Hybrid Value-Driven Delivery Model

Agile/Offshore Delivery Process Specifications
Version 2.1

By

Joel Grenon,
Stephane Rainville,
Sebastien Guimont

Software Development Center
Covansys Canada

Legal stuff goes here

Date	Version	Description	Author
February 17th, 2004	1.0	Initial Version	S. Rainville, J. Grenon, S. Guimont
May 5th, 2004	2.0	In-depth overview	J. Grenon
October 7th, 2004	2.1	Added roles	J. Grenon

Table of content

Agile Software Approach

HVDD is based on the agile software development approach. Agile is a set of principles and best practices putting emphasis on communication and flexibility instead of relying on rigid processes and large amount of documentation.

Agile Manifesto

Agile is driven by high-level principles that must be respected by all agile developers. These principles support an attitude where change is the normal state and customer-value is the most important thing in every project.

1. *Our highest priority is to satisfy the customer through early and continuous delivery of valuable software.*
2. *Welcome changing requirements, even late in development. Agile processes harness change for the customer's competitive advantage.*
3. *Deliver working software frequently, from a couple of weeks to a couple of months, with a preference for the shorter time scale.*
4. *Business people and developers must work together daily throughout the project.*
5. *Build projects around motivated individuals. Give them the environment and support they need, and trust them to get the job done.*
6. *The most efficient and effective method of conveying information to and within a development team is face-to-face conversation.*
7. *Working software is primary measure of progress.*
8. *Agile processes promote sustainable development. The sponsors, developers and users should be able to maintain a constant pace indefinitely.*
9. *Continuous attention to technical excellence and good design enhances agility.*
10. *Simplicity – the art of maximizing the amount of work not done – is essential.*
11. *The best architectures, requirements and designs emerge from self-organizing teams.*
12. *At regular intervals, the team reflects on how to become more effective and the tunes and adjusts its behavior accordingly.*

Individuals and interactions over processes and tools

Extract from Agile Manifesto, Agile Alliance, 2001

Teams of people build software systems and to do that they need to work together effectively with programmers, testers, project managers, modelers and customers. Who do you think would develop a better system: five software developers with their own tools working together in a single room or five low-skilled "hamburger-flippers" with a well-defined process, the most sophisticated tools available and the best offices money could buy? If the project was reasonably complex, my money would be on the software developers, wouldn't yours? The point is that the most important factors to consider are the people and how they work together, because if you don't get that right, the best tools and processes won't be of any use. Tools and processes are important, don't get me wrong, it's just that they're not as important as working together effectively. Remember

the old adage: a fool with a tool is still a fool. This can be difficult for management to accept because they often want to believe that people and time, or men and months, are interchangeable [Brooks 1995].

Working software over comprehensive documentation

Extract from Agile Manifesto, Agile Alliance, 2001

When you ask the user whether they want a 50-page document describing what you intend to build or the actual software itself, what do you think they'll pick? My guess is that 99 times out of 100 they'll choose working software. If that is the case, doesn't it make more sense to work in such manner that you produce software quickly and often, giving your users what they prefer? Furthermore, I suspect that users will understand any software that you produce much more easily than they will understand complex technical diagrams describing its internal workings or describing an abstraction of its usage, don't you? Documentation has its place, written properly, it is a valuable guide for people's understanding of how and why a system is built and how to work with the system. However, never forget that the primary goal of software development is to create software, not documents – otherwise we would call it documentation development, wouldn't we?

Customer collaboration over contract negotiation

Extract from Agile Manifesto, Agile Alliance, 2001

Only your customers can tell you what they want. Yes, they likely do not have the skills to exactly specify the system. Yes, they likely won't get it right the first time. Yes, they'll likely change their minds. Working together with your customers is hard, but that's the reality of the job. Having a contract with your customers is important and having an understanding of everyone's rights and responsibilities may form the foundation of that contract, but a contract isn't a substitute for communication. Successful developers work closely with their customers; they invest the effort to discover what their customers need and they educate their customers along the way.

Responding to change over following a plan

Extract from Agile Manifesto, Agile Alliance, 2001

People change their priorities for a variety of reasons. As work progresses on your system, your project stakeholder's understanding of the problem domain and of what you are building changes. The business environment changes. Technology changes over time, although not always for the better. Change is a reality of software development, a reality that your software process must reflect. There is nothing wrong with having a project plan. In fact, I would be worried about any project that didn't have one. However, a project plan must be malleable, there must be room to change it as your situation changes; otherwise, your plan quickly becomes irrelevant.

Agile Values

Communication

This principle is in fact about effective communication. Communication is the most important principle of any agile project. Communication in a software project is utterly complex. Customer communicates requirements, designers communicate their view of the solution, programmers communicate their implementation by writing code and project manager communicates budget and deadlines to everybody.
Communication is affected by physical proximity, temporal proximity, tools used to communicate and various non-quantitative factors like amicability and commitment.
In an offshore project, communication mechanisms are pushed to their extreme limits as people from two different continents, in radically different time zones, work together to reach the same goal. To communicate this goal effectively, to communicate ideas and problems, forces everybody to use all available communication tools. Agile developers use the best communication mechanism available. When face-to-face meetings are not possible, they rely on explicit e-mails, phone calls, video-conferences, code comments or any other way to keep pass their message.

Simplicity

Simplicity is about taking the simplest solution to one problem. Simplicity doesn't necessarily imply doing simple things; it's just building for today's needs and worry about tomorrow's problems tomorrow. Simplicity is often hard to achieve. The complex pattern temptation is always present in the mind of the developer. Applying complex patterns too soon may quickly result in complex class hierarchies that will solve today's and the next hundred years problems. If that's the simplest solution, do it.

With ever changing requirements, simplicity means that you will loose the least amount of work possible for any change you apply to the system. Simplicity also tends to avoid solving false problems that could occur tomorrow. You solve today's problem that's all.

Feedback

The only way you can determine whether your work is correct is to obtain feedback. That includes feedback from the customer but also feedback from your peers, feedback from the management team, feedback from your unit tests. The faster the feedback comes, the most effective it will be. Agile projects should strive to get customer's feedback as often as possible and perform review activities as early as possible. If communication is the engine of the project, feedback is the steering wheel.

Courage

Agile software development principles challenge the status quo and that's threatening for many people. It is a lot easier to sit back and accept the current situation, to not try to improve things or to wait until someone else comes along and fixes thinks. Agile developers should have the courage to trust other people and himself. Agile developers should have the courage to trust that he can solve tomorrow's problems tomorrow. Agile developers should have the courage to only documentation what they feel is required.

Agile developers should let business people take business decisions and business people should let technical people take technical decisions; that takes courage too.
Agile developers should have the courage to recognize that they can, and will, make mistakes and have the courage to learn from them.

Humility

Humility is about recognizing that as good as you are you can always learn something new from someone else. The best agile developers are the one able to share their knowledge and ask questions to team mates. Agile developers understand that fellow developers and their project stakeholders have their own area of expertise and have value to add to a project. Agile developers have the humility to respect people that they work with, realizing that others likely have different priorities and experiences that they do and therefore will have different viewpoints. Agile developers are humble and more effective as a result.

Agile Core Principles

Agile core principles guide all aspect of software development. They are general principles that must be integrated in day-to-day activities and always applied to get an agile process. They are listed here because they are the foundation of every agile process and ours is no exception.

Software is your primary goal

The primary goal of software development is to produce high-quality software that meets the needs of your project stakeholders in an effective manner. Your primary goal is to deliver value to the customer and value usually comes from functional software. While documentation may help you improve communication and achieve better quality software, it must not be a goal in itself. In short, any activity that does not directly contribute to the goal of producing quality should be questioned and avoided if it cannot be adequately justified.

Enabling the next effort is your secondary goal

Your project can still be considered a failure even when your team delivers a working system to your users. Part of fulfilling the needs of your project stakeholders is to ensure that your system is robust enough so that it can be extended over time. Your next effort may be to develop the major release of your system or it may simply be to operate and support the current version that you are building. To enable it, you will not only want to develop quality software, but also create just enough documentation so that the people doing the next effort can be effective, transfer skills from your developers to others, motivate existing staff to stay and develop the next release of your system or simply motivate team members to stay with your organization. In short, when you work on your system, you need to keep an eye on the future.

Travel Light

Traveling light means that you create just enough models and documentation to get by. Every artifact that you create, and then decide to keep, will need to be maintained over

time. This includes models, documents and project management artifacts such as schedules, test suites and source code. The more complex/detailed your models are, the more likely it is that any given change will be harder to accomplish. Every time you decide to keep a model, you trade off agility for the convenience of having that information available to your team in an abstract manner. Never underestimate the seriousness of this trade off.

You need good communication among your team to be able to effectively travel lightly; if developers don't understand your requirements or your architectural approach or at least if there is no one that they can work with to get their questions answered readily, then you are in serious trouble. Clearly, good communication is a requisite to support traveling light. In short, traveling light enables simplicity in your approach to development because your artifact maintenance efforts during development are dramatically decreased.

Assume simplicity

As you develop, you should assume that the simplest solution is the best solution. The vast majority of the time the simplest solution works well and because it is simple, it is easy to implement. The advantage is that you aren't investing extra time implementing difficult solutions, approaches that take more time and effort to put in place. The advantage is that a few times where the simplest solution proves not to work, you have time to implement a more difficult one because you haven't wasted resources elsewhere. Furthermore, the simplest solution is also the easiest to maintain and enhance.

Will taking the simplest approach work every time? Likely not, but it will work the vast majority of the time. When it doesn't work you will have learned something and will very likely have failed very early in your efforts. Contrast that to taking a complicated approach – complicated approaches fail as well – where you've invested significant resources to discover that your ideas didn't work.

Embrace Change

Accept the fact that change happens. Revel in it. Change is one of the things that make software development exciting. Requirements evolve over time. Your project stakeholders' understanding of their requirements changes over time. Project stakeholders can change as your project moves forward, new people are added and existing ones leave. Project stakeholders can change their viewpoints as well, potentially changing the golas and success criteria for your effort. Furthermore, your business and technological environments change as your project evolves; things occur that are often beyond the scope of your control. The implication is that your project's environment changes over time.

Agile developers embrace change. They understand that change is a common occurrence on software projects. Agile developers know that their work will be affected by changes; they actively strive to communicate with their project stakeholders, to seek their feedback, so they can identify changes and then act accordingly.

Agile developers also recognize an inherent danger in embracing change – the tendency to get sloppy when doing up-front work such as requirements analysis. Why invest a lot of time understanding requirements if they're only going to change? The answer is that you always have to invest time to understand the requirements to the best of your ability

now and implement software based on those requirements. Some requirements will change and you need to embrace this fact, but many requirements won't change, at least not soon.

Incremental Change

To embrace change you need to take an incremental approach to your own development efforts, to change your system a small portion at a time instead of trying to get everything accomplished in one big release. You can make a big change as a series of small, incremental changes. You should deliver working software frequently. An important concept to understand in agile projects is that you don't need to get everything right the first time. In fact, it is very unlikely that you could do so even if you tried. It takes humility to accept that you can't get it right the first time or even the nth time and courage to admit it. Make it run, make it right and then make it fast. (Kent Beck, 2000)

Work with purpose

Many developers worry about whether their artifacts – such as models, source code or documents – are detailed enough of if they are too detailed or similarly if they are sufficiently accurate. What they're not doing is stepping back and asking why they're creating the artifact in the first place and whom are they creating it for. This requires humility; you aren't developing solely for your personal satisfaction; instead you are developing to fulfill the needs of your project stakeholders.

Quality Work

Nobody likes sloppy work. The people doing the work don't like it because it's something they can't be proud of. The people coming along later to refactor the work, perhaps yourself a few weeks later when your requirements change or perhaps a maintenance developer who as been assigned to evolve your system, don't like it because sloppy work is harder to understand and therefore harder to update. In other words, quality work improves communication on your project. Your end users won't like your sloppy work because it's typically fragile and/or doesn't meet their expectations. Senior management or customers won't like your sloppy work because they will feel they aren't getting good value for their investment in your efforts.

Agile developers understand that they should invest the effort to make permanent artifacts, such as source code, user documentation and technical system documentation of sufficient quality. Similarly, agile developers don't invest much effort in artifacts that they intend to discard, particularly sketches or low-fidelity artifacts such as essential user interface prototypes. In other words, they have the humility to spend their time wisely because they realize they would be wasting their project stake holder's resources otherwise.

Rapid Feedback

The time between an action and the feedback on that action is critical, that's why agile developers strive to maximize rapid feedback. There are two reasons why rapid feedback is important: We make most of our mistakes in the early aspects of development and the cost of fixing defects increases exponentially the later they are found. This happens because of the nature of software development – work is performed based on work

performed previously. If a requirement was misunderstood, all designs, code, tests created for this requirement are potentially invalid. If the only feedback you receive is the errors detected late in the lifecycle of your project, during testing in the large or after the application has been released, they are likely to be very expensive to fix. However, if you receive feedback quickly, just after misunderstanding what you were originally told, it will be much less expensive to address the misunderstanding.

Maximize stakeholder investment

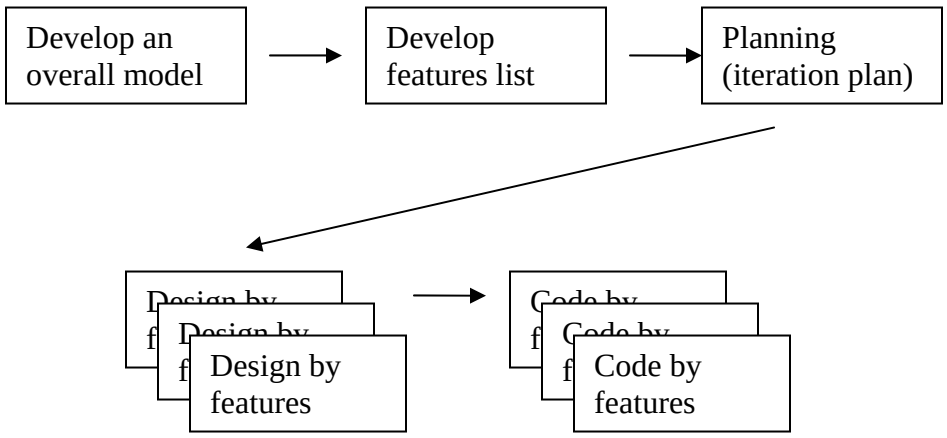
Your project stakeholders are investing resources – time, money, facilities and so on – to have software developed that meets their needs. Stakeholders deserve to invest their resources the best way possible and to not have them frittered away by your team. Furthermore, stakeholders deserve to have the final say in how those resources are invested or not invested. If it was your money would you want it any other way?

Feature-Driven Development

Feature-driven development is an agile approach that concentrates on customer value as a way to manage software projects. Systems are decomposed in functions labeled in the customer language. Typical functions involve an action, a result and a subject (ex: Calculate the total amount of a sale).

The benefits of FDD for customers are that project tracking is simpler because all activities are centered on features. Progress is tracked using completed features, designers design features in whole (no layering or componentization), developers develop features and features are tested as a functional unit. Efforts are concentrated on functions instead of infrastructure, which result in a better usage of resources, enforcing agile core principles.

Projects are organized in iterations where a fix number of features are implemented. Features may not be fully functional after a single iteration (iterative change principle), but it should be functional in order to give feedback opportunities. The following diagram describes a typical FDD process.



Develop an overall model

An overall model of the system to develop is required in order to get the high-level view required to identify features and later organize them into a development plan. The overall model should be detailed enough to help all team members understand the system. This isn't a formal specification but detailed enough to get the team started and get a feel of the features to implement.

Develop features list

Features are then extracted from the overall model. A structured features list, prioritized based on dependencies and customer's need is created.

Planning

Planning is about creating a development plan where features are assigned to iterations and a team is created to perform work in all those iterations. The plan is concrete for the first two or three iterations and much more incomplete as we go in time. This is again in accordance to the agile approach where documentation quality is important but we have to invest just enough effort to have things right. We don't need to invest a lot of time in getting a concrete planning for future iterations because this time would surely be wasted because of changes.

Design by features

All features are designed independently. In pure feature driven, source code artifacts like classes are owned by one developer and all changes go through this single person. Design tends to be self contained with references to reusable components or services identified in the architecture.

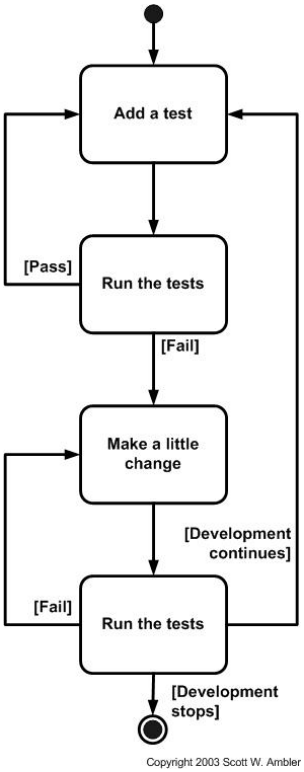
Code by features

As soon as the design is completed, the feature is forwarded to a developer who will be responsible for the implementation of all specified modules used to deliver the feature. This is a vertical decomposition where components of many architectural layers may be in the responsibility of a single developer.

Test-Driven Implementation

TDD completely turns traditional development around. Instead of writing functional code first and then your testing code as an afterthought, if you write it at all, you instead write your test code before your functional code. Furthermore, you do so in very small steps – one test and a small bit of corresponding functional code at a time. A programmer taking a TDD approach refuses to write a new function until there is first a test that fails because that function isn't present. In fact, they refuse to add even a single line of code until a test exists for it. Once the test is in place they then do the work required to ensure that the test suite now passes (your new code may break several existing tests as well as the new one). This sounds simple in principle, but when you are first learning to take a TDD approach it proves require great discipline because it is easy to "slip" and write functional code without first writing a new test.

The first step is to quickly add a test, basically just enough code to fail. Next you run your tests, often the complete test suite although for sake of speed you may decide to run only a subset, to ensure that the new test does in fact fail. You then update your functional code to make it pass the new tests. The fourth step is to run your tests again. If they fail you need to update your functional code and retest. Once the tests pass the next step is to start over (you may also want to refactor any duplication out of your design as needed).



Kent Beck, who popularized TDD in eXtreme Programming (XP) (Beck 2000), defines two simple rules for TDD (Beck 2003). First, you should write new business code only when an automated test has failed. Second, you should eliminate any duplication that you find. Beck explains how these two simple rules generate complex individual and group behavior:

- You design organically, with the running code providing feedback between decisions.
- You write your own tests because you can't wait 20 times per day for someone else to write them for you.
- Your development environment must provide rapid response to small changes (e.g you need a fast compiler and regression test suite).
- Your designs must consist of highly cohesive, loosely coupled components (e.g. your design is highly normalized) to make testing easier (this also makes evolution and maintenance of your system easier too).

For developers, the implication is that they need to learn how to write effective unit tests. Beck's experience is that good unit tests:

- Run fast (they have short setups, run times, and break downs).
- Run in isolation (you should be able to reorder them).
- Use data that makes them easy to read and to understand.
- Use real data (e.g. copies of production data) when they need to.
- Represent one step towards your overall goal.

This approach ensure that all code with be testable and tested throughout the development process. It also reduces the stabilization period that occurs at release time but increase a little bit the effort of writing code.

Delivery Approach

Delivery approaches are defined as the way we organize a team of skilled people to release a system. Lessons learned in the industry direct us toward the hybrid model where the right skill set is found in various sites. This section quickly covers two pure models and goes in-depth with the hybrid one.

Pure Internal Delivery

Pure internal team model is the classic approach where an enterprise is hiring all resources used to deliver internal IT systems. This approach was interesting in the past with a limited number of systems and also a limited number of skills required to deliver those systems. But today, with an ever growing number of systems, e-business requirements and full business process automation, it is hard for any enterprise to be able to keep all required skills in-house. Enterprises have to be agile. Similarly to agile software development, if you keep too many different resources to train in various skills, you quickly trade agility with the convenience of having the resource available in your team (see Travel Light).
For all those reasons, the pure internal delivery model is rarely a choice in today's IT landscape.

Pure offshore delivery

Pure offshore is a delivery approach where all (or nearly all) development activities are performed offshore. This model is the classic offshore approach where client work directly with an offshore workforce. For this model to work, there are various requirements that needs to be put in place. First, requirements must be well defined and very formal. This helps minimize communication problems and make the offshore team autonomous. In this model, you really have to minimize the number of questions/answers loops, which could really slow down the project pace to a crawl due to usually enormous time lag. Second, the model is usually at his best with projects (iterations) of 1 to 3 months where the offshore team can work at its pace and enough people can be trained to participate in the project. The problem is that you don't see anything for 1 to 3 months so feedback comes late, which increases the defect resolution costs. This model tends to slow down development by 20% - 35% and usually requires more resources then projects delivered using the internal delivery model. With the kind of price found in offshore companies, there is still a direct benefit, but really minimized by time-to-market delays and lack of flexibility in delivering the system.

Pure-play offshore providers sometimes brings one or two resources onsite to help in making the process more flexible, but the project success becomes directly linked to those resources quality and thus makes a weak model for reproducible successes.

Hybrid delivery

The hybrid delivery model merges both models previously discussed to create a mix of skills between onsite, offsite and offshore locations. This approach helps reduce offshore management complexities by adding a local team, architect, analysts, designers and developers to get faster feedback loops. This approach enables a partnership work with

onsite and offsite resources for requirements discovery, analysis and architecture and a partnership with offsite and offshore for design and coding. While individual resource costs are higher than in pure offshore, the efficiency obtained by having closer technical resources offset this problem. Depending on the system environment, as defined in the initial assessment, offshore ratio up to 70% of the total effort can be achieved. Additional benefits like improved time to market, improved flexibility regarding requirements evolution and the capacity to use agile software development approach in this team model makes the hybrid model the ideal solution for most software development projects. Customers get the required skills from any of the three locations and also avoid direct management of the offshore team, which requires special project management abilities, structured communication and distributed team management experience.

HVDD Model Explained

The Hybrid Value-Driven Delivery Model is a adapted from the all processes and approaches listed in the previous section. HVDD is a unique process targeting the following objectives:

A. Excel in highly volatile requirement environment.
B. Provide excellent budget and function driven control mechanisms.
C. Minimized costs through the use of offshore team members.
D. Deliver High-Quality system through integrated QA mechanisms

These objectives are addressed by integrating various elements from the open-source world, agile programming and project management techniques.

Open-source inspiration

Open-source has revolutionized the way software is created in the 21st century. While not every company has adopted this licensing model, the techniques employed for managing and organizing activities in an open-source project are gradually accepted as best practices by all development teams. The HVDD process inherits from certain key aspects of open-source development.

Single distributed team

The HVDD process uses a variant of the hybrid offshore delivery model where all persons working on the project are part of a single distributed team. This is a little bit different than the classic way of grouping responsibilities by physical locations. The classic approach would be to define formal specifications as input and expect an output in agreement with them. While this approach is interesting, it lack the agility required by the HVDD process and doesn't support well other aspects of the process.

In HVDD, everybody is part of a single team. Physical locations have no meaning as work is separated on a skill based approach. Work flow any ways, between onsite, offsite and offshore resources based on sequences prescribed by the process. This way, an onsite analyst could have a feature designed by an offshore designer and implemented by an offsite developer or vice-versa.

For this approach to succeed there must be a good communication between all team members. The challenge is in having single feature information flow fluently between all persons involved in its implementation. The fact that all features are developed in silo, in a well-defined architecture and with a good overall model, helps reduce communication complexity and simplify work organization.

Centralized source repository

Like all open-source projects, HVDD projects use a centralized source and documentation repository. These repositories are available to all team members and are acting as the central channel for work organization. A lot of information goes through

these repositories, where analysis documents, design documents, source code, tests and configuration scripts flow between all team members. Joined with automated tools, these repositories help track results and give cross-feature roles (architect, project manager and analyst) a feeling of the evolution of the project.

Automated build tools

Automated build tools is another essential part of the project. Usually, the approach is to make sure that all developers have a working copy of the system on their machine to quickly update it and perform debugging activities. Automated building tools help them quickly setup their environment and build all artifacts required for the system to function correctly.

In addition to the development environment, automated build tools are used to support continuous builds, which run at every hour or faster, extracting everything needed for the source repositories to build a working system. Each time these build executes, all unit tests are executed to provide rapid feedback to all developers on potential system failures caused by new code integration. This quick feedback helps developers detect cross-features conflicts and are used as a net when performing refactoring or optimization activities during the integration phase (see process).

Frequent releases

Release often is the motto of all open-source projects. By releasing often, you seek feedback from your user community or from other developers. You also avoid the big bang deployment where all issues are detected simultaneously and slow down new development considerably.

All HVDD projects should use a continuous build and perform a development release every day or every week depending on the project scope (or the duration of an iteration).

Feature and Test Driven

Feature Driven development and test driven development provides the foundation of our HVDD process. They key to a success system delivery is in feature discovery mechanisms and in the way the feature pool is managed. The silo implementation is supported by the test first approach which enables later refactoring of the system without wondering if something got broken due to the presence of a “net” of unit tests covering most of the code.

Feature Pool Management

Feature management is a complex aspect of the HVDD process. In order for the system to follow customer expectation, feature development must be organized following a certain number of rules that reflects customer priority and other factors. Pool management is performed by the project manager with the customer. The analyst (including customer representative) is responsible for adding new features to the pool but the prioritization of these features is performed by the customer and the project manager. The architect may also be involved in this activity for architectural support advising.

Considering that any significant system may contains from 200 to 1000 features, their effective organization and prioritization is essential to the success of the project. HVDD defines various rules that should be taken into account when organizing the feature pool. More details about organization rules can be found in the *process* section.

Iteration Planning

Iteration planning is about extracting a certain number of features from the pool, usually the top ones, based on all applied prioritization rules, and organize them within a given iteration. An iteration is fix-length unit of work where a certain number of features will be develop. The result of an iteration should always be a functional system, at least for the listed features. The goal of the iterative process is to get rapid feedback from the end-user in order to fine-tune the execution of future iterations. Iteration duration is arbitrary but is usually around 2 to 6 weeks, depending on the team size, the required velocity and volatility of the requirements. The velocity is a metric indicating the average number of features developed by a team during a given iteration.

As defined by the macro process (see process section), iterations are not equals and may contains different activities depending on the current macro phase of the project. This as to be taken into account when planning an iteration. The net result of the macro process is that actual feature implementation should start at least an iteration later than analysis and discovery iterations. The ideal situation is to have two (2) defined and analyzed iterations ready to be developed and a third in progress. This gives developers a stable set of requirements and provides designers with plenty of visibility to understand all features and get clarifications from the analyst.

Once planned and started, an iteration should never be changed. This is an important rule as it implies that changes are welcomed, but in a future iteration. This enables the team to concentrate on a well defined workload and help them guarantee a delivery date and the maximum level of quality. Constant changes creates delays, even more delays when the hybrid delivery model is applied, and will definitively result in reduce quality and cost increases. It is usually better to get an implemented feature that is slightly wrong compared to new requirements than to get a functionally right feature with wrong implementation.

Test First, Code to make tests run

At implementation level, TDD (test-driven development) approach is used to ensure minimal code to fulfill feature success criteria and at the same time to generate customer value. Feature analysis contains a set of success criteria that are mandatory for the feature to be accepted by the customer. These criteria guide the priority of designers and developers toward implementing the right thing to generate value. Though TDD, implementation is kept minimal as features are grown organically inside a structure and well-defined architecture. TDD also warrants that the feature will be easily tested in the future, which is a requirement to support integration activities (refactoring, performance, security, etc).
In HVDD, the TDD approach is a little bit modified in a way that tests are defined in groups by the designer and implemented by the developer. We adopt this approach to

maximize the designer experience in testing feature and maximize the developer skill in developing the test and testable code. New unit tests are usually added by developers as they progress in implementing the feature. These tests are confirmed by the designer, which may also add new tests when the feature implementation is complete.

Accountability

In a model where responsibilities are shared between all members of a distributed team, the need for accountability is essential. While development effort is split between multiple sites, responsibilities are always assigned to one and only one role during the processing of a feature. This accountability acts as a driving force for the whole development process as everybody is willing to do a good job to ensure that the rest of the process will go smoothly.

Here is a summary of this effect:

The analyst is responsible for the functional decomposition of the system. He is responsible for reaching customer success criteria and making sure that they have been implemented. If his analysis is not good enough, he will put a lot of pressure on the designer, which in turn is responsible for the technical implementation of the feature. With bad input, chances are good that the designer will generate a wrong design and that the developer will also provide a wrong implementation. The designer is responsible for the developer work and is the ultimate responsible of the technical aspects of a feature. If the feature doesn't work, it's the responsibility of the designer to correct the wrongs and gets things running.

As you can see, this chain of responsibility will drive all team members to help their partners in the implementation chain because they are all responsible for a part of the work.

Failure Management

When problems occur, the process provides enough inputs to determine what went wrong and help executive take good decisions to improve the team efficiency. Failure may be caused by unrealistic estimates, bad communication between the analyst and the customer, wrong design or lack of technical skills from the developer or a faulty architecture or an error during the integration or deployment of a feature. The cause of all failures must be determined in order to constantly improve the quality of the process.

Risk Driven Development

Risk driven development is another aspect of the HVDD process that helps in delivering value, on budget. All activities are handled using a risk-based approach. Before starting any activity, the team member is required to assess risks associated with the activity. These risks are usually things that will affect quality or deadlines. For example, a designer might evaluate that a specific portion of a feature might be hard to communicate to the developer or an analyst might find that there are some fields that have multiple values in certain conditions, which may not be fully understood by the designer or developer.

With a list of all risks associated with an activity, the team member is now able to put in place a plan that addresses those risks within the estimated time. This might be to put a little more time to draw a UML state diagram explaining the complex logic or just to take the time to send an e-mail to the mailing, explaining to all the complex field value logic. Everybody is responsible for the identification of all risks associated with their activity or finding help to help assess them. Everybody is also responsible for taking all actions, within estimated time, to reduce those risks. A valid action might be to ask the PM for additional time to complete the feature if quality is at play.

HVDD Roles

The HVDD process defines various roles that are responsible for all activities in a project. This section describes each role and provides an overview of their responsibilities. Additional details will be provided in the process section, where all activities will be assigned to a role.

Project Manager

The project manager is in charge of the execution of the work. He is in charge of maximizing resources throughout the project by monitoring skill sets and required skills for each feature. The project manager is responsible for organizing the feature pool with the customer and organizes work in iteration, based on prioritization rules. The PM is also responsible for the management of the iteration budget, customer invoicing and effort estimations.

Analyst

In HVDD the analyst role is really a functional analyst. He is responsible of decomposing the system into functional units called features that have a minimum of dependencies and can be performed by a single man within 1 to 15 days of work. The analyst is responsible for helping the customer in feature discovery and provides detailed specification of each feature to the designers. He is also in charge of the managing customer sign-off on features and of performing the final functional review of the feature to ensure that all success criteria have been met.

Architect

The architect is responsible for creating a technical environment where feature development can be performed. Architecture in HVDD involved selection of third-party libraries, selection of the platform, definition of API, definitions of all layers of the system, database schema and much more. The architect is the final responsible for the whole system performance, security, maintainability and stability. To achieve this, the architect participates to all design reviews to ensure that they fit in the existing architecture and that they have maximized all reusable components available to implement the feature.

Designer

Designers are responsible for finding technical solutions to implement features. They design each feature independently using whatever communication mechanism they have. The preferred way to document is usually source code comments (TODO, FIXME or others), UML diagrams and finally text documents for more complex problems. They are responsible for designing a testable feature. They design unit tests to ensure that success criteria are met. Designers are also involved in code review and enhancement, within a given feature.

Developer

Developers are responsible for the implementation of all features and their associated unit tests. Developers are responsible for creating the simplest implementation to satisfy the designed unit tests for the given feature. They are also responsible for adding new unit tests for any additional classes or method they have to add in their internal implementation as proposed by the TDD approach. Developers are also responsible for keeping the continuous build running and making sure that all unit tests are running properly before delivering their code.

Integrator

Integrators are the final person touching a feature implementation before it is being released. Integration in HVDD is the action of performing cross-feature refactoring to either optimize the speed, security, maintainability or respect of coding guidelines of the system. They are experienced developers looking for quick ways to improve the code base with efficient refactoring. They are knowledgeable in design patterns and strive to apply them where they see fit, always within the estimated effort. Their work ensures the delivered system will be top quality and their feedback to the designers and developers helps them improve and always strive to take better design decisions. Integrators are knowleable in the architecture (the architect is often himself an integrator) and look for ways to promote feature code to the architecture and/or increase reuse of architectural components within a feature implementation.

Deployer

Deployers are responsible for packaging and installing the system in various environments. They are also responsible for maintaining all environments to ensure that they are ready to accept the system. Deployers are knowledgeable in system configuration and system administration. They are also responsible for fine tuning complex system configuration, cluster configuration, security and other system related activities.

Administrator

The administrator is responsible for supporting the project team in their development activities but is not directly involved in system delivery. He is responsible for managing the source repository, create and maintain developers' accounts and managing development tools like WIKI sites, planning tools and mailing list.

Domain Expert

The domain expert usually interacts with the analyst to define the requirements and explain what is expected from the system and in which form. It is important that the domain expert be present throughout the project to ensure that developers get quick answers and to avoid developing useless code.

Project Sponsor

The project sponsor is usually the one who pay for the project or the one that believe in the value of the system (or both). The implication of the project sponsor is usually with

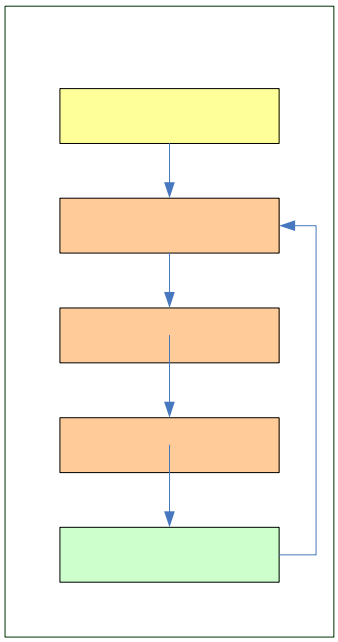
the project manager to ensure that the project is on track and that the identified value will be present at the end of the project.

HVDD Delivery Process

This section covers in details all activities and their sequence in day-to-day project delivery. HVDD has two (2) levels of process called ‘macro’ and ‘micro’ processes. These processes are not independents as they affect each other but execute at different levels.

Macro Delivery Process

The macro delivery process affects iterations as a whole. It affects the kind of activities that will be performed during the iteration and also the number of resources affected to new development and maintenance. The macro development process is repeated on a much longer time span and usually span tens of iterations for typical projects. The following diagram describes all phases of the macro delivery process.



The macro process affects system evolution as a whole and will follow the system from inception to evolution.

Assessment

The assessment is an optional but important phase of the macro process. Optional because the project team may already have enough information to take team model decisions but important because we need to understand the context of the system in order to fine tune the process according to the customer needs.

The assessment is about questioning the current situation and deriving a team model that will most likely fit this context. Aspects like:

- *Number of applications to develop/maintain*

- *Platform and programming languages*
- *Legacy or external systems integration levels*
- *Available skills at customer site*
- *Mentoring needs / Outsourcing level*
- *Security constraints (limited access to systems, VPN, etc)*
- *Backward compatibility requirements*
- *Development tools*
- *Development environment portability*
- *Special Skills requirements*
- *Current costs*
- *Feature Pooling strategy*
- *Etc.*

These factors will help the project team define a delivery model adapted to the context. For example, complex algorithm may require the integration of a special customer programmer in the team or tight system integration might prevent the use of offshore system if connectivity isn't good enough.

This phase deliverable is a report explaining estimated savings, proposed team model and project management assumptions. This document must be approved by the customer before the team is put in place.

Feature Discovery

This phase is about finding and prioritizing features to be developed. Discovery is usually performed by the customer with the help of the analyst and sometimes the architect and targets the creation of a feature pool from which the project manager will extract features to feed the development team. There are various techniques to discover features. Usually, some business sponsors have a general idea of a system that would help him somehow. The analyst is responsible of creating a concrete set of features from this raw idea.

Iterations performed during this phase usually involve only the analyst and architect of the system. For larger system, where more complex prototype are required, a complete development team may be put in place.

System Modeling

Sometimes the feature set is found through Business Process Modeling or Re-engineering. This approach helps understand how the business works and the information flows that are exchanged between organizations or internally between departments. By studying these flows, an analyst may find applications which are transformed in feature sets, documented and placed in the pool.

There is a single pool per system or per customer. All application features are placed in the pool and prioritized using well-defined rules (examples shown in next section). This way many applications may be developed concurrently or one at a time depending on the priorities of the customer.

Prototyping

Prototyping is also a good way to discover features. This approach is to create a dummy application front-end (web or smart client) and ask the user to help organize forms and information models. Prototyping is the most interesting method to get concrete requirements but doesn't cover every aspect of the system. All backend aspects are left to the architect and are usually found during the creation of the architectural model. While prototyping is optional, it is strongly recommended as a good representation of customer's needs.

Architectural Model

The architectural model is a strategic document describing the new system architecture. This document explains in details the structure of the system, its various high-level components and communication protocols or API used to communicate between them. It also describes the selected platform and technology stack, with exact versions to ensure consistency during the system development.

The physical architecture (deployment) architecture is also included in this document. All hardware and software used to host the system as well as network architecture, clusters, messaging and databases.

This document becomes the spine of the system and must be revised each time a new discovery phase is started (macro process).

Feature Pooling

Discovered features must be organized to make sure they all gets implemented to create the required system. Manual pool management is possible for small systems but with systems with more than 250 features, assisted pool management is required to compute all rules and always provide a sorted and current view of the pool.

Pooling is controlled through a set of rules that are agreed by the customer and the project team during the assessment phase. The following rules are usually used for most pool management.

Pooling Rule #1: Customer Value

The most important rule is to sort features based on the value they bring to the customer. Only the customer or the analyst can know the value a feature will bring and even in most cases, value may be something very subjective.

Pooling Rule #2: Feature Dependencies

While HVDD strive to minimize dependencies between features in order to keep the best agility possible, situations may occur when it more effective to group certain similar features together or wait for another feature to complete before starting related ones. These strategic dependencies are usually imposed by the architect and / or the project manager in order to improve delivery efficiency. These dependencies have to be taken into account when organizing the feature pool.

Pooling Rule #3: Feature Aging

For how long as this feature been in the pool? This is important that features gain importance as they stay for a long time in the pool to ensure that they aren't always preempted by new ones, usually always more important. Old features should grow in importance for every iteration they are in the pool. Low priority feature entered 4 iterations ago may well become critical as time advance. If feature priority is stagnant, the project manager should question the reasons and maybe drop this feature altogether to clean-up the pool.

Pooling Rule #4: Sub-system trend

While HVDD strive to develop features in silo, situations may happen when a customer wants the completion of a certain sub-system before another one. At this moment, all features assigned to this sub-system have a higher priority than others from other sub-systems. This priority may come from external constraints or from simple common sense when a sub-system is nearly completed.

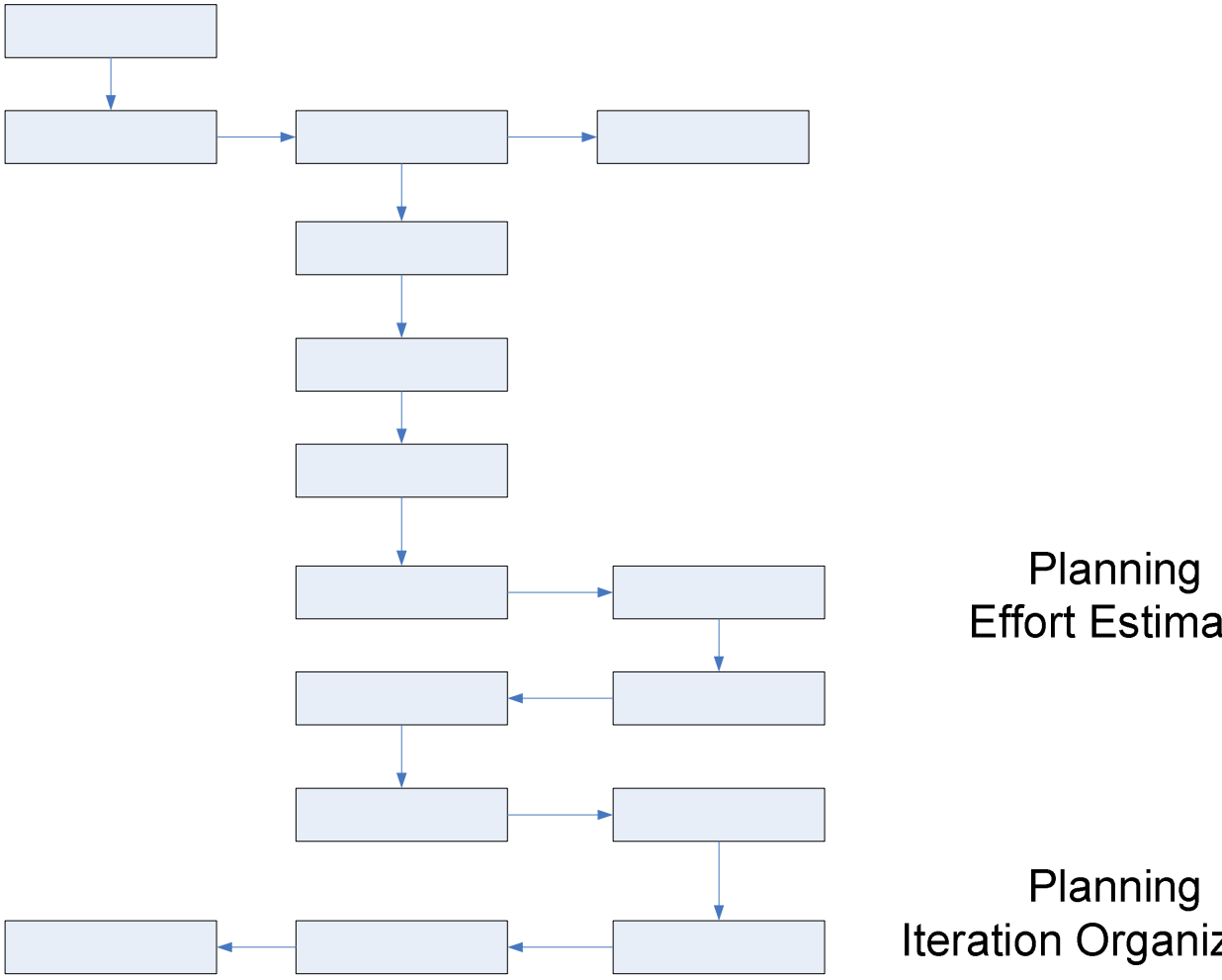
Pooling Rule #5: Architectural Support

Architectural support is the capacity of the system of cost-effectively support the required feature. Some features may be requested by customer through the analyst that may not be easily supported by the current system architecture. Usually, instead of creating a large feature (with more effort assigned to it), an architecture task is registered and the architect (or someone appointed by him) perform preparation work in order to receive this new feature. As long as this work is not completed, the feature may not be cost-effectively implemented. Customer may ask for it anyway, but this may result in additional delays and integration problems.

Realization

This phase is where most of the effort is spend during the development of a system. During this phase, features get organized in iterations and implemented using the TDD approach explained earlier in the document,

The following diagram describes the sequence of all realization activities.



Realization iterations usually involve the whole development team.

Planning

Planning occurs during each iteration and usually targets the next two iterations. If the process is followed correctly, the current iteration is already planned (frozen), the next iteration is also planned and the current planning activities are for the following iteration, which should be opened to changes. While this process may work with a single frozen iteration, it is recommended to get a additional buffer iteration in order to ensure that requirement gathering and analysis problems will not slow the development team down. It may happens that customers have to perform multiple meetings in order to get requirements clear and these delays may force the development team to change their plans or to slow down if the pool is empty.

The two activities associated with planning are effort and iteration organization.

Estimating feature effort

The HVDD process being risk-based, estimates are critical to its success. Estimating the effort to deliver a feature must become a simple activity as the project manager may have to estimate hundreds even thousands of feature in a given project. Luckily, features usually fits in one or six complexity categories numbered 0 to 5. Features assigned to category 0 are not implemented (pending features). Features of category 5 are the most complex and will take much more time than a feature of category 3 or 4. Features are evaluated in an exponential scale where each bigger category is about twice as long as the previous one. Exact time associated with each category is left to each project manager as it heavily depends on the attributes of the team associated to the project.

In addition to this scale, risks factor have to be assigned to each feature depending of the quality of the knowledge available at planning time. Risks factors are usually percentages applied to the scale to increase the estimated effort.

Total estimation should include all activities executed for a given feature (as listed in the previous diagram). The following guidelines have been studied in various projects using the HVDD process.

Activity	Ratio
Project Management	**3%**
Analysis	**9%**
Architecture	**3%**
Design	**25%**
Development	**55%**
Integration	**5%**

Iteration Planning

Iteration planning is the activity of taking a number of features and assigning them to a team to fill-up fixed-length iterations. Planning depends on three factors, the current macro process phase, the iteration budget and the number of features (velocity) required by the customer. The planner is responsible for providing an iteration plan stating which features are to be implemented and the budget required for the iteration.

Macro process dependencies

The macro process affects the composition of an iteration. For example, early iterations performed during feature discovery phase might consist of analysis and architecture with a little prototyping. They might involve only onsite resources or a small offsite team to help in putting in place a prototype. On the other hand, evolution iterations might include a certain amount of maintenance activities (defects) in addition to the normal evolution (improvement) work. The planner must keep track of the macro status of the iteration before performing the planning.

Budget Driven Planning

Customers are usually budget driven. They want the maximum number of features in an iteration for a maximum amount. The planner must take the budget into consideration

when constructing the iteration and must try to exploit the right resource to improve efficiency and thus increase the team velocity.

Feature Driven Planning

On some occasions, marketing is driving the project and some features were promised to their end-users for a certain date. This situation is where the planner tries to increase the team to fit the required feature in the fixed-length iteration. This approach is a little bit more complex as team as to scale up and down based on customer requirements. This approach should be used sparely because of its lack of cost control mechanisms and the pressure put on the development team.

Analysis

Analysis is the activity where feature are defined with enough formalism to be sent to design and development. Analysis role is to establish feature boundaries and make sure that all requirements are well understood by designers. The analyst is also responsible for making available background documentation on the project that provides additional information on the context in which the features are developed.

Feature Analysis

All features are described in an analysis document (see appendix A) containing various mandatory and optional sections. Here's a brief summary of the major ones.

Description

All features must be fully described in order for all team members to understand the nature of the feature, its interest to the customer and its place in the whole system. This description is critical to provide common vocabulary to everybody and to get a shared understanding of the feature.

Business Rules

Business rules are usually defined by the business environment of the customer. They may be generated by regulations, by market forces or simply by common senses. Business rules include all types of validation, routing and special algorithms.

Success Criteria Identification

Success criteria are the goal of the team when implementing a feature. They state the winning conditions that must happen to get a functional feature for the customer. These criteria must be clear and concise to ensure that they are well understood by all team members working to deliver the feature.

Feature Sign-Off

All features must be signed-off by the customer to ensure that the analyst has a good understanding of their requirements and to make sure that the customer is aware of the final feature set that will be implemented. Features may sometimes be quite different from the initial customer request due to decomposition and architectural constraints. The

sign-off give a chance to the customer to understand what is going on an to stop the process if something goes wrong.

Feature Review

This activity occurs after (or during) a feature implementation. The analyst is responsible for the functional side of the feature. Reviewing the implementation is critical to ensure that all success criteria were taken into account and also to make sure all identified business rules have been covered and are handled correctly by the feature.
A failed review means that the feature may not be delivered to the customer and extra time must be spent on it to ensure conformance with specifications. This is why analysts should try to have a look on features during development to limit the extra effort and correct the approach as it goes.

Design

Design is the angular stone of the HVDD process. Designers have many responsibilities and, while they are supported by the analyst, project manager and architect, they are the central point of communication between all team members. Designers are responsible for creating a technical solution that satisfies all business requirements and that fits inside the prescribed architecture. Doing so, they are accountable for the technical implementation of the features they design, even if they don't completely control development. They are responsible for providing enough documentation (risk-based) to support development and to provide punctual help to developers when needs arise.
They have the latitude to select the kind of documentation that produces for any given feature based on the risks assessed for its development. They usually provides code stubs (see below) and some UML diagrams to guide developers in their work.

Analysis Q&A

The first task of a designer is to make sure he understands correctly the feature he is about to design. For this, he needs to read and gets answers to any pending questions he may have. For cost-effectiveness, this activity is usually groups for many features or in one single meeting at the beginning of the iteration. The mailing list is also a good communication channel for these questions as their answers are accessible to all team members.

Feature Design

Designing a feature is crafting a technical solution that complies with the customer needs and with the system architecture. Designers use design patterns and their understanding of what reusable in the architecture to quickly build an idea of their implementation strategic for any given feature. This activity is closer to the "napkin" design than to the full modeling activity. The designers just need to provide enough documentation to ensure a quality implementation and minimal functional and technical defects. Knowing what is enough comes with experience. To make sure that their designs are correct, they may ask the help of the architect who is in charge of reviewing all designs for their technical qualities. Architects have usually more experience and will guide designers in describing their solutions.

Unit Test Design

After the designer has defined his implementation strategic, in the form of a small set of UML diagrams or a simple scanned drawing, tests have to be designed. The designer must provides two types of tests: functional tests which will cover success criteria and business rules and help in reviewing features later on and implementation unit tests where all components of the feature solution are tested separately (unit tests).
These tests aren't implemented by the designer though some designers prefer to provide code stubs for them to ensure good signature and architecture compatibility.

Code Stub

Code stubs are the preferred way to communicate design to developers. A stub is a skeletal implementation of all components of the technical solutions with dummy implementation to ensure there are no compile time errors but not functional neither. Stubs are usually filled with comments about the actual work to be performed, notes on reusability, notes on similar codes elsewhere in the system, references to external diagrams when required. Code stubs reduce the communication burden and improve team efficiency. Developers get to "fill the blanks" and provide additional support classes that may not have been stubbed by the designers.
Not all features need to get full code stubs and again, risk-driven development must be employed in order to create the right amount of stub code. Sometimes, it's better to refer developers to a sample and explain the changes than to recreate a new stub.

Code Review / Corrections

The designer, being responsible for the whole feature, must review the feature implementation to ensure that it's up to his expectations and project quality level. All unit tests must be run successfully and selective code review, of the most critical part of the solution, should be reviewed for form, correctness and performance. Any problems encountered are the responsibility of the designer, which might select to have them resolved by the developer or by himself if it's more effective.

Development

Development is the actual coding of the selected technical solution for a feature using a variant of the TDD approach.

Developers must first code one unit test as specified by the designer. This test will fails as the feature is not yet implemented. As TDD states, feature code must then be added to ensure that the test is working correctly and new tests must also be added if additional variants are discovered by the developer.

This development approach ensures that minimal code is created in order for the feature to be functional (bring value to the customer). It also creates a "test net" that will be used throughout the rest of the process, during integration and feature review.

Integration

As all features are developed independently to the sake of simplicity, there must be a way to link them together, unify implementation and make sure performance and security are uniform. This is all performed during the integration activity. The integrator is responsible for performing cross-feature refactoring to clean class hierarchy and perform code optimization and guidelines review.

Feature Integration

Feature integration is about finding similarities in multiple features and either join them by creating dependencies between features or promote the common code to the architecture for future reuse. Architectural promotion can only be performed by the architect. That's why integration is often performed by or with the architect. Refactoring is an agile development approach introduced a few years ago which list a number of operations that may be applied to source code in order to improve readability, maintainability and performance.

Coding Guideline reviews

Coding guidelines is part of the mechanisms employed to improve communication within the team. By working on code with the same syntax, developers don't loose time figuring special syntax variants. Coding guidelines may vary from one project to the other (when defined by the customer) but should be consistent within a single project life time.
The integrator is responsible for doing spot checks on programming guidelines and notifies the team when they aren't followed.

Performance tuning

In the process, designers are focusing on providing simple solutions to each and every feature the design. While the simplest solution is usually the best (see agile concepts), it is sometimes not the most efficient. When performance is a requirement, the integrator must invest some time in finding bottlenecks and modify the solution to improve its performance.
When performance is stated in the analysis document (usually as a success criteria), designers should try to comply with this requirement. The architect will usually help in finding the best solution at design-time. But for feature without specific performance-related success criteria, it may happen that one become too slow because of a multitude of reasons. That's when the integrator comes in play and improve the overall feature (or system) performance.

Security auditing

Security is usually at the forefront of most information system. The architecture will have integrated security mechanisms and designers will usually integrate them in their solutions, there is still an overall system review to perform in order to make sure that no breach has been created using the silo development model. The integrator is responsible for performing security tests and to make sure that all features are secure.

Operation

This phase includes iterations targeting the transfer of the system to production. It covers all steps from internal development releases, user acceptance releases to full-production environment. It also covers the installation of hardware and software that will host the system, configuration of various appliances, network segments, databases, etc. It also includes system packaging and configuration activities.

Deployment

Deployment groups all activities related to the packaging and configuration of the system for development releases, user acceptance releases or production releases.

System packaging

System packaging is performed by the deployer with the help of the architect and/or integrator. Packaging vary greatly depending on the technology that is used to develop the system but usually includes putting all components together in a form understood by the host environment and creating install scripts.

System configuration

System configuration includes the configuration of the system itself, which is the tuning of various configuration file to a specific environment and also the configuration of the host environment, including application servers, database servers, mail servers and any other infrastructure component available.

Evolution

This macro phase includes all support activities for a production environment. It is still important to put in place an iterative model in order to structure the delivery of defects and their integration into production through downtime windows.
Evolution also includes activities that will generate new requirements and so, provides inputs for the feature discovery phase. Evolution is usually integrated with the Covansys Hybrid Maintenance approach.

Appendix A: Analysis Document Template

Brief Description

Put a concise description of the feature here

Feature

Provides all specification of the feature here.

Special Requirements

Provide dependencies or similarities to other features here.
Provide external requirements like performance and security here

Preconditions

Conditions that must be true before this feature can be used or accessible in the system. Maybe to have been authentication or being part of a specific role. Or to have a specific object instance in the HTTP session.

Post conditions

Indicates conditions that must be true when the feature exits. List database state after, session state, etc.

Extension points

Capacity of the feature to be extended by other features. List all extension points, their API, etc.

Notes

Important notes that a general to the feature. May be background information to get a better understanding of the feature or algorithms used in validation.

Validation Rules

List all validation rules, in bullet point form, for this feature.

Success criteria

List all success criteria, in bullet point form, for this feature.

Appendix B: Bibliography

Document	Author	Reference
Agile Software Development	Alistair Cockburn	**ISBN**: 0-201-69969-9
Agile Modeling	Scott Ambler	
Test Driven Development by example	Kent Beck	**ISBN:** 0321146530
Feature-Driven Development	Jef De Luca	**www.featuredrivendevelopment.com**

