Part VII.

Engineering

34. Introduction to Engineering

In HCI, engineering refers to the application of principles and methods from engineering research to build interactive systems. We need engineering, because building an interactive system is hard. A wide range of technical functions must be realized in software or hardware while at the same time addressing a broad set of human-related factors. Engineering methods systematically tackle this challenge and ensure that the resulting system can be used efficiently and safely, manufactured at low-cost and high quality, and supported after launch and eventual disposal.

HCI research in this area is concerned with understanding of engineering methods can be used more effectively and efficiently in developing new methods, tools, and systems. Below are two representative examples:

- Optical see-through augmented reality (AR) enables users to wear AR glasses and interact by articulating gestures using their fingers and hands in 3D space. A difficult problem for such a system to be realized is to identify such gestures within a stream of unrelated movements, this is referred to as *key gesture spotting*. If the system does not work well then it results in user frustration as intended users are not recognized correctly and unintended gestures suddenly give rise to commands in the system. *Gesture Spotter* [744] is a rapid prototyping software tool (Chapter 38) that allows developers to easily add their own gestures and have them reliably recognized among a continuous stream of hand and finger movement. The tool includes both a graphical user interface that allow rapid design iteration of different gesture sets and an application programming interface for deploying apps with integrated gesture recognition. Evaluations with developers demonstrated both were effective in enabling non-experts to support key gesture spotting in AR glasses applications.

- Safety critical user interfaces can give rise to human errors that turn out to be disastrous (Chapter 37). As a result there is work on using formal methods, rigorously defined mathematical models, when designing such user interfaces (Chapter 38). Formal methods allows user interfaces to be verified to be correct against a specification. There are several tools available that allow HCI designers and developers to use formal methods to verify that user interfaces are behaving correctly. However, these tools have been developed to examine specific issues of user interface design and it is unclear how such tools can be fruitfully used as part of a user-centered design process (Chapter 33). Campos et al. [123] investigated three such tools to understand how each of them would fit within a user-centered design process. A finding is that these three tools are largely complementary and address different aspects in a user-centered design process. For example, one tool was more suitable for rapid prototyping while another tool was advantageous when addressing general usability concerns. The paper guides developers on how to select a suitable tool by providing 22 criteria for evaluation.

Engineering is tightly interwoven with design, which we discussed in the previous part (Part VI). A system cannot be 'just built': its construction needs to be designed. Therefore, problem-solving, and exploring and evaluating options are central also for engineering. In this sense, an engineer is also a designer, designing bridges, motors, circuit boards, mechanical structures, software, systems, and so on.

In this book, we do not attempt to clearly delineate design and engineering. The difference is rather a shift in focus: in engineering, we focus on what is required for *building* a system that has desirable properties. The system cannot be just a plan or prototype, but must actually work and be effective, efficient, safe, and fulfill users' needs and wants. In order to realize this, we must design the technology itself—identify a functional architecture for the system, translate given functions to 'function carriers', ensure that requirements are correct and have been met, and so on.

This Part introduces perspectives, ideas, principles, and processes from engineering and explain how they can be used in the context of interactive systems.

The design of interactive systems is a *systems engineering* problem where a key challenge is to understand the *system* we are trying to build and its relation to other systems. Systems are pervasive: we are surrounded by them. We live in an ecosystem and may take part in economic and political systems. We are typically part of some form of organizations, such as university, and we work using computer systems that are interconnected using the Internet. Systems are reliant on other systems to function. For instance, our computers are reliant on the electrical grid. Further, systems are embedded within other systems: a laptop is reliant on a display, central processing unit, memory circuits, input and output devices, and so on. The human body consists of several subsystems, such as the circular and digestive systems.

Chapter 35 introduces *systems thinking*, which is a perspective in engineering that allows us to reason about the complexity of systems and a set of principles that help us design, build and support systems. In addition, it is also important to be able to understand a system, which can be done by mapping out systems from various perspectives, such as for example from the perspective of flow of documents and information in a large organization. Chapter 35 introduces several such system mapping techniques which can be used to gain a holistic understanding of the internal mechanisms driving a system and relationships to other systems.

Another key facet of engineering is conceiving, understanding, and managing appropriate processes that allow us to systematically arriving at interactive systems with desired qualities, namely that they are effective, efficient, safe and fulfill users' needs and wants.

We will therefore introduce *design engineering* as an approach to engineering HCI systems (item VII). Design engineering, or engineering design, is an engineering discipline concerned with processes that allow systematic creation of products, systems and services. We believe design engineering is very useful in guiding the design of interactive systems, which typically exhibit a high degree of complexity and are often required to interface with many different types of end-users in many use-contexts.

As we will see, design engineering has an overlap with the design processes we discuss in the chapter on design practice (item VI). The distinction we make in this book is that design engineering is focused on arriving at a *system solution* for a solution-neutral problem statement that is specified in such a detail that it can be built, verified, and validated. As such, design engineering here provides practical guidance to the reader on how to realize a system. Thus we believe it is more useful to treat design engineering as an individual chapter in the engineering part, which is overall focused on providing perspectives and techniques for realizing systems.

A third key facet of engineering is *safety and risk*. A system must be safe to use and understanding how to design an interactive system that does not intentionally or unintentionally harm users is vital. To this end Chapter 37 introduces methods that can be used to ensure interactive systems are kept safe. We will explain what is meant by *human error*, such as taxonomies for human error and how human errors are analyzed. We will also discuss risk by defining what is meant by risk and reason about how we can assess and manage risks. Chapter 37 presents several *risk management* strategies that can be used to achieve this for interactive systems. Finally, another important aspect of safety and risk in an interactive system is the concept of a *failure*—an unintended outcome of a system. We will explain how *reliability engineering* allows us to statistically model probabilities of failure and thereby provides us with a mathematical method for estimating the reliability of various system structures.

Interactive systems rely on complex *software*. Chapter 38 introduces principles and approaches in *software design and architecture* that enables constructing interactive user interfaces. It then explains how *toolkits* can be designed to assist developers in implementing interactive systems software. A third aspect is how to enable end-users to program computer systems. This is known as *end-user development* and there are several approaches to design such systems, such as programming by example. Finally, to ensure our interactive systems work correctly in software we can construct a formal model of interaction that allows us to reason about properties of interaction. For example, we can represent interaction as a finite-state machine and use techniques from computer science to reason about whether a user can reach all states in the system. We can also use a formal representation to represent a system function, such as *undo*, and prove various properties of such a system.

Finally, engineering is also about using computational for tackling challenging tasks in interaction. Chapter 39 presents *computational methods* for interactive systems. The central notion in this chapter is the existence of a model that allows algorithmic or mathematical reasoning about some aspect of interaction. This chapter will begin by explaining how such models may be created and how they can be used to analyze an interactive system at an early stage of the design. We will discover that such models can capture controllable and uncontrollable parameters of a system. Uncontrollable parameters are parameters that affect the system's outcomes but are outside the control of the designer, for example, the user's level of motor control, cognition, or experience with a particular system. Controllable parameters are parameters we can tune and adjust to improve outcomes. Thus, they allow us to perform *optimization*. However, we can do more than reason about potential outcomes and optimize controllable parameters. In addition, we can build a system that infers or predicts users' actions or makes decisions in collaboration with or on the behalf of users. Techniques from *pattern recognition and machine learning* allow designers to model tasks as inference problems that the computer can address. For example, a system can recognize users' 3D gesture articulation from observations from a depth sensor. Chapter 39 will outline what type of tasks such approaches can support and some key design issues to consider when using such methods in interactive systems.

34.1. Engineering Perspectives in HCI

It is natural to ask what engineering can bring to HCI that is not already there. After all, HCI already has multiple principles, approaches, and methods in its disposal when designing interactive systems. Below we set out to answer this question by explaining how engineering can serve as a *complementary* approach to established HCI design practice.

34.1.1. Building the Right Thing

One fundamental aspect of engineering is understanding what to do and how to do it. This is called *task clarification*. An easy—but critical—mistake to make is to design a solution for the wrong problem, or to create a solution that does not solve any problem users are facing—a so called "solution in search of a problem". This can be prevented by understanding how to arrive at a valid *problem statement* that is relevant to the goals users are trying to achieve.

To arrive at a valid problem statement can be more difficult than it sounds as it requires a deep understanding of the nature of the problem which often will necessitate knowing about all relevant aspects of the problem context, users' needs and wants and the system that the interactive system or service will be embedded within.

However—surprisingly—it is insufficient to merely arrive at a problem statement. This is due to a design pitfall known as *design fixation*, which in essence amounts to immediately settling on the obvious solution that first comes to mind (see section VI). This "engineer's decease" prevents the designer, or design team, from fully exploring a larger set of possible solutions and artificially constrains the design space. It therefore frequently leads to less creative and less optimal design solutions. This may sound trivial but it is a very real design problem with potentially a very high cost. Failing to be creative about how to do things leads to noncompetitive products and services and even failing to be creative about how to *not* do things may lead to expensive product or service failings later on. For example, failing to be creative about how to test a product may lead to a failure to detect a product flaw which later results in expensive product recall actions.

One technique to avoid design fixation in engineering is to transform a problem statement into a *solution-neutral problem statement*, which describes the problem at a suitable level of abstraction without making any direct references to solutions. For example, instead of starting out with a problem statement such as "Build a physical thumb keyboard that uses the QWERTY layout for a mobile device" the design team may consider a solution-neutral variant: “Devise a method that allows a user to enter text on a mobile device”, or “Devise a method that allows a user to transmit information in a mobile setting”, which has an even higher abstraction level. Raising the abstraction level increases the search space.

Another technique in engineering to avoid design fixation is to first design a product or service at a *functional level*, which only makes reference to the relationships of the *functions*. For example, a keyboard can have an overall designated function `Enter Text`, which can then be decomposed into its key subfunctions, such as `Type Key`, `Provide Word Prediction`, etc. Importantly none of these functions make reference to a *solution*. For example, the function `Type Key` can be realized using hard keys, chiclet keys, membrane keys or touchscreen keys. These are all examples of *function carriers*. That is, they are designated solutions that carry a function. Then we can later systematically explore *combinations* of solutions that may lead to promising designs. This is known as concept generation and evaluation.

34.1.2. Building the Thing Right

Having understood what to build we also need to ensure we built it correctly. To do this we need to understand the *requirements* involved in building the interactive system. There are many variants of requirements, such as user requirements, business requirements and technical requirements, and it helps greatly to have an accurate understanding of all of the relevant requirements at an early stage of the design process. The tasks of eliciting and managing requirements throughout a design process is known as *requirements engineering*.

Requirements are important because they tell us how to “build the thing right”—a product or service that meets its requirements specification will pass verification. Therefore managing requirements and verification is in practice a tightly interwoven process.

Since requirements are so important to successfully arriving at a successful interactive system, they also carry a lot of risk. It is very easy to fail to capture accurate requirements. This can for example happen if there is a process error when eliciting requirements from users—leading to an incorrect understanding of users’ true needs and wants. For example, one common source of error is undetected bias, which can easily happen using for instance a survey if only respondents representing a particular segment of the target audience responds (for example, people who do not have to work during the day, such as retired people). Another failure that can have disastrous consequences is failing to set up a necessary requirement to begin with, which means it then has to be introduced late in the design process. Managing changes in requirements throughout the design process is known as *change management* and it is in practice a very important aspect of project management. As a general rule, the later a requirement has to be changed the more expensive and difficult it will be to implement that change. Therefore, requirements engineering is a very important aspect of project management.

Principles of software engineering are used to implement systems that fulfill requirements. They inform the design of software architectures that can help manage the high complexity of interactive systems. A key aspect in software architecture is achieving *abstraction*, which means hiding implementation details at a lower layer to enable designers to focus on essential design tasks.

34.1.3. Verification and Validation

Having built a system we need to ensure it works as intended and solves the problems users were having to begin with. The former is known as *verification* and it is process of systematic checking that the system meets all requirements in its requirements specification. The latter is known as *validation* and it is the process of ensuring that the deployed system fulfills its purpose for users.

Verification is by its very nature tightly coupled to the management of requirements as only testable requirements can be verified in the first place. Further, successful verification of a requirement demands a clear specification of a verification procedure, verification environment and success criteria, which all rely on the context giving rise to the requirement in the first place.

Successful verification means that a system fulfilling all its requirements as specified in the requirements specification. However, a system succeeding in verification does not mean the system is fit for purpose. This is because there may be errors in the requirements specification. In practice, inaccurate or missing requirements is common. Further, circumstances may have changed or new factors may have been introduced which were unknown when the requirements were evolved. For this reason it is also critical to validate a system, which means ensuring that the system is fit for purpose in that it achieves its overall intended function in actual use contexts.

While HCI is to a large extent concerned with *evaluation* (Part VIII), verification and validation in engineering brings their own perspectives on how to ensure an interactive system is built correctly and fulfills its purpose.

34.1.4. Systems Thinking

Another perspective engineering brings to HCI is *systems thinking* (Chapter 35). User interfaces are interactive systems that are both embedded within other systems and related to other systems. They can also consist of subsystems that must work together to form a coherent whole.

Further, users and other relevant stakeholders are often relevant parts of the wider systems that interactive systems find themselves embedded within. For example, a pen injector for injecting a medical drug may have to interact with an app calculating the correct dosage. This system may then be further embedded within a complex healthcare system involving a nurse administrating the dose and a patient receiving the treatment. Such *joint* systems that involve both users and user interfaces necessitates for designers to have a deeper understanding of the systems that an interactive system needs to be used within.

Engineering provides a toolbox to assist with such an understanding of systems in the form of *system mapping* methods that allow designers to create elaborate maps of how systems are composed and how, for example, people relate to each other and how information flows within an organization.

Systems thinking requires some similar skills design practice (item VI). The sidebox describes a case.

Paper Example 34.1.1 : Engineering requires getting the problem right

Engineers need not only to find best systems-level solutions but also agree on what they are designing for. Bucciarelli and Bucciarelli [107] describe studies of practicing engineers. The authors were motivated by the observation that engineering is often presented as an instrumental process and not as a situated social process where much work is needed to negotiate and define projects with others. The book *Designing Engineers* describes three projects: an x-ray inspection system for airports, a photoprint machine, and a a residential photovoltaic energy system. What sets their studies apart is that they observed engineers in these projects in their meetings and at their desks. The author concludes that engineering design is a social process. It involves multiple parties beyond engineers: marketing people, researchers, accountants, and customers. A large and critical part across all three projects was defining what it is to be done.

34.1.5. Understanding risk and keeping people safe

We increasingly rely on interactive systems for critical tasks where the consequences can be extremely serious if users are unable to reliably carry out their tasks. Engineering provides methods for understanding and minimizing human error and systematically analyzing systems to ensure their safety.

A related aspect is *risk.* which is when the system gives rise to undesired behavior. Such behavior is called a *hazard.* The exposure of a hazard is the degree to which a hazard is able to result in undesirable consequences. The impact of a hazard is a measurement of how serious the consequences are if the system is impacted by the hazard. Risk is the combination of exposure and impact. For something to be very high risk there needs to be both a high exposure to hazard and a high impact of the hazard. Some hazards have very serious consequences, including the death of a user. However, if the exposure is very low the risk may still be low.

Risks are ubiquitous in societies and the wider systems they work within and since user interfaces are embedded within these systems they also need to consider risk. Engineering provides a rich set of *risk management* methods for assessing, mitigating, monitoring and managing other aspects of risks.

Paper Example 34.1.2 : Designing better warnings to keep web users safe

Phishing is a scamming technique used to collect user's personal information by exposing users to fake websites that mimic legitimate ones. To keep users safe websites try to warn users about potential phishing websites. Egelman et al. [216] examined the effectiveness of such warnings in a laboratory study. They simulated a phishing attack and found that 97 % of their participants (N=60) fell to at least one of the attacks they used in the study. They noted that active warnings were more effective than passive warnings. In active warning, the user is required to read the warning and click an acknowledgement of the threat in order to continue. In a passive warning, a recommendation is provided but there is no need to acknowledge it – it can be simply dismissed.

The authors discuss this finding in terms of a model of warnings by Wogalter [888]. The model has consists of a communication flow between a source (delivering warning), a channel, and a receiver. However, the receiver is receiving the warning with other stimulation (distractions) that may distract the processing of the warning. Egelman et al. [216] proposed seven questions to help design safer, more effective warnings:

1. Do users notice the indicator that a warning has appeared?
2. Do users know what the indicators mean?
3. Do users know what they are supposed to do with the indicators?
4. Do they believe the indicators to be correct?
5. Are they motivated to do the correct actions?
6. Will they actually performe those actions?
7. Will the indicators interact with other environental stimuli?

This model can be used as a walkthrough to design better warnings.

34.1.6. Managing the Process

An interactive system is a complex system by its very nature as it relies on user interaction and user behavior can rarely be precisely or accurately modeled. This complexity is exacerbated by interactive systems tending to increase in complexity as software and hardware become more sophisticated and allow designers to incorporate further functions in response to users' needs and wants. In addition, interactive systems are increasingly becoming embedded in wider systems, such as manufacturing, smart home, education, government and healthcare systems, which necessitates understanding the wider implications of embedding an interactive system within a wider system.

To handle such design it is useful to systematically managing the process by using, or at least be aware of, design engineering approaches which have been conceived to allow diverse design teams to systematically explore conceptual, embodied and detailed designs all the way to deployment while managing requirements, risks and other considerations, such as safety and usability.

34.1.7. Systematic Approaches to HCI Problems

Finally, engineering allows us to view certain HCI design problems as that can be attacked computationally. We can represent aspects of interaction formally consequently apply a corresponding systematic approach, such as solving an equation, systematically searching for an approximate or exact solution, or simulating system behavior.

Computational methods in HCI model an aspect of HCI as some formalism, such as a regression model or neural network, and then reason about that aspect, for example making inferences about users, predicting their behavior, or finding designs that best suit them. Computational methods have been used in HCI to, for example, to optimize input methods and ensure that systems used in healthcare settings can be used safely.

One powerful way engineering allows us to improve HCI design is through optimization. Certain HCI design problems, such as menu structures, the organization of keys on keyboards and haptic design parameters can be systematically optimized to achieve as good user performance as possible - within the assumptions that have been made. Various ways to optimize a design have been proposed in the wider engineering literature and many of these approaches are useful for HCI design problems too. An awareness of such methods may allow for better designs than what may be possible to arrive at by traditional design iteration.

Computational approaches also allow us building systems that can infer or predict users' intentions. Such systems rely on a pattern recognition and machine learning to achieve their tasks. Such *inference* brings its own challenges in terms of ensuring such interactive systems have a high performance, are resilient to noise and changing use contexts, have access to representative training data and allow humans to interpret their machine learning models and machine learning results. Engineering provides methods for systematically tackling such challenges, although several of these methods are currently a work-in-progress and subject to active research.

In summary, effective, efficient, and safe interactive systems that satisfy users' needs and wants benefit from designers aware of systems thinking and the use of computational principles that allow them to articulate a detailed rationale behind a particular implementation and solve it with the help of algorithms. These allow designers to both build the right thing and build the thing right.

Summary

- Engineering is about using systematic principles and methods to build systems.
- Systems thinking is paramount when building interactive systems. All interactive systems are systems that are reliant upon other systems and embedded within several other systems.

- To build systems we need to ensure we build the right thing and that we build the thing right. Design engineering is a design process that is focusing on translating a solution-neutral problem statement into a deployed system that can be verified and validated.

- Safety and risk are central notions when engineering interactive systems. Risk is pervasive and it is as a consequence impossible to design a risk-free system. Risk therefore has to be assessed and managed using systematic approaches.

- Engineering methods and principles and techniques from software development allow us to systematically build interactive systems. Examples of such approaches include engineering models, optimization, formal methods, pattern recognition and machine learning, and software design and software architecture principles.

35. Systems

Systems are pervasive in our lives. When you open a door, pull down the handle and pull the door open you are interacting with a system. The door handle is a system consisting of fittings, screws, a plate, a handle, and it is interacting with another system—the door itself. The door in turn, is interacting with the wall it is mounted on using hinges, and the door frame itself might even be load bearing. Further, the user opening the door is attempting to reach a goal state, to that of the door being open, and is doing so by planned movements executed by the user's human motor control system. The action of the door being opened is monitored by the user's visual and auditory systems. At the cellular level, neurons and synapses form complex systems carrying signals through the human body to enable perception and action. Even the very screw is part of an intricate design, manufacturing, and logistics system, which includes designing the screw, efficiently producing it, packaging it, and distributing it. Finally, doors are subject to regulation, in particular building control regulation, which will specify, among other things, acceptable dimensions, such as the door width for accessibility.

This exercise can be repeated for just about any "trivial" everyday interaction. The very fact you are alive when reading these words is a testament to a vast array of systems within your body functioning. Your need for shelter, food, electricity, and so on, is reliant on several complex systems, including a political system, an economic system, a food production system, a transportation system, and an electricity grid, among many others. Systems are truly pervasive and our lives depend on them.

We learn a lot from such a simple exercise. First, *systems are complex*—they have many dependencies with other systems and entities. A complete understanding of a system is therefore likely to be challenging, for example, we still do not fully understand how the human brain works. A successful HCI system is likely to be complex in its design and implementation. However, it is also required to interact with other complex systems, including humans, and other technological systems, such as databases, sensors, and machine learning systems.

Second, *systems operate at multiple levels*, ranging from a very low level, such as a nanoscale motor protein, to a very high level, such as the exascale star cluster Omega Centauri. In HCI, we frequently need to work with systems at multiple levels, as a computer is an elegant example of systems working at multiple levels in both its software and hardware organization, ranging from NMOS and PMOS transistors giving rise to CMOS chips at the hardware level, to various software libraries and toolkits providing support for drawing primitive graphics, managing user interface elements, and so on. An app on a mobile phone frequently needs to interface with both the graphical user interface toolkit and the many sensor systems available on the phone, such as the camera, gyro, accelerometer, and so on.

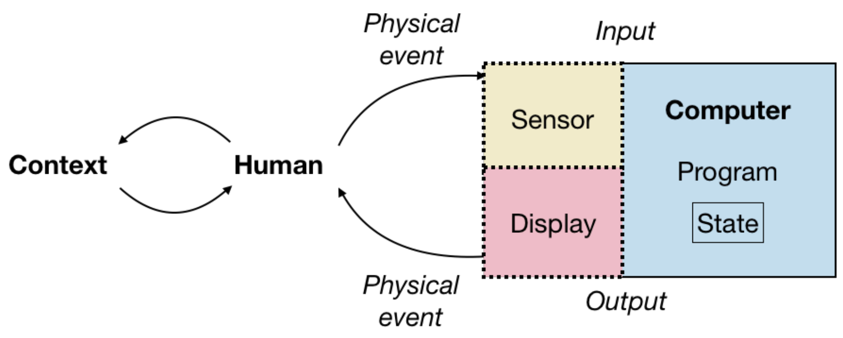


Figure 35.1.: A diagram of a simple interactive system.

Third, *systems are coupled*, that is, they depend and rely on each other in various ways. For example, the movement of a mouse around on a desk is reflected in the movement of a pointer on a display. This is a form of coupling. However, the coupling is mediated via software that allows manipulation of the ratio of pointer movement in relation to mouse movement. Manipulating this ratio allows features such as mouse acceleration. In general, a computer decouples hardware from applications by the use of an operating system. The more tightly coupled systems are, the more difficult it is to make changes to one system without affecting the other.

Fourth, *systems have emergent properties*. Systems give rise to effects that are more than the sum of their parts. The parts, modules, functions, and subsystems within a system all work together to generate properties, qualities, and outputs that the individual parts cannot provide unless they are arranged within the system.

So how do we think about systems in HCI? Let us consider an *interactive system*, which is a central concept in HCI. It is system that is able to (1) receive and respond to input events via its sensor; (2) perform computations; (3) maintain and update its state; and (4) display its output. Figure 35.1 shows an overview of a simple interactive system. This definition exposes challenges that characterize interactive systems engineering.

From social media applications to intelligent keyboards on mobiles, we use interactive systems all the time. However, Figure 35.1 makes it already clear that this interactive system is a coupled system connecting a user with a computer. However, it is also embedded within a wider system providing a context for interaction. Further, the technology within the interactive system is reliant on several subsystems, including sensor and display systems, as well as means of performing computations.

As the simple diagram in Figure 35.1 makes clear, even a rudimentary interactive systems is reliant on some understanding of several systems.

However, HCI research frequently needs to consider even wider systems, such as the following three examples.

- O'Hara et al. [599] reports on how to use depth sensors to enable surgeons to interact with image data in a sterile environments using gestures. The research problem itself is due to prior research on understanding the systems surgeons are operating with and within. A key takeaway is that for such a system to be successful it is has to go beyond the idea of touchless interaction in sterile environments and consider the sociotechnical aspects that affect surgeons interacting with technology. This includes taking into account the collaborative aspects of surgery, the fact surgeons are not always in front of a display when they desire data, the risk gestures are mistakenly recognized when a surgeon is merely moving their hands, and so on.

- Su and Liu [782] introduces a mobile application for nurses that enables the hospital to move away from paper forms and at the same time provide nurses with additional functionality. The work involves understanding existing nursing processes and forms, needs and wants from nurses and the hospital, and the type and flow of information.
- Pu et al. [662] introduces a gesture recognition system for an entire home by analyzing wireless signals, such as Wi-Fi. Wireless signals do not suffer from line-of-sight problems, such as walls blocking infrared light, and therefore can be potentially used to sense gestures regardless of where the user is in the home. The technique detects the Doppler shift that is induced by a user moving closer and further away from a transmitter. This information is then fed into a gesture recognizer. To avoid false activations, the user carries out a distinct "preamble" gesture to signal the start of a gesture. This is an example of an interactive system providing benefits for a wider system—the house—by exploiting an existing subsystem, the house's Wi-Fi network.

The rest of this chapter is dedicated to this purpose: understanding systems. The next section will begin by introducing systems thinking, a holistic approach to thinking about systems. Systems thinking does not inform on how to build a system. Instead, systems thinking about how to holistically consider the important aspects of systems. As such, systems thinking is useful to keep in mind but we also need means of understanding systems. This chapter will explain how system mapping can be used to describe a system in order to reveal how, for example, users, tasks, information and processes link together to form a working system. Such understanding can then be used to carry out design activities (Chapter 33) or to carry out risk assessments (Chapter 37).

35.1. Systems thinking

Engineering is the practice of using systematic principles to design and construct machines, structures and other systems, including interactive systems.

We live in a complex world where we are immersed in and surrounded by systems which we often take for granted. For example, people typically live in societies and such societies are arranged in various governing structures, such as local governments and national governments. In addition, international agreements and frameworks across countries govern many defense, trade, finance and climate change concerns. Societies are in turn embedded on planet earth and interface with a variety of natural systems, such as our ecosystems. In addition, modern civilization is supported by a range of man-made systems governing production and delivery of food, energy and healthcare as well systems for controlling banking, insurance, finance, manufacturing and defense and many other vital aspects. A typical house is a complex system relying on subsystems such as electrical wiring, plumbing and telecommunications to allow people to live in homes that are heated and provide us with the everyday facilitates we take for granted, such as the ability to clean our clothes using a washing machine, heat food using a microwave oven or access the Internet using a laptop or mobile phone. Every human and animal residing in such a house is a system consisting of for example the musculoskeletal, circulatory, respiratory, digestive, reproductive, and neural subsystems.

To understand a system, we often first identify its subsystems. For example, the electrical wiring in a house is reliant on protective systems, such as fuses and residual current devices (RCDs) to protect against faults. Fuses and RCDs are systems that are designed to interface with electrical systems. A typical touchscreen mobile phone consists of several subsystems such as the screen, the logic board, the central processing unit and memory circuits. The phone software is driven by an operating system managing drivers, processes, security and ensuring apps can run and access all required functionality. Apps rely on software libraries to display user interface elements and to receive notifications of the user's actions.

As is evident in the above description, systems are pervasive in our environments—and they are complex and difficult to understand, design, build and maintain. Systems thinking [218] is an approach to allow designers to reason about systems. The terminology arose in engineering as a result of a recognized need to approach system design as a holistic cross-disciplinary team activity that considers system design across the entire life cycle of the system. This including the system's design, integration, management, maintenance and eventual disposal. It also includes considerations of the system's role in its operating environment, which may in turn be a larger system or necessary interaction between several systems or subsystems.

A systems thinking perspective is helpful in HCI from several perspectives. First, many HCI artefacts provide critical capability by being embedded within a larger system. For example, an interactive wearable patient monitoring device is embedded within a hospital environment, which is a system with flows of patients, doctors, medication, equipment, and information, as well as organizational hierarchies of departments and worker roles. In addition, activities in a hospital has to follow both its internal rules and professional and government regulation. For a wearable patient monitoring device embedded in such an environment to be successful, it is critical the designers are aware of this complex operational system that the device must operate within.

Second, HCI artefacts are systems in their own right. For example, a mobile phone app is a system fulfilling a certain purpose by exposing the user to necessary functionality. As HCI progresses, many HCI artefacts find themselves relying on hardware, such as sensors, or necessitate developing novel hardware input or output solutions. The HCI system is now a combined hardware and software system that requires expertise across the domains of electrical engineering, computer science, design, and psychology.

A *system* is here defined as a set of parts or modules that in combination provide emergent qualities that are not present in the individual parts or modules themselves. Synthetic systems are designed by people and the design of such systems has an objective:

to provide capability. It is the desire for this capability that leads to the creation of a system in the first place. Users rarely care about the creation of a system for its own sake. Instead system emerge out of needs that have to be fulfilled by creating systems that are able to provide the desired capability.

A simple course hierarchy of system complexity can be decomposed into three levels (adapted from Elliott and Deasley [218]):

Level 1 A subsystem that is managed primarily within one technical discipline and organization. Examples include a logic board, an app and a wearable health monitor device.

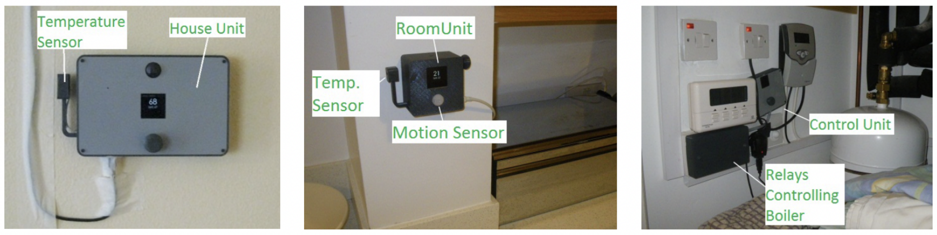
Level 2 A system that either involves two or more technical disciplines or two or more organizations. The design of a car's core functionality is one example as it primarily involves mechanical and electrical engineering. A mobile phone and its operating system is another example involving electrical engineering and computer science. Another example is a national electronic medical records management software that is designed to work with a variety of hospitals and other health providers.

Level 3 A system that involves many technical disciplines and is impacted by wide-reaching social, economic, political or environmental factors. Typical examples are major infrastructure, such power grids, air-traffic control, and military command and control.

An individual HCI system is situated within one level of the above hierarchy but is nearly always set to operate within another system, which may be at a different level in the hierarchy. For example, an electronic voting user interface is embedded within at least two systems: first, the voting machine design itself, which consists of electronic components providing the user with means of inputting information and receiving confirmation; Second, the voting machine is providing a critical function for society and as a consequence is operating in a highly regulative and political system. Successful HCI design necessitates being sensitive to this wider system, including understanding the target audience, for example, users with disabilities, and the fact that the voting machine is embedded within a political system [58].

Paper Example 35.1.1 : Heating homes more efficiently by sensing and predicting room usage

PreHeat [729] is a system for more efficiently heating homes by sensing and predicting people's room usage. It involves several subsystems, such as motion sensors, temperature units, control units, and it has to interface with the existing heating system in the house. The figure below shows some of these systems in one installation.



Preheat then using the sensor data to estimate people's room usage needs and uses a prediction algorithm to estimate when an individual room should be heated up.

PreHeat was deployed to five homes and was shown to heat the homes more efficiently without users having to program thermostat schedules.

There are several accounts of systems thinking and no de-facto process to adopt. However, a simple way to adopt systems thinking is to adopt the following six principles [218]:

Debate, define, revise and pursue the purpose Refine the problem and identify the three parameters cost, performance and timescale involved when designing the system. Finally, do not neglect the fourth parameter: risk, which needs to be understood for each of the three prior parameters. No system is optimal which implies there are trade-offs in the system. These trade-offs must be understood. Finally, requirements constrain the design space and make system designs tractable. However, for any non-trivial system it is nearly impossible to initially arrive at the correct set of requirements. Requirement evolve with system design.

Think holistic Systems have boundaries as without boundaries we do not have definitions of systems. Systems are embedded within other systems. Integrated systems need to consider the system as a whole, taking into account all components of the system, across the lifespan of the system. The components of the system and the environment in which the system operate are necessary to understand. In addition, the processes, tools, and people required for design, build, deployment, maintenance, and support are all part of the system.

Follow a systematic procedure Systems are planned, designed, and built. It is necessary to identify and manage uncertainty throughout a project and use a robust design process that allow iteration, working with stakeholders, and the management of risk, changes, and rework.

Be creative Use both innovative and conventional thinking to understand together with stakeholders what the system must achieve, to create the system architecture, and to help guide every stage throughout the life of the system, ranging from the design and build stages, to the support and dismantle stages.

Take account of the people People are part of systems and they are critical for the success of systems. People build, install, use, and support systems. They may also have to defend, challenge, or tolerate systems. People's motivation, competence, attitude and capability to deliver quality matter for the success of a system. Ergonomics, ethics, and trust, among other factors, are also critical.

Manage the project and the relationships Complex systems involve complex projects to build them. They require many stakeholders with many roles, scattered across different organizations. Just like systems are designed, projects to build, deploy, maintain, and support systems also need to be designed to take all relevant factors into consideration.

A weakness with systems thinking is that while it prescribes an approach it does not specify specific methods or procedures that allow systems thinking. However, nonetheless a lack of systems thinking can often be linked to failures with systems. For example, an analysis of 12 problems in systems linked them to four categories of system thinking failures [541]:

- A failure to consider the environment in which the system operated. For example, wind conditions.
- A failure to understand that non-technical factors, such as organizational, political, economic, or environmental factors, were necessary to understand and take into account in order to solve the system problem.
- A failure to correctly address planned and unplanned interactions between components within the system and interactions with system's environment.
- A failure to take into account that many products are part of a wider user experience system and that the product can therefore only thrive if such a user experience system exists and provides adequate services.

35.2. System mapping

To understand a system we will need to describe it. *System mapping* refers to a set of techniques for achieving this. System mapping allows us to describe a system in terms of its processes, people, and flow of information.

The first step in system mapping is to establish the *system boundary*. Such boundary setting clarifies the scope of the system. Determining a system boundary is particularly important for risk assessment and we discuss this step in more detail in the chapter on safety and risk (Chapter 37).

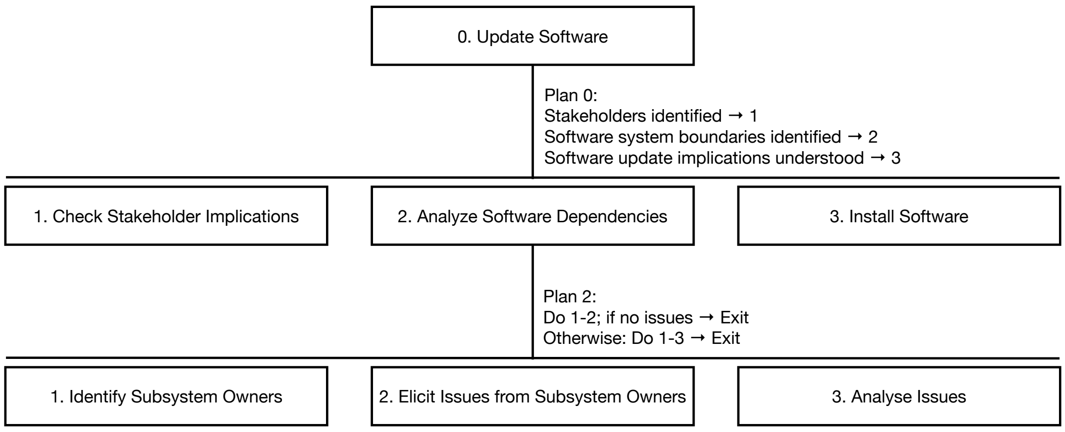

0. Update Software
Plan 0:
Stakeholders identified → 1
Software system boundaries identified → 2
Software update implications understood → 3
1. Check Stakeholder Implications
2. Analyze Software Dependencies
3. Install Software
Plan 2:
Do 1-2; if no issues → Exit
Otherwise: Do 1-3 → Exit
1. Identify Subsystem Owners
2. Elicit Issues from Subsystem Owners
3. Analyse Issues

Figure 35.2.: An example of a task diagram for updating software in an organization.

The purpose of system mapping is to ensure there is a clear understanding of the structure and behavior of the system. System mapping is carried by describing the structure and behavior of the system. Any technique that is suitable for this purpose can in principle be used. Here we review set of common system mapping techniques.

35.2.1. Task diagram

A task diagram is a hierarchical representation of tasks and the necessary conditions for carrying out the tasks. Diagrammatically a task diagram represents tasks as nodes and relationships between the tasks are depicted as links. Since task diagrams are hierarchical they can be as detailed as required. They are often used to describe complete processes, organization of work, and user interface workflows.

Task diagrams focus on processes and procedures, user behavior, and human-computer interaction. Tasks diagrams are frequently used to describe HCI systems but can also be used to explain a wider system that an HCI system is finding itself embedded within. For example, a medical app used by a nurse may be embedded within a hospital environment, which prescribes a set of processes for the nurse to carry out a task. Task diagrams allow such processes, which may involve steps *outside* the app, to be mapped out and understood by the design team.

Figure 35.2 shows an example of a task diagram for an organization that needs to update software. Only some tasks are shown in the figure. Each task can be augmented with a plan that elaborates on the steps in each task and how tasks relate to each other.

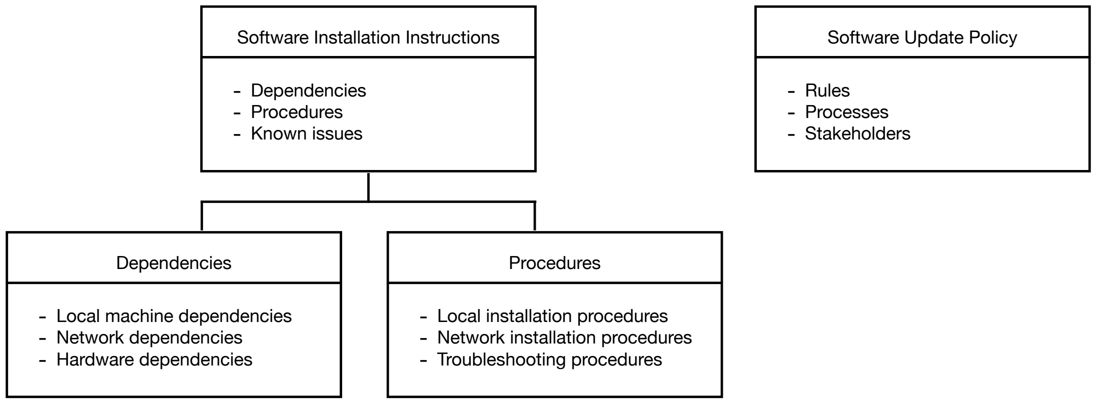


Figure 35.3.: An example of an information diagram showing a part of the hierarchy of information that a system administrator would require to do a software update in an organization.

35.2.2. Information diagram

An information diagram is a hierarchical representation of documentation. An information diagram represents documents as nodes and the relationship between documents as links.

Information diagrams are used to gain an understanding of the documentation structures within an organization, such as the degree of standardization of documents and the way documents relate to each other. As documentation can be both electronic and on paper, information diagrams can capture both types of documents and link their dependencies and usages.

Figure 35.3 shows an example of an information diagram for an organization that needs to update software. Only a part of the information hierarchy is shown in the figure. The software update policy, for example, can be further decomposed into information about rules and processes around software updating in the organization. The stakeholders involved might be more usefully mapped using another diagram, such as an organization diagram, which we describe next.

35.2.3. Organizational diagram

An organizational diagram is a hierarchical representation of people and their roles in an organization. They represent teams, individuals, departments, etc. as nodes and their relationships as links.

Organizational diagrams make it possible to identify users and their roles, and in particular they allow the identification of stakeholders in an organization or wider system. This can guide further system mapping and data collection by probing further investigation into stakeholders in the system that might at first appearance not seem central to the problems considered.

Figure 35.4 shows an example of an organizational diagram for an organization. In the diagram it is clear that both system administrators, part of the IT office, and engineers,

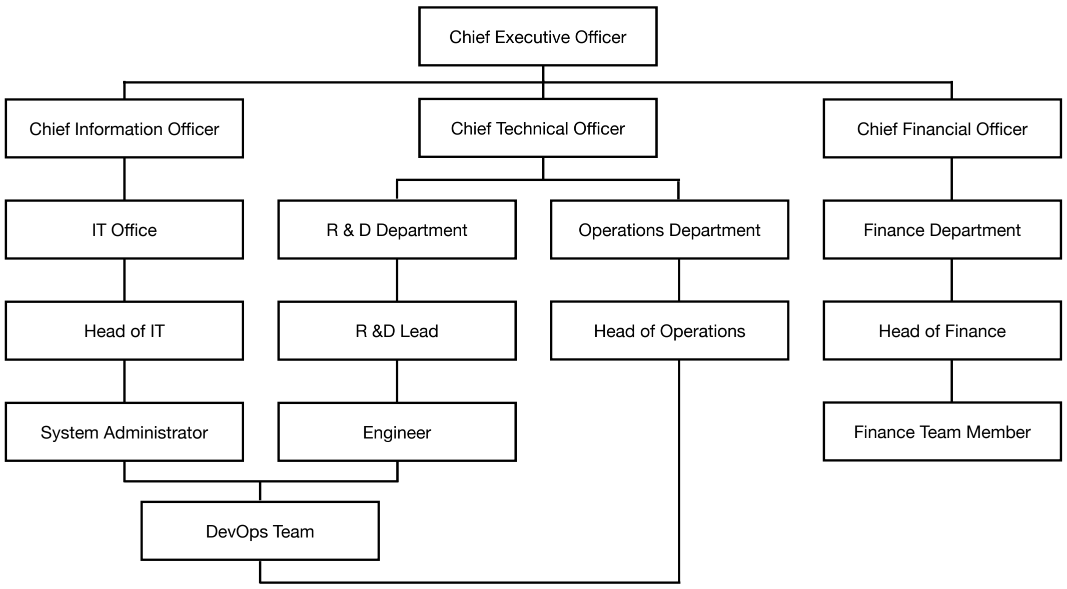


Figure 35.4.: An example of an organizational diagram for an organization subject to a software update. DevOps stands for development operations. The DevOps team combines the roles of software development and IT operations as one function.

part of the research and development department, form a cross-disciplinary team managed by the head of operations.

35.2.4. System diagram

A system diagram is used to understand how data is is transformed through processes and activities in the system. System diagrams show where such is stored and how such data activities and processes are sequenced to allow such data transformations.

System diagrams allow mapping out processes, in particular how users interact with systems in order to achieve tasks. They show data-driven processes and activities. As such, system diagrams allow analysis of data flows, functions and states in a system. For example, system diagrams can be used to identify possible risks in a people-centric process by allowing designers to analyze possible failure modes in such a process.

System diagram consists of two parts. First, activities that indicate the flow of data between activities. Second, states and state transitions that indicate the state conditions for a transition and the actions arising from a transition.

Figure 35.5 shows an example of a system diagram for an organization. The top part shows activities as circles that a system administrator ('sysadmin') has to engage with in order to update software. Generated data that gives rise to data flow are indicated below the activities. The flow of data is indicated by arrows. The bottom part shows the states and state transitions in this process. Transition conditions, such as 'Corrections required', are indicated next to each arrow (transition). Transition actions, when applicable, are also indicated next to each arrow, preceded with a dash ('-') to disambiguate transition conditions and actions.

35.2.5. Process diagram

A process diagram shows how serial and parallel processes and activities are structured as a series of steps. A very common instance of a process diagram is a flow chart, though other forms of process diagrams also exist. The nodes in process diagrams represent the steps in a process and the links represent conditions from transitioning from one step to another. Process diagrams show the ordering of steps within activities, if such activities serial.

As alluded to, a flow chart is a form of process diagram. It is possible to annotate such flow charts to include additional information, for example, by linking flow charts to organizational diagrams in order to make it clear which stakeholders are involved in an activity. Process diagrams can thereby be a basis to understand an overall process in a system, linking in, for example, relevant stakeholders, documents and tasks.

Figure 35.6 shows an example of a process diagram in the form of a swim lane diagram. The lanes, indicated with dash lines, delineate roles in the process, such as sysadmin, subsystem owner, engineer, and so on. The flow chart begins and ends with rounded rectangles indicating the start and end of the process. The thick horizontal bars indicate forks and joins of activities. The rectangles indicate activities and the diamonds indicate decisions. Arrows show process flow.

35.2.6. Communication diagram

Users that interact through a common process share information. A communication diagram is a way to represent such flow of information between users. The nodes represent users, or an entity representing a user group, and the links represent the flow of information.

Communication diagrams are used to show the flow of information between people within the same teams, different teams, or even different organizations. Communication diagrams can also be used to depict flows of information across different entities, such as different departments in a company.

Figure 35.7 shows an example of a communication diagram that indicates the flow of information between the system administrator, subsystem owner, engineer, and head of IT that is required for a decision to approve a software update.

35.3. Principles for legal and ethical systems

Systems thinking invites designers to think and analyze systems beyond merely the technology. In particular, systems that are deployed must meet regulatory requirements, that is, they have to be lawful. In addition, systems should not induce adverse effects, whether intentional or accidental, on its users or people otherwise affected by the technology.

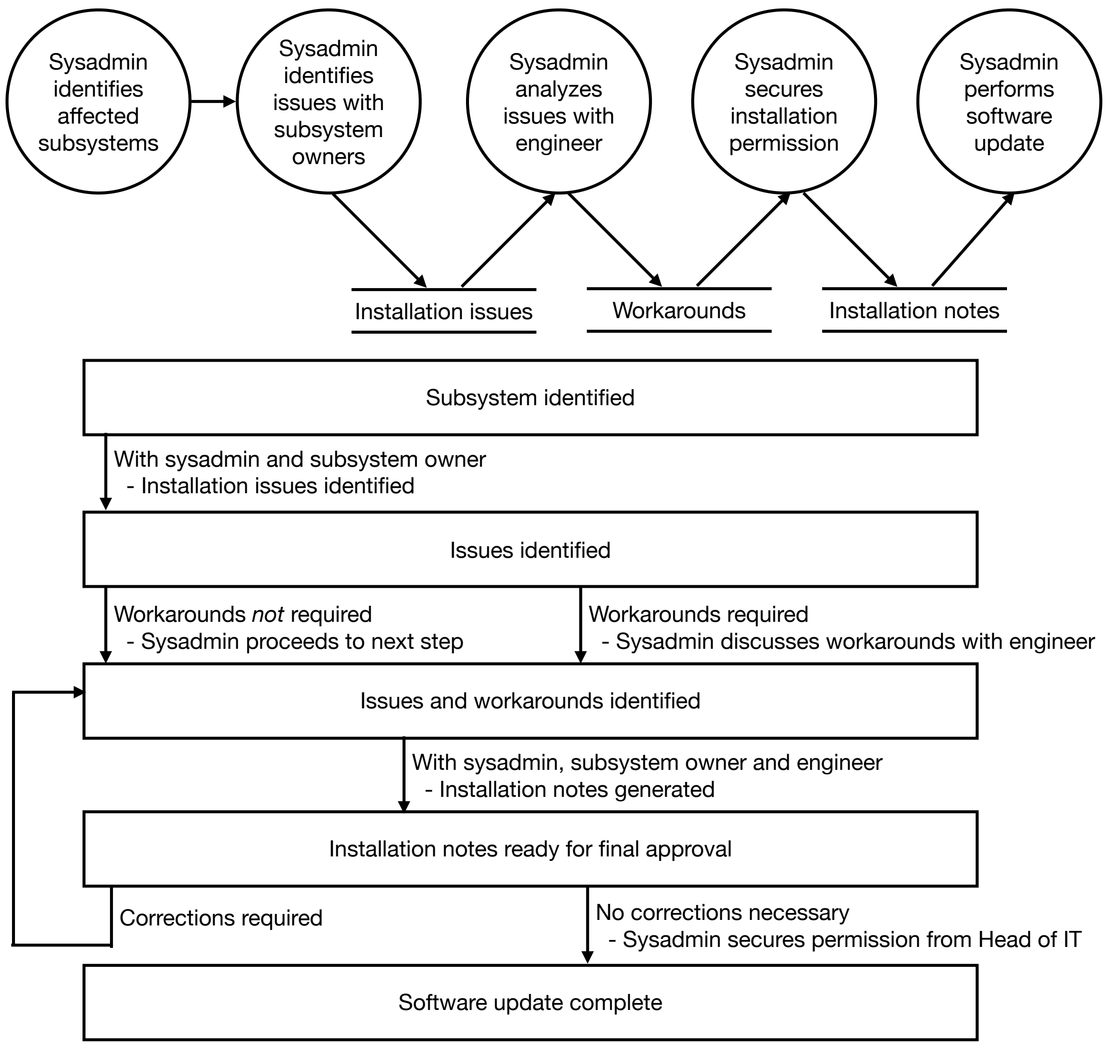

Sysadmin identifies affected subsystems
Sysadmin identifies issues with subsystem owners
Sysadmin analyzes issues with engineer
Sysadmin secures installation permission
Sysadmin performs software update
Installation issues
Workarounds
Installation notes
Subsystem identified
With sysadmin and subsystem owner
- Installation issues identified
Issues identified
Workarounds *not* required
- Sysadmin proceeds to next step
Workarounds required
- Sysadmin discusses workarounds with engineer
Issues and workarounds identified
With sysadmin, subsystem owner and engineer
- Installation notes generated
Installation notes ready for final approval
Corrections required
No corrections necessary
- Sysadmin secures permission from Head of IT
Software update complete

Figure 35.5.: An example of a system diagram for an organization in the process of updating software. The top part shows the flow of data between activities. These activities give rise to data which links some of the activities. The bottom part shows states as boxes and transitions between states as arrows. Conditions for a transition are indicated next to each arrow as a textual description and actions are indicated as a textual description preceded with a dash ('-').

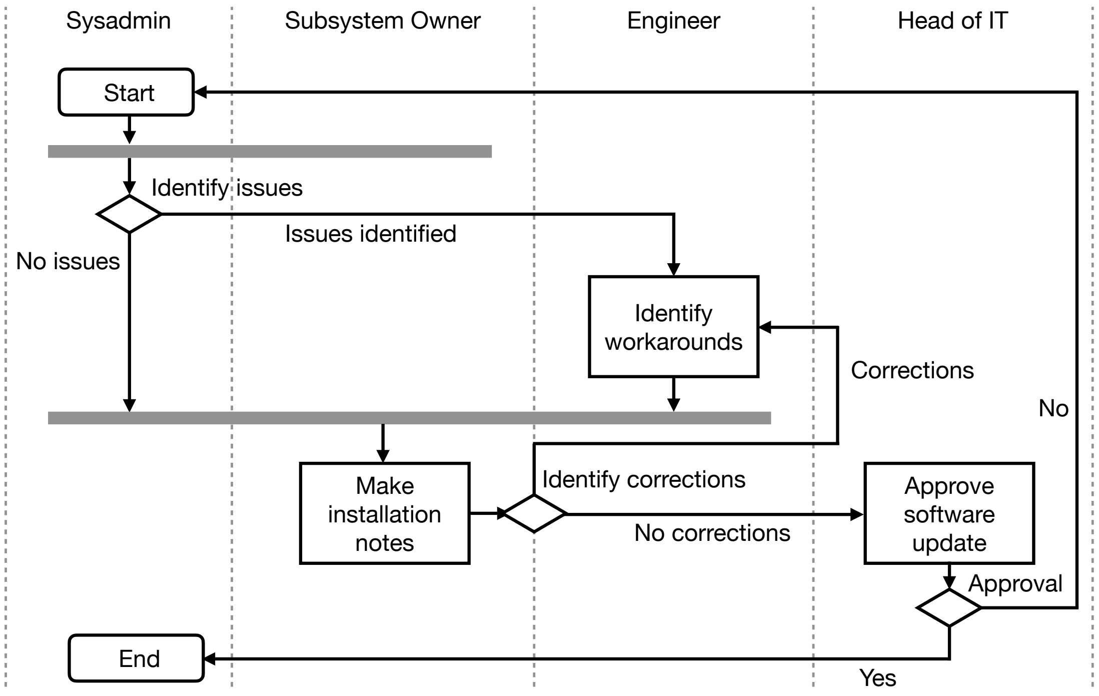


Figure 35.6.: An example of a process diagram for an organization making a decision on whether it is ready to update software.

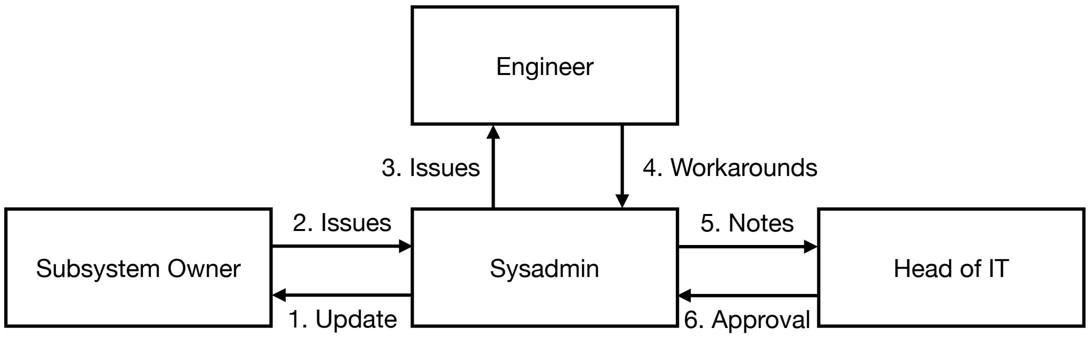


Figure 35.7.: An example of a communication diagram for an organization making a decision on whether it is ready to update software.

There is an important distinction between *legal* and *ethical* concerns. Legal issues refer to systems that are in some ways unlawful. For example, medical devices and consumer electronics are regulated by regional laws. Systems that break such laws can lead to prosecution. Ethical issues are not necessarily referring to particular laws that are broken but rather adverse outcomes of technology that are undesired according to norms. Several communities, such as the Association for Computing Machinery, encodes these norms as codes of ethics and professional conduct.

Systems using artificial intelligence (AI) are the latest frontier of ethical and legal debate, for the simple reason that they touch an increasing number of everyday decisions. Interactive systems relying on AI to carry out parts of their functions affect users and other stakeholders in varying ways with social and financial consequences. Multiple guidelines have been proposed to the purpose of governing and evaluating systems. Fjeld et al. [241] presents a comprehensive summary of such principles, which we elaborate on below, using the eight themes they suggest: privacy, accountability, safety and security, transparency and explainability, fairness and non-discrimination, human control of technology, professional responsibility, and promotion of human values.

Privacy Data should not be used without a person's knowledge. Further, users should have control over the data and an ability to restrict the processing of their data. Users should also have the right to correct data or remove it altogether. In general, privacy concerns should be integrated within the AI design process. There is also wide recognition that privacy issues in AI systems require new regulatory frameworks by governments.

Accountability For systems to be accountable it is important they produce verifiable, replicable results and can be examined in sufficient detail for their underpinning operations and mechanisms to be validated. The impact of systems must also be assessed in so-called impact assessments. Such impact assessments can be related to impacts on human rights but they can also more generally study negative impacts of systems. One example is ecological impact which highlights the need for environmental responsibility. To enable accountability it is also important to build systems that can be audited and that the findings of such audits can be fed back to the system. It may be necessary to create a monitoring body, such as an internal review board, to ensure best practices during design, development and deployment. If a system is automatic and makes decisions involving humans then there need to be an appeal function in place to challenge such decisions. Related, once a system's decision has been rectified, there needs to be a mechanism to remedy the consequences of system decisions. In general, liability and legal responsibility for system actions must be clearly defined. As a consequence, many guides recommend considering new regulation to ensure systems are accountable.

Safety and security Safety means ensuring the internal functions of the system are working as expected and the need to avoid unintended consequences leading to harm. Risk management best practices can be used to address safety. Risk management is discussed in Chapter 37 in this book. Security is the need to address external threats to the system. There is a need to ensure such systems are resilient and a principle known as security-by-design should be applied to ensure abstract ideas about security are translated into working principles in implementations by ensuring there is a link throughout the design, implementation and deployment process. Finally, systems should be predictable to ensure that it is possible to recognize if systems have been compromised.

Transparency and explainability Systems should be transparent, which in this context means they should be designed, implemented and deployed in such a way that it is possible to have oversight of their internal operations. Explainablity concerns the need for systems to be able to generate outputs that are intelligble and comprehensive and thus suitable for evaluation by humans.

This is particularly important for systems with the potential to cause harm, such as the use of AI in healthcare. One way to improve transparency and explainability is to ensure data and algorithms used to drive system decisions are available to the public. Governments also need to be open about their procurement and use of technology. Importantly, people have a right to information about their involvement and interaction with AI, for example, personal data, and mechanisms used to form decisions about individuals. Related, individuals should be notified when decisions are made by an AI and when they interact with an AI. In general, regular reporting can assist in AI systems supporting transparency and explainability by routine logging of, for example, AI decisions and the data and factors leading to such decisions.

Fairness and non-discrimination Bias means a particular option has a disproportionate weight. Algorithmic bias is the phenomenon of bias being introduced in AI systems through various ways, including the particular algorithms used, the training data, or the way AI is used. Therefore it is important to work towards non-discrimination and prevent bias in AI systems. Representative and high-quality data is therefore essential. Another important principle is fairness, the idea that AI systems treat individuals fairly and the design of such systems make efforts to avoid bias and other factors that can potentially lead to unfair outcomes for individuals. A related principle is equity. This is the that people should have the same opportunities and protections by AI systems regardless of whether they are all similarly situated. It is also important to ensure the benefits and positive impacts of AI are inclusive and thus reach as many people as possible. Finally, AI systems should be inclusive in their design and thus involve diverse participation in their various design, development and deployment processes.

Human control of technology People should be able review automated decisions and opt out of automated decisions. Another important aspect is the ability to allow humans to intervene a system.

Professional responsibility The people involved in designing, implementing and deploying systems have responsibilities. One such responsibility is to ensure systems are accurate, that is, they mostly arrive at correct decisions. As it is nearly always impossible to ensure perfect accuracy this principle is sometimes phrased as a goal rather than an absolute requirement. Related, developers need to be careful and thoughtful. This includes considering the long-term impacts and effects of their systems. Another concern is the need to consult the multiple stakeholder groups that may be affected by a system. Finally, systems should be built by people with scientific integrity and guided by established professional values and standards.

Promotion of human values Systems should protect human values ensure humans can flourish. This means access to technology is a critical concern to avoid systems contributing to a growth in inequality. This not only means that the positive impacts are felt relatively evenly across society but also that people have access to education so that they can benefit from it as much as possible. Another factor is how access to technology can provide people with disabilities with more opportunities, for example, by providing automatic closed captions of online videos. A final principle is the idea that systems should be leveraged to benefit society, that is, it should contribute to human well-being, the environment and a more sustainable society.

35.4. Is systems thinking actionable?

Systems thinking sounds nice and a *lack* of systems thinking can be used to explain designs that have failed in practice. However, there is no concrete framework, toolkit or procedure that can be used to ensure a systems thinking is embodied in a design.

The six principles on systems thinking we introduced earlier in this chapter are high-level principles, such as "think holistic" and "follow a systematic procedure". However, precisely *how* do I as a designer know that I am aware of all relevant components, subsystems, extended systems, people, and other concerns so I can set a correct system boundary? Further, even if I as a designer want to follow a "systematic procedure", *which* systematic procedure should I follow? There are many design approaches, techniques, methods, and toolkits. Even for a niche area, such as user interface verification, there are several sophisticated formal tools available and it is not obvious for a designer which tool is the most suitable for a user-centered design process [123]. Further, work is required to provide HCI designers with sufficiently robust systems thinking frameworks that can integrated in user-centered design processes.

In addition to a lack of concrete guidance, an overemphasis on analysis can lead to *analysis paralysis*: a state where decision making is halted, or making very slow progress, due to the fear of making an error or somehow missing out on a superior solution. An overemphasis on systems thinking may induce this state by having designers despair in the face of an incredibly large number of options. It may also give rise to a sensation of hopelessness when designers realize that the systems they are working with are incredibly complex and nobody truly understands them. Finally, it may result in the design team spending too much time trying to define a complete and correct requirements specification and thereby never reaching a state where they can iterate implementation and evaluation. A fallacy is to assume there is a perfect system when in actuality a perfect system is exceedingly unlikely to exist due to various tradeoffs that are intrinsic to the task the system is trying to achieve.

In practice, design needs to balance overall rigor against the time and cost required to arrive at a working system to begin with. This tradeoff can be readily seen in the small uptake of formal methods, such as system verification, except for safety critical user interfaces where potential errors can be catastrophic.

Summary

- Systems are pervasive and interactive systems are typically embedded within and have to interact with other systems.
- Systems thinking is an approach for reasoning about systems, including how they are embedded within other systems, how they work over time, and how subsystems and other parts of a system contribute to the wider system.
- System mapping is a set of techniques for describing systems in terms of their processes, people, and flow of information, among other things. Several system mapping techniques have been developed to tease out particular aspects of systems, such as task and organization diagrams.
- Principles can assist designers in ensuring HCI systems are ethical and legal.

Exercises

1. System as a concept. Consider the following systems: (1) a fall-detector device for elderly patients in a hospital; (2) a streaming music service app; (3) a virtual reality head-mounted display; and (4) a spreadsheet used by a professor working in a university to record exam marks and grades. Which systems are they embedded within and which systems do they have to interact with?
2. Systems thinking and system failure. Read up on the initial motivation and eventual fate of the following products: (1) the 3DO Interactive Multiplayer; (2) the Apple Newton; and (3) Microsoft Zune. Can systems thinking explain why these products failed? Which of the six systems thinking principles could have possibly been useful to avoid these failures?
3. System mapping. You are tasked with developing an app that assists seven-year-old children to learn multiplication tables. The app is to run on a tablet and to provide a scoreboard so can children can compare their results (via anonymous aliases). Teachers and parents are required access to the results to monitor the progress of individual children. Use system mapping techniques to describe the system.
4. AI ethics. Use the legal and ethical principles of systems to explore how they would apply to the following AI systems: (1) a sentence generating function that

allows nonspeaking individuals with motor disabilities to communicate faster by allowing such users to directly select entire sentences when they type; (2) a magnetic resonance imaging (MRI) scanner interface that takes the output from an MRI machine with a patient inside it in a hospital, sends the data to an offsite location, uses AI to perform image enhancements, and thereafter sends the AI-processed data back to the doctor in the hospital for analysis; and (3) an automated dialogue agent for handling customer complaints in a bank.

5. Safety engineering. A design team is is creating a helmet that enables rapid triage for people in the vicinity of blast explosions in a war zone. The idea is that the helmet has sensors that can sense the risk of brain injury due to an explosion and communicate this information using a visual display. (a) Derive a solution-neutral problem statement for this problem (see item VII). (b) Suggest a suitable sensor for the helmet. (c) Propose a system boundary. Briefly motivate your choice of boundary. (d) Draw a function structure diagram for the key functions in this design. (e) Create a morphological chart for the key functions in the function structure model identified in (d). (f) Generate a suitable concept through concept evaluation and provide a brief narrative motivating the chosen concept. Provide justifications for all criteria, weightings, and scores. (g) Carry out an FMEA within the system boundary identified in (c) (for FMEA, see Chapter 37).

6. A patient-controlled analgesia (PCA) pump enables a patient to self-administer pain relief medicine. Such a PCA pump is set up by a nurse, indicating medication dosage and schedule among other things, and subsequently used by the patient. A support engineer is responsible for ensuring the software is up to date and that the settings are cleared between each patient. Research has shown that a frequent source of patient injury or death is due to the user interface being difficult to use, resulting in the United States Food and Drug Administration (FDA) to specify a set of requirements that a user interface for a PCA pump must comply with. A design team is tasked to create a user interface for a PCA pump. (a) Derive a solution-neutral problem statement for this design problem. (b) Identify a system boundary for this design problem and motivate the choice of boundary.

   The following additional tasks can be done if the relevant chapters have been covered during class: (c) Suggest a user-centred design process suitable for this design problem. Motivate the choice of design process with reference to other user-centred design processes that are less suitable. (d) Propose a risk management strategy for assessing, monitoring, and communicating risk for the new PCA pump user interface design. (e) One FDA requirement is the following: ”Clearing of the pump settings and resetting of the pump shall require confirmation.” Explain why this requirement is ambiguous and raises questions about the user interface design for a PCA pump that are not explicit in the requirement.

36. Design Engineering

This chapter introduces a *design engineering* perspective to HCI system design. As we have discussed in this Part, the word *system* has many meanings in HCI. In the context of this chapter, the term means a set of interconnected components that give rise to emergent properties that are not attributable to the individual components alone. As the number of components and the interactions among the components increase, a system becomes more complex. *Integrated system design* is the challenge of designing an integrated system that meets the stated objectives.

Here are three examples of successful design engineering from the HCI research literature.

- The data glove [910] is an instrumented glove-based system that allows a computer system to sense position and orientation information of a user's hand and fingers. Thus it allows a computer system to track the user's hand and finger positions, allowing the user actions such as picking up virtual objects and rotating them, among other things. It also enables tactile sensation feedback below each finger. The system is a combined hardware-system covering all aspects required to allow a user to perform gestures and receive tactile feedback, including a manual and automatic calibration procedures.

- *KinectFusion* [371] is a real-time 3D reconstruction and interaction system that is based on a moving depth camera. Using a custom pipeline KinectFusion enables all actions for enabling such features using a commodity depth camera and graphics card. Example applications include low-cost 3D scanning and augmented reality that is aware of the physical surroundings. Further, the system enables tracking of a user's fingers, thereby any physical surface to be reappropriated as a multi-touch surface with passive haptic feedback.

- *Dexmo* [299] is a force-feedback glove intended for virtual reality applications. Dexmo simulates forces using a mechanical exoskeleton. The Dexmo system tracks user motion and when it detects that the user hands fingertips are grasping a virtual object it uses a passive haptic mechanical approach to block user's finger movements. Thus, unlike tactile feedback, the user experiences a real force. Importantly, Dexmo is lightweight and inexpensive to manufacture. A later commercialized version introduced variable force feedback.

We introduce a method for integrated system design in HCI which is adapted from design engineering [617, 251, 819].[1] The previous Part on design (Part VI) explained

[1]Sometimes referred to as engineering design.

methods for creating ideas and prototypes. This chapter elaborates on how to transform *problem statements* into products and services using a systematic approach. In other words, the focus here is on arriving at a system that fulfills a set of *requirements* enabling to be readily realized.

We will focus on specific parts of the typical design engineering process that are relevant to designing, implementing and evaluating interactive systems: task clarification, conceptual design and verification. Other parts of design engineering, such as embodiment design and detailed design tackle the specifics of constructing electro-mechanical systems and are thus not covered. Likewise, we will not cover project management specific techniques, such as change management. Risk management is an important aspect of all system design which we touch on in Chapter 37.

In addition to providing students of HCI a better understanding of how engineers are typically taught to design systems, the methods introduced in this chapter are versatile and addresses a number of common challenges that are often encountered in the design process.

One such challenge is *design fixation*, the risk of too early committing to one particular idea without considering others (see section VI). Another is the inability to keep in mind all objectives and constraints relevant in design. Designers also often exhibit implicit or explicit bias, which can lead to short-circuiting the design exploration stage. When we fail to fully explore design options, we often do not end up identifying the best solutions. A systematic design process reduces this risk by enforcing a consistent process for exploring alternatives.

Another design process challenge is making *informed trade-off decisions*. An integrated system design is rarely, if ever, optimal in all possible aspects, mostly due to many design factors having a negative correlation with each other. A simple example is security and usability of a system: by improving security for example by stricter authentication methods, one often compromises its usability. However, often these trade-offs are not obvious and there is a risk that trade-off decisions are implicitly dictated by the design itself by accident. This can lead to suboptimal designs that are only revealed as such when the system is deployed and the net effect of all those trade-off decisions suddenly become apparent. A systematic design process reduces this risk by considering the translation of the set of functions in the design into a set of *function carriers* in an implemented system as a search problem with multiple solutions. By making the advantages and disadvantages of each solution explicit, implicit trade-offs are made explicit.

A third challenge is *communication*. In practice, integrated design of an HCI system needs coordination of multiple people, or even teams, which are often distributed in different geographies and across companies or company divisions. By documenting key steps in a decision process, they are also made scrutinizable by other people.

A fourth challenge is *integration* of multiple technical disciplines, design knowledge, people, business and regulatory concerns into a functioning system. Examples of difficult integration challenges that arise in HCI systems include 1) systems that rely on software, electronics and mechanical behavior, for instance, wearable devices, medical devices and 3D printers; 2) AI-infused systems that rely on subsystems inferring or predicting users' intention or dynamically adapt based on user behavior; and 3) human-in-the-loop systems that rely on real-time human behavior for successful operation of the system. An example of a system integration challenge is given in the Sidebox, which discusses the engineering of *VuMan*, an early wearable computer.

A fifth challenge is carrying out an appropriate risk assessment and consequently implement a suitable *risk management* strategy for the HCI system. In order to carry out a risk assessment it is critical to understand the technical boundary of the system and the system architecture. Only then is it possible to perform risk assessment at the relevant level for the system. This is essential, especially for safety-critical systems. Due to the complexities and nuanced involved when addressing risk in design, risk management is covered in its own chapter in this book.

Paper Example 36.0.1 : Designing a wearable computer

VuMan is a wearable computer that can assist humans in navigation tasks in the real world, such as providing information moving about in a museum or campus [50]. Developed in early 1990s, VuMan consists of a handheld controller and a wearable display (Private Eye). By rotating the dial and clicking the buttons, a user can move a cursor and interact with a map (e.g., a blueprint of a house) shown on the wearable display.

Three versions of VuMan developed over several years in a multi-disciplinary project. A key challenge to the team was integrated system engineering: how to engineer a wearable computer that meets several requirements for use in the wild, like different temperatures, dirt, water, shock etcetera. The project started with requirements elicitation but later centered of the development the dial. Over several iterations of user studies, and hardware and software engineering, the design became lighter weight, smaller, more robust, more energy-efficient, quicker for users to use, and cheaper to manufacture.

36.1. Design Process

The design engineering process can be decomposed into six interleaved activities:

1. Identifying the purpose of the HCI system.
2. Creating a requirements specification.
3. Arriving at a conceptual design.
4. Translating the conceptual design into an embodiment design.
5. Implementing the embodiment design into a detailed design, which is either ready for manufacturing or deployable as a purely digital product.
6. Verifying that the system fulfils the requirements and validating that the system is usable for its purpose.

While the process appears linear and the above activities will be described in the above order, it is important to note that in practice this process is rarely linear. Moreover, in parallel to the above design process, *risk management* is carried out through the process as a way to reduce both project risk and emergent risks of the deployed system.

36.2. Identifying the Purpose

One of the important tasks when designing a system is to identify the overall *purpose* of the system. It may seem obvious, however, it is often surprisingly difficult for a design team and different stakeholders to come to an agreement of the overall purpose of a proposed system. Writing down a technical description of the purpose of a system helps ensuring the entire design team and all stakeholders share a common understanding of the overall objective. A useful technique for systematically arriving at such a purpose is to evolve a *solution-neutral problem statement*.

A solution-neutral problem statement expresses the overall objective as a problem statement that avoids framing the problem in solution-dependent terms. This eliminates premature commitments to apparent-but-irrelevant constraints. It thereby helps avoiding initial design fixation on a prescribed solution, which otherwise risk resulting in a suboptimal due to an over-constrained design space, prohibiting sufficient exploration of alternative designs.

The process of formulating a solution-neutral problem statement is straight-forward in theory. The first step is to arrive at an initial problem statement and reflect on this problem statement. Does it clearly express a problem worth solving? Is there anything critical missing in this statement? Having critically challenged the initial problem statement, the next step is to progressively raise the level of abstraction of the problem statement to a level that is appropriate given the problem context.

The main idea in raising the level of abstraction of the problem statement is to progressively reformulate it into solution-neutral terms. This is achieved in two steps. The first step is to remove requirements and constraints that have no direct relationship to addressing the problem statement. The second step is to transform quantitative statements into qualitative statements.

By using this two-step process to gradually increase the abstraction level, the search space of possible solutions increases as the problem statement is progressively decoupled from solution-dependent terms. In practice, a solution-neutral problem statement is in a continuum between the two extremes of either being completely solution-dependent or completely solution-independent. Context and professional judgement determines the appropriate abstraction level.

As an example, consider the following initial problem statement: “Design an updated form-filling warehouse inventory management interface based on last year’s version of a mobile app running on capacitive touchscreen-enabled phone, consisting of five forms on five separate pages, each with a Submit and a Cancel button”. It can be successively evolved into increasingly solution-neutral problem statement as follows:

- Design a form-filling warehouse inventory management interface for a capacitive touchscreen-enabled device, consisting of five forms on five separate pages, each with a Submit and a Cancel button.

- Design a form-filling warehouse inventory management interface for a mobile device.
- Devise a means for inputting structured information for warehouse inventory management.
- Devise a means for managing warehouse inventory.
- Devise a means for managing information.

The above example illustrates that there is a range of successively solution-neutral problem statements for any initial solution-dependent problem statement. However, at some point the level of abstraction is so high that the statement is provides little to no guidance. In practice, careful judgment is required to set the solution-neutral problem statement at the correct abstraction level.

Once a solution-neutral problem statement has been evolved it can be used to denote the overall function of the system. As we shall see when performing conceptual design, an overall function is a good starting point for decomposing a function structure of the system.

Running example: A device for affection across distance

As an example of a hypothetical design, consider a wearable device that couples affection between two individuals separated in space. For example, such a device could take the form of two electronic rings connected via the Internet that will glow when one of the ring users is rubbing or turning the ring a certain way. The overall function of such a system could be described as Couple Affection and a solution-neutral problem statement may take the form of, for example, "Design a wearable device that couples affection between two individuals that may be present in separate locations."

36.3. Specifying Requirements

Having identified the overall purpose of the system it is now possible to identify more clearly the objectives and constraints of the system. These are imposed in the form of *requirements* on system behavior and qualities either intrinsic or emergent from the system.

A *requirement specification* is a document that specifies characteristics of a system. It is determined in the beginning of the design process. While a requirements specification is ideally perfect in its first instantiation and remains unchanged throughout the design process this is very unlikely to happen in practice. Rather, a requirements specification is often referred to as a *live document* that is updated as new information alters the design specifications.

As a requirements specification is referred to frequently in the design process it is vital it is correct. Requirements specifications are therefore reviewed and the process of arriving at a requirements specification is often following a prescribed process, which is often field- or organisation-specific. The process of arriving at a correct requirements specification is known as *requirements engineering* [422] and it is an active area of research [651].

The importance of a correct requirements specification increases with the complexity of the system. Unfortunately, in practice, it is notoriously difficult to arrive at a correct requirements specification as this demands a complete understanding of every relevant facet of the design. For example, requirements relating to users' needs or wants are difficult to elicit and unlikely to be complete or fully representative of the needs and wants of a diverse target audience. This is due to a number of factors, including sampling error, the robustness of the methodology, and the difficulty of users to fully anticipate their own future needs and wants. This is one reason why requirements specifications are live documents.

A good requirements specification provides an in-depth understanding of the problem and covers all relevant aspects of the system design. It is also clearly written and thus helps improve communication between different team members. As a consequence, a well-written requirements specification reduces time and cost in the design process and is more likely to give rise to a high-quality system.

While requirements cover all aspects of system design, they can generally be divided into four categories:

Technical requirements cover functional and performance characteristics of the system, such as latency and recognition accuracy.

Business requirements cover costs, scheduling and other management-related aspects of the system and the design of the system.

Regulatory requirements cover governing laws, industrial standards and product regulations.

User-elicited requirements cover the needs and wants of users.

A useful requirement has the following properties:

Solution independent A requirement should normally not prescribe a specific pre-determined solution. A requirements should specify what needs to be done, not how it is done.

Clear A requirement should be unambiguous and understandable by all members of the design team that need to be aware of it.

Concise The wording in a requirement should be phrased succinctly.

Testable A requirement should be testable to ensure it is possible to later verify that the requirement has been met. In many cases, testability is ensured by introducing quantitative target values, limits, tolerances or ranges. In other cases, testability can be ensured by prescribing acceptance criteria.

ID	Requirement	Source
1	Device must indicate its power on status	Technical team
2	Device must be comfortable to wear	Market research
3	Device must be able to sense at least five levels of communicated affection from the user	Focus group research and literature review
4	Device must be able to wirelessly receive an affective signal from a remote paired user	Focus group and market research
5	Device must communicate affection from a paired remote user in five discernible levels	Experimental research

Table 36.1.: Example requirements for an affective wearable device.

Traceable A requirement should be traceable to its source. If the need ever arises to modify the original requirement it is important to understand why the requirement was specified in particular way in the first place.

In addition, the requirements specification as a whole should be complete and include all areas of concern throughout all relevant phases of the system, which, for example, may include requirements on disposal or re-use of the system, or system support. It is difficult and error-prone to write a complete requirements specification without reference to complete system functionality. Therefore requirements specifications are often written while designing the function structure of the system, which is addressed in the next section.

Some example requirements considering a hypothetical wearable device for coupling affection between two individuals separated in space are shown in Table 36.1.

36.4. Conceptual Design

Having identified the overall function of a system the *conceptual design* phase is concerned with two main tasks. The first is to elaborate the specific functions that the system will be required to carry. Such *functional modeling* allows the design team to reason about required functionality without premature commitment to specific *solution principles* that dictate a specific method to carry out a function.

Figure 36.1 illustrates the conceptual design process as an information processing activity. A technical description, such as an initial requirements specification, and design resources, including design knowledge, feeds into a transformation process that turns a

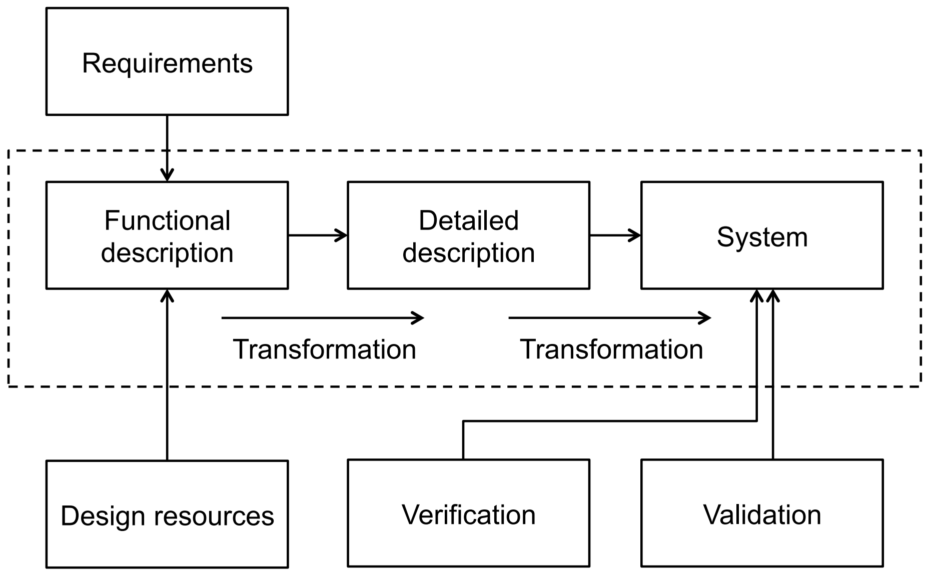


Figure 36.1.: Design engineering as an information processing activity [429].

functional description into a detailed description. A common mistake is to prematurely commit to a detailed description without a complete understanding of the functional description. A way to think about it is to consider the functional description the *what*—what should we build?—and the detailed description the *how*—how should we build it?

Also shown in Figure 36.1 is a transformation process from the detailed description to working system. Once a working system has been constructed it is possible to undertake verification and validation to ensure that all requirements have been met and that the deployed system successfully addresses the solution-neutral problem statement.

36.4.1. Function Modeling

Two useful simple modeling abstractions are *function structures* and *FAST diagrams*. FAST stands for Function Analysis Systems Technique. In both methods, a *function* is described as an active verb followed by a noun ("push button", "select shape", "start motor", etc.).

A function structure is derived by first identifying the overall function. It is possible, and often necessary, to repeat the process for several overall functions of the design. Having identified an overall function its interaction with the environment is modeled by flows of energy, materials and signals give rise to an overall function with inputs and outputs of energy, materials and signals as shown in Figure 36.2.

The rounded rectangle in Figure 36.2 is the *technical boundary of the system.* Anything outside it, is not modeled except as inflows and outflows of energy, material and signals. The technical boundary of the system is important and must be set with care. For

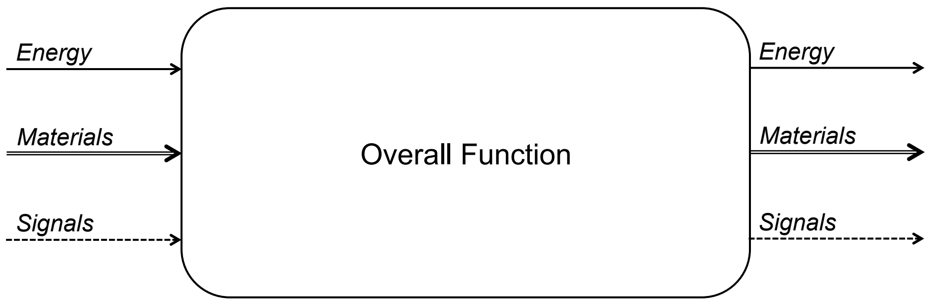


Figure 36.2.: A function structure of the overall function of a system, showing the flows of energy, material and signals. The rounded rectangle around the overall function forms the technical boundary of the system.

example, in some cases users are included within the technical boundary of the system because these users are carrying out functions necessary to carry out the overall function.

Function structures can be nested, which makes it possible to increase the granularity of the functional description. Figure 36.3 illustrates this idea by breaking down the overall function in Figure 36.2. Figure 36.3 shows an internal decomposition of the overall function split into four main functions. Function 1 and 2 use energy to process material, suggesting they are *process* functions. Function 3 and 4 manage the flow of signals and suggest they are *control* signals. Further decomposition of, for example, Function 2 could reveal, sub-functions further detailing the control and process aspects.

In the case of the wearable device coupling affection between two individuals, the overall function is 'Couple Affection'. While simple, the function structure has already recorded one of the most vital piece of information of any design: the overall function. To carry out its purpose the design needs to manage flows of energy and signals.

This overall function structure for Couple Affection in **??** can then be decomposed into a number of key sub-function as shown in Figure 36.4. The incoming signals to the overall function have now been detailed as *User Affection*, sensed locally by the wearable device, and *Remote Affection*, transmitted from a remote device. The function structure identifies a functional need to Supply Energy in order to sense, modulate and send locally sensed affection to a remote device and to be able to receive and display remote affection provided from a remote device. Crucially, the function structure is focused on revealing critical functions and their dependencies and does not prescribe any specific solution or function carrier to carry out any of the functions.

FAST diagrams is another function model design method which can replace or complement function structures. The fundamental idea behind FAST diagrams is to gradually decompose higher-order functions at a higher abstraction level into one or several lower-order functions at a lower abstraction level. Figure 36.5 illustrates this idea diagrammatically. The horizontal axis is the level of *abstraction*, ranging from high abstract to the left to low abstraction to the right. In other words, the further a function is to the right the

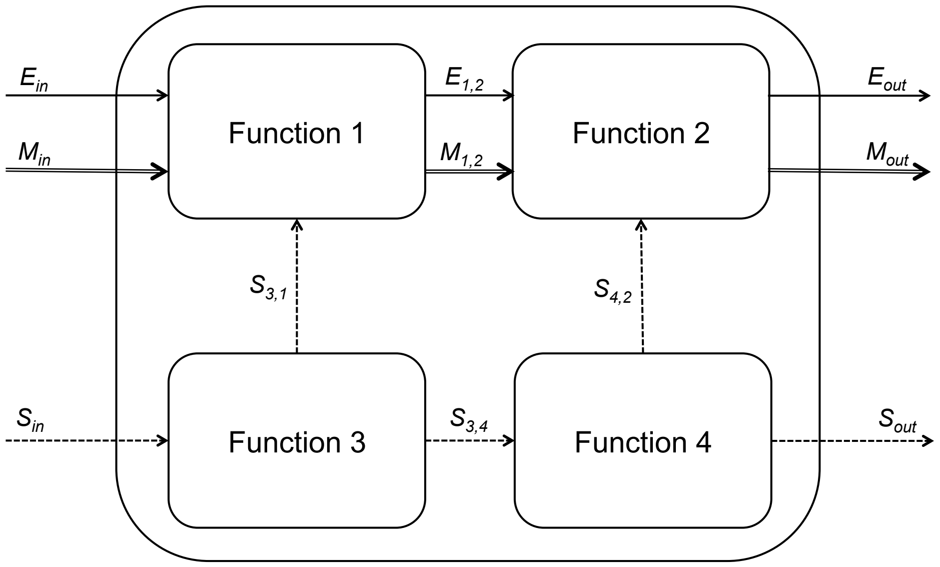


Figure 36.3.: A breakdown of the overall function structure in Figure 36.2, revealing four main functions and the flows of energy, material and signals between them.

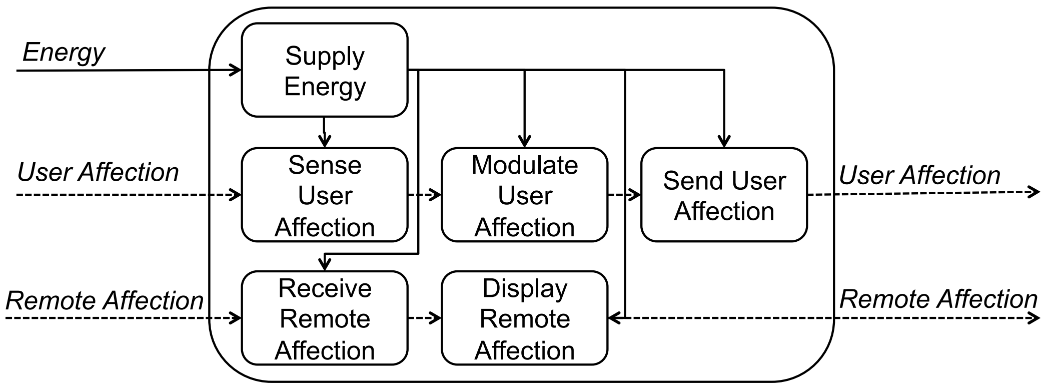


Figure 36.4.: A decomposition of the overall function structure of the wearable device coupling affection in ??, revealing the main functions and the flows of energy and signals between them.

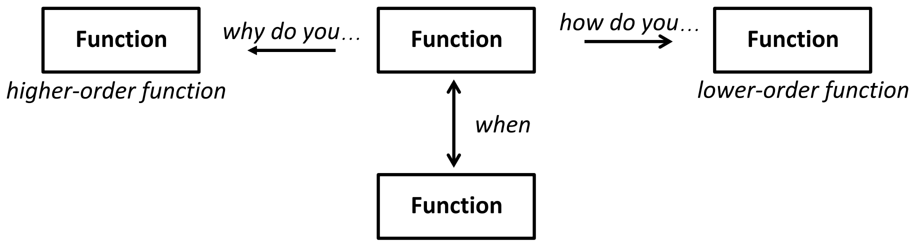


Figure 36.5.: A diagrammatic representation of the grammar of a FAST diagram. The horizontal axis represents abstraction—functions at a higher abstraction level are to the left of functions at a lower abstraction level. The vertical axis represents time or sequential ordering of functions.

more concrete it is. The vertical axis is *time*, in other words, *when* a function is carried out. A function to the left of a function answers the question "*Why* do you carry out this function?" while a function to the right answers the question "*How* do you carry out this function?".

Figure 36.6 shows an example of a partial FAST diagram for the wearable device coupling affection. The overall function is Couple Affection. In order to Couple Affection the device has to be able to Communicate Affection and Perceive Affection. To Communicate Affection the device has to Sense User Action, Module Signal and Transmit Signal. The FAST diagram can also be read the other way around. Why does the design require a Sense User Action? This is because it needs to Communicate Affection. Why does it need to Communicate Affection? This is because it needs to Couple Affection.

Several controllable and uncontrollable parameters can be attached to any of these functions as part of an early analysis of required function characteristics. This analysis in turn can drive another iteration of the requirements specification.

For instance, consider the function Sense User Affection in Figure 36.6. A signal corresponding to a proxy of a user's affection can be transmitted from the user to the wearable device in a number of ways, for example, by touch, by rubbing the device, or by applying pressure. Another possible solution is to allow the user to communicate affection by the duration they touch or apply pressure against the device. In such a case, *duration* is an example of a controllable parameter of the function because it can be controlled by the device designer. An example of an uncontrollable parameter would be the certainty on whether an affective signal was intended to be communicated by the user, and if so, the magnitude of affection that was intended. It is not possible be completely sure about the user's intention in this regard, even if the system detected some form of user interaction with the device. A misdetection can happen for many reasons: sensor error, human motor control noise, or a cognitive error, such as misunderstanding how affection is meant to be communciated via the wearable device.

Another example is the function Display Signal in Figure 36.6. This function also

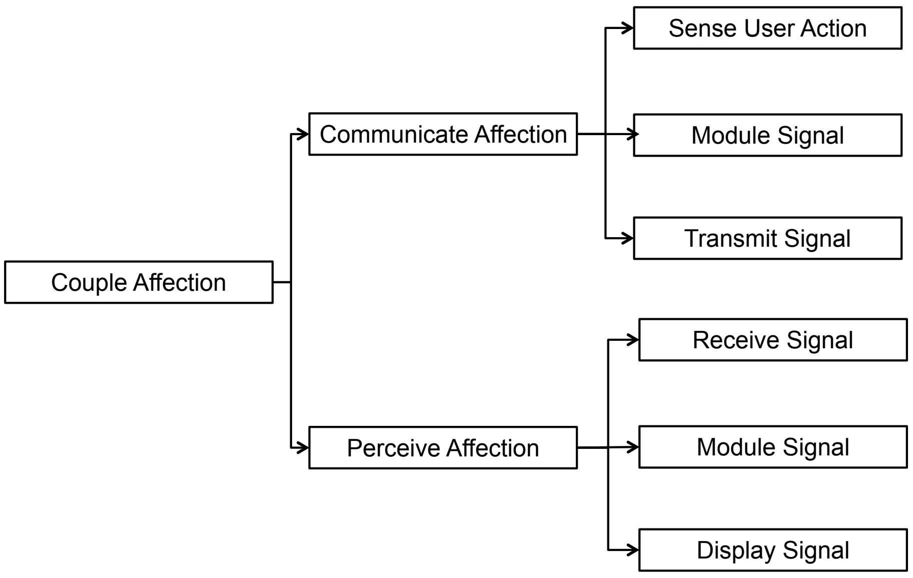


Figure 36.6.: A high-level FAST diagram for a wearable device coupling affection between two individuals.

depends on a number of parameters. Among the controllable parameters, it includes concerns such as whether the device displays affection as a continuous or discrete signal, the display resolution (for example, three discrete levels of affection), the media used (for example, light, sound, vibration) and the governing parameters around the chosen media. As is evident, the degree of parameterization depends to some extent on how solution dependent the FAST diagram is. An example of an uncontrollable parameter is the external environment the device and the user is operating in, which will have consequences for how well a function such as Display Signal can operate. Another example of an uncontrollable parameter is context of use, including the specific situation the device is used in. A user may for example be standing still alone, walking in a crowded room, or drive a car. While uncontrollable parameters cannot be explicitly controlled by the design, their effect on the design can still be analyzed and taken into account. For example, the effect of different room illumination on the display of an affective signal conveyed using a LED array can be analyzed by experimentation.

To create a FAST diagram it is often best to start with general functions ("Which must be carried out?") and then progressively evolve a more specific function structure. For example, in the example of a wearable device that couples affection, the two key functions are Communicate Affection and Perceive Affection. Each function should be described as simple as possible, ideally with a just a verb and a noun. Then chronologically trace through each function that must be realized for the design to work. In practice, this often requires both careful self-reflection and consultation with other members of the design team or stakeholders to ensure all necessary functions are included. It is important

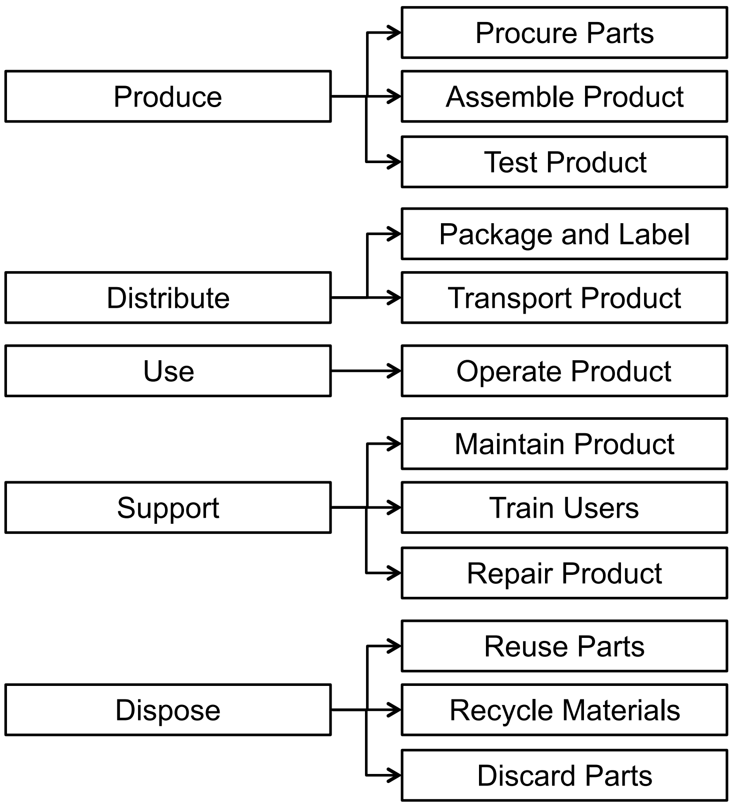


Figure 36.7.: A FAST diagram presenting an overview of the main functions across the lifecycle of a generic product.

to remember to include all special modes of the design, such as stand-by, start-up and re-start of the system. The sole focus should be around system behavior. Therefore, avoid descriptions around the embodiment of the design and in particular avoid specifying shape, form, structure of specific solutions. Finally, it is possible to customise the FAST-diagram by annotating it, for example, by indicating critical functions, possible parameters that function performance is crucially dependent on, etc.

FAST diagrams are often used by engineers and designers to ensure all relevant functionality is captured in the requirement specification. The decomposition process forces everybody in the design team to be clear about exactly which functions that must be carried out to realize an overall function. Another use of FAST diagrams is lifecycle analysis. Figure 36.7 illustrates how the lifecycle phases of a product can be decomposed into sets of FAST-diagrams for each relevant phase. Again, functional modeling of the lifecycle of a product is useful to inform the requirements capture process of all necessary functionality throughout all the phases of a product, including assembly, distribution, support and eventual disposal or reuse.

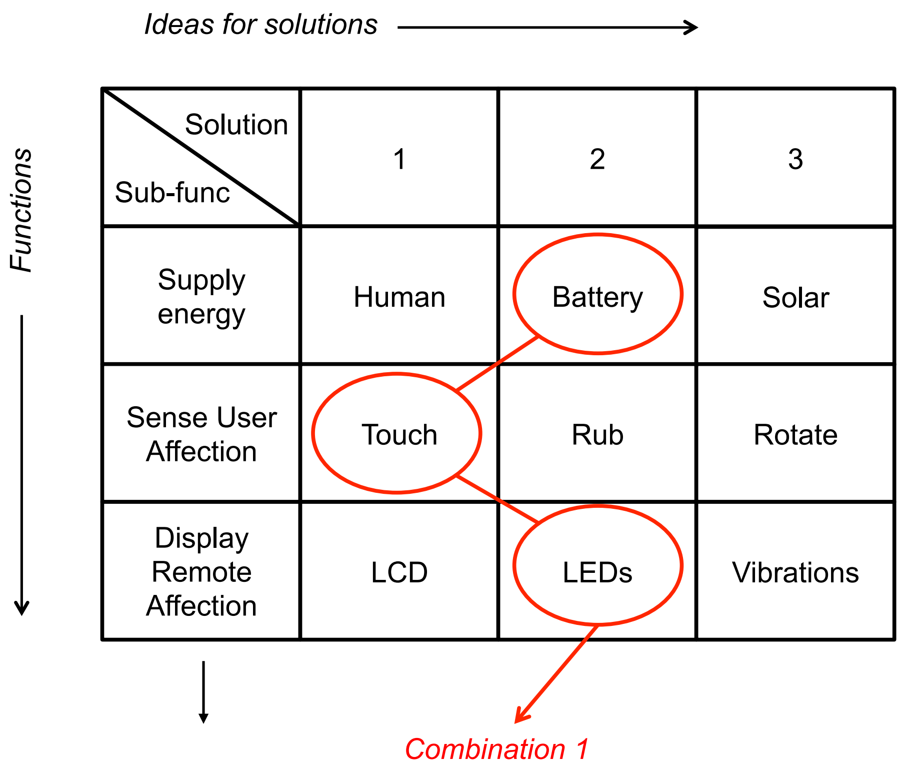


Figure 36.8.: A morphological chart illustrating a set of alternative solutions for a subset of functions in a pulse meter.

36.4.2. Translating Functions into Function Carriers

Having identified the functions that must be realized by the system, the next step is to understand how to best implement these functions. In other words, the objective is now to carry out a design process that considers the best alternatives for translating the functions into *function carriers*. A starting point for this design process is the construction of a *morphological chart*.

A morphological chart is a table in which every row maps a specific *function* to a set of candidate *function carriers*, often referred to as *solutions*, and sometimes as *solution principles*. This chapter uses the terms function carrier and solution interchangeably.

Figure 36.8 illustrates this idea for three functions of a wearable device for coupling affection between two individuals. The function Supply Energy can, for example, be mapped to three different solutions: 1) the device can harvest some of the human user's energy; 2) the device can use a battery; or 3) the device can use solar. The function Sense User Affection can be realized by, for example, by 1) touch; 2) a rubbing motion; or 3) a rotating motion. Finally, the function Display Remote Affection can be realized using, for example, 1) an LCD screen; 2) a LED array; or 3) by communicating tactile vibrations to the user's skin in preset patterns (for instance, using separate codes for three levels of affection).

All these solutions can be further detailed in a second iteration of a morphological chart following initial selection of a preferred solution. For instance, an energy source, such as a battery can be further explored in terms of kind of battery is the most suitable, should it be replacable, rechargeable, and so on.

Having constructed the morphological chart the idea is now to generate combinations by going through each function in the morphological chart and selecting a set of corresponding solutions that realize all the functions required to carry out the overall function. For example, in the morphological chart in Figure 36.8, Combination 1 has been generated by selecting a partial design for a wearable device based on using a battery as an energy source, sensing the user's affection using touch and displaying remote affection using a LED array.

Theoretically, assuming each function maps to exactly k solutions, there are nk combinations for n functions. This means that the morphological chart in Figure 36.8 can generate nine unique theoretical combinations. As both the number of functions and the number of solutions tend to be large in practice, it is not meaningful to exhaustively generate all, or a large set of combinations. Instead, the design team uses professional judgement, requirements and any other sources of design insight to generate a subset of promising combinations.

36.4.3. Concept Selection

Having arrived at a set of combinations, the next step is to rank these combinations and arrive at a final recommendation. As a starting point for a discussion, this can be done by scoring conceptual designs (combinations of solutions) against relevant criteria. Figure 36.9 score two concepts so that they can be compared numerically.

The scoring works as follows. First, each criterion is assigned a weighting in a nominal range between 1 and 5. Thereafter each concept is scored against each criteria. This results in another value in a nominal range between 1 and 5. This value is then scaled by the weighting for that feature. For example, in Figure 36.9, the value for Concept 1 for the criteria Weight is 2. Since the weighting for the criteria Weight is 3 the weighted value is 6.

The final score for a concept is thus a linear combination of the form $c_1v_1 + c_2v_2 + \ldots + c_nv_n$, where c_i and v_i is the ith criterion and corresponding value respectively. By calculating the linear combination scores for the concepts the scoring mechanism provides an overall score. For example, for Concept 1 in Figure 36.9 the combined score is calculated as $2 \cdot 3 + 2 \cdot 3 + 2 \cdot 2 + 3 \cdot 4 = 6 + 6 + 4 + 12 = 28$. For Concept 2 the calculation is $4 \cdot 3 + 4 \cdot 3 + 4 \cdot 2 + 1 \cdot 4 = 12 + 12 + 8 + 4 = 36$.

It is often useful to include a point of comparison that calibrates the quantification. One method to do this is to introduce an ideal concept that attains the maximum weighted value for each criterion. Figure 36.9 shows the scoring for a hypothetical ideal concept.

Another possibility is to compare against a *datum*, such as, for example, an existing design based on an older product or a competitor's product. In the latter case, the scoring

		Concept 1		Concept 2		Ideal
Criteria	Weighting	Value	Wt val	Value	Wt val	Wt val
Weight	3	2	6	4	12	15
Appearance	3	2	6	4	12	15
Power	2	2	4	4	8	10
Accuracy	4	3	12	1	4	20
			28		36	60

Figure 36.9.: A comparison of two conceptual designs scored across a set of features against a theoretical ideal concept that scores perfectly in every category.

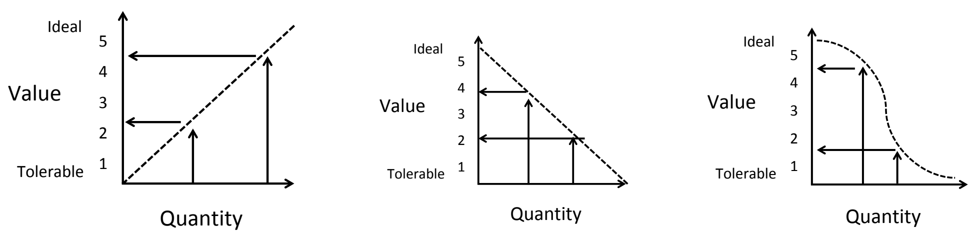


Figure 36.10.: An illustration of three different functions for translating a quantity (battery life, weight, accuracy in sensing, etc. into a value that can be used to score concepts.

can be used as part of a *competitive analysis* to better understand which criteria a new design could realistically excel at compared to an existing or hypothetical competitor in the market.

To calculate the weighted two quantities are required: 1) the weighting for the criteria; and 2) a function that converts a quantity, for example, battery life, weight, height, accuracy, into a nominal value. Weightings can arise them from many sources, such as organizational knowledge of the priority of criteria, for example by previous sales figures and support data. It can also arise from market research based on, for example, focus groups asking participants existing or potential customers to rank product qualities. Weightings may also arise from domain knowledge, business concerns or regulatory requirements.

There are also many methods for converting a quantity into a nominal value. The simplest method is to use a zero-intercept linear function over the effective quantity range to linearly map a quantity to a nominal value as shown in Figure 36.10 (left). An example is a quantity such as battery life. For a hypothetical wearable device worn around the wrist the viable range may be determined by the designers to reside between 8 hours and 24 hours. Battery life is then mapped such that an 8 hour battery life is mapped to a nominal value of 1, a 24 hour battery life is mapped to a nominal value 5, and any battery life between 8 and 24 hours is linearly interpolated to a corresponding nominal value.

For some criteria, such as weight, it is more natural to map the inverse, as illustrated in Figure 36.10 (middle). For example, If a design team has determined that the weight of wearable device must reside in a range between 100 g and 250 g then a 100 g is mapped to a nominal value 5, 250 g is mapped to a nominal value of 1, and any weight in between is linearly interpolated.

The function used to calculate a nominal value does not need to be linear. Figure 36.10 (right) illustrates a non-linear mapping between a quantity and a nominal value. The function can also be discrete, for example, mapping a wide range of quantities into the same nominal value.

As the scoring method is merely calculating linear combinations it is very simple, which is an advantage as it makes it easy to understand and apply. Another advantage is that the scoring method maps directly to criteria and thus helps to ensure the conceptual design is addressing the criteria articulated in a requirements specification. It may also make the design team more aware of the significance of certain criteria and question whether all relevant criteria have been fairly represented are fully reflected in the requirements specification. Finally, the analysis elucidates intrinsic tradeoffs that emerge in any non-trivial design, such as the inherent conflict between device size and battery life and forces the design team to carefully consider how to balance them.

However, the simplicity of scoring concepts as linear combinations comes at a cost of reduced accuracy and increased uncertainty. Therefore it is important to be aware of the inherent limitations of such a scoring approach to conceptual design selection.

First, the method assumes all relevant criteria have been represented, which is hopefully true, but often in practice difficult to ensure. Second, the method assumes weightings and concept values are error-free point estimates when in reality they are uncertain. Third, the method assumes all criteria can be related mathematically as a linear combination.

For these reasons, concept scoring is error prone and the results should be used with caution. Any concept selection should not be made with sole reference to how well the concept scores, in particular not the aggregated score, which is more useful as a method for an initial ranking of concepts than basis for final selection. A decision to select a particular concept should be complemented with a written narrative that explains the reasons for selecting a concept with reference to the criteria, weightings, other concepts and any datum, competitor product or ideal concept.

As an example of the danger of relying solely on concept scores, consider the two concepts for a pulse meter in Figure 36.9. Concept 2 has a higher aggregate score (36) compared to Concept 1 (28). At first glance, Concept 2 appear superior. However, Concept 2 is scored only 1 for accuracy compared to 3 for Concept 1. Furthermore, accuracy is the criterion with the highest weighting. Therefore, despite other drawbacks, it may be wiser to proceed with Concept 1 instead of Concept 2 and investigate whether the drawbacks of Concept 1, such as the increased weight, worse appearance, and poorer power solution can be mitigated. Alternatively, further research may be required to investigate whether the accuracy can be improved for Concept 2. In this way, concept scoring is used to drive further design exploration, rather than being relied upon as a clearcut method to identify the best concept.

36.4.4. Product Architecture

Having arrived at a functional model it is possible to consider the allocation of *functions* to *modules* and how such modules interact. A product architecture is in a continuum between *integral* and *modular*. In a fully integrated architecture every function maps to a single module. In a fully modularised architecture, every function maps to its own module.

Product architecture was originally proposed by Ulrich [818] for the design of physical products. Here a module can be seen as a physical component. In a more modular architecture, interfaces and interactions between modules are more decoupled. This means changes in individual modules does not necessarily imply any changes in other modules, a similar reasoning as the encapsulation principle in, for example, object-oriented programming. Because of this, it may be possible to design individual modules separately. In theory, everything else equal, this allows reuse of existing modules across products and thus can reduce design cost and in particular allow a variety of different products. An example is a modular video camera with a separate tripod, battery, carrying bag, lens system, etc. In contrast, in a more integral architecture boundaries and interactions between modules are coupled and modifications in one part of the system is likely to affect other parts. This means modules must be designed in collaboration and it is difficult to allow for variety or change. Such integration may enable optimization at the cost of flexibility.

A modular architecture implies some form of interface mechanism between modules. Therefore, Ulrich [818] also proposed three archetypical modular architectures:

Slot modularity A slot modularity means modules are connected using a range of standard interfaces. An example of this is a TV with a high-definition multimedia interface (HDMI) connector, an antenna connector, etc.

Bus modularity A bus modularity means modules are connected using a single standard interface, such as the Universal Serial Bus (USB).

Sectional modularity A sectional modularity means there is no standard interface and no main module. An example is pipework.

Product architecture choices matter throughout the lifecycle of the product or the system. At the design stage, how to realize functions, divide up design tasks and potential reuse of existing designs depend on the architecture. At the production system stage of a physical system, the architecture affects assembly sequences, tooling, reuse and the equipment involved. At the production stage, the architecture helps determine unit cost,

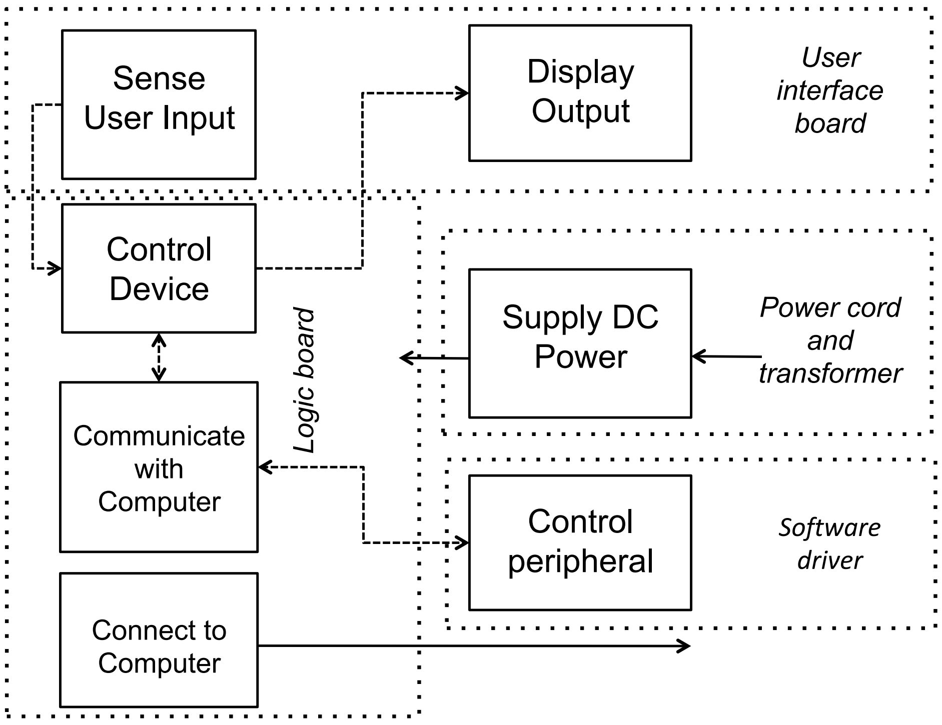


Figure 36.11.: A function structure of a computer peripheral indicating the flows of energy and signals. If, for example, this peripheral was a laser printer, there would also be flow of material (paper and toner). The function structure has been modularized into four modules based on functional affinity: 1) user interface board; 2) logic board; 3) cord and AC transformer; and 4) software driver.

the ability to reuse standard parts or modules, viability of late customization and the ability to offer product variety. During usage, the architecture will determine whether the product is optimized for performance (which usually involves a high degree of integration) versus generalizability and flexibility. Other factors that are may affected during usage include the ability to adapt and maintain the system during use. Post production, the architecture affects service and maintenance, such as the ease of replacing physical parts, and the ability to upgrade individual software modules. In addition, the ability to change the product or system at a much later stage is also in part determined by the level of modularity.

The starting point for creating a modular product architecture is a schematic overview of the product, for example, a function structure model of the system. Then select the desired level of product variety depending on factors such as level of reuse, ability to change the product and variety of product offerings. Finally, create modules by clustering functions in the schematic overview. An example of this process is shown in Figure 36.11

which shows a function structure model of a computer peripheral and the flows of energy and signals between the functional elements. This schematic can be clustered into four modules based on functional affinity: 1) a user interface board handling user input and output; 2) a logic board controlling the device and communicating and connecting to the computer; 3) a power cord and transformer; and 4) a software driver. This ability to easily modularize a system early in the design is another advantage of function structure models.

It is also possible to design a computer peripheral using a more integral architecture, for example, by integrating the user interface board, logic board and power into a single unit, for example, by allowing the peripheral to draw power from a USB connection, thus eliminating the need for a power cord and transformer. This latter architecture can make sense for instance if the computer peripheral is an integrated hand-held unit, such as a hand scanner. In contrast, the former architecture illustrated in Figure 36.11 can serve as a blueprint for a heavy duty computer peripheral that requires external power. Examples of such peripherals include laser printers, virtual reality headsets and 3D printers.

36.4.5. Next Steps: Embodiment and Detailed Design

Once a concept has been agreed upon, the next two stages is *embodiment design* and *detailed design*. Embodiment design is the stage in the design process that succeeds the identification of a conceptual design. The embodiment stage results in design sketches, prototypes, layouts of mechanical components, identification of critical software modules and electronic components, and so on.

Detailed design is the phase in the design where the system is either ready to be either directly distributed in the form of software or sent to manufacturing in the form of manufacturing instructions. For example, for a software-only system, the detailed design phase would involve designing the software architecture, class descriptions,

36.5. Verification and Validation

A fundamental activity in design engineering is verification and validation. *Verification* means systematically checking that all requirements have been met. In other words, verification is a process for ensuring a built system meets its specification. *Validation*, in contrast, is the process of ensuring the system fulfils its intended purpose. There is therefore a clear distinction between verification, ensuring the system is built correctly according to its specification, and validation, ensuring the system is capable of assisting end-users in solving their problem. Due to the separation of verification and validation it is possible to succeed in one and not the other. For example, it is possible to pass verification by building a system that fully meets its requirements, but nevertheless fail in validation if the system is incapable of carrying out its intended function or fulfilling its purpose. It is also possible to fail in verification and succeed in validation; a system can be usable for its intended purpose even if it does not fulfil all requirements in its specification. A common problem is a system succeeding in verification and failing in validation. This problem arises due to the high risk of a misalignment between the requirements of a system and the qualities required of the system to be usable in practice.

36.5.1. Verification

Verification is the process of ensuring requirements have been met. In order to succeed with verification it often helps to structure the verification process into a *verification cross-reference matrix* (VCRM). A VCRM is a row-by-row specification for how to verify individual requirements. There are many ways to set up a VCRM, however, they usually contain following columns:

Requirement ID An identifier that matches the identifier of a requirement in the specification.

Requirement The requirement addressed by the verification method.

Verification Method A specification that details how verification is to be carried out.

Allocation A list of components, parts, subsystems, etc. that are affected by this verification.

Success criteria A free-text clear description of what constitutes success. This may be a qualitative, quantitative or a combination of both.

When considering a verification method it is useful to consider a few fundamental higher-level verification approaches.

Inspection, sometimes referred to as *examination*, means using one or several of the five senses to non-destructively ensure a system posses a certain quality. For example, a requirement to paint a door blue can be inspected by visually checking the color of the door.

Demonstration refers to ensuring that manipulating the system according to its intended use results in expected outcomes. For example, checking that a registered user's finger touch unlocks a finger-print lock.

Test is checking that feeding predefined inputs yields expected outputs. For example, calling a software function with set input values and then programmatically checking that that the output values of the function are correct.

Analysis means carrying out verification using calculations, models or testing equipment to predict system characteristics based on, for example, a sampled subset of components or samples of test data.

Table 36.2 shows an example of a partial VCRM for a wearable device coupling affection between two individuals.

It is useful to develop a VCRM in conjunction with a requirements specification as a VCRM forces the designers to explicitly consider viable verification methods success criteria at an early stage in the decision process. Such design activities can also trigger the need to construct prototypes or perform system simulation to assess the viability of adhering to requirements at an early stage. For example, the VCRM in Table 36.2 specifies

ID	Requirement	Verification Method	Allocation	Success Criteria
1	Device must indicate its power on status	Inspection	User Interface	Visible and legible power on status
2	Device must be comfortable to wear	Test	Device Case	Median ratings above 4/5 on Likert scale by representative user sample from target audience ($n > 24$)
3	Device must be able to sense at least five levels of communicated affection from the user	Test	User Interface	Consistent ability (> 95%) by representative user sample from target audience to communicate five different levels of intended affection ($n > 24$)
4	Device must be able to wirelessly receive an affective signal from a remote paired user	Demonstration	Wireless Stack	Ability to receive and respond to test signal.
5	Device must communicate affection from a paired remote user in five discernible levels	Test	User Interface	Consistent ability (> 95%) by representative user sample from target audience to detect five distinct levels of intended affection ($n > 24$)

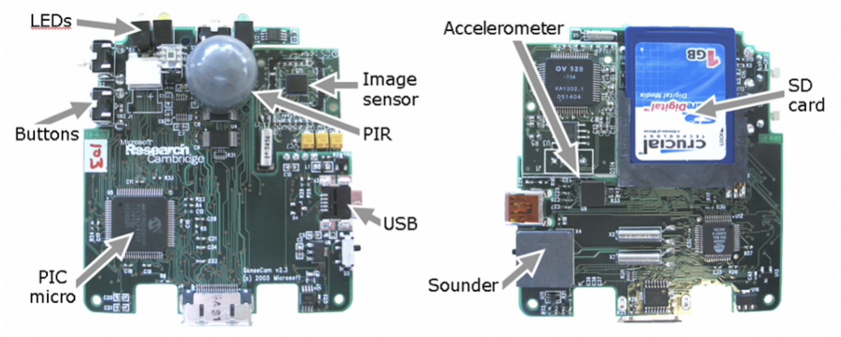
Table 36.2.: Example requirements for an affective wearable device.

five distinct levels of affective signals to be sensed by the user. This may require early prototyping and experimentation to be confident that any intended hardware/software solution is capable of satisfying such a requirement.

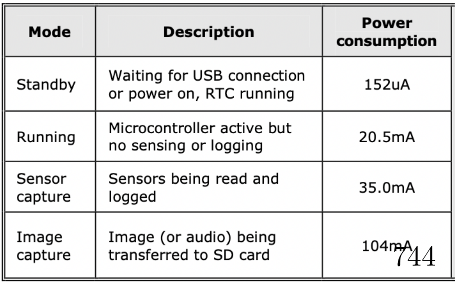
Paper Example 36.5.1 : Designing a life-logging device

SenseCam [339] is a life-logging device in the form of a wearable camera that is worn around the neck of the user, see the figure below. SenseCam takes photos periodically based on a timer or sensor events, such as a significant change in the light level of the camera image. In addition, it also allows the user to trigger image capture.

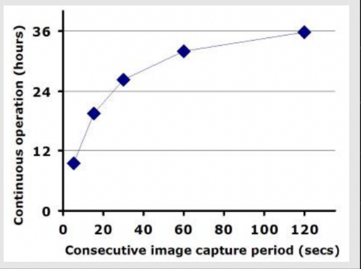
The SenseCam system was developed as a wearable retrospective memory-aid and its design process begins with a requirements specification that identifies important design requirements, such as the form factor, battery life, storage capacity, and ease of use. This specification is then translated into a software-hardware system that realizes the system functions, such as LEDs displaying system status, buttons for triggering user functions, an image sensor, and so on. The hardware design is shown below.



An important design parameter of SenseCam is battery life. The analysis below (left) shows that taking images consumes the most power. The plot below (right) shows continuous operation of SenseCam in hours as a function of the period between image capture in seconds. The operating point chosen was a period between image capture of 30 seconds, which according to the plot results in the SenseCam being able to operate continuously for approximately 24 hours before it has to recharged.



Mode	Description	Power consumption
Standby	Waiting for USB connection or power on, RTC running	152uA
Running	Microcontroller active but no sensing or logging	20.5mA
Sensor capture	Sensors being read and logged	35.0mA
Image capture	Image (or audio) being transferred to SD card	104mA



SenseCam has been validated over a 12-month period with one user with amnesia. The validation results show that the user was able to recall events that were previously impossible to recall. At a final test the user was able to recall 76% of events with the aid of SenseCam images, which is seen as exceptional for a user with amnesia.

36.5.2. Validation

Validation is the process of ensuring the system is fulfilling its purpose for the intended users. As such, validation is intrinsically dependent on the particular system that has been created. See Part VIII in this textbook for detailed description of a variety of empirical methodologies that can be used to assess whether users are able to use a system for its intended purpose.

36.6. Design engineering and user-centered design

This chapter has introduced design engineering as a design process that addresses the entire spectrum from the specification of a solution-neutral problem statement to the implementation, verification and validation of an interactive system.

While there are clear overlaps between design engineering and the user-centered design processes we discussed in Chapter 33, there are three important aspects of design engineering that in our experience can assist HCI system designers.

The first is the explicit separation of *functions*—the *what*—from *function carriers*—the *how*. This allows HCI system designers to explore *function models* that can relate functions in various ways, such as the flow of information between functions using function structures, or the level of abstraction between functions in FAST-diagrams. This enables exploration of solutions *before* even considering any concrete design solutions.

The second aspect is the systematic exploration of a mapping between functions and function carriers using, for example, morphological charts. Such an exploration can be aided by scoring the suitability of different function carriers using estimates of their suitability as a function of some known quantity, such as the suitability of a battery given battery life. This allows *informed* exploration of a wide spectrum of possible solutions as well as the construction of a narrative clearly explaining a chosen concept with reference to a set of criteria.

The third aspect is the idea that function models and conceptual designs can be parameterized and we can use such parameterizaton to investigate the optimal values of controllable parameters—parameters the designer can influence—and understand how the system behaves in relation to uncontrollable parameters—parameters that are outside the designer's control. This allows us to carry out analyses before deciding on specific implementation strategies and before carrying out in-depth user studies.

Finally, when building a working interactive system the management of project risk is essential. Changing requirements late in a project may be very expensive, or even impossible. By taking function models and the conceptual design seriously, it is possible to rectify many design mistakes and discover many unknowns early in the design process.

Thus, while design engineering shares many traits with user-centered design process, the emphasis on integrated system design amplifies the aspects of an implementation that sometimes must be considered very early on in a design. For example, for the life-logger SenseCam [339] to be effective, it was absolutely necessary to understand how the period between image capture related to a useful battery life. This type of analysis can be fruitfully carried out *before* building a device and engaging in user studies, in particular studies with users with disabilities.

Summary

- Systematic design processes minimize risk and improve the quality of system design.
- Design engineering can be broken down into the following activities: arriving at a solution-neutral problem statement, elaborating a requirements specification, conceptual design, embodiment design, detailed design, and verification and validation.
- A function model allows a designer to specify the interrelations of functions in a system without considering how these functions should be carried out. It therefore enables exploration.

Exercises

1. Concepts. Explain the difference between verification and validation.
2. Problem statements. Derive solution-neutral problem statements, ranging from low abstraction to high abstraction, for the following design tasks: 1) an updated model of a coffee maker for domestic use; 2) a mechanism for detecting falls among patients in a hospital environment; and 3) a wearable device that measures the user's pulse and communicates the readings to the user.
3. Application case. An existing simple printer design allows the user to print by connecting the printer to a computer via a Universal Serial Bus (USB) port. The printer draws power from an AC mains electrical socket. The printer has a manual paper feed, status LED indicators and buttons for turning it on and off and for printing a test page.
    a) Provide a solution-neutral problem statement for the design of the printer.
    b) Identify the overall function structures, including their functional elements and the flows of energy, materials and signals.
    c) Distill a modular product architecture for the printer by modularising the function structures identified in (b).
    d) A competitor has introduced a new printer which allows a wireless connection to the computer and contains a paper magazine, allowing the printer to print multiple pages without manual loading. Create a morphological chart to carry out a competitive analysis against this printer and identify solution principles and conceptual designs that may ensure a redesign of the original printer can result in a competitive product.

37. Safety and Risk

An interactive system should be safe; that is, it should not harm its users or people depending on it. What we mean by safety depends on the application domain. In human factors and HCI research, safety can mean:

- Safety from death. Example: human error in the operation of a medical device that administers medicine to a patient's blood. Medical error is a leading cause of death [498], and it is associated with errors at the user interface.
- Safety from physical harm. Example: physical injuries caused by machinery at factories is often associated with errors associated to inattention, fatigue, or multitasking during operation.
- Safety from economic or social harm. Example: personal data should not be inadvertently leaked to outsiders.

Safety is a wickedly challenging engineering objective. This is due to many factors, but perhaps most importantly, understanding rare events that can cause harm requires a thorough understanding of users, users' tasks and workflows, operating environments and the systems themselves. Safety a complex *systems problem* that requires a systems approach (Chapter 35).

In this chapter we discuss the concept of safety and methods for ensuring a system is safe to use. A first important realization is that for us to ensure a system is safe we need to agree on a *level of safety*: no system can be proven to be completely safe. At a high level, the level of safety depends on the frequency of occurrence and severity of an incident. We next need to gauge factors that affect the level safety. What is the frequency of occurrence of possibly adverse events? Does it range from improbable to daily? And what is the severity, does it range from minor injury to death? Depending on the system, it may be unavoidable to avoid harm or even death at some probable level of occurrence.

To answer these questions, we need to understand 'the human factor' in systems safety. Cook [168] summarizes these in his book *How Complex Systems Fail.* First, *all* interesting systems are complex, because they need to handle different contingencies associated with things people value. This pertains to systems related to physical safety, such as those in healthcare, transportation, and manufacturing. But it also pertains to systems in finance (e.g., banking), politics (e.g,. voting machines), personal data (e.g., tax data), and even entertainment (e.g., gaming avatars). Second, complex systems are heavily defended against failure. For an accident to happen, multiple failures must take place. Third, human users have a critical ole in building those defenses via their actions. They adopt safety procedures, take training and safety certificates, and so on. Humans are not passive actors in complex systems and therefore in their safety. Fourth, because of the complexity, practitioner actions are gambles. All actions are taken under uncertainty. A driver turning on an automated lanekeeping feature in car takes a risk; there is no guarantee that can be given to its operation under all circumstances the car can enter. Fifth, because all of this, attributing cause in the case of accidents is hard. Human error is rarely the sole culprit. It is just one cause in a complex network of factors including organizational, managerial, and design-related factors.

Questions of safety are serious and deeply ethical. At the extreme, safety engineering needs to calculate with factors associated with death. When designing systems that are inherently dangerous, such as training facilities for firefighters, how often are people 'allowed' to die? Safety thereby becomes a management problem—the management of risk. At a high level, the fundamental strategies of managing risk are to transfer the risk somewhere else, avoid the risk altogether, reduce the negative effect of the risk, or accept some or all consequences of a risk. At the same time, safety becomes an ethical question: what is the acceptability of harm, to whom, and how much is one willing to invest to prevent it? While methods we learn in this chapter do not help answer the ethical question, they do help provide the necessary information to take them.

A central concept when reasoning about safety and risk is the *system boundary.* The system boundary defines the system that will be considered. Importantly, in nearly every case, the system boundary extends *beyond* a user interface or interactive device. At the very least, the system boundary will include the user, but frequently further elements, such as the operating environment, any training facilities provided for the user, other people the user will need to interact with, and so on.

As we will see, in risk assessment the setting of a system boundary is fundamental as it defines what will be risk assessed—anything outside the system boundary is not considered. This both serves to make risk assessment tractable and inclusive of any relevant elements that can affect the risk of the system within the boundary.

For example, an infusion pump is a medical device that delivers a fluid into a patient's body. Infusion pumps have settings that control dosage. A nurse may use a calculator to calculate the correct dosage, for example, a calculator app on the nurse's mobile phone. The calculator interface of this app can now be said to be a system. However, for risk assessment purposes we will need to include the app, the mobile phone, the nurse, the patient, the infusion pump, and possibly the hospital context as well, for example, if the environment provides distractions that may overload the nurse when carrying out calculations.

Risks are pervasive in complex systems. They emerge from uncertainty in, for example, project management, financial markets, legal liabilities, regulation, design, unexpected usage patterns of a product or system, or deliberate attacks. It follows that, in addition to physical risk, there are many other types of risk, such as performance risk, management risk, development risks, commercial risks, service risks and external risks, for instance, new regulation or unexpected market competitors.

The term *risk* has many definitions. Here we will view risk from the point of view of expected system behavior. A system has a certain purpose, such as allowing users to quickly and accurately enter text, manage their information, allow them to stay in touch with their relatives, etc. The possibility that the system does *not* behave as expected can give rise to *undesired behavior*. The risk of an incident that results in undesired behavior of the system can then be viewed as the expected value of undesired system behavior:

$$risk = likelihood \cdot impact, \tag{37.1}$$

where the *likelihood* is the probability of a specific incident and the *impact* is the expected loss due to this incident. This definition restricts incidents to be binary—they either happen or they do not happen.

In this chapter we will learn how to manage risks using a variety of techniques that allow us to first identify and assess risks and then minimize, monitor, and control the probabilities of undesired events.

Paper Example 37.0.1 : Usability-related risks in electronic voting.

Usability can be a factor associated with risk. example, electronic voting systems must be usable by all citizens eligible to vote regardless of age, disability, level of education, poverty status, etc. The design of voting technology can influence the outcomes in elections and people's willingness to accept election results.

In this example paper, a range of usability issues were discovered when studying electronic voting systems that allow users to vote using touchscreens [58]. The paper compared electronic voting systems, such as the one used in the controversial 2000 presidential election in the U.S. Examples of such voting machines are shown in the figure below.

The paper illustrates two important issues where concepts from the safety and risk literature can assist further analysis. The first is the importance of setting the system boundary of the voting system, which does not only include the voting machine, but also the voter, the voting environment, and the system in place to ensure voting integrity. As a simple example, voters are exposed to an unfamiliar system in an intimidating setting where other voters impatiently wait for their turn to vote. Thus the system boundary needs to capture the voting environment in order to assess the risks of the user failing to achieve their goal in voting for their preferred choices.

The second issue is that such systems are inherently complex and as a result it is unsurprising that they exhibit a large number of usability problems. Given their critical importance in society, voting technology should be systematically risk assessed to minimize the probability that a voter is unable to vote on their preferred choices.

A *hazard* is the possibility of an object, situation, information, or energy to cause an adverse effect. For example, a sharp edge in the moulding of a wearable device can cause a cut on the user and a confusingly labeled button can cause users to accidentally delete data.

Exposure is the likely extent the user (or other adversely affected agents in the system) are exposed, or can be influenced by, the hazard. For example, if the sharp edge in the moulding of the wearable device is covered by rubber coating there may be no exposure of the hazard to the user. A confusingly labeled button that can cause a user to accidentally delete data would have limited exposure if access to the button is restricted.

For a risk to exist there must be *both* a hazard *and* exposure to the hazard. An unexposed hazard is not a risk. A risk is the product of the likelihood of an incident and the impact of the incident. A risk can therefore take on a range of values. However, a risk is always preconditioned on the presence of a hazard.

However, sometimes it is exceedingly difficult to anticipate the consequences of future systems as such systems may give rise to unanticipated outcomes. Consider for example a a popular social network that uses machine learning algorithms to recommend news story for its users to discuss with their friends. Such a system can easily reinforce what is referred to as "filter bubbles" that may amplify a group's perception, which may lead to users being convinced about conspiracy theories or otherwise become radicalized. The *precautionary principle* states that there is a social responsibility to protect users from significant harm if there is a risk of such harm happening.

The rest of this chapter will explain approaches and methods for both understanding safety and risk and mitigating them. We begin by examining the nature of human error and reflect on the problem of attributing cause or fault when accidents happen. We will then introduce risk management and see that it not only involves analyzing risks, but also how to monitor and control risk. Since there are many risk assessment methods and these methods have been developed for particular purposes, we will look at a selection of risk assessment methods that are useful for analyzing interactive systems from different perspectives, in particular in terms identifying, communicating, and assessing risks. We will also learn about a few tactics for thinking about risk in design. Finally, we will treat possible failures of components and systems statistically using a framework known as reliability engineering. We will consider simple series and parallel systems and we will see that if have data on failures we can calculate the reliability of a system, subsystem, or component, which can both serve to inform design decisions and help with assessing risks in a system.

37.1. Human error

Perfect users would never make errors. So why is that users make them? Norman [579] argues many human errors involve tasks that rely on short-term memory, such as entering numerical values into a device, having just been told the values by another person. He presented an early account of human error analysis based on four limitations of human short-term memory (see Chapter 5 for a detailed account of human memory): (1) the capacity of short-term memory is limited, typically between five and ten items; (2) the rate to search information in short-term memory is also limited, to approximately 100 ms per item; (3) there are only a few main types of encoding—verbal, motor, pictorial, spatial—, and they engage separate short-term memory functions; and 4) rehearsal is needed to retain information in short-term memory; other activities can interfere with rehearsal. Another cognitive cause for human error is attention: we are not able to share attention nor focus as much as we want.

However, human error is not sufficient nor necessary for accidents [579]. Accidents can happen without errors and many errors do not lead to accidents. Further, when errors do lead to accidents then such accidents are usually the case of multiple factors that can be difficult, and sometimes impossible, to untangle. However, the errors themselves can be considered. Beyond system induced errors, Norman [579] proposes two types of human error.

The first type of human error is *mistakes*. Mistakes are about the formation of an intention. That is, the user finds themselves in some form of situation, performs a situational analysis, and decides to take some form of action. The second type of human error is *slips*. Once the user has formed an intention the user needs to execute this intention, that is, take an action. Slips are errors in the execution of an intention.

If the user has a lack of knowledge, for instance, due to no or little experience of a system, or a lack of training, then this can give rise to both mistakes and slips. Mistakes can happen because the inexperienced user forms the wrong intentions due to a lack of knowledge of the interactive system. Slips may also happen due to the inexperienced user being unable to correctly carry out actions after having formed the correct intention. Such mistakes and slips can, in theory, be mitigated by training users. However, even an expert user can have a lack of knowledge because of incomplete knowledge of a situation because there are errors in the system or because the user is overloaded. Such errors are typically mistakes and such errors can, in principle, be mitigated by system redesigns.

However, even when the user has full knowledge mistakes and slips happen. For example, a user may fail to correctly identify the situation they are finding themselves in, or they may simply take incorrect decisions. Such errors can be induced by the system the user is working with, or within, which places demands on the user, such as task demands or environmental demands. It also possible that users make the correct intentions but fail to carry these out correctly, resulting in slips.

Norman [579] notes that there is a fundamental problem of attributing a definitive cause or fault when an accident has happened. Instead, accidents typically occur because there is some cascade of factors that interact that may relate to social, environmental, or human factors-related aspects of the system, its environment, and the task users are carrying out.

37.1.1. Skills, rules, and knowledge

A few years later, a highly influential paper entitled *Skills, Rules, and Knowledge; Signals, Signs, and Symbols, and Other Distinctions in Human Performance Models* by Rasmussen [670] presented a framework for understanding the performance of skilled users.

Rasmussen [670] states that there are three levels of control involved in the operation of complex systems. These levels are the skill-, rule-, and knowledge-based performance. A simplified model of how these level of performance interrelate is shown in Figure 37.1.

Skill-based behavior emerges following a user having an intention. Such behavior has high automaticity and happen without conscious control. Only occasionally does skill-based behavior rely on feedback control and in such cases the feedback is simple, such as for example a tracking task (see Chapter 4 for an in-depth discussion on motor

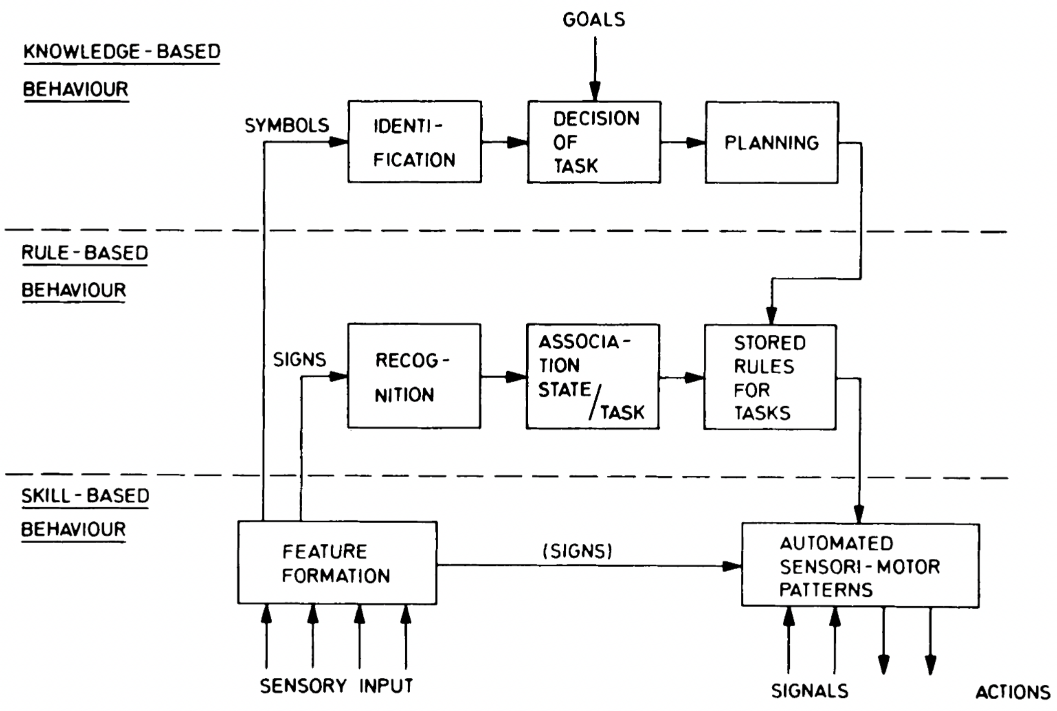


Figure 37.1.: Skill-, rule-, and knowledge-based control involved in the operation of complex systems [670]. The SRK model can be used to explain how human operators succeed and when they fail.

control). Automaticity allows users to free up cognitive resources to focus on higher-level aspects of the task at hand.

At the next level, rule-based behavior is characterized by the user employing stored procedures or rules of the type ”if X then Y”. Such rules can be learned or acquired by experience. Performance is goal-oriented but structured by the execution of a stored rule or procedure. The user’s selection of which rules to apply is governed by previous successful application of rules in similar situations. If feedback correction is required then this will require the user to functionally analyze the situation, which may rely on knowledge-based behavior, which is described next.

At the highest level, in knowledge-based control the user is faced with unfamiliar situations where the user has not developed any rules or knowledge for how to control the system. In such a situation the user explicitly formulates a goal and develops a plan to achieve this goal, which may be evolved and carried out in a trial-and-error fashion or by contemplating the consequences of various actions.

Rasmussen [670] also makes a distinction between signals, signs, and symbols. Signals are low-level continuous control signals that affect skill-based behavior. Such signals have no intrinsic meaning except as control signals. Signs represent perceived information that serves the purpose of guiding the user’s activation of predetermined actions or manipulations of the system. Signs therefore guide the selection or modification of rules for the sequence of actions necessary to carry out a skilled procedure. However, signs do not have any functional meaning and therefore cannot be used to generate new rules or reason about responses in an unfamiliar situation. Finally, symbols relate concepts to functional properties of the system or environment and can therefore be used to reason, for instance, to conceive a plan to achieve a goal. While signs are part of the external world outside the user’s brain, symbols are representations of the external world inside the user’s brain.

37.1.2. Taxonomies for understanding error

Reason [675] later contributed to an influential classification of human error. He classifies error types based their behavioral, contextual, and conceptual level: skill-based slips, rule-based mistakes, and knowledge-based mistakes, with the three error types corresponding to Rasmussen’s three levels of performance [670].

Reason [675] separates out slips and lapses, which are due to a failure to execute a particular plan, and mistakes, which are due to the plan itself being inadequate to achieve a goal. In other words, errors can either be intentional or unintentional. Intentional errors are when the user sets out to carry out the wrong action, such as inputting an incorrect parameter into a system. A mistake is when the user correctly carries out an incorrect action, such as when the user believes pressing the Escape key will exit an application.

Unintentional errors can be divided into slips and lapses. Slips are incorrect operations, such as mistakenly hitting a neighboring key on a keyboard. Lapses means the user is omitting a particular action necessary to reach the goal.

There is a large body of literature on human error. Given the importance of reducing human error, several design processes have been developed for this explicit purpose, most notable human factors engineering, which is discussed in Chapter 33).

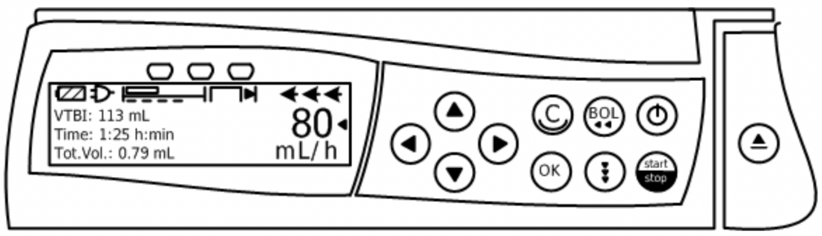
37.1.3. Analysis of human error

There are in-depth analysis techniques for understanding the underlying causes of accidents, such as *root cause analysis*. At a high-level, a root cause analysis consists of four steps: (1) arriving at an identification and description of the problem; (2) establishing a sequence of events relating contributory factors, the root cause, and the investigated problem; (3) distinguishing between root causes, causal factors and non-causal factors; and (4) establishing a path diagram linking the root cause to the problem through a sequence of events.

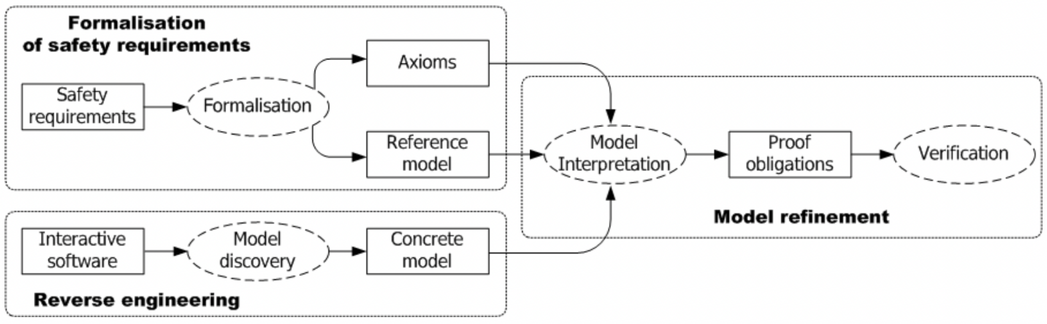
We will now discuss design methods that can be used to tackle safety and risk. We will begin by discussing how manage risk in the next section.

Paper Example 37.1.1 : Ensuring infusion pumps have safe user interfaces.

One way to help ensure a user interface is safe is to link safety requirements of the user interface to verification proofs of correct operation using a formal model. Masci et al. [507] shows how to fruitfully use this approach in the medical device domain. A patient controlled analgesia infusion pump is a method of pain relief where the user can administer a drug themselves using a medical device. An example of the user interface of such a device is shown below.



The paper first formalizes the United States Food and Drug Administration (FDA) requirements for infusion pumps. This in itself reveals ambiguities in these requirements. Thereafter the paper reverse engineer a formal model of an infusion pump product's user interface.



37.2. Risk management

Having an an overall understanding of human error, it is important to design systems to minimize risks. At its core, *risk management* is about identifying and assessing risks and minimizing, monitoring and controlling probabilities of undesired events. There are many approaches for managing risk and the precise risk management strategy will depend on the nature of the particular interactive system. At the high-level, risk management is a process with five steps:

Hazard identification Identify unintended system behavior that can cause unwanted outcomes.

Risk estimation Arrive at a risk by assessing the likelihood and severity of each hazard.

Risk evaluation Decide whether risks are acceptable.

Risk control Reduce unacceptable risks to acceptable levels.

Risk monitoring Manage a process that ensures acceptable risk levels are maintained throughout the system's lifetime.

Risk analysis is hazard identification and risk estimation. It involves identifying the intended purpose of the system and its intended use, identifying possible hazards, estimating the risks of these hazards.

Risk evaluation is about risk acceptability decisions, that is, deciding the level of risk that is going to be acceptable for the system.

Risk analysis and risk evaluation jointly form the part of risk management known as *risk assessment*. Many risk assessments exist for a variety of purposes. This chapter will review some of the more prominent risk assessment methods that may be suitable for assessing risk in interactive systems.

Risk control is about analyzing the options for risk management, such as changing the way a function is carried out to reduce risk, eliminate a function, change the system to introduce more redundancy, etc. Having identified options it is also necessary to implement suitable options and carry out a residual risk evaluation to determine the level of risk in the system after changes to the system. Finally, it is important to decide the overall level of risk designers are willing to accept for the system.

Risk monitoring is about continuously assessing the implemented system and its level of risk, taking appropriate actions as necessary. This step may also include reviews of outcomes, the experience, and reflections on the risk management process as a whole.

To assist risk assessment it is often helpful to perform *system mapping*, which as its name suggest is about creating maps of the system to understand its architecture, organization, the role of people and functions, and the flow of various signals and materials, such as, for example, paper forms in a hospital and emails in an office.

One very important aspect is to define the system boundary. This boundary helps everybody involved to understand the scope of risk assessment. To set a boundary it is typically necessary to describe a system that is more extensive than the system that is to be studied and then, based on an understanding of this extended system, it is now possible to set a boundary. Anything outside the boundary will not be considered in risk assessment and, conversely, anything inside the boundary *will* be considered. It is therefore very important to set the system boundary in such a way that factors that are likely to affect risk are in fact captured. Failing to do so means there are unassessed risks in the system, which can be very dangerous and give rise to catastrophic failures when the system is deployed.

System mapping is frequently carried out using various diagrammatic methods, such as task diagrams, system diagrams, or a communication diagram. We discuss various system mapping techniques in the chapter on systems (Chapter 35).

37.2.1. Risk assessment

Having identified the purpose of the system and its intended use, set the system boundary, and mapped out the system, we can now carry out risk assessment.

A large number of risk assessment methods exist that have been developed for various purposes, including medical devices, chemical process systems, aerospace systems, healthcare, etc. At a high level they can be considered along four dimensions. First, how in-depth they are, which affects the resources required to use them. Second, how much they focus on 1) identifying risks; 2) communicating risks; and 3) assessing risks.

Structured what-if technique

The Structured What-If Technique (SWIFT) is a team-based risk assessment method that prompts the team with *what-if* questions to stimulate thinking about identifying risks and hazards in a system. The focus on SWIFT is to allow the design team to explore different scenarios and contexts, and their resulting consequences, causes and impacts. Based on such discussions it is possible to arrive at hazards and assess risks, and ultimately risk controls.

SWIFT is based on a vocabulary which serves as prompts. The words in the vocabulary are used by a facilitator to discuss possible scenarios, issues, operating environment conditions, etc. that may give rise to hazards and risks. The words used as prompts typically focus on deviations, such as "failure to detect", "wrong message", "wrong time", "wrong delay", etc.

In practice, a SWIFT analysis is carried out by filling out a table where each row has a set of columns. The precise set of columns can be different depending on the application of SWIFT. Below is one example.

Identifier An identifier for the particular issue discussed.

What-if questions Questions triggering an assessment, such as "how much", "how many", etc.

Hazards and risks Any hazards and associated risks that may occur.

Relevant controls The controls that are in place or need to be in place to mitigate the risk.

Risk ranking The ranking of this risk relative to other risks identified in the exercise.

Action notes Next steps to be taken.

SWIFT can be fast and efficient with an experienced team with a good facilitator. It focuses on solutions, not just consequences of failure. At the same time, SWIFT is highly dependent on the experience of the team and facilitator, in particular as it focuses exclusively on the knowledge in the team. It produces mostly qualitative results and the results are difficult to audit as there is no formal structure to review the results against.

Compared to a method such as failure mode and effects analysis, which we describe next, SWIFT is a relatively lightweight method.

Failure mode and effects analysis

Similar to SWIFT, Failure Mode and Effects Analysis (FMEA) is a team based risk assessment technique. It can be used to analyze human error at both the individual and the team level. In general, FMEA analyzes component failures in the system. We will see later in this chapter how to model the failure of components mathematically when we discuss reliability.

FMEA considers each component's *failure modes* and assesses possible causes for failures, their likelihood and severity, the recovery steps available, and actions for eliminating or mitigating the consequence of the failure mode.

There are multiple ways of carrying out FMEA. Below follows an example of the columns that an FMEA may include.

Identifier An identifier for the particular issue.

Component The component assessed.

Failure mode The specific failure mode of the component.

Causes The possible causes of the failure mode.

Probability The probability of the failure mode. This is an estimate. It may be an actual probability if the failure rate for the component is available. Otherwise, it may be rating between, for example, 1–5 indicating a range between extremely unlikely to almost inevitable.

Severity The severity of the failure mode.

Risk The risk of the failure mode, in other words the product of probability and severity.

Recovery Steps to mitigate the failure mode, which may include requirements for recovery and mitigation.

Action notes Next steps to be taken, for instance, further investigation or changes to components.

An FMEA may include quantitative estimates of probability if, for example, actual failure rates of components are available. The outcome of an FMEA is a list of actions of required changes in design to mitigate risks.

Fault tree

A fault tree is a diagrammatic method for identifying and analyzing factors contributing to a fault—unintended behavior. Fault trees are, as their name indicates, tree diagrams linking factors to a fault using logical relationships, such as *and* and *or*.

A fault tree is created by starting with the fault as the top-level event and then progressively analyzing the factors that may contribute to the fault. Figure 37.2 is a schematic illustration of a fault tree.

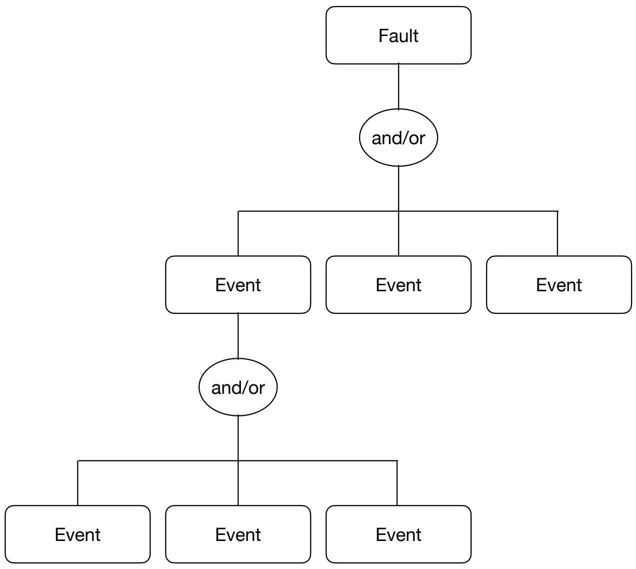


Figure 37.2.: A schematic illustration of a fault tree. The fault is decomposed into a series of events that may lead up the fault. The events are connected by logical relationships, such as *and* and *or*.

Fault tree can be used to analyze causes of human error. They highlight interrelationships between components, where components may be both system components and users.

Risk matrix

A risk matrix is a simple visualization technique for communicating risk. A risk matrix has two axes, impact and likelihood. Risks are indicated in the matrix accordingly.

A risk matrix makes it easy for a team to visualize important risks. In general, risks in the top-right are more severe and they indicate risks with a high impact and a high likelihood. Risk matrices are typically used in conjunction with other risk assessment methods to assist with ranking and prioritizing risks.

37.3. Reliability

Reliability is the quality of a system to *not fail*. An unreliable system gives rise to a high risk of failures, some of which can have catastrophic consequences.

Reliability engineering is a discipline that assesses the reliability of components and systems. Any component is bound to fail eventually and therefore it is important to estimate the probability of a component or system failure as a function of time. This is a statistical problem that at its core requires estimating a *reliability function* that models the probability of experiencing no failure as a function of time.

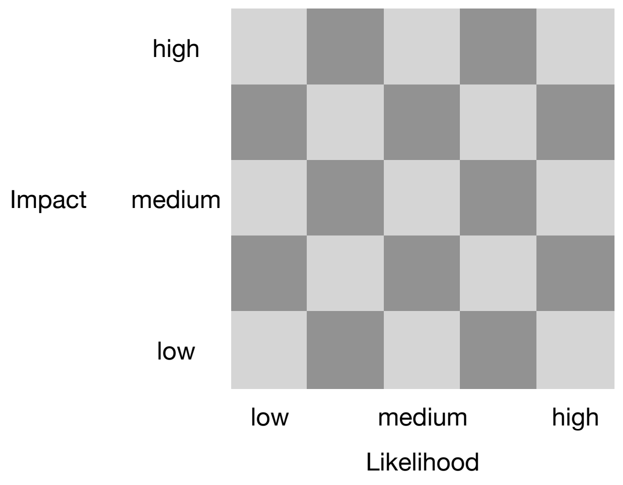


Figure 37.3.: An illustration of a risk matrix. Risks are indicated on the matrix in terms of their impact and likelihood. Progress in risk management can be visualized by observing risks to move from the top-right corner to the bottom-left.

A component, subsystem and system can refer to a machine, an algorithm, a user interface element, a physical control, a user, etc.

37.3.1. Series and parallel systems

To understand the reliability of any system, including an interactive system, we need to understand the topology of the system. That is, how the components in the overall system depend on each other in order to carry out the overall function.

The simplest system is a series system (Figure 37.4). For a series system to work, *all* components need to work. The traditional example of a series system is a decorative lights system wired in series. If a single light bulb burns out, current can no longer flow through the wire as the burnt out light bulb becomes an open circuit. As a result, all light bulbs in the entire light system stop working. In HCI, many systems are in fact series systems as system functions are frequently achieved by components carrying out tasks in a sequence, the current step in the sequence being reliant on correct execution of the previous step. An example of a series system in HCI is a typical wizard dialog that guides the user through a series step to assist the user to achieve a certain goal, such as to install an application. If such a wizard dialog is purely sequential then if any step fails in the wizard then the user will be unable to achieve their goal. To avoid such an outcome, the wizard must allow for some form of redundancy by providing some alternative means of progressing if any step in the sequence fails.

The opposite of a series system is a parallel system (Figure 37.5). In a parallel system the system works as long as one component is working. Consider a graphical indicator that monitors some system function, such as a printer having sufficient paper loaded.

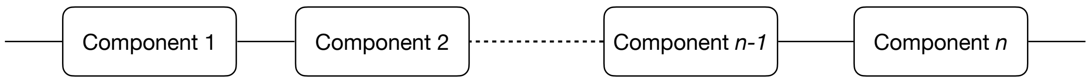


Figure 37.4.: A series system with n components. All components need to function for the system to be able to carry out its overall function.

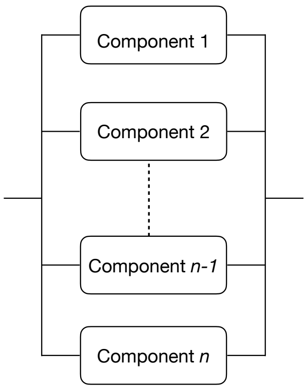


Figure 37.5.: A parallel system with n components. It is enough that one component functions for the system to be able to carry out its overall function.

If the indicator relies on a single sensor and this sensor fails then the system will fail to warn the user if the printer has insufficient paper loaded to print. However, if the printer uses two sensors then the system exhibits parallel redundancy: it is not sufficient for one sensor to fail for the system to fail to carry out its function—both sensors must fail. This example also hints at why parallel systems are not as common as they maybe should be: installing two sensors increases the complexity of the printer and the cost of manufacturing the printer. In HCI, a simple example of a parallel system is the way a user can delete a file on the desktop. The user may drag the file to the bin or the user can right-click the file and select delete. If the user does not know how to carry out one method, the other method suffices for this simple joint human-computer system to carry out its function of deleting a file. Again, the cost of parallel redundancy is that the system must support multiple means of achieving the user's goal.

Now let us consider series and parallel systems mathematically. Consider a series system with n components C_i. Now assume each component C_i has a certain probability of failure P_i and a corresponding *reliability* $R_i = 1 - P_i$.

A *series* topology means *all* components must work satisfactorily for the system to work correctly. The reliability R_{series} of a series topology is therefore n independent events of non-failure:

$$R_{series} = R_1 R_2, \ldots, R_{n-1}, R_n. \tag{37.2}$$

A *parallel* topology means *all* components must fail for the system to fail. The

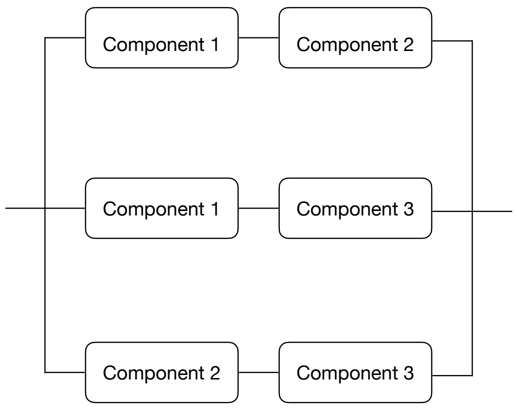


Figure 37.6.: An illustration of a 2-out-of-3 system. Note that the diagram describes the function but not the implementation of such a system.

probability of all n components failing is the product of the individual failure probabilities P_i: $P_1P_2, \ldots, P_{n-1}, P_n = (1 - R_1)(1 - R_2), \ldots, (1 - R_{n-1}), (1 - R_n)$. Therefore the reliability $R_{parallel}$ of a parallel topology is:

$$R_{parallel} = 1 - (1 - R_1)(1 - R_2), \ldots, (1 - R_{n-1}), (1 - R_n) \tag{37.3}$$

Using the expressions for R_{series} and $R_{parallel}$ we can now compare these two topologies for a simple case. Consider a rudimentary system with two components C_1 and C_2 with corresponding probabilities of failure $P_1 = P_2 = 0.1$. That is, each component has a 10% probability of failure for some chosen time interval. In a series topology the reliability $R_{series} = R_1R_2 = (1 - P_1)(1 - P_2) = 0.9 \cdot 0.9 = 0.81$. In a parallel topology the reliability $R_{parallel} = 1 - (1 - R_1)(1 - R_2) = 1 - P_1P_2 = 1 - 0.1 \cdot 0.1 = 0.99$. In other words, within the time interval, the probability is 99% that the system will succeed in carrying out its overall function if the components are in parallel. If the components are in series, the probability of success within the time period is reduced to 81%. This difference between the series and parallel system will only increase if we increase the number of components in the system. Assuming each component still has a $P_i = 0.1$ probability of failure, a series system with five components will have a reliability of 59.049% while a parallel system will have a reliability of 99.999%.

It is also possible to create systems in between all-series and all-parallel systems. For example, a k-out-of-n system is a system with n components that will function if at least k components function. For example, a 2-out-of-3 system for a sensor array may take sensor readings from three different sensors. If two out of three sensors agree then the system uses the sensor reading from the two sensors that agree and possibly trigger an alert that one sensor might be faulty. A series system is the special case when $k = n$ and a parallel system is the special case when $k = 1$. Figure 37.6 shows an example of such a system when $k = 2$ and $n = 3$.

Paper Example 37.3.1 : The Boeing 737 Max disaster

The Boeing 737 Max was an aircraft model that experienced catastrophic failures. An analysis of the failings of this model [807] explains both why it is dangerous to fully automate a function and leave the user completely out of the loop and why parallel redundancy, in the form of a user working alongside a machine, can be of uttermost importance in safety-critical system.

Boeing management wanted to increase the number of passenger seats and this meant increasing the size of the engines. However, to avoid the engines touching the ground, their shape needed to be changed as well. This led to change in the engine's centerline toward the plane's nose. This led to the plane pitch up and raise its nose when thrust is applied. In turn, this led to the possibility of the plane stalling in circumstances in which regular Boeing 737 would not stall.

Instead of rethinking the engine design, which is expensive, Boeing engineers added a software-side solution, which is inexpensive. This ensured that 737 Max could be branded and sold as "just another 737". After all there is just one added feature in comparison to 737. This also meant significant savings, as pilots required minimal retraining and pilot training is also expensive.

The software solution involved a system called MCAS (Maneuvering Characteristics Augmentation System), which "pushes the nose of the plane down when the system thinks the plane might exceed its angle-of-attack limits; it does so to avoid an aerodynamic stall." The pilot controls pushes or pulls something called control columns, essentially a joystick, to raise or lower the aircraft's nose. When the MCAS systems believes a stall is occurring it will autonomously instruct the aircraft to push the pilot's control columns forward and the pilot will not be able to regain control by pulling back. This was an explicit design decision. This decision was taken despite the fact that "a human pilot could just look out the windshield to confirm visually and directly that, no, the aircraft is not pitched up dangerously. That's the ultimate check and should go directly to the pilot's ultimate sovereignty. Unfortunately, the current implementation of MCAS denies that sovereignty. It denies the pilots the ability to respond to what's before their own eyes."

If a sensor read out was incorrect then the MCAS would take over and the results would be catastrophic as the pilot was left with no means to recover. There was no parallel redundancy in the form of a pilot being able to override the MCAS.

Automated systems that are designed to increase safety can easily backfire. Failures are bound to happen in any complex system and thus failures should be considered normal and there should be systems and procedures in place that allows for recovery.

37.3.2. Reliability and failure rate

By tracking failures it is possible to construct a histogram of failures as a function of the lifetime of a component or system. We can fit a probability density function $f(t)$ to this histogram. Now introduce a random variable T denoting waiting time until failure.

From $f(t)$ we can find the cumulative distribution function $F(t)$ by integrating $f(t)$.

$F(t) = \Pr\{T < t\}$ models the probability of failure in time t.

We can now find the *reliability function* $R(t) = 1 - F(t)$. $R(t) = \Pr\{T \geq t\}$. It models the probability of a non-failure in time t.

In general, *reliability* is defined as $1-$ the probability of failure. Reliability is a probability that a product, system or service will carry out its intended function without failure in time t. The *reliability function* $R(t)$ models the probability of no failure occurring before time t.

Failure rate is the frequency of failure of a product, system or service in terms of failures per time unit. It is often denoted λ. Often failure rates are modeled as constant. This makes sense because of a phenomenon known as the bathtub curve (Figure 37.7). Many systems experience an initial phase with early failures. This may be due to manufacturing errors of hardware, installation problems, users being inexperienced in using the product, etc. At the end of the lifespan of the system late failures start to increase as the system ages and experiences fatigue, critical software components increasingly become obsolete, new working practices means users tend to use the old system incorrectly, etc. During the entire lifespan of the system the system experiences random failures due to slips, arbitrary component failures, users dropping a device, etc. The net effect from these failure rates is an observed failure rate that resembles a bathtub. Many different systems experience this bathtub curve. Electronic devices, in particular, tend to closely, follow the bathtub curve. It is important to be aware that the bathtub curve does not model an individual system's observed failure rate. It models the observed failure rate of a population of systems.

Assuming the failure rate is constant then a useful, and very common, metric is mean time between failures (MTBF), which is defined as $1/\lambda$.

37.3.3. Hazard function

The hazard function $h(t)$ (sometimes written as $\lambda(t)$ and called the failure rate function) is the failure rate of the component at the next instant, *given* that the component has not failed up to time t. It is therefore a conditional probability density function:

$$\begin{aligned} h(t) =& \frac{\Pr\{t \leq T < t + dt | T \geq t\}}{dt} \\ \Rightarrow& h(t)dt = f(t)dt/R(t) \\ \Rightarrow& h(t) = f(t)R(t) \end{aligned} \tag{37.4}$$

This is because of the following. First, $f(t)dt$ is the joint probability of T in the interval $t \leq T < t + dt$ and at the same time $T \geq t$, which is the same as being in the interval. Second, $R(t)$ is the marginal probability that $T \geq t$.

A particularly interesting and important case arises when the time t until a failure is exponentially distributed. The probability density function of failure is then of an exponential distribution:

$$f(t) = \lambda e^{-\lambda t}. \tag{37.5}$$

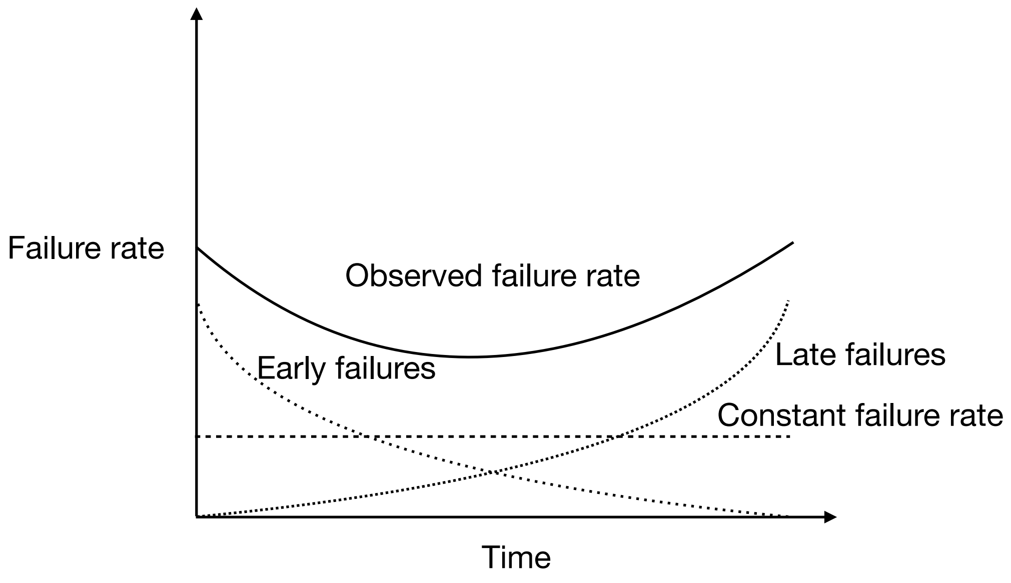


Figure 37.7.: An illustration of the phenomenon known as the bathtub curve. Many systems experience an initial phase of early failures and a final phase of late failures. During the lifespan of the system there random failures, which give rise to a constant failure rate. The observed failure rate due to these three failure rates is the observed failure rate, which resembles a bathtub.

The cumulative distribution function of an exponential distribution is $F(t) = 1 - e^{-\lambda t}$. The reliability function is then:

$$R(t) = 1 - F(t) = 1 - (1 - e^{-\lambda t}) = e^{-\lambda t} \tag{37.6}$$

Hence the hazard function is:

$$h(t) = \frac{f(t)}{R(t)} = \frac{\lambda e^{-\lambda t}}{e^{-\lambda t}} = \lambda \tag{37.7}$$

In other words, the hazard function is constant and the rate parameter λ of the exponential probability distribution completely determines the hazard function. This follows from the *memoryless* property of the exponential distribution: the remaining time until failure, given no failure in time t, does not depend on t. This models a constant failure rate in the bathtub curve.

The constant failure rate is very common in practice as it often suffices to models random failures due to accidents, mistakes, sudden component failures, such as a loose wire due to a user dropping a device, or other unforeseen events. For example, the probability a bicycle tire will experience a puncture due to hitting a nail on the road can be modeled using a constant failure rate.

For this reason, many systems are analyzed under the assumption of a constant hazard function. This simplifies calculations but it is important to understand that in complex systems there are interdependencies that may give rise to hazard functions that are not constant.

37.4. Security

Another aspect of safety and risk is *security*. Security is about ensuring systems are resilient to harm from other people. As such, security is closely related to risk management as security can be viewed as an aspect of risk. Examples of physical security includes fences, security guards, cameras, and so on. Examples of security involving computer systems are antivirus, cryptography, password systems, spam filters, anti-phishing mechanisms built into web browsers, as well as physical security, such as direct access to machines.

Usable security is a discipline intersecting HCI that focuses on ensuring computer security is usable by end-users. After all, a security system can only be as strong as its users. If users do not understand how to use security mechanisms in the right way, such as passwords or encryption, then there is effectively no security at all.

Whitten and Tygar [868] provide a definition of usable security:

> Security software is usable if the people who are expected to use it:
>
> 1. are reliably made aware of the security tasks they need to perform;
> 2. are able to figure out how to successfully perform those tasks;
> 3. don't make dangerous errors; and
> 4. are sufficiently comfortable with the interface to continue using it.

37. *Safety and Risk*

Paper Example 37.4.1 : Why Alice and Bob cannot encrypt

The availability of security technology does not guarantee that security is established. For this to happen users have to be able to achieve their goal of security and this is only possible if the interfaces that they have to interact with are usable. In the classic 1999 paper *Why Johnny Can't Encrypt: A Usability Evaluation of PGP 5.0*, Whitten and Tygar [868] explore the reasons why end-users are unable to use the graphical user interface for the encryption program Pretty Good Privacy (PGP) 5.0. PGP enables users to encrypt and decrypt text as a means for secure communication, for instance, by email.

The paper identifies five intrinsic properties of security that give rise to usability problems:

Unmotivated users Users are usually not interested in security because their goal in using a system is something else, such as sending an email, browsing the web, etc.

Abstraction Security is implemented by enforcing security policies that consist of abstract rules that end-users may find difficult to understand.

Lack of feedback It is difficult for security software to provide meaningful feedback to the user since (1) security configurations are complex; and (2) checking whether a security configuration is accurate is only possible if users know what they want.

Irreversibility When security fails it is typically not possible to take corrective action and return the previous protected state. For example, if a secret is unprotected then there is no way to know whether that secret is now in the hands of individuals external to the organization.

Weakest link A security system is only as strong as its weakest link. This means users cannot explore a security system using trial-by-error but instead must be carefully guided to ensure that security is maintained at every step.

The paper examines the usability of PGP through a cognitive walkthrough and a usability study. These methods are discussed in detail in the chapters on analytical evaluation methods (Chapter 41) and experiments (Chapter 43).

The cognitive walkthrough revealed that PGP failed to satisfy many of the properties in a usability standard for PGP established in the paper. Examples of problems were the use of the visual metaphors and the handling of irreverisble actions.

The user study asked 12 educated participants experienced with email to use PGP to send secure email. Only one third of the participants were able to correctly sign and encrypt an email. A quarter of the participants also exposed the secret key by inadvertently sending it in an unprotected email they believed had been correctly encrypted.

The paper concludes that while PGP uses a graphical user interface that is thoughtful in its design, it is ultimately only usable for people who understand public key cryptography and digital signatures. Following basic user interface design principles is insufficient to ensure usable security.

A central concept in security that of a *threat model*. Threat models are systematic analyses of the mechanisms of a security threat. A threat model answers a series of questions, such as: what kinds of agents might be interested in attacking the system?; what is the flow of data in the system?; which parts of the flow could an agent attack and how?; what is the likelihood of such attack succeeding?; what would be the impact of such attack?; how could the system be safeguarded against the attack? Human factors related to security (e.g., due to phishing) can be analyzed like this.

37.5. Are there risk-free systems?

When discussing safety and risk it may be natural for designers or stakeholders to envision a system that is completely safe, posing no risk of any adverse consequence to any user or other person explicitly or implicitly affected by it. However, such a stance is incorrect and not helpful. Any meaningful system will induce a level of risk. This is the reason for designers determining the level of acceptable risk in a system. Accidents and threats are inevitable in a system – they are normal [631]. It is better to design against them in advance than after the fact.

Summary

- Protecting people from harm and therefore ensuring interactive systems are safe to use is a central principle and core value in HCI.
- It is not possible to eliminate all risks in any system. Instead, designers must aim to design a system with an acceptable level of risk.
- Human error is typically just one of many factors involved in an accident.
- Risk management is a continuous activity that does not only involve assessing risks but also monitoring and controlling risks throughout the lifespan of a system.
- Reliability is a statistical property of a system. It depends on the system's topology and the individual probabilities of failure of individual components.
- Security is about ensuring systems are resilient to harm from other people. For security to be effective it needs to be usable.

Exercises

1. Methods. Determine the system boundary for the following cases: (1) an auto-correct algorithm for a touchscreen keyboard; (2) a 3D printer; (3) an app that allow children to share videos online with their friends; and (4) a smartwatch app that allow people to wirelessly transfer small amounts of money directly between different users of the app by simply holding two smartwatches close together. Which risk assessments methods would be useful for which case? Motivate your answer.

2. Human error. You have been asked to design a numeric keypad for a machine that delivers medicine to a patient. The numeric keypad allows nurses to input the exact doses of medicine to be administered. The keypad must provide the following 13 keys: `0--9`, `Set Dose`, `Administer Dose` and `Reset`. To provide a dose to a patient, the nurse is meant to first input `Set Dose` and then use the number keys `0-9` to input a number in the range [1, 999]. Thereafter the nurse can administer the dose by inputting `Administer Dose`. If the nurse makes an input mistake, the nurse can reset the process by inputting `Reset`, which allows the nurse to start over by first inputting `Set Dose` again.
   a) Identify the overall function of the system and identify its system boundary. Briefly explain the choice of system boundary.
   b) Identify three sources of human error permitted by the system. What types of human errors are they?
   c) Briefly discuss how the system can be augmented to mitigate these sources of human error.

3. Fault trees. A user is typing on a mobile phone using a touchscreen keyboard with auto-correct and word prediction. Draw a fault tree for the event that the user ends up typing a different set of characters for a word than the intended ones.

4. Failures. Assume a system with three components with each component having a constant failure rate. The probability P_i of a failure of an individual component i is independent of the other components and $P_1 = P_2 = P_3$. The mean time between failures (MTBF) for an individual component is 100,000 hours. Calculate the probability of the system failing within a 5-year period when (1) all three components wired in series; and (2) all three components are wired in parallel. You will discover the reliability is substantially higher for the parallel system. So why is not every interactive system, or any system in general, a parallel system?

5. An interactive hardware prototype has a subsystem that consists of n identical components configured for parallel redundancy.
   a) Give a definition of the hazard function and explain why it is a conditional failure rate.
   b) Assume each component has hazard function $h(t) = \lambda$, where λ is the failure rate. Derive an expression for the probability of a system failure in time t.
   c) Show that the hazard function $h(t)$ can be expressed as $h(t) = -\frac{d}{dt} lnR(t)$, where $R(t)$ is the reliability function.
   d) Now assume the n components of the system are changed into a different set of n identical components where each component has an identical hazard function that varies linearly with time with an intercept of zero. Derive an expression for the probability of a system failure in time t.

6. Case study. Consider a wearable ring device prototype that can be interacted with using a small analog pointing stick.
   a) Testing reveals that the probability density that the pointing stick fails to function at time t during its useful lifespan is:
   $$f(t) = (a + bt)\exp\left(-at - \frac{b}{2}t^2\right) \tag{37.8}$$
   where a and b are parameters. Derive the hazard function of this distribution.
   b) Does the hazard function found in (a) seem reasonable? Explain your reasoning, including any assumptions of parameter values.
   c) Further testing reveals that a subset of pointing sticks exhibit a Uniform probability density of failure at time t during their useful lifespans. Derive the hazard function for this distribution.
   d) Briefly discuss the reliability implications of the hazard function in (b) compared to the hazard function in (c), including any assumptions of parameter values.
   e) A user wears the ring device and interacts with it using the pointing stick. Derive an expression for the probability of failure of the pointing stick in time t due to the user dropping the ring on a hard surface. Explain your reasoning.

7. The book by Perrow *Normal Accidents: Living with High-Risk Technologies* [631] reports on a series of sociological analyses of accidents in tightly coupled systems with high complexities. The central thesis in the book is that accidents are normal in the sense that accidents are inevitable in highly coupled complex systems, in particular if they have catastrophic potential. This does not mean normal accidents are frequent: just as a human is expected to die at some point, a normal accident is considered a normal, albeit infrequent, event. (a) Explain why a tightly coupled complex system cannot be risk-free. (b) Two operators are tasked with monitoring the nuclear power plant. One operator is incapacitated due to tripping over a raised threshold and burning themselves with hot coffee in the process. The second operator is busy calling for medical assistance. At this time, the automated system has detected a deviating sensor reading indicating a reactor fault. However, as it so happens, this sensor reading is due to faulty sensor and not due to any actual reactor fault. Nonetheless, since the operators are both either incapacitated or busy, they do not take the actions the protocol dictates for this particular sensor reading. As a result, the automated system issues an alert. As a fail-safe, if the alert is not attending to by either of the two operators, the alert is propagated to a third operator situated in a different location with no possibility to assess whether the sensor reading is accurate or not. As a consequence, the third operator takes an action according to the protocol that results in a nuclear meltdown. State all hazards in this description and draw a fault tree that explains the sources leading to a catastrophic outcome.

   For the two following subtasks, see Chapter 20. (c) Consider the user interface design of a nuclear power plant. To avoid accidents, the user interface will incorporate redundancy in the form of an automated system that checks that the operators have followed the correct protocol and taken correct actions given sensor readings indicated through the user interface. If the automated system detects that the protocol is not followed, the automated system seizes control and proceeds to follow the protocol. What is the type and level of automation in this system? (d) It is discovered that it is possible that a sensor fault leads to the automated system in (c) to incorrectly seize control and apply an incorrect protocol. The system is subsequently modified to merely alert operators of an incorrect action and suggest corrective steps to ensure the appropriate protocol is followed. Suggest three principles from mixed-initiative interfaces that are particularly important for this task and explain how they apply to this particular design problem.

38. Software

Interactive systems ultimately rely on *software*. Software is what enables a design to be translated into an implementation. To ensure we implement software that fulfills the requirements set for it we need to apply *software engineering* principles.

Implementing software that is both correct and enables all desired interactions is challenging as there are many concerns, such as handling events, interfacing with external entities, such as network connections, and ensuring responsiveness and robust functioning during interaction. The book *The Mythical Man-Month* [99] famously pointed out that managing software development is inherently difficult and that – counter-intuitively – adding more developers to a project can actually delay it not speed it up [99].

Software engineering is fundamentally about working with systems from a few critical viewpoints. As we have already seen in the chapter on systems (Chapter 35), systems frequently consist of subsystems and one way to manage system complexity is by *abstraction*. This is the fundamental approach taken in user interface software—instead of a developer writing low-level interface code to explicitly read and work with digitized analog signals and low-level code that specifies the exact pixel contents and locations on a graphical display, the developer works at different high-level abstraction layers. This is achieved through *software design* and *software architecture* that allows developers to benefit from established programming paradigms, such as event-driven programming, and design patterns, such as model-view-controller. Further, common functionality for realizing user interface software is provided through *software libraries* and *toolkits* that enable developers to directly create sophisticated user interface elements, such as scrollpanes, and provide support for advanced actions, such as gesture recognition.

HCI research has tackled many of these aspects, for example:

- The *Amulet Environment* [558] is an early example of a user interface developer environment supporting cross-platform deployment across multiple operating systems. Amulet is a toolkit that provides support for building sophisticated user interface through constructs such as graphical objects, interactors, command objects, constraints, gesture recognition, and animation. This allows a developer to build, for example, a graphical interactive circuit designer tool in many fewer lines of code than using the standard software libraries.

- *KidSim* [761] is an end-user programming environment that allows children to create their own computer programs.

38.1. Software design and architecture

User interfaces need to be programmed. However, programming one from the ground up is difficult. There are many factors that must be taken into account. For example, managing user interface components, such as buttons, scrollbars, pull-down menus, etc. means the programmer has to both be able to draw these elements correctly on the screen and be able to detect and react to user input. This programming challenge is further exacerbated by the need to support many different possible designs, which may also rapidly change in light of additional user insight or design ideas.

To assist programmers in creating compelling user interfaces, several ideas, principles, design patterns and tools have gradually emerged as user interfaces have become increasingly sophisticated during the years. We here provide a high-level exposition of some of the key ideas in the area of user interface software that have achieved wide adoption.

Software design is a broad term that captures the entire process including all relevant concerns when arriving at software. This includes design concepts such as the level of modularization and abstraction. The structure of the software is referred to as *software architecture* and it is a component of software design. HCI-based user interfaces rely established software design practices and architectures for their operation.

A user interface reacts and responds to user input. As technology has evolved, such user interfaces have evolved from simple command line interfaces into sophisticated graphical user interfaces with various graphical elements users can interact with: windows, panels, pull-down menus, buttons, tables, plots, and other user interface components.

A modern GUI consists of certain expected elements: windows, panels, pull-down menus, and buttons, and other user interface controls. A GUI relies on rules implemented in software to manage these elements in such a way that the GUI behaves as expected. A GUI toolkit is set of software routines that allow a software developer to easily construct and manage a GUI.

One of the key concepts in a GUI is its *representation*: how user interface elements are represented as a model accessible to the programmer. The typical method is to represent a user interface component as an *object*. In object-oriented programming, an object maintains state and behavior. The state of an object represents the data it holds and the behavior of an object is represented by code. For example, a Button can have data indicating its size and location on the screen, its outline color, fill color, text color, text font, text size and the text itself. The code associated with a button can consist of code detecting when the button is pressed, changing the appearance of the button when it is pressed, and code notifying the rest of the program that a user has pushed the button.

38.1.1. Layout

The *layout* is a set of rules that determine where user interface components are displayed in relation to each other. The simplest set of layout rules is to have none: this means the programmer has to specify the exact pixel coordinates of each user interface component. For example, if a user interface designer desires a button to be placed at the top left corner of a panel then the programmer would set the button to be displayed such that the top

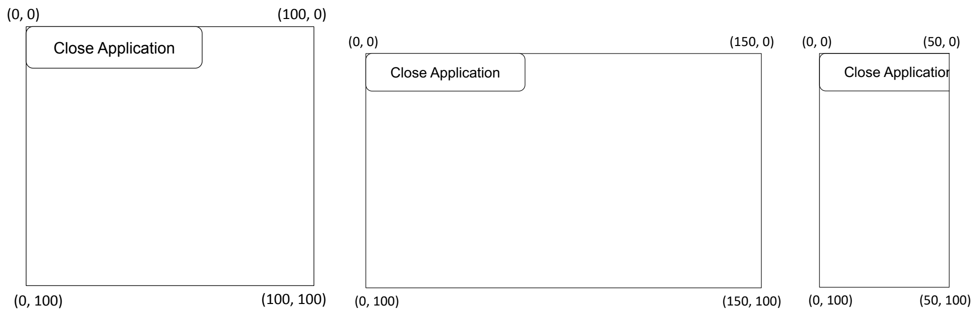


Figure 38.1.: An example of using no layout and instead setting exact pixel coordinates of each user interface component, in this case a `Close Application` button set to have its top left corner at $(0, 0)$, relative to the panel containing the button. When the panel is resized the button maintains its fixed location and size.

left corner of the button is in the origin of the coordinate system $(0, 0)$ (see Figure 38.1).

Such *absolute* positioning makes it possible to specify the precise location and size of each user interface component. This allows a programmer to implement a highly specific layout that is a very close to an intended design. However, this expressiveness comes at a cost: when the panel is resized, for example because the user has resized the application window, the absolute positioning of all user interface components in the panel remains in their exact original positions. Figure 38.1 shows how the location and size of the button is unchanged despite the panel changing its size. As a result, the button can get clipped or the panel can contain excessive amounts of unused space.

To solve such problems, various layout rules have been introduced. One such layout policy is to split the panel into five sections: north, south, west, east and the center (this layout is known as 'border layout' in the Java Swing GUI library). Figure 38.2 shows different variants building on this approach.

38.1.2. Event handling

Most of the time, a user interface is doing nothing more than waiting for the user to interact with it. This has led to the concept of *event-driven programming.* In event-driven programming, the program flow is driven by events. The overall program logic takes no action until an event triggers an action. Examples of such events include the user triggering a user interface control, for example, pushing a button, or a timer triggering after a preset time duration. Figure 38.3 presents an example of a *listener* listening to a button and its programmed response logic.

38.1.3. Model-view-controller and other design patterns

Software *design patterns* are reusable strategies or solutions to common programming problems. Several design patterns have been conceived in response to the difficulty

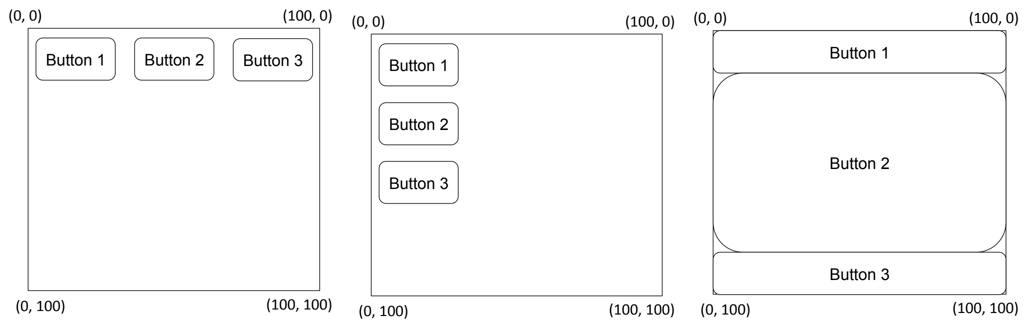


Figure 38.2.: Three buttons laid out in three separate ways depending on their layout. The leftmost figure shows three buttons laid out using a horizontal layout rule. The spacing between the buttons is a fixed distance parameter configured by the programmer. The width and height of each button are parameters unique to each button set by the programmer. In the figure the width and height of each button has been set to identical values. The layout rule respects these width and height parameters and will as a consequence render each button according to its individual width and height parameters. The middle figure shows the same layout rule applied vertically instead of horizontally. The rightmost figure shows a layout using compass rules. This layout splits the parent component into five regions: north, south, west, east and center.

in writing scalable and correct code for managing complex user interfaces. The most well-known design pattern, which we will here use to illustrate this concept, is probably *Model-View-Controller* (MVC).

MVC addresses the problem of decoupling three functional concerns of a typical graphical user interface [**?**]. The first concern is the *model* representing the data the user is manipulating. This may, for example, be text, a list of numbers or other objects, or a simple time series, such as $(0, 2), (1, 4), (2, 6), (2, 8)$. The second concern is the *view*, which defines what is presented to the user. The third is a controller.

MVC consists of three fundamental components:

Model The model contains the data.

View The view presents the data to the user. Note that there can be more than one view of the same model.

Controller The controller managers user interaction by accepting user input to manipulate the model and presenting model output to the user using one or several views.

38.2. Toolkits

Another design strategy is to provide an existing codebase at a higher level of abstraction designed specifically to make it easy to use it for user interface development. A *toolkit* is

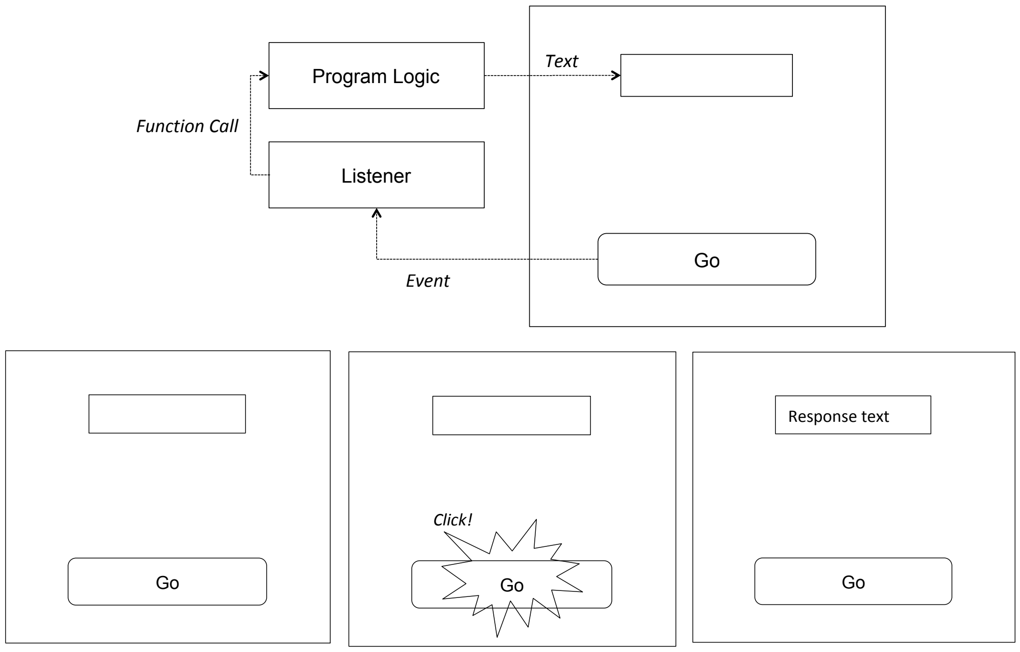


Figure 38.3.: An example of event driven programming. The button `Go` is monitored by a `Listener`. When the user triggers the button, this event is captured by the `Listener`, which results in the listener making a function call to code represented by `Program Logic`. In this example, the text “Response” is displayed inside the initially the empty label.

a software library defining commonly used widgets like menus, buttons, scroll bars etc. [554].

A toolkit can be called by an application. The advantake is consistency: all UIs using a toolkit will have a similar look-and-feel. Moreover, developers do not need to rewrite standard widgets. However, toolkits may not be easily extended.

38.3. End-user development

Software agents have the potential to assist users in achieving their goals. However, individual users may have individual goals. How should a user then instruct a software agent in order for it to be able to carry out that individual user's specific goal? This question gives rise to the *end-user programming problem* [761]: how can end-users, who are typically not programmers, tell a computer system what to do?

38.3.1. Macros

One approach is known as a *macro*. A macro is a collection of user actions that are grouped together into a program. The macro is recorded by system following some form of trigger, such as user action, which causes the system to monitor and store the user's actions. These actions are then form a list of actions as a program. The user can now rerun this program when desired. A system may also allow the user to inspect the macro and make changes to the individual user actions.

A macro thus allows a user to group a set of repetitive actions into a single action, thus potentially saving time and frustration in repeating mundane tasks. Macros are relatively easy for users to understand. The user merely has to trigger the system to record actions and then trigger the system to run the macro when desired.

However, macros also have disadvantages [177]. First, they only record low-level events, such as key presses, mouse clicks, specific pointer locations, etc. Second, macros do not *generalize*. A pure macro does not have a notion of semantics that matches user intent. For example, a macro may record the precise pixel location where a user clicks an icon and can thus trigger a subsequent action when the macro is played back by the user. However, if the icon is moved to another location on the display, or moved into another folder, then the macro will fail [177]. As another example, if a user selects the words following a prefix such as "Dear", as in selecting "Professor Oulasvirta" from the text "Dear Professor Oulasvirta" and selecting "Professor Antti Oulasvirta" from the text "Dear Professor Antti Oulasvirta", then the macro cannot automatically select "Professor Kasper Hornbæk" from "Dear Professor Kasper Hornbæk" as the macro is unable to generalize to select text in the form of all words following the prefix "Dear" [177].

These limitations of macros has led to the development of systems that are able to infer users' intended programs by *demonstration*, which is usually referred to as programming by example. It is worth noting that some interactive systems now provide system functions called "macros" that do not behave as pure macros and allow some generalization.

38.3.2. Programming by example

The limitations of pure macros lead to the idea of programming the computer by providing examples. This is in general known as *programming by example*. An early system is Pygmalion [759] from 1975, which allows a user to create sketches of programs using icons as representations of internal program mechanisms, such as variables, data structures, and program flow. The system does not infer a program but instead allowed a programmer to sketch the program interactively and have the system remember the actions, thus gradually generating program code. When the system has incomplete information on the program it traps program execution, prompting the programmer for more information. An example of a partial program for computing a factorial is shown in Figure 38.4. It is difficult to succinctly explain how a programmer can use Pygmalion to sketch and execute a program. For a complete worked example on sketching the function required to compute a factorial we refer the reader to [760].

A later example is *Tinker* [466], which is a learning environment that allows novice programmers to teach a software agent to program. The novice programmer provides illustrative examples and in response the system generates code that manages generalizations of the examples. The system relies on the user to provide additional information to do this. For example, if the user specifies more than one example for a function then the system asks the user to specify a test that it can use to distinguish the examples.

The prior examples of programming by demonstration were aimed to programmers. In contrast, the *Metamouse* [512] is a software agent called Basil that assists end-users in their tasks in a direct manipulation environment. The system monitors user activity in a drawing program (Figure 38.5). The software agent follows along the user's task, occasionally interrupts the user asking for a clarification, and eventually takes over the task. Figure 38.5 shows a complete example.

The Metamouse system induces a program by observing repetitive user behavior in a drawing program. An important design insight is that any software agent inducing a program for a repetitive task is going to be sensitive to user variation—a user performing the same thing using a variety of actions. The Metamouse system uses interactivity to reduce such variation by alternating between executing the induced program and building it. When the user rejects the next step in the program, or no step is available in the program, the system interrupts the user to ask for the next operation. In this sense, the Metamouse behaves as an "eager apprentice" [512].

Eager [177] is a software agent that monitors user actions in a graphical user interface. Eager automatically monitors user actions and checks for iterations. When an iteration is detected the system suggests the next action to the user (**??**). When Eager has shown a sufficient number of correctly predicted actions the user can feel confident Eager can automate the task. By pressing the Eager button the system then automates the task for the user. Unlike Metamouse, Eager cannot infer conditionals. On the other hand it relieves the user of several burdens: it does not require the user to (1) indicate the start of an example; (2) answer questions about generalization of the induced program; and (3) approve or reject each proposed action.

The design philosophy in Eager is to not burden the user by requiring confirmation of

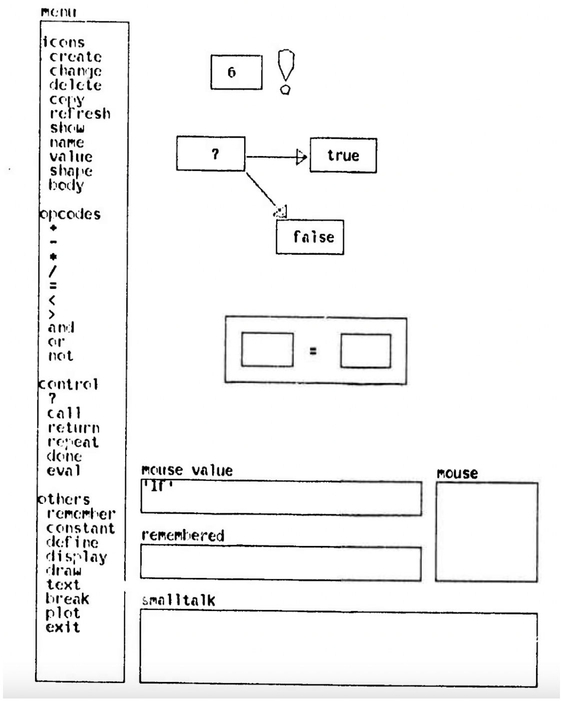


Figure 38.4.: A snapshot of a programmer sketching out a function for computing 6! in Pygmalion [759]. The boxes in the top-right represent icons. These icons represent aspects of the program. For example, the equality is represented as an icon containing two subicons. When the programmer drags a value, such as 6, to either subicon of this equality icon then the programmer is gradually completing the program. When both subicons have values assigned then the equality icon can evaluate the equality, given rise to a True or False value. These values can in turn be dragged to the conditional icon, represented as the icon with the question mark. Depending on the result of the conditional, either the icon marked *true* or the icon marked *false* will execute. Pygmalion attempts to execute code as soon as possible. However, whenever more information is required the system will trap this state and request the programmer for additional information. In this way Pygmalion allows the programmer to sketch out code, which is recorded by the system. However, note that the system does not actively infer any program from user behavior.

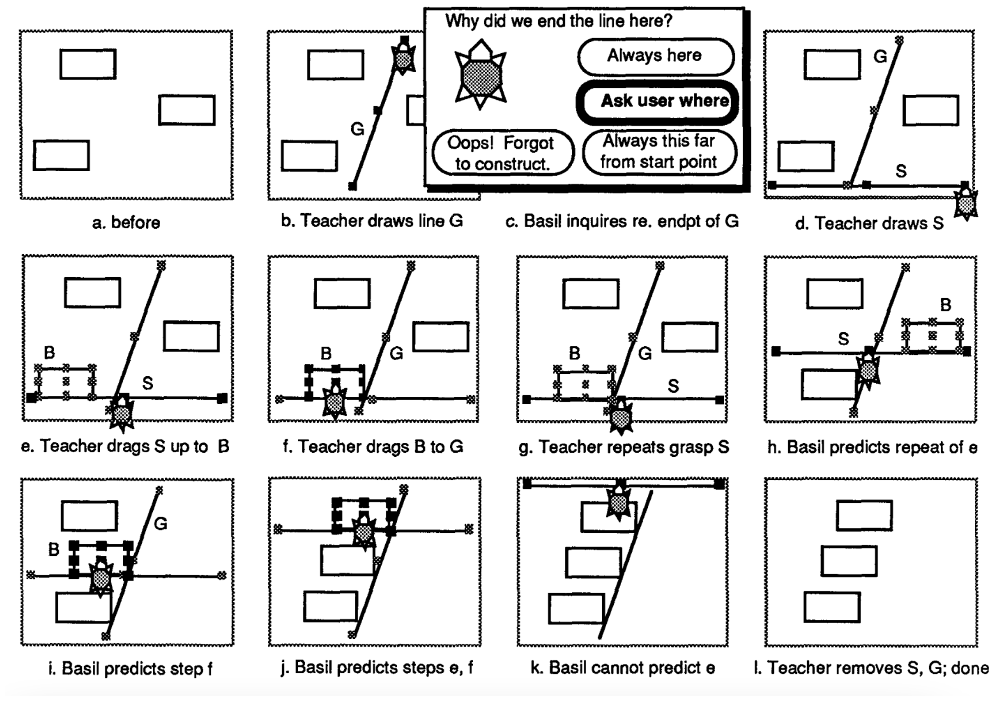

Why did we end the line here?
Always here
Ask user where
Oops! Forgot to construct.
Always this far from start point
G
S
B
a. before
b. Teacher draws line G
c. Basil inquires re. endpt of G
d. Teacher draws S
e. Teacher drags S up to B
f. Teacher drags B to G
g. Teacher repeats grasp S
h. Basil predicts repeat of e
i. Basil predicts step f
j. Basil predicts steps e, f
k. Basil cannot predict e
l. Teacher removes S, G; done

Figure 38.5.: An example of the Metamouse system's software agent called Basil, depicted as a turtle, learning to move rectangles by predefined constraints, as taught by the user [512]. The user (teacher) begins by drawing a reference line *G*. Since the end points of the line do not make contact with any shape the software agents asks the user if these locations are fixed or can change during runtime. The user indicates that they can change during runtime. The user then draws a second reference line *S* and moves the line so that it touches the bottom of rectangle *B*. The user then moves rectangle *B* to reference line *G*. When the user repeats the selection of reference line *S* the software agent Basil has detected a loop and consequently predicts that the user will want to move reference line *S* to the bottom of a new rectangle. The user accepts this suggestion. Basil subsequently wants to move the middle box to the left of reference line *G*, inferring this action by ranking the constraints and ignoring the weakest constraint, that of direction, in this instance. Subsequently Basil can automate the remaining alignment tasks until the reference line *S* ends up up beyond the final box. Having no new information Basil terminates.

each automation step. Instead, Eager learns implicitly that its prediction is incorrect by observing the user taking a different action than the action highlighted by Eager. As such, Eager is an example of a mixed-initiative interface in which the user takes initiative using direct manipulation to execute the program and thereby commence automation. The software agent takes initiative in building a program in the background and communicates its progress and current version of this program by highlighting the next element in the graphical user interface during interaction. Importantly, the user is not required to take any action should the user not desire any automation.

38.3.3. Visual programming languages

Another way to allow end-users to develop software is to change the medium of programming from a textual list of instructions to a graphical environment. *Visual programming languages* provide users with a graphical interface that allow users to create and modify programs by

Visual programming languages have long history. For example, Smith [759] reviews examples that were known in 1975, including a visual language based on a circuit metaphor by W.R. Sutherland that allows the user to use a lightpen to draw logic circuits.

It is also possible to combine a visual programming language with programming by demonstration. *KidSim* [761] introduces a graphical user interface that acts as a simulation environment that enables children to construct programs graphically. The system uses so-called graphical rewrite rules [259], also called before-after rules, to enable the user to specify behavior.

38.4. Formal methods

We have already concluded that software systems supporting interaction are inherently complex. So how do we know they behave correctly? This is

Another way to work with models is to use *formal methods*. This term refers to a set of methods use rigorous mathematical models that can be used to specify, develop, and verify a system.

38.4.1. Finite state machines

One example of a formal method that can be used in HCI is modeling using finite-state machines (sometimes called a finite state automata). Such finite-state machines can be used for a variety of applications. One usage in HCI is to use them to provide a design vocabulary of input devices. Buxton [114] presents a set of state diagrams modeling input (Figure 38.6).

Figure 38.6 (a) shows a state diagram for mouse operation. The input device is in State 1, *Tracking*, when it is tracking, that is, when the user is moving the mouse around a surface and thereby moving the mouse cursor around on the display. However, when the user pushes a button on the mouse this results in a state transition and the input device

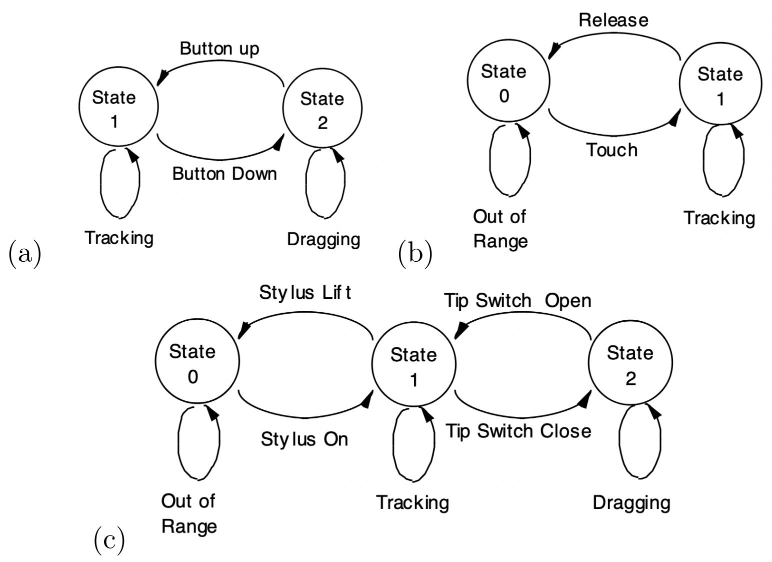


Figure 38.6.: Three state diagrams modeling input [430].

is now in its *Dragging* state. The input device will remain in this state until the button is released, thereby returning the input device to its *Tracking* state.

Figure 38.6 (b) shows a different state diagram for a touchscreen. Here there is a State 0, which denotes *Out of range*. When the user's finger is outside the range of the touchscreen, the device has notion of it. However, when the user is within the range, this registers as a touch and the touchscreen transitions from State 0 to State 1 and is now in its *Tracking* state. That is, moving the finger when the finger is within the range of the touchscreen results in, or at least can result in, a tracking cursor following the user's finger.

Note that while Figure 38.6 (a) and (b) look very similar they capture different state transitions. There is a difference between State 0 (out of range), State 1 (tracking) and State 2 (dragging). The mouse does not have a State 0 (out of range) as there is no notion of a mouse being out of range in this sense. Vice versa, the touchscreen does not have a notion of State 2 (dragging) as it is modeled as only supporting two states: registered finger touch (State 1; tracking) or no registered finger touch (State 0; out of range).

Thus both models capture interaction but different kinds of interaction for different types of input devices. Can we also consider an input devices that supports all three states—out of range, tracking, and dragging? Figure 38.6 (c) shows one such example. It is a stylus that is operated with in conjunction with a graphical tablet. The stylus can be moved above the tablet by the user. However, if the stylus is out of range (State 0) then the system is unaware of such movement. A state transition occurs when the stylus is registered by the tablet and the system is now in State 1 (tracking)—moving the stylus can now results in, for example, cursor movement being displayed on the tablet. However, if the user provides additional pressure on the stylus it is possible to sense this using, for example, a switch in the tip of the pen. This can then lead to a state transition from State 1 (tracking) to State 2 (dragging). In this sense, the stylus and the graphical tablet provides a richer interaction potential than the mouse or the touchscreen since it supports three rather than two states.

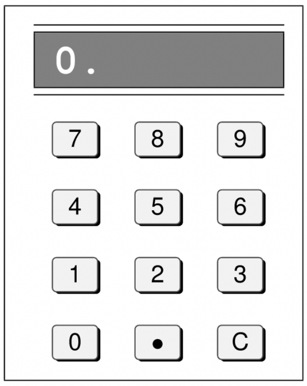
This is an example of using state diagrams to provide a design vocabulary. It allows us to discuss the expressiveness of, in this case, input devices, to compare them, and to consider whether we can add additional states. For example, is it possible to add a State 2 (dragging) for a touchscreen, and if so, how?

Paper Example 38.4.1 :

Designing safe number entry systems.

Data entry is pervasive and used in many critical environments, including healthcare, aviation, and finance. An analysis of number entry interfaces [800] found that such interfaces often contain design problems that give rise to errors.

The figure below shows a typical number interface from the paper (used under Creative Commons License). As the paper points out, the layout does not specify any interaction decisions. For instance, does the button `C` serve to make a correction, or does it cancel?



Thimbleby [800] identifies a large number of design problems in number entry interfaces, including unclear and unstated design requirements, unclear record keeping protocols of interaction steps, problems with interacting with negative numbers, problems with timeouts, etc. In total, the paper identifies 16 common design defects on numeric keypads.

Thimbleby [800] argues user testing is insufficient as it is not possible for user testing to guarantee coverage. Instead a formal approach is necessary. Thimbleby [800] demonstrates that it is possible to design formal rules for interaction with numeric keypads using Hoarle logic. An important insight is that many years of user-centered research has failed to identify easily rectified design defects. Thus, user-centered design for critical user interfaces should be complemented by formal approaches that can provide coverage to detect such design defects.

38.4.2. Formal modeling of undo

Other formal methods than state diagrams are possible, and sometimes more fruitful. As an example, let us consider the *undo* function that is present in many interactive systems. Undo allows users to revert the system state to a previous state, for instance, if the user has made an error or changed their mind.

However, the undo function is surprisingly complex. First, there are several ways of supporting undo in a system. The simplest model, called *flip undo*, simply allows a single level of undo and redo. In such a system the user may input a sequence of commands and the effect of the last command can be removed by invoking undo and re-added by invoking redo [196].

A more common implementation of undo, called *backtrack undo*, involves the system maintaining a record of every command that has been invoked. When the user invokes undo, the last command (excluding meta-commands, such as undo) is removed. When the user invokes undo again, then the next last command is removed, and so on. When the user issues a command and then all undo commands are forgotten [196].

A variant of backtrack undo, called *stack-based undo/redo*, further allows the user to *redo*, that is, to undo the effect of an undo. Such a system typically maintains a separate list of undone commands. When a redo is invoked the previously undone commands in this list are invoked again. This effectively allows a user to move back and forth in the state history of the system by repeatedly invoking undo and redo. When the user invokes an ordinary command then the list of undone commands are forgotten [196].

As you may now realize, implementing a correct undo policy is not as easy as it may initially seem. A straight-forward definition of undo is that if we invoke *undo* following a command c then the state of the system should be return such that command c effectively never happened. We can write this down in a formal way as follows:

$$c \frown \mathit{undo} \sim \mathit{null} \tag{38.1}$$

This notation specifies that the system state of a command c that is followed by ($\frown$) *undo* should be equivalent to ($\sim$) the system state of invoking a *null* command, that is, a command that does nothing. If we exclude *undo* from any valid command c (treating undo as a meta-command) then this property is called the *strong-cu* property.

However, we can also introduce a similar notion for undo commands, called the *strong-uu* property—that is, an *undo* followed by an *undo* should result in the same system state as invoking a *null* command:

$$\mathit{undo} \frown \mathit{undo} \sim \mathit{null} \tag{38.2}$$

The combination of the strong-cu and strong-uu properties seem intuitively appealing. However, they have been shown to be inconsistent. In fact, it can be proven that no undo system with more than two internal states can support both of these properties simultaneously.

This can be illustrated by drawing a state diagram, which we leave to the reader. In this case, we have three states, our initial state s_0, and states s_a and s_b, which arise after the user has invoked command a and b respectively. Now, the user can go back from state s_a or s_b to the initial state s_0 by invoking *undo*. In other words, $a \frown \mathit{undo} \sim \mathit{null}$, and $b \frown \mathit{undo} \sim \mathit{null}$. This follows from the strong-cu property that a command followed by undo results in the system state as if merely invoked the null command. That is, if undo follows a command we are back at our initial system state. However, what happens when the user thereafter invokes undo again? According to the strong-uu property this should result in the same system state as had we originally invoked the null command. In other words, $\mathit{undo} \frown \mathit{undo} \sim \mathit{null}$. However, if the diagram is correctly drawn, it can be seen that an undo followed by an undo is now inherently ambiguous. We may according to the strong-uu property either end up in state s_a or state s_b. In other words $s_a = s_b$. However, now consider that a and b are arbitrarily chosen commands. This means *any* command will result in the same effect in state s_0, which for any real system is obviously not the case. Hence, undo cannot be treated as an ordinary command. It must be seen as a meta-command with special properties.

Summary

- Software design and architecture enables user interfaces to be efficiently and effectively developed. A prominent design pattern for user interfaces is called model-view-controller.

- End-user development refers to techniques, approaches, and systems that allow end-users to designing and building executable programs.

- Formal methods enable designers to specify, develop, and verify user interfaces. Formal methods have wide uses, for instance in modeling undo functionality, but are most frequently used in critical applications.

Exercises

1. Consider the following systems: (1) a fall-detector device for elderly patients in a hospital; (2) a streaming music service app; (3) a virtual reality head-mounted display; and (4) a spreadsheet used by a professor working in a university to record exam marks and grades. Which systems are they embedded within and which systems do they have to interact with?

39. Computational Representations and Models

Previously in this book, we have explored a number of methods to design and engineer interaction. These approaches apply knowledge from user research, theories, technical and business requirements, designers' know-how, and so on, to create a system. In these processes, available data influences the final design *through the designer*. It is the designer who interprets the data and creates solutions based on their understanding.

In this chapter we consider a complementary perspective that allows computing appropriate solutions to well-defined problems in interaction. We *represent* key features of interaction computationally, pull several such features together to a *model*, and then *compute* on this model. Such *computational methods* can optimize interfaces, adapt them, make inferences about users, and learn from interaction. Examples include the following:

- Nonspeaking individuals with motor disabilities frequently rely on augmentative and alternative communication devices to communicate. Such devices provide users with a means to enter text that can then be read out to the speaking partner through a speech synthesizer. A common problem in this area are low communication rates. As a consequence, researchers have been investigating techniques to improve rates, including the use of word prediction, sentence retrieval, and sentence generation techniques that adapt to the user's own vocabulary over time. By creating a computational model of interaction it is possible to predict the performance of such systems before testing them in deployment studies spanning weeks or even months.

- Augmented reality glasses allow users to interact with digital information registered in the physical world. However, for such virtual content to be effective, the user interface that is used to manipulate them has to be salient to users (see Chapter 3). By creating a model of visual saliency for such user interfaces it is possible to optimize the presentation to make it maximally visible to users.

- Graphical user interface design is hard. Designers need to coordinate decisions between multiple levels from pixel-level (padding between buttons) to high-level decisions (functionalities offered to a user). Computational methods can assist designers by generating design ideas. These are often of two types: explorative ideas or ideas that refine an existing design. Under the hood, these methods model key features of the existing design and link it to variables of interest, such as those related to visual appeal or usability. They can algorithmically search for alternatives, thus complementing designers' creative effort.

What these examples have in common is that they need to *represent* interaction in order to create a *computational model.* Once such a model has been obtained it is possible to *compute* with it.

Models are at the centerpoint of such methods. Models are simplified representations of interactive systems that capture some essential aspects in some formalism that allows reasoning with it. Mathematics and computer science offer us a rich toolbox of formalisms to this end, each with different properties. In this book, we have learned about a number of models, such as saliency models using deep nets, motor control laws based on regression equations, and models of human cognition.

When we have a model, we can *predict*, for example, the performance of a system, before engaging in empirical user testing. For example, models of visual discrimination ability (see Chapter 3) are used for engineering the quality of displays. A model can also be used to *simulate* outcomes. Simulations allows us to learn about system behavior. We can detect risks and analyse the underlying causes. For example, models of pointing are used to analyze intelligent text entry methods. They can simulate how users type, including their errors, which helps then test methods like autocorrection. We can also use models to study the limits of behavior. Models allow us to chart the *operating envelopes* of user interfaces – the upper and lower bounds of performance with different (extreme but realistic) assumptions. In *sensitivity analysis*, we do this systematically by varying input to a model and study the effect on outcomes at the system level. 'What would happen to performance of a search engine if the queries were formulated by a non-native speaker with spelling mistakes?' Such analyses can guide us by telling which conditions are likely to hurt system performance. We will discuss the nature of models and explore how we can use such models to analyze system outcomes.

We can also use models to drive algorithmic improvement of designs. One of such methods is called *optimization.* This involves engaging in a systematic search process to obtain such values to parameters that allow desirable outcomes of the system. For example, a keyboard can be optimized by trying positioning the keys on the keyboard in such a way that the average predicted time it takes for a user to type the next key is minimized. Optimization can algorithmically generate a keyboard layout that maximizes entry rate. In this chapter we will explore the concept of optimization and demonstrate it in the case of keyboards.

Models can also be incorporated *into* interactive systems. Models can help infer user intent, make recommendations, and carry out actions automatically. Methods of *machine learning and pattern recognition* are used to create a model of interaction. Such model can enable new forms of interaction or enhance existing ones. For example, a speech recognition system receives data on the user's speech as an audio signals and uses a machine learning model to translate these observations into hypotheses about what the user wanted to communicate.

In this chapter we will discuss how to create models that allow us to perform prediction, envelope analysis, optimization, and inference. However, to do so we must first understand how to represent interaction in a format that enable us to create a model that can be subsequently used in computations.

39.1. Representations and models

A *computational model* allows the computer, or us as designers, to perform computations relevant for interaction. For example, a computational model of visual saliency for a user interface will allow us to maximize visual saliency, as described by the model, by finding the parameter settings that result in the model predicting maximum saliency. The "as described by the model" part is important as such a hypothetical model is only as good as the model is in accurately representing the relevant elements of interaction.

This leads us to *representations*. A key part in arriving at a good computational model of interaction is to identify a suitable *representation*. A representation here means a way to describe relevant elements of interaction. Such a representation can be simple or incredibly complex depending on the purpose of the modeling effort. In general, a computational representation is about identifying the most important *features* of interaction and a computational model is about a description of how these features interact to give rise to outcomes relevant for interaction.

Any computational model in HCI can be analyzed from this perspecctive. An example of a simple but popular model is Fitts' law, which we discuss in detail in Chapter 4:

$$MT = a + b \log_2 \left(\frac{D}{W} + 1 \right) \tag{39.1}$$

Briefly, Fitts' law predicts that the average movement time MT is a function of the distance D to a target and the width W of a target. In other words, the interaction we are studying is the average time it takes a user to press a button, select an icon, and so on.

Fitts' law *represents* interaction (pointing) is through three constructs: movement time, which is what we wish to calculate, and two *features* of interaction: (1) the distance to a target; and (2) its width. Note that there are other potential features that might be hypothetically relevant, such as the color of the target. However, according to the model, such additional features are not informative.

Having arrived at a representation we can now investigate a *model*. The model in Fitts' law has two components. The first is the construct *index of difficulty* (ID), which is usually defined as follows:

$$ID = \log_2 \left(\frac{D}{W} + 1 \right) \tag{39.2}$$

This construct captures the so-called "difficulty" required to select a target at a nominal distance D with nominal width W. Since the base of the logarithm is 2 the units are in bits. It implies causality in that the experimenter is meant to manipulate D and W, and hence create a distinct value of ID in order to influence average movement time. A higher difficulty implies an increased average movement time. The second construct in the model underpinning Fitts' law is that movement time MT varies linearly with index of difficulty (ID). This linear relationship can be written as

$$MT = \alpha + \beta ID \tag{39.3}$$

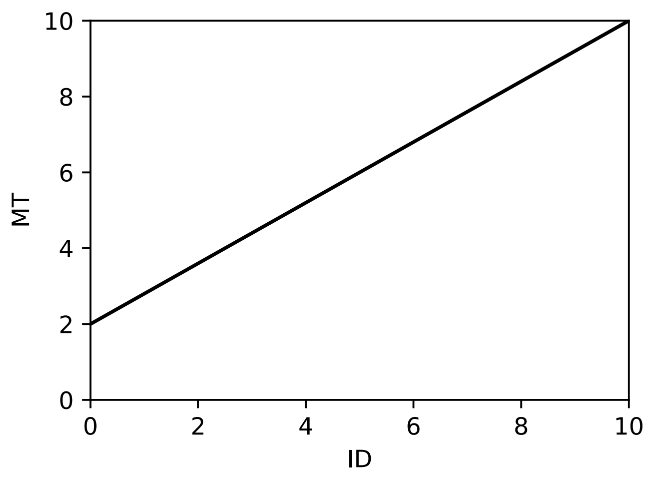


Figure 39.1.: Fitts' law is an example of a model that links movement time (MT) to an index of difficulty (ID). The equation underpinning the model is $MT = \alpha + \beta ID$.

Here α and β are parameters with α determining the intercept and β the slope. The linear relationship is shown in Figure 39.1.

Assuming α and β are controllable, that is, we are able set them to what we want, we could optimize movement time by attempting to set a and b to values that minimize MT. However, as it turns out, these parameters are task and device dependent and therefore not generally controllable. We cannot just set them to arbitrary values. Instead, we can minimize average movement time by minimizing D, that is, attempt to bring targets that the user are likely to hit in succession closer together, and by maximizing W, that is, trying to make targets the user is likely to want to hit as large as possible. In this way, Fitts' law can serve as a building in an optimizing process, allowing a designer to explore different user interface configurations in order to minimize average movement time.

In summary, Fitts' law is a model that relies on a representation of interaction: it represents the act of hitting a target as involving two fundamental controllable parameters: the distance to the target, and the width of the target. In particular, it does not, for example, involve additional elements, such as the color of the target in its representation, as such a representation is an irrelevant factor for determining movement time. Much of research is about understanding how to represent things and interaction is no different. If we want to infer users' actions or optimize an interface in a systematic way, we need a model to represent what we mean by those actions. Further, this model needs to be described in such a way that we can encode it into a format a computer can understand so we can build systems that can use such models.

39.1.1. Feature engineering

The first step to arrive at a computational model is to determine a suitable representation. Only when we know how to represent interaction we are able to model it. In Fitts' law we represent the act of a user attempting to point at a target by considering the target's width and the distance to it, and the average movement time required to hit the intended target. If we instead wish to come up with a model for human speech, we would desire a representation that is informative for that domain. In such a case, an expert might represent a waveform of human speech as Mel-frequency cepstral coefficients, which is a computational representation of human speech that models the human auditory system's response. In general, we typically want to identify the qualities, or features, of interaction that are useful for building a model of interaction.

A *feature* is a quality of interest to our representation. A way to help understand which features are important is a process called *feature selection*, sometimes referred to as *feature engineering.* Feature selection has traditional gained a lot of attention in pattern recognition and machine learning and several procedures exist for finding the most effective features in that domain.

Consider the problem of allowing users to issue commands by touchscreen gestures. Such a gesture recognition system translates a series of touch point observations into hypotheses about which gesture class the user intended to articulate. This is a *classification problem.* We have a set of labeled gestures in the system and we wish to assign a label to an unknown gesture articulation provided by the user.

One way to achieve this task is to use a Rubine recognizer [698]. This is a linear classifier treats a user's articulation of a gesture as a *feature vector* consisting of 11 features representing properties of a 2D single stroke gesture trace, such as the initial angle of the gesture, the length of the gesture, and so on. Collectively, these features result in an 11-dimensional feature vector. The Rubine recognizer can now be trained by showing it example gestures for each gesture class. For example, a gesture class that is meant to resembling a rectangle can be trained by showing the system a set of gesture articulations that represent this gesture. This step allows the Rubine recognizer to train weights for each of the features of the feature vector for each gesture class. At classification time, the recognizer extracts the same 11-dimensional feature vector from the user's gesture articulation and identifies the gesture class that is the closest to this feature vector. The Rubine recognizer is reliant on its representation of a gesture as an 11-dimensional feature vector. Because of the number of dimensions, it is hard to visualize the features. For this reason it is frequently useful to try to plot 2D and 3D visualizations to understand how such features work in practice.

More generally, features helpful for creating a representation may have one or several of the following properties:

Observable For a feature to be effective there must be a means to capture the feature by some measurement procedure. A directly observable feature is a feature that can be measured through a sensor or direct observation. An indirectly observable feature is a feature we can only capture by a proxy measurement. For example, a touch point an a capacitive touchscreen is directly observable feature while a user's emotive state is only indirectly observable.

Informative A feature should provide useful information about the interaction that is represented. For example, if the goal is to identify a computational representation about usable scatter plots then a useful feature would tell us something about the visual saliency of scatter plots.

Distinct A feature should provide complementary information in relation to other features. Features that have a very high correlation provide redundant information. For example, if the goal is to identify a computational representation of a

Computable For a feature to be effective it must be possible to arrive at the feature by some systematic procedure. This means it must be tractable to, for example, calculate a feature value or estimate it by some procedure. Further, even if a feature is computable it may still not be practical to use the feature if the computation takes a long time, is unreliable, or requires significant resources. For example, while it may be informative to have a feature representing users' energy expenditure when they perform fine-grained finger movement, such as a feature is very difficult to measure and hence compute.

Comprehensive Collectively, a set of features should be as complete as possible in forming a computational representation of the interaction of interest. That is, all relevant features that can be captured in practice and complement each other, should be part of the feature set. For example, a model of pointing is not comprehensive if it only considers target distance but not target width.

39.1.2. Modeling

Having arrived at a representation it is now possible to develop a *computational model*. Such models utilize representations to capture and compute on relevant qualities of interaction which can subsequently be used in a design, or to guide design.

To provide the intuition of models, let us consider two variables, x and y. A very simple model is then (identity):

$$y = x \tag{39.4}$$

This model is shown in Figure 39.2. This model is making a claim: the independent variable y takes on the same value as the dependent variable x. However, what if y is not related to x?

Consider the following model:

$$y = c \tag{39.5}$$

where c is a constant. Here y represents the *dependent* variable, the effect, and c is a parameter that can be set to any value, such as $c = 2$. This model is shown in Figure 39.3.

This model is also making a claim: the dependent variable y is unchanged as a function of the independent variable x. That is, the model claims y is invariant, or independent, of any value of x.

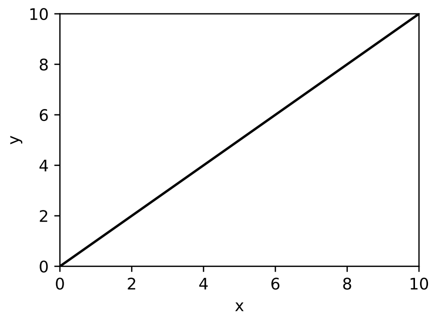


Figure 39.2.: The dependent variable y varies linearly with the independent variable x. The equation underpinning the model is $y = x$.

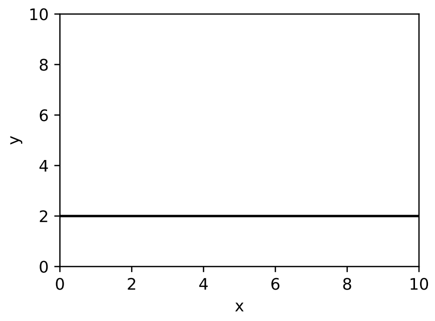


Figure 39.3.: The dependent variable y is invariant of the independent variable x. The equation underpinning the model is $y = c$, where c is a constant and $c = 2$.

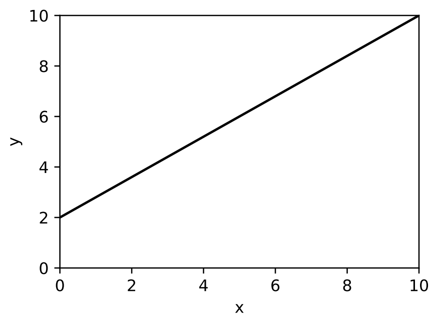


Figure 39.4.: The dependent variable y is varying linearly with the independent variable x. The intercept is $\alpha = 2$ and the slope is $\beta = 1.25$. The equation underpinning the model is $y = \alpha + \beta x$.

More commonly, our models do predict y varies as a function of x and the model makes a claim about this particular relationship. We previously saw Fits' law which makes a claim that movement time MT varies linearly with the index of difficulty ID for hitting a target. In general we can replace movement time MT, our independent variable, with y and index of difficulty ID, our dependent variable, with x. We then arrive at the following expression:

$$y = \alpha + \beta x \tag{39.6}$$

This model predicts y varies linearly with x and is shown in Figure 39.4. If it is a model of interaction then there is usually implied causality: we cannot directly control y but we have ways to change x. For example, in Fitts' law we cannot directly control average movement time. However, we can directly control how large the targets are and where they are located on a visual display.

In this equation we have model of interaction with three parameters: (1) α; (2) β; and (3) x. The independent variable x is a variable we control, perhaps by changing some quality in the user interface. The parameter α is here the intercept: it specifies the value of the dependent variable y when the dependent variable $x = 0$. In interaction is is often thought of as some form of offset. For example, if the dependent variable x represents task time then $x = 0$ is the point in the model when the user has not begun the task. The parameter β describes the slope of the line in Figure 39.4. This parameter frequently describes a *rate* because it informs us how much y changes when we change x. For example, if we assume y represents search time and x represents a measure of visual display clutter, then the rate β would tell us how much search time increases when the measure of visual display clutter increases. The intercept would in this instance provide us with an offset or baseline, as it would tell us what the predicted search time is when the measure of visual display clutter is zero.

More generally, models are functions—expressed as formulas or as computer programs– that produce an output given an input. They contain two types of mathematical constructs (1) parameters; and (2) mechanisms that link parameters together.

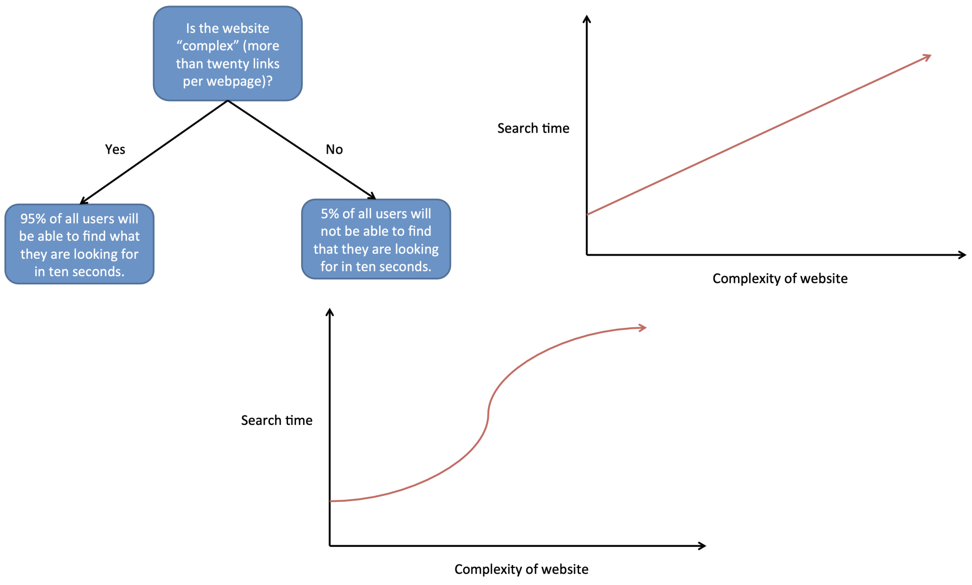

Is the website "complex" (more than twenty links per webpage)?
Yes
No
95% of all users will be able to find what they are looking for in ten seconds.
5% of all users will not be able to find that they are looking for in ten seconds.
Search time
Complexity of website
Search time
Complexity of website

Figure 39.5.: Three examples of simple models: a decision tree, a linear model, and a non-linear model.

To build a model, we require some formal mechanism that mediates inputs and outputs. Such a mechanism can be, for example, a regression equation, a deep neural network, a decision tree, a physical simulation, or something else.

These mechanisms work with variables which are typically referred to as *parameters*. Parameters express the values of some event or state. These parameters need to be updated with different inputs. For example, the parameters of a visual saliency model are millions of neural network parameters that are trained using eye-tracking data (see Chapter 3). When fed with an image a user might see (input), these parameters gain values (they are activated) in order to predict which regions of the image the user might attend (output). A different image gives rise to a different neural activation.

Problems in HCI can be modeled in different ways. Figure 39.5 illustrates this. It shows three different models for a toy task. We have a single parameter, the complexity of a website, which we define as the number of links on the page. We also have a single outcome, search time, which we will define as the time it takes a user to find the link they are looking for.

The first model in Figure 39.5 is using a a tree structure to model a binary outcome based on a threshold. If the complexity is above a threshold then the outcome is predicted to be either below or above a predefined value. This model is simple to understand but it is also very limited. First, the model is based on a threshold that needs to be set somehow. Second, the outcome is binary and lacks resolution.

The second model in Figure 39.5 is a linear model. It predicts the outcome as a function of the parameter value. As a model this is more informative. The slope of the line tells us how the model predicts search time increases as complexity increases. The intercept encodes a fixed cost in search time, it tells us that no matter the complexity the model predicts there will be an initial cost in search time. Unlike the tree-based model this model predicts a range of values in the outcome and it does not rely on a threshold to make this prediction.

The third model in Figure 39.5 is a non-linear model. It is similar to the linear model except the values of the outcome is not predicted to vary linearly with the values of the parameter. This property allows the non-linear model to make richer predictions. For example, unlike the linear model this model predicts that at a certain range of complexity the search time will rapidly increase.

The three models are simplistic. First, it is unlikely a single parameter, such as complexity, can fully determine search time. A useful model would at the very least need to capture complexity through several parameters, such as the number of links on the page, but also the number of additional elements, their organization, and so on. Second, the user's strategy in engaging with the system may also affect the outcome. A complete model would therefore not only have to account for the parameters of the system but also the parameters governing the user's interaction. While the three models may not be the most valid models, there may still be cases in which they could be useful. In engineering applications, accuracy and practicality are equally important.

By using more sophisticated models we can perform *simulations* of user interface outcomes. A simulation refers to the operation of a model, "the intention is to draw conclusions, qualitative or quantitative, about the behaviour or properties of a real-world process or system over time" [553]. In practice, simulations are executions of a computational model to approximate the behavior of an interactive system or part of. Simulations allow studying different conditions in human-computer interaction, including extreme and risky conditions. For example, we can simulate handover situations in a semiautonomous vehicle without actually risking anyone.

39.2. Envelope analysis

Models allow us to gain an understanding of the effect of parameter values on outcomes. As an example, suppose we wish to provide users with a touchscreen interface that allows users to correct speech recognition mistakes using a touch modality instead of a speech modality. We will use an example of such a design problem from the literature [**?**]. This research tackles this problem by providing the user with two methods to correct speech recognition utterances (Figure 39.6).

The speech recognizer generates a search lattice to identify the user's utterance. The system processes this speech lattice into a word confusion network. A word confusion network is a time-ordered series of clusters of candidate words along with their posterior probabilities.This network is shown in Figure 39.6. Every column consists of the following elements: (1) the current output word (which may be changed by the user); (2) the 1-best hypothesis, that is the speech recognizer's best guess on the user's intended word; (3) a series of alternative words ordered by their posterior probabilities from the cluster in the word confusion network at this location; and (4) a delete button that deletes the word.

The system supports the following actions:

Substituting words The user can substitute a word by touching an alternative word in the word confusion network. The user can also substitute multiple contiguous words by crossing them. The user can also touch a word which opens up a keyboard interface. In this interface the system shows morphological variants of the word, for example, morphological variants of the word "constitutional" in Figure 39.6 include "constitute", "constitutes", and "constitution".

Editing words The user can edit words by touching them in the top row or by double-tapping any other word. This opens up a keyboard interface that allows the user to either edit the word or type a new word.

Deleting words The user can delete words by touching the 'X' buttons in the bottom row. The user can also slide their finger and delete a sequence of words at the same time.

Inserting words The user can delete a word using three methods. First, the user can touch a word in the word confusion network and then drag that word to the top location. This is useful if the intended word is on the display but is not present in the correct column. Second, the user can touch the space between two words at the top which opens up a keyboard interface, allowing the user to type in a new word to occupy the space in between the two surrounding words. Third, the user can select a preceding word, which opens up the keyboard interface, and then type a space followed by a new word.

The different interfaces supporting these actions are shown in Figure 39.6. This system is on the surface fairly complex with many possible design choices for parameters. For example, how many clusters should the system display? If the system displays too few then there is a higher probability the user's intended word is not present and the user has to rely on the keyboard. On the other hand, if we show too many then the buttons will be small and difficult to select. Can we somehow quantify the effect this parameter setting will have on the desired outcome?

The researchers did this by means of simulation [831]. By essentially playing back recorded audio they are able to construct word confusion networks from the resulting speech lattices and then simulate different system parameters to assess their effects on the outcome. The outcome of interest in this case is word error rate, a lower word error rate is better. To simulate user behavior they used an oracle. An oracle is an assumed expert user that performs optimally in the interface.

Figure 39.7 (left) shows the result of one such analysis. The word error rate achieved by the oracle is shown as a function of the number of word alternatives that are provided in each column (cluster). The figure plots four performance curves for four different user configurations. We can observe that increasing the cluster size reduces word error rate, however the majority of the gain is realized by providing the first 3–4 word suggestions for a recognized word. This informs a trade-off design decision on the balance between

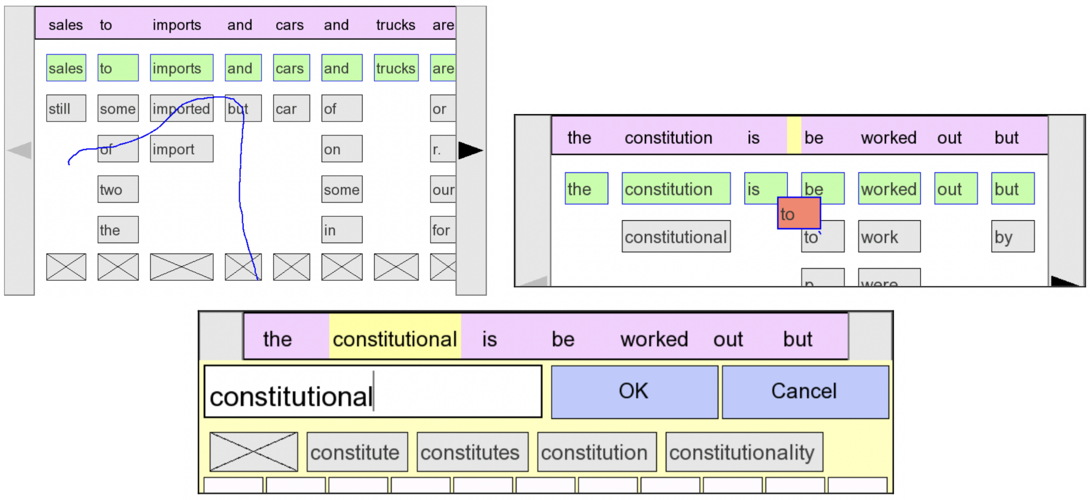

sales to imports and cars and trucks are
still some imported but car of or
of import on r.
two some our
the in for
the constitution is be worked out but
constitutional to work by
the constitutional is be worked out but
constitutional
OK
Cancel
constitute constitutes constitution constitutionality

Figure 39.6.: An example of a touchscreen-based mobile speech interface [831]. The interface allows the user to correct recognized utterances (shown at the top) by either direct manipulation of the word confusion network or by engaging with a keyboard that provides word prediction and a choice of morphological variants of words.

providing more word suggestions at the cost of having to make the buttons smaller. Adding the capability to delete a word reduces word error rate, and adding the copy option, and the ability of the user to touch a word and choose a morphological variant of the word directly reduces the word error rate further.

Figure 39.7 (right) investigates the reduction in word error rate as a function of the number of word predictions offered by the user. The constant line at the top represents a baseline which does not offer any word predictions. Since this baseline is independent of the number of word predictions offered, it is constant. We can observe reductions in word error rate are obtainable using a variety of word prediction methods. The highest reductions in word error rate is possible by using an acoustic model that predicts words that sound acoustically similar to the word. However, such acoustic predictions are difficult for a user to understand, for example, "aback" is acoustically similar to "attack" [831]. Similar reductions in word error rate are obtainable using statistical language models that take preceding or following words into account. However, such predictions depend on surrounding words and may be difficult for a user to understand. Finally, the lowest word error rate reduction is realized by using morphological variants. In the paper, the designers chose this option because although it did not achieve the highest word error rate reduction, these predictions are easier for users to understand [831], demonstrating that these analyses are guidance for design, rather than strict design directives.

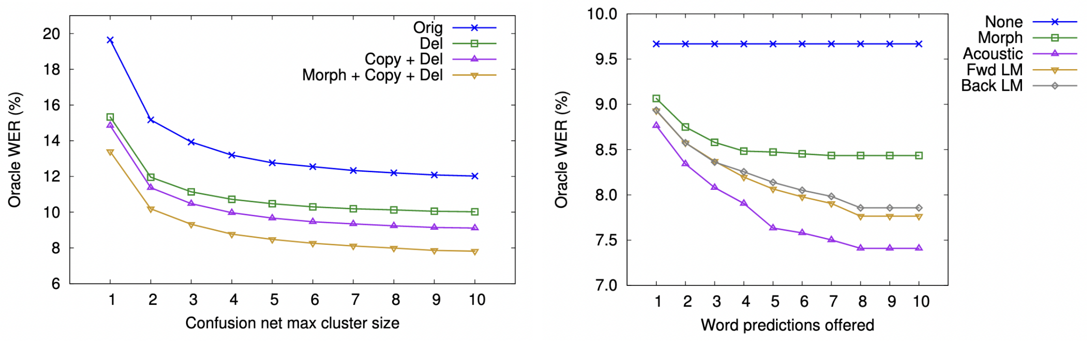


Figure 39.7.: Analyses of the speech recognition interface shown in Figure 39.6 generated by playing back recorded audio and then simulating a perfect user. This has resulted in the ability to plot the *oracle* word error rate as a function of a design parameter for several different interaction methods.

39.2.1. Carrying out envelope analysis

As the previous example shows, simulating outcomes and analyzing the effect on outcomes can be a very useful, complementary, method to understand HCI systems. There is no formal term for this type of analysis but we are going to call this *envelope analysis* (cf. flight envelope, back-of- the-envelope analysis, etc.). The idea is to extract design parameters from the functional description of the system and simulate potential performance by investigating a range of parameter choices.

We first have to consider the function parameters. Having identified a functional model it is possible to parameterize the model. There are fundamentally two classes of parameters:

Controllable parameters These govern function execution and can be set by the designer. These parameters are of great interest as they enable *optimization* towards design objectives.

Uncontrollable parameters These govern function execution and cannot be explicitly set by the designer. Understanding these is also possible as they allow us to carry out a *sensitivity analysis* to understand how sensitivity the outcomes to variations in these parameters.

It is also worth noting that the effects of these parameters can be investigated in various ways besides envelope analysis, such as in experiments with users, prototyping and measurements, etc. There are three common strategies: (1) keystroke level method and similar approaches; (2) Wizard of Oz methods; and (3) computational experiments.

One way to carry out envelope analysis is to use the *keystroke level method*. This method assumes error-free expert behavior and further assumes reliable fixed time estimates are available for all operators. This method, and other variants, is described in detail in the chapter on analytic evaluation methods (Chapter 41).

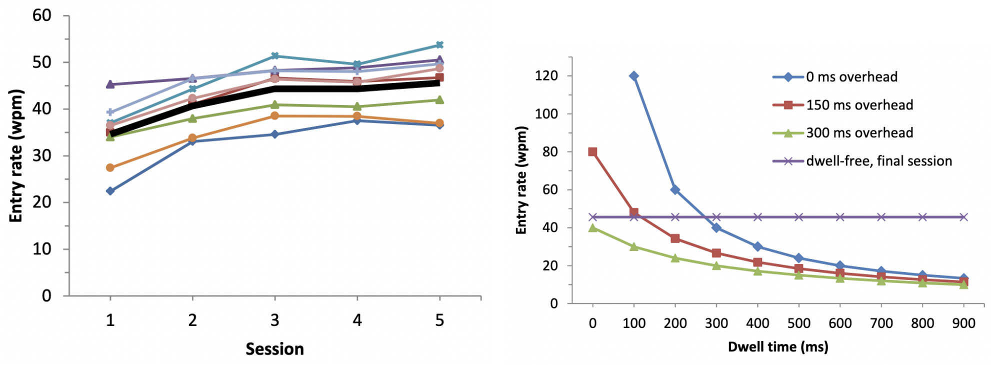


Figure 39.8.: The first plot show measured entry rate as a function of session for a dwell-free eye typing using a a simulated recognizer. The second plot shows different operating points of dwell-free eye typing. The entry rate varies as a function of two parameters: (1) the uncontrollable parameter overhead; and (2) the controllable parameter dwell time. For any realistic operating point involving these two parameters, dwell-free eye-typing is faster [430].

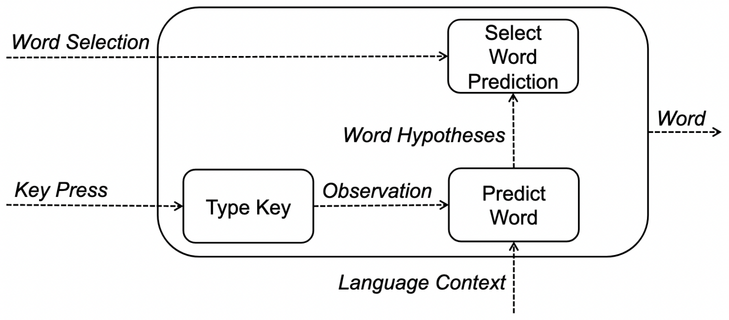
Another method is using *computational experiments* that generate system outcomes through a model, thus allowing designers to study the impact of parameters on performance. This is the approach taken by the mobile speech interface we discussed earlier in this section. For such experiments it is often useful to represent an expert user. This representation is sometimes referred to as an *oracle*. An oracle in user interface simulation is a simulation of a user that performs optimally. It is also possible to create an imperfect oracle, sometimes referred to as a *stochastic oracle* [429]. Simulation by an oracle is a very useful way to carry out rudimentary envelope analysis to tease out trade-offs and to help prioritizing features.

Paper Example 39.2.1 :

Understanding why word prediction may not be beneficial

Word prediction is a common functionality on mobile touchscreen keyboards. However, large-scale empirical research has suggested word predictions may not be that useful [620]. However, what is the *mechanism* explaining this observation? Using envelope analysis it is possible to gain an understanding of the factors contributing to this empirical observation [429].

A high-level function structure (see item VII) of a word prediction system is shown below:



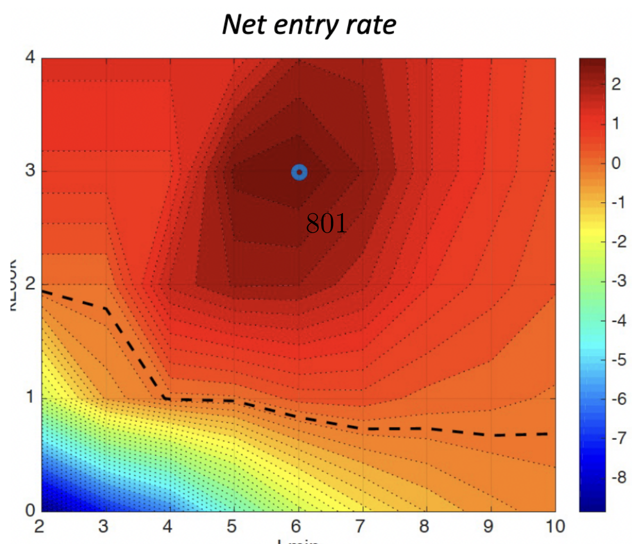
From this functional model we observe that we have two input signals, and they can be parameterized as follows:

- T_{key}: the time it takes the user to hit a key.
- T_{react}: the time it takes a user to react to a word prediction suggestion.

A latent set of parameters, *strategy*, generates process interaction between the design parameters:

- L_{min}: restrict word prediction usage to words of a minimum length.
- k_{look}: type k keys, then look.
- p_{max}: only attempt to check word predictions up to p_{max} keystrokes per word.

Note that all these parameters are uncontrollable. Other strategies are of course possible and in reality a user is likely using a mix of strategies. This parameters can now be investigated by implementing a simulation of a user typing text and interfacing with word predictions using the above parameters. The result is a set of envelopes of performance, one of them is shown below.



39.3. Optimization

Optimization refers to algorithmic search for solutions that best meets stated objectives. In HCI, optimization serves as a computational method for design and adaptation of user interfaces [614]. It has been applied to the computational design of keyboards, menus, information displays, information visualizations, and interaction techniques. In adaptive UIs, it has been applied to enhance distributed UIs, cross-reality displays, and web layouts. The method can be beneficial in conditions in which obtaining a good design is hard, for example because of a large number of decisions, constraints, or objectives. It can also applied in cases where design-choices must be automated, for example when adapting or personalizing a user interface. The main benefit of the method is the exploitation of powerful search algorithms (solvers). The main drawback is that optimization task must be precisely defined in order for the outcomes to be relevant.

Optimization is a *constructive method*: it can not only evaluate a solution, it can generate solutions. It can propose values of *decision variables* such that the solution that they jointly describe achieves a minimum or maximum of some stated objective. These could also be the types of representations we have discussed earlier. For example, to decide the position of a button on a display, one would need to decide its x and y coordinates, so two variables. To also determine its size, one would need to decide the (x, y) coordinates of both the top-left and bottom-right corner, so four variables. The *objective function* assigns each candidate solution a score that tells how good it is. For example, we could use Fitts' law to assess how quick to use button designs are.

An objective function states what makes a design good. The function can be about anything regarded desirable, such as surface features of the interface (e.g., minimal white space) or expected performance of users (e.g., 'task A should be completed as quickly as possible') and so on. In HCI, the quality of a interface is primarily determined by reference to end-users, for example user performance and experiences. Several objective functions have been proposed, ranging from performance to ergonomics, learnability, and aesthetics [614]. There are many principled (and unprincipled) ways to obtain objective functions: 1) literature (a model or theory from prior work), 2) data-driven approach (e.g., learning a function from human data), 3) standards (e.g., W3C accessibility requirements), and 4) design heuristics.

After a practical problem has been formulated like this, numerous efficient algorithmic methods (solvers) are available to solve it. They can search large numbers of possible solutions to the problem. The solution that best meets the objectives is called the *optimal design*. The process of 'solving' can be intuitively understood as search for highest peaks (when maximizing), or lowest valleys (when minimizing), in a vast 'landscape'.

Optimization offers a rigorous and actionable formal definition of what design *is*. *Design task* is an optimization task that consists of three elements:

A design space (a.k.a. search space, candidate space): a defined set of alternative designs to be considered. The design space is defined by definining a set of *decision variables* (e.g., "The x-coordinate of the 'Cancel' button") with limited ranges.

Objective function Defines what a 'good' or 'desirable' design is. In practice, maps a design candidate (that belongs to the design space) to an *objective score* that quantifies its goodness.

Task instance Defines task-specific variables (e.g., which menu commands must be placed into a menu system).

Formally, optimization in HCI can be defined as search for a design vector in a design space that maximizes the objective function:

$$\text{Find } \boldsymbol{x} = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix} \in \boldsymbol{X} \text{ which maximizes } \boldsymbol{f_\theta}(\boldsymbol{x})$$

where:

- $\boldsymbol{x}$ is an n-dimensional *design vector*, each dimension describing a *design variable*;
- $\boldsymbol{X}$ is the set of *candidate designs*;
- $\boldsymbol{f}$ is the *objective function*; and
- $\boldsymbol{\theta}$ is a set of parameters defining the *task instance.*

The first benefit of this approach for engineering is clear now: Defining a design space like this allows examining its size and structure. When we start doing this for design problems in HCI, we quickly realize that HCI's design problems can be very large. For example, for n functionality, there are $2^n - 1$ to combine them to an application, which for only 50 functions means 1,125,899,906,842,623 possibilities. Combinatorics can be used to analyze problem sizes.

Another benefit is that optimization provides methods to quantify how certain we can be that our result in design is good. In optimization, a design is considered to be the *global optimum* when the search method guarantees that it achieves the highest possible objective score in the whole design space. Such guarantees are offered by methods like exhaustive search and so-called exact methods (e.g., integer programming). When we do not have access to those methods, for example because the complexity of the objective function, we may claim global optimality. A design is *approximately optimum* when its objective value is within some margin of the optimal design, or when there is a good chance that only marginally better designs exist. Most of the methods we use in HCI can only offer approximatively optimum solutions. To sum up, to say that a design is optimal is to say that in the defined set of candidate designs, there is no better design (global optimum has been identified). Indeed, the term 'optimal' is overused and misused. In order to claim that a design is optimal, one must answer the following questions: 1) The best out of which options? (design space); 2) The best for what? (design objectives and task instance); 3) The best with what guarantees? (search method)

In comparison to formal methods in HCI – such as logic or state machines – optimization offers an effective yet flexible way of expressing design knowledge and objectives in a computationally approachable manner. Compared to data-driven approaches such as artificial neural networks, optimization retains direct control of assumptions in design, in particular via the definition of the design objectives. Optimization, furthermore, does not depend on one's ability to obtain a large training set, although its input parameters can be learned from datasets using machine learning methods, if needed. On the negative side, formalizing an optimization task does require careful analysis of the main factors in play in a domain.

One outstanding problem is how to allow designers apply optimization in their practice. The main issue is that designers are rarely able to express their problem at the level of precision required to form an optimization. task. The "one shot" paradigm of optimization is often impractical. In design practice, problems are ill-defined and designers learn and update the problem definition and the design space (see item VI). Optimizer-in-the-loop methods can help designers address this problem. In interactive design tools, designers may explore designs, compare them, and find rationale for them.

39.3.1. Example: Assignment problems

The assignment problem is the most widely studied class of optimization problem in HCI research. The task is to find a one-to-one correspondence between n items (e.g., letters) and n locations (e.g., keyslots), such that the total cost of the assignment is minimized. A valid assignment x can be defined formally such that every item must be assigned to exactly one location, and every location must have exactly one item.

In the linear variant of the assignment problem, the objective function is a linear function, for example

$$\sum_{i,k} c_{ik} x_{ik} \tag{39.7}$$

Here, x_{ik} tells the assignment, i.e. that letter i was assigned to slot j, and c_{ik} tells the cost of that assignment. Modern solvers can find the optimum solution efficiently.

However, in many HCI applications of the assignment problem, we need to account for interactions among items and locations. These require adding a quadratic term of the form

$$\sum_{i,j,k,\ell} c_{ijk\ell} x_{ik} x_{j\ell} \tag{39.8}$$

to the objective function. Here, we need to consider all pairwise assignments x_{ik} and $x_{j\ell}$. For example, the cost can be the time cost of moving the finger from key to another key when typing typical bigrams or phrases in a language. In this case, c would be the probability of that transition in language multiplied by the time cost. In Chapter 4, we provided several motor control models, such as Fitts' law, which can predict this time as a function of movement amplitude and target size.

This quadratic formulation is flexible, as more objectives can be combined into the cost term c, and they can be individually weighted against each other. For example, Feit et al.

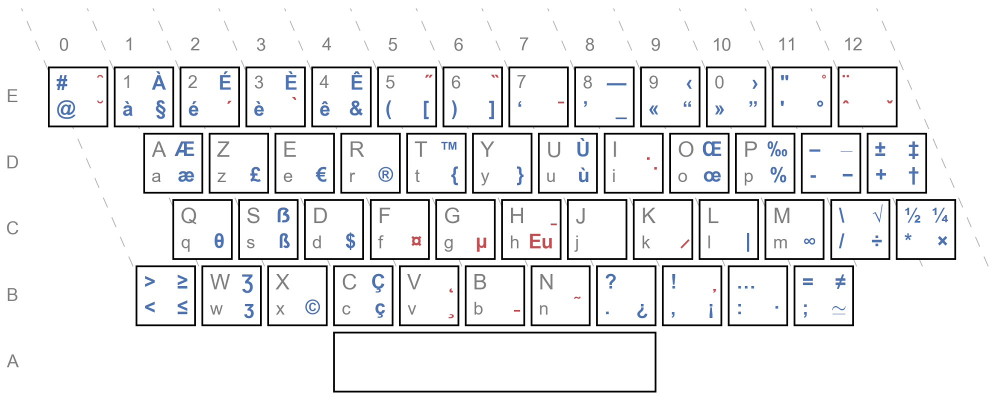


Figure 39.9.: The new Azerty standard was designed with the help of multi-objective optimization [235].

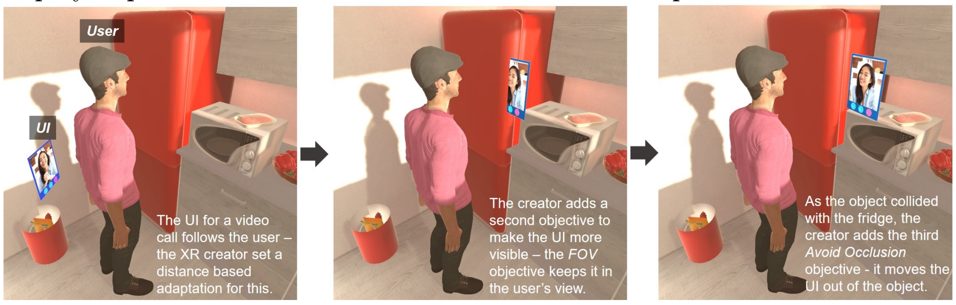
used this formulation to help redesign the new French keyboard standard for Azerty [235]. They optimized the assignment of special characters against four objectives: performance, intuitiveness, learnability, and ergonomics. These were based on HCI research or models of data collected specifically for this study. Examples of assignment problems beyond keyboards are the linear and grid menu problems. Given a set of commands and positions, the goal is to assign the commands to positions in the menu.

Paper Example 39.3.1 :

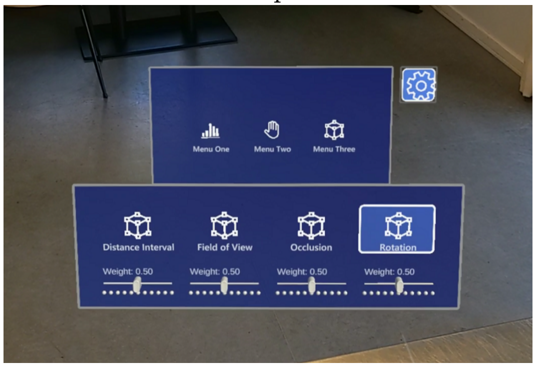
Adapting XR display locations with optimization

In cross-reality interaction, information displays need to be placed such that they are reachable and visible to the user. Moreover, they should not occlude virtual nor physical objects of interest. How to achieve this without writing a handcrafted rule set?

Belo and colleagues presented an optimization-based approach to adapting information displays while considering the geometric relationships that a user has with the environment [230]. They define an optimization task with weights for user-defined objectives. The objective function is a weighted sum of several objective terms: reachability, visibility, occlusion etc. These objectives control where in the 3D space a display is positioned in relation to the user's current position:


User
UI
The UI for a video call follows the user – the XR creator set a distance based adaptation for this.
The creator adds a second objective to make the UI more visible – the *FOV* objective keeps it in the user's view.
As the object collided with the fridge, the creator adds the third *Avoid Occlusion* objective - it moves the UI out of the object.

End-users and developers can determine the weights:



Distance Interval Weight: 0.50
Field of View Weight: 0.50
Occlusion Weight: 0.50
Rotation Weight: 0.50

Task instance is defined by the current positions of the display and the user in its environment. Simulated annealing is used to solve the resulting optimization problem in real time.

39.4. Pattern recognition and machine learning

Machine learning is fundamentally about computer programs that do not need to be fully prespecified but that learn from data. That is, a programmer does not need to hand code the precise behavior of the system. Instead, the system learns how to make, for example, decisions by observing training data.

In general, a machine learning system extracts *features* from domain objects, that is, the objects we which to process, for example photographs that we wish to classify. The features feed in as data into a *model* trained by a *learning algorithm*. The *model* provides output. In other words, the machine learning task is the model taking data as input and providing output in the form of, for example, classification decisions. The *learning problem* is to come up with learning algorithm that can take training data as input and provide a model as output.

Machine learning tasks are solved by using the right features to build the right models. Thus, tasks are addressed by models. Learning problems are solved by learning algorithms that produce models.

There are several tasks that we can use machine learning for:

Classification This means learning a classifier that can assign a label to data points, called feature vectors, it has not seen before. A special case is *binary classification* where we only consider two classes, such as classifying something as either true or false.

Regression This means learning a real-valued function from training examples labeled with the true function values. An example of a regression model is linear regression.

Clustering This means grouping data without prior information on the groups.

In addition, to these tasks there two fundamental classes of machine learning approaches. First, we can learn from from labeled training data. This can for example be photographs that are labeled with salient keywords describing the key objects in the photographs. This is called *supervised learning*. However, it is also possible to learn from unlabeled data. This is called *unsupervised learning*. In general, unsupervised learning is a more challenging machine learning problem. However, it is useful as it may be difficult to collect a large amount of representative labeled training data.

An important aspect when using machine learning in interactive systems is validation—how do we evaluate machine learning algorithms? Unlike many other technical problems, machine learning problems usually do not have a correct answer. Data is nearly always noisy. For example, training data points are mislabeled and features contain errors, which means it does not make sense to perfectly predict the training data as this would lead to overfitting and a poor ability to generalize.

We therefore need to evaluate machine learning algorithms based on their ability to perform classification, regression, and other machine learning tasks. For example, if we count the number of correctly predicted data points and dividing by the total number of data points we have tested, this provides us with a proportion which is known as *accuracy*.

To avoid overfitting on training data, that is, a poor ability to generalize, it is important to split the data into *training* and *test* sets. To improve the robustness of this procedure we typically apply this train- test split repeatedly in a process called *cross-validation*. Split the data randomly into k parts of equal size and use $k - 1$ parts for training and 1 part for testing. We do this k times (k folds) and use the average test performance, along with other measures of uncertainty such as standard deviation, as an indicator of performance.

Machine learning is a research field in itself and there are a wide array of approaches available. Here we will describe a few fundamental machine learning algorithms in some detail to give a flavor of how such models operate.

39.4.1. An example of machine learning: linear machines

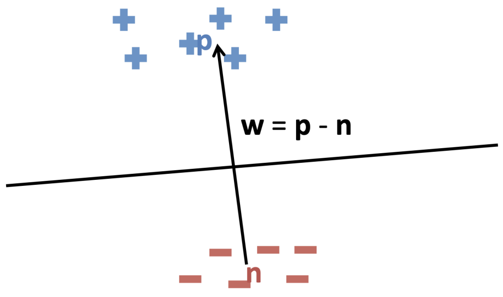
As an example of what machine learning is about, let us consider what is known as a *linear classifier*. This class of machine learning algorithms is also referred to as *linear machines*.

We will consider what is called *binary classification*, which means we will attempt to classify unknown data points as belong to one out of two predefined classes.

Linear discriminant functions. If there exists a linear decision boundary separating two classes then the two classes are said to be linearly separable. If this is true then we can form a linear decision boundary. This is defined by the equation $\mathbf{w} \cdot \mathbf{x} = t$, where $\mathbf{w}$ is a vector perpendicular to the decision boundary, $\mathbf{x}$ points to an arbitrary point on the decision boundary, and t is the decision threshold.

The vector $\mathbf{w}$ is pointing from the center of mass of the negative examples $\mathbf{n}$ to the center of mass of the positive examples $\mathbf{p}$. $\mathbf{w}$ is therefore proportional to $\mathbf{p} - \mathbf{n}$.

By setting the decision threshold appropriately we can intersect the line from $\mathbf{n}$ to $\mathbf{p}$ at its midpoint. This line is called a *linear discriminant function* in machine learning, and $\mathbf{w}$ is a linear combination of the examples. Figure 39.11 illustrates this idea.



Any unknown data point will be classified as a positive example if it is above the line in Figure 39.11 and otherwise as a negative example. This example uses binary classification as it is straightforward to visualize. However, linear discriminant functions can also be used to perform multi-class classification.

A linear classifier based on a linear discriminant function is a simple model that is optimal under certain specific assumptions. However, when these assumptions do not hold, which is frequently the case, then this method performs poorly. An example of its use in HCI is as a machine learning method for allowing a designer to specify pen gestures that can subsequently be classified by the system [698].

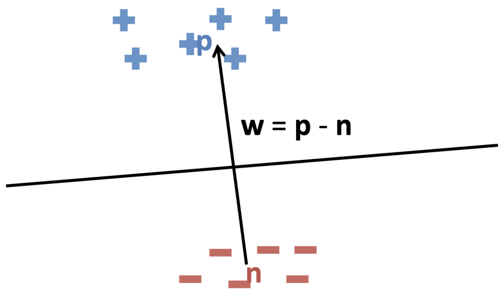


Figure 39.10.: An illustration of a linear decision boundary separating positive and negative examples.

Support vector machines. For high-dimensional data, the data space is vast and as a consequence the examples tend to be much further apart. Linearly separable data does not uniquely indicate a decision boundary in such cases and therefore we face a problem: which of the decision boundaries should we choose?

A *margin* of a linear classifier is the distance between the decision boundary and the closest instance. A common decision is to prefer a classifier where the margin is the largest. A particularly well-known example of a learning algorithm that attempts to identify such "maximum margins" is what is known as a *support vector machine* (SVM). This classification method is also popular in HCI applications that require classification of, for example, sensor data as an SVM is more robust to noise than using a linear discriminant function.

Assume we have some form of distance or scoring function that defines a decision boundary: $\mathbf{w} \cdot \mathbf{x} - t$. Then a true positive example data point $\mathbf{x}_i$ will have the following margin:

$$\mathbf{w} \cdot \mathbf{x}_i - t > 0 \tag{39.9}$$

Further, a true negative example data point $\mathbf{x}_j$ will have the following margin:

$$-(\mathbf{w} \cdot \mathbf{x}_j) - t < 0 \tag{39.10}$$

For a given training set and decision boundary, let m^+ be the smallest margin of any positive example data point and let m^- be the smallest margin of any negative example data point. Then the sum of m^+ and m^- should be maximized for the maximum margin. Further, since this sum is independent of the decision threshold t, we can readjust t so that of m^+ and m^- are equal. The figure below illustrates a support vector machine with a decision boundary and a margin separating positive and negative data points.

If the data is not linearly separable then we need to modify our support vector machine. We can do this by relaxing the constraints when identifying the support vectors by allowing some of the examples to be inside or at the wrong side of the margin. This results in a *soft margin support vector machine*. The degree to how much the margin

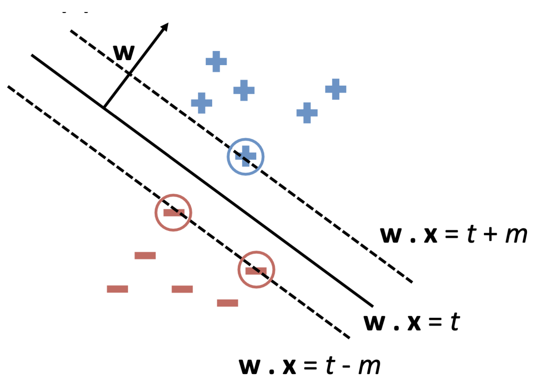


Figure 39.11.: An illustration of a suppor vector machine separating positive and negative examples with a decision boundary (indicated by solid line) with a margin (indicated by dashed lines).

is allowed to be influenced by this relaxation is controlled by a *complexity parameter*, typically denoted C, which controls the optimization.

Support vector machines are capable of generalizing beyond linear decision boundaries. This because support vector machines represent a notion of distance between data points, and this distance is represented by a *kernel*. The default kernel is linear but several other kernels exist. Changing the kernel allows a support vector machine to find nonlinear decision boundaries in the data space. It is also possible to convert the distance between a data point and the decision boundary to a probability through a process known as calibration.

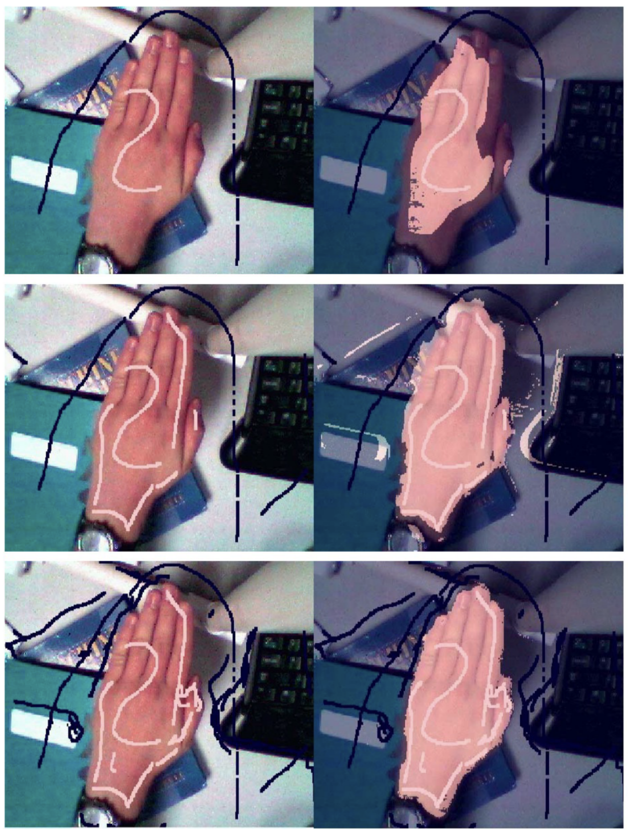
Paper Example 39.4.1 :

Interactive machine learning

Interaction with AI can benefit AI systems developers as well as end-users. Interactive machine learning [231] is the idea to assist the machine learning algorithm by allowing the designer to rapidly provide training data to correct errors made by the learning algorithm.

The idea is exemplified a system that lets designers to quickly build a pixel-based image classifier that allows the designer to focus on classification problems rather than image processing, algorithm selection, or parameter tuning.

The image below illustrates the concept. The user first imports images to the system and then performs manual classification. This provides the system with labeled images. The system uses this dataset to train a classifier and provides visual feedback on classification accuracy. The designer can now refine the classifier by providing manual classifications. In the image below this is illustrated by the system's increasing ability to detect the hand in the image following additional classifications by the designer.



39.5. The limits of a model define the limits of its application

The success and failure of a computational method critically rests on a central element—the information contained in its model. However, situations in HCI can be represented in various of ways. Consider the creation of an algorithm that generates an information visualization for a provided dataset. This visualization should allow users to quickly understand the key facts in the dataset. To be useful, such a model need to cover the key factors about the user's task, the dataset, and possibly information about human perception and decision-making. Such information can be represented in many ways: as rules, mathematical models, and neural networks, among many other potential representations. However, how do we know if the *right* information is there? To answer this question, we need to open up ways in which a model can fail to represent the world.

One way for the model to fail is if it does not represent the true tendencies in the world. For example, the model could hypothetically fail to represent the way in which the color blue is hard to discern against a dark or red background. A common reason for such failures is that there is simply too much variation in the data. This results in the algorithm being unable to pick up the underlying trends in the data. The term *aleatoric uncertainty* refers to such uncertainty caused by stochasticity.

Another way a model can fail is that it lacks knowledge about a relevant factor. This is a much more challenging problem. For example, the visualization algorithm might lack information about the limits of visual attention. Without such knowledge, there is no reason why such an algorithm should not try to pack all information in a dataset into a single graph which would be impossible for a user to comprehend. The model underpinning the algorithm could also fail to contain information about the situations in which its outputs are used. A mobile user, for example, can only distill the gist of a complex dataset due the small form factor of the display. Without such knowledge, the algorithm will generate charts that are unusable for such a situation. The term *epistemic uncertainty* refers to uncertainty caused by lack of knowledge.

Models can also be biased in the way they capture certain factors. There are two interpretations of *bias*, which are often confused. *Statistical bias* refers to the misrepresentation of a distribution. For example, the visualization algorithm might favor certain plot types, such as line charts, over others, with no basis in real data. The much harder type of bias to catch is *ethical bias*, which refers to ethically questionable applications of a model's result. For example, our visualization algorithm could, without it "knowing," use labeling or color combinations that are considered offensive to some users.

HCI research plays a critical, but often underappreciated, role in identifying and rectifying inadequacies of models. It is through user research that we can expose requirements that make us realize we need to include additional factors. Further, it is through evaluations that we can expose biases and uncertainties in a model. Such shortcomings may not be visible when examining the algorithmic accuracy alone. However, at the end of the day, observations about models must be translated to the model, either through new modeling assumptions or by making use of new datasets. Here HCI researchers have a critical role in helping to ensure the representation is the right one, so that it represents what it is supposed to represent without biases.

Summary

- Computational methods rely on models that represent aspects of interaction for computation.
- In envelope analysis, we chart the range of behaviors of an interactive system in different conditions.
- In optimization, we search for best solutions to a well-stated design problem.
- In learning, we learn a model from data in order to solve a learning problem such as classification.

Exercises

1. Feature engineering. Consider an accelerometer as a sensor. You are given the task of designing an input technique using the sensor. When the sensor is lifted quickly, it should trigger an event. What kinds of features would be informative for detecting this action?
2. Optimization. Consider deciding the size of a button using Fitts' law. According to this model, if you want to minimize selection time, what happens to the size of the button? Now, this design problem becomes much more interesting if you have two or more buttons. Next, consider the case where you have a screen that has two buttons next to each other. The screen has limited size of w_s. You now need to optimize the sizes of the two button to minimize expected selection time. What happens to the sizes? Let us next assume that the two buttons 1 and 2 are accessed with probabilities p_1 and $1 - p_1$, respectively. What happens to the sizes of the buttons?
3. Envelope analysis. This task continues from the previous one. Consider 1) button size to be a controllable parameter, limited in the way described above, and 2) parameter b to be uncontrollable parameters describing users' properties. Carry out envelope analysis for button size; that is, plot the envelope of button size as a function of b.
4. Computational design. Consider a linear menu with n items. Formulate the task of deciding the order in which the items should appear. Assume that you are given a distribution that tells the frequency with which the items are selected by users. To solve this exercise, we recommend first verbally defining the task, then defining the variables that need to be decided, then defining the objective function, and then possible constraints. Hint: This can be formulated as an assignment problem.
5. Developing a gesture recognizer. This task is for students familiar with neural networks. The homepage of the book offers a real dataset on gestures recorded using a computer vision sensor. The homepage offers a skeleton for a Pytorch-based

recognizer. Define a classification problem, a neural network structure, a loss function, and then train the network. What would be a sufficient level of accuracy for a real application?