Follow @derekknox

Chapters
+

Light
-

Chapter 3 Programming & Visual Design

#

Design

O
n the surface, design is both a process and a product. At its core however, design is a single thing.

> *Design is an accumulation of decisions*.

That is it. Both the process of designing something and the resulting product of that process (at any moment in time) are an accumulation of decisions. These decisions are intentional or unintentional. They are informed or uninformed. A good designer's decisions are intentional and informed. A poor designer's decisions are unintentional and uninformed. Just as we all breathe, we are all designers. Not all of us are good designers however. Few are great. We can all improve.

Visual, sound, experience, circuit, hardware, and software are just a few examples of various design domains. Design is often associated solely with the visual, but it is important to acknowledge that it exists in all domains. That said, we will be working with the visual and software domains moving forward.

In the context of visual design we design *visuals to communicate*. We *encode ideas visually*. These visuals are often used to instruct or entertain. They may be intended to elicit action or provoke thought. A design may be completely abstract even. Regardless, when a viewer perceives a design, an opportunity for interpretation (decoding) presents itself. Remember, as visual designers we may only influence. The meaning extracted is

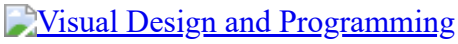
up to the viewer. Intentional and informed design increases the likelihood of the desired meaning to be interpreted by the viewer.

In the context of software (coding) design we design *systems to communicate*. We *encode ideas programmatically*. When we want computers to understand these ideas (typically for translation to on-screen visuals), we author code using different coding languages. On the web platform for example, we use a markup language (structure via HTML), a style language (style via CSS), and a programming language (behavior via JavaScript). A distinction between coding and programming is often made where programming refers solely to behavior where coding refers to all three. I will use this distinction moving forward.

Learning a programming language, like learning virtually anything, takes effort, practice, repetition, and time. Learning something new relative to something already understood can help decrease this time dramatically. This is my aim.

I want to leverage your visual design knowledge, experience, and intuition. We will review the Elements, Principles, and Constructs of Visual Design *relative to programming*. In doing so, we will empower and strengthen your ability to understand programming and its core concepts at a deeper and more intuitive level.

With the elements as primitives and the principles as an organization of those primitives, the constructs are the compound results. Unfortunately there is not a one-to-one mapping from the visual realm to a programming equivalent. We will be able to come close enough however and create a solid bridge for comparison in the process.


Visual Design and Programming

Through the application of the elements, principles, and constructs a visual designer may influence the meaning a viewer extracts from a visual design. A good visual designer is cognizant of their application of them where a poor designer is not. A good visual designer takes advantage and applies them with intention.

Just as a visual designer has the power to evoke meaning for a viewer, so too does a coder. This coder's code often gets transformed into a visual design on-screen that is *interactive* and *alive*. These interactive designs entertain, educate, and empower people. Coding is very rewarding for these reasons. Empower yourself.

#

Elements and Elements

The elements of visual design are the *primitive* pieces of visual design. Though some may argue the exact list, we will use the one below. A related set of primitives are listed for programming code though there is not a formal one-to-one mapping. A bunch of new terms are introduced here, but don't expect to grasp them all at once. We'll continually revisit each throughout the rest of this book.


The Elements

#

Point and Expression

The *point* is the first element. A point by itself may, at most, represent a single idea or value at a given time. Similarly an *expression* represents, at most, a single idea or value at a given time. Both are the smallest meaningful and composable elements of visual design and programming respectively.

Point:


Point

Expression:

```
'#2D7DD2'
```

#

Line and Statement

More than one point or expression enables another level of meaning to be possible. For example connecting two points forms the second visual element, a *line*. Similarly, connecting expressions forms the second programming element, a *statement*. Each are the building blocks of visual design and programming respectively.

Line:


Line

Statement:

```
var color = '#2D7DD2';
```

#

Shape and Function

Combining lines together in specific ways enables the third visual element, a *shape*. A shape is ultimately a set of lines, straight or organic, that creates an enclosure. A *function*, the third programming element, is a great way to create an enclosure of statements. Take note that a function, like a shape, is considered a self-contained unit.

Shape:


Shape

Function:

```
function changeTheColor() {
  // Function code goes here
}
```

#

Space and Scope

Naturally, by creating an enclosure, you simultaneously create *space*, the fourth visual design element. In programming, this fourth element can also be thought of as space, but the term used is *scope*. Shapes allow their enclosed space to be distinct from the space outside. A function's scope allows it to be distinct from the scope outside too. Distinct space and distinct scope allow various areas to exist simultaneously without clashing. This is super valuable when programming.

Space:


Space

Scope:

```
// External scope
function changeTheColor() {
  // Internal scope
}
```

#

Color and Signature

Color, the fifth element of visual design provides a common way to differentiate similar or otherwise identical shapes. A function's *signature* allows functions to differentiate themselves too. Adding a signature to a function, like color to a shape, gives it an identity.

Color:


Color

Signature:

```
function changeTheColor(newColor) {
  color = newColor;
}
```

#

Value and Arguments

Value may be applied to a shape's color to modify the result. Similarly in a function's signature, different *arguments* may be applied to change the function's result. Value allows a shape with a specific color to be reused with a different effect. Arguments allow a function with a specific signature to be reused with a different effect too.

Value:


Value

Arguments:

```
changeTheColor('#8D6D00');
```

#

Form and Object

Grouping the aforementioned visual design elements in a particular way gives rise to a particular *form*. Similarly, grouping the aforementioned programming elements can give rise to a particular *object*. As a shape and a function are considered self-contained units, a form and an object are more powerful versions. Each embody some configuration of their respective prior elements. A form often embodies more than one shape just as an object often embodies more than one function.

Form:


Form

Object:

```
var someObject = {
  changeTheColor: function (newColor) {
```

```
    color = newColor;
  },
  changeSomethingElse: function () {
    // Function code goes here
  }
}
```

#

Texture and State

Texture when applied to a shape or a form gives it a richer quality. This richness is exemplified as a sense of time, where a texture is aged and weathered for example. Similarly, a function or object with *state* gives it a richer quality, a sense of time as well. A shape or form that lacks texture often lacks richness. The same is true for a function or object that lacks state. It is worth noting that state is vital to programming where texture in a visual design is not. State is a necessity in programming interactive designs.

Texture:


Texture
Texture

State:

```
function Colorer () {
  var color = '#FFFFFF';
  var oldColor = '#000000';

  function changeNew (newColor) {
    oldColor = color;
    color = newColor;
  }

  function changeOld () {
    color = oldColor;
  }

  function getColor () {
    return color;
  }

  return {
    changeTheColor: changeNew,
    revertTheColor: changeOld,
    getTheColor: getColor,
  }
}
```

#

Summary

The linear walk through above is intentional as each primitive builds upon the previous. Points connect to create lines just as expressions connect to create statements. By combining these lines in specific ways, a particular shape is created. The shape encloses space. Similarly, combining statements and enclosing them in a function creates a scope. Adding color to a shape gives it identity just as adding a signature to a function does. Assigning value to a color gives it a particular effect or result. Arguments assigned to a function provide a particular result as well. A form exists as a particular combination of the prior visual elements. An object, like a form, exists as a particular combination, but of the previous programming elements. Finally, richness is achieved through texture for visuals and state for code.

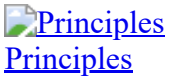
An infinite amount of visual and programming designs are possible using the elements above. Your creativity and experience are the only limiting factors.

Over time, the visual and programming communities at large acknowledged common patterns that were useful in using the aforementioned elements. In visual design these are known as the *design principles*. In programming they are called *design patterns*. Both are extremely useful in their respective domains.

#

Principles and Patterns

With the elements of visual design as primitives, the principles are the *organization of those primitives*. There are generally considered to be eight though names and count are debatable. I will use the list below.


Principles

The core takeaway is that these are recognized approaches for organizing visual design elements for consistent effect. Humans perceptually and often unconsciously absorb specific meaning when the principles in a visual design are applied. Being intentional and informed is to your advantage.

Programming's equivalent to design principles are *design patterns*. Unlike the principles, there are many more than eight. Like the principles however, they are recognized approaches for organizing the elements.

Both the visual designer and the programmer leverage the principles and patterns respectively to achieve consistent effect. A visual designer applies the principles for the viewer's benefit. The programmer applies design patterns for his or her and other coders' benefit. This may sound selfish, but the reality is that the user of a software product benefits too. The benefits could be speed, reliability, consistency, and flexibility for example. Additionally, using design patterns allows other programmers to better understand a program's code design.

Design patterns allow the programmer to take an approach that solves a common or general problem, one that has been solved before. Design patterns are not a beginner topic so we will not be going into detail. That said, I recommend two books for when you feel ready. *Learning JavaScript Design Patterns* by Addy Osmani and *Design Patterns: Elements of Reusable Object-Oriented Software* by the Gang of Four (GoF) are both amazing resources.

I will not leave you hanging though. Below are a few examples that illustrate some of the more common design patterns.

#

Pattern Examples

Often times when programming, you only ever want to have a single instance of a type of object exist in your program. In a sports video game for example you usually only ever track a single game's score. Depending on the sport you likely only ever want a single ball or puck. In a drawing application you may only ever want a single pen or brush to be selected at any given time. The *singleton* design pattern is a useful pattern for enforcing this oneness.

In forgiving sports video games or drawing applications, the software lets you undo and redo your actions. This is an extremely common desire in software, the ability to undo and redo actions. The *command* and *flux* design patterns are extremely handy for giving an application, and thus its users, this ability.

A sports video game will likely consist of two teams, each with a certain amount of players. Design applications may provide numerous tools like a pen, brush, or eraser with variations of each. A few flavors of the *factory* design pattern are useful in stamping out these products (team, player, or tool).

There are many other design patterns and all are useful in particular scenarios when programming. Sometimes as you are learning to code, you connect dots on your own and unknowingly use some of the design patterns. These experiences are rewarding as you figure out better ways to code on your own.

By leveraging the elements and principles in visual design you may construct more meaningful visuals. In programming, by leveraging the aforementioned elements and patterns you may construct more meaningful and useful code. Be intentional and informed.

#

Constructs and Components

The constructs embody the elements and principles in an effort to communicate specific structure, ideas, and messages. With the elements as primitives and the principles as an organization of those primitives, the constructs are the compound results. The constructs give visual designers the greatest chance for successfully communicating through a design. Remember, we may only influence. There are three core constructs:

1. Grid
2. Imagery
    - Photography
    - Iconography
    - Illustration
3. Typography
    - Content
    - Style

The lack of a grid can be useful, but more often than not a grid will exist in a visual design. A grid instantly provides the scaffolding, scaling, and patterns of space in which imagery and typography will live. Imagery in its various forms and typography are the core communicative constructs of a visual design. Imagery typically communicates faster than typography, but the content of text itself has the least likelihood of misinterpretation.

In programming, components embody the elements and design patterns in an effort to encapsulate specific functionality and ways of communicating with other components. They provide flexible functionality and manage subsystems of the root system that is the program itself. Components give the coder reliable and often reusable functionality through a single object, a composition of objects, or a family of objects.

There is no sensible relation between the three core constructs of visual design to an equivalent in programming. There do happen to be three high level component groupings however:

1. Built-in
    - Language
    - Environment
2. Third-party
3. Custom

Now is a good time to share an often deemphasized fact that programmers benefit from. A lot—and I mean a lot—of code you will write is already written. How can you write what is already written? Put another way, you often get to copy and paste (this is not an excuse to not understand that code though). Additionally, programmers leverage their powerful code editors to generate code automatically. Rarely does a programmer start from scratch. In fact, the first two component groupings mentioned above can be used as a checklist before having to write your own custom code.

Built-in components are those that come with the language you are authoring in or the code's execution environment. The initial author(s) of the programming language provide components that are "built-in". Additionally, the environment often provides components built upon the language to provide even more functionality.

As you may have guessed, a programming language never has all the functionality you need. This is where *third-party* components often come to the rescue. These are written by other programmers and they are often shared for us to use (for free or purchase). In fact, you can share the code you create for others to use too. We

all help each other out. Finally, you write custom code by combining built-in and third-party components with the code you create on your own.

A component at its core is synonymous with an object that may or may not parent other objects. Think of a component as one or more objects that achieve some set of functionality. This functionality is achieved through the object's expressions, statements, and functions in addition to the other elements. The patterns help guide the set of functionality a component is capable of. Combined, the elements, patterns, and components allow you and other programmers to bring life into otherwise static visuals that display on-screen. Coding and programming are expressive, fun, valuable, and powerful skills that enable you to bring your visual designs—your ideas—to life!

Chapter 2 Structure, Style, & Behavior

Chapter 4 Interactive Code