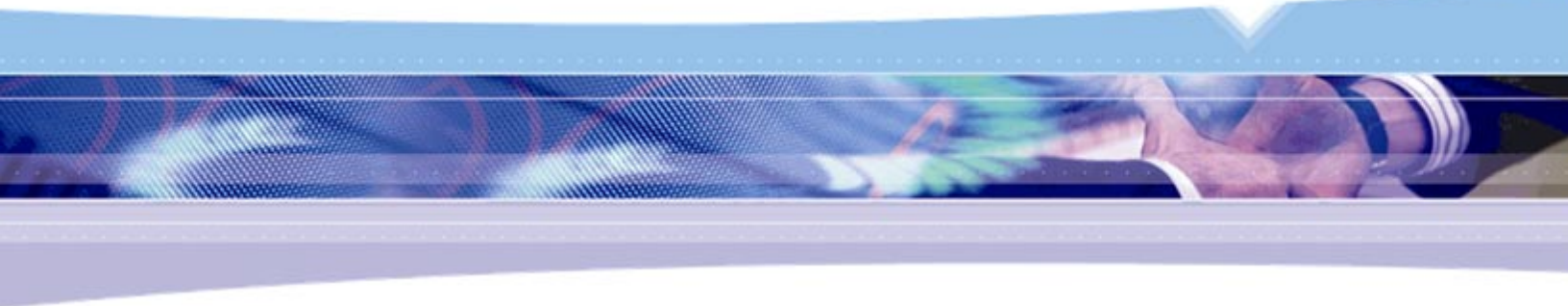


Integration Driven Development

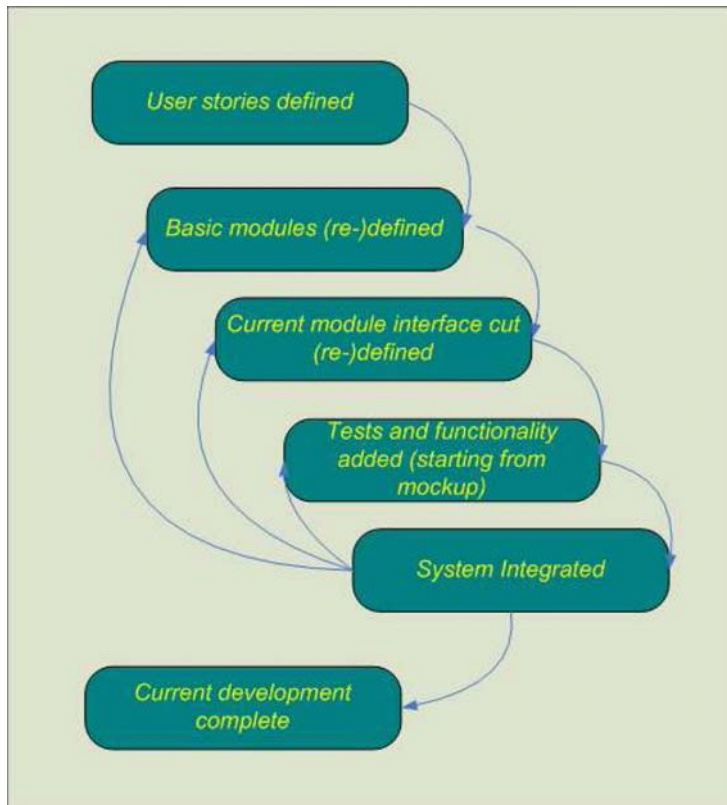
By Alex Yakima and Don Widrig



In this article, we consider the main principles and techniques of Integration Driven Development (IDD), appealing to real life Agile development case studies.

By IDD, we understand development based upon fast development of basic module mockups, immediate integration of those mockups, and a gradual build-up of corresponding functionality in modules up to the complete implementation of the entire system. We assume that interfaces between modules may change, module content may change, and actual modules may appear or disappear within the system.

So, how is it supposed to work? Let's see a typical IDD workflow:



Within this article, we will be following an example of IDD based upon a real project – a cluster system with intensive document processing. We will consider the transition to a new architecture as an example of efficient use of IDD.

Up-front Design vs. Emerging Design

There's no way to predict all relevant risks in software development. Thus, it becomes fairly dubious that big up-front design (being an inherent Waterfall attribute) will be efficient. On the other hand, 100% emerging design will not likely lead to maximum efficiency either. Let's compare two paradigms in terms of a daily walk to the office:

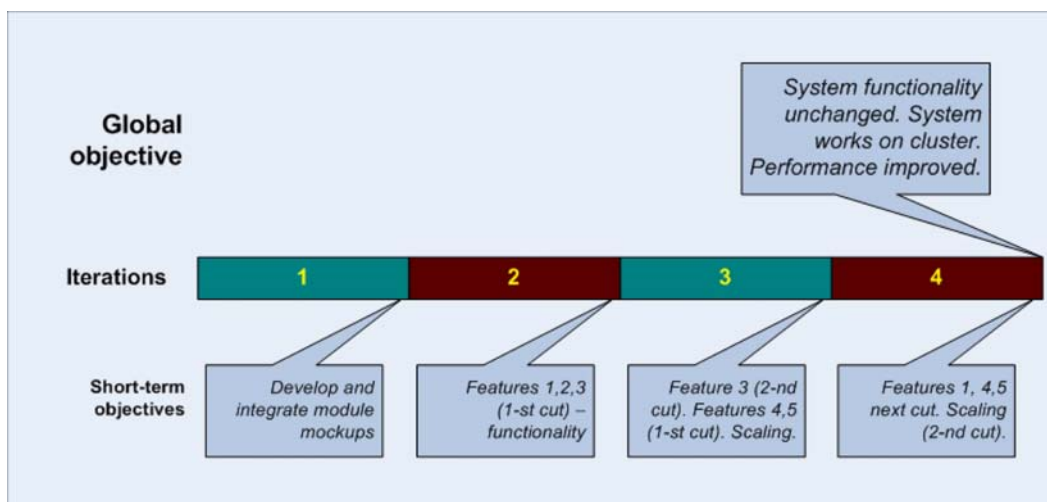
- Emerging design. Wake up. My apartment does not have a balcony – I just don't need it! I head out of the doorway, close my eyes, and start moving by touch. Every 5 minutes, I ask pedestrians to tell me how far I am from my office.
- Up-front design. I wake up in the morning and head to my balcony. I see my office building near the horizon. I try to remember the entire route to my office and start my trip. I move on and look around with confidence and compare the reality to the view from my balcony. I'm continually asking myself – "where am I?"

This could seem trivial but we always go to our job being aware of where we need to finally be. Moreover, we're using our "vision" (I mean vision literally here, like eye-vision). We cannot see thru the walls but still are efficiently using the view to the next corner.

So, using our vision right, we define the basic components of our product, the modules. These modules are responsible for logically consistent pieces of functionality. Of course, we define our modules together with their interactions. Based upon a superficial description of the interactions, we can define our interfaces between modules. Very important: module and interface definition is the responsibility of the entire team, not a system architect. It does not eliminate the role of an architect from the team. This only means that:

- Every team member has an adequate understanding of system architecture
- Every team member contributes to the system design

We are using one-week iterations (development time-boxes) in our case. The transition to a new architecture has been planned for a four-iteration period. Our plan looks like this:



Initial interface discussions and elaboration takes approximately 1 day. These are some pretty intense moments. This is done as a chain of short brainstorming sessions (the entire team participates). Whiteboards are extensively used within these sessions. While discussing interfaces, any questions are appreciated, even if they seem primitive. This is an early validation of the new architecture. The team must have real courage to fill up

the entire whiteboard with the new system architecture outlined and start architecture elaboration all over from scratch in case they can not answer one or more “primitive” questions.

An important thing: one of the goals of IDD is to develop mockups but not the interface. By mockup, we define a module such that all its entities of primitive types are hardcoded constants.

Exactly this kind of (rough if you want) notation of mockup and real data (even though it is hardcoded) helps us prevent mistakes in interfaces.

Spontaneous Pair Programming

Further interface clarification is done in pairs and coincides in time with the initiation of mockup development. Every team member commits to develop some module (or modules). If module A interfaces module B, then the developers who are responsible for A and B respectively should work as a pair. It does not mean classic Pair Programming. In real life, this is spontaneous pair programming and its goal is:

- Joint discussion of interfaces
- Exchange of knowledge on system design
- Joint debugging/solving problems with current implementation issues.

Developers spend approximately 10%-15% of their time in pairs. We call it spontaneous because developers work in pairs as required – spontaneously. Since each module usually interfaces to more than one module – pairs are regrouping themselves. This helps every developer to understand the entire system architecture.

Rotation

We also use rotation of developers from one module to another. There’s a simple rule: no more than 2 weeks spent on the same module. As soon as a developer inherits the module, they may do anything they like with it in the light of their commitment. The first thing to do is usually refactoring. While investigating the code, the developer makes decisions to add some unit tests, change implementation of methods, etc.

Change of Interfaces

We have already pointed that IDD is not about big up-front design. That’s why in IDD we assume the evolution of interfaces within the development process. This does not hinder their planning and thus distinguishes IDD from the “emerging design” paradigm. Here’s the major things that cause interface changes:

- New user story. This is normal for Agile. Not just normal, but even necessary. Scope changes keep both team and code in good form.
- Interface planning mistakes. This is an absolutely normal occurrence. It always takes place. While planning interfaces and creating module mockups, one can never know if the interface covers all the “needs” of future functionality of this module and the ones adjacent to it. This kind of problem is solved as functionality is gradually added.
- Performance. While implementing the system, it may turn out that it is not fast enough or can not handle some amount of concurrent requests, etc. This kind of stuff leads to further refactoring.

All these changes to architecture lead to qualitative changes of architecture or qualitative increments. We call them cuts.

Planned Evolution

In our case, we consider two aspects of the design implementation plan: short-term and long-term.

The short-term aspect is the vision for one-two iterations ahead. This is our “view to the next corner”. This is the goal which will not require dramatic refactoring efforts within short-term implementations because it is quite visible.

The long-term aspect is the entire “path to the office”. The long-term system design plan is a fuzzy plan. It assumes gaining of some high-level objectives but does not define architecture in as much detail as the short-term plan does. Let’s also notice that the design definition (whether this is short- or long-term) does not imply any CASE-artifacts, etc. All the necessary modeling can be accomplished on whiteboards in free form.

The long-term plan implies two kinds of refactoring:

- Unexpected. We’ve actually described it previously
- Planned. We assume how the system should evolve at a high level and also assume that, at some point, we will need refactoring. The goal is to turn to a qualitatively new level of architecture.

Let’s consider examples of short-term and long-term planning. In our example, at the beginning, we had a short-term plan for iterations 1 and 2. Iteration 1 – module and interface definition, iteration 2 – adding functionality to the modules in order to implement system functionality partially. We need those functions which will let us uncover the majority of risks and to get a basic functioning system. The current (short-term) plan implies a single-server architecture. The long-term architecture requirement was a cluster system. Thus, we’ve got two types of refactoring: unexpected (interface mistakes, etc.) and planned (from the very beginning the system was developed for single-server architecture, so its modules were supposed to be C++ libraries. In iterations 3 and 4, we’re refactoring it in order to turn C++ libraries into network services with corresponding concurrent access policies, etc.).

A short-term plan is created at the beginning of every iteration. Of course, building a current short-term plan may easily result in changes to the long-term plan.

Advantages

One of the biggest advantages of IDD is a high velocity of development. We’re not expecting the final design to emerge. We’re trying to make provision for it as far as our “vision” allows us to. Rapid architecture validation via integration starting right at the mockups development stage minimizes major system architecture mistakes. Another advantage – the IDD approach will work in cases when both up-front and emerging design will appear most likely as helpless. This is most important in all systems where performance is one of the key system requirements.

In our case, performance was one of the most important points. The high-level objective was fast document processing based on a total storage of a couple of terabytes. IDD proved to be successful. Without short-term and long-term design planning, we would not be able to implement the system at that stage.

Finally, IDD works fine within Scrum or environments close to it. Since IDD is only a system design planning and implementation approach, it does not impose strict requirements or constraints onto programming techniques.