Introduction

This book is about how to build interactive software. Computers have always interacted with people. What has changed is the way in which computer software has adapted to the needs of human beings. From the very beginning, programming languages were designed to simplify people’s ability to tell the computer what needed to be done. The interaction times were in hours and days rather than milliseconds, but the goal of telling the computer what we wanted done was always there.

The forces that have created the interactive computing world we now enjoy are best summarized by Gordon Moore in XXXX [xx]. He stated that computing power doubles every 18 months with no increase in cost. Anything with such exponential growth is an amazing phenomenon. This means, for example, that in the next 18 months we...
will have has much increase in computing power as all of computing progress in our history. To get a rough grasp of the impact of this growth, in 10 years computing capacity will increase 100 times with no increase in cost. That kind of growth has fundamentally changed how we use computers about every 10 years.

As interesting as this increase in power is, it has not had near the impact on HCI (Human Computer Interaction) as the inverse of this law.

The inverse of Moore’s law holds computing power constant and tracks the cost of that amount of computing power over time. Cost is generally cut in half every 18 months. This process of cost reduction places increasing amounts of computing power within economic reach of increasing numbers of people. This also makes computing feasible for an increasing number of tasks. On a 10 year horizon we see computing costing 1/100 of what it does today. That has an enormous impact on what we use computers for.
This increase in computing power has made it possible for many computational tasks to be completed in less than 1/30 of a second (faster than the eye can see). Whenever the time to complete a computational task falls below a few seconds, that task becomes interactive and responsive to human control. When the costs fall to a few hundreds to a few thousands, that task becomes economically feasible for many ordinary people. That is the domain of HCI.

**Human Computer Interaction**

Despite the marvels of computing technology, human capacities have not changed. The highest bandwidth path for communicating information to a human being is through the eyes. The second fastest channel is through the ears. The eyes are approximately 1,000 times faster than the ears in absorbing information. The ears are about 1,000 faster than touch. This is the fundamental reason why graphical user interfaces are so important. This book will focus exclusively on graphical user interfaces because they are so very important to human beings.

There are really three basic questions in human-computer interaction:

1. How do people behave and respond?
2. How do we design interactions that serve human needs in a compelling way?
3. How do we architect software/hardware systems that meet these needs?

This book is focused on the software/hardware architecture of graphical interactions. As we go along, results of research into how people behave and respond will be sited where useful (for example the relative bandwidth of eyes and ears) but this is not a book for such study. Some basic ideas from design will also be presented as appropriate, but the design process is the topic of a companion book. This is a book for programmers and it outlines the basic ideas for architecting software.

**Model-View-Controller**
Since the times of the Xerox Star, which really popularized graphical interaction, the model–view-controller architecture has dominated the way interactive systems are developed. Figure 1-3 shows the classical view of this architecture.

![Model-View-Controller Diagram](image)

**Figure 1-3 – Model-View-Controller**

**Model**

The *model* is the heart of the interactive system. This is the information that the user is trying to understand and possibly modify. In a graphical user interface we almost always represent interaction as the perception of information, a decision process followed by the expression of some modification of the model by the user.

The model can be changed in a variety of ways. The user may express a change through the controller. Some other computational process may change the model, such as a search or some analysis. Another user may change the model. Regardless of the reason why the model has changed, it must notify the view whenever such changes occur. This
notification process is a key part of the software architecture and will be revisited many times throughout this book.

**View**
The role of the view is to translate the model into a presentation that is suitable for the user’s current needs. Not that the view does not necessarily present the entire model. Most models are too large to be perceived at one time. Many models have more information that is actually needed by the user at a particular time. The available display space (particularly on a smartphone) may impose presentation constraints. The role of the view is to present some portion of the model for the user. The view not only translates the model into a visual presentation but also must respond to changes in the model. It is essential that the view always reflect the current state of the model. We will discuss several architectural approaches to make this happen.

Note that there may also be multiple views of the same model. For example a spreadsheet shows a view of the cells of the sheet in one view with the formula contents of one of the cells in another view. These two views need to be consistent with the model and with each other.
In important part of the usability of an application is the appropriate design of the views. Don Norman (Norman, 1988) has identified the gulf of evaluation where the user looks at the information being presented by the view and perceives something different than what the model actually represents. A simple example of this is the basic calculator shown in figure 1-5. When performing addition, or most other operations, there are actually two operands being added. However, the simple calculator shows only one. By omitting the second operand from the view, users frequently become confused about what is being added. They usually compensate by doing the operation multiple times.

Though omission of important information is a common source of the gulf of evaluation, there are others such as: confusing screen layouts that make it hard to find information, obscure encodings or jargon that is unfamiliar.

Moore’s law has had a huge impact on the software architecture of the view. For most interactive programs, the drawing of a presentation consumes by far the most computing resources. Compare for example the computational effort for a word processor to insert a new character into a document with the computation required to redraw the page to
reflect the new word wrapping caused by the insertion. Efficiency in updating the view was formerly a huge part of quality interactive software. Even on today’s smart phones there are more than a billion machine instructions available to update the screen in 1 second after an interaction. This leads us to more flexible view architectures than were possible before.

**Controller**

Based on what the user has perceived they make a decision about what they want to do next and express their desires by generating input events. Input events come in a variety of forms and each new type of interactive device creates new styles of events. From a software architecture perspective we must associate each event with some code that should process the event. This code is part of the controller and the event/code binding is a fundamental problem in interactive software.

The binding of events to code depends a great deal on the programming language being used. The ability of C to handle pointers to functions influenced the X-Windows event handling mechanism[xx]. Older languages such as Lisp and newer languages such as JavaScript take advantage the fact that the compiler/interpreter is available at runtime. In such languages the binding of events to code involves the dynamic parsing and execution of code snippets at runtime. When Smalltalk was developed for the Xerox Star, object-oriented programming was uses as the primary mechanism for event-code binding. This technique has continued to develop through C++, Java, and C#. Each of these languages provided improvements on how programmers can link input to code.

The controller design, in conjunction with the view gives rise to what Don Norman (Norman, 1988) calls the *gulf of execution*. This is when the user understands what they want to do but cannot determine what input expressions are necessary to accomplish the purpose. User interfaces with a command line have serious gulf of execution problems. It is difficult for a user to find out the right command and syntax to accomplish a desired goal. Graphical user interfaces do not necessarily
solve this problem. Icons can be rather obscure in their meaning. There may be so many choices that it is hard to find the desired command or setting in all of the menus and dialog boxes. Just because a feature is there does not mean that the user can take advantage of it.

One of the roles of the controller is to aggregate several input events into a meaningful action. For example a simple *click* action actually consists of a mouse-down event followed by zero or more mouse-move events that do not move very far and a mouse-up event that occurs in a very short time after the mouse-down. It is the controller that assembles these together into a simple click.

**Human capacity**

We will not spend a lot of time on human capabilities in this book. Those issues arise more importantly when discussing the design of the actual user experience. However, there are some key facts that are important to know:

- The human retina is sampled by the brain approximately 30 times per second. This means that for animations or other smooth behaviors they must be updated at least 30 times per second. Any faster than that is invisible to the human eye and makes no interactive difference.

  - They human eye moves rapidly from place to place either voluntarily or involuntarily about 5 times per second. This means that interactive movements that occur at least that fast may not appear smooth, but they do not appear cumbersome. The eye is never sure if a jump was eye motion or hand motion. Getting things done in 1/5 of a second will yield good interactive behavior.

- Most of your retina consists of rods, which cannot sense color. This means that most of what you see is actually some shade of gray. The macula contains many cones which can see color. Cones sense red, green, or blue. This is why we can represent any visible color as a combination of red, green and blue to
deceive the eye into seeing something else, like yellow (combination of red and green).

These basic facts will be enough for our discussion of software architectures. A more detailed knowledge is necessary for design work.

**Computing Capacity**

One of the tenants of computing theory is that algorithms should be compared based on their relative behavior when the number of items pushes to infinity. This has helped to fundamentally understand the nature of various algorithms. However, in interactive systems, N never remotely approaches infinity and never can because of the limitations of human beings.

Consider today’s 2 megapixel screens. Many of us have multiple screens but we interact on only one at a time. The reason is that we cannot see them all at the same time. It is nice to glance over to another screen and see information we have posted there, but when we start to interact on the other screen, we of necessity stop interaction on the previous one. We are physically incapable of actively working in such a large visual space. Having many screens allow us to instantly shift our focus of interaction, which is very helpful but we still are interacting on one at a time.

Suppose that we could fit a unit of information into a 10x10 pixel space (about the size of a single character). We could put 20,000 such pieces of information on the screen. Today’s processors perform about a billion instructions per second. That means that every second we can devote 50,000 instructions to every piece of information. In every 1/30 of a second we can devote 1,600 instructions to every individual piece of information.

Filling every 10x10 space on the screen with important information would be a terrible presentation design. Human beings could not find anything or understand all of it. In practice about 1,000 pieces of information on a screen is the maximum usable density. This means 2
million instructions per piece of information every second. Historically interactive architecture was dominated by the need to sustain interactive rates of response. Our thought experiment shows that this is no longer true and with Moore’s law is fading rapidly.

We can also consider the limits of interactivity. A blazingly fast typist can type 100 words or 600 characters per minute. That is 10 characters per second, which is very fast. However, we have 100 million instructions available to us to process each character input.

If we are interacting with 1,000 pieces of information and have 1GB of RAM, that leaves us with 1 million bytes per piece of information. Memory is obviously not our problem either. With desktop computers having a terabyte of disk and Google storing the entire web, storage space is no longer an issue either. Smartphones have smaller capacities but they also have much smaller display capacity. The maximum information density on a smartphone is tens to hundreds of items.

The point here is that computing capacity is not the key constraint for our interactive software. Today’s interactive problems are in building the right user interface. Finding new interactive styles that enhance desirability and productivity is more important than implementing efficient B-tree search structures. Reliability, correctness, collaboration with others and ease of development all dominate our architecture discussions. We will use many "inefficient" and simplistic algorithms because rapidity of design, reliability of the implementation and satisfaction with the user experience are far more important.

**Key ideas**

There are four key ideas that will occur throughout the book. They are:

- Data Transformation
- Vision and Drawing
- Geometry
- Change propagation and notification
Most of our architecture will be based around translating back and forth between the data that is the model and the data that is the view. Representing the data in a form that we can easily express such transformations will be important in our architecture discussions. It seems a little odd that human interaction is built around data manipulation but in modern user interface software that is increasingly true. Obviously vision is a key issue and all of the machinery necessary to draw a picture on the screen will be important to the view. Geometry raises its head in a number of ways. One of the most important is that we need geometry to decide if some portion of a picture has been selected. We will also need it for a variety of other interactive manipulations. Lastly there is change propagation. Whenever the model, the view, the data base or the remote web service changes, the other parts of the system need to be notified. Keeping all of these pieces consistent with each other is a fundamental task of our software.

Software examples
This book is intended to be independent of any particular system. The goal is to show the key ideas and how they fit within a variety of systems. Because this is a book about software architecture, there will be a lot of code snippets. These will generally follow a Java syntax. However, the code examples are not designed to follow any particular language or library. Frequently information that is not relevant to the discussion will be left out or simplified. None of the code examples are intended to be compiled directly. The goal is to show the essence of the ideas that can then be recoded in a variety of languages and platforms.

Problems
1. Pick a computing device of your choice. Imagine that the device has 100 times (10 years) the computing and memory capacity that it has today. Describe 10 interesting things that this device might do that it cannot do today.
2. Pick a computing device of your choice. Imagine that the device is 100 times cheaper (10 years) than it is today. Describe 10
interesting new uses for a computer that this would make possible.

---

**Citations**