HOCHSCHULE DARMSTADT
UNIVERSITY OF APPLIED SCIENCES

h_da

Agile Software Development

Part 5: Lean Software Development

Prof. Dr. A. del Pino

Lean Software Development

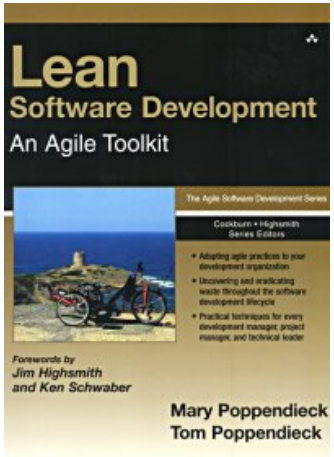
Can we apply lean principles on software development ?

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 translated 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*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003



Recommended Literature:

M. Poppendieck, T. Poppendieck. *Lean Software Development – An Agile Toolkit*. Addison-Wesley, 2003

Lean Software Development

Lean development principles

Lean development principles have been tried and proven in the automotive industry, which has a design environment arguably as complex as most software development environments.

Moreover, the theory behind lean development borrows heavily from the theory of lean manufacturing, so *lean principles in general are both understood and proven* by managers in many disciplines outside of software development.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean development principles

1. **Eliminate waste:** Spend time only on what adds real customer value.
2. **Amplify learning:** When you have tough problems, increase feedback.
3. **Decide as late as possible:** Keep your options open as long as practical, but no longer.
4. **Deliver as fast as possible:** Deliver value to customers as soon as they ask for it.
5. **Empower the team:** Let the people who add value use their full potential.
6. **Build integrity in:** Don't try to tack on integrity after the fact – build it in.
7. **See the whole:** Beware of the temptation to optimize parts at the expense of the whole.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean principle: Eliminate waste

Toyota transferred its concept of waste from manufacturing to product development. When a development project is started, the goal is to complete it *as rapidly as possible*, because *all of the work that goes into development is not adding value until a car rolls off the production line* [...]

In a sense, *ongoing development projects* are just like *inventory* sitting around a factory. [...]

Eliminating waste is the most fundamental lean principle, the one from which all the other principles follow. Thus, the first step is to uncover the biggest sources of waste and eliminate them. The next step is to uncover the biggest remaining sources of waste and eliminate them. The next step is to do it again. After a while, even things that seem essential can be gradually eliminated. [...]

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 1 – Seeing waste

Learning to see waste is the first step in developing breakthroughs with lean thinking. If something does *not directly add value* as perceived by the customer, *it is waste*. If there is a way to do without it, it is waste. [...]

The Seven Wastes of Manufacturing	**The Seven Wastes of Software Development**
Inventory	Partially Done Work
Extra Processing	Extra Processes
Overproduction	Extra Features
Transportation	Task Switching
Waiting	Waiting
Motion	Motion
Defects	Defects

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Partially done work

> Partially done software development has the tendency to become *obsolete*, and it gets in the way of other development that might need to be done. But the big problem with partially done software is that you might have no idea *whether or not it will eventually work*.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

- Partially done work a.k.a. work-in-process

> The paradigm shift required for software is to see that the *paper specifications and unfinished programs are in effect stocks or work-in-process*.
>
> The real cost is not the storage of the paper; the real cost is in the *errors that remain undiscovered* in the documentation until later.
>
> By that time the authors have forgotten all the rich details of the situation they were describing and the project has moved on; *so correcting the errors will be far more expensive*.
>
> Source: P. Middleton: *Lean Software Development: Two Case Studies.* Software Quality Journal, Vol. 9, pg. 241-252, 2001

Lean Software Development

Extra processes

Paperwork that no one cares to read adds no value. [...]

A good test of the value of paperwork is to see *if there is someone waiting* for what is being produced.

If an analyst fills out templates, makes tables, or writes use cases that others are eager to use - for coding, testing, and writing training manuals - then these probably add value.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Extra Features

Developers might like to add a new technical capability just to see how it works. This may seem harmless, but on the contrary, it is serious waste.

Every bit of code in the system has to be tracked, compiled, integrated, and tested every time the code is touched, and then it has to be maintained for the life of the system.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Task switching

Assigning people to multiple projects is a source of waste. Every time software developers switch between tasks, *a significant switching time is incurred* as they get their thoughts gathered and get into the flow of the new task.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

See also: T. DeMarco, *Peopleware*, chapter 10: Brain time versus body time.

Waiting

One of the biggest wastes in software development is usually *waiting for things to happen*. Delays in starting a project, delays in staffing, delays due to excessive requirements documentation, delays in reviews and approvals, delays in testing, and delays in deployment are waste.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Motion

Development is an activity that requires great concentration, so walking down the hall takes a lot more time than you might think. It will probably take the developer several times as long to *reestablish focus* as it took to get the question answered.

It is for this reason that agile software development practices generally recommend that a team work *in a single workroom* where everyone has access to developers, to testers, and to customers or customer representatives.

Source: M. Poppendieck, T. Poppendieck: Lean Software Development – An Agile Toolkit, Addison-Wesley, 2003

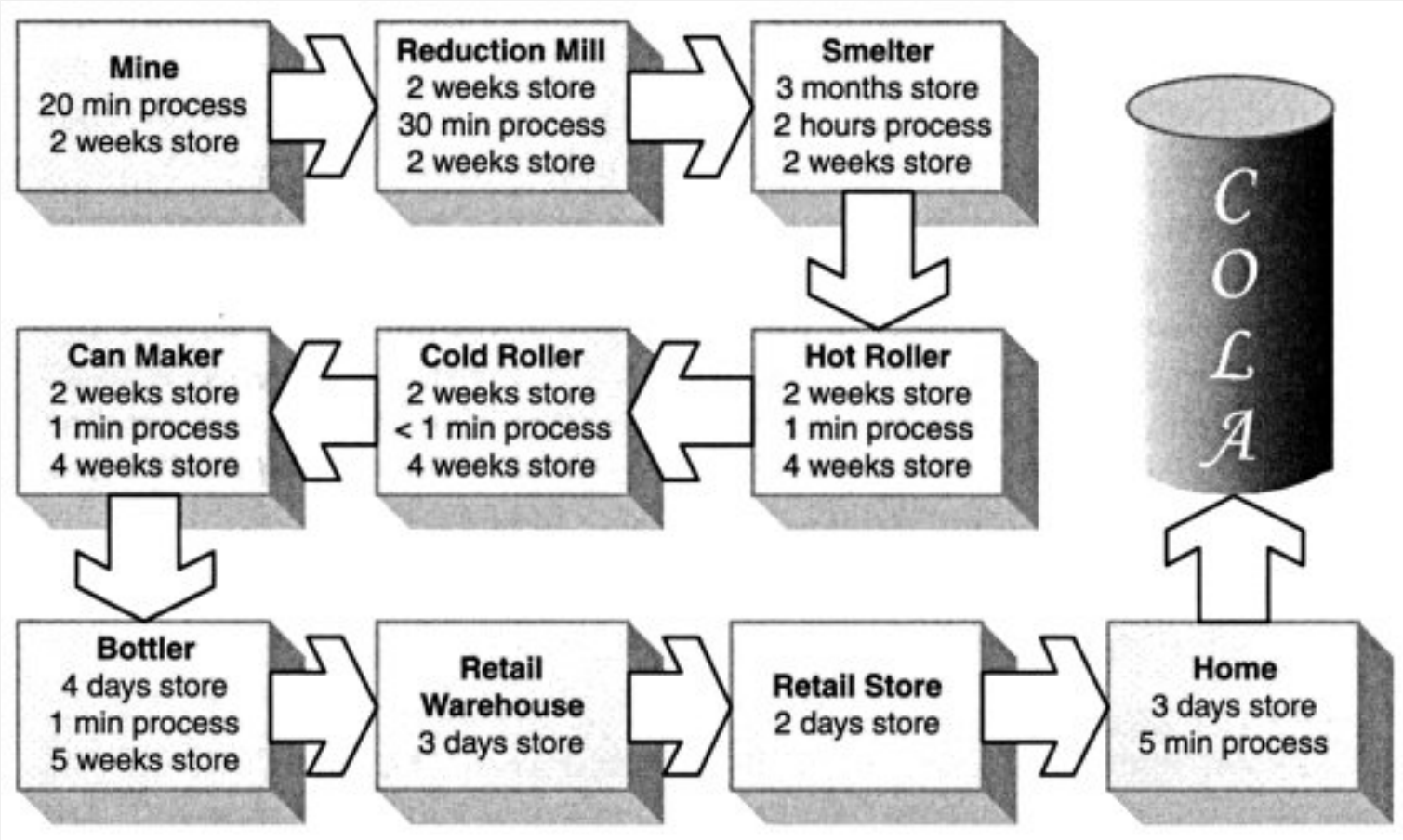
Defects

The amount of *waste caused by a defect* is *the product of the defect impact and the time it goes undetected*. [...] The way to reduce the impact of defects is to find them as soon as they occur. Thus, the way to reduce waste due to defects is to test immediately, integrate often, and release to production as soon as possible.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 2: Value stream mapping



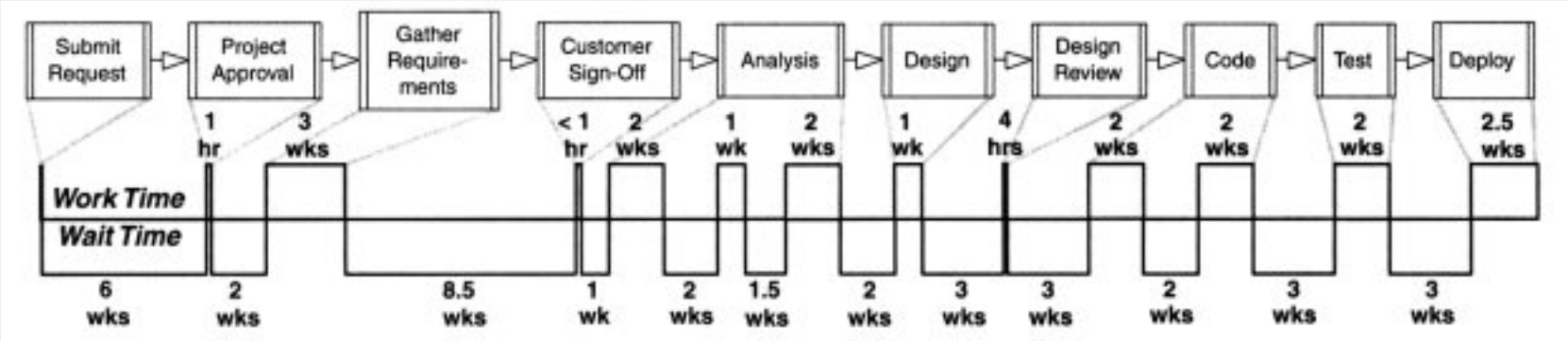
[...] it takes a cola can an average of *319 days* to move from mine to consumption, while the *processing time* – the time that value is actually being added – *is only 3 hours*, or 0.04 percent of the total time.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

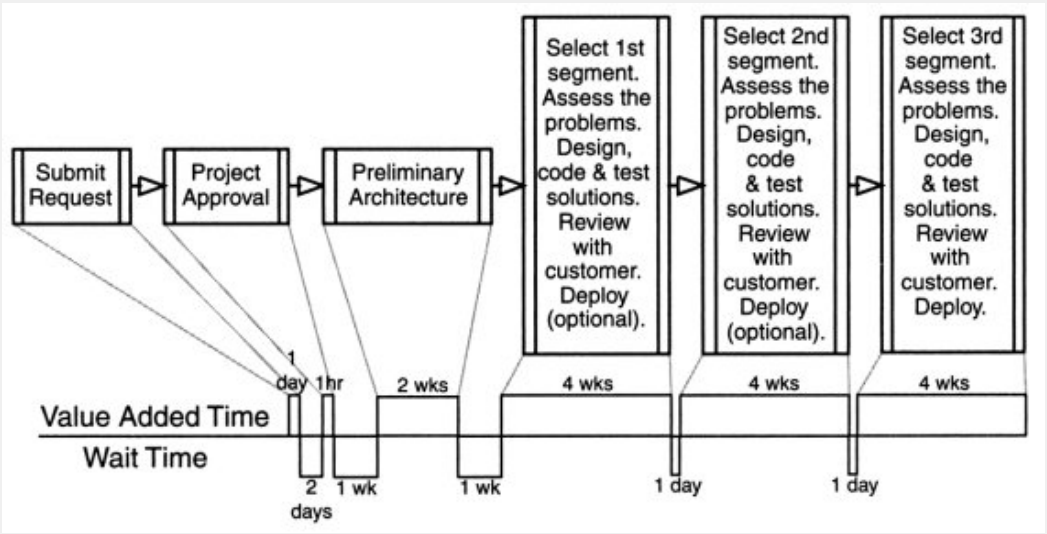
Lean Software Development

Value stream mapping

This map shows that an *average project is ready to deploy in a year*, with about *a third of the time spent on value-adding activity*.



Agile value stream map:



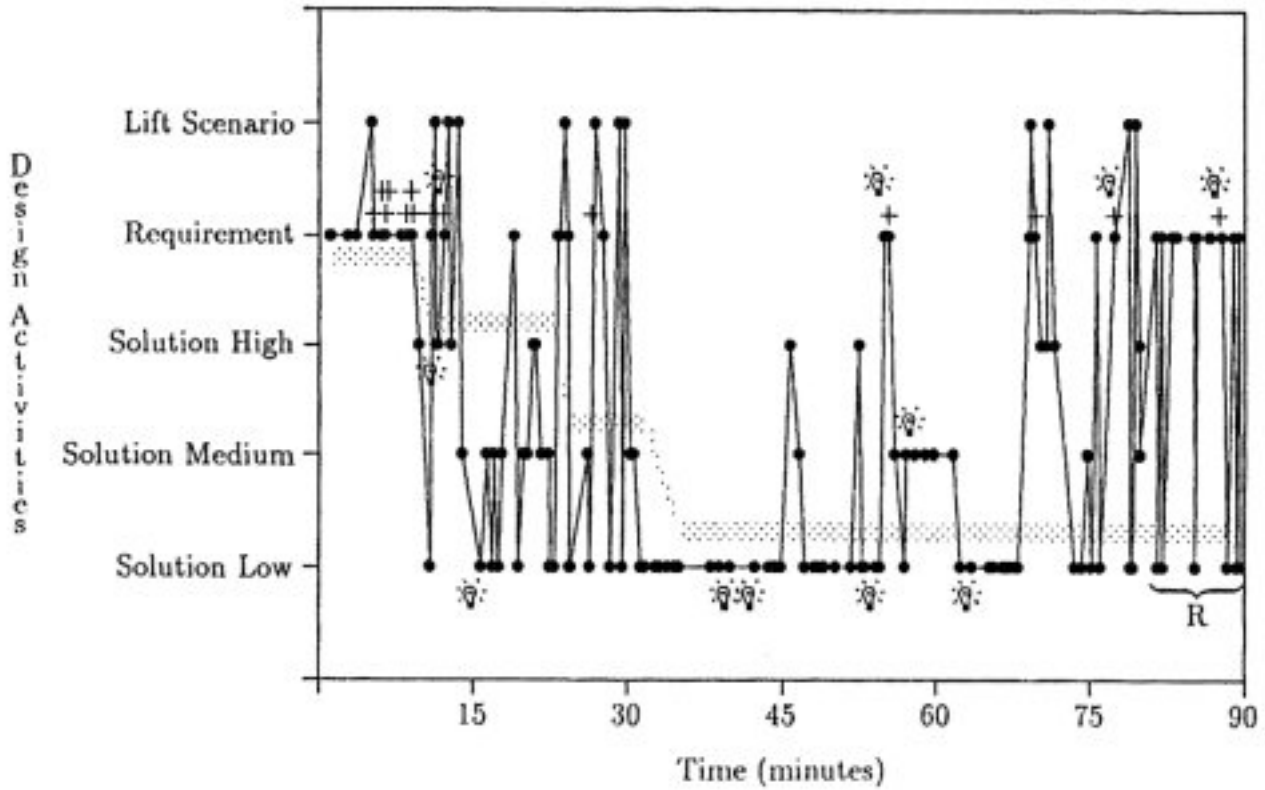
Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean principle: Amplify learning

Raymonde Guindon, 1990: Designing an elevator control system

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.



Source: R. Guindon: *Designing the Design Process: Exploiting Opportunistic Thoughts*. In Human-Computer Interaction, 5 (2) p. 305-344, cit. in: M. Poppendieck, T. Poppendieck: *Lean Software Development*, Addison-Wesley, 2003

Lean Software Development

Learning cycles

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 though such *learning cycles*. [...]

There are two schools of thought in developing software. One is to encourage developers to be sure that each design and 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 important question in development is, *How can I learn most effectively?* The answer is often to have many short learning cycles.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 3: Feedback

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 inhouse and test them.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

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* [...].
>
> 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*.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

- For iteration planning see Ken Schwaber: *Agile Project Management with Scrum.*

Lean Software Development

Iterations, convergence, and negociable scope

> 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. [...]
>
> A good strategy for achieving convergence is to work on *top priority items first*, leaving the low priority items fall off the to-do list. [...]
>
> 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.
>
> It is therefore 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 negociable in detail.
>
> With negociable scope, iterative development will generally converge.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

- Use burndown charts to visualize and communicate convergence. See also Ken Schwaber: *Agile Project Management with Scrum*.

Lean Software Development

Tool 5: Synchronization

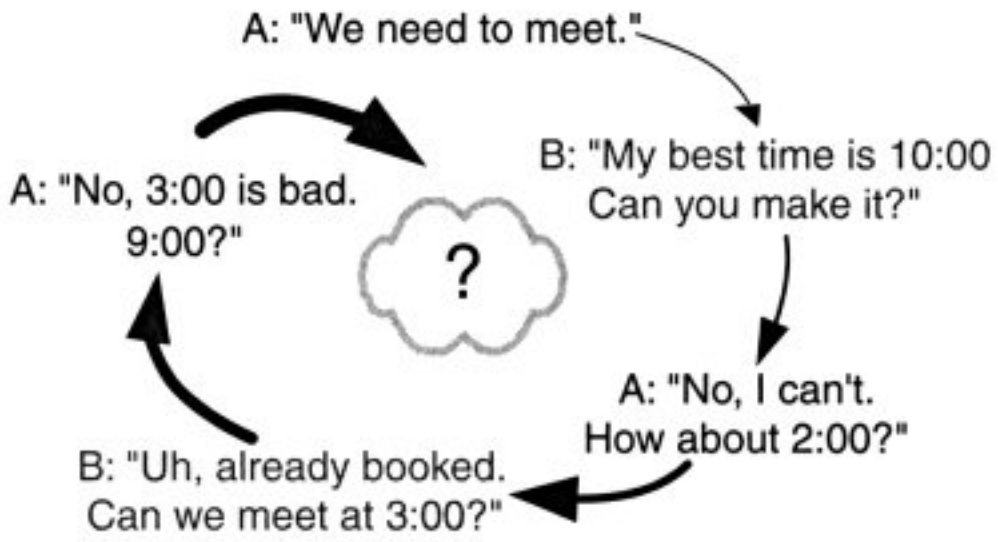
> Most agile approaches recommend *common ownership of code*, although Feature-Driven Development (FDD) maintains individual ownership of modules, or classes [...].
>
> Whenever serveral individuals are working on the same thing, a *need for synchronization* occurs.
>
> So in FDD, synchronizing the several people working in 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.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

- Synch and stabilize through frequent builds. See also Kent Beck: *Extreme Programming Explained*.

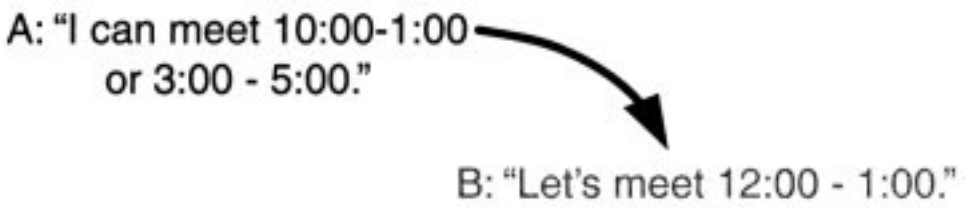
Lean Software Development

Tool 6: Set-based development

Point-based meeting scheduling



Set-based meeting scheduling



Source: Diagrams adapted from D. Sobek, cit. in: M. Poppendieck, T. Poppendieck: *Lean Software Development*, Addison-Wesley, 2003

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* [...].

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Set-based software development

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

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Develop multiple solutions

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 [...].

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

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. [...]

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* [...].

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.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

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.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean principle: Decide as late as possible

An example

When sheet metal is formed into a car body, a massive stamping machine presses the metal into shape.

The stamping machine has a huge metal die, which makes contact with the sheet metal and presses it into the shape of a fender, door, or another body panel.

Designing and cutting dies to the proper shape accounts for half of the capital investment of a new car development program and drives the critical path.

If a mistake ruins a die, the entire development program suffers a huge setback. If there is one thing that automakers want to do right, it is the die design and cutting.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

die = Stempel, Pressform. fender = Kotflügel

Lean Software Development

Lean principle: Decide as late as possible

An example...

The problem is, as the car development progresses, engineers keep making changes to the car, and these find their way to the die design.

No matter how hard the engineers try to freeze the design, they are not able to do so.

In Detroit in the 1980s the cost of changes to the design was 30 to 50 percent of the total die cost, while in Japan it was 10 to 20 percent.

These numbers seem to indicate the Japanese companies must have been much better at preventing change after the die specs were released to the tool and die shop.

But such was not the case.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean principle: Decide as late as possible

An example...

> The U.S. strategy for making a die was to wait until the design specs were frozen, and then send the final design to the tool and die maker, which triggered the process of ordering the block of steel and cutting it.
>
> Any changes went through an arduous change approval process.
>
> It took about two years from ordering the steel to the time the die would be used in production.
>
> In Japan, however, the tool and die makers order up the steel blocks and start rough cutting at the same time the car design is starting.
>
> This is called concurrent development.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Concurrent software development

Programming is a lot like die cutting. The stakes are often high, and mistakes can be costly, so sequential development, that is, establishing requirements before development begins, is commonly thought of as a way to protect against serious errors.

The problem with sequential development is that it forces designers to take a *depth-first rather than a breadth-first* approach to design.

Depth-first forces making low-level dependent decisions before experiencing the consequences of the high-level decisions.

The most costly mistakes are made by forgetting to consider something important at the beginning. The easiest way to make such a big mistake is *to drill down to detail too fast*.

Once you set down the detailed path, you can't back up and are unlikely to realize that you should. When big mistakes may be made, it is best to survey the landscape and delay the detailed decisions.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Moving to concurrent software development

Moving from sequential development to concurrent development means you *start programming the highest value features* as soon as a high-level conceptual design is determined, even while detailed requirements are being investigated.

This may sound counterintuitive, but think of it as an exploratory approach that permits you to learn by trying a variety of options before you lock in on a direction that constrains implementation of less important features.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 7: Options thinking

A hotel reservation is an option on a hotel room in the future. The price of the option is the cost to make the reservation, which may include a reservation fee.

If you exercise the option - if you show up at the hotel - you pay the price negociated at the time the reservation was made. If you cancel the trip all you lose is the reservation fee.

[...]

Options thinking is an important tool in software development as long as it is accompanied by recognition that *options are not free* and it takes expertise to know which options to keep open.

Options do not guarantee success; they set the stage for success if the uncertain future moves in a favourable direction.

Options allow fact-based decisions based on learning rather than speculation.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 8: The last responsible moment

Concurrent software development means starting developing when only partial requirements are known and developing in short iterations that provide feedback that *causes the system to emerge*.

Concurrent development makes it possible to delay commitment until the last responsible moment, that is, the moment a which failing to make a decision eliminates an important alternative.

If commitments are delayed beyond the last responsible moment, then decisions are made by default, which is generally not a good approach to making decisions.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

How does e.g. the usage of interfaces in OO languages relate to this topic ?

Lean Software Development

Tool 9: Making decisions

Gary Klein studied decision making of emergency responders, military personnel, airline pilots, critical-care nurses, and others, to see how they make life-and-death decisions. [...]

When he started the study, he was amazed to discover that fire commanders felt they rarely, if ever, made decisions.

Fire commanders were very experienced, or they would not have their jobs.

They claimed that they just know what to do based on their experience; there was no decision making involved.

We call this *intuitive decision making*.

[...]

It is much more important to *develop people with the expertise to make wise decisions* than it is to develop decision-making processes that purportedly think for people.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Simple rules

Simple rules for development, such as these seven lean principles:

> It is not so important that the rules give detailed guidance; it is important that people know that these rules are guidelines, which gives them the freedom to make their own decisions.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean principle: Deliver as fast as possible

Customers like rapid delivery. [...]

Even as its customers are realizing the benefits of rapid delivery, savvy businesses are saving money. Rapid delivery means companies can deliver *faster than customers can change their minds*.

It means that companies have *fewer resources tied up in work-in-process*, whether inventory or partially done development.

When work-in-process represents risk, rapid delivery reduces risk.

[...]

For example, Dell Computer believes that *inventory obsolescence is its biggest risk*, so Dell waits until it receives an order and then makes and ships the computer in less than a week.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Complementary principles

[The] principle *deliver as fast as possible* complements *decide as late as possible*.

- The faster you can deliver, the longer you can delay decisions.
- For example, if you can make a software change in a week, then you do not have to decide exactly what you are going to do until a week before the change is needed.
- On the other hand, if it takes you a month to make the change, then you have to decide on the details of the change a whole month before it is due.
- Rapid delivery is an *options-friendly approach* to software development. It lets you keep your options open until you have reduced uncertainty and can make more informed, fact-based decisions.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 10: Pull systems

There are two ways to assure that workers make the most effective use of their time. You can either *tell them what to do* or set things up so *they can figure it out for themselves*.

In a fast-moving environment, only the second option works.

People who routinely deal with fluid situations, such as emergency workers and military personnel, do not depend on a remote commander to tell them how to respond to the latest development. They figure out how to respond to events with the other people who are on the scene.

When things are happening quickly, there is not enough time for information to travel up the chain of command and then come back down as directives.

Therefore, methods for local signaling and commitment must be developed to coordinate work. One of the key ways to do this is to *let customers' needs pull the work* rather than have a schedule push the work.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Kanban

Pull systems use a mechanism called *kanban*, which was originally patterned after restocking grocery store shelves. Kanban means *sign* or *placard* in Japanese.

The interesting thing about pull scheduling is that *it takes the manager out of the loop* of having to tell workers what to do. The work is self-directing. The managers spend their time coaching the team.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

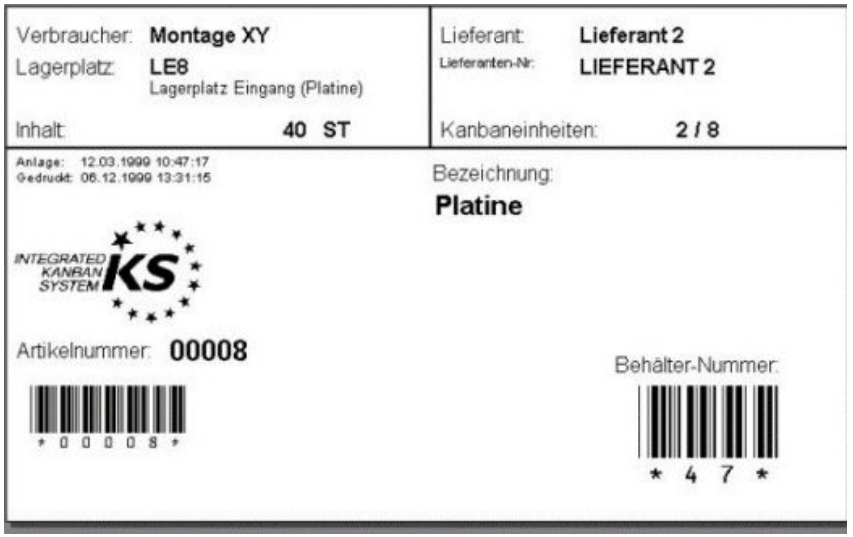


Image source: http://www.ebz-beratungszentrum.de/pps_seiten/KANBAN/KANBAN2.htm

Lean Software Development

Software pull systems

The starting point for a pull system in software development is *short iterations based on customer input at the beginning of each iteration*.

Let's assume that at the beginning of each iteration, the customers or customer representatives write down descriptions of features they need on *index cards*. There are many other ways to document what customers want, but *index cards are a lot like kanban cards* [...].

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

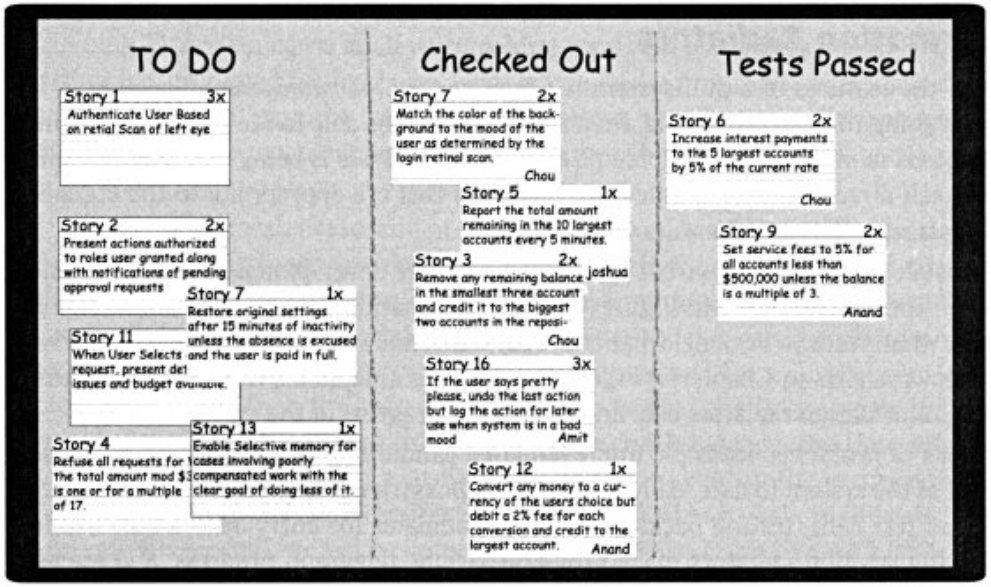


Image source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit.* Addison-Wesley, 2003

Lean Software Development

Visual control

One of the features of a pull system is *visual control*, or management by sight.

If work is going to be self-directing, then everyone must be able to see what is going on, what needs to be done, what problems exist, what progress is being made.

Work cannot be self-directing until simple visual controls that are appropriate to the domain are in place, updated, and used to direct work.

[...] *The kanban board* [...] *is an information radiator* that shows many things: what needs to be done, what is already done, and who is working on what.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 11: Queuing theory

The fundamental measurement of a queue is *cycle time* - that is, the average time it takes something to get from one end of a process to the other.

The cycle time clock starts when something enters a queue and keeps on ticking away while it waits in the queue, while it gets service, while it waits in the next queue, gets the next service, and so on, until it pops out at the other end of the process.

[...]

There are two ways to reduce cycle time; one is to look at *the way work arrives* and the other is to look at *the way work is processed*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Arrival of work

In some systems, it is not possible to influence the rate of arrival of work, but in others, policies can be established to even out incoming demand.

Pricing policies are often used for this purpose. A phone company that offers very low night and weekend rates is doing this to even out peak demand. [...]

One way to control the rate of work arrival is to *release small packages of work*. If you have to wait for a large batch of work to arrive before you can start processing it, then the queue will be at least as long as the whole batch. If the same work is released in small batches, the queue can be much smaller. [...]

Many managers still believe that it is good to group projects into a single priority-setting process to have more projects to compare at one time. Queueing theory suggests that they would probably be *better off releasing projects more frequently* - monthly or even weekly - to even out the arrival of work in the development area.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 12: Cost of delay

Software development is a discovery process in which technical people make *continual tradeoff decisions* in order to reach what they consider an optimal result. [...]

One of the biggest challenges for software development leaders is to assure that the constant tradeoff decisions *being made by everyone* produce an *optimal result*.

All too often, a software development team is told that it *must meet cost, feature, and introduction date objectives simultaneously; there can be no tradeoffs*. This sends two messages to the development team:

- Support costs aren't important because they weren't mentioned.
- When something has to give, make your own tradeoffs.

Since various team members are likely to have different perceptions of what is important, the tradeoffs made by some will probably offset tradeoffs made by others - with the result that *all objectives will be compromised*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Economic model

Give the team an *economic model*, and you have empowered the members to figure out for themselves what is important for the business.

You given everyone the same *frame of reference* so they can all work from the same assumptions.

Finally, the team is more likely to come up with an economic success, since the members now know *what economic success means*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean Principle: Empower the team

An organization that respects software developers as professionals will expect them *to design their own jobs* with proper training, coaching, and assistance.

It will expect them *to improve continually the way they to their work* as part of a learning process.

Finally, it will give them the *time* and *equipment necessary* to do their jobs well.

In a lean organization, the people who add value are the center of organizational energy. Frontline workers have *process design authority* and *decision-making responsibility*; they are the focus of resources, information and training.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 13: Self-determination

Example

> In 1982, General Motors closed its Freemont, California, plant. No one was surprised; the place was a *disaster*. Productivity was among the lowest of any GM plant [...].
>
> Two years later, the same plant was *reopened* by New United Motor Manufacturing, Inc., or NUMMI, a joint venture between Toyota and GM. Toyota managed the plant but was *required to rehire the former GM employees*. [...]
>
> Within two years, NUMMI's productivity was higher than any GM plant - double that of the original plant. [...]
>
> Clearly, something in the management practices made all the difference to NUMMI employees, and those practices have been sustainable. [...]
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Self-determination

Example ...

> Other GM plants have been *unable to copy* the management practices of NUMMI, although other Toyota-managed plants in the United States have successfully done so with similar results. [...]
>
> This first thing the managers at the NUMMI plant did was get *stopwatches for everyone*, and they taught workers how to design their own jobs.
>
> All work at NUMMI is done in *teams of six to eight people*, one of whom is the *team leader*.
>
> The team designs *its own* work procedures, coordinating work standards with teams doing the same work on alternate shifts.
>
> Management's role is to *coach, train, and assist* the teams.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

The FIX-IT group

A development group at a large company (let's call it the FIX-IT group) became frustrated at working in a chaotic environment, so after a particularly difficult delivery, *the group members convinced their manager to give them some time to put some discipline in place*. [...]

As time went on, the FIX-IT group delivered software faster and had *customers who were happier than those of other groups*. The FIX-IT group was also regarded as the *best place to work* by the developers in the complany.

A *vice president decided* that the success of the FIX-IT group should be *replicated* in the rest of the company.

A *staff group* was formed to document the processes used by the FIX-IT group and teach them to the rest of the groups in the company.

The goal was *to create uniform processes* so the company could deliver *consistent results*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

The FIX-IT group...

Not surprisingly, the staff group overlooked the principle behind the practices used by the FIX-IT group - the principle that the developers were responsible for defining [...] *their own practices*.

Since redefining practices did not fit into the goal of uniformity, the staff group considered it a bad habit that would have to stop. [...]

One year later, the company had a *book of documented processes* that even the most inexperienced developer could follow.

Most development groups *ignored* the staff group's efforts [...].

The FIX-IT group *continued to produce better software faster* and had more satisfied customers than any other group.

The *attempt to duplicate* its success elsewhere, however, *was largely a failure*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Similarities between NUMMI and FIX-IT ?

Lean Software Development

Transferring practices

We believe that transferring *practices* from one environment to another is often a mistake.

Instead, one must understand the fundamental *principles behind practices* and transform those principles into new practices for a new environment.

In fact, Toyota *did not* transfer Japanese production practices en masse to NUMMI.

But it did transfer its belief that the *foundation for human respect* is to provide an environment where *capable workers actively participate* in running and improving their work areas and are able to fully use their capabilities.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 14: Motivation

Intrinsic motivation comes from the work we do, from *pride in workmanship* and a sense of helping the customer. Purpose is what makes work energizing and engaging.

People need *more than a list of tasks*. If their work is to provide motivation, they need to understand and commit to the *purpose* of the work.

Intrinsic motivation is especially powerful if people on a team commit together to accomplishing a purpose they care about.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Gaining and holding a sense of purpose

There are many things you can do to help a team gain and hold a sense of purpose:

- Start with a *clear* and *compelling* purpose. [...]
- Be sure the purpose is *achievable*. [...]
- Give the team *access to customers*. [...]
- Let the team make *its own commitments*. [...]
- Management's role is to *run interference*. [...]
- *Keep skeptics away* from the team. [...]

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 15: Leadership

[John] Kotter draws a sharp distinction between *managers* and *leaders* [...]

Managers	**Leaders**
Cope with Complexity	Copy with Change
Plan and Budget	Set Direction
Organize and Staff	Align People
Track and Control	Enable Motivation

[...]

Leaders only flourish in organizations *that want them to be there*. An organization has to value leadership in order to develop leaders.

We notice that organizations that hold technical leaders in high esteem seem to have plenty of these leaders *grow up from the ranks*. [...]

Software development leaders will not flourish in an organization that values process, documentation, and conformance to plan *above all else*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 16: Expertise

Matrix organizational structures are very useful for providing communities of expertise, but even if a company does not use a matrix structure, it is imperative to have *communities of expertise*.

The first step is to identify the *technical* and *domain-specific competencies* that are critical to the organization's success.

These might include competencies such as database administration, user interface design, security, architecture, embedded programming, testing, and safety analysis.

Many companies then create *forums* - monthly meetings, newsletters, speakers - for these communities.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Lean Principle: Build integrity in

> Sometimes you come across software that suits you so well that you think the designer *must have been inside your head*. [...]
>
> If you had to rebuild your computer tomorrow, loaded with only the software you regularly use, *how many products would you load?* If you wiped out all your bookmarks, *which ones would you add back immediately?*
>
> These are the products and services you perceive to be *relevant to your life*, the products with *perceived integrity*.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Examples:

- Google
- IPod

Lean Software Development

Conceptual integrity

Conceptual integrity means that a system's central concepts work together as a *smooth, cohesive whole*.

The components match and work well together; the architecture achieves an effective balance between flexibility, maintainability, efficiency, and responsiveness. [...]

Conceptual integrity is a *prerequisite* for perceived integrity.

When a system does not have a consistent set of design ideas, usability will suffer, because the user does *not* have a single metaphor for the application, strategies for doing the application, and user-interface tactics.

Conceptual integrity *emerges* as the system evolves and matures.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Changes over time

Although conceptual integrity is *necessary* for perceived integrity, it is not sufficient. If the most elegant architecture in the world does not do an exceptional job of meeting users' needs, users will not notice the underlying conceptual integrity.

It is for this reason that *a system's architecture must evolve and mature*; perceived integrity will change over time, and thus the underlying architecture must do so also.

As new features are added to the system to maintain perceived integrity, the underlying capability of the architecture to support the features in a cohesive manner *must also be added*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

What causes integrity ?

Perceived integrity is caused by an *excellent information flow* between *customers and users* on the one side and the *developers* on the other side.

Conceptual integrity is caused by an *excellent information flow within the development team*.

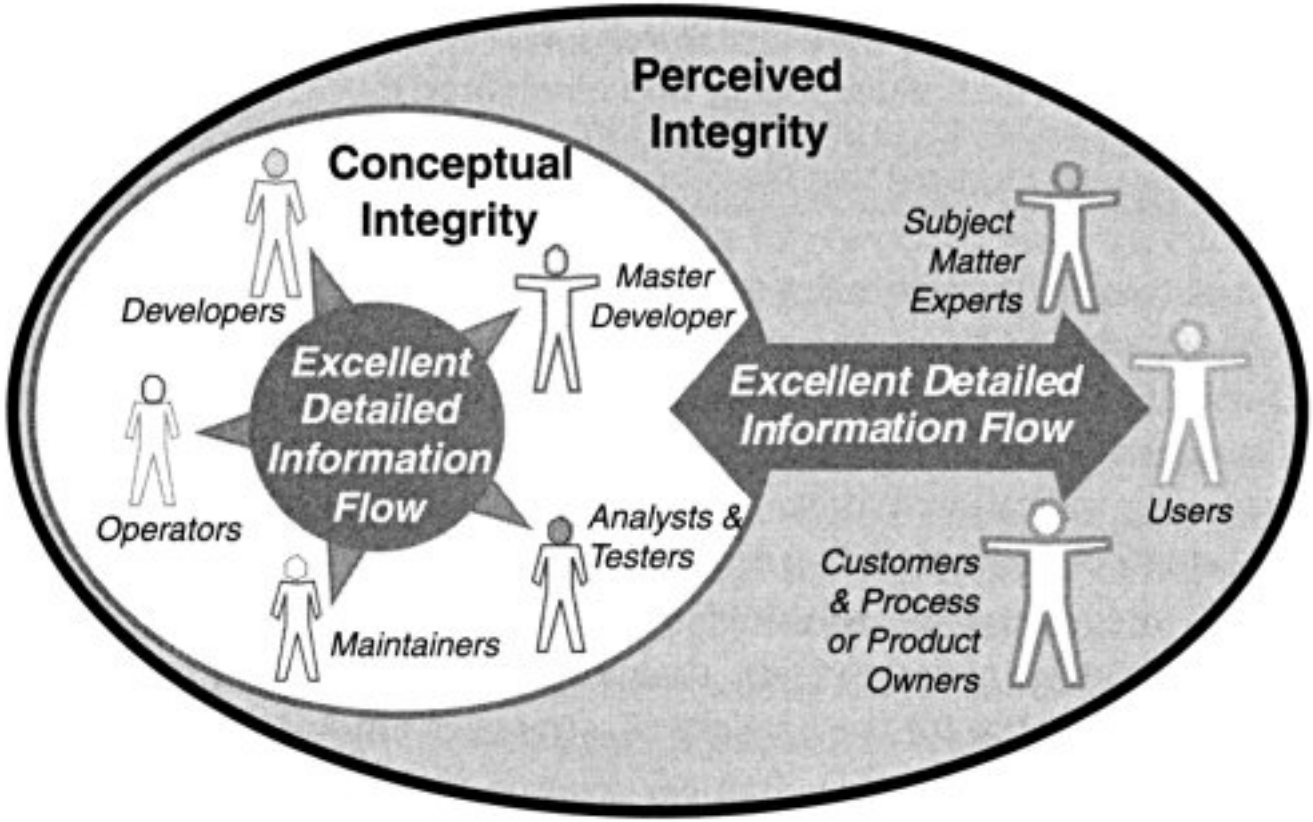


Image source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit.* Addison-Wesley, 2003

Lean Software Development

Tool 17: Perceived integrity

Companies that consistently achieve perceived integrity have a way of constantly *keeping customer values in front of the technical people* making detailed design decisions.

In most Japanese automakers, this is done by a *chief engineer*, who has developed a *vision* of what the target customer segment wants in a car.

The chief engineer spends a lot of time walking around, talking with the engineers as they make tradeoffs, making sure *that these engineers have a good idea of what the customer will find important*.

If the vision of perceived integrity isn't refreshed regularly, the engineers have a *tendency to get lost* in the technical details and forget the customer values.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Examples ?

Lean Software Development

Maintaining perceived integrity

Maintaining *institutional memory* about a system is key to assuring its long-term integrity.

There have been many attempts to use *documentation* created during design to do this. However, design documentation rarely reflects the system as it was actually built, so it is widely ignored by maintainance programmers. If this is the only purpose documentation serves, it was a *waste* to create it.

One way to maintain institutional memory about a system is to *make the developers responsible for ongoing updates*. Alternatively, the developers and maintenance programmers can *work jointly over a period of time* to transfer tacit knowledge. [...]

But the best way to maintain institutional knowledge about a system and keep it maintainable is to *deliver a suite of automated tests along the code*, supplemented by a high-level overview model created at the end of the initial development effort.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 18: Conceptual integrity

Conceptual integrity means that a system's central concepts work together as a smooth cohesive whole. The components match and work well together; the architecture achieves an effective balance between flexibility, maintainability, efficiency, and responsiveness.

The architecture of a software system refers to the way in which the system is structured to provide the desired features and capabilities. An *effective architecture* is what gives a system conceptual integrity.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Key practices to achieve conceptual integrity

There are two *key practices* used by automotive companies to achieve conceptual integrity.

First the *use of existing parts* immediately removes many degrees of freedom and this reduces the complexity and need for communication.

When a new car has novel body styling and a new engine, it helps to use a proven suspension system.

The second practice automotive companies use to achieve conceptual integrity is to use *integrated problem solving* to assure excellent technical information flow. [...]

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Integrated problem solving

Just what does *integrated problem solving* mean in practice? It means that

- *Understanding* the problem and *solving* the problem *happen at the same time*, not sequentially.
- Preliminary information is released early; information *flow is not delayed* until complete information is available.
- Information is transmitted frequently in *small batches*, not all at once in a large batch.
- Information flows in *two directions*, not just one.
- The preferred media for transmitting information is *face-to-face communication* as opposed to documents.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 19: Refactoring

Engineering historian *Henry Petroski* has written extensively about how *design* actually takes place. Engineers start with something that works, learn from its weaknesses, and *improve* the design.

Improvement comes not just from meeting customer demands or adding features; improvements are also necessary because complex systems have *effects that are not well understood at design time*.

Suboptimal choices are an intrinsic part of the process of engineering complex designs in the real world.

It is *not reasonable to expect a flawless design* that anticipates all likely contingencies and cascading effects of simple changes. [...]

The more complex the system, the more important *design evolution* becomes.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

A Reward for developers

An example

> Microsoft keeps the *same team* working on a product such as Excel over multiple releases.
>
> After the team has worked hard to complete a release, the members are rewarded with a couple of months in which they are *allowed to clean up* the underlying structures of the code that bothered them the most while working on the release.
>
> Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Also known as *code maintenance* period.

Lean Software Development

Isn't refactoring rework?

Conventional wisdom holds that sequential development should result in better products with less risk, while overlapping design and development will lead to expensive and time-consuming *rework*.

On the contrary [...] concurrent development usually means results in better, cheaper products, faster and with less risk.

Concurrent development means that the design of the *product emerges* throughout the development process.

Improving a design during the development process is most certainly not rework; it is good design practice.

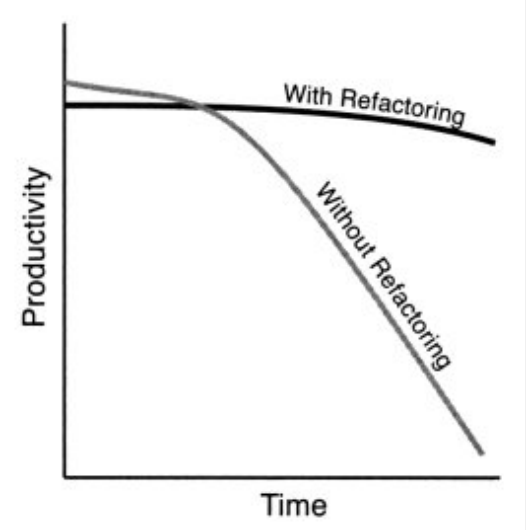
Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Cost of not refactoring

[...] there isn't time *not* to refactor. Work will only go *slower* as the code becomes more complex and obscure.

[...] incurring a *refactoring debt* will kill team productivity. [...]



No one at Toyota would think that stopping a line to find and fix a problem slows things down. They know that focusing on relentless improvement makes the line go *faster*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Tool 20: Testing

When developers write code, there should be a test to be sure that each feature works as intended and that all of the pieces work together. These tests have been categorized as *unit tests, system tests, and integation tests*.

As we move from programming one module at a time to *programming entire capabilities and features*, the distinction between unit, system, and integration tests has less meaning.

A better name for these tests might be *developer tests*, because their purpose is to assure that *code does what the developer intended it to do*.

Tests to be sure that the system does what customers want have been called *acceptance tests*, but this term has traditionally been used to refer to tests that run *at the end* of development.

A better name for tests that make sure that a system does what the customers intend is *customer tests*, since their purpose is to assure that the *system will do what customers expect it to do*. Customer tests are run *throughout* the development, not just at the end.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003

Lean Software Development

Scaffolding

Scaffolding is a supporting framework that allows workers to do things that would otherwise be dangerous.

If you develop software in *iterations*, delay decisions until the *last responsible moment*, and use *set-based* development and *refactoring*, you are going to be making *serious changes to code once it has been written*. [...]

To make changes safely, there must be a way to immediately *find and fix unintended consequences*.

The most effective way to facilitate change is to have an *automated test suite* that tests the mechanisms the developers intend to implement and the behavior the customers need to have.

A good test suite will find unintended consequences right away, and if it is good, it will also pinpoint the *cause of the problem*.

Source: M. Poppendieck, T. Poppendieck: *Lean Software Development – An Agile Toolkit*, Addison-Wesley, 2003