Chapter 2 – Amplify Learning

The Nature of Software Development

The origins of lean thinking lie in production, but lean *principles* are broadly applicable to other disciplines. However, lean production *practices* – specific guidelines on what to do – cannot be transplanted directly from a manufacturing plant to software development. Many attempts to apply lean production practices to software development have been unsuccessful, because generating good software is not a production process, it is a development process.

Development is quite different than production. Think of development as creating a recipe, and production as following the recipe. These are very different activities, and they should be carried out with different approaches. Developing a recipe is a learning process involving trial and error. You would not expect an expert chef's first attempt at a new dish to be the last attempt. In fact, the whole idea of developing a recipe is to try many variations on a theme and discover the best dish.

Once a chef has developed a recipe, preparing the dish means following the recipe. This is equivalent to manufacturing, where the objective is to reproduce a 'recipe' faithfully many times with a minimum of variation. The difference between development and production is outlined in table 2-1.[1]

Development	**Production**
Designs the Recipe	*Produces the Dish*
✓ Quality is fitness for use	✓ Quality is conformance to requirements
✓ Variable results are good	✓ Variable results are bad
✓ Iteration generates value	✓ Iteration generates waste (called rework)

Table 2-1. Development vs. Production

Perspectives on Quality

In production, quality is defined as conformance to requirements specified in the design or 'recipe'. In the service industry, a different perspective on quality has emerged.

[1] See Ballard, 'Positive vs Negative Iteration in Design' (2000)

The Service View of Quality

Walt Disney designed Disneyland as a giant stage where several hundred actors make it their job to be sure every guest has a wonderful time. One guest's requirements for 'having a wonderful time' are quite different from the next, and the actors are supposed to figure out exactly what each guest thinks a quality experience should be, and make sure they have it.

Quality at Disneyland

At Disneyland, even the tram drivers are actors. A friend told me the story of a tram driver who noticed a small girl crying on her way back to the Disneyland hotel. He asked her why she was crying, and found out that the crowd around Mickey Mouse was too large, so the girl had not been able to talk to him. He called ahead, and when the tram arrived at the hotel, there was Mickey Mouse, waiting to meet it. The girl was thrilled, and the driver had done his job of making sure she had a quality experience.

– Mary

The service view of quality takes into account that every customer has a different idea of what constitutes a quality experience. In a service economy, quality does not mean conformance to a script; it means adapting to meet the changing expectations of many different customers.[2]

Quality in Software Development

Quality in software development results in a system with both ***perceived integrity*** and ***conceptual integrity***. ***Perceived integrity*** means that the totality of the product achieves a balance of function, usability, reliability and economy that delights customers.[3] ***Conceptual integrity***[4] means that the system's central concepts work together as a smooth, cohesive whole. We devote Chapter 6 to the important topic of software integrity.

[2] See Prahalad, 'The New Meaning of Quality in the Information Age' (1999) and Prahalad, The Dynamic Synchronization of Strategy and Information Technology' (2002)

[3] The definition of perceived and conceptual integrity is adapted Clark, *Product Development Performance* (1991) p 30

[4] A term found in Brooks, *Mythical Man Month* (1995) p 42

Customers of a software system will perceive integrity in a system if it solves their problem in an easy-to-use and cost-effective manner. It does not matter whether the problem is poorly understood, changes over time, or is dependent on outside factors; a system with perceived integrity is one that continues to solve the problem in an effective manner. Thus, quality in design means ***realization of purpose*** or ***fitness for use*** rather than *conformance to requirements.*

Variability

When you think of quality in a service business such as Disney World, the one thing you can count on is that each customer will have different expectations. True, the theme park has to be clean and the rides have to work, but if you tried to provide one experience to all customers, your theme park would not be widely popular. The difference between providing a service and manufacturing a product is that in service, dynamically shifting customer expectations require variation, while in manufacturing, variation is the enemy. Manufacturing assumes a homogeneous, unchanging set of customer expectations, so the objective is to make a product the same way every time.

Somehow, the idea that variation is bad has found its way into software development, where people have tried to develop standardized processes to reduce variation and achieve repeatable results every time. But development is not intended to produce repeatable results; development produces appropriate solutions to unique customer problems.

Design Cycles

It was once thought that good programmers develop software though a structured, top-down approach.[5] In 1990, Raymonde Guindon evaluated the paradigm that top-down decomposition is the best approach to software design. She reported on research in which experienced designers were asked to design an elevator control system, while describing each step of their thought process to researchers. She found that when experienced designers are presented with ill-defined problems, their design activities are not at all top-down. They move repeatedly between scenario examination, requirements elucidation, high-level

[5] See Yourdon (1979), particularly the articles: 'Structured Programming' by Dijkstra and 'On the Composition of Well-Structured Programs' by Niklaus Wirth. See also Brooks (1995) p 143.

solution segmentation, and low-level design of difficult elements. (See Figure 2 - 1.)

Shifts in design activities and levels of abstraction of Designer 2. Plus signs indicate newly inferred or added requirements. Light bulbs indicate sudden discovery of partial solutions or requirements. The region marked by *R* indicates the period of solution review.

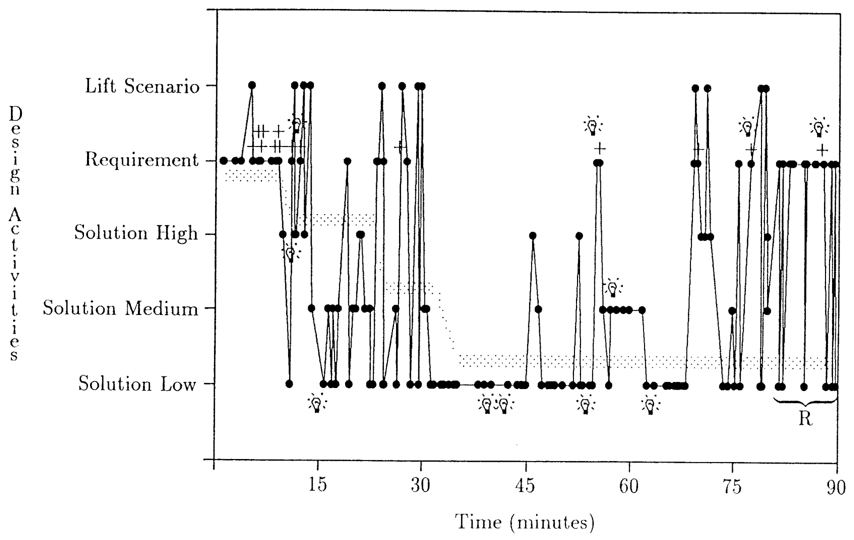

Figure 2-1. Design Activity[6]

Guindon found that cycling between high-level design and detailed solution was typical of good designers when dealing with ill-structured problems, that is, problems that do not have a single right answer or a best way to arrive at a solution. She theorized that this unstructured approach is necessary to understand and give structure to such problems.[7]

The bulk of the work of software development is a problem-solving activity similar to that investigated by Guindon. Software problems are solved at many levels, by all members of the development team. Software architects are clearly involved in a design activity, but so are developers who write the code. The process of writing code involves deep problem understanding, recognition of

[6] From Guindon, 'Designing the Design Process' (1990) p 319. Used with Permission.

[7] Guindon, 'Designing the Design Process' (1990)

patterns from experience, experimentation with various approaches, testing the results, and determination of the best approach.

Today it is widely accepted that design is a problem-solving process that involves discovering solutions through short, repeated cycles of investigation, experimentation, and checking the results. Software development, like all design, is most naturally done through such learning cycles.

Do It Right the First Time?

In order to solve problems that have not been solved before, it is necessary to generate information. For complex problems, the preferred approach to a solution is to use the scientific method: observe, create a hypothesis, devise an experiment to test the hypothesis, run the experiment, and see if the results are consistent with the hypothesis. One of the interesting features of the scientific method is that if your hypothesis is always correct, you are not going to learn very much. The maximum amount of information is generated when the probability of failure is 50 percent, not when the hypotheses are always correct. It is necessary to have a reasonable failure rate in order to generate a reasonable amount of new information.[8]

There are two schools of thought in developing software. One is to encourage developers to be sure that each design and each segment of code is perfect the first time. The second school of thought holds that it is better to have small, rapid *try-it test-it fix-it* cycles than it is to make sure the design and code are perfect the first time. The first school of thought leaves little room for knowledge generation through experimentation; instead, it believes that knowledge generation should happen through deliberation and review. The *right the first time* approach may work for well-structured problems,[9] but the *try-it test-it fix-it* approach is usually the better approach for ill-structured problems.

If the *right the first time* approach is preferred in your organization, you might ask yourself why this is a value. As Yourdon points out, "a piece of program logic often needs to be rewritten three or four times before it can be considered an

[8] Reinertsen, *Managing the Design Factory* (1997) p 71

[9] Well-structured problems have a single right solution and a preferred approach to arriving at the solution. For example, most problems children encounter in elementary school are well-structured problems.

elegant, professional piece of work." Why, he asks, do we object to revising programming logic when we are quite happy to re-write prose three or four times to achieve a professional result?[10]

Your objective should be to balance experimentation with deliberation and review. In order to do this, consider how you can generate the most knowledge at the least cost in your circumstances. For instance, if the cost of testing is very high, you will want more knowledge to be generated through deliberation and review. If experimentation is relatively inexpensive and yields better knowledge faster, then it is the least expensive, most effective approach. Usually some combination of experimentation, peer review and iteration will yield the best results.

Learning Cycles

Quite often, the problem to be solved is understood best by the people in the business with the problem, so it is usually necessary to have business people – or representatives such as focus groups – in the knowledge-generation loop. In this case, it is important to speak to the business people with a representation they readily grasp, or the knowledge generation will be inefficient. There are many ways to represent the system, from models to prototypes, to incremental deliveries, but the important thing is to select the representation that gathers the most knowledge. Most users relate better to seeing working screens than to a requirements document, so working software tends to generate better knowledge faster.

Iterations with refactoring – improving the design as the system develops – have been found to be one of the most effective ways to generate knowledge, find answers early, and generate a system with integrity, because this approach generates knowledge most effectively for ill-defined problems. The important question in development is 'How can I learn most effectively?' The answer is often to have many, short learning cycles. If you ask instead 'How can I minimize the number of learning cycles?' you are likely to get long cycles, large batches, long feedback loops, and as a result, ineffective learning.

[10] Yourdon, *Classics in Software Engineering* (1979) p151

The True Story of a Death March Project – Part 2: Weekly Iterations

As the first installment of this drama drew to a close in Chapter 1, we had just released a very shaky system to production in a mission-critical area. Only half of the features worked, but the law required the new logic, so against our better judgment, we went live. The customer agreed to work around missing features manually, while we agreed to release new capabilities to the system every week.

We made a list of missing features and known defects, which we called a punch list. Every week we had the customer review and prioritize the list. On Friday, the developers selected from the top of the punch list those features that they thought they could complete in a week. Users ran a lengthy, manual regression test on the new release the following Thursday, and usually we had to re-build and re-test on Friday. We did not allow new features into the build after the first regression test, so we usually could release the build to production after the second regression test. If not, we tested over the weekend. Almost every Monday morning for three months, a new release went into production. Generally scripts were run on the database as part of the release, so once production started, there was no going back to the previous release.

Releasing a new version of a mission-critical system to 100 users every week, with no fallback, seems like a high risk approach. But we never had a disaster and the weekly releases caused remarkably few problems. The discipline of the regression testing coupled with the small increments of functionality worked like magic. Development and testing was done at the customer site, so if there were questions or problems, feedback was immediate.

Once most of the features were delivered, the customers no longer wanted the hassle of weekly regression tests, so the iterations stretched to two or three weeks. We found that it was devilishly difficult to pass regression testing with the longer increments. As release intervals stretched out, it became tempting to add just one last feature to a release even after its first or second regression test. This was invariably a mistake, making another build and more testing necessary, causing the interval to stretch out, making it more tempting to add more features to the current release. Stretching out intervals was a vicious circle.

Things never went so well as during that heady time when things were so bad that weekly production releases seemed to be the only option. As the urgency fad ed and we lengthened the feedback cycle, it got more and more difficult for a new release to pass the regression tests. We never were able to automate the regression tests, but were we to do this over again, that would be the first step.

– Mary

Tool 3 – Feedback

It's two in the morning and you are driving home. The traffic light is red, and there's not another car in sight. But the traffic light is red, so you stop. And wait. And wait. Finally, the light changes, after allowing time for lots of non-existent cross-traffic. You think to yourself, "It's going to be a long drive home". And sure enough, the next light is also red. But as you approach the light, it turns green. "Ah HA!" You think to yourself. "An automatic sensor. That light is smart enough to know I'm here and there's no one else around. I hope the rest of the lights are like that!"

The difference between the two lights is feedback. The first light was pre-programmed based on the assumption that there will be three times as much traffic on the main road as on the side road, so you sat through a long light. The second light had sensors buried throughout the intersection and was programmed to adjust its cycle based on traffic patterns as they vary throughout the day and night.

Figure 2-2 shows how the first traffic signals works.

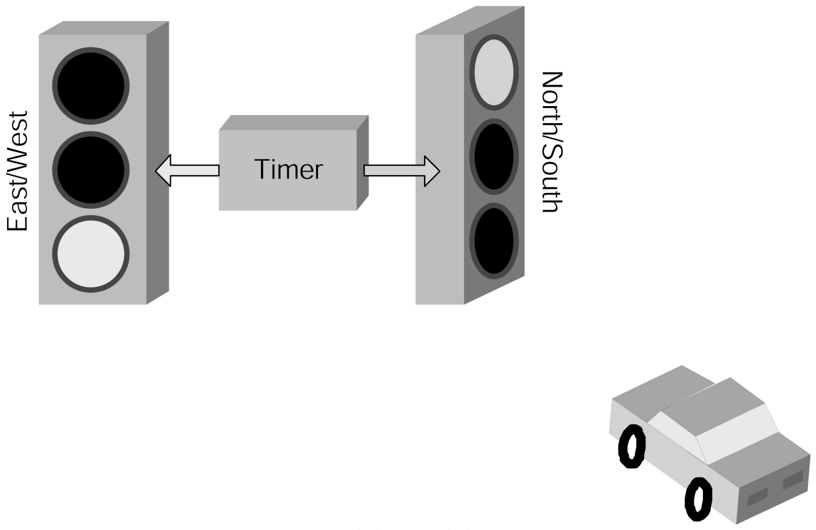

Figure 2-2. Timed Traffic Light

Figure 2-3 shows how the second traffic signal works.

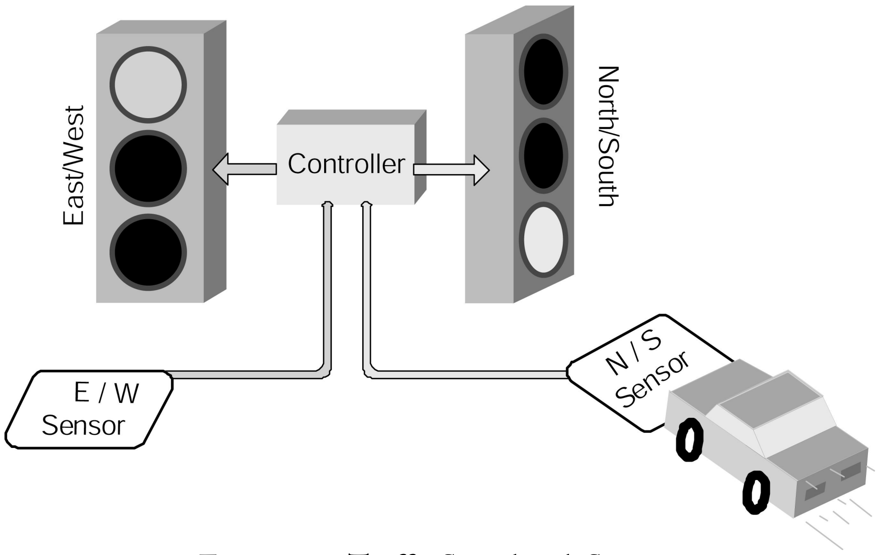

Figure 2-3. Traffic Signal with Sensors

Notice that the second set of traffic signals have more components, more logic, and more things to go wrong. But traffic lights with feedback are desirable despite their increased complexity. Feedback adds considerable value, and thus it is very common. Your home heater and air conditioner are controlled with a feedback loop, as is your oven. Figure 2 -4 shows a feedback loop for an oven:

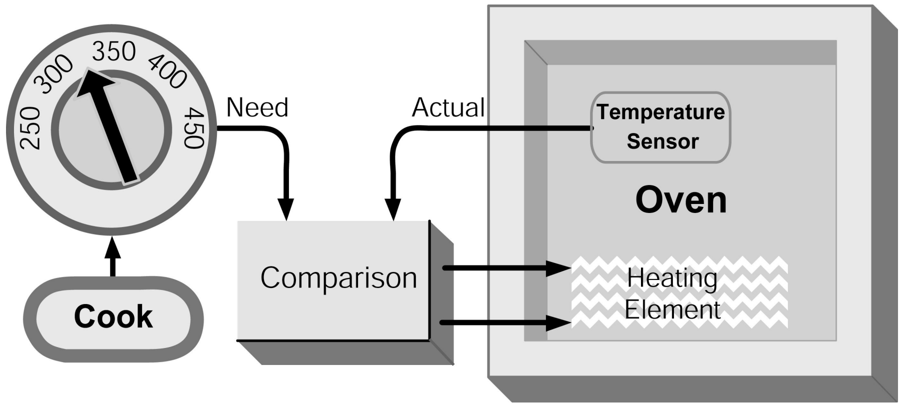

Figure 2-4. Oven

In a steel mill or a tape manufacturing plant, there are many variables to control: speed, pressure, heat, thickness. The formula for making tape or steel includes a setpoint for each variable. Operators or computers dial in the setpoint, and then a feedback loop provides the control for each variable. It is rare to find control without feedback, because feedback gives much better control and predictability than attempting to control complicated processes with pre-defined algorithms.

Software Development Feedback Loops

There are many unforeseeable events in developing software, so why would anyone think that software systems should be developed without feedback loops? Winston Royce proposed a sequential software design process in 1970 which closely resembled the sequential product development processes of the time. He advocated creating detailed documentation at each step, but also pointed out that waiting to test the system until the end was not practical, because the feedback provided by testing was needed early in the development process. Therefore he suggested that an early prototype be built to provide feedback. [11]

[11] See Royce, 'Managing the Development of Large Software Systems' (1970) Figure 7.

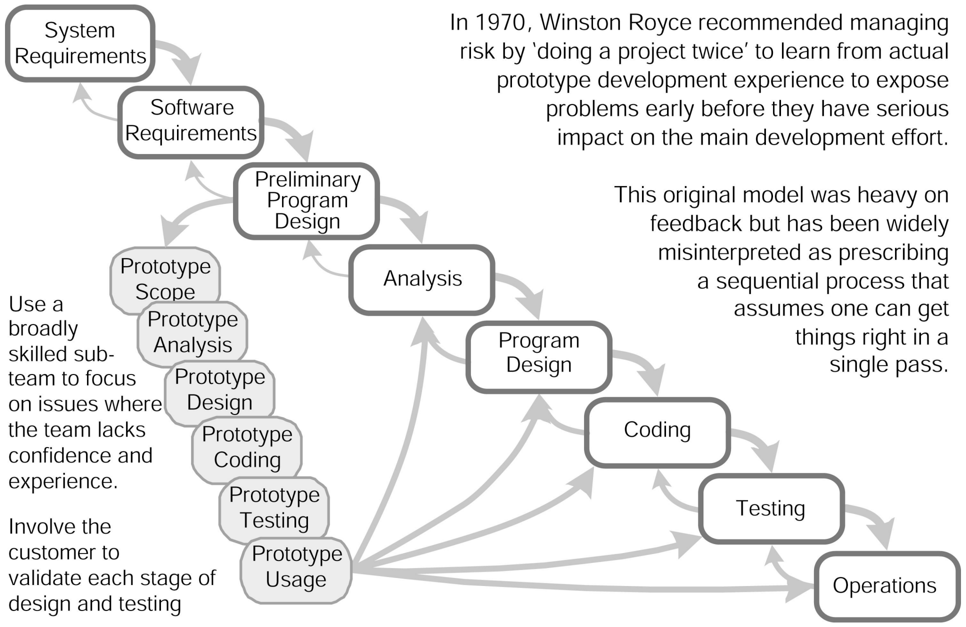
System Requirements
Software Requirements
Preliminary Program Design
Analysis
Program Design
Coding
Testing
Operations
Prototype Scope
Prototype Analysis
Prototype Design
Prototype Coding
Prototype Testing
Prototype Usage
In 1970, Winston Royce recommended managing risk by 'doing a project twice' to learn from actual prototype development experience to expose problems early before they have serious impact on the main development effort.
This original model was heavy on feedback but has been widely misinterpreted as prescribing a sequential process that assumes one can get things right in a single pass.
Use a broadly skilled sub-team to focus on issues where the team lacks confidence and experience.
Involve the customer to validate each stage of design and testing

Figure 2-5. Original Royce "Waterfall" Recommendation

In 1975, Fred Brooks wrote: "Plan to throw one away; you will anyhow."[12] Brooks retracted this in 1995, saying: "Don't build one to throw away – the waterfall model is wrong."[13] He notes that his original quote implicitly assumed a sequential development process, while it has become clear that a incremental model with progressive refinement is the proper approach.[14]

As actually implemented, the sequential, or waterfall, development model does not usually provide for much feedback; it is generally thought of as a single pass model. This can be called a deterministic model because it assumes that the details of a project are determined at the beginning. A deterministic model is favored by project management disciplines that have their origins in contract administration. The contract-inspired model of project management generally favors a sequential development process with specifications fixed at the start of the project, customer sign-off on the specifications, and a change authorization process intended to minimize changes. There is a perception that these processes

[12] Brooks, *Mythical Man Month* (1995) p116
[13] Brooks, *Mythical Man Month* (1995) p 264
[14] Brooks, *Mythical Man Month* (1995) p 267

give greater control and predictability, although sequential development processes with low feedback have a dismal record in this regard.[15]

Traditional project management approaches often consider feedback loops to be threatening because there is concern that the learning involved in feedback might modify the pre-determined plan. The conventional wisdom in project management values managing scope, cost, and schedule to the original plan. Sometimes this is done at the expense of receiving and acting on feedback that might change the plan; sometimes it is done at the expense of achieving the overall business goal. This mental model is so entrenched in project management thinking that its underlying assumptions are rarely questioned. This might explain waterfall model of software development is so difficult to abandon.

Imagine Deterministic Cruise Control

You are driving along the highway. You get up to the speed you want to go, turn on the cruise control and push ***set***. The car has a control loop which operates every few seconds, checking the actual speed of the car against the speed you set (the *setpoint*). If the car speed is less than the setpoint, the cruise control depresses the accelerator a bit. If the speed is higher than the setpoint, the cruise control lets up on the accelerator.

Imagine driving a car where the position of the accelerator was preprogrammed at the factory. If you want to go 60 mph, it moves the accelerator to position A; if you want to go 65 mph, it moves the accelerator to position B and so on. This might work on flat terrain, but when it gets to a steep hill the car would slow to a crawl. Upon reaching the top, the car would careen dangerously fast down the other side.

Deterministic control simply does not work when there is variability in the terrain.

When an organization has software development challenges, there is a tendency to impose a more 'disciplined' process on the organization. The prevailing concept of a more disciplined software process is one with more rigorous sequential processing: requirements are documented more completely, all agreements with the customer are written, changes are controlled more carefully and each re-

[15] Johnson, Keynote Third International Conference on Extreme Programming (2002)

quirement must be traced to code. This amounts to imposing additional deterministic controls on a dynamic environment, lengthening the feedback loop. Just as control theory predicts, this generally makes a bad situation worse.

In most cases, increasing feedback, not decreasing it, is the single most effective way to deal with troubled software development projects and environments.

- ✓ Instead of letting defects accumulate, run tests as soon as the code is written.
- ✓ Instead of adding more documentation or detailed planning, try checking out ideas by writing code.
- ✓ Instead of gathering more requirements from users, show them an assortment of potential user screens and get their input.
- ✓ Instead of studying more carefully which tool to use, bring the top three candidates in-house and test them.
- ✓ Instead of trying to figure out how to convert an entire system in a single massive effort, create a web front end to the legacy system and try the new idea out.

Whenever a person does work, they should be doing it for an *immediate customer*; that is, they should have someone, somewhere, eager to make use of the results of their work. Everyone should know their immediate customer and have ways for that customer to give them regular feedback. When a problem develops, the first thing to do is to make sure the feedback loops are all in place; that is, make sure everyone knows who their immediate customer is. The next thing to do is to increase the frequency of the feedback loops in the problem areas.

The True Story of A Death March Project – Part 3: Amplifying Feedback

When I took over the project, it was stuck. The design was supposed to be done, but there were no designers on the team. No one could agree on what constituted an appropriate design format. The analysts did not know what to do, and the programmers did not find the existing documents detailed enough to work from. Wheels were spinning, but nothing was happening.

I was new, so I could change things. I asked the analysts to choose a small part of the system and take a day to write use cases, then sit down with the developers and see if the use cases were useful. Working together, the analysts and developers were to discover the level of detail needed in a use case that was possible for the analysts to provide and sufficient for programming to proceed. Then the developers were to write code for the small part of the system and have analysts test it to see if it was what they had in mind.

After two weeks the log jam was broken and code started flowing. The analysts developed a style of writing use cases that the developers found useful, and the developers started holding regular meetings with the analysts so they could ask questions that were not covered in the use cases. It was a start.

– Mary

Tool 4 – Iterations

If a manufacturer wants to start applying lean production principles, there is one starting point that always works – use just-in-time inventory flow. The simple act of working to fill customer orders rather than working to meet a schedule drives a host of other improvements. One reason just-in-time flow is so effective is that it requires significantly improved worker-to-worker communication and surfaces quality problems as soon as they occur.

In concurrent product development, which we will discuss in the next chapter, there is an equivalent universal starting point that always works – drive the effort with prototypes at closely placed milestones. A prototype synchronizes efforts toward a well-understood short term goal without the need for detailed scheduling. Regular prototype milestones make concurrent product development possible because they provide a focal point around which cross-functional communication can and must occur. Prototypes also provide early feedback on design problems and customer preferences.

There is an equivalent universal starting point for all agile software development approaches: iterations. An iteration is a useful increment of software that is designed, programmed, tested, integrated, and delivered during a short, fixed timeframe. It is very similar to a prototype in product development, except that an iteration produces a working portion of the final product. This software will be improved in future iterations, but it is working, tested, integrated code from the beginning. Iterations provide a dramatic increase in feedback over sequential software development, thus providing much broader communication between customers/users and developers and among various people who have an interest in the system. Testers are involved from the first iteration; hardware and software environments are considered early. Design problems are exposed early, and as changes occur, change-tolerance is built into the system.

There are three fundamental principles at work here. First, as we will see in Chapter 4, small batches moving rapidly through a system lead to all manner of good things. Small batches enforce quality and worker-level communication, while allowing for greater resource utilization. They provide short feedback loops, which enhances control. For this reason, short, complete iterations are as fundamental to lean development as small batches are to lean manufacturing.

Second, short iterations are an options-based approach to software development. They allow the system to respond to facts rather than forecasts. There are few endeavors in which it is more important to keep options open than software development. In Chapter 3 we see that options-based approaches are fundamentally risk-reduction strategies, and as counterintuitive as it may sound, you actually reduce your risk by keeping options open, rather than freezing design early.[16]

Finally, iterations are points of synchronization across individual and multiple teams and with the customer. Iterations are the points when feature sets are completed and the system is brought as close to possible to a 'releasable' or 'shippable' state – even if it will not actually be released. Thus iterations force decisions to be made. Frequent points of synchronization allow teams to work independently, yet never stray far from the work of other teams or the interests of customers and users.

[16] See Thimbleby, 'Delaying Commitment,' (1988) pp 78-86.

Iteration Planning[17]

What work should be done in each iteration? The idea is to implement a coherent set of features in each iteration. A feature is something that delivers meaningful business value to the customer but is small enough that the team can confidently estimate how much effort delivering it will require. If a feature can't be done in a single iteration, it should be broken down into smaller features. Features come from customers or customer representatives in the form of use cases or stories or backlog items.[18]

At the beginning of each iteration, a planning session occurs at which the development team estimates the level of difficulty of the features under consideration and the customers or customer representatives decide which features are most important, given their estimated cost. The highest priority features should be developed first, in order to deliver the highest business value first. High-risk items should be addressed earlier rather than later.

An iteration should have a fixed time-box. Some people suggest keeping all iterations to the same length, to establish a rhythm. Others vary the iteration length based on local circumstances. How long should the iteration time-box be? It should be long enough to support a meaningful design-build-test cycle, and short enough to provide frequent feedback from customers that the system is on track. Some people feel a one-month time-box is ideal. Others suggest time-boxes of a couple of weeks. Some companies use six to ten week time-boxes, but these are coupled with daily builds and extensive weekly testing.

The development team must be free to accept only the amount of work for an iteration than team members believe they can complete within the time-box. Customers will probably want to load iterations with lots of features, but it is important to resist the temptation to be accommodating at the expense of setting unreasonable expectations. If iterations are short and delivery is reliable, customers should be content to wait for the next iteration. If a development team over-

[17] See Schwaber, *Agile Software Development with Scrum* (2002) pp 47-50 for a discussion of planning a sprint in Scrum. Beck, *Extreme Programming Explained* (2001) Chapters 17 and 18 discuss iteration planning in eXtreme Programming.

[18] The best reference on use cases is Cockburn, *Writing Effective Use Cases* (2000). Stories are used in eXtreme Programming. See Beck, *Extreme Programming Explained* (2000) A backlog List is used in Scrum. See Schwaber, *Agile Software Development with Scrum* (2002).

commits –which often happens to inexperienced teams – it is best to deliver some of the features on time rather than all of them late.

Team Commitment

A development team can evaluate a list of features and with a little bit of investigation, they can come up with a good idea of what they can do in few weeks or a month. If you ask a team to choose items from the top of a list that they believe they can do in a short time-box, they will probably choose and commit to a reasonable set of features. Once the team has committed to a set of features that they think they can complete, they will probably figure out how to get those features done within the time-box.

A team should not be expected to set and meet time-box goals without organizational support.[19]

- ✓ The team must be small and be staffed with the necessary expertise. Some team members must be experienced in the domain and some in each critical technology.
- ✓ The team must have enough information about requested features to be able decide what is feasible to accomplish in the time-box.
- ✓ The team must be assured of getting the resources it needs.
- ✓ Team members must have the freedom, support, and skill to figure out for themselves how to meet their commitments.
- ✓ The team must have or create the basic environment for good programming:
  - → Automated build process
  - → Automated testing
  - → Coding standards
  - → Version Control Tool
  - → Etc.

[19] See Schwaber, *Agile Software Development with Scrum* (2001)

Good iteration planning gives customers a way to ask for features that are important to them, while creating a motivating environment for the development team. The best part about these benefits is that they feed upon success. As customers see the features they regard as highest priority actually implemented in code, they start to believe the system is going to be real and begin to envision what it can do for them. They become comfortable that features scheduled for future iterations will actually be delivered. At the same time, developers gain a sense of accomplishment, and as customers begin to appreciate their work, they are even more motivated to satisfy the customers.

Convergence

Iterations sound like a good idea, yet there is a significant reluctance to use them. The reason behind this can often be traced to a fear that the software development effort will not converge. There is a concern that the project will continue on indefinitely if it does not have a predefined stopping point.[20] This is a valid concern; how can you be sure that any system with a feedback loop will converge on a solution? In fact, books on control theory have more pages on convergence than on any other topic. It is not a concern to be taken lightly.

A fluid business situation might send unpredictable and constantly changing signals to the software development process. It is not unusual for a situation called *thrashing* to develop, that is, the feedback changes so fast the system doesn't have time to complete one response before being told to go in the opposite direction.

Consider a thermostat. It does not turn on the furnace the moment the room temperature falls below the setpoint, and then turn it off the moment the temperature rises above the setpoint. If this happened, the furnace would cycle on and off constantly, something that is not good for furnaces. Instead, the thermostat turns on heat when the temperature falls a couple of degrees below the setpoint, and leaves the furnace on until the temperature is a degree or two above the setpoint.

An iterative software development process achieves this same effect by limiting customer requests for feature changes to the beginning of each iteration. During the iteration, the team concentrates on delivering the features they committed to

[20] Highsmith, *Adaptive Software Development* (2000) p 87

at the beginning of the iteration. If the iterations are short – 2-4 weeks – the feedback loop is still quite short.

Delaying response to feedback must be handled with care; long delays in feedback tend to cause system oscillation. Convergence requires small, frequent adjustments. For example, a cruise control adjusts the accelerator only slightly when the car falls below the desired speed. Similarly, if software is delivered in small, frequent increments, the customer can see business value increasing with each increment and make adjustments on a regular basis. Delivering large increments on an infrequent basis is far more likely to produce oscillations than accepting frequent feedback.

There is an optimal window for feedback – it should be as short as possible without being so short as to create thrashing. The optimal size of this window depends on the dynamics of the situation, but in general, environments that are more dynamic require more rapid feedback. Some have found that larger teams do better with more frequent feedback, because if a large team gets off track, it is more difficult to reverse direction.

Negotiable Scope

A good strategy for achieving convergence is to work on top priority items first, leaving the low priority items to fall off the to-do list. By delivering high priority features first, it is likely that you will deliver most of the business value long before the customer's wish list is completed. Here comes the tricky part. If you are working under the expectation that development is not complete until a fixed, detailed scope is achieved, then the system may indeed not converge. So it's best to avoid this expectation, either by stating at the front that scope is negotiable, or by defining scope at a high level so it is negotiable in detail. With negotiable scope, iterative development will generally converge.

Why should a customer accept the idea of negotiable scope? In the introduction to this book, we told the story of how Florida and Minnesota each set out to develop a SACWIS (Statewide Automated Child Welfare Information System). The systems are quite similar, but the Florida system will take about 15 years and cost about $230 million, while the Minnesota system was completed in two years at the cost of $1.1 million. This vast difference in time and cost for developing essentially the same system is credited to two factors: Minnesota used a standardized infrastructure and minimized requirements. [21]

A Standish Group study found that 45% of features in a typical system are never used and 19% are rarely used.[22] Since customers often don't know exactly what they want at the beginning of a project, they tend to ask for everything they think they might need, especially if they think they will get only one shot at it. This is one of the best ways we know to increase the scope of a project well beyond what is necessary to accomplish the project's overall mission.

If you let customers ask for just their highest priority features, then deliver them quickly, and ask for the next highest priority, you are more likely to get short lists of what is important. Moreover, you can respond to their changing circumstances. Therefore, it is usually a good idea to work down a prioritized feature list from the top. In general, this strategy will accomplish the overall mission by the time the allocated resources are up.

This approach to project management may seem to lead to unpredictable results, but quite the opposite is true. Once a track record of delivering working software is established, it is easy to project how much work will be done in each iteration as the project proceeds. By tracking the team *velocity*, you can forecast from past work how much work will probably be done in the future. Velocity measurements are significantly more accurate tools than scope-based controls, because they are measuring how much time it actually took to deliver complete, tested, releasable code at the end of each iteration. You know exactly where things stand after only a few iterations, which provides highly reliable early predictions of project performance.

It is a good idea to make progress visible, both to the development team and the customer. One way to do this is with burn-down charts.[23] Let's assume that you develop a high level list of features to be delivered, and make a preliminary estimate of the development time of each feature. You add all the estimated times and get a time-to-complete number, say 500 staff days. Assume for simplicity that your iterations are one month long. After the first iteration, the customer

[21] Johnson, Keynote, Third International Conference on Extreme Programming (2002)

[22] Johnson, Keynote, Third International Conference on Extreme Programming (2002)

[23] More detail on using burn-down charts can be found in Schwaber, *Agile Software Development with Scrum* (2001) pp 63-68.

may have added more items, and the team will have completed some items. You add up the time to complete and notice that it is actually larger than the month before, say 620 staff days. After 4 months, your graph might look like the left hand burn-down chart in Figure 2-6, which shows that the system is not converging very fast:

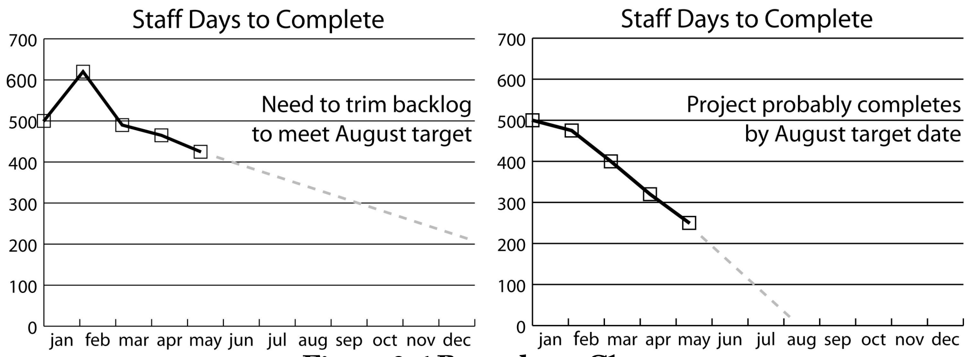

Figure 2-6 Burn-down Charts

If you expect the system to be done in nine months, you should be seeing convergence more like the right hand burn-down chart in Figure 2-6. Since that is not what is happening, you know after a couple of months that action is necessary. If the customer is adding new features as fast as the team is completing others, it is time to consider deleting features from the list. If the team is bogging down, it is time to get them help. In any case, this kind of burn-down chart gives actionable data to all parties, so that convergence – or lack thereof – is visible early in the project.

Another chart commonly used to show convergence in agile software development is a chart showing the rate at which acceptance tests – and thus features – are being added to the system, and the rate at which these tests have passed. For an example, see Figure 2-7[24]

[24] See Jeffries, *Extreme Programming Installed* (2001) p 139

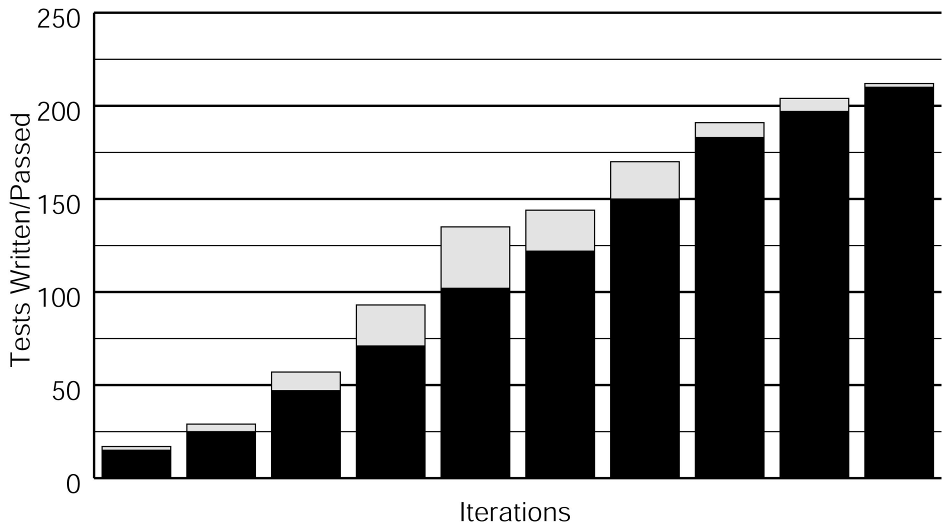

Figure 2-7. Acceptance Tests Written and Passed.

Tool 5 – Synchronization

Iterations are planned by selecting features that are important to customers, and if multiple teams are involved, they generally divide the work by feature. One of the problems with a feature-based approach to software development is that a feature will most likely involve several different areas of the code. Traditionally, the integrity of a module was ensured by having only one developer, who understood it clearly, assigned to work on it. Most agile approaches recommend common ownership of code, although Feature Driven Development (FDD) maintains individual ownership of modules, or classes.[25] Since individual features require several different classes to be modified, FDD forms feature teams consisting of the relevant class owners.

Any place where several individuals are working on the same thing, a need for synchronization occurs. So in FDD, synchronizing the several people working on a feature is necessary, while common code ownership requires that several people working on the same piece of code must be synchronized. The need for synchronization is fundamental to any complex development process.

[25] Plamer, *A Practical Guide to Feature-Driven Development* (2002) pp 42-44

The same problem occurs in automobile design. A slight change in hood slope for better aerodynamics might have an impact on the shape of the front fenders or the layout of components under the hood. When things get complicated in automotive design, there is no substitute for building a mock-up to see how things actually fit together. Toyota builds far more prototypes than most other automakers, because they are such an effective way to rapidly synchronize the efforts of many people.

Synch and Stabilize[26]

In a software development environment with collective code ownership, the idea is to build the system every day, after a very small batch of work has been done by each of the developers. In the morning, developers check out source code from a configuration management system, make changes, test their changes in a 'private build,' check to see if anyone else has made change to the same code, and if so, check for conflicts, then check in the new code. At the end of the day, a build takes place, followed by a set of automated tests. If the build works and the tests pass, the developers have been synchronized. This technique is often called the *daily build and smoke test*.

There are many variations on this theme: a build might occur every few days, or it might run every time new code is checked in. More frequent builds are better; they provide much more rapid feedback. Builds and build tests should be automated. If they are not, the build process itself will introduce errors and the amount of manual work will prohibit sufficiently frequent builds.

Sometimes the build is of the whole system; sometimes only subsets of the system are built, because the whole system is too large. Sometimes an entire suite of tests is run, sometimes, especially when tests are manual, only some tests are run. The general principle is that if builds and test suites take too long, they will not be used, so invest in making them fast. This provides a bias toward more frequent builds but less comprehensive tests, but it is still important to run all the tests overnight or every weekend.

A standard approach to keeping automated tests reasonable in size is to stub-out or simulate slow layers to keep up the speed. For example, you probably want to stub-out database access and the user interface. If you are designing software to

[26] See Cusumano, 'How Microsoft Makes Large Teams Work Like Small Teams' (1997)

control a device, you will want to simulate the hardware performance as you develop the system. The *span* of a build and test operation is an important development decision.

If the entire system is not spanned in the daily build and smoke test, full system tests should be run as frequently as possible. Remember the rule of small batches: if you integrate changes in small batches, it will be infinitely easier to detect and fix problems. Keep it simple by doing it as often as possible. The goal should be to have workable code at the end of every day.

Spanning Application[27]

Another way to synchronize the work of several teams is to start by having a small 'advance' team develop a simple spanning application through the system. For example, suppose you are converting an insurance system to a new environment. You might begin by choosing a simple policy type, preferably one with low volume. The advance team develops a spanning application for that type of policy all the way through the system. This includes establishing a new policy, renewing the policy, handling a claim, and terminating the policy. If possible, the spanning application should go into production when it's done.

Once the spanning application is developed, you have in effect driven a nail through the system, sort of like a carpenter positioning a piece of wood. When the spanning application is proven in production, you know you have a workable approach. At this point, multiple teams can use the same approach and drive in many nails at the same time.

A spanning application works well to test various commercial components. Say you have three possible vendors for middleware, and you are not quite sure which one will really work in your environment. By having a small team build a simple spanning application, you can get a real understanding of the strengths and weakness of each possibility before you commit to any single solution.

[27] There are several different names used for a spanning application. The description here is modeled after the ***Thread*** described Simons, 'Big and Agile?' (2002). Hunt, *The Pragmatic Programmer* (2000) p 48-52 calls the same concept a ***Tracer Bullet***, Cockburn uses the term ***Walking Skeleton***, and Hohmann, *Beyond Software Architecture* (2003) calls it a ***Spike.*** In Jeffries, *Extreme Programming Installed* (2001) a ***Spike*** is an experiment to validate an estimate.

Matrix

A more traditional approach to synchronizing multiple teams is to sketch out an overall architecture, and then have teams develop separate components or subsystems. This approach is particularly appropriate when the different teams are not located in the same place, because it allows them to go about their work with a minimum of communication with other teams. The problem, of course, comes at the interfaces. When the various team's components have to work together, high bandwidth communication is usually necessary to resolve the many detailed design issues involved. Moreover, if the teams have already developed their subsystems, they are not going to be eager to change what they have done.

Therefore, the matrix approach starts by developing the interfaces, and then the subsystems. All points of cross-team interaction should be laid out at the beginning; teams should be assigned to each of these interaction points. The interface should be developed first, stubbing-out the components to allow the cross-component software to be demonstrated. After the interfaces are working, the component teams can work reasonably independently to develop their subsystem. But they should be integrating their code into the full system regularly, to be sure that the interface continues to work.

This approach was used by Motorola to design a new communication system.[28] Teams from around the world were involved, and each team was responsible for developing the software in a single piece of hardware. Before the teams got started with their subsystem designs, they assembled in a single place to study the overall architecture and define the interactions among the devices. Each link between devices, called a *strata,* was identified, and a team, consisting of people from the two device teams in question, was assigned to each strata. This is illustrated in figure 2-8 which shows the strata among devices A, B, C, D, and E.

[28] See Battin, 'Leveraging Resources in Global Software Development' (2001). The *cluster* concept in that paper has been renamed *strata*. This is more fully described in Crocker, *Large Scale Agile Software Development* (2003)

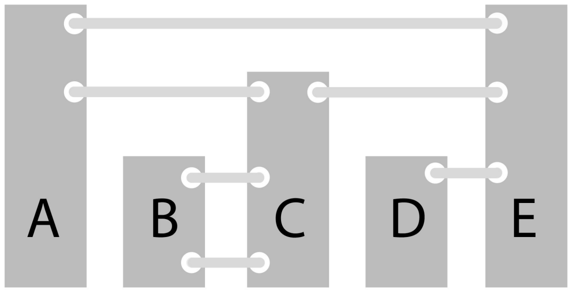

Figure 2-8. Implement interfaces first

Each strata was developed and validated independently, focusing principally on the interactions across devices. They did this by stubbing-out the interaction of the strata with the individual devices, and focusing on the cross-device communication first. As the various strata reached some level of maturity, they were integrated into the devices. This 'internal' integration was the easy part, since each device team was co-located in a particular country, and members were used to working together.

The beauty of this approach is that the highest risk areas which were likely to cause the biggest delays and create the biggest communications problems were the inter-team interactions; these were resolved at the beginning of the project, when there was plenty of time and there was no prior code to change. The easier part, the device integration, was saved for later in the project. This technique provided superior synchronization throughout the project, because a team could integrate into the overall structure regularly, making sure that whatever they did inside their team did not compromise the overall system.

Tool 6: Set-Based Development

Set-Based vs. Point-Based

Let's say you want to set up a meeting. There are two ways to go about it, you can use a point-based or a set-based approach. Figures 2-9 illustrates the point based approach: first you choose a meeting time and then you refine it until it works. Unfortunately, it may take several iterations to find an acceptable meeting time, and the process may never converge. Figure 2-10 illustrates the set-based approach: you start by defining everyone's constraints and then select a meeting time that fits within those constraints. This approach involves considerably less communication, yet it quickly converges on an acceptable meeting time.

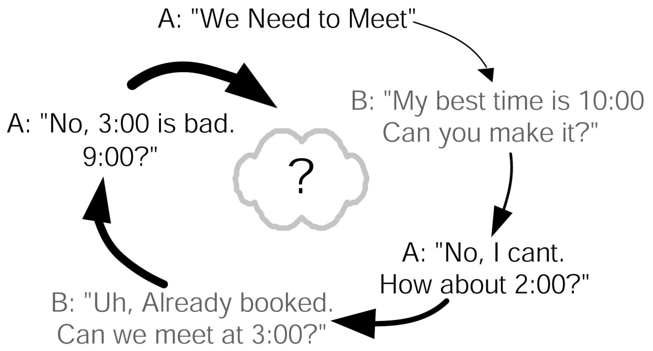
A: "We Need to Meet"
B: "My best time is 10:00 Can you make it?"
A: "No, I cant. How about 2:00?"
B: "Uh, Already booked. Can we meet at 3:00?"
A: "No, 3:00 is bad. 9:00?"
?

Figure 2-9. Point-Based Scheduling[29]

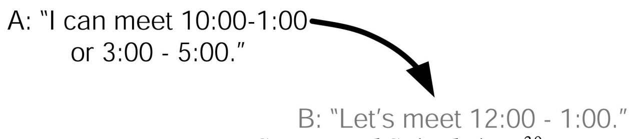
A: "I can meet 10:00-1:00 or 3:00 - 5:00."
B: "Let's meet 12:00 - 1:00."

Figure 2-10. Set-Based Scheduling[30]

In set-based development, communication is about constraints, not choices. This turns out to be a very powerful form of communication, requiring significantly less data to convey far more information. In addition, talking about constraints instead of choices defers making choices until they have to be made, that is, until the *Last Responsible Moment*, which we discuss in Chapter 3.

Let's consider how constraint-based communication can speed up large-scale product development. Durward Sobek studied Toyota and Chrysler product development approaches for his 1997 thesis at the University of Michigan.[31] He found that a primary engineering discipline at Toyota is to maintain and refer to *checklists* which record known trade-offs and constraints.

For example, a styling engineer, might want a rear fender section with a dramatic new look. However, the manufacturing engineer might suspect that the new

[29] Diagram from Durward Sobek. Used with permission.

[30] Diagram from Durward Sobek. Used with permission.

[31] Sobek, *Principles That Shape Product Development Systems: A Toyota-Chrysler Comparison* (1997)

design is going to be difficult to manufacture. Instead of expressing a vague doubt, the manufacturing engineer would send the styling engineer a checklist showing it the time it takes to stamp body panels with certain characteristics, and detailing the limits of those characteristics. The checklist isn't necessarily a list, it is often a graph of the boundary conditions similar to Figure 2-11. The styling engineer would examine the checklist along with many similar checklists and come up with two or three designs that take all of the constraints into consideration.

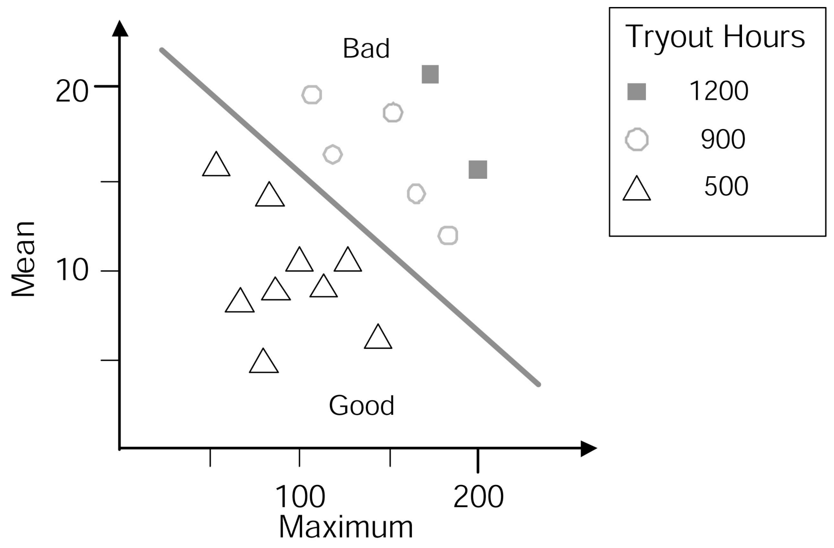

Figure 2-11. Checklist: Rear Quarter Panel Cross Section Deformity Ratio[32]

If you were a manufacturing engineer at Chrysler, it is more likely that the styling engineers would send you one or two possible styles and ask for comments. You would respond that you think the panel is going to be difficult to manufacture. At the same time, many other engineers would have problems with the proposed style, so meetings would be called to resolve the issues. However, once you get a style you think you can manufacture, perhaps the design of the gas cap will might have gotten difficult or maybe there is not enough room left for all of the targeted wheel sizes. More meetings are needed to iron these problems out, which will no doubt lead to more problems. A never-ending game ensues, reminiscent of point-based meeting scheduling.

[32] Diagram from Durward Sobek. Used with permission.

Figures 2-12 and 2-13 show how the two approaches work:

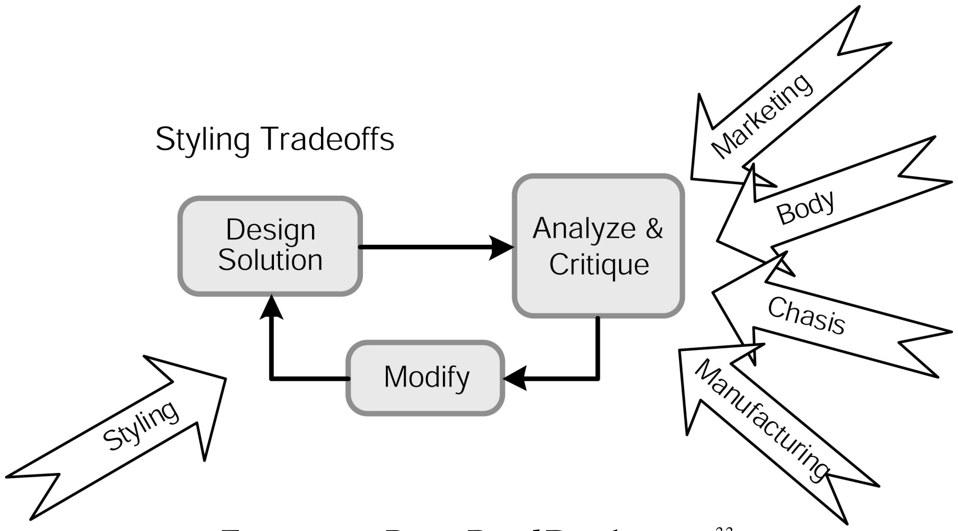

Figure 2-12. Point-Based Development[33]

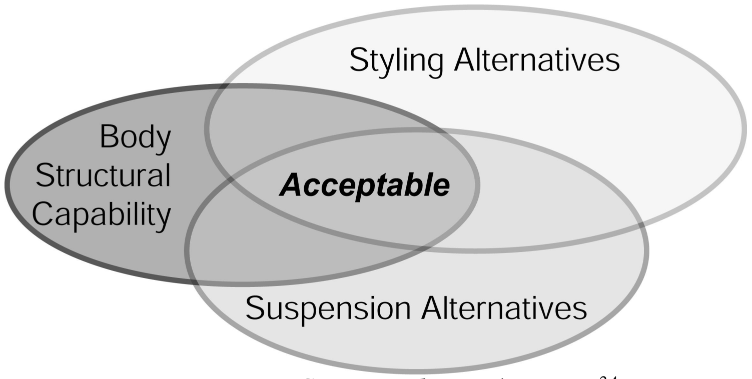

Figure 2-13. Set-Based Development[34]

Toyota explores a large number of concepts at the beginning of a vehicle program, expending significantly more resources than other automakers. It maintains a large number of options throughout the development process and produces an extraordinarily number of prototypes of subsystems and clay models of vehicles. Final body dimensions are fixed far later in the development process

[33] Diagram from Durward Sobek. Used with permission.

[34] Diagram from Durward Sobek. Used with permission.

than other automakers, and final specifications are released to suppliers very late in the development process. The quality, popularity, and profitability of the cars it produces indicate that Toyota's development process is highly effective.[35]

Set-Based Software Development

So how do you apply set-based development to software? You develop multiple options, communicate constraints, and let solutions emerge.

Develop Multiple Options

When you have a difficult problem, try this: develop a set of alternative solutions to a problem, see how well they actually work, and then merge the best features of the solutions or choose one of the alternatives. It might seem wasteful to develop multiple solutions to the same problem, but set-based development can lead to better solutions faster, as the examples in the sidebars illustrate.

Set-Based Embedded Software Development

A software development manager from a medical device company described to us how he runs a new program:

The first thing I do is have the user champion describe to a group of people what problem needs to be solved. Now, I don't think anyone can put into words what they really want, so I set a team to working on maybe a half dozen possibilities. This is the first iteration, and it lasts a month. Then I have the developers show the champion their work, and we can narrow down what is really needed to a couple of the prototypes. At this point, I reduce the team size and have some developers continue developing the most promising options for the next iteration. Based on this work, the champion can usually let the developers know exactly what is needed, and by that time, the work is better than half-done. The champion is always happy and we get results very fast.

[35] Ward, 'The Second Toyota Paradox' (1995)

Set-Based Technology Selection

A friend from a company that does enterprise applications told us how he made a critical decision:

We had to choose a technical platform for a system. However, it was not clear which of the three available options was going to be the winner, let alone meet our needs. So we started developing on all three. This required the underlying development to be a bit more general than otherwise, but it turned out to be quite robust because of that. It was really not necessary to decide on a platform until quite near to the end of the project, and by that time, the correct choice was pretty obvious, but it was not the one we would have made in the beginning.

Set-Based Web Site Design:

A colleague from a company that does web designs for many customers told us how she answers difficult usability questions:

When we can't agree on how to structure the web site, what we do is create two or three versions, with different paths and page layouts. We then do usability testing with several target users. It turns out that there is never one design that stands out above the others. Instead, we find that some features from each design are good, and some are rather poor. We put together the best features of all the options, and retest. Invariably we get a far better usability score with the combination. We're thinking that we should design all of our sites this way.

Set-based development does not replace iterative development, it adds a new dimension. During early iterations, multiple choices are developed for key features; in later iterations, they are merged or narrowed to a single choice.

Communicate Constraints

Set-based development means that you communicate constraints, not solutions. On the surface, this might seem to be the opposite of using an iterative approach. Since you are supposed to produce working, deployable code with each iteration, an iteration might seem like a point-based solution, the opposite of set-based development.

Thinking of an iteration as a point-based solution is a misinterpretation of iterative development. In an iteration, you only implement the minimum amount of functionality necessary to demonstrate the core concepts of that iteration. For example, you do not start with an entire database design in the first iteration; you use a simple persistence layer to deal with the current subset of features. The design will evolve, and in that sense, the early iteration is a prototype of a piece of the overall design.

In Chapter 6 we will discuss refactoring, that is, restructuring the code as the design evolves. Aggressive refactoring is the key to making sure that iterative development converges on a solution. When an iteration implements 'frozen' code that is not available for refactoring, then it is a point-based solution and can lead to the same circular iterations we saw in point-based meeting scheduling. When an iteration implements a design that is available for refactoring, then the design is an instance of a range of options that can be refined later in development, similar to a prototype in set-based development.

An iteration should be considered a demonstration of a possible solution; it should not be considered the only solution. Early iterations should leave wide latitude for implementing the rest of the system in many possible ways. As iterations progress and more choices are made, the design space should be gradually narrowed.

Let the Solution Emerge

Communicating constraints is very useful when tackling a particularly difficult problem, because it helps assure that the solution is worked out by all concerned. As the group grapples with the problem, resist the temptation to jump to a solution; keep the constraints of the problem visible so that the team can discover the intersection of the design space that will work for all concerned.

The True Story of A Death March Project – Part 4: A Solution Emerges

The team had a disagreement on how to translate the data from the old database to the new database. It was necessary to use a new database key, but when customers sent in changes for legacy data, the new key would not be available to the data entry clerk. There was a raging debate between the people who understood the legacy database structure, those designing the new database, the people designing the new GUI and the managers of the data entry clerks.

I had each of the interested parties list out the range of options that could work in their area, instead of their preferred solution. We had a series of meetings where each group presented their constraints, rather than their solutions, to the others. At each meeting, we would toss out any ideas that were completely unworkable, and then break for a couple of days while each group reevaluated their options. At first, the options expanded rather than contracted, because each group found that ideas from other groups expanded their idea of what might work. Every few days, the groups met again and repeated the process.

The solution that emerged was novel and very well thought out. It wasn't something that anyone would have thought of at the beginning, but it was probably about the only thing that would have worked. Once everyone agreed on the approach at a fine level of detail, the development team mounted a massive effort to implement it quickly. Despite the many changes involved, this was one area of the code that worked from the first day it was released.

–Mary

Try This

1. Take your most difficult problem and devise a way to increase feedback:
   a. Increase the feedback of development teams to management by asking each team at the end of each iteration:
      i. Was the team properly staffed for this iteration?
      ii. Were there any needed resources that were not forthcoming?
      iii. How can things be changed to make things go better and faster?
      iv. What is getting in the way?

b. Increase the feedback of customers to development teams by holding a customer focus group at the end of each iteration. Ask questions such as:

   i. How well does this section solve the problem it was meant to solve?

   ii. How could it be improved?

   iii. How does this iteration affect your view of what you need?

   iv. What do you need to put this part of the system into production?

c. Increase the feedback of the product to the development team by:

   i. Having the team write and run unit tests as they write the code.

   ii. Having the testing team write and run acceptance tests as the developers work on the code.

d. Increase the feedback within the team by:

   i. Making testers an integral part of the development team

   ii. Involving operations people at the beginning of the project

   iii. Establishing the policy that the development team maintains the product

2. Start iterations with a negotiation session between customers and developers. Customers should indicate which features are the highest priority and developers should select and commit to only those features from the top of the priority list which they can realistically expect to complete in the iteration time-box.

3. Make a progress chart for your current project so the team can see what needs to be done and everyone can see how the project is converging.

4. If you divide a system across multiple teams, make every effort to have a divisible architecture which allows teams to work on their own areas as independently as possible. Find ways for multiple teams to synchronize as often as possible by integrating their code and running automated tests.

5. If stratus teams work for machine interfaces, consider them four user interfaces also. If you have several teams working on different components of a system, consider forming strata teams focused on user interfaces that cross components.

6. Find your toughest outstanding development problem and have the development team come up with three options on how to solve it. Instead of choosing one of the solutions, have the team explore all three options at the same time.