Big Design Up Front

Summary:

The term BigDesignUpFront is commonly used to describe methods of software development where a "big" detailed design is created *before* coding and testing takes place. Several ExtremeProgramming (XP) advocates have said that such "big" designs are not necessary, and that most design should occur throughout the development process. While Xp does have initial design (the SystemMetaphor), it is considered to be a relatively "small" design. Much of this page disputes the amount of up-front design required for software projects.

"every OOA/D begins with identifying the problem space entities that are relevant to solving the problem. Once the relevant entities are identified one identifies the relationships in the problem space between those entities that are relevant to solving the problem in hand. One records both in a Class Diagram. Only then does one move on to identifying detailed responsibilities, dynamics, and flow of control collaborations." - H.S.Lahman (in 2006, folks)

Design as part of the planning process

"In preparing for battle I have always found that plans are useless, but planning is indispensable." [Eisenhower quote taken from PlanningExtremeProgramming]

Or as JerryWeinberg put it "the *documentation* is nothing; the *documenting* is everything".

"A mistake in initial dispositions can seldom be put right." and "No plan survives its first contact with the enemy." [Field Marshal von Moltke] -- JoshStults

Perhaps it is the social and intellectual activity of designing, rather than the designs themselves that are important. Is it the scope of design or the attitude *towards* design that needs to change? -- ChrisSteinbach

---

Some Up Front Design Required?

[This section originally followed the Smalltalk design/prototyping discussion.]

Even in Smalltalk, though, some design has to be done upfront [...]. [What goes on the desktop, number of services and support machines, legacy connections, etc.]

Which of these things can't be decided one at a time, when its turn comes to be implemented?

I can't really stand up in front of a project review team for a $60M call center installation and suggest that they "read the code" in order to understand whether the resulting system will satisfy the very real 3-second response-time constraint.

What makes reading the document more convincing? When our top customer challenged whether our system could be fast enough, we measured it, ran it in 10 parallel processes and measured it again. Did the math. **That** was convincing. -- RonJeffries

Did you document those tests? Make them repeatable? Did you prove that they would scale and the results would scale predictably?

Which of these things can't be decided one at a time, when its turn comes to be implemented?

The ones that have to be funded by the client *before* the subsystem is built. The ones where the client wants to know ahead of time how much the total solution is likely to cost.

What makes reading the document more convincing?

The fact that the document is written in English, and most of the review team (particularly the managers and recommenders whose approval is needed to close the phase and pay the bill) haven't read or written code in 30 years, if at all. Many system problems, such as call centers, don't succumb so readily to simple demonstrations of blazing speed. In my example, the **response time** of a call center is as likely to be affected by network load and the number of network transfers (because of network latency) as the performance of any one system. A data transfer that requires 100 hops at 100 milliseconds per hop is going to burn 10 seconds in transfer time, even if the participating systems are infinitely fast.

In the hardware world, the "design", especially of a complex system, is how you answer questions about how close to critical limits you are, what margins of safety you've built in, how you know that a particular beam is thick enough or cooling element big enough. The same sort of thing has to be done in software, especially in mission- or life-critical applications. And it is NOT all in the source code, it's as much in the interconnections among components. And those *aren't* in the source code. No code is an island.

Design and Architecture

The words BigDesignUpFront do bring to mind something of the aim of much architectural design in the building sense. Good design often follows where Big Ideas precede Detail Design. Conversely it is rare for small scale, pragmatic Detail Design alone to bring forth Great (or even worthy) Architecture. Of course, some Big Ideas are so far removed from the pragmatics as to be spurious as Design generators. As noted in NooHasNothingToDoWithSoftware, the key seems to be to correlate Big Ideas that have a genuine relationship with the item being designed.

BigDesignUpFront in architecture does not require that every aspect of the design is predetermined. However it does provide a powerful reference for evaluation of subsequent ideas in the iterative design process.

-- MartinNoutch

We are perhaps learning the need to be humbler at this stage of software development, Martin. But I guess the nearest parallel in ExtremeProgramming to what you're referring to as good and necessary BigDesignUpFront in physical architecture is the SystemMetaphor. After that greatness comes (if it ever does) through brilliant teamwork on the detail. But see also the unresolved questions in ExternalAndInternalDesign. -- RichardDrake

I'm not sure about the ExtremeProgramming people or the exact meaning of BigDesignUpFront, but a high-level up-front design is generally regarded by most as important to large-scale object-oriented software systems. There's a tremendous amount of quantification to support the position. However, in software, this high-level design is very dynamic and changes as implementation is created. For a long time, one could get away with nothing more than CodeAndTest and Refactoring, but as systems become more complex, a need arises for some consideration to high-level designs and strategies. This is maybe why there is such a rift between SoftwareArchitects and ExtremeProgramming. Extreme Programmers don't agree with this position - and they may be right for the type of apps they develop. But some ExtremeProgramming enthusiasts are even hostile to the idea of SoftwareArchitecture. -- RobertDiFalco

---

Design Documents

I'm clearly the only programmer left who doesn't use Smalltalk at all.

I'm in an environment where I get pieces of a very large system to write. I start out by writing up a little analysis document that I use to ensure that I understand the requirements. I then write up a high level design and check to make sure that it will take care of all the requirements in the analysis. Then I refine the high level design into a low level design. I often write code to do this, sometimes I decide that TheSourceCodeIsTheDesign, other times I UsePrototypesToChallengeAssumptions. I don't, however, allow the design document to omit any information that is needed in order to understand the code. That means that I'm always twiddling the design doc while I'm writing the code. It's not really all that onerous either because I use the design doc as a sort of checklist that tells me when I'm done with the code. If I change my mind about how to do something I change the doc so that I can check off what I've really done. If I didn't do that then I'd have stuff on the doc that never got checked off and I'd have code that got written for no apparent reason. It seems like the same kind of relationship the Extreme guys have with Unit Tests. One result is that the design is definitely not done until the code is. We keep our design docs in revision control and I find that I always check in my designs when I check in my code.

I guess my point is that I believe strongly in BigDesignUpFront - but only for a little part of the whole program at a time. *Wouldn't that be LittleDesignAllAlong?*

An ExtremeProgrammer might write tests to represent what the code had to do, and then code until they ran. What is the documentation cycle adding to the story? If you have the tests, why aren't they sufficient to know when you're done? And if you don't, how do you know when to check something off? -- RonJeffries

How do you test the tests? How do you explain the purpose behind the test? How do you relate the test to the system? Design and documentation are still necessary.

What output, in the end, does one have from the testing process? Is it correct that essentially one has, after the tests have run, simply a linear numbered list such as "test0001 - OK. test0002 - OK. test0003 - failed. test0004 - OK."? Such a list fails to relate the significance of the test to the architecture of the system, in terms the customer can understand. Therefore, it could be argued that what the documentation cycle "adds to the story" is an additional level of abstraction of describing what the system should do in an arbitrarily expressive form (i.e. natural or formal language). However, with additional tools, perhaps the output of the testing process could be raised to a higher level of abstraction - say, by putting tests within a hierarchy, modeling dependencies between tests, etc. However, perhaps exactly this sort of modeling is the type of "Design" that XP claims is unnecessary.

Actually, it helps if the tests are actually named after what they're testing - "testFunctionBlahReturnsTrueForGoodValues - OK, testGettingResultsFromDatabase - OK, testSomeOtherBusinessRule - OK"

SmalltalkAsDesignPrototypeLanguage moved to another page

Other Discussion:

I don't keep up to date on the methodology wars, and it seems everyone's more on a patterns bandwagon of late, but last I recall ~92 the prevailing sensible view (i.e. matching mine) seemed to be some variation on cyclical development (recursive-parallel, iterated "analyze a little, design a little, code a little, test a little", whatever).

Haven't looked at TheDeadline, but the write-up here make it sound as though a lot of experience is being discarded, especially as indicated in the "don't allow implementation until the very last minute" notion.

I'm big on design, compared to many here (to me the work is in design; coding is the process of converting that into a specific computer language and typing it in, and occurs quickly and semi-automatically) but always within that context of a larger cycle. This BigDesignUpFront thing as described seems to be trouble waiting to happen. -- JimPerry

---

What is the problem statement that ExtremeProgramming is a good solution for? What is the problem statement that BigDesignUpFront is a good solution for? I read that Nickolas Tesla pictured a fully designed and working alternating current generator in his head before he even started working on one.

Maybe BigDesignUpFront is the correct solution when you already know everything you need to know and are ever going to know about a system before you start it. Why experiment and iterate if you aren't going to learn something?

Sounds impossible as stated, boring if it were. If I knew everything, wouldn't I know the design and not need BigDesignUpFront? -- r

Maybe BigDesignUpFront is the correct solution when the medium you are working in is incomprehensible once it is built/written (e.g. machine code).

BigDesignUpFront is only appropriate when reached via a moment of satori. --MarkJanssen

No one ever built a machine code program without debugging. We used to do lots of thinking and desk checking not because it was better than debugging, but because it was hard to get time on the machine. -- r

Maybe BigDesignUpFront is the correct solution when the medium you are working in is unchangeable once it is built/written (e.g. Hubble Telescope).

[actually, wasn't there a big problem (focus) when Hubble (the telescope) first went up in 1990? Oh, and since it was only corrected 3 1/2 years later, this is where BDUF comes in...]

Actually the Hubble was built on the ground, and changed in space. I think, though, that the learning point is most important. You will learn: what's the best way in your circumstances? -- RonJeffries

Interestingly, the Hubble's big problem came about because a test was incorrectly set up, and the telescope was built to pass that test. -- DaveVanBuren

-- StanSilver

Off-topic clarification: The Hubble's main defractor was ground based on a faulty reference lens. The design was perfect, but an implementation flaw crept in. The flaw was an aberration of some sort, I think the result of a incorrectly calibrated instrument. In relation to that lens, though, the main Hubble mirror was the most precisely built instrument ever at the time it was created. The application of the aluminum coating, used for ultraviolet reflection, was a big worry (if the aluminum oxidized, it would have been about as reflective as black felt to ultraviolet light), but went off without a hitch because the process was so well designed and understood in advance. The reflectivity exceeded their best hopes! They called the fix "glasses" for the Hubble, but I forget whether it was optical or electronic. There are many web sites devoted to the instrument. I'm not sure whether this anecdote has any application to this discussion. Beware of metaphors! -- Brent

*The Hubble's mirror physically warped in space because of thermal contraction. This is something that could have been tested here, on the ground. This was not an implementation flaw so far as I can see; this was a "what could possibly go wrong?" kind of flaw (not sure what you'd call that - a **process** flaw perhaps?). The fix for this was a hunk of glass ground to a lens with a deliberate* counter-astigmatism. *The basic idea is the same as you find in Schmidt-Cassegrain telescopes: those use a* spherical *mirror, and a front-end lens with a* hyperbolic *form factor to counter the spherical aberration that the primary mirror introduces. As you can see, the fix is aptly named - it* literally *is a set of glasses for the telescope.*

Additional notes to the above clarification, at the risk of drifting further OT:

Additional notes to the above clarification, at the risk of drifting further OT: The corrective lens was optical, and called COSTAR. They were actually two lenses on a single device, that was placed between the Big Lens and space. They defracted the light in such a way that it nullified the imperfection in the main lens (light from the center and light from sides weren't being focussed on the same place, resulting in blur) The problem occurred due to a tiny piece of paint that bladdered off a metal tube, thus creating an imperfection on one of the calibration lenses, results shown above. To bring it back to the main point of this page: I don't think any BDUF-designer could have thought up that something as miniscule as a small bladder of paint would have cause this problem, as it's not in the scope of the Big Design. Had the engineers followed a continuous testing-implementing-testing cycle they would have been more likely to spot such a problem, I think. (Sorry if this is 'Wrong'-ish on this Wiki, first time posting after lots of reading with great interest.) -- SanderBol

---

To continue my last statement: Where is the most efficient and cost effective time and medium to do your experimenting, tinkering, and rearranging?

Maybe the primary purpose (primary use?) of design notations is for tinkering, not for specifying? Maybe the better they are for specifying, the worse they are for tinkering, so the complicated ones don't get used?

If I understand correctly, XPers seem to have come to the conclusion that stories, CRC, units tests, Smalltalk (and Wiki?) are the best phases/mediums for doing their tinkering and rearranging. Others do major tinkering in some DesignPhase/design notation during iterations; still others do major tinkering in a big design phase/design notation at the beginning of a project. -- StanSilver

I can't speak for XPers, but I prefer to do my tinkering and rearranging in a medium that can tell me whether my current tinkering is working. I just don't trust my ability to think through everything - I know from experience that I always overlook something. I do most of my tinkering in code because it lets me know right away when my thinking is flawed. -- JasonArhart

How about UseTheLeastExpensiveMediumInWhichYouCanLearnAsMuchAsIsProductiveAboutTheParticularProblemDomainYouAreExploring? Or maybe, MatchTheCostOfYourMediumToTheLevelOfKnowledgeUncertainty? Hard to make it pithy. Anyway, here is the core of the issue from my perspective: In the phrase "Big Design Up Front", what is meant by the word "Design"? What problem domain are you exploring when you're "tinkering" or "designing" or whatever you want to call it? Are you exploring the essential requirements and user-perceptible form of the product (what should it do?), or are you exploring how to satisfy those requirements with libraries, data structures, and algorithms (how should it do it)? If the former, then code is a *very* expensive medium in which to learn that huge product changes are necessary - and trust me, if you haven't prototyped it, you will learn that huge product changes are necessary. (Try building a prototype with paper, or Powerpoint, or even VB - much cheaper media). If the latter, then by all means start coding - unless you've built the product once already, you're going to need the code to keep you honest, and writing reams of techn specs (that no one will read) will only waste time. (No harm in some interactive white-boarding along the way though.) Obviously, at some point there is a fine line between exploring the product and exploring its implementation ... But there is great efficiency to be gained if you can find where that line is and use it to keep projects from flailing. -- JesseWatson

George Orwell is quoted as saying 'I write to clarify my own thinking.' In the same way, I use design is as a way to clarify my own thinking about the problem and possible solutions. I find that tinkering with the problem in English or pseudo-code helps me realize a good enough design to get started with. It is all about having a reasonable enough initial approach that can be iterated on throughout the development process. -- BryanEdds

---

Another angle (to steal a knowledge representation technique from WhatIsAnalysis page):

- **Group 0**, software development is a trial and error process
- **Group 1**, software development is a series of deterministic transformations

Group 0 tries to partition their trial and error (into different mediums and time intervals) in a way that minimizes cost and time to market.

Group 1 concerns themselves with specifying and capturing the complete and correct output of each transformation phase (since the assumption is that it is correct).

When some of group 1's code isn't accepted by the user, it is a "mistake" - they didn't do one of their transformations correctly. When some of group 0's code isn't accepted by the user, they fix it. -- StanSilver

I think the tendency to do BigDesignUpFront may come from the confusion that non-programmers (I suppose) have: They see coding as manufacturing, not as design. So, quite logically, they want the product to be designed before it's manufactured, like in any other industry. Except that, after the supposed design is done, another design (plus refactorings) must be done by the coders. I suggest that, most of the time, seeing coding as manufacturing is a NotionSmell. -- AnonymousDonor

---

I like that characterization. :-) -- AlistairCockburn

Two comments:

- I'm never sure whether the word `design` means what some thing looks like to the user, or what it looks like to the people creating and/or fixing the same thing.
- The best size of *design up front* surely depends on both the situation, and how fast that situation is changing.

-- DickBotting

'Design' has to be one of the most over-used and mis-used words around at present. Consider whether other words are really meant, e.g. 'make', 'write' etc. For me, 'Design' encompasses a wide ranging process of investigation, evaluation, problem-solving and creative decision-making. Thoughtful analysis is the key to this.

The outrage of so much of modern (building/environmental) 'design' is that there is clearly no commitment to this process of thoughtful analysis, just a quick, commercially led and superficial decision-making process that only just precedes the actual making of the building by the Contractors.

-- MartinNoutch

A nice quote from a "MurphysLaw" calendar we have at work:

> Reisner's Rule of Conceptual Inertia
>
> "If you think big enough, you'll never have to do it."

What is it that makes some of us want to procrastinate, often in very creative ways? I think this may be part of the dynamic of BigDesignUpFront.

-- JeffMantei

A good example IMHO of ConceptualInertia is the TunesProject.

I guess I'm a little fuzzy on what is meant by BigDesignUpFront. Does it mean detailed design? Or a high level design of the overall project/product? Because I believe you need some sort of high level design up front. A customer cannot "steer" the project if they do not know - on a high level - what they want this project to become. Details can change and features can be added and deleted, but you should know from the outset if you are building an IDE or just a word processor.

Next With

-- NicoleWilson

I think BigDesignUpFront (a.k.a. BDUF) is a waterfall-style attempt to do lots and lots of one phase (specifically design) before starting the next, and thus, to forsake any chance to get feedback.

The alternative probably ought to be called DesignAsYouGo (later: or better still, ContinuousDesign). -- PaulChisholm

Actually, I think that it should be called NotEnoughDesign - I don't have complete enough design that I can formally test against a formal requirements (in XP being code and unit tests), but I have enough of a design to send me on a wrong way with costly re-writes.

I'd argue that all people do some level of up front design (on any level of granularity), simply by knowing the context. Of course, it would be a very interesting experiment, to try to build a system w/o people who build it knowing the context. Say just give a programmer a story (or part of), existing code w/o anything else. And then have a control team that works with the full context knowledge.

I see this fairly often: programmers build "infrastructure" without knowing how those systems will be used. The result is typically bloated and difficult to use, because YouMightNeedIt. A clear idea of requirements and/or the YAGNI principle is needed to cure this ailment.

[Don't know whether the following still belongs here]

I prefer now instead of building a system say discovering a system. Then, you think about your system as a tree with the "whole" represented by the root and definite details as the leaves. With traditional methodologies, you do breadth first search/discovery. With agile ones, you do depth first. As anyone who implemented tree search algorithms, in depth first you need to hold less state -> your tree can actually change outside the branch you're just exploring w/o affecting your search.

-- VladEnder

I think XP is more about going from the leaves up to the root, and BigDesignUpFront is more about going from the root to the leaves. This is why XP tends to *discover* the tree. It comes from the leaves (the *real need* of the customers), and refactor as it discovers more leaves. Then it design some branches, and so on, up to the root. For me, that's the meaning of DesignAsYouGo or ContinuousDesign.

-- CyrilleGachot

A customer cannot "steer" the project if they do not know - on a high level - what they want this project to become.

I found this statement interesting because I found it 180 degrees opposed to my viewpoint. I feel the customer (specifically the front line users) know exactly what the project should become. It is the *developers* who need to know what the project should become. The question are the more effective and efficient means of communication from the users to the developers.

The problems I see with "Big Design Up Front" are that the two parties most interested in the communication (users and developers) are omitted; and the context of the user environment is lost when a large document listing precisely defined requirements is created.

I have become firmly convinced that there is no substitute for having individual developers spend a day with individual users and understanding the environment of the user's job. The developers invariably come back from the experience with drastically changed views of what they should be doing. Developers have to intuitively make hundreds of unconscious decisions while creating software; direct user experience provides a context for those decisions.

Again, the point is how best to get information and knowledge communicated from users to developers.

-- WayneMack

Is BigDesignUpFront the same as BigDesign? Should the pages be merged or at least refactored and crosslinked?

I think we should preserve the wording of "Big Design" linked with "Up Front". I don't think we can limit the size of the design, just the amount we do up front. Design should be an ongoing activity.

Has anyone else considered whether methods of design and implementation are related to one's mental skills and abilities as much or more than they are about finding the "best" objective way to produce a machine? If someone is a spatial thinker, they will use very different ways of envisioning the way something works than will someone whose preferred way of thinking is logically-based. Most up front designers, I would guess, have spatial thinking skills - they want to utilized their skill with understanding structural relationships, while XPers might be better at process logic, and want to focus on the working interaction of components. But modes of thinking and abstracting have a big effect on how we go about constructing things. -- Brent

I'm not sure that you can prefer spatial thinking to logical. Both are heavily ingrained in the brain, as evidenced by a few sentences taken from any conversation. Things that have nothing to do with space or force are expressed in those terms.

I think both types of thinking are needed, the question is the order in which they are applied. In BigDesignUpFront, one tries to apply only spatial thinking to create a design followed by only logical thinking to implement the design. In the XP iterative approach, one first applies logical thinking to DoTheSimplestThingThatCouldPossiblyWork, then one applies spatial thinking to ReFactor, then one repeats the entire cycle. -- WayneMack

---

The discussions on this page were reordered for clarity. Does anyone but me see the irony of this?

Sorry. Such is the nature of ThreadMode. This page is not a design - it is an ongoing discussion waiting to be trimmed down to DialecticMode some fine day. Have at it.

I missed the irony. Is it that the BigDesignUpFront page was not designed and developed in one pass, but was done iteratively?

The BigDesignUpFront may also be seen as the HighLevelDesignUpFront. I know that I don't always think of the details of how I will implement the tiny constructs (the new ones to add to the system) as much I am looking at the over all picture. In taking my senior level courses, we always were told to approach what was to by developed on a large scale. Though using this term confused most of us at first because we were trying to do everything at once. Stepping back and rethinking about what was being stated, we realized that thinking on a more abstract scale or of the overall design was large whereas thinking of every single detail brought us closer to getting the minor things solved. The BigDesignUpFront is how you would like to get things rolling, and then time and the details will help to focus on what to change and to make sure your initial vision is maintained, if still needed. -- YemiIDBedu

---

This seems to overlap AnalysisParalysis

I stumbled across this set of comments by accident while looking for something on RequirementsTracking, and have been fascinated by the different points of view on design. As with most areas of disagreement, I believe many of these disagreements come from differing perspectives and experience.

My long years of experience in systems (over 35 years) have taught me that up-front design is always good. The only problem is in how much knowledge

up front design is always good. The only problem is in how much knowledge is available to do the design. BDUF is only possible in some situations. Consider these differing types of projects:

- knowledge of requirements: a lot is known (redevelopment of a legacy system) versus a leading edge relationship marketing system where requirements are developed as the system is built
- size of project: small (under ten staff) to large (over 100 staff)
- knowledge of the system: the customer knows the system but the developers are new, or the senior staff of the project knows what is needed but the programmers do not, or the development team is experienced and has done this type of system before but the customer has not

I believe one can see that in some cases above, more information is available to contribute to a good design, and in other cases, little information is available. In some cases (small project, little knowledge of requirements) the XP approach may be best. In other cases (small project, customers know the system), the XP approach will likely be as successful as any other approach.

But in many other cases, up-front design will save overall time and money, and contribute to a better system. For example, in a large project (over 100 staff), how do you even know it is a large system unless some up-front work is done? Can a project manager set 100 programmers loose without some idea of what is to be built? (maybe in the case where the team has done the system many times already...).

An experienced project manager knows that a plan (or design) is not perfect, and will evolve as more information becomes available. Good development methodologies allow for multiple iterative passes from the high level to detail, and call for prototyping of critical components early in the project.

I hope this contributes to the debate. -- NeilCarscadden

I've spent a great many years studying various project methodologies. On this particular issue, it is my opinion that both camps have points to concede.

First off, the BDUF crowd needs to acknowledge that even in their world architecture and design are not completely specified up front but are in fact emergent. In nearly all projects, a certain amount of experimental work has to be done to answer any significant questions or mitigate any serious risks that could impact the success of the effort. There nearly always is a set of unknowns that can be resolved only through actual construction. This process is generally called R&D (research and development), and the result is a prototype that defines the basic nature of the end product. It establishes a framework upon which everything else can be built. The BDUF crowd tends to delineate this as a separate task (or sometimes even a separate project), and in many cases it's performed by a different team. XP, on the other hand, just considers it to be part of the overall process.

Second, the XP crowd needs to acknowledge that even they commit to a design early on. One has to in order to get anything done. One cannot rework the whole system for each new feature.

- Any environment that requires system-wide rework for many features is not using XP. The whole point of refactoring is to minimize the change impact time and cost. Instead of 'commit to a design' you negotiate features with the client, which gives them what they most want in a shorter timespan than BDUF (and is much safer).
  - The 'point' of refactoring may be to minimize change impact time and cost, but the *actual mechanism* of refactoring relies on the *assumption* that the functionality you'll be adding in the future look very similar to functionality you added in the past. This is not the case for all features, especially for NonFunctionalRequirements and various 'emergent' features. As a consequence, you are wrong: one can be 'using XP' and still end up requiring system-wide rework, and refactoring won't always help you. That said, BigDesignUpFront isn't always the best answer even in situations that may end up requiring serious rework (very often you won't know all of your requirements up front). Reworking the system - essentially refactoring the entire codebase - can be acceptable so long as it doesn't happen too often. It's the entire basis of prototypes and PlanToThrowOneAway.

There comes a point where a particular approach must be selected and adhered to. Again, the BDUF crowd tends to explicitly demarcate when this happens, whereas XP does not.

Really, the big difference between the two approaches is in how they treat R&D. BDUF tends to separate it from actual production work, with one leading into the other. XP, on the other hand, combines the two, thus enforcing the idea of emergence.

-- Milo Hyson

Okay, I'm new and this is my first (literally first ever post here) I would think the RightThing is something like ConvergentDesgin or EmergentDesign depending on how clearly you understand where you want to end up. (If that's been defined elsewhere, just point me and off I'll go.) As an effort to scratch my own itch, IWannaLearnPython, I'm trying to write Something that does financial business modelling and forecasting. I'm quite sure there's stuff that exists that does that - the principal point is my learning to write a non-trivial piece of software - this just happens to be something I know about. I don't have a BigDesign, but I do have a BigIdea or 2, though, and I want to see how I can implement them, both as design and as code

-- DanEsch (edited for CamelCaseification after reading some styleguides)

See: ClarityUpFront, DesignDiagramsArentEvil, WhyIsntSamCodingYet, TheDeadlineOnBigDesignUpFront, AdequateArchitectureUpFront, CanAnArchitectureEmerge, MeasureTwiceCutOnce, DesignUpFrontButExtremeImplementation, OfMiceAndMen, SystemSizeMetrics, DesignApproachTina, BigReductionUpFront, YagNi

CategoryDesignIssues, CategoryPlanning, CategoryAnalysis, CategoryDecisionMaking

---

Last edit April 13, 2013, See github about remodeling.