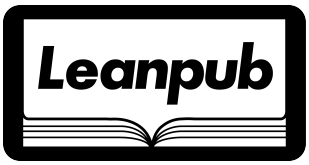
Stephan Schwab

Smarter Software with Activity-Centered Design

2012

Smarter Software with Activity-Centered Design

This version was published on 2012-06-21



This is a Leanpub book, for sale at:

http://leanpub.com/activitycentereddesign

Leanpub helps authors to self-publish in-progress ebooks. We call this idea Lean Publishing. To learn more about Lean Publishing, go to: http://leanpub.com/manifesto

To learn more about Leanpub, go to: http://leanpub.com

Tweet This Book!

Please help Stephan Schwab by spreading the word about this book on Twitter!

The suggested hashtag for this book is #activitycentereddesign.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search/#activitycentereddesign

Contents

Frequent Feedback Desired

Traditionally books have been offered by publishing houses where staff selects and edits what authors send in. On one hand that approach ensures some basic quality. It also works as some filter, as not everything that authors would like to publish actually gets out. On the other hand the traditional approach makes the collection and reaction to feedback from readers hard. It does not prevent feedback but have you ever tried to get in touch with an author of a book you like to dislike?

I have contemplated going with an established publisher for this book. Eventually I decided that to me interactions and collaboration as well as responding to change - key elements of the Agile Manifesto - are more important than the potential burst in recognition and marketing services provided by a well-known publisher.

This book is published initially as an eBook at Leanpub[1]. Leanpub offers books as a subscription. Whenever I update the book you will receive an email so that you can download the latest version.

For conversations about Activity-Centered Design there is a Google Group at

> https://groups.google.com/forum/#!forum/activity-centered-design

The website

> http://activitycentereddesign.com

is there to hold it all together.

[1] http://leanpub.com

You Are Reading The Sample Book

What you are reading right now is a sample book. It contains a few sections of the actual book. To read the whole book please go to

http://leanpub.com/activitycentereddesign

Preface

All too often simply the wrong software gets created. It may work but not as the users or stakeholders expected. Or it may solve the wrong problem altogether. Despite the vast improvements in the way how software teams create software since the Agile Manifesto was published in 2001 still many software projects fail in one way or another.

To many, the word design only describes how something looks but in fact it means how it works and looks. Well designed products are created based on a deep understanding of which activities users want to perform and are well engineered. This fitness makes these products superior to their competition and in some cases such products shake up an entire industry.

Activity-Theory traces its roots back to the thoughts of German philosophers, poets and early scientists Johann Wolfgang von Goethe (1749 - 1832), Immanuel Kant (1724 - 1804) and Georg Wilhelm Friedrich Hegel (1770 - 1831).

Russian psychologist Lev Vygotsky (1896 - 1934) laid the foundation for Activity Theory (cultural-historical psychology), inspired by the work of Karl Marx (1818 - 1883), who is considered as one of the founders of modern social science, when he was working in the field of learning and child development. Later Alexei Nikolaevich Leontiev (1903 - 1979) together with Alexander Luria (1902 - 1977) formulated Activity Theory based on Vygotsky's framework. Through the work of Yrjö Engeström Activity Theory has found its way into Human-Computer Interaction and Software Design.

This book shows how Activity Theory can be used to create smarter software. You will learn how a sucessful software development team can effectively work with users and stakeholders to deliver high quality solutions.

My own journey towards Activity Theory started when practicing Acceptance Test-Driven Development (ATDD) inspired by Gojko Adzic's book *Specification by Example*. I was coaching teams on writing good Cucumber scenarios and frequently saw how difficult it was for them. Since my early days of software development I felt that software development is more a design activity than one of engineering and when I came across the 1996 book *Bringing Design to Software* edited by Terry Winograd I learned about Activity-Centered Design in one of the chapters. It was written by Peter Denning and Pamela Dargan. They proposed *activity action-centered design* as the basis of a discipline of software architecture.

Since I started writing code sometime back in 1981 I have worked on a large number of different platforms using about 20 different programming languages. I have lived in many countries and recently moved to the U.S. where I work as a Software Development Coach. It is probably my multi-cultural background (I am fluent in German, English and Spanish) that also attracts me to something where culture and history of those using software or other tools is taken into account during the design of these tools.

Stephan Schwab, May 2012

The Being of Software

Software is hard to understand. Even for programmers, the makers of software, it is not easy to agree on a common definition of what software is. One of the problems is that software is almost everywhere and appears in so many … I was going to say shapes, but that would not fit as it's too simple. Maybe saying 'in so many forms' would do it justice but then again this is another attempt of comparing it to something known, which in itself requires that software is similar to the other thing - which it isn't.

Software is something else and can be many things. How about the following definition?

> Software is an executable model for an activity. It provides affordances to perform actions that contribute to the activity. It is a mediating tool that guides, supports, and influences user actions and perceptions.

Activities are not performed without a reason. We need a motive to do something and the motive exists because there is a need we want to satisfy. It doesn't matter whether a person has a need or an organization has a need. The same principle applies.

Affordances can be elements of the user interface. In a graphical operating environment there may be widgets on the screen to control the software. For a command line tool there may be command line switches and other input parameters. The software may require an input file with a series of commands. Anything that allows a human user or another system to interact with the software is an affordance to control it and thus perform actions.

As commonly understood software is a tool. However, it is not a simple tool, which someone uses to perform a task. A task, according to Activity Theory, may be something like drawing a circle, entering some text, applying a format to the text or, a bit more complex, sending an email.

Instead, software is a *mediating tool.* A mediating tool is being used by someone but it also influences whom is using the tool and it influences what that someone is doing.

As I write this book I'm using a regular text editor. Initially I was using a word processor. When I was using the word processor I saw a WYSIWYG (What You See Is What You Get) representation of the pages in a print layout. Now, in the regular text editor, I just see the plain text and have no idea how the text might look like when printed. Before the formatting and dealing with all the options of the word processor was distracting. Initially I felt that it was an advantage to see how the book's pages are filled with text. Later in the process I discovered that my original assumption about how I write were wrong. Instead of allowing the tool to force me into a specific method of writing I switched to another tool. Had I been unable to perform that switch the tool would have impacted my writing in a negative way.

There is more about this when we talk about software quality. First something else.

I was just saying *someone* is using software. Not only is software used by humans. It is also used by other software. Therefore, being a mediating tool, software also influences how other software is made or being used. That is easy to demonstrate in the case of software that exposes an API.

Instead of implementing the functionality of Google Maps themselves, someone making a software for realtors can use the API exposed by Google Maps to show where houses are located. The feature set offered by Google Maps and how a human operates the electronic map will then have influence on how the realtor software will be used and, if done well, how the user interface of the realtor application will be designed.

Another example is the social network Twitter. Twitter exposes an API that allows other applications to send and receive messages. The API also includes searching. The design of the API and rules set forth by the owners of Twitter influence third-party applications. Some third-party Twitter clients have been perceived as being better suited to follow conversations while the official user interface for the service did not offer the same level of functionality. That is a case where others using an API have used it a way not forseen by its creators. Software as a mediating tool has influenced its users, the community using the overall service and other software - mediating tools in itself.

Tests Are Software Too

In modern software development teams want to use automated tests to verify that their code works. Automated tests are software themselves but for a different purpose than the software these tests are verifying. The definition for software above can then be rephrased to describe a test:

> A test is an executable model of a specific action. It is a mediating tool that guides and supports the creation of software and shapes, by simulation, the executable model of an activity.

Tests are a mediating tool that is used to develop the production code. As a mediating tool the tests influence the team working on the production code and on the tests themselves.

Quality

Whenever something is made there is the question of how good it is. The users and buyers of a thing base their decision to buy and use something to a good part on quality. The makers of a thing target a certain level of quality based on their target market or the target price range. There may be other factors but I think those are the most common ones.

Quality seems to be important on both sides - to the buyer/user and to the seller/maker. But what is quality?

The dictionary offers two definitions:

- *The standard of something as measured against other things of a similar kind.*
- *The degree of excellence of something.*

Further a thesaurus tells us that the term quality can also be used to substitute the words *standard*, *excellence* and *feature*.

Measuring Against Other Things of a Similar Kind

In the case of widgets coming out of a manufacturing line we can easily compare each one to the next one in line. We can look at its size. We can look at the color and measure it to the color specified in some document so that all the widgets have the same color.

Can we do the same in the case of software?

Above I used *software is an executable model for an activity* to explain what software is. If we further create an activity system in which the subject is the user of spreadsheet software and the object is the spreadsheet software itself, then we can find *other things of a similar kind* and thus we are able to compare them. We can compare features or abilities between all the products we look at. This form of comparison makes sense being a buyer or potential user. We want to know whether the quality of software A vs software B meets our expectations.

However, when we are the maker of a spreadsheet application we cannot use this definition of quality to determine the quality of our product. Simply because our software exists as one of a kind.

Degree of Excellence of Something

Excellence means something like *extremely good*. The word originates from latin *excellere*, which means *to surpass*. So that's a first hint.

Software is an executable model for an activity is our definition for software as inspired by Activity Theory. If software is a model, then the degree of excellence is an expression of how good we have modeled the activity someone wants to perform.

I don't think it is helpful for the definition of quality to use an extrinsic point of view. We would be running behind someone else and do what someone else did. It will be hard to surpass that someone, because we already have locked ourselves into the role of a follower. Therefore I think it makes more sense to understand quality as how well we have made the executable model of an activity. Which then makes us pay close attention to the quality of our analysis and understanding of the activity system we want to implement in software.

Activity Theory

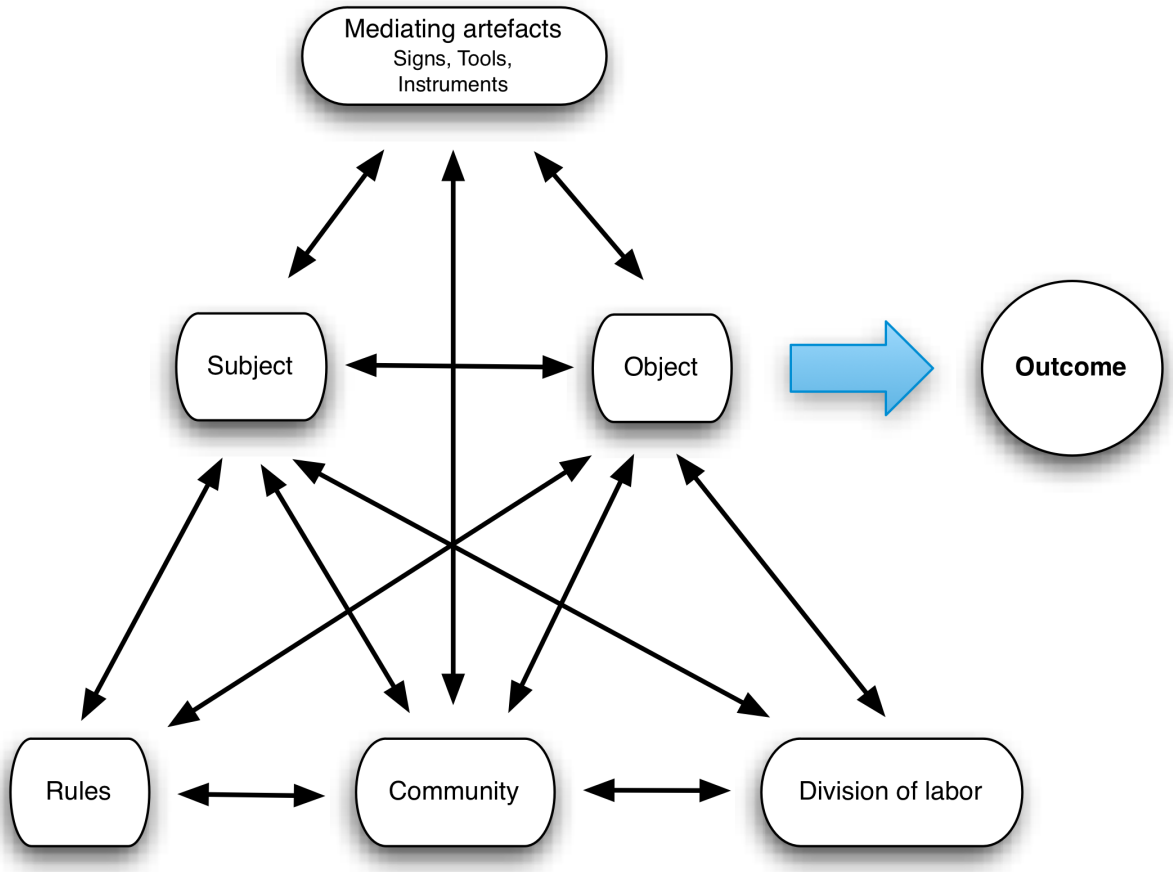
Activity Theory is based on work by psychologists Lev Vygotsky, Alexei Leont'ev, Sergei Rubinstein and others starting in the 1930s. A good introduction and historic background can be found in *Acting with Technology - Activity Theory and Interaction Design*[2] by Victor Kaptelinin and Bonnie A. Nardi.

Activity is seen as the most basic concept that describes behavior of humans, of groups of humans or any other subject. Activity is defined as a purposeful interaction of a subject with its environment. Further, activity is the source of the subject's development. A subject engaging in an activity will change as a result of the activity. Subjects are entities that exist in the world and have needs. It's the needs that make the subject engage in an activity to fulfill a need. Another interesting aspect of Activity Theory is that it takes into account the environment, the history of the person and culture as well as the role of the artifact that is being used for the activity. In Activity Theory there is a system, which is formed by the object, subject, mediating artifacts (the tools and signs used or known to the subject), rules, the community the subject belongs to and some established division of labor. The reason for performing an activity stems from the tensions and contradictions within the system.

Early models of human-computer interaction (HCI) focus on the interaction between humans (the user) and an information system as an object. There is a similarity between the subject-object relationship as described in Activity Theory. However, the early HCI models focus primarily on lower-level tasks, how the user interacts with a system, without considering the purpose of the interaction or the context in which the activity happens.

In Activity Theory the smallest unit of analysis is the activity, such as listening to music on a portable device, and not the particular human-system interaction, such as how to select a song followed by touching a play icon.

[2] MITPress, 2006, ISBN0-262-11298-1



The arrows in the diagram mean either “influences” or “perform activity on/with”. What a subject (a person) does with tools to an object is influenced by rules, which are influenced by a community the subject belongs to. Tools can be signs, which are also influenced by the community and the local culture. For example, not every pictogram works anywhere in the world due to cultural differences.

How work is organized, the division of labor, also influences how a subject performs an activity on an object to produce a certain outcome. The design of an interactive computer system will certainly have to take into account any existing division of labor - at least in the initial design - unless the change of the existing work structure is intentional.

Human Activity, Actions, And Operations

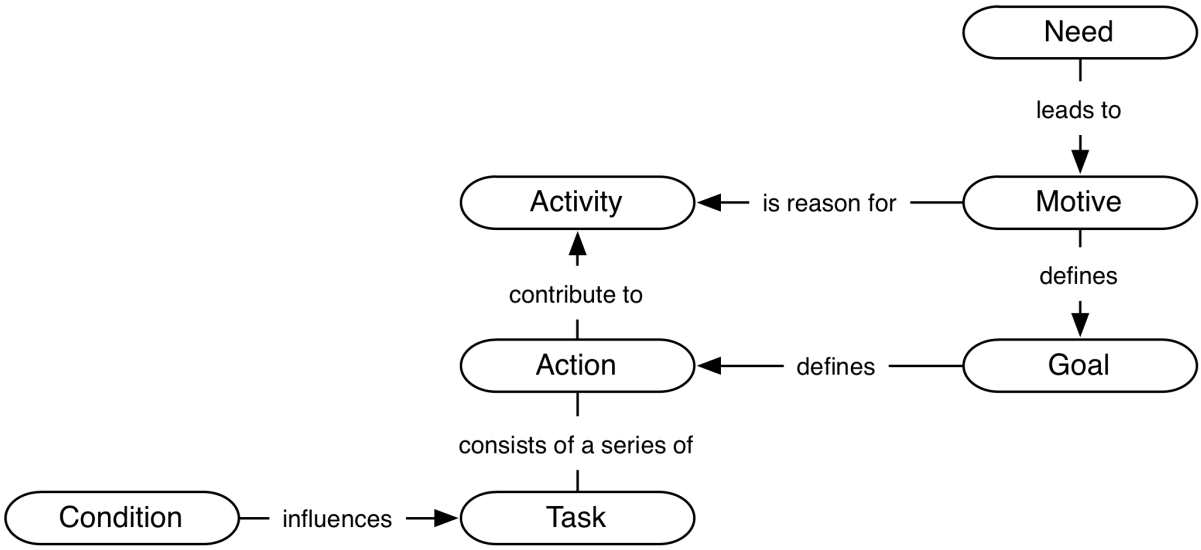
Humans perform an activity because they have a motive in doing so. The motive stems from a need that people want to fulfill. So they start an activity directed towards one or more goals. The goals

define the actions of individuals. Conditions shape tasks, which these individuals execute as part of the larger goal-oriented actions.

A writer may be driven by a desire to get a certain idea in front of other people. The wish of getting his idea out in public is the need that turns into his motive for performing the activity of writing. Writing is the overall activity. It is not really what the writer does exclusively and all the time.

He will conduct research, which is an action directed towards the goal of acquiring more knowledge about something related to the idea he wants to publish. He will write down what he discovers during his research. Writing things down is an action with the goal of not loosing important information.

While conducting research the writer will probably use his computer to access files on his hard-drive or somewhere on the Internet. To write things down he may use a notepad application on his computer or use another device such as old fashioned pen and paper. These lower level tasks are influenced by conditions, such as where the material is located (on the local hard-drive or on the Internet) and what type of writing tool (notepad application or pen and paper) is available. These operations also depend on personal preferences, which are another form of condition.



As software developers we may risk creating a beautiful technical solution to the completely wrong problem, if we don't understand what people actually want to do and why. We may focus way too much on lower-level operations - how to use our technical solution - and forget about the goals of our customer in the context of his motives.

Activity Theory provides the theoretical framework for finding out what our customer/user wants to do and why so that we can propose the right goal-oriented actions and offer the right operations within the conditions our customer/user lives/works in.

Learning And Development

The Activity Theory triangle uses arrows that point in both directions. A subject performs an action to an object but then the action performed on the object influences the subject. Similarly a subject is influenced by a community but also the community is influenced by the subject.

Whenever humans do something they also learn and develop.

Do you remember when you learned how to drive a car? If you are like me, then you learned to drive with a manual transmission that requires you to shift gears by applying the clutch and then use the shift stick to change into another gear. Initially to learn how to drive a car has been a goal, which then defined the actions of finding a driving school, taking lessons, learn to shift gears properly and many other things.

After learning the basic mechanics and getting more and more practice it becomes second nature and one done think anymore about it. Shifting gears while driving becomes a meaningless operation that one performs almost unconsciously.

However, remember when you first had to drive another car? Suddenly you were back to shifting gears consciously. The other car probably had a different transmission, was newer or older than the one where you learned it and because of any number of differences (conditions) you had to focus again on the task of shifting gears.

With that example I want to explain that something can be a goal driven action or it can be a task influenced by conditions or it may completely fall off our radar while we still do it. The reason for that is learning and the fact that we develop while we perform actions and tasks.

In the case of software this is something we should take into account. We need to understand for whom we make software and at some point our software - or the software driven tool we are making - needs to adapt to our learning and developing users.

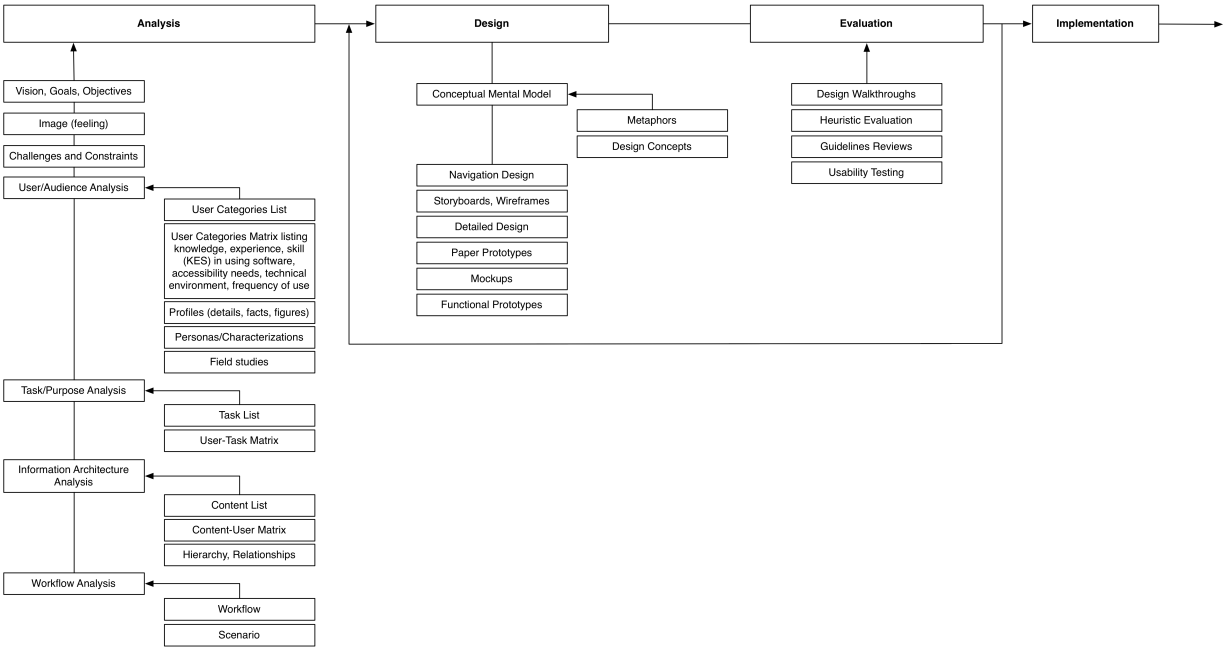
Putting It Together

Traditionally the user experience was designed before programming would start. The assumption was that the user experience designer should complete the design process before the expensive task of programming the system for real would start. To an extend that **made** sense but today we have so much more powerful tools at our disposal that we loose more than we gain by following a step by step approach. Modern development environments, languages and other tools allow us to create a working prototype literally within minutes. In the past we would have relied on paper prototypes, walked users through wireframes and other forms of low-fidelity simulations. Don't get me wrong. Those low-fidelity prototypes still make sense but we don't have to delay programming the real software anymore.

As I was saying in the beginning building software, as in writing the code and getting it to execute, is cheap. Therefore we can design the user experience while we go. Now I am sure that some user experience designers will heavily contradict me. Allow me to elaborate a little further. For that let's review the traditional User-Centered Design (UCD) process.

Review Of User-Centered Design

The following diagram depicts a typical process for the development of a web application[3].



User-Centered Design Process

[3] http://www.w3.org/WAI/EO/2003/ucd

User-Centered Design is based on findings and thoughts from participatory design (around 1970), contextual design (1998), ergonomics (1943), cognitive science (1967) and interaction design (1990).

Traditionally user-centered design methodologies use some form of analysis phase first, followed by design, evaluation, and implementation phases and finally the product gets delivered during the deployment phase. Although there is a lot of usability testing and hence a lot of feedback opportunities there are a number of problems. The biggest problem is that all that testing is not being done using the actual software but something like wireframes, mockups or a paper prototype. This is bad because the behavior of the actual software is missing. With those substitutes all you have is how it looks like and the behavior only appears in the words of the person showing you e.g. the wireframes or simulating the software using the paper prototype.

UCD, if practiced in this fashion, is heavy on documentation. The main problem I see with all those documents is that their purpose is to tell someone else what to make. Making software in itself is a design activity and the same people designing it are also making it. There is no place for blueprints as instructions for someone else.

On the other hand, there is a value in something like a style guide for the user interface of the application. Defining personas and doing field studies to find out what people want has even more value. That is an area where UCD meets Activity Theory and becomes Activity-Centered Design because we look at the activities those personas want to perform using the software.

Focus On Activities And Working Software

Form is supposed to follow function. That means that the look for something is secondary to what it does. Of course we do want a nice looking product in the end but while we are still figuring out what it should do and how it will do what it should, we should not be worried about its look. Nice looking things can be extremely difficult to use. Nice looking software can be a pain to get something done. So if we cast aside all consideration for the look of the software, then we are back to pure functionality and hence to what the software does. We are now looking exclusively at the activities.

Previously we took a concept map of the business activity of signing up for a conference to develop some user stories. Then we created a Cucumber scenario based on one of the stories and implemented a piece of working software that passes the acceptance test in the form of the Cucumber scenario. We did not bother with how the web application is going to look like. We were interested in what the application does.

In the case of a web application we code the user interface in HTML, CSS and probably some JavaScript too. These technologies work in such a way that we don't have to use them all at once unless we want to. There is nothing wrong in creating a bare bones HTML user interface without any CSS styles. We can add that later to make it look pretty. We don't need user-friendly date pickers written in JavaScript just to be able to enter a date in what is technically a text field anyways.

In the case of a desktop application we don't have to come up with custom controls. All the user

interface toolkits I have seen so far in the last 20 years of software development do allow the programmer to create a decent user interface. The buttons, drop down lists and other things may not meet everybody's taste standards but they are part of the general look & feel of the platform and they are functional and allow a human to operate the software system. That is all that counts while we figure out what the software should do.

Once we are sure that the software does what our stakeholders want and need we can gradually improve the look & feel of it. In the case of a web application we dress it up by adding CSS styles to it. Desktop applications created with a UI toolkit most of the times already look ok but still we may want to rearrange things a bit. Without changing the functionality we may try to reorder and rearrange the elements on the screen to provide a more pleasant user experience. There may be little things like, for example, two related elements are too far from each other and therefore make the user go a longer way than needed. That kind of thing we need to fix and it is not difficult to do.

The Activity-Centered Design Loop

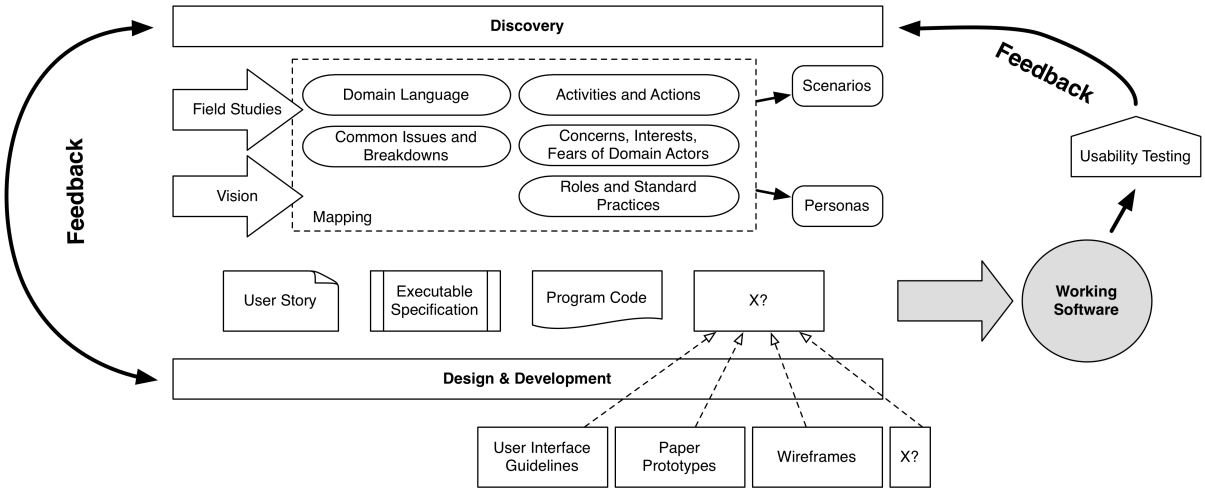
Activity-Centered Design is about making software that allows people to perform an activity. Allow me to repeat my earlier definition of what software is:

> Software is an executable model for an activity. It provides affordances to perform actions that contribute to the activity. It is a mediating tool that guides, supports, and influences user actions and perceptions.

So we want to

- focus on the activity
- allow a human to use the software to perfom an activity
- develop the software based on feedback from its usage

In Activity-Centered Design we bounce back and forth between discovery and design & development to produce an outcome from which we gather feedback that again feeds into a new round of discovery.



Activity-Centered Design

We start with some discovery activities. One good way to learn about the problem we want to solve is to perform some field studies. We can go out and see what people are doing. Or we can start with a vision for a new product. In any case we have a reason and an initial direction for the discovery process.

As a team we map out as much as we can about the language used in the domain, common issues and breakdowns that people experience and their concerns, interests, fears. We identify roles and standard procedures and develop activity systems to learn about activities and actions. Based on all that we end up with some initial scenarios (what people want to use the software for) and, of course, a number of personas based on the roles we discovered.

Our new wealth of knowledge allows us to come up with good user stories, which help us to schedule our development activities. We should use executable specifications to test-drive our development. Depending on the type of software we may want to employ a number of additional techniques or be guided by something like an existing user interface style guide.

And of course we do want to perform a lot of usability testing as frequently and early as possible. If ever possible, we should use real working software for that. In some cases we may use wireframes or a paper prototype for the same purpose. However, the preference is always on real working software.

Isn't that a process or a methodology?

No, I don't think so. The diagram for Activity-Centered Design does not prescribe any order of activities, as you would find in diagrams for processes. Work oscillates between the activities of discovery and design & development. We do a little bit of everything, when needed, to produce working software that, by being put in front of human users

who want to perform an activity, provides us the ultimate feedback. We go back and forth between those activities each time we have learned something on either side.
