

Digital Notes



**AGILE SOFTWARE DEVELOPMENT**

**UNIT – IV**

**Feature-Driven**

**Development**



**Feature Driven Development**

* Introduction
* Incremental software development
* Regaining Control - The motivation behind FDD
* Planning an iterative project
* Architecture centric
* FDD and XP

**Introduction**

* Planning, managing and monitoring projects that are agile, adaptive and incremental can be very difficult. As was illustrated, although many of the ideas behind methods such as XP can, and indeed have been very successfully applied, it does not mean that it is easy or that they are particularly scalable.
* Adaptive, iterative projects are more complex to control, and to plan, than more traditional, linear, waterfall models (partly because they reflect reality but more on this later).
* In the linear model, life is simpler, no design starts until all analysis has been completed, in turn no implementation starts until all the design is finished. Thus, at any one time, it should be very clear what is being done, by whom and why
* In addition, the requirements are fixed back at the start of the whole project, making things a lot simpler for the poor developer. Of course, the reality is, that not only may the requirements have been wrong in the first place or they may have missed some important behavior
* But from the perspective of managing the project, many of the variables have been removed in a linear project, so it is easier to plan
* The additional complexity facing the project manager of an agile, adaptive and incremental project is an inevitable consequence of the acknowledge that the real world has many variables in it that can change. It is also a consequence of the parallel nature and dependencies between multiple iterations
* One solution to control the complexity inherent in agile, incremental projects is to apply a feature-centric process. A feature-centric process is one that tries to provide a way for management to handle questions such as:
  + What must we do next to add value to the client?
  + How are we progressing against time and budgets?
  + What issues and risks does the project face?
  + How can the issues and risks be addressed or mitigated?
  + What should we do next?



* Feature-centric processes do this while retaining the motivations behind the agile movement such as:
  + Individuals and interactions over processes and tools.
  + Working software over comprehensive documentation.
  + Customer collaboration over contract negotiation.
  + Responding to change over following a plan.
* With the key aims being:
  + To satisfy the customer.
  + Deliver working software that adds value to the customer.
  + Working software is the primary measure of progress.
  + Promote sustainable development.
  + Keep any process as simple as is reasonably possible.
* The term feature-centric refers to development processes that attempt to focus on combining the units of requirements, with the units of planning and the units of work. This allows:
  + the things users want,
  + to be planned for and monitored,
  + and to be used as the basis of work allocation

**Incremental software development**

* An incremental software development process is one that does not try to complete the whole design task in one go. This is in contrast to the more traditional waterfall model of software development.
* One of the features of the waterfall model of software engineering used by many design methods (see Figure 1) is that it primarily assumes that you will complete the requirements analysis before you start the design phase.
* In turn, you will complete the design phase before you start the implementation phase, and so on. It does accept that there may be some feedback of information from one phase to any preceding phases and that this feedback may impact upon the products of the preceding phases.
* However, this is a secondary issue and the assumption is that you will be able to complete the vast majority of one phase before ever considering the next phase.
* When applying some form of iterative approach, the intention is that each iteration adds something to the evolving system. Some iterations may lead to a release of the software system, while others may not. Each iteration:

1. Determines what will be done during the iteration.

2. Designs and implements the required functionality.



3. Tests the new functionality.

4. (Optionally) Creates a new release.

5. Reviews what has been done before moving to the next iteration

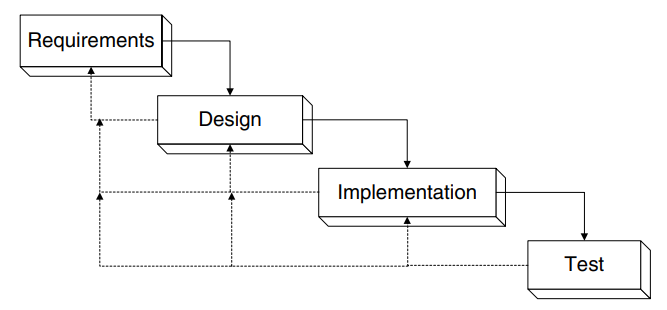


Fig 1 – The Waterfall Model

* Figure 2 depicts the spiral nature of this approach to software development. Note that in effect, each iteration around the spiral is a mini-software development project.
* The end result is that you incrementally produce the system being designed. While you do this, you explicitly identify the risks to your design/system upfront and deal with them early on.
* Note that this neither means that you are hacking the system together nor carrying out some form of rapid prototyping.
* However, it does mean that a great deal of planning is required, both upfront and as the design develops.

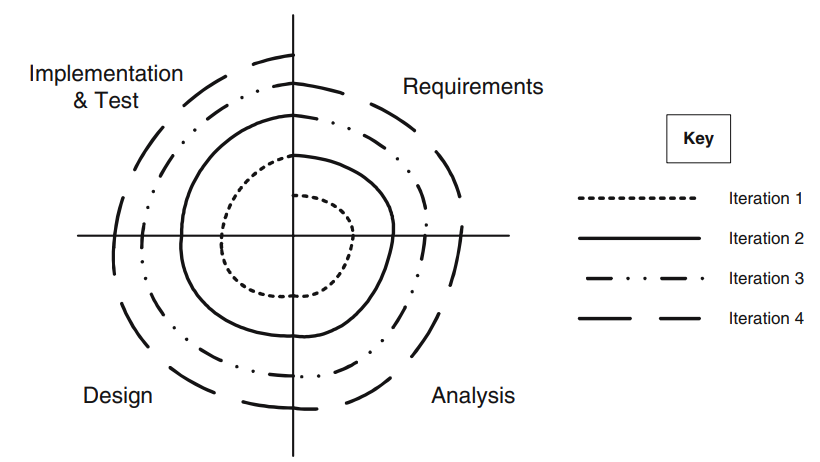


Fig 2 – The Spiral Model of software development



**Regaining Control - The motivation behind FDD**

* An important aspect to address at this point is the potential explosion in planning effort that may be required to deal with the iterative lifecycle model that is being described here.
* It is certainly more complex than a linear waterfall lifecycle to plan and manage.
* However, given that our goal is to simplify the lifecycle in order that we can deal with the risks and complexities as well as uncertainties of the development process, we need to regain some control of the planning and management aspects of such a project.
* Feature-centric is not the only aspect of regaining control of an iterative project; another feature is that of timeboxing each iteration. The final aspect is being adaptive. Thus, to regain control of an iterative project the guidelines are:
  + **The process should be feature-centric**. This means that the units of requirements (e.g., use cases, user stories) should be unified with the units of planning (e.g., work packages and tasks).
  + **Project planning should be based around timeboxes** (rather than phases) so that the length of each iteration is known.
  + **The project plan should be adaptive** that is responsive to the changing risks and benefits of the system and business environment.
* **Feature-Centric Development**
  + The term feature-centric is used to refer to development processes that combine the expression of requirements with the units of activity for planning purposes.
  + A feature in such a process can be viewed as a unit of “plannable functionality.”
  + A feature is a schedulable piece of functionality, something that delivers value to the user.
  + A feature is derived from a planning perspective rather than from the user perspective or indeed the requirements perspective. This is an important distinction and why features differ from requirements, user stories or use cases.
  + To aid in planning, features go further, they must also be associated with:
    - a priority (so that they can be ordered),
    - a cost (so that they can be accounted for),
    - resources (so that they can be scheduled).
  + Costs and resources can be determined by examining the number of person days taken to accomplish the feature.
  + Priority can be harder to determine but should take into account:
    - Architectural importance of the feature.
    - Utility to the user.
    - Risk involved.
    - Requirements of the system/use cases.
  + In some cases, a user story or requirement might relate directly to a single feature, or to many features. In turn, a feature might relate to a single requirement.
  + However, it is also possible for a feature to support many requirements



* + So, features are Not necessarily driven from user stories/requirements/use cases?
  + There are other sources of information which can lead to features including (but not limited to):
    - Bug fixes.
    - Maintenance enhancements (for example, due to changing to a new version of a language or operating system).
    - General tidying operations.
* **Timeboxing Iterations**
  + The emphasis in most formal development processes is on the phases a project goes through and then on the steps within the phases that may or may not be carried out iteratively.
  + This is true of modern processes such as the standard Unified Process as much as of more traditional processes such as the waterfall process
  + However, with iterative processes, although the details of the current iteration may be known, details of the next or subsequent iterations are less clear. Indeed, as the aim of this approach is to allow an iteration to be planned, at the start of the iteration, the time taken for an iteration may only become clear at the start of that iteration.
  + This is not very good for budget planning, for release planning or indeed for management of the project. It certainly has issues with fixed release dates.
  + There is therefore a conflict between the flexible and responsive nature of an iterative approach and the constraints of budgets and timescales.
  + This is where timeboxing comes in. Rather than defining each iteration by the features it will implement, it is possible to define an iteration in terms of the time period it will take and the ordered list of features that will be attempted during the time period.
  + Features that are lower down the priority list will only be attempted if time allows, otherwise they will be relegated to a later iteration.
  + Timeboxing iteration has a number of benefits including the ability to:
    - schedule and plan for incremental releases of the software,
    - schedule and implement features,
    - manage budgets,
    - monitor progress within fixed time constraints
  + All within a flexible and responsive process.
  + What this also means is that regular reviews (typically on a weekly basis) are required to consider how features and tasks are progressing, which (if any) are behind schedule and why.
  + It is also necessary to consider what the impact on the project will be, both from a technical and a business point of view. This may then lead to modification of the work packages and features to be included in the current iteration.
  + This does not necessarily mean that the plan is revised weekly, rather that the implications are considered and appropriate action taken if necessary



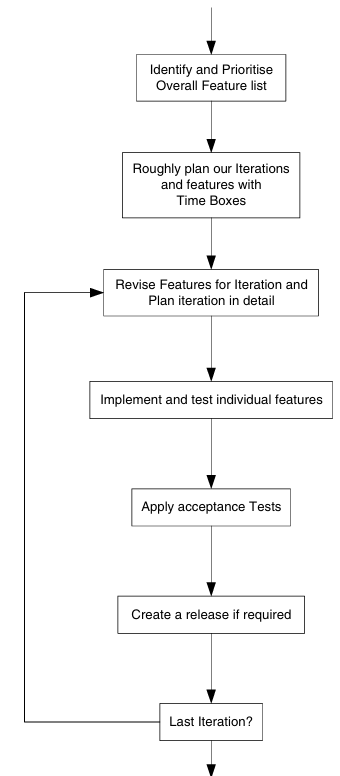
* **Being Adaptive but Managed**
  + What is required is a management process that is flexible enough to deal with the changing requirements of the business and users, and to deal with the emerging uncertainties.
  + It also needs to be one in which we can still monitor progress, determine resources, ensure quality and guarantee delivery.
  + Most traditional management styles set out what will be done when and for how long right at the start of a project and well before detailed design and implementation has begun.
  + The project is then measured against these estimates with little or no room for change. However, an iterative project explicitly acknowledges the need for change and the need for ongoing management.
  + To this end, an iterative project is effectively planned and re-planned at each stage of the spiral presented back in Figure 2.
  + There is an overall planning activity before the whole process starts and then there are planning activities at the start of each iteration.
  + In addition, regular (weekly) reviews may also affect the current plan for an iteration
  + In terms of management monitoring of project activity, person days for tasks should be monitored (on a weekly basis) and fed into the project plan to determine how the project is progressing relative to the planned effort.
  + However, due to issues such as holidays, sick leave, etc. It is also necessary to compare the current progress in elapsed time with the project plan.

**Planning an iterative project**

* Before any project embarks on an iteration development process there are a number of steps that should be followed. These steps are:
  + Identify and prioritize features (the feature list should be continually revised throughout the project).
  + Roughly identify iterations and allocate features.
  + Time box iterations/calculate costs.
  + For each iteration
    - Plan iterations (which should be continually revised during lifetime of project).
    - Identify tasks required to implement features.
    - Allocate tasks to resources (that is, allocate tasks to project members).
    - Implement iteration.
* The key here is that iterations are based on “timeboxes” so that their length is known and can be managed.
* Iterations are also based on tasks constructed around features so that they can be responsive to user feedback and to changing business requirements.



* Note that depending upon the size and complexity of the tasks, we can at times group related tasks together to form packages of work that make high level project planning easier.
* However, for smaller features, we tend to always work directly at the task level.
* **Iterations, Timeboxes and Releases**
  + This includes reviewing the overall project planning process.





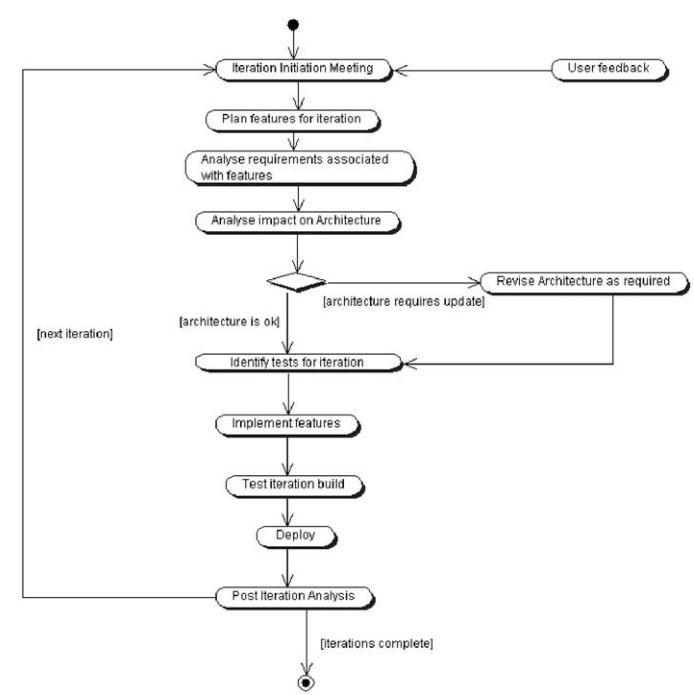
* + At the start of the project, the project team along with various project stakeholders create a prioritized feature list.
  + Note that this cannot be done without the collaboration of those project stakeholders who can state what the priorities of the features should be.
  + Thus, as with XP, the “business” is an essential part of this process.
  + In general, giving features priorities such as High, Medium and Low is enough
  + Associated with the features at this stage is a cost.
  + A cost is related to how many person days it will take to implement the feature. As we are only roughly estimating at this point, and typically we use a “Three-Point” estimation approach but essentially this requires a best estimate, an average estimate and a worst-case estimate to be given.
  + The overall cost is derived from these three estimates. Finally, the number of software engineers involved with the feature is also estimated. We now have our initial feature list
  + Next, we try to determine how many iterations we expect to have, how long the iterations will last and which features will be in the iterations. This can only be done with the involvement of those business representatives who have the knowledge and authority to agree to the timescales being discussed.
  + In general, we have found that this usually involves a series of meetings with the client representatives, during which exact timescales for timeboxes, features for iterations, etc. are agreed. In general, timeboxes should not change, but the features implemented within the iterations defined by the timeboxes may
  + From this, we emerge with an outline plan for what will be done when and at what point we will be completing various iterations of the end system.
* **Planning an Iteration**
* Each iteration will be comprised of a similar set of steps. These steps are presented graphically in Flowchart. The key steps in any iteration are:

1. **Iteration initiation meeting**. The length of the iteration should already have been determined butmay be revised at this point. The features to be addressed in this iteration should be revised and confirmed along with the resources to be applied, etc. This meeting should involve all stakeholders in the project
2. **Plan features for iteration**. Having agreed the features to be addressed, a detailed plan should be produced mapping features to work packages and work packages to tasks. The tasks in turn should be allocated to actual resources, etc. This plan must be accepted by the key stakeholders (including the clients).
3. **Analyse the requirements associated with the features**. This may involve writing or revising a use case document, designing new GUI displays, determining the user interaction sequence, etc. The acceptance criteria for this iteration should also be identified and agreed. 
4. **Analyse impact on the architecture**. The architecture is the backbone upon which the iterative process operates, therefore the next step to perform is to examine the impact any new features are likely to have on the architecture. It may also involve identifying new architecturally significant entities that feed into the next step.



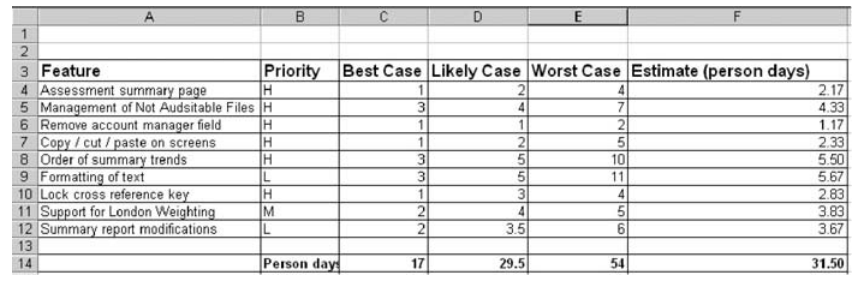
1. **Revise architecture as required**. This step involves revisiting and amending the application architecture in response to the features required by this implementation. Note that this may mean that some of the design and analysis work associated with core features may be performed at this stage to determine their architectural impact.
2. **Next a new acceptance test plan and specification should be written for this iteration**. Note this document may not include all tests as some features may only be implemented if time allows. The specification of the tests for these features should be considered to be a task within the work package that will address that feature. Also, note the difference here between the iteration test specification and the JUnit tests that might be written as part of a feature. The test specification is oriented towards the system as a whole (its overall operation) whereas JUnit tests tend to be oriented towards individual class, subsystems or systems. There is therefore a major difference of focus – both are required.
3. **The next step involves implementing the features**. The features are actually implemented via tasks that should be monitored as normal (although reference should be made to the timebox of the iteration). It is recommended that each feature should have a set of associated unit tests that must be passed before the feature is taken to have been completed. These unit tests should be part of the code released for the feature and ideally should be added to a unit test framework (such as JUnit) that can easily be re-run at regular intervals (such as every time a new build of the system occurs).
4. **Once the features are implemented, the new system should be tested** (this includes the generation of a test report). This includes unit tests and acceptance tests. All tests should pass before the iteration is allowed to proceed. If any tests fail, then the release cannot be deployed and the problems must be corrected. If earlier steps have been adhered to, the unit tests should pass and thus it should only be the acceptance tests that cause a problem. At this point, the project needs to determine why the system/acceptance tests have failed. If this is because of some features that have been moved to a subsequent iteration, then the acceptance tests need to be changed. If it is due to some missing or erroneous functionality then this functionality to be removed (in order to release on time) or if it must be included then the problems must be addressed. In general, time should be allocated to this process to allow for such unforeseen problems (as with the best will in the world they will occur).
5. **The new application should then be deployed** to the client who should then perform any agreed user acceptance tests. This may lead to the revision of the deployed system, if and when deficiencies are identified.
6. **A post iteration meeting should review** the progress made during the iteration, it should consider any issues that arose and re-prioritize any features that were not addressed. Again, this should involve all project stakeholders.
7. At this point, a decision should also be made regarding the validity of the next iteration and whether any further iterations are required.

REFER THE FLOWCHART BELOW FOR THIS TOPIC

* **Estimating the Cost of a Feature**
* The approach that has taken over the last few years towards estimating the cost of features involves applying three-point estimating. Three-point estimating involves producing three estimates of the effort that will be required to implement a feature. These are:
  + The best-case scenario. This represents the situations where everything goes as well as it could and there are absolutely no surprises or problems encountered.
  + The best guess at what it will really take. This represents the situation where mostly everything goes okay, but one or two unexpected situations occur which take a little longer than originally expected to handle.
  + The worst-case scenario. This represents the situation where major issues were overlooked (because they were not obvious until implementation started).
* These three estimates acknowledge the difficulty of estimating how long something will take to implement before you actually implement it. It also reflects the fact that some people are more optimistic and some more pessimistic than others. And, thus a range of estimates can capture their different views.

* A table illustrating this style of estimating for a list of features is illustrated in Figure below. The table illustrates the feature, its priority and the three estimates for best, most likely and worst-case scenarios

****

* **Architecture centric**
* **Why Architecture Centric?**
  + Feature-driven development orfeature-centric development (FCD)is onlyfeasible (and successful) if there is a solid architecture on which each iteration can be built.
* **Architecture Defined**
  + One problem with an iterative and incremental approach is that if no order or structure was defined for the application it could (would?) grow more and more unwieldy andmore andmore dis-organised as each iteration progressed
  + To ensure that all the various parts fit together, there needs to be something. That something is the architecture.
  + A good architecture will be resilient to change and to the evolving design and implementation. The Unified Process explicitly acknowledges the need for this architecture by being architecture centric
  + It describes how you identify what should be the part of the architecture and how you go about designing and implementing the architecture. The remainder of the Unified Process then refers back to that architecture.
* **Why Have an Architecture?**

*understand the system*. Software systems can be large, complex andmust meetconflicting requirements. An architecture provides a convenient blueprint or model of the system to be produced. It abstracts out much of the implementation detail, but “positions” the elements that must meet the various functional requirements.

*organise development*. It helps organise “plumbers” and “electricians.” That is, it helps firstly to separate out different concerns so that those involved in the “plumbing” of the system only need to worry about plumbing issues. However, it also identifies how they are related, so that the points at which different concerns intersect, are well documented and clearly specified (for example, in the central heating boiler).

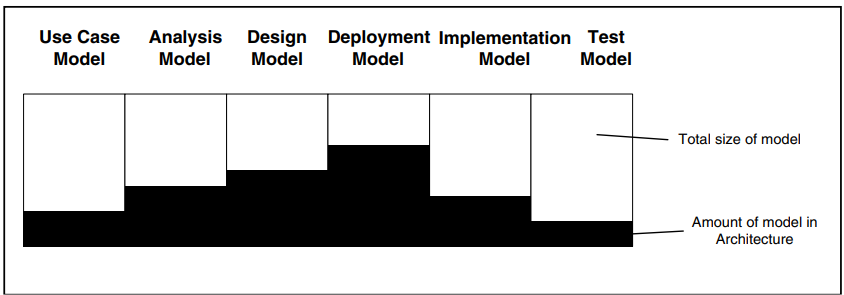
*promote reuse*. The problem with writing reusable code is that you need to identify that what you are producing is reusable. I have personally been in situations where two people on one project are reproducing the same solution but from different aspects. In at least one case, they were sitting opposite to each other. It is certainly easier to produce reusable code the second, third or even fourth time you are designing and implementing a system than the first. Indeed, in many systems, the only form of reuse that occurs is at the class level, i.e. at a very detailed level and is identified by the coder during implementation.However, an architecture can help at amuch higher level by identifying critical systems and subsystems early on. Common subsystems can then be made reusable.

*promote continued development*. Few systems of any size or consequence are produced and never altered. Instead, it is much more common for a system to evolve over time with new requirements being identified and new functionality added or existing functionality modified. The original architecture can be essential in helping to control the evolution of the system over time. Indeed, a good architecture need change little over the lifecycle of a system but can be instrumental to the success of future releases. This is because it provides the overall structure into which the new additions or modifications must be fitted. Often, the actual design of the system is too detailed to allow an overview to be gained, and thus, future designers and implementers may misinterpret part of a design or (worse) ignore it. The architecture can help to minimise such problems

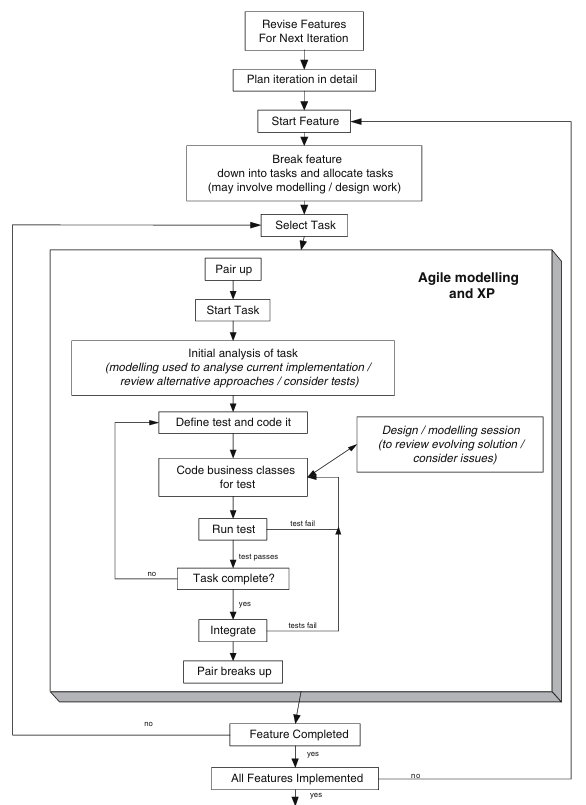
* **Architecture Myths**
  + For a start, it is important to realise that the architecture and the design are not the same thing but it is important to re-iterate this.
  + The architecture highlights the most significant elements of the design. These include the major systems and subsystems, their interfaces, how the system will be deployed, etc. It does not include many details of the systems and subsystems and how they are implemented – that is the job of the design
  + It is useful to picture the level of detail in the architecture and the level of detail in the final design
  + As can be seen from this diagram, the architecture leaves much out, while the design must address many more aspects in detail.
  + Another myth to be debunked is that the architecture and the infrastructure are the same thing. This is an easy mistake to make (not least given what we have said about the role of the architecture). However, it is important to remember that the architecture only captures those elements of the design that are necessary to provide an understanding of the overall design of the system







* **FDD and XP**
* Till now we have focused almost exclusively on planning an iterative project and how FDD can help with this. But what about modelling your solution or implementing that solution? This is where agile methods such as Agile Modelling and eXtreme Programming come in.
* Feature-driven development provides a way of controlling the iterative and incremental nature of agile projects. It does not really have anything to say about how you implement those features.
* Within a development model in which FDD is used to plan the details of iterations and in which features are treated as the tasks to be performed, then applying Agile Modelling and XP practices can result in a workflow resembling that presented in below Figure









* Note that within this approach, we are using Agile Modelling to allow any modelling activities to take place and XP practices to implement the required behavior. Also, note that we are assuming here an explicit analysis step that involves some design and/or modelling work in order to determine how the feature should be implemented or broken down into tasks
* An important point to note, and one that might mean that hardened XP developers will say that we are not doing XP is that we are not applying the planning game. This is because the planning game is effectively subsumed by the role played by the feature-driven development process itself.
* A final point with regard to the application of XP practices is that the remaining XP practices are still relevant and still applicable within an FDD planned project. Whether you apply them or not is to some extent an issue for the particular project at hand
* **Planning a Sample FDD Project**
* **The Overall Project Plan**
  + The end result of the initial project meetings was a document summarizing the features to be implemented, their priority, cost and resources required, framed within the context of a set of fixed time iterations
  + This feature proved important to the business in a number of ways. Firstly, it specified the maximum cost for an iteration
* **Planning the First Iteration**
  + Selecting Features for Iteration 1
  + Features are selected based on a number of criteria:

1. Importance to the business

2. Level of risk

3. Application requirement

* **Feature to Task Mapping**
  + Features represent schedulable requirements and the activity that will realize those requirements.
  + However, there are two issues, with directly allocating features to designers/developers to work on:

1. Typically, they represent large scale “functionality” that would be difficult to monitor except at the highest of levels.

2. Typically, they cut across multiple layers in the architecture requiring modifications to low-level, back-end frameworks as much as to front-end GUI components. These different areas require different knowledge and skill sets that are rarely possessed by a single individual.

* **The Gantt Chart for Iteration 1**
  + The following Gantt chart provides a detailed breakdown of the tasks to be performed for iteration 1 of the project. Note that the “Person Days” column relates to the number of person days spent on the task, whereas the “Elapsed Duration” column relates to the elapsed time spent on the task.
  + Thus, a task of 12 person days, worked on by three people would have an elapsed duration of 4 days. Equally, an ongoing management task has an elapsed time of 65 days, but the number of person days spent on this task is 13
  + In the Gantt chart presented in Fig. some buffer has been left within this plan for unscheduled events (such as holidays and illness). We have not tried to have everyone utilizing 100% of the time. We believe that this reflects reality; as well as allowing developers time to expand their experience. However, in the context of an agile development, it also reflects the need to allow for pair programming time.



* “In this case, pair programming was not used as a standard way of developing” at the time you may have thought this was an odd way. This is because the development team was encouraged to employ pair programming techniques when they thought it would be to their advantage. For example, when they were concerned that they were straying into unknown territory, or needed to work out a difficult algorithm, etc.
* This approach worked very well but needs to allow time within the schedule for developers to leave the task they are working on and to pair program with a co-worker

