

Simplifying for Usability

[Print version \(PDF\)](#)

Introduction

1. Simplification

- [Simplification: What Does it Mean, What are its Benefits?](#)

2. The Users

- [Who are Your Users, What are Their Needs?](#)
- [Some Myths about Users](#)
- [What Does Cognitive Psychology Tell Us about Users?](#)

3. Design Principles

- [Global Goals for Simplification](#)
- [Design Principles for Simplification](#)
- [Simplicity](#)
 - [Reduction](#)
 - [Organization, Structure](#)
 - [Integration](#)
 - [Prioritizing](#)
- [Transparency](#)
 - [Understandability](#)
 - [Learnability](#)
 - [Self-Descriptiveness](#)
 - [Consistency, Conformity to User Expectations](#)
 - [Stability, Continuity](#)
 - [Feedback](#)
 - [Metaphors](#)
 - [Efficient \(Re\)presentation](#)
 - [Aesthetics](#)
- [Effectiveness and Efficiency](#)
 - [Goal Orientation](#)
 - [Suitability for the Task](#)
 - [Error Robustness](#)
 - [Distribution of Tasks \(Division of Labor\)](#)
 - [Balance of Controllability vs. Guidance – Amodality vs. Modality](#)
 - [Parallelism](#)

4. Tips and Tricks

- [What Is Really Necessary?](#)
- [Who Does the Work?](#)
- [Who Is in Control?](#)
- [Understanding, Learning and Relearning](#)

 [top](#)

What's in this Guide?

Why a Guide About Simplification?

Today, with the ever growing spread of computers into our daily lives, software application design is more and more focusing on untrained and casual users. These users have very different requirements from professional users who for a long time dominated the business scene. Developers need to take this new breed of users seriously and design applications that do not frustrate them.

"Simplifying for Usability" sets out to help software developers in this effort by proposing a number of "simplification principles" that can be used as guidelines for achieving simple and easy-to-use applications. These guidelines are not specific to a certain technology and can be applied to any user interface design.

Outline of this Guide

This guide starts by explaining what simplification means and what its benefits are. We then take a look at the users, their goals, and requirements. Here, we set out to "kill" some myths about users that have been deliberately told by the software industry. We also list some basic principles from cognitive psychology, which help us to understand how humans process information. This section is supplemented by some practical applications of these principles.

The main section of this guide presents general design principles that help you to develop simple and efficient applications. For each principle, we explain what it actually means for the users and how you can translate these principles into actions within your applications. In order to structure this discussion, we group these principles into the three fundamental categories: simplicity, transparency, and effectiveness.

The last section contains a number of "tips and tricks." Here, we present examples that illustrate how the simplification principles connect to actual design problems.

Status

Version 1.0.1, October 2004: Minor revisions and corrections

This guideline can be found in *Resources* on the SAP Design Guild Website (www.sapdesignguild.org).

Source: [Simplifying for Usability](#)

What's in this Guide?

Why a Guide About Simplification?

Today, with the ever growing spread of computers into our daily lives, software application design is more and more focusing on untrained and casual users. These users have very different requirements from professional users who for a long time dominated the business scene. Developers need to take this new breed of users seriously and design applications that do not frustrate them.

"Simplifying for Usability" sets out to help software developers in this effort by proposing a number of "simplification principles" that can be used as guidelines for achieving simple and easy-to-use applications. These guidelines are not specific to a certain technology and can be applied to any user interface design.

Outline of this Guide

This guide starts by explaining what simplification means and what its benefits are. We then take a look at the users, their goals, and requirements. Here, we set out to "kill" some myths about users that have been deliberately told by the software industry. We also list some basic principles from cognitive psychology, which help us to understand how humans process information. This section is supplemented by some practical applications of these principles.

The main section of this guide presents general design principles that help you to develop simple and efficient applications. For each principle, we explain what it actually means for the users and how you can translate these principles into actions within your applications. In order to structure this discussion, we group these principles into the three fundamental categories: simplicity, transparency, and effectiveness.

The last section contains a number of "tips and tricks." Here, we present examples that illustrate how the simplification principles connect to actual design problems.

Status

Version 1.0.1, October 2004: Minor revisions and corrections

This guideline can be found in *Resources* on the SAP Design Guild Website (www.sapdesignguild.org).

Source: [Simplifying for Usability](#)

Simplification: What Does It Mean? What Are Its Benefits?

[What Does Simplification Mean? What Are its Benefits?](#) | [When Does Simplification not Suffice?](#) | [How Does Simplification Work?](#) | [How Can You Achieve Simplicity?](#)

What Does Simplification Mean? What Are Its Benefits?

Today's software usually boasts **functionality**. This functionality is, however, often **inaccessible** to users because the software is:

- **Not transparent:** Users do not find the many "hidden features" of an application
- **Complex:** Users are disoriented and do not know how to proceed and how to achieve their goals
- **Cumbersome to use:** Users are hindered and frustrated by inefficient procedures

Such software not only frustrates users, but also costs money because it is inefficient. Users must put a lot of effort into mastering their software instead of performing their tasks.

Simplification is one possible solution to this problem. Put simply, "simplifying for usability" means **developing simple applications that are easy to understand, use, and learn**. This can be done from scratch, as is preferred, or by transforming an existing complex application, or set of applications, into one or more simple applications. The latter is much harder and often leads developers into the trap of trying to squeeze all the existing functionality into the "easy" applications, which usually turn out to be as complex as their predecessors.

Simplification is not a goal in itself. By making applications simpler, you make them more **effective, efficient**, and, hopefully, **more fun to use**. Simplification reduces the barrier that computer software creates between users and their tasks. It **increases productivity** and **user satisfaction** and thus **reduces costs**.

So, possible results of successfully simplifying an application are:

- The software is **accessible** to a wider range of users (e.g. untrained users); tasks, which were formerly performed by specialists, can be delegated to employees or customers (e.g. self-service applications).
- Users are more **efficient** and **satisfied**.
- Increased efficiency in the use of software results in **higher productivity** and thus **reduces costs**. Furthermore, training costs can also be reduced.

When Does Simplification Not Suffice?

Simplification is not the only possible answer to the above-mentioned dilemma of "feature growth" and software complexity. There are cases where the best solution is to provide different types of users with different versions of an application. There may be a place for an "easy application," but professional users may still need a powerful and flexible application that is hard to use for untrained or casual users.

In other cases, the procedures are inherently complicated and complex. Really complex things cannot be made simple enough that everybody can do them without prior training. There are limits for simplification, if one does not want to sacrifice power or oversimplify certain processes. Thus, often a careful analysis is needed to find out, which elements of a process can be simplified and performed by untrained users and which must still be reserved for experts. Such an analysis can lead to a totally new application design and

restructuring of the business processes. So, the answer may be "simplification – yes" but only for certain parts of a process.

How Does Simplification Work?

Simplification may work in several ways. Here are some examples:

- Reducing functionality to the essential functions makes it easier for users to find the required functions
- Reducing the number of steps in a procedure lets users complete their tasks faster
- Making software transparent to users makes it easier to learn for beginners and easier to use for casual users
- Transferring parts of a task to the system reduces the users' work load and lets them do their tasks faster and with less errors
- Guiding users through procedures leads to less failures and frustration

How Can You Achieve Simplicity?

Do you know how to achieve simplicity? And do you know how to achieve it without sacrificing other goals? You will find the answers to these questions in this guide! Here, we propose and illustrate general design principles that set the focus on developing simple and efficient applications. We also cover some of the problems that may arise when their requirements conflict with each other.

Please note that many of the principles in this guide are also applicable for professional users. These users can, however, be confronted with greater complexity, more functionality, and less user guidance: They know the procedures, how software works in general, and how to get out of deadlocks and errors.



[top](#)

Source: [Simplifying for Usability](#)

Who Are Your Users? What Are Their Needs?

Users of business software use computers for a purpose. They want to get their job done with the software - no more and no less. They also want to get their job done efficiently. They are usually in a hurry and have a pile of work on their desktop. They do not want additional work load caused by cumbersome, inefficient or complicated applications. Equally, they do not want to look stupid, as Alan Cooper points out - neither in their own, nor in their colleagues' eyes.

So, let's summarize:

Users of business software

- are goal oriented
- want to get their job done
- want to get it done efficiently (in time and resources)
- and are, therefore, impatient

Users of business software do not want to

- be concerned with the software instead of doing their tasks
- take more steps than is necessary
- learn more than is absolutely necessary for their task
- relearn when there are well-trained procedures
- look stupid

Note: If you develop for a special audience, such as Web users, your users may be different to usual users of business software.



[top](#)

Source: [Simplifying for Usability](#)

Some Myths about Users

Software vendors and marketing departments have told many myths about users. These myths, in part, have led to today's graphical interfaces and GUI-based applications. But despite all the marketing hype, these interfaces and applications bear a lot of problems for users. This guide addresses some of these problems.

As an example of such myths, we present some of Apple Computer's statements from its Human Interface Guidelines. You will find similar statements in the guidelines from other companies. Note that many of these myths about users originated in universities and scientific environments. These institutions are quite different from companies, factories or other places where business software is used. In a business setting, people work with software because it is their job, not because they are fascinated by computers.

Apple's Myths

- People are instinctively curious. They want to learn, and they learn best by active self-directed exploration of their environment.

Our opinion: Users do not want to learn - at least not more than is absolutely necessary for carrying out their tasks. The best method for gaining this knowledge is to ask colleagues. Users do not want to explore their environment. They want to get their job done efficiently.

- People strive to master their environment. They like to have a sense of control over what they are doing, to see and understand the results of their own actions.

Our opinion: Of course, people want to master their software. But they do not like software which is a challenge to master. They simply want to get their job done. Of course, people do not want to look stupid - as Alan Cooper often emphasizes. So they often hesitate, or are reluctant to use a computer or piece of software. Most people can do without the challenges of complex software that is hard to use!

The second statement, however, is quite true. Users want to know what is going on in the software, what to do next, why the software reacts in the way it does, and so on.

- People are also skilled at manipulating symbolic representations. They love to communicate in verbal, visual, and gestural languages.

Our opinion: Obviously it is not the computer that people in a working environment like to communicate with, but their colleagues.

- Finally, people are both imaginative and artistic when provided with a comfortable context. They are most productive and effective when the environment in which they work and play is enjoyable and challenging.

Our opinion: That's why many people take courses at evening schools. Very few users of business software try to be artistic or imaginative (only when some of them try to figure out how to use a software - most of them will give up or ask for help). In addition, their job is often challenging enough. They do not need challenging software.

Source: [Simplifying for Usability](#)

What Does Cognitive Psychology Tell Us about Users?

Information Processing Principles | Applying the Principles

Information Processing Principles

Cognitive psychology derived a number of principles regarding how humans process information. These principles, though quite general, give good advice for making software easier to use.

- As a rule of thumb, the **working memory span** is about 7 ± 2 items.
- Human **processing capacity is limited** and has to be divided between tasks.
- High frequency repetition of tasks can lead to "**automatic**" **execution**, thus increasing the capacity for other tasks. However, automation requires hundreds to thousands of learning trials (performance follows a power function in time).
- **Recognition** is easier than **remembering (recall)**.
- **Elaboration** of information leads to better memory.
- Good **organization** of information (chunking, clustering, categorizing) leads to better remembering.
- Keeping the **context** for learning and recall (e.g. software use) constant leads to better remembering.
- **Pictures plus text** are remembered best. Pictures alone are remembered better than text alone.
- **Pictures** are extremely well recognized.
- Reaction time increases (logarithmically) in relation to the **number of alternatives**.
- Users create **mental models** of their software when they are working with it. These models are simplistic (incomplete), informal, often concrete, and may contain errors. Mental models help users to envision, predict, and explain system actions. They aid users in planning their actions.
- **Metaphors** may help users to transfer their prior knowledge to a new software.

Applying the Principles

In the following, we present some applications of the principles listed above. As you will see, these general principles are very useful heuristics.

Principle	Application
Working memory span is 7 ± 2	Do not ask users to remember more than 5 to 7 items. Example: Do not ask users to remember long lists of items or commands. Extension: Do not ask users to remember anything that the system already knows.
Limited human processing capacity	If users have to struggle with their software, there is no capacity left for their task. Example: Do not include animated GIF images in a Web page, as these images distract users. Example: Do not require users to remember anything while engaged in another challenging task.
Automation of tasks	You can expect "automatic" execution only for tasks that are repeated over and over. Example: Do not expect casual users to become proficient with your software. Instead, make it simple to use!

Recognition vs. remembering	<p>Provide choices in the form of menus or lists instead of requiring the users to key in a choice.</p> <p>Example: Provide help on input values (F4 help in R/3).</p> <p>Extension: Functions on pushbuttons in the R/3 application toolbar are easier to find than functions that are located somewhere in the menu tree only, because users need not remember their position in the menu tree.</p>
Elaboration of information	<p>Superficial engagement in a software will not lead to an understanding of its working. Therefore motivate users to use your software!</p> <p>Extension: Do not frustrate users. Frustrated users will no longer be engaged in your application and its requirements.</p>
Organization of information	<p>Try to organize information, functions, and screen elements in sensible units.</p> <p>Example: Order lists alphabetically, by date, or any other scheme that makes sense in the application context.</p> <p>Example: Cluster fields into sensible groups.</p>
Constant context for learning and recall	<p>Create a stable working environment for your users.</p> <p>Example: Reserve a permanent area for navigation buttons.</p>
Remembering of text and pictures	<p>Try to use pictures, or pictures with text, where possible.</p> <p>Example: Present regions of a country as a map.</p>
Recognizing pictures	<p>Familiar icons may provide clues to the functioning of your software, even for casual users.</p> <p>Example: Use familiar navigation buttons.</p>
Number of alternatives	<p>Try to limit the number of alternatives for menu choices, function selections, and so on.</p> <p>Example: Instead of letting users scan the menu tree, provide pushbuttons with the relevant functions.</p>
Mental models	<p>Help users to create simple, effective mental models of your application. Provide clues that lead to such models in an intuitive way.</p> <p>Example: Provide clues about the structure of your application or about the sequence of steps in a procedure.</p> <p>Example: Provide clues about how users should navigate in your application.</p>
Metaphors	<p>If the task at hand is suited, use real world metaphors to organize your software.</p> <p>Example: You can use a room metaphor for your Web application: Each room corresponds to a certain type of processing. Organize the rooms and the transitions between them in a way that makes sense to users.</p> <p>Example: You can use a book metaphor for a Website that provides information.</p>

Source: [Simplifying for Usability](#)

Global Goals for Simplification

What Does "Simple" Mean? | Simplicity | Transparency | Effectiveness and Efficiency

What Does "Simple" Mean?

What simplicity is, may seem quite obvious to you. However, if you take a closer look, it turns out that simplicity has many facets. Some overlap but others may even contradict each other. Thus, the inconspicuous word "simple" can have diverse meanings for different people and under different perspectives. The picture below, might be a possible result of a brainstorming session on the theme of: "What does simple mean?":

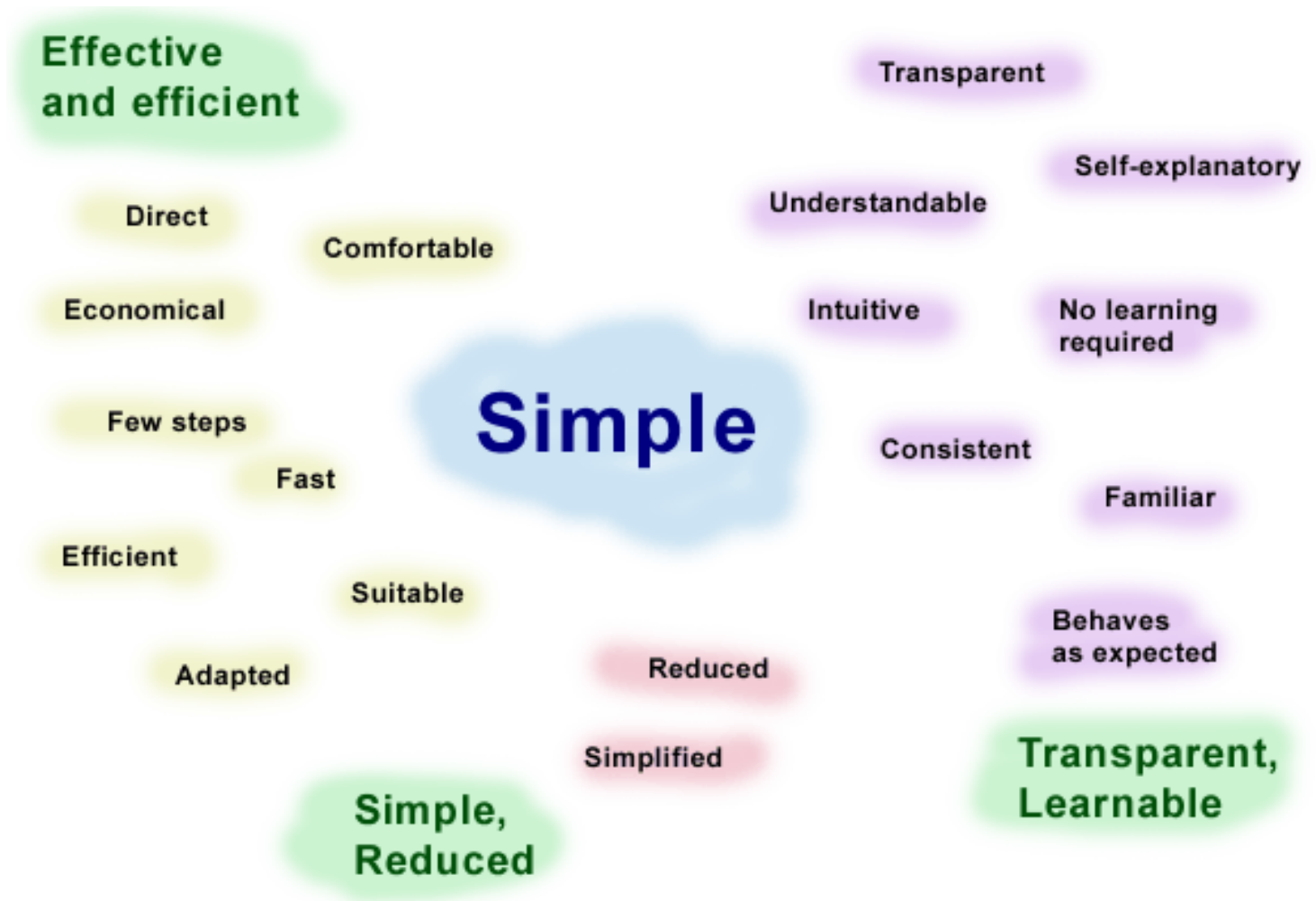


Figure: The many meanings of "simple"

While at first sight there seems to be no structure in the many aspects of "simple," a closer reveals the following three basic aspects of simplicity:

- [Simplicity](#)
- [Transparency](#)
- [Effectiveness and Efficiency](#)

We will use these "global goals for simplification" as **organizers** for further "secondary" principles that elaborate the basic aspects. Before doing so, let's take a short look at the three global goals!

Simplicity

Simplicity means that things are simple. However, making things simple is not an easy task in today's complex business environments. Simplification is a continuous struggle between task requirements and user requirements. From the users' point of view, an application should be simple in the sense that it does not build up a barrier between them and their tasks. It should be more or less "invisible" to the users and enable them to focus on their tasks.

Why Simplicity?

There are many answers to this question. We will mainly provide a psychological argument. Your users are humans, not computers or machines. People have certain abilities but also a number of limitations with respect to their information processing capabilities. It is important to know and respect these limitations – otherwise, you overstrain and frustrate users. (You find some of their characteristics in [What does cognitive psychology tell us about users?](#))

With respect to the users' personal goals, as well as to company goals, many of the arguments in favor of simplicity relate to efficiency. We will cover those arguments there.

Why Not Simplicity Alone?

Simplicity alone is not a sensible goal for application design. Making an application too simple may mean that it does not achieve its goals and, in the worst case, may be useless. You have to balance simplicity with other goals, such as the purpose of a program, which is reflected in its functionality. Making an application too complex, on the other hand, may render it unusable. Thus, application design is the art of finding the right balance between the opposing threats of an application being useless or unusable!

Transparency

Users should always understand what is going on in an application with respect to their tasks and the required steps to take – not with respect to the inner working of the application. This is the essence of transparency! For example, users should always know the state of their task, what to do next, where to go next, what the consequences of certain user actions are, and so on.

Transparency has several aspects. It relates to the overall structure of an application, as well as to the structure of the navigation, procedures and functionality in general.

Why Transparency?

Transparency helps users

- To maintain orientation, which is an important and essential human need
- To focus on the task because they are not distracted by other tasks, such as deciphering how the application works

It also reduces the users' working memory load because they need to remember less information about the application, its state, and its working principles.

Effectiveness and Efficiency

Effectiveness and efficiency are the primary goals for any software in a business environment. Effectiveness means that your application does what you designed it for and that users can accomplish their goals with your application. Efficiency requires that performance is fast and error-free. Both goals have also to be viewed with respect to the prospective users of your application. Users may vary in their skills and abilities and thus may require different user interface designs.

Why Effectiveness and Efficiency?

Customers buy your application in order to computerize their business processes and to make them more efficient. If their employees cannot use your application, this is wasted money for the customer!

More often, users somehow manage to master an application, but they work inefficiently. Inefficient software also costs money:

- Users take longer to accomplish their tasks.
- Users are frustrated, because the application is seen as an obstacle to accomplishing their tasks. This in turn leads to a vicious circle with respect to costs, efficiency, and user satisfaction.



[top](#)

Source: [Simplifying for Usability](#)

Design Principles for Simplification

[Overview](#) | [Simplicity](#) | [Transparency](#) | [Effectiveness and Efficiency](#)

In the following, we elaborate the global simplification goals by proposing a number of related "secondary" design principles, which cover the many aspects of simplicity. These principles allow you to focus on certain simplicity aspects and also maintain an overview of the requirements, which are often in conflict with each other.

Note that some of these principles are direct requirements of the ISO norm 9241 part 10 for software usability.

If you are in a hurry, you can browse through the short descriptions of the principles presented on this page, and then jump directly to section 4 "[Tips and Tricks](#)." There you will find a selection of design questions and practical examples regarding simplicity.

Overview

Below, we briefly present each principle and assign it to a single global aspect, although the principles often serve more than one goal. In the following sections, we will explain how each of these principles fulfills the goal of simplification and demonstrate how it can be used in the design of applications that are easy and efficient to use.

Simplicity

Reduction

Reduction means to reduce an application to its essentials. This principle is applicable to all aspects of application design:

- Reduce the functionality (goals)
- Reduce the structural and navigational complexity
- Reduce the interface (screen) complexity

Organization, Structure

Organizing and **structuring** an application relates to:

- The general application structure (screens, pages, and so on)
- The navigational structure
- The structure of the functionality
- The screen or page layout

Integration

The principle of **integration** stresses the importance of integrating simple, elementary tasks into a coherent framework.

Prioritizing

Prioritizing means that applications should focus on the essential tasks and not try to serve a multitude of, possibly diverse, goals. This includes optimization with respect to the important aspects of a task.

 [top](#)

Transparency

Understandability

Software should be comprehensible for users – not only its creators. There are many aspects to **understandability**: application structure, navigation, procedures, and terminology. Or more concrete: This principle requires that users always know the state of their task, what to do next, how the application reacts to certain inputs, and so on.

Learnability (ISO)

An application should be **easy to learn**. It should also be easy to relearn, so that casual users can master it.

Self-descriptiveness (ISO)

Self-descriptiveness requires that an application can explain itself to the user.

Continuity, Stability

An application should provide a **stable** and **familiar** working **environment** for users.

Consistency, Conformity to User Expectations (ISO)

An application should be **consistent** and meet the users' expectations with respect to work practice as well as user interface standards.

Feedback

Feedback provides users with clues about the effectiveness of their actions and what is going on in an application.

Metaphors

Metaphors, usually taken from the real world, help users to transfer their prior knowledge to new software applications. This facilitates the use of applications, especially for novices and casual users. Metaphors can be also an effective integration tool.

Efficient (Re)presentation

Using efficient **representations**, for instance a graphical representation instead of a textual one, can make an application much easier to understand.

Aesthetic

Aesthetic appeal is an important factor, especially on the Web. An aesthetically pleasing design can contribute to the overall simplicity and acceptance of an application.

 [Top](#)

Effectiveness and Efficiency

Goal Orientation (ISO)

An application has to fulfill certain **goals**. There are goals dictated by the task at hand, and there are user goals, which may not coincide with the first ones. In the context of simplicity it is important to minimize the number of goals for an application – preferably to one.

Suitability for the Task (ISO)

An application is used for performing a certain task. With respect to simplification it should not only enable users to accomplish the task, but should also let them do so in an easy and intuitive way. That is, the application should "fit the task."

Error Robustness (ISO)

An application should be **permissive to errors**, or, preferably, should prevent them.

Distribution of Tasks (Division of Labor)

An application accomplishes its goals through the interplay of user and system actions. You as the developer determine how much work load is put on the user and how much work is taken over by the system.

Balance of Controllability vs. Guidance - Amodality vs. Modality (ISO)

Controllability is often translated as "the user is in charge, not the computer." Users should at all stages of the processing be able to decide what to do next. With respect to simplification, you have to balance controllability with guidance to achieve an optimal fit for the respective user groups.

Amodality and modality are two important aspects: Amodality means that the application does not force users into predetermined actions. Modality, on the other hand, forces users to perform a certain action.

Parallelism

Parallelism has several aspects:

- Parallel processing of tasks
- Parallel provision of functionality
- Parallel provision of information

Typically, it makes applications more efficient, but it also increases complexity.

 [Top](#)

Source: [Simplifying for Usability](#)

Reduction

[Why Reduction?](#) | [How You Can Apply Reduction](#) | [Reducing Goal Complexity](#) | [Reducing Structural and Navigational Complexity](#) | [Reducing Functional Complexity](#) | [Reducing Procedural Complexity – Counting Steps](#) | [Reducing Interface Complexity](#) | [Reducing Terminological Complexity](#)

Reduction is the principle closest to simplification. It means that you work out what is really essential for your software and what is not and can be left out.

See also [What is Really Necessary?](#) for more information and examples on this topic.

Why Reduction?

Reducing the complexity of an application reduces the users' work load, their working memory load, and leads to faster performance with fewer errors.

How You Can Apply Reduction

Reduction is applicable to all aspects of application design:

- Reducing the **goals** (coarse-grain functionality)
- Reducing the **structural complexity**
- Reducing the **navigational complexity**
- Reducing the **functionality** (fine-grain functionality)
- Reducing the **procedural complexity** (number of steps)
- Reducing the **interface (screen) complexity**
- Reducing the **terminological complexity**

These aspects are not independent of each other. For instance, reducing the structural complexity should usually also reduce the navigational complexity.

Reducing Goal Complexity

You can apply the principle of reduction at all stages of the development. However, the key stage for reduction is when you determine the design goals for your application. Methods, such as task analysis, help you to determine the essential goals for your application and to avoid future complexity.

Reducing Structural and Navigational Complexity

After the basic goals of an application have been determined, you translate your abstract application model into a prototype with actual interface elements. This process may, for example, proceed via use case models

that are procedures to use context models that contain screens and generalized interface elements to a prototype with real-world interface elements.

In this transition, it is useful to sketch a model of the application and navigation structure in order to keep track of the complexity of these structures. Graphical methods, may it be paper and pencil or graphic tools, assists you in this effort. Do not trust your own mental model. With an application growing and growing, this model will become more and more incomplete and complex. Do not expect users to understand such a model.

Reducing Functional Complexity

Functional complexity results from feature overload, bad structuring of functionality, or unclear goals for an application. Functional complexity can be reduced by focusing on the essential goals of an application and by leaving unimportant features out.

Web, Dialog Windows, Wizards: In-place Functionality

Sometimes, functional complexity has to be reduced for technical reasons. For example, on a Web page, in a dialog window, or in a wizard window, there are no menus. Each function has to be provided through pushbuttons on the screen (in-place functionality). This restriction automatically enforces a reduction to the essential functions. Do not try to "smuggle in" functional complexity through the back door.

Reducing Procedural Complexity – Counting Steps

Reduction is also important at a microscopic level, that is, with regard the actual steps users must take. Count the steps in a procedure, compare alternative solutions if there are too many steps, and choose the fastest solution (this may not be the one with the least steps). Each delay caused by an additional dialog window, confirmation button, etc. is a nuisance to the users!

Note, however, that there are exceptions to this rule. Users often find it easier to use a small set of commands and use simple commands repeatedly. They prefer this to learning additional commands that accomplish more complex and powerful steps. Usually, this phenomenon relates to the frequency of command usage.

Reducing Interface Complexity

Interface complexity often results from feature overload. For example, users are required to fill in fields that are never used in their work context. So, the above-mentioned reductions also help to reduce interface complexity.

Another source of interface complexity results from bad screen design. Fields may be misaligned, not arranged in the order of the work process, scattered over the screen, and so on. We do not want to go into detail here but would like to remind you of the importance of screen design. This issue is much more important on the Web. An HTML page has no grid and provides more options for arranging elements on a screen – and also more ways to do it wrong.

Reducing Terminological Complexity

Although many developers do not pay much attention to terminology, the terms that appear on a screen play an important role as to whether users will be able to carry out their tasks or not. Possible problems are:

- The same object or function is named differently in different parts of your application
- Different objects or functions use the same term
- Your application uses terms that are not familiar to the user and / or the work context

Some developers like to invent their own terminology. However, usability guidelines (ISO 9241) require that an application speaks the user's language, not the developer's.



[top](#)

Source: [Simplifying for Usability](#)

Organization, Structure

[Why Organization/Structure?](#) | [How You Can Apply Organization / Structure](#) | [Designing the Overall Application Structure](#) | [Designing the Navigational Structure](#) | [Designing the Structure of the Functionality](#) | [Designing the Screen or Page Layout](#)

Organizing and structuring an application is similar to being the architect of a building.

Why Organization/Structure?

Human memory is a huge knowledge structure that is organized according to principles that we are not aware of. But we do know that human performance depends on an efficient organization of facts and procedures.

This principle also applies to software applications. Users' performance is better if an application's overall structure, navigation, functionality, and screens are well organized. An efficient organization also simplifies an application, which has an additional positive impact on performance.

How You Can Apply Organization/Structure

The task of organizing and structuring an application relates to aspects such as:

- Overall application structure (distribution of tasks onto screens, pages, etc.)
- Navigational structure (navigation between tasks/screens)
- Structure of the functionality ("Packaging" of related functionality)
- Screen or page layout

Designing the Overall Application Structure

The overall application structure usually depends on the object to be processed and / or the task to be accomplished. The more goals an application tries to fulfill, the more complex this structure becomes. To avoid that a huge, intransparent, and ever growing application structure emerges, begin the design of your application with a task analysis. You will find information on this topic in the "Design Process" section of the Design Guild. Use paper and pencil or graphic tools for sketching the application structure early on, as well as at later design stages.

Task steps, metaphors, or frameworks can be useful organizers for the overall structure of an application.

Designing the Navigational Structure

The navigational structure of an application refers to the paths (screen changes, page changes, and so on) that users have to follow when they proceed with their tasks. It depends largely on the overall structure of an

application. However, you as the developer decide, whether to make more or less navigational options available to users.

Applying principles such as [parallelism](#) may lead to alternate application structures with varying navigational complexity. Therefore, consider alternatives to your solution – the differences may be dramatic. Sketching an application structure and the navigational paths helps you to compare alternative designs.

When you use a metaphor for organizing your application, this usually also leads to a "natural" navigational structure.

Designing the Structure of the Functionality

Functionality refers to the set of functions that an application provides. There are high-level functions that serve the main goals of the application, and there are also numerous low-level functions, such as simple editing functions.

Usually, an application's functionality is provided in its menus, one or more toolbars, and through pushbuttons on the screen or page. Most functions are accessible through several options. While in modern applications the menu bar has a widely standardized structure, this structure is not evident to many users. They often scan the complete menu tree in search of a function. In addition, the conventional menu structure is not task-oriented. It simply opens up a universe of functions which users must select the right ones from. Procedures are segmented into units that are not connected in obvious and comprehensible ways.

Here are some alternatives for structuring the functionality of an application:

- Provide a reduced, essential functionality for critical tasks, for example, for tasks where users might need guidance:
 - On the Web you are restricted to pushbuttons on the page. Here you have no menus and therefore no choice other than to provide a reduced functionality.
 - In R/3 or other applications you can use assistants or dialog boxes to restrict the functionality.
- Use specialized toolbars, that include the functions for a certain task or for often used global tasks (e.g. a navigation toolbar).
- Use flow charts or other graphical aids to provide a sequence-oriented and task-oriented functionality.

Use categories, which make sense to users, to structure the functionality, not abstract categories that are only understood by academics.

Designing the Screen or Page Layout

Here, we simply remind you that designing the screen or page layout is an important factor in organizing an application. Refer to the respective styleguides on the SAP Design Guild for more details.

Integration

[Why Integration?](#) | [How You Can Apply Integration](#) | [Integration Through Task Flow or Work Flow](#) | [Integration Through the Object Structure](#) | [Integration Through Metaphors](#) | [Integration Through a Framework](#)

Simplification can lead to the creation of many simple, isolated tasks, and thus applications. Often, it is necessary to integrate these tasks in order to make them accessible to users. For example, a number of isolated self-service tasks can be integrated into a framework that provides easy access to the services.

Today, integration is often provided through a huge menu tree, leaving the users helplessly alone in a maze of functionality. Abstract integration using trees or net structures does not conform to human mental habits and is often a very inefficient way to organize tasks.

Yet, simplification by integration, or put differently, the integration of simple, elementary tasks into a coherent unit can be a real challenge.

Why Integration?

Integration serves several purposes:

- It makes a diverse or complex functionality that is distributed on several different program modules available and **accessible** for users.
- It simplifies the **use** of the different modules, for example, by providing predetermined sequences that conform to standard tasks
- It provides a common **framework** for different modules that enables users to create useful mental models of the application

How You Can Apply Integration

Integration may be achieved through several approaches. Here, are a few examples:

- Using the task flow or work flow
- Using the object structure
- Using [metaphors](#)
- Using a [framework](#), such as a business environment, a company organization, etc.

Integration through Task Flow or Work Flow

A task may consist of steps that are separate units of their own but must be processed in a specific order. Organizational aids that display the task flow or work flow help users to step through their tasks in case that the separate tasks are more complex.

Such organizers may be simple lists, trees, or graphical representations, such as flow charts or more concrete

pictures.

Integration through the Object Structure

If the tasks are related to processing the different parts or aspects of an object, the object structure may be used to integrate the subtasks.

Again, simple lists, trees, or graphical representations of the object may serve as integrators of the subtasks.

Integration through Metaphors

In some cases, the task may be closely related to real-world environments or may have a structure that corresponds to such an environment. Here, you can use a real-world environment as an integration [metaphor](#) for your simple applications.

An often used example is the house or room metaphor, where different rooms correspond to different subtasks or simple applications. Moving from one task to the other means moving from one room to the other. The floor plan can serve as an overview of the applications.

Integration through a Framework

It is not always possible to find a useful metaphor for integrating tasks. Sometimes a more abstract [framework](#) can provide the same effect. For instance, the organizational structure of a company or institution may serve as an integration arena.



[top](#)

Source: [Simplifying for Usability](#)

Prioritizing

[Why Prioritizing?](#) | [How You Can Apply Prioritizing](#) | [Optimization](#)

Today's software often wants to be everybody's darling, resulting in feature overload and unnecessary complexity that cannot be mastered by the users. The "priority" principle reminds us to focus on the essential tasks and user goals in order to avoid overkill and user frustration.

Why Prioritizing?

Software developers are often in a dilemma. They have to develop software for a wide range of customers with a very diffuse understanding of what this software should accomplish. This often leads the developer to squeeze in any conceivable functionality requested by at least one customer, just in case it could be needed. This usually results in complex applications that are hard and cumbersome to use.

While we cannot free developers from this dilemma, we can at least point to this potential problem. In the early design stages, a decision must be made as to whether an application is to fulfill all requirements, or is intended as a "lean" and efficient application that may not serve every purpose and user.

How You Can Apply Prioritizing

Prioritizing makes most sense in the early design stages and not during implementation. Careful analysis of the users' needs and of the task requirements are necessary before any decisions about the importance of certain subtasks or features can be made. The most important questions are:

- Which functionality is needed in most cases ("80% case")? Which is the most important, essential, or frequently used one?
- Which functionality can be totally discarded?
- Which functionality has to be offered but can be hidden from most users' view?
- How can the interface be designed so that (1) the essential functionality is presented directly to the users, (2) the less important functionality (if it is to be included) is hidden from direct view but is also accessible?

Optimization

Many people have a misunderstanding about the nature of optimizing. As there are always compromises to be made and conflicting requirements to be balanced, it is impossible to "make everything better." Optimizing processes means making overall performance better, that is, making the important and frequently used procedures easier and more efficient and also making the less important and rarely used features harder to use.

As an example, this might result in placing important functions into a toolbar and hiding rarely used functions from view and placing them into menus.

Understandability

[Why Understandability?](#) | [How You Can Achieve Understandability](#)

Software should be comprehensible for users – not only for its creators. There are many aspects to understandability: Application structure, navigation, procedures, terminology. Or more concrete: This principle requires that users always know the state of their task, what to do next, how the application reacts to certain inputs, and so on.

See also [Understanding, Learning and Relearning](#) for more information and examples on this topic.

Why Understandability?

Understandability is an ultimate prerequisite in order for users to perform any task. The better users understand an application, the more effectively they can use it because they know a greater portion of the application's functionality. This also leads to more efficiency because they can use functionality that achieves goals faster, with fewer steps, and fewer errors.

Understandability is closely related to [learnability](#). The easier an application is to understand, the easier it is to learn and relearn. Understandability is also closely related to and an aspect of **transparency**. The more transparent an application is, the easier it is to understand, and vice versa. Last not least, principles such as **self-descriptiveness** and **feedback** help you to realize understandability.

How You Can Achieve Understandability

An application may be understandable for the following reasons:

- The application is small and simple
- The task and its procedures are self-evident
- The application explains itself ([self-descriptiveness](#)) and thus is understandable through cues, such as:
 - Screen, area and group titles indicating the purpose of the respective interface element
 - On-screen instructions, diagrams or affordances
 - [Metaphors](#)
 - Explanations (more or less extensive) that are available on request. Whether these explanations actually make an application understandable, depends on the quality of the explanations and the language they use
- There are no implicit or tacit assumptions about how users are expected to behave, particularly none that contradict users' expectations
- Feedback is given on user actions, system actions, and the system state

See also [Understanding, Learning and Relearning](#) for more information and examples on this topic.



[top](#)

Source: [Simplifying for Usability](#)

Learnability (ISO 9241)

[Why Learnability?](#) | [How You Can Achieve Learnability](#)

Software should be easy to learn. Note, however, that software that is easy to learn not always is also efficient to use.

See also [Understanding, Learning and Relearning](#) for more information and examples on this topic.

Why Learnability?

A simple answer could be: Because ISO 9241 requires it. But why does this guideline require learnability? The answer is that learnability saves a lot of time and money:

- A company will profit from users that get sooner productive with a software application
- The easier and faster users learn a software application, the greater will be their motivation to use it – which in turn increases productivity
- Casual users will also profit from learnability – they never have a chance to become proficient with an application that they seldom use

Do not blame users if they cannot master your software. You as a developer or designer did something wrong, if users have problems with using your software.

How You Can Achieve Learnability

You can achieve learnability by applying the principles in this guide. Here, are just a few examples of how you can improve learnability:

- **Consistency and conformity to user expectations** Minimize the number of rules that users have to learn
- **Simple applications** which serve fewer purposes and thus have less functionality also require less learning effort
- **Error robustness** or applications where errors cannot occur relieve users from learning error recovery strategies
- **Feedback** about the success or failure of actions helps users to develop successful strategies for interacting with the application. Feedback about system states relieves users from having to remember these states, which in turn makes learning easier.
- **Transparent and self-explanatory applications** do not require users to remember details of how to use an application. Such applications are particularly suited to casual users.

See also [Understanding, Learning and Relearning](#) for more information and examples on this topic.

Self-Descriptiveness (ISO 9241)

[Why Self-Descriptiveness?](#) | [How You Can Achieve Self-Descriptiveness](#) | [Help on Request](#) | [Software without Documentation – On-screen Help or Instructions](#) | [Affordances](#) | [Metaphors](#) | [New Approaches: Assistants and Agents](#)

Self-descriptiveness requires that the system describes itself. It is closely related to understandability and one approach to making an application easier to understand.

See also [Understanding, Learning and Relearning](#) for more information and examples on this topic.

Why Self-Descriptiveness?

Self-descriptiveness provides simplicity by reducing users' memory load. Users can retain their capacity for their tasks instead of bothering with the system. They can work more efficiently.

How You Can Achieve Self-Descriptiveness

An application can be self-descriptive in a number of ways, for example:

- The application is so **simple**, the task and its procedures are so well known (or well-trained) that no further explanation is necessary.
- The application explains itself through **cues**, such as instructions or diagrams, which are presented on the screen or by using metaphors.
- **Explanations** (more or less extensive) are available on request.

Help on Request

F1 help, F4 help, and extended help (online documentation) in the R/3 system are examples of help that is provided on the users' request. Such kind of assistance has, however, severe limitations. For example, it forces users to become active, interferes with the task, and takes time and effort. Many users prefer to ask their colleagues instead of using such aids or reading documentation.

Software without Documentation? – On-screen Help or Instructions

Some authors demand that software should be usable without documentation and should be self-explanatory. For example, they embed their help or instructions (explanations, instructions, diagrams, flowcharts, and so on) within the application. Typically, the following instruction types are needed:

- A general instruction describing what the application is about, for example, what its purpose is and what it accomplishes

- Step-by-step instructions, if necessary, for accomplishing the basic tasks
- Explanations for important fields or fields with complex dependencies, legends, etc.

Instructions can be provided in the form of text or graphics, depending on what is more useful. Often these instructions can be integrated into existing screen elements, such as group headers or field labels, by expanding these text elements and using a more natural style of language. Longer texts can be placed into text boxes above or beside the work area.

Diagrams are especially useful for more complex steps and can also indicate the current status of the user.

This approach, however, has its limitations:

- It costs valuable screen space
- Professional users may be disturbed by the instructions (thus, provide options to hide or circumvent the instructions)
- Developers and documentation people must work closely together in order to design the screens or pages and the instruction texts

Affordances

The term "affordance" refers to the fact that certain interface elements have the ability to "speak" in an obvious way to the users that helps them to interact with the software. For example, buttons say "Click me!" to the users, indicating that there is some functionality that can be used. Utilizing affordances helps interface designers to do with less or no documentation (on-screen or off-screen) because the interface "speaks for itself."

Metaphors

[Metaphors](#) can also make an application self-explanatory because they allow users to transfer existing knowledge to the application.

New Approaches: Assistants and Agents

There are new developments under way, such as assistants and agents. While assistants are already used in a variety of scenarios, it is still unclear whether people like "automatic" assistance as is provided by agents. Assistants on the other hand, take the initiative away from the users and guide them. Not every user likes to be restricted in this way. In addition, in a Web environment that is inherently amodal and has no dialog windows, assistants cannot be strictly modal. Users may interrupt the procedure, move to other pages or applications, abort the wizard, and the like.



Consistency, Conformity to User Expectations (ISO 9241)

[Why these Principles?](#) | [Consistency](#) | [Conformity to User Expectations](#) | [How You Can Apply these Principles](#)
| [A Simple Rule: Do not be too Creative!](#)

These principles demand that an application should be consistent and in accordance with the users' expectations. A lot of standardization work is done to achieve these goals.

Why these Principles?

These principles make applications simpler to use because users have fewer rules to learn. Thus, they can learn more effectively, as there are fewer exceptions to the rules, and can train procedures more often. In addition, users can utilize their computer and real world experience and transfer existing knowledge from one application, procedure, or screen to another, and also from the real world to the computer system.

Consistency

A consistent application and user interface always follows the same general principles, the same interaction principles, uses the same terminology, and so on. Consistency can also be extended over several applications, a whole operating system, or even across computer platforms.

Note, however, that consistency is not a goal as such – it is just a "rule of thumb" in order to achieve simplicity. There are many cases, where users do not care about consistency, or do not even notice inconsistencies. Do not enforce consistency, just because it is a widely accepted principle and a good foundation for a "theoretical argument," if users behave differently and find it easier to disregard it. Often, only user tests can provide a final answer to the question of which solution is the best one for the users.

Conformity to User Expectations

Conformity to user expectations demands that an application behaves as users expect it to do. This principle goes beyond mere consistency, because it is not restricted to the computer systems but also connects the application to the real world.

Note that user expectations can vary largely, depending on the background and learning history of your prospective users. Computer literate users will expect that your application conforms to well-established interface standards, while beginners who are domain experts will expect that your application will behave similarly to their business practices.

How You Can Apply these Principles

In summary, you can apply these principles by doing the following:

- Let **procedures** resemble real world procedures or procedures in similar or related applications. This does, however, not mean that you should blindly copy the existing business practices in your application, if there is room for improvement or restructuring of the tasks.
- Let the **terminology** be that of the application domain or at least that of similar or related applications.

- Let the **behavior** of the application conform to **standards** and to similar or related applications.
- Define a **clear and finite set of objects and actions** that serve as useful points of reference for the users.

A Simple Rule: Do not Be Too Creative!

These principles require you to adhere to **standards**. Yes, it is great that you are a creative developer, but there should be limits to your creativity. Do not follow your personal taste, where generally accepted standards exist. Your users will acknowledge this and be grateful that you made life easier for them.



[top](#)

Source: [Simplifying for Usability](#)

Stability, Continuity

[Why these Principles?](#) | [\(Perceived\) Stability](#) | [Continuity](#) | [How You Can Apply these Principles](#)

These principles help to provide users with a stable and familiar working environment, and also to provide an impression of "continuity" when navigating through an application.

Why these Principles?

A familiar and stable working environment keeps user oriented within an application, and reduces the application's visual and perceived structural complexity. In the end, these measures increase efficiency because users do not get disoriented – which might result in floundering, dead ends with frequent backups, or even breakdowns.

These principles can also be extended across several applications.

(Perceived) Stability

Stability provides simplicity with respect to the user interface because a uniform and stable interface is less complex. Therefore, it reduces working memory load and increases familiarity with the application.

The term "perceived" indicates that stability, in this context, is a psychological, not a technical issue. What counts is what users experience as stable, not which technical interface elements are stable. For example, presenting different views in a tabstrip provides psychologically a more stable environment to users than screen changes, even though the information being presented may be (almost) identical.

Continuity

Continuity refers to the users' experience when they navigate through an application. Users feel more comfortable when the environment changes gradually. Dramatic changes can lead to uneasy feelings and even loss of orientation.

How You Can Apply these Principles

In summary, you can apply these principles by doing the following:

- Make the **environment**, that is, the overall look and feel of your application, uniform with no drastic or unexpected changes.
- Make the **presentation** of interface elements, functionality, and data uniform with no drastic or unexpected changes.
- Note: The "global" stability may be more important than the "local" or fine-grained stability.

Feedback

[Why Feedback?](#) | [How You Can Provide Feedback](#) | [Visual Feedback](#) | [Acoustic Feedback](#)

Feedback provides users with clues about the effectiveness of their actions and what is going on in an application.

See also [Understanding, Learning and Relearning](#) for more information and examples on this topic.

Why Feedback?

Feedback can relate to user actions but also to system actions or the system status.

Feedback helps users to maintain orientation in an application. Users usually gather information on the current status, perform an action, and then check the results with respect to their goals. Therefore, immediate feedback on user actions is very important to make such judgments possible.

Feedback is also very effective for learning and for establishing trust in an application. By experiencing which actions are successful and which are not, users can build up the right strategies for interacting with an application.

How You Can Provide Feedback

Feedback can be given in a number of ways, for example, by:

- Sending **messages** in status areas or dialog boxes. These messages may tell the system status, the success or failure of user actions, or may prompt the user for additional input or actions
- Providing **visual feedback**, for example, by highlighting active interface elements
- Providing **acoustic feedback**, for example, if an action is not possible

There are discussions about the pros and cons of visual and acoustic feedback. Our opinion is that both have their uses, although acoustic feedback always needs visual support – at least if it is turned off. You cannot rely on acoustic feedback alone because there are situations where it cannot be used.

Visual Feedback

- **Pro:** Can be very flexible with respect to representation (text, graphics, animation, etc.), style, length, etc.
- **Pro:** Spatial = parallel presentation that can be easily scanned
- **Pro:** Does not usually go away but stays on screen so that it will catch the users' attention.
Example: Error or status messages in reserved areas
Bad Example: Messages in status areas that disappear after a mouse move or click
- **Pro:** Can be "in place," that is, closely related to the interface element, which it refers to
Example: Feedback for links or button presses
- **Pro:** Does not disturb colleagues or other people in the same room
- **Con:** Often competes for screen space with the information which it refers to
Example: Error messages are often difficult to place on the screen, at least close to the object which they refer to

- **Con:** Require users to look at the screen

Example: Users want to work on other things while a longer lasting process is going on. They have to repeatedly look at the screen, in order to see the current status of the process

As a general rule, visual feedback is well suited to situations where more complex information has to be presented to the user and where acoustic feedback is not adequate because it may disturb other people or the environment is noisy.

Acoustic Feedback

- **Pro:** Does not compete with information on the screen

Example: Error beeps during a mouse drag do not clutter the screen

- **Pro:** Is on a different sensory channel than screen information; therefore, humans can process it in parallel to visual information

Example: Error beeps during a mouse drag do not require the user to shift attention and to look at error messages somewhere on the screen

- **Con:** Limited sound vocabulary (except if longer texts are read); assignment of sounds to actions or elements is often arbitrary

Example: A beep or a "blurb" may have many interpretations

- **Con:** Is a sequential presentation that has to be listened to more or less for the whole message

- **Con:** Is transient, that is, the message or sound is gone after it has been issued; users may not have listened to it (in some cases it may be possible to repeat the sound)

- **Con:** There is no spatial relationship to the interface element that it refers to. A relationship can only be established by closeness in time or meaning. For example, if the sound appears directly during a mouse or keyboard action a connection can be established. Otherwise, however, the referred element has to be named

- **Con:** May disturb colleagues or other people that are in the same room (or requires the use of headsets)

- **Con:** Not usable in noisy environments (or requires the use of headsets)

As a general rule, acoustic feedback is well suited to direct physical actions.



[top](#)

Source: [Simplifying for Usability](#)

Metaphors

[Why Metaphors?](#) | [How You Can Apply Metaphors](#) | [Real-World Metaphors vs. Structural or Abstract Metaphors](#) | [Frameworks](#) | [Problems with Metaphors](#)

Metaphors can be a very effective means to make software intuitively comprehensible, and thus simple to use. However, metaphors are most useful when they are closely related to the users' tasks or goals. It does not make much sense to locate the processing of a requisition in a maze or in a jungle scenery. Metaphors may also be used to integrate software components into a coherent unit.

Why Metaphors?

Metaphors help users to transfer their real world knowledge to the application. This simplifies learning, relearning, and using an application.

How You Can Apply Metaphors

You can apply metaphors in a number of ways:

- Use a **terminology** that is borrowed from real-world objects and processes, for the objects and processes of your application.
- Use **graphics** that correspond to real world objects (and maybe processes or objects corresponding to processes).
- Design the **processes** so that they resemble or are identical to real-world processes.
- Design the **desktop** or at least the application window so that it resembles the corresponding real-world setting (usually by using graphics).
- And, of course, you can combine all of these methods.

Real-World Metaphors vs. Structural or Abstract Metaphors

The most useful metaphors are borrowed from the real world because they are the most direct and intuitive ones. However, objects and processes in an application are often abstract entities and there are no direct correspondences to the real world.

If there is no direct correspondence to the real word, you may:

- Use metaphors from the real world that have an abstract, for instance a structural, correspondence to your application
- Use abstract models

Note, however, that abstract metaphors may be less useful for users because they are not intuitive, and users have to "translate" the metaphor into their own language.

Frameworks

Frameworks provide a structure that is well known to users and helps to organize an application – that is, they provide an organizational framework for the application. For example, a company or an institution may provide an appropriate structural scheme for an application.

Frameworks are not metaphors in the ordinary sense. Typically, they are fairly abstract but well enough known to be useful. They can be used instead of completely abstract organizational schemes, such as trees, networks, or tables.

Problems with Metaphors

There are two main problems with metaphors:

- The metaphor may not be suitable with respect to all software applications that are based on it. There are occasions where people use their real-world knowledge but run into an error because the metaphor is not applicable. Such cases can pose serious problems to users because the source of the error is not obvious to them. Some authors even discourage the use of metaphors for this reason.
- The metaphor may be too remote or abstract, so that users have to invest too much effort into translating the metaphor into their world. In this case, the metaphor does not help users. Instead, it forms an obstacle to them.



[top](#)

Source: [Simplifying for Usability](#)

Efficient (Re)presentation

[Why Efficient Representations?](#) | [How You Can Apply Efficient Representations](#) | [Efficient Data Representation for Data to be Manipulated by Users](#) | [Efficient Representation of Functionality](#)

This principle states that it is important to choose the right representation.

Why Efficient Representations?

Using efficient representations can make certain tasks dramatically easier. The proper choice of a representation can let users work more efficiently and reduces or even prevents errors (which also makes users more efficient).

How You Can Apply Efficient Representations

Choosing an efficient representation is important for data and functionality. This covers:

- **Data to be processed** by the program
- **Data to be manipulated by the user**, e.g. input values
- **Functions** to be selected and initiated by users.

Efficient Data Representation for Data Being Processed

The purpose of many applications is to present information to the users. The better the chosen representation conveys the **central aspects** of the data, the better and faster users will understand what the data means. So, ask first what the central aspects of the data are and then choose a representation.

Do not use fancy representations that are "modern" (for example, 3D graphics) but actually obscure or distort the data.

Consult text books on data display if you are in doubt which representation to choose. Consider alternate representations for clarifying different aspects of the data.

Efficient Data Representation for Data to Be Manipulated by Users

If users also have to manipulate the data you may have to choose a different presentation for that purpose. Here, the most important aspect is the **ease-of-use of manipulating the data**. You may also want to provide different views for judging the results of the manipulations.

Efficient Representation of Functionality

This is an often overlooked but nonetheless very important aspect of application design. Efficient representation of functionality means that users can easily find and access the functions in your application. This in turn means that users perform their tasks efficiently with your application.

This aspect is closely related to the psychological concept of **availability**, which is also a good "rule of thumb" for user interface designers. This concept states that people are only aware of the things that reside in their working memory or are easily visible in their environment. For example, when thinking about statistical problems, people only consider the cases that they know well and may come up with very biased conclusions. In the context of interface design, availability means that users often only use the functionality and information that they are directly aware of. For example, they use only those functions that are presented in toolbars or as pushbuttons on the screen but do not search the menus for them. Other examples are screen titles and status bars: These are often overlooked because they are not in the users' focus of attention as they are presented in peripheral screen locations.

In graphical user interfaces, applications usually present their functionality in – more or less standardized – menus. In addition, most applications provide toolbars, floating palettes, or pushbuttons on the screen for easier access to frequently-used functions. Usually, functions are accessed through the mouse. In some cases users find using the mouse rather disturbing. Therefore, there are also several methods accessing the functionality with the keyboard.

Here are some tips for making access to functions more efficient:

- **Important / Frequently used functions:** Check which functions are the most important and / or often used ones. Try to give priority access to these functions and make the access as easy as possible. Do not bury frequently used functions in deeply nested menus. Remember that functions that are hard to find may never be known to the users (availability heuristic: you only use what you see).
If you are not sure which functions are most often used / important consider customizable toolbars.
- **In-place functionality:** Place functions close to the screen elements they act on, so that it is obvious which elements they refer to. Provide in-place functionality where possible because this makes functions easily available to users.
- **Missing menus:** Web applications do not have menus. You have to provide all functionality on pushbuttons. The same is true for dialog windows. Therefore, limit the functionality in such cases to the necessary in order not to clutter the screen (or to confuse the user).
- **Keyboard access:** For power users, keyboard access can be very important. Therefore, provide keyboard access at least to the most important and / or frequently used functions.



[top](#)

Source: [Simplifying for Usability](#)

Aesthetics

[Why Aesthetics?](#) | [How You Can Achieve Aesthetic Appeal](#) | [Aesthetics on the Web](#)

Aesthetic appeal concerns the overall appearance of an application. You deal with these issues at a relatively late stage of the development process. For users, however, it is important from the beginning. It can determine users' opinions of your application and whether they like it or not.

Why Aesthetics?

Aesthetics has several aspects in application design. It serves as a motivational factor as well as an organizational factor. Both factors eventually contribute to efficiency.

Motivation: People consider aesthetics as a basic need. They like to work in environments that meet at least basic aesthetic requirements. They dislike ugly environments. As a result, they are more motivated and perform better if their aesthetic needs are met.

Trust: Though trust is more a psychological issue, it is nonetheless very important for the relationship between users and the software they use. This is especially true on the Web. A high quality visual design builds up trust in an application while a dull, and even more a flawed, visual design reduces trust. This phenomenon can also be applied to other aspects of interface design, such as terminology and errors. Bugs, jargon, and typos considerably decrease users' trust in an application and their motivation to use it.

Organization: Though it is still unclear, which factors lead to an aesthetic design, or how to measure aesthetics, it is well established that an aesthetic design is usually better organized than an unaesthetic one. A better organized spatial layout of screen elements helps people to find information or functions easier and faster, thus leading to a better performance.

How You Can Achieve Aesthetic Appeal

As long as only the screen layout is involved, follow the guidelines for screen design. Especially, respect the Gestalt laws that act as "unconscious information organizers." A simple rule to follow is: Place together, what belongs together. Use enough white space to separate elements, and do not cram up your screens.

If graphic design is involved, as is the case with Web applications, incorporate professional designers into the development team or let external graphic designers do the art work. Do not try to be a designer yourself!

Aesthetics on the Web

A Website or application is the showcase of a company or institution on the Web. To a high degree, its success depends on aesthetic criteria. Aesthetic appeal is most important for the first encounter or encounters with a Website, but it is also important in the long. A well-designed Website stimulates motivation. It also serves to improve efficiency, because users will navigate the site better, get the desired information, and do their tasks faster.

Goal Orientation

[The Task's Goals](#) | [The Users' Goals](#) | [How You Can Achieve Goal Orientation](#)

An application performs a certain task, or – from the users' point of view – serves a certain goal. With regard to simplification, this goal should be intuitive and understandable. Ideally, an application should serve one simple and understandable goal, or just a few of them.

The Task's Goals

We put task goals and company goals together, because – at least ideally – an application should reach the company's goals. For this reason, it has been bought and made "productive" – or not, if it fails. An application can fail in a number of ways. We will describe the two most important ones.

The simplest but most severe cause of a failure is that an application does not meet its goals – or at least that its users fail to perform the required task. While in the first case, the design specification may be flawed, in the second case the design specification may be correct, but the way the software tries to accomplish its goals are not appropriate: The procedures may be awkward, the functionality obscure, the flow of control unclear, and so on. This is usually an indication of a design process that did not incorporate users. In other cases, the users may have been asked, observed, and analyzed in the initial development stages, but the design was never verified with actual users. As we say below, the best insurance against such failures is to implement a user-oriented design process.

The second reason for a failure is that an application tries to perform a multitude of tasks, resulting in a feature-laden, overly complex application – a typical characteristic of today's software. Often, this multitude has been required by the customers (by different customers that use the same software or sometimes even by a single customer), and the developers have merely done what the customers told them. However, this is not a valid excuse: You as a developer or designer have to take into account that customers and end users are not the same. Among "customers" we count the people who buy the software and the people who are responsible for choosing and maintaining it. Both groups are "loyal" to their decisions and both usually care little about end users. However, they really should do so because the decision for a certain piece of software largely determines how efficient the end users will be.

End users are the people who eventually have to use the software in their daily work. From the end users' point of view, applications that serve a multitude of goals and subgoals are a hassle. They have to keep all these goals in mind and to keep track of them. Even worse, often either the relationships between tasks and subtasks are unclear, or the tasks (and goals) form a deeply nested hierarchy that users find difficult to understand. Therefore do the following:

1. Make sure that task(s) coincide with the user goal(s) for your application
2. Let the users know what your application's goal is and how they can achieve it

Ideally, a simple application should serve just one goal, or at the most, a small set of closely related goals.

The Users' Goals

Alan Cooper has drawn our attention to the users' goals, which often may not coincide with the program's or

the the company's goals. However, if you know the users' goals, your application:

- Will not frustrate users because it takes care of the users' goals
- Will also find ways to integrate company and user goals, so that both parties' interests are satisfied

We have already listed most of the typical user goals in the section on [users and their needs](#). Therefore, we recall only the two most important user goals, namely that (1) users want to accomplish their tasks quickly and efficiently, and that (2) they do not want to look stupid. Most of the guidelines in this guide just care for these two user goals.

If you design a new application you should, however, carefully look for additional user goals that are important for the interface design and not rely on just these two.

How You Can Achieve Goal Orientation

Use methods found in the "Design Process" section of the Design Guild to determine your application's goal(s). Ask users and domain experts in order to identify their goals and needs. This is the one big issue here. It depends largely upon how the development process is institutionalized in your company and how user-oriented it is. The following items are just a few "reminders":

- Design a simple and transparent application that users can understand, so that they can achieve their goals
- Avoid distracting users from their tasks. For example:
 - Protect users from technical details
 - Use a terminology that is task-oriented, instead of technical jargon
- If possible, use metaphors that help users to transfer their domain knowledge to the application



[top](#)

Source: [Simplifying for Usability](#)

Suitability for the Task (ISO 9241)

[Why Suitability?](#) | [How You Can Achieve Suitability](#)

Why Suitability?

"Why suitability?" may be the most stupid question in this whole guide. It is, however, alarming to see how many applications fail to meet this requirement – not ten or twenty years ago, but today. Although we have the most advanced interface technology ever at our disposal, many users are still dissatisfied with, or frustrated by the software they use.

So, the answer to the above question is very simple. People want to accomplish their tasks easily and efficiently with the software they use. The question of why things are how they are today is, however, much harder to answer and not within the scope of this guide. Here, we can only try to offer some ideas as to how you can design an application so that it better fits the task it has to perform. You will find these ideas below and throughout this guide.

How You Can Achieve Suitability

Again, we strongly recommend you use the methods found in the "Design Process" section of the Design Guild or similar methods for task and user analysis as found in the usability literature.

Key elements of such an analysis are:

- A good understanding of the **task domain**, the **current work practice**, the **task** itself and its embedding into the organizational and communication structure of a company. Site visits are a good approach to observing the current work practice and envisioning improvements
- Knowledge about the prospective **users**, their abilities, needs, and goals
- A look at the **market** and the **competition**. The marketing department should assist the development groups in gathering and consolidating this information
- Creating **prototypical users** (like Alan Cooper's "personas") and **scenarios**. Scenarios are realistic usage situations that demonstrate how typical tasks are accomplished with the software.
- **Reusing the scenarios in user days**: Scenarios can be used for first demonstrations of the concepts, for "quick and dirty" tests with prototypes, as well as for final tests of the implemented application – be it in the lab or in the field. In addition, scenarios can be reused for reviews, where usability experts have a close look at the application.



[top](#)

Source: [Simplifying for Usability](#)

Error Robustness (ISO 9241)

Why Error Robustness? | How You Can Achieve Error Robustness

Error robustness means that a software should be permissive to errors. For example, it should let users undo erroneous steps. This robustness to errors helps to provide a more "relaxed" climate for users. Users need no longer be anxious about causing damage to data, the software, or even the computer.

However, error recovery procedures usually do not make a system simpler. It is far better to design a software so that errors cannot occur (or are prevented in unobtrusive ways).

Why Error Robustness?

Error robustness is one of the ISO 9241 requirements. Errors are possibly the most important psychological barrier between users and their software. People are anxious about committing errors, often blame themselves for system errors, and may be unwilling to use an application that produces errors. This antipathy may even carry over to other applications and to computers in general.

Errors are also a productivity stopper. Errors reduce productivity by requiring users to put effort into error recovery procedures instead of productive work. They may even bring productivity to a halt if nothing works anymore.

How You Can Achieve Error Robustness

Here are some guidelines for achieving error robustness or error tolerance:

- **No user error philosophy:** An application should never make users feel that they are in the wrong
- **Reversible actions:** Make actions reversible whenever possible and inform users about any action that is irreversible
- **Protection against destructive errors:** Prevent and detect errors, particularly errors that have destructive consequences

However, following these guidelines may not always be an easy task. Implementing an undo function may be difficult or even impossible. In business applications, for example, there is no undo possible for transactions. You have to cancel them.

There are some precautions that you can take in a business application without an *Undo* function:

- Provide **help on input values**. Wherever possible, provide a fixed set of input values which users can select from.
- Wherever possible, **limit choices** depending on the current application context. This applies to user inputs and options, as well as to functionality. Limiting the number of options is one of the most effective error prevention strategies.
- Provide **safety prompts** for "dangerous" actions, such as quitting an application or booking a flight. Restrict safety prompts to the really dangerous cases.

In general, provide a consistent user interface that makes interaction predictable for users.



[top](#)

Source: [Simplifying for Usability](#)

Distribution of Tasks (Division of Labor)

[Why Pay Attention to Distribution of Tasks?](#) | [A Simple Rule](#) | [Restructure the Task](#) | [The Fewer Steps, the Better](#)

When you design an application, you automatically distribute the steps that are needed to execute a task or to attain a goal between the computer and the user. This becomes particularly evident when you use a method such as Essential Modeling. Here you list the user's tasks or intentions in one column and the system's tasks in a second column. You can easily evaluate the user's workload and the system's workload.

Example: Getting Cash – Essential Use Case *)

User Intention	System Responsibility
Step 1: identify self	
	Step 2: verify identity
	Step 3: offer choices
Step 4: choose	
	Step 5: dispense cash
Step 6: take cash	

See also [Who Does the Work?](#) for more information and examples on this topic.

Why Pay Attention to Distribution of Tasks?

Usually, a computer system running an application is interactive. The system and the user communicate in order to attain a goal or to fulfill a task.

With respect to distribution of tasks, the most important aspect of this communication is that the users have to provide data that the system needs for processing the task. This may mean more or less work for the user. This depends, among others, on:

- How **much** data the users must enter
- How **often** users must enter data
- Whether users have to **calculate** or otherwise **provide** intermediate results
- Whether users have to **check** intermediate and / or final results

Heavy workload for users may result in strain and, possibly, a lot of errors.

A Simple Rule

As a simple rule, let the computer do as much of the work as possible.

For example, do not prompt the user for information that the application already knows or can get otherwise. Also, do not let the user perform steps that the application can do for him or her.

Restructure the Task

Sometimes it may be helpful to restructure a task in order to simplify the interaction and to enable the system to take over a larger share of the work load.

The Fewer Steps, the Better

Remember, the less steps users have to perform, the fewer errors they can make, and the faster they get their job done!

Note, however, that there is no rule without exception. Often, users prefer to use simple commands that require repeated keystrokes to more powerful and complex commands that require just one or a few keystrokes. User behavior typically depends on the frequency of the actions. For rarely used actions, users find it easier to learn fewer commands and to use just a "handful" of simple commands that require more physical actions.

*) Example taken from Larry L. Constantine & Lucy A. D. Lockwood (1999), Software for Use. Reading, MA: Addison-Wesley.



[top](#)

Source: [Simplifying for Usability](#)

Balance of Controllability vs. Guidance – Amodality vs. Modality

[Why Controllability?](#) | [Why Guidance?](#) | [When User Control?](#) | [When Guidance?](#) | [How You Can Achieve Controllability](#) | [How You Can Achieve Guidance](#) | [Questions to Consider Regarding Control Issues](#)

Controllability is often translated as "the user is in charge, not the computer." The user should hold the initiative at all stages of the processing. **Guidance** is complimentary to controllability. There are situations or users, which require that the system guides the user.

Amodality is one way to put the users in control. The system does not force the users into a predetermined sequence of steps. The users can, at any time, choose how to proceed. **Modality** – forcing users to do just one or more required steps – is just the opposite. Its use has been widely discouraged by proponents of graphical user interfaces.

However, the "ideal" amodal graphical user interface fails in many situations. New approaches, such as assistants, try to address this problem. Modal dialogs, although often considered as bad interaction design, can also be used to guide users.

With regard to simplification, you have to balance controllability with guidance for an optimal solution. Consider the **task requirements** as well as the prospective **users**, when you decide whether to concede the users more or less control.

See also [What Is in Control?](#) for more information and examples on this topic.

Why Controllability?

Letting the user be in charge improves the users' motivation and thus improves their engagement with your application.

Why Guidance?

Use guidance to:

- Prevent users from floundering
- Prevent user errors
- Let users do certain tasks more efficiently
- Collect data required by the system

When User Control?

Setting the user in control should be your **primary option** when you are considering control issues.

When Guidance?

Guidance may be necessary for a number of reasons:

- The **users** are beginners or casual users that do not know or remember the required steps, options etc. However, even proficient users may require or prefer guidance in some of the following situations.
- There are **many options or functions**. It is unclear which are relevant in a certain context.
- The **procedures or decisions are complex**. Users do not know how to proceed in a procedure or which decision is the right one.
- There are **dependencies** between tasks steps. Your application has to ensure that these dependencies are observed by the users.

There is **time pressure** or **critical situations** that have to be handled with your application. This may also apply to **routine procedures** where users do not want to take more steps than absolutely necessary.

How You Can Achieve Controllability

You can achieve controllability by:

- Using **event-driven procedures** that are triggered through user actions
- Using **amodality**. That is, by not forcing users to do a certain action
- Confronting users with the prospective **consequences** of their actions or choices

Some examples controls that put users in control:

- Use **amodal (parallel) windows** that do not block the main processing, where possible. Avoid modal dialog windows.
- Use **tabstrips** or navigational menus to ensure an unrestricted choice of views, options, functions etc.

Avoid dependencies between data or steps. Dependencies may, to a high degree, contribute to an application's complexity.

How You Can Achieve Guidance

Older computer systems guided users through "brute-force modality." They required users to enter data in a specified sequence, to select items or functions from just one menu, and so on. Nowadays, there are more guidance options at your disposal.

Modal dialog windows guide users by requiring them to process a dialog window before they can continue with the main processing.

Assistants (Wizards) consist of a number of views with processing options. These are provided on a dialog window with standardized navigational controls (*Back, Forward, Cancel*). Assistants give users limited control when they step through the views. Usually, they also provide previews for the different choices to make decisions more clear for the users.

Both means are based on dialog windows, meaning that they are tailored to secondary processing. It makes little sense to put your application's main processing into a dialog window or to design your application as guided-only (there may be exceptions).

Context-specific functionality means that you dynamically restrict choices of functions, options, navigational targets, etc. to the relevant ones. This way, you prevent users from selecting irrelevant ones or from being confused through a large number of choices. You can make functionality context-specific by:

- Enabling and disabling menu items or pushbuttons (in toolbars or palettes)
- Providing contextual menus

In addition, you can let users **customize** toolbars, contextual menus, menus etc. to their needs (so that they can "restrict" themselves).

Questions to Consider Regarding Control Issues

There are two simple questions that you should ask when considering control issues for your application:

- What can users **do** (or not do) in this situation?
- What can users **do wrong** in this situation?

The first question reveals hidden modalities or restrictions and may help you to break these unnecessary barriers for users.

The second question may often reveal hidden complexities and traps that users can step into in your application. This helps you to redesign your application or to introduce more guidance in special situations.



[top](#)

Source: [Simplifying for Usability](#)

Parallelism

[Why Parallelism?](#) | [Problems with Parallelism](#) | [How You Can Apply Parallelism](#)

Parallelism can mean:

- Parallel processing of tasks
- Parallel provision of functionality
- Parallel provision of information

Any of these aspects may be exploited alone or in combination with others to make an application easier and more efficient to use.

Why Parallelism?

Parallelism can improve applications in several ways. It can improve:

- **Controllability** because users are not forced to follow a given path or have more interaction options
- **Transparency**, particularly its **self-descriptiveness** aspect, because more information is available
- **Efficiency** because tasks may be executed simultaneously, in a flexible order, or because users do not have to wait for the final result of a subtask
- **Navigation** because the navigational complexity can be reduced

Problems with Parallelism

Parallelism may increase the **complexity** of an application. Any of its advantages may turn into a disadvantage if not applied properly.

How You Can Apply Parallelism

Here are just a few examples, because the possibilities to apply parallelism are very varied:

- Use parallel windows, frames, screen areas or similar techniques to display different information, or processing options, in parallel
- Use amodal windows or floating palettes to offer functions or options permanently
- Use alternate representations for data that can be viewed in parallel



[top](#)

Source: [Simplifying for Usability](#)

What is Really Necessary?

[Tasks, Goals](#) | [Structure, Navigation](#) | [Functionality](#) | [Procedures \(Number of Steps\)](#) | [Data](#) | [Internal Structures](#) | [Interface \(Screen Elements\)](#) | [Terminology](#)

Reduction is the "basic" principle of simplification! Here we discuss some possibilities for reduction.

Motto: No special wishes. More is less!

Tasks, Goals

Web applications should follow the rule "One task - application"!

Example: A typical question might be whether the tasks of changing prices and finding information about prices should be one or two applications.

Structure, Navigation

Reduce applications to as few screens as possible, and keep navigation at a minimum.

Example: Tabstrips reduce the number of, and avoid navigation between screens. Moreover, they help to preserve the context of the application.

Example: Be cautious with "rampant growth" of the application structure. Possible causes for this can be jumps to search screens or "trips" to related applications (e.g. for maintaining master data, such as customer data).

Functionality

Offer only the essential functionality (80/20 rule)! Offer this functionality in ways that users can easily find it.

Note: As easy Web transactions do not have menus, and can therefore only offer a limited number of functions, it is important to choose the functions wisely!

Example: Only the functionality that catches the users' eyes is really used!

If there is less important functionality, hide it in order not to clutter the screen.

Note: Optimization means making the important things easier, and the less important things harder!

Procedures (Number of Steps)

Count the number of mouse clicks needed for a procedure! In most cases it is a good rule of thumb to assume that fewer clicks leads to easier procedures.

Web Guru Nielsen: In the Web users vote "with their feet (=Clicks)". A click too many, and the users will leave your Website!

Example: Dialog windows often disturb the flow of the processing. As easy web transactions do not have dialog windows, this lack of a feature is your chance for achieving smoother procedures!

Data

Complexity of data can be reduced with regard to

- amount, structure and presentation

Example: Filters allow the reduction of the amount of data to be displayed.

The **presentation** for a data set can be more simple than the structure of it, especially if the amount of data is small.

Examples: A hierarchy often can be presented as an ordered list, e.g. a 5-level hierarchy of goods can be reduced to two levels or even one, if the number of goods is not too large.

Example: In case that data can be presented as a flat or two-level list, presentation with a tree control is not appropriate - even if everyone regards this as „cool“!

Internal Structures

Do not harm users with the complexities of the inner working of your application or its data structures!

Example: Do not require users to enter a group name for salespersons, if the company has only three salespersons.

Interface (Screen Elements)

Put only those interface elements on a screen or page that are really needed! This is especially true for fields.

Example: If users usually fill only three fields in a form, why display 30 fields that "might be needed" by some users? Often these three fields are "somewhere" in the middle of the screen and hard to find for users.

Example: Many users believe in the myth that, if there is a field on the screen, they have to fill it. So, help

them by dropping unnecessary fields!

Terminology

Talk the users' language. Do not use abstract language or jargon.

Example: Users could not find the "parts list" in an application, because it was called differently.

Example: SAP uses its own terminology for many business terms. Users often do not understand these terms.



[top](#)

Source: [Simplifying for Usability](#)

Who Does the Work?

[Knowledge](#) | [Actions](#) | [System Weaknesses](#)

Computer work means that user and system work together on the user's tasks. This means that there is a division of labor and responsibilities between the two. Your goal is to make the work as easy and efficient as possible for the user - and not for the system. Therefore, check what the system can do, and what has to be left to the user.

Motto: The system should serve the user, not the other way round!

Knowledge

Do not require users to enter data that the system already knows or could retrieve from elsewhere!

Example: If the user already entered parts of an address that clearly identify the time zone - like "Chicago" and "USA", why require users to enter the time zone manually?

Note: The system can remember data for the user, like the history of input values or their frequency, and provide useful default values for many fields.

Actions

Do not require users to do things that the system could do for them.

Example: Do not let users enter the VAT if the system can calculate it.

System Weaknesses

Do not let users suffer from system weaknesses!

Example: "Zero indicator" - Users were required to set a "zero indicator", because the system could not distinguish between the entry of a zero and no entry.

Example: Provide templates for "formatted" data such as dates, times, and so on.



[top](#)

Source: [Simplifying for Usability](#)

Who Is in Control?

No Deception! | Brute Force: No! | Brute Force: Yes! | Transparency

To make a system appear simple, it is important to choose the right mixture between system guidance and user control. While beginners and casual users often fare better with appropriate guidance from the system, more experienced users prefer to be in control. Often it is not an easy to find the right mixture, and designers also need to be aware of some of the traps they might fall into.

Motto: Me at the steering wheel!

No Deception!

Logical dependencies between data or task steps often exist, which require that users do the steps, or at least some of them, one after the other. The user interface should not pretend that these steps can be done in an arbitrary order, if this is not the case!

Example: A toolbar or a collection of buttons implies that these buttons can be pressed in any order. If this is not the case, this control or control array is the wrong choice! Online instructions are needed that inform users about the required sequence of button presses.

Example: In an existing application two steps had to be done one after another. In either case a dialogue window was presented whenever the respective button for the step had been pressed. However, if the user pressed the wrong button, the "correct" dialogue that belonged to the other button was presented. This is not the preferred way to teach users how to use an application!

Example: Do not use tabstrips if there are dependencies between views! Users expect that they can click the tabs in any order.

Hint: If there are dependencies, use a wizard that guides the users through a fixed, or at least system-determined, sequence of steps.

Hint: Minimize the use of - or better, do not use - data that controls the flow of the processing (this is a frequently used R/3 "feature"). Users often do not understand dependencies. Dependencies are not intuitive and make an application difficult to learn!

Brute Force: No!

Opposite to the problem mentioned above, do not enforce a sequence of steps, if users can do these in any order!

Example: Use a tabstrip instead of a sequence of screens, if views are independent from each other.

Brute Force: Yes!

As a general rule, guide the users, even if they are professional, if processes are complex and intransparent.

Example: Use wizards or screen sequences to enforce a certain processing order. Wizards make the process more transparent to the users, if they indicate the current position in the process, provide an overview of the process as a whole, and present a preview of the consequences of possible actions.

Transparency

Users often feel uncomfortable and dominated by the system, if it assumes control. Therefore, if system guidance is appropriate, inform the users what is happening and where they are going.

Example: Wizards are a good way to guide users and to keep them informed.



[top](#)

Source: [Simplifying for Usability](#)

Understanding, Learning and Relearning?

[Purpose of a Screen](#) | [Order of Screen Elements](#) | [Support](#) | [Affordances](#) | [Implicit / Tacit Assumptions](#) | [Standards](#) | [Complexity](#) | [Feedback](#) | [\(Re-\)Learning](#)

Users consider an application to be easy, if they understand it, especially if they even understand it having not used it for a long period of time. Thus, simplicity can be achieved by making applications understandable and easy to learn and re-learn.

Motto: Understand what's going on!

Purpose of a Screen

Provide screens, screen areas, and groups with meaningful names!

Example: As a negative example, the current SAP ESS applications do not name areas - and users struggled in the user tests...

Example: The Windows title bar provides information about the purpose of a screen. However, this information is mostly overlooked!

Order of Screen Elements

Arrange areas and screen element in a logical order that fits the order of processing or reflects logical dependencies!

Avoid a "Yoyo" style of flow of control where processing jumps up and down.

Example: Do not reuse a screen area above another area for displaying detailed data from items in the lower area - just because it is currently unused.

Better: Exchange item list and detailed data in the same area, and keep the context stable.

Support

Use written explanations, diagrams, graphics, and so on, on the screen as online support. Users prefer this support to reading documentation.

Example: Online documentation is rarely used, because most users do not like to read longer texts on the screen.

Affordances

The term "affordance" refers to controls' ability to create certain user expectations about their functions, or to "invite" users to interact with them. This way users can be lead intuitively to a certain behavior, and thus guided without using any force.

Example: Buttons "talk" to users, "click me" fields say "enter data" and so on.

Example: User tests revealed that links in tables have a stronger affordance for users to click them than icons in tables.

Implicit/Tacit Assumptions

Avoid implicit / tacit assumptions about how an application has to be interacted with. Make such assumptions explicit by displaying instructions, or using other methods, to make the interface self-explanatory!

Example: In an application users could only cancel an order by providing a reason for the cancellation. The users, however, were looking for a "Delete" icon to cancel the order.

Standards

Consistency makes an application easier to use, learn and re-learn. One way to achieve consistency is to adhere to proven interface standards as they are laid down in style guides.

Therefore, refrain from too much creativity in interaction design, and adhere to the standards - your users will appreciate this! They expect your application to function in certain ways. Do not surprise them!

Note: The infamous word from developers "Style guide? Never read it!" is out!

Complexity

Do not put elements onto your screen that are unnecessary and "might" be used by some unknown users! Those unnecessary elements shift the relation between foreground and background information so that users have difficulties in finding the elements they need.

Example: Do not put seldom or never used fields on the screen!

Example: Do not display table columns that are not generally needed!

Hint: Provide access to rarely used fields or table columns if users occasionally really need to see them. However, you can make this access a little bit harder without harming most users.

Note: The infamous quote "The final selection of fields / table column will be done when the system is

customized" is no valid excuse for a bad interface design! This quote just serves as a justification for imposing the design on the customer!

Feedback

Provide feedback to the users regarding

- the current state of the processing,
- how correct user entries are,
- the consequences of possible user actions.

If users know what is going on and what might happen, they feel much happier and find the system easier to use.

Example: Display the progress of a process (for example, of a copy process).

Example: Indicate how far the user is in a procedure that consists of a number of steps.

Example: Provide a preview for possible settings and options (for example, a diagram assistant might provide a preview of the diagram in case certain settings are selected).

Example: Tell users about the consequences of actions, especially if the consequences might be severe for the users (for example, tell the users what will happen if they hit the "Order Now" button).

(Re-)Learning

(Re-)Learning is made easier through...

- understandability of the user interface
 - for beginners, and
 - for casual users
- simple and consistent interaction rules
 - Similar or equal procedures should always work the same way!
 - But note: Sometimes consistent and simple rules may lead to less efficient procedures. However, users often remember only the simple rules and do not use the efficient procedures.



[top](#)

Source: [Simplifying for Usability](#)