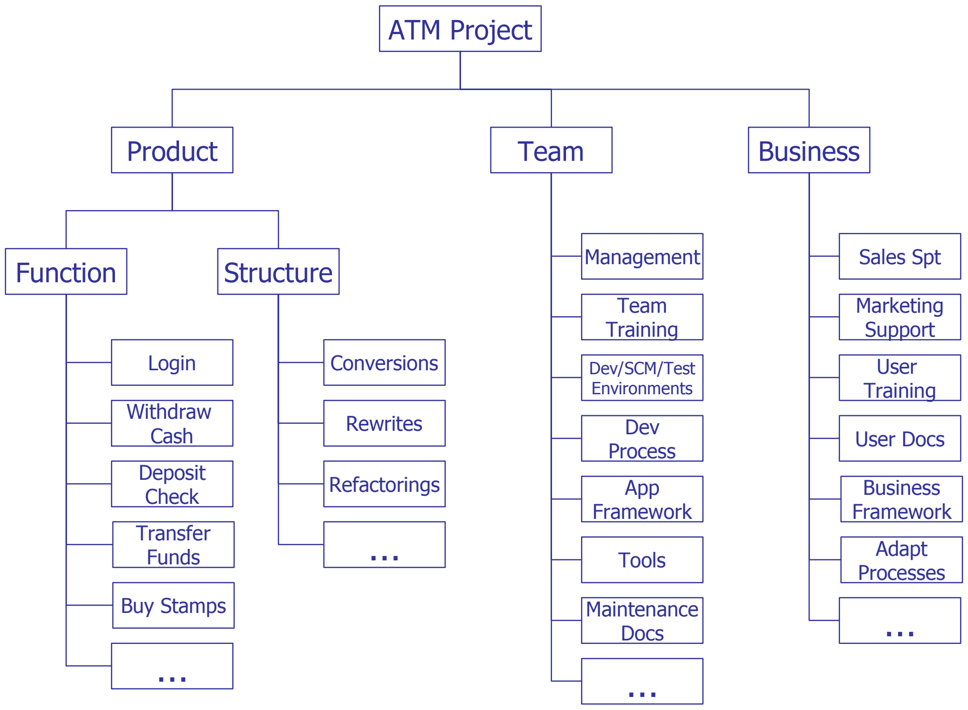
Management needs a view of the work that shows the "big picture" rather than a focus on the order of the work being produced. Luckily, standard project management has such a structure already invented for us – the Work Breakdown Structure (WBS), defined as "A deliverable-oriented grouping of project elements which organizes and defines the total scope of the project." (source: PMI:PMBOK)

The WBS that I use organizes the work into three main areas:

- *Product* – work that is directed to actually producing the software product;
- *Team* – work that allows/enables the team to be able to produce the product; and
- *Business* – work that allows the business to market, sell, or deliver the product to users.

Even within this overall structure there are many variations that are project dependent. The structure of the WBS (what rollup buckets there are) is a business matter. The stories and tasks that live at "the bottom" of these buckets are the ones that appear in the work backlog.

I like to organize the functionality of the system around use cases; another common way is to use features. In any case, a software project's WBS should look something like the following, representing a fictitious ATM development project.



[4] An information radiator is something (anything) that displays useful information in a place where passersby can see it.

As we shall see, this WBS has many uses that are important to the business. For now, though, let's discuss the features we find in the "Function" leg of the WBS. In this simple discussion I will assume that these features take the form of Use Cases.

## *Use Cases as WBS "buckets"*

Use Cases[5] represent functionality that is to be delivered, and are often complex. Because of the complexity, it usually takes a number of stories to deliver the functionality of a use case. In fact, since there is always "something else" to do, the use case is never complete; we actually just deliver as many of these stories as seem reasonable, and we plan for them during the planning game.

Because of the inherent complexity of a use case, we would expect there to be many different strategies for developing and assigning stories to them. There are, but I'm only going to present on of them by looking at the "Login" use case.

This strategy is used when we are using use cases as our unit of analysis and requirements. It provides a comprehensive list of stories, including stories for analysis, design, development, and test. This is useful when you have a heterogeneous team consisting of analysts, developers, testers, and the like. I refer to this as the "whole team" strategy, and is what I usually see when I work with a Scrum (as opposed to an XP) team. Note that these stories are based on analysis and development of the scenarios of the use case and that they have their own tasks (or, possibly, sub-stories, if the tasks are 'big enough').

In our list for Login we have five stories identified, and one (the main success scenario) has been further decomposed into sub-stories/tasks.

| "Whole Team" Stories and sub-stories/tasks |
|---|
| • Gather Stakeholder Concerns and determine Postconditions |
| • Determine MSS of Login |
| • Code up Main Success Scenario |
| o Unfold to software elements (validate with architects) |
| o Code up Login UI |
| o Update database with Password info |
| o Write Login Module |
| o Develop functional tests |
| o Integrate and test |
| • Analyze primary business extensions |
| • Code up "3 strikes and You're Out" business rule |

As you can see, there are many different kinds of stories using this strategy, such as:

- Analysis Stories - done in order to produce other (analysis/development) stories,
- Development Stories – designing and producing code,
- Test Development Stories – producing the functional tests that will validate the code, and
- Integration and Test Stories – actually producing the testing the product.

[5] A *Use Case* represents a collection of interactions users can have with the system in order to achieve a goal. As an analytic artifact it focuses on the goal; as a driver for development, it is viewed as a collection of scenarios, or testable interactions.

There can be other types of stories, as well. The basic idea is that if the team is doing work that is producing product, then that work is in here somewhere. In any case, the goal of the WBS Functional leg is to provide high-level descriptions of functionality (in this case, use cases) that have stories assigned to them. Each story has its own value, and the WBS "bucket" structure is used to organize them in an appropriate way to be viewed at the 50,000ft level. This provides the organization that management usually needs, but is not apparent from the Work Backlog.

## *Business Value*

By definition, all stories have value, but as noted before, business value is what management is willing to pay for. This implies that we need some (not necessarily perfect) way to measure business value of a story, in order to help us prioritize them.

We do this in a straightforward way, using the Work Breakdown Structure, and actually provide the business value of any WBS bucket, including stories – the ones at the bottom.

The strategy is simple:

1. Determine the Business Value of the Project. Some organizations will do formal ROI (Return on Investment) or PNL (Profit and Loss) calculations to provide a current dollar value; others will be less formal and apply some other form of additive value, like "this project is a '3' – that one is a '1'." In any case, we need some form of additive[6], numerical value, represented by BV(Project)
2. Calculate a Business Value Index (BVI) for each WBS bucket. The BVI represents the percentage of the total business value of the project that this bucket contains. The method of calculating the BVI is presented below.
3. Calculate the business value of a bucket by simply multiplying the two values, that is, *BV(bucket*) = *BV(Project)* x *BVI(Bucket)*.

### *Calculating the BVI*

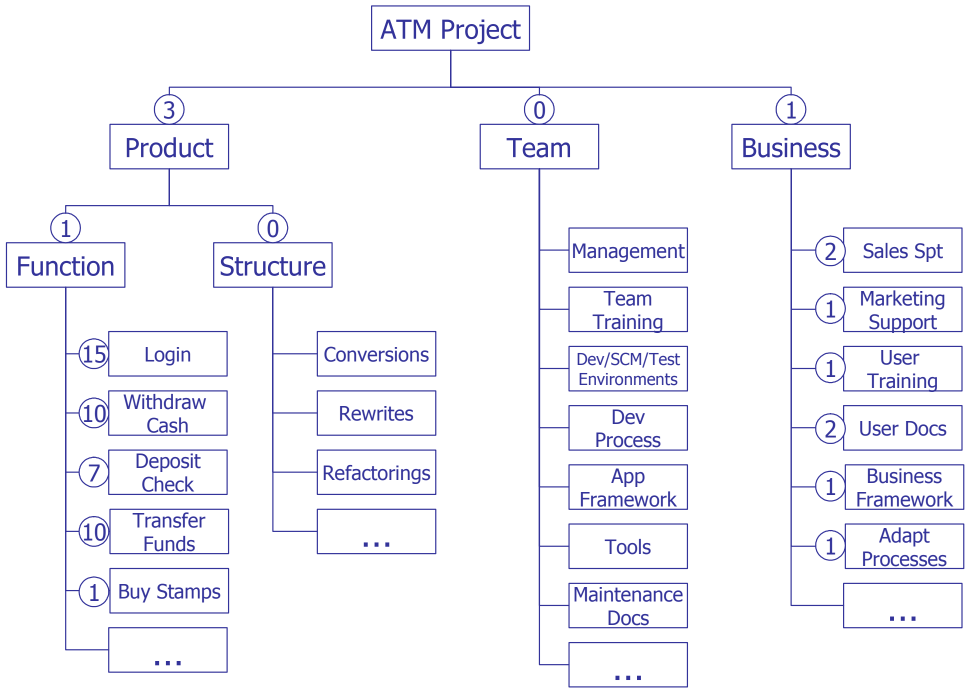
First, we assign additive weights to the legs of the WBS. Even though all stories have value, only some of them have business value – the rest are orphans, as we discussed earlier. In the simple view in this article, we only get business value by either putting functionality in the code, or by doing something that lets the business market or sell the product – there is no business value to the Team leg of the WBS.

I'm using relative weights on the WBS legs here, and this WBS is telling the following story:

- Our Product Leg is worth 3 times as much as our Business Leg, and our Team leg provides no business value;
- "Login" is the most important use case, worth 1.5 times as much as "Withdraw Cash", and so on; and
- User Documentation is the most important business category, but is still not worth very much compared to functionality in the code – it's worth approximately as much as the "Buy Stamps" use case.

[6] Additive values are ones that "add up" – that is; if something has twice the value, it's worth twice as much; if A has value '1', B has value '1', and C has value '2', then A and B together are worth as much as C (in a business sense).

Our WBS (with weights) could look something like this[7]…



After we've assigned weights to the WBS legs, we further assign additive weights to the Stories within each WBS "bucket" (we don't need to assign weights for stories in buckets that themselves have no weight, of course). For example, for the "Whole Team" Stories we find in the "Login" use case we could have the following weights:

Login Use Case (wt: 15)

- Gather Stakeholder Concerns and determine Postconditions (wt: 0)
- Determine Main Success Scenario (wt: 0)
- Code up Main Success Scenario (wt: 10)
- Analyze primary business extensions (wt: 0)
- Code up "3 strikes and You're Out" (wt: 3)

Note that the analysis stories have no weight; they only exist because we need them to provide the development stories that have business value.

Given this assignment of weights, we can now calculate the Business Value Index (BVI) of any WBS bucket, including Stories – the ones at the bottom. The calculation is recursive, as we move down the WBS tree. The value for the whole thing is 1 – getting everything done gives you 100% of the business value, and then we have the formula:

[7] It is up to the business, or customer, to assign the weights to the WBS legs and stories. This is not a technical matter. There is no guaranteed formula. For example, is "Team Training" something with business value or not? I don't know, it's up to your project – typically it doesn't provide value if paid for out of project funds, but might if paid for out of overhead (training) funds. The only thing required is that the weights be additive, as described before.

$$BVI(bucket) = BVI(parent) \times \left[ \frac{wt(bucket)}{wt(bucket) + \sum_{siblings} wt(sibling)} \right]$$

The BVI as calculated this way has some interesting properties:

- Adding a new use case "waters down" the business value of the other use cases.
- Adding a new scenario for a use case waters down the value of the other scenarios.

I claim that this is the behavior you want to have in an Business Value, and shows how finding our more stuff your system "should do" decreases the value of what you already know it should do.

What does this really mean? Let's do an example, and calculate how much the "Login" Main Success Scenario is worth. According to the last table, the functional leg is worth ¾ of the total value of the project; the Login use case is worth 15/43 of the functional leg; and the MSS is worth 10/43 of the total value of the Login.

Therefore, the BVI of the Login MSS is (3/4)x(1/1)x(15/43)x(10/13) ≈ 20% of the total. If we were to add another use case with a weight of 10 (say "Account Report"), this would drop to 16%.

## ***The Planning Game***

So, we now have a way to calculate a Business Value for a story[8]. What can we do with it? Well, we use it to help with the prioritization of stories that Customer has to do. This is appropriate as the BV itself is determined by weights assigned by the Customer, so the BV represents some reasonable approximation of what the Customer believes.

But first, let's have a high-level description of the planning game, the session that determines which stories will be worked on for the next iteration. It's actually pretty simple, and has three main steps:

1. The Customer presents a collection of stories to be investigated for work in the iteration. They are usually stories left over from the Front Burner as well as stories from the Back Burner. There may also be some stories that were promoted from the Fridge or just "showed up", as well. But this is not the whole collection…
2. The Developers analyze these stories. This includes a number of things:
   - Decompose the stories into tasks,
   - Have stories adopt orphan tasks (as necessary),
   - Determine dependencies between stories and add new stories, if necessary, and
   - Estimate the tasks.

   The idea is that the Customer's proposed stories have been refined into a (more complete) collection of stories and estimated tasks that the developers believe will accomplish the Customer's needs for the iteration.

[8] If the business has not assigned a *BV(Project)* we just use *BV(Project) = 1* in order to get relative values for prioritization. In essence, we just use the BVI for our prioritization.

3. The Customer chooses which stories to actually work on, based on the velocity of the team.

There are two steps (1. and 3.) when the Customer is prioritizing stories. This is when the BV is used. Unfortunately, it's not completely straightforward…

This is because prioritization decisions should not be done purely on the basis of benefit (BV), but on the basis of cost-benefit analysis. In other words, we need to calculate the ratio benefit/cost, and prioritize based on this value.

Luckily we can do this. We know that the benefit of a story is *BV(story)*, so we merely need to be able to calculate a cost:

- In step 1. we have a size of the story, which represents its cost. All we need is to convert the sizes into numbers, and do the calculations. I recommend the values: Small = 1, Medium = 2, and Large = 4. For Stories with other size values we must either combine them (if size is Tiny) or decompose them with an analysis story. So, benefit/cost for a story is calculated by

$$BV(story) \big/ size(story)$$

- In step 3. the developers have already decomposed the stories into tasks and estimated them (and added orphans as necessary). So, we can calculate the cost by adding up the estimates (in hours) of the tasks in that story. The BV of the story hasn't changed, so the benefit/cost is calculated by

$$BV(story) \Big/ \sum_{subtasks} estimate(task)$$

These calculations can't replace the Customer's judgment, but they might help with the (subjective) evaluations that need to be made during the planning game.

## *Earned Business Value*

As I stated at the beginning of this article, management is interested in the question: "How much have we got done?" This is an interesting question, as it leads to the return question: "Got done of what?" to which I say "Business Value, of course!"

Since I know how to calculate Business Value (BV), I *can* measure Earned Business Value (EBV), which I define as "the percentage of the known business value that is coded up and running." In other words, we add up the business values for all those stories that have been completed – that have *earned* their business value.

$$EBV(Project) = \sum_{completed} BV(Story) = BV(Project) \times \left( \sum_{completed} BVI(Story) \right)$$

There are two parts to this calculation:

- A *subjective* part that determines the weights allowing us to determine the BVI, and
- An *objective* part that determines whether or not a story is "done".

We already know about the first part, and the second is actually pretty simple. Once we have our planning game done, we have tasks defined for our stories. The simple philosophy is that a story is complete (has *earned* its BV) once all its tasks are completed and it meets its validation criteria (its tests pass, for example).

So, for our Login use case, we have tasks for the "Code up Main Success Scenario" story. Once they are all done, and code passes its tests, we earn the value for the story. Once again, note that the analysis stories have no weight; they only exist because we need them to provide the development stories that have business value. Also note that a story earns its business value once all of its tasks have been completed and verified, and it's passed its own validation criteria. In most cases this means that functional tests have been developed and run successfully[9].

Login Use Case (wt: 15)

- Gather Stakeholder Concerns and determine Postconditions (wt: 0)
- Determine Main Success Scenario (wt: 0)
- Code up Main Success Scenario (wt: 10)
  - Unfold to software elements
  - Code up Login UI
  - Update database with Password info
  - Write Login Module
  - Develop functional tests
  - Integrate and test
- Analyze primary business extensions (wt: 0)
- Code up "3 strikes and You're Out" (wt: 3)

All tasks must be successfully completed, and tests pass for the story to earn its value

The formula for calculating the *EBV(Project)* is pretty simple, as shown above. We can also calculate the EBV for any bucket in the WBS by using the formula

$$EBV(bucket) = BV(Project) \times EBVI(bucket)$$

Where *EBVI(bucket)* represents the *percentage of its own value the bucket has earned* and the following recursive formula for calculating *EBVI(bucket)*. The formula for EBVI is what you'd expect, if you were familiar with using WBS's as "rollup structures". EBVI is a recursive function, calculated by the following formula:

$$EBVI(bucket) = \frac{\sum\limits_{subbuckets}\left(wt(bucket) \times EBVI(bucket)\right)}{\sum\limits_{subbuckets} wt(bucket)},$$

and the weight of a story is either 0 or 1 (it's either done or it's not).

Because a story earns its value by working, there are some interesting things that happen if it fails to work. That is, if a bug is found in a scenario, then either:

- You to lose your EBV for that scenario (because your tests don't pass now), or

[9] This philosophy is similar to Ron Jeffries' Running Tested Features (RTF) metric that he uses for XP projects, but by using weights we get a more robust BVI calculation. Note that it is possible to be more precise than this, and calculate the EBV for a story based on different categories of test. For 'critical' tests, failing means EBV = 0; for 'nice to have' tests, failing means EBV is decreased by 5% (accumulated multiplicatively); for 'trivial' tests, failing means EBV is decreased by 1%; and so on – you get the idea…

- You must add a new scenario for the use case for the (sub)scenario that fails, thus watering down the EBV for the whole use case (since you have added a new scenario…).

In either case, I claim it is reasonable behavior for an EBV calculation, and shows how "doneness" can be "undone" when things stop working.

## *Color of Money Problem*

Many organizations like to use the WBS to track the money of various kinds. For example, they may have a pot of money to be spent on analysis, another pot to be spent on design, another for coding, another for management, another for testing, and so on. Because of this, they often construct a WBS where the legs represent funding sources. That is, their WBS represents the "colors" of money they are spending.

Our WBS is not structured this way, so what do we do? We call this the "color of money" problem, and solve it in a quite simple way. We make sure that either our stories or our tasks within the stories are a specific color of money. This allows us to track actual costs against each funding source, as the tasks are managed.

For example, if we have analysis money, design money, development money, and testing money, we make sure that every functional story has analysis tasks, design tasks, development tasks, and testing tasks. It's not complicated, but has thrown many organizations off track.

Note that this requires us to track actual expenditures of time against the tasks. This is a double-edged sword. On the one hand, it requires detailed bookkeeping at the task level, something we usually recommend against in an agile project. On the other hand, it allows us to manage our money down where the work is, allowing us to fine-tune our planning.

## *Summary*

There are two views of the work on a software project, the Developer's view and the Business view. Our standard Work Backlog is the proper view for development, but is deficient for providing the 50,000$^{ft}$ view for management.

Adding a functional Work Breakdown Structure (WBS) provides the additional formality management needs, as well as providing a framework for calculating Business Value (BV), Earned Business Value (EBV) and solving the "color of money" problem.

In an agile project, both of these views are "in motion" based on reality-based planning, constant monitoring, and frequent verification and validation.