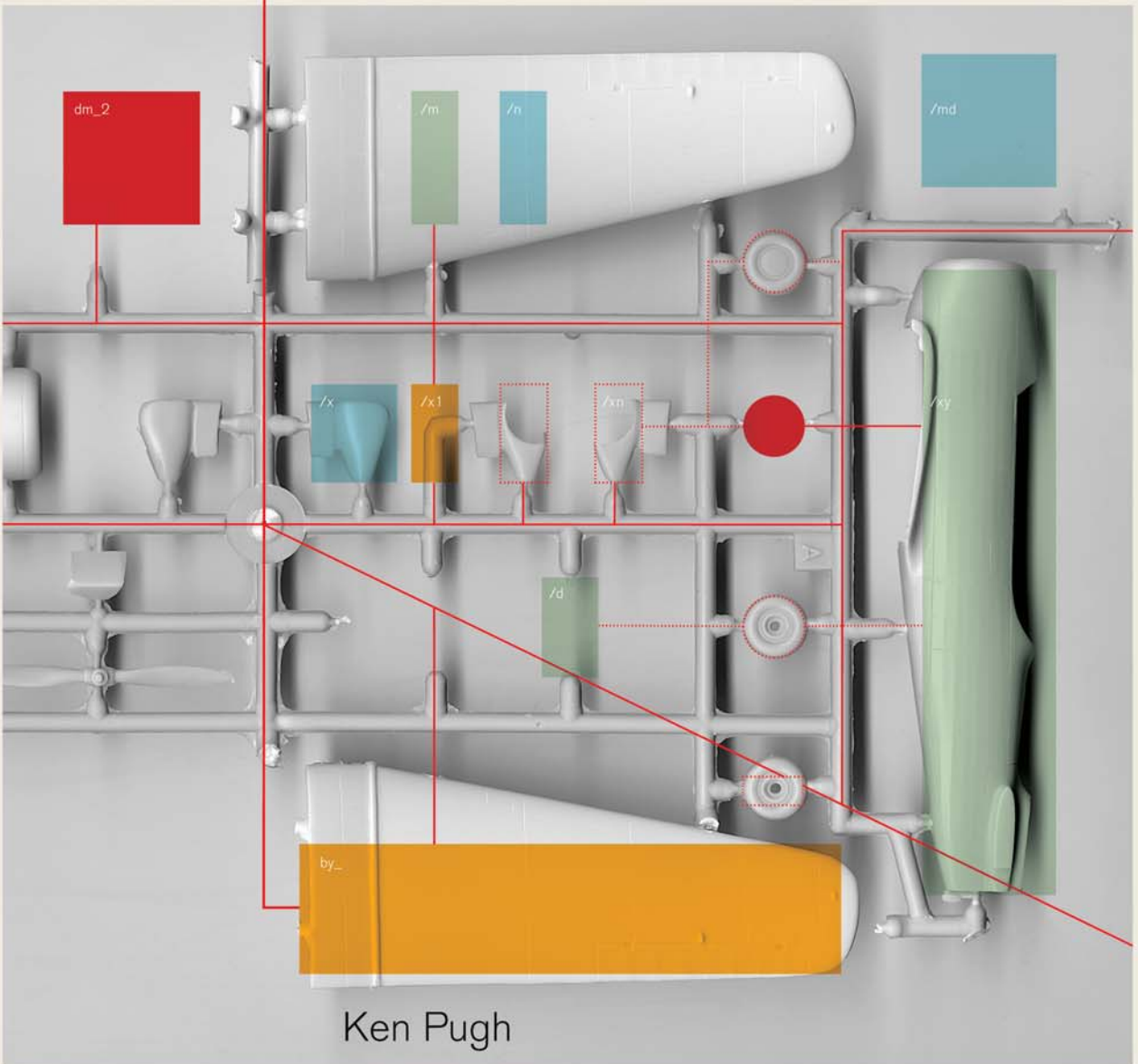


# Prefactoring

Extreme Abstraction • Extreme Separation • Extreme Readability



## The System in So Many Words

**W**E MEET **SAM, THE CLIENT, FOR WHOM WE ARE DEVELOPING A SYSTEM.** Tim, a co-developer, and I interact with Sam to get an overall view of what he wants the system to do through use cases and prototypes. We work together to determine a common vocabulary to describe the system's requirements.

### Meet Sam

Systems are not developed in a vacuum. They are created to meet an organization's needs. The client for whom a system is developed is the source of the requirements for the system and is the final decider of whether a system meets those requirements. Sam, the client, represents a composite of clients for whom I have developed systems over the years.

Sam owns the business CD Rental and Lawn Mower Repair. He started out with lawn mower repair and discovered that people who use lawn mowers like to listen to CDs and they prefer listening to a different CD each time they mow. Therefore, Sam came up with the idea of renting CDs. The service started out as a whim, but it has grown dramatically.

Sam contacted me about creating a system for keeping track of rentals in his store. His current system of using cards similar to library cards works, but it is unable to provide him the reports he feels his growing business requires.

Currently Sam has only one store. Since business is booming, he is considering opening several more stores. He wants us to design the system not only so it works in his store today, but so he can change it easily to accommodate multiple stores tomorrow.

### **Tim the Developer**

Tim introduced me to Sam. Tim studied computer science in college and worked summers at Sam's CD Rental and Lawn Mower Repair. He has been working as a programmer for five years, the last couple of years with me. He is back in school getting a master's degree. We still work together, but mostly remotely. He takes courses and does some teaching, so he is often unavailable during the day, when I am talking with Sam.

Tim represents an amalgamation of programmers with whom I have worked. We work together on approaches to solutions, but usually work separately on code due to the remoteness factor. Because of our physical separation, code readability is extremely important.

### **Sam's Request**

Sam came up with some features that he wants to incorporate into his system. They are based on what he already does with his index cards, as well as additional ideas that he developed in his head. He listed them on a sheet of paper:

- I want to be able to keep track of where each CD is, both when it is in the store and when someone has rented it (including who has rented it).
- I want the system to report when CDs are overdue.
- I want a catalog so that customers can see what CDs are available and what songs are on them.
- When I have multiple stores, the system should show which stores have a particular CD.
- I want to be able to offer discounts to frequent renters.
- I want a charge system that enables me to bill customers per month rather than per rental.

### **Sam's Use Cases**

Sam, Tim, and I will develop the system with this list as the informal requirements for the system. We will transform these requirements into use cases.\* I have found that writing use cases helps both me and the customer understand what the system needs to do. The cases are specified in sufficient detail so that we can appreciate the problem's scope and magnitude. Additional detail can be added later so that when it comes time to design or code, the necessary detail is available. Working through a system's use cases also helps to identify the users who should be consulted when their parts of the system undergo detailed specification and design.

\* For further information on use cases, see *Writing Effective Use Cases* by Alistair Cockburn (Addison-Wesley Professional, 2000).

Sam and I explore, in more detail, the first item on his list:

“Sam, who do you want to keep track of CDs?” I asked.

Sam replied, “What do you mean?”

I continued, “Do you want the customers to be able to check out their own CDs?”

“No,” he answered, “only the clerk should perform checkout.”

“Then let’s go through the steps of checking out a CD, since that sounds like a primary use case,” I suggested.

“How about returning a CD?” Sam queried.

“That sounds like another one, so why don’t you tell me about both of them,” I requested.

Sam explained the checkout and return procedures. After some discussion, I put his narration into the following informal use case descriptions. These use cases describe how the new system should work and are based on how the current manual process operates:

#### *Checkout\_a\_CD*

- a. The customer brings a CD case to the clerk.
- b. The clerk retrieves the actual CD corresponding to the case.
- c. The customer presents his CD Store Customer Card. (The customer already has one of these for the manual system.) The clerk types the customer ID into the system.
- d. The clerk types the CD ID into the system.
- e. The system records that the CD is rented to that particular customer. It prints a rental contract.
- f. The customer signs the rental contract.
- g. The clerk files the rental contract in a file box named “Signed Rental Contracts.”

#### *Return\_a\_CD*

- a. The clerk types the CD ID into the system.
- b. The system records that the CD has been returned.
- c. The system determines whether late fees apply and informs the clerk if that is the case.

An informal use case description can lead to enlightenment, especially when combined with an examination of what can go wrong at each step. I asked Sam what the clerk did in their current system if a customer did not have his CD Store Customer Card. He said that the clerk looks for the corresponding card in the customer file. We captured this information in another informal case.

*Checkout\_a\_CD (extension if no customer ID)*

- a. The customer identifies himself using a photo ID.
- b. The clerk enters information from the ID into the system.
- c. The system searches for the customer ID using the information from the photo ID.  
If no matching customer is found, the system indicates an error.

The use cases presented so far are informal. They mix user actions with system actions. Some authors (e.g., Larry L. Constantine and Lucy A.D. Lockwood in *Software for Use: A Practical Guide to the Models and Methods of Usage Centered Design* [Addison-Wesley Professional, 1999]) suggest that the use cases be written in a more technology-independent manner. Actions that are unrelated to system processing will drop out of the flow and appear as comments. For example, the *Checkout\_a\_CD* use case might look like this:

*Checkout\_a\_CD*

- a. The user enters the customer ID and the CD ID into the system.
- b. The system records the entry. It responds by creating a rental contract for the customer to sign.

Notice that the actual way in which the IDs are entered into the system is not specified. The IDs could be scanned in using a bar code reader, typed in, or spoken and translated by a speech recognition program.

Initial use case descriptions might state exactly how customers expect the system to work. Before implementing the use cases, you can rework them into a more abstract, less-technology-dependent description. Removing the implementation details can help focus on the business policies and procedures that need to be programmed.

## **The Ilities**

The purpose of requirements is not only to validate what the user wants, but also to verify that the implemented systems meet those wants. Requirements include more than just functional specifications as supplied by use cases. These other requirements relate to the software's quality. They are often called the *ilities*, since many of the terms end in *ility*. These quality specifications include reliability, testability, deployability, and performance.

Often these quality requirements are not documented for a system. They are implicit, such as in the case of Sam's system. It is assumed that the system will meet reasonable requirements. For example, its performance should ensure that users do not notice a delay.

For larger, more complex systems, documenting these quality requirements is essential. You can find further information on eliciting and documenting requirements in *Software Requirements*, Second Edition, by Karl E. Wiegers (Microsoft Press, 2003).

## Reinvention Avoidance

Once Tim and I understand Sam’s system requirements, our first step is to determine whether an existing program provides the features that we need. There is no sense in re-creating the wheel if an existing wheel works the way we want. Our goal as developers is to solve the client’s problem, not to just write code.

Sam had searched for a commercial program and did not find anything. It appears that he is in a unique business, so nothing has been written, which is not surprising.\*

We suggested to him that the process of renting a CD is similar to the process of renting a videotape or DVD. He could purchase one of those programs and it would already have many of the features that he wanted. He decided that he would rather have his own custom program instead of dealing with the terminology and handling differences among CDs and DVDs. We recommended that if he decides to expand into selling CDs, we should investigate retail sales systems. A lot of functionality already exists in those systems that should not be re-created. If a preexisting solution fits into the overall system, at least that part of the wheel need not be recreated.†

---

### DON’T REINVENT THE WHEEL

**Look for existing solutions to problems before creating new solutions.**

---

Since Sam wants us to develop a custom system, Tim and I start to analyze the problem. We need to outline the concepts involved in the problem and clarify our understanding of what needs to be solved.

## What’s in a Name?

Names are important, not just for the code but also for requirements and analysis. If you don’t know what you’re talking about, it’s hard to design for it.

Sam described how he wants to keep track of the CDs. He also desired a catalog of all the CDs that he has for rent.

“So, what is a CD?” I asked Sam.

\* Finding existing solutions can be problematic. Sometimes it can be hard to describe the solution you seek in such a way that Google™ can find a match.

† See *Software Tools* by Brian W. Kernighan (Addison-Wesley Professional, 1976) for the earliest discussion I have found on the issue of using tools to create solutions.

He paused for a moment and looked at me with a questioning expression on his face. He must have thought I was crazy. “You know, one of those round things you put in a CD player,” he said.

“So, when you said you want a CD catalog, do you mean you want an entry in it for every round thing you have in your store?” I asked.

He paused again. “No, I want only one for each title, regardless of how many copies I have in the store.”

I suggested, “So, let’s decide to use two terms, one for the CD title and one for the CD copy. This way we minimize the opportunity for misunderstanding. What do you want to call each thing?”

“Now I see what you mean,” he replied. “What do you suggest?”

I replied, “Let’s call the title a *CDRelease*, and the other a *CDDisc*. We could use the name *CDTitle*, but that would start to get confusing when we talk about the title of a *CDTitle*. To clarify what we mean even further, we can describe each term with a sentence:

*A CDRelease* is a CD identified by its Universal Product Code (UPC).

*A CDDisc* is a physical copy of a *CDRelease*. *CDDiscs*, not *CDReleases*, are what are rented.

“Now is it possible that a CD which a customer would be looking for would be related to two different UPCs?” I asked.

“It’s possible,” he said. “But I don’t think we need to worry about that. One would usually have the term *rerelease* in its title.”

“We can always revisit this question if things change,” I said. “Let’s alter your requirements and the use cases to utilize these terms.”

At this point, Sam and I came up with the following list of modified requirements:

- Keep track of where each *CDDisc* is, both when it is in the store and when someone has rented it (including who has rented it).
- Report when a *CDDisc* is overdue.
- Have a catalog so that customers can see which *CDReleases* are stocked, what songs are on each *CDRelease*, and which corresponding *CDDiscs* are available in the store.

Here is a modified use case:

*Checkout\_a\_CDDisc*

- a. The user enters the customer ID and the *CDDisc* ID into the system.
- b. The system records the entry. It responds by printing a rental contract for the customer to sign.

Names are subjective. As long as you and the client agree on a name, it does not matter if the name makes sense to the outside world. Here are some other possibilities for names of these two concepts:

*CDUPC*

A CD identified by a UPC

*CDPhysical*

A physical CD of a particular CDUPC

*CDCatalogItem*

A CD identified by a UPC

*CDRentalItem*

A physical CD copy of a particular CDCatalogItem

Attributes of these classes should use the same names as the customer uses. If the customer uses a full name, avoid making up an abbreviation for it. If the customer uses an abbreviation or acronym, use that. If you have a hard time recalling what the short form means, ask the customer to supply a longer name.

---

**A ROSE BY ANY OTHER NAME IS NOT A ROSE**

**Create a clearly defined name for each concept in a system.\***

---

## **Splitters Versus Lumpers**

If the world were perfect, you would have exactly one unique name for each concept in a system. In this imperfect world, having two concepts with the same name leads to confusion. In Sam's case, the term *CD* was applied to both a *CDRelease* and a *CDDisc*. Separating the two concepts with two names clarified the requirements.

Using two different names for a single idea can also be confusing, albeit less so than two ideas with a single name. Referring to a physical CD as both a *CDDisc* and a *CDPhysical* might be justified by political measures. ("This department calls it this and that department calls it that.") Sam referred to the act of renting a CD as both renting a CD and checking out a CD. If these two terms really encompass the same operation, the duality of reference can be annoying, but might not be confusing.

Sometimes it is hard to determine whether you have two independent concepts or one. Try making up a one-line definition for a name. If it is difficult to create a simple definition, go ahead and use two names. Later on, if you find that the distinction was meaningless, you

\* See <http://www.literateprogramming.com/> for a discussion of names.

## AIRLINE FLIGHTS

I worked with a group responsible for developing an airline reservation system. There is a lot of interesting terminology in the airline business. Think of your concept of a “flight” as a passenger. When you fly from Boston to Peoria, do you say that you are taking a flight to Peoria? Do you say that, even if you make a connection in Chicago? In that case, would you refer to it as a “connecting flight”? If you were on Flight 80 from Boston to Chicago and on Flight 100 from Chicago to Peoria, would you say that you are on two different flights? Suppose that Flight 80 landed in Albany on the way to Chicago. Are you on three different flights?

The reservations people agreed on the following definitions:

### *Segment*

A trip with a single departure and arrival. For example:

Boston to Albany on Flight 80

Albany to Chicago on Flight 80

### *Flight*

A numerically designated set of segments, with the departure location of the subsequent segment the same as the arrival location of the previous segment. For example:

Flight 80 Miami to Boston to Albany to Chicago to Seattle

Flight 100 Dallas to Chicago to Peoria

### *Leg*

A set of one or more consecutive flight segments on a flight. For example:

Flight 80 Boston to Albany to Chicago

### *Journey*

A set of one or more legs for a passenger that takes a passenger from a departure location to an arrival location. For example:

Flight 80 Boston to Albany to Chicago and Flight 100 Chicago to Peoria

These consensus definitions made it easy to create higher-level concepts, such as “marriage.” A marriage is a journey, for which the cancellation of one leg results in the cancellation of all legs of the journey.

can always declare the two names to be synonyms. Suppose that Sam and I came up with the terms *CDAlbum* and *CDRelease*. We might distinguish them by stating that a *CDAlbum* is a collection of songs with a title given to the set, and a *CDRelease* is a collection of songs that was released on a single *CDDisc*.

The conversion from one style of architecture, design, or coding to another is not necessarily symmetrical. Suppose that a single name has been used to denote two ideas. Later you decide that you need to replace that name with appropriate names for each idea. You need to examine each usage of the term carefully to determine which of the two concepts it represents. On the other hand, suppose that you have used two different names for a single concept. If you want to combine those into a single name, you can do a simple global replacement.

For example, suppose we have a class called `Message`, which represents messages displayed to the user. We think at the beginning that these messages are going to behave differently, so we divide them into `WarningMessages`, `ErrorMessages`, `SevereErrorMessages`, and `ReallySevereErrorMessages`. We make every message an object of one of these four classes. Later on, we realize that `SevereErrorMessages` and `ReallySevereErrorMessages` really do not behave differently. We can eliminate the distinction using a simple search and replace. Conversely, if we had not distinguished the two and later found that there should be a difference, we would have to look closely at each object of `SevereErrorMessage` to determine whether it should be categorized as `ReallySevereErrorMessage`.

---

## SPLITTERS CAN BE LUMPED MORE EASILY THAN LUMPERS CAN BE SPLIT

**It is easier to combine two concepts than it is to separate them.**

---

## Clumping

When Sam described his customers in detail, he mentioned that he needed to keep track of each customer's home address, including street, city, state, and Zip Code, as well as credit card billing address, including street, city, state, and Zip Code.

I asked him, "Do both of those addresses contain the same information?"

He replied affirmatively.

I said, "Then let's just describe the combination as an `Address`. That way, you don't have to keep mentioning all the parts unless there is something different about them."

"OK," he answered.

We clumped the data into a class, as follows:

```
class Address
{
    String line1;
    String line2;
    String city;
    String state;
    String zip;
}
```

At this point, we simply clump the related data, even though we have not assigned any behavior to the class. This data object helps in abstraction and in cutting down parameter

lists. Even though the class contains only data at this point, we might be able to assign responsibility to it later on.\*

Clumping and lumping look similar, but they have distinctly different meanings. *Clumping* involves combining a set of attributes into a single named concept. The attributes should form a cohesive whole. *Lumping* involves using a single name for two different concepts. Clumping is an abstraction technique, which makes for an efficient description of a set of data. Lumping can hide relevant distinctions between concepts.†

---

## CLUMP DATA SO THAT THERE IS LESS TO THINK ABOUT

**Clumping data cuts down on the number of concepts that have to be kept in mind.**

---

## Abstracting

In creating a description of a use case or a model of a possible class, avoid using primitive data types. Pretend that ints or doubles do not exist. Almost every type of number can be described with an *abstract data type* (ADT). Items are priced in Dollars (or CurrencyUnits, if you are globally oriented). The number of physical copies of an item in an inventory is a Count. The discount that a good customer receives is denoted with a Percentage. The size of a CD is expressed as a Length (or LengthInMeters or LengthInInches if you are going to be sending a satellite into space). The time for a single song on a CD release could be stored in a TimePeriod.

Using an ADT places the focus on what can be done with the type, not on how the type is represented. An ADT shows what you intend to do with the variable. You can declare the variable as a primitive data type and name the variable to reflect that intent. However, a variable declared as an abstract data type can have built-in validation, whereas a variable declared as a primitive cannot.

Each ADT needs a related description. For example, a Count represents a number of items. A Count can be zero or positive. If a Count is negative, it represents an invalid count. Declaring a variable as a Count conveys this information. You can create variations of Count. You may have a CountWithLimit data type with a maximum count that, if exceeded, would signal an error.

\* Not all objects have behavior. Objects that contain just data (sometimes called *data transfer objects*) are useful in interfacing with GUI classes and passing as objects between networked systems. Data transfer objects are covered in Chapter 7.

† One reviewer notes that clumping without a responsibility definition for the class can lead to data-polluted classes. Clumping should also involve assigning operations to the clumped concept.

You can place limits on many different data types. For example, Ages (of humans) can range between 0 and 150 years, SpeedLimits (for automobiles) between 5 and 80 mph, and Elevations (for normal flying) between 0 and 60,000 feet.\* All these types can be represented by an `int` or a `double`, but that is an implementation issue, not an abstraction issue.

Abstract types can contain more than just validation. A price can be represented in Dollars. The string representation of a Dollar differs from the string representation of a double. A string for Dollar has at least a currency symbol and perhaps some thousands separators (e.g., commas). Multiplying a Dollar by a number can result in a Dollar with cents, but not fractions of a cent. Here is a possible Dollar class:

```
class Dollar
{
    Dollar multiply_with_rounding(double multiplier);
    Dollar add(Dollar another_dollar);
    Dollar subtract(Dollar another_dollar);
    String to_string(); //
};
```

If your language provides the ability to define operators for a class (such as `+` and `-`), you can use arithmetic symbols for the corresponding operations. You can also use the appropriate method name to have a Dollar be converted automatically to a String if it appears in an appropriate context. How you represent the abstract data type that you use for a value is an implementation detail. You can make up a class for each type. If you work with C++, you can make up typedefs for each simple type for which there is no additional functionality. For other languages, you can convert some simple types into primitive types. In that case, you might want to use variable names that include the type (e.g., `double price_in_dollars`).

---

## WHEN YOU'RE ABSTRACT, BE ABSTRACT ALL THE WAY

**Do not describe data items using primitive data types.**

---

This guideline suggests using explicit typing in describing the problem and the solution. By using abstract data types in the beginning, you are, in effect, more abstract. If you explicitly type all your attributes and parameters, you can always switch to implicit typing if the explicit typing gets in the way. It is much harder to go in reverse.†

\* Once upon a time, Montana had no speed limit other than “reasonable speed.” The upper limit still could be a reasonable number (e.g., 200).

† For an article on strong testing with languages that have implicit typing, see <http://www.artima.com/weblogs/viewpost.jsp?thread=4639>.

## ADTS AND SPEED OF DEVELOPMENT

I was the chief judge of the Droege Developer Competition. In this event, pairs of developers competed to code a project for a nonprofit organization in one day. They were provided specifications and the results were judged the next day. Developers who used a product from Magic Software (<http://www.magicsoftware.com/>) consistently scored high and often won the competition. That product includes the ability to define data types that contain validation rules, display rules (e.g., drop-down lists or radio buttons), and formatting rules. The record layout in the tables is defined using these data types, rather than primitive Structured Query Language (SQL) types. To display a record, the record fields were placed onto the display window. The requisite formatting and validation for each field were already coded. Additional validation could be added when necessary.

The data typing feature, which is a particular implementation of abstract data typing, was key in the ability to create a working system quickly.

### Not Just a String

A more descriptive data type can represent many data types represented typically by a String. For example, although a name can be declared as a String, you could declare it as a clumped data object:

```
class Name
{
    String first_name;
    String last_name;
    String title;           // e.g. Mr. Mrs.
    String suffix;        // e.g. Jr. III
};
```

To avoid the “everything is a String” syndrome, come up with a different type name to describe a variable that holds a set of characters that does not have any validation, formatting, or other meta-information associated with it. Suppose you decide on `CommonString`. Use that name in place of String to declare the data types of attributes, and reserve String as an implementation type. Then ask the question “Is that attribute really a `CommonString`?”

Let us revisit the Address class. Using `CommonString`, we can describe the class as follows:

```
class Address
{
    CommonString line1;
    CommonString line2;
    CommonString city;
    CommonString state;
    CommonString zipcode;
}
```

`CommonStrings` can contain any characters, just like an `int` can contain any integer values (within hardware limits). In `Address`, some fields are definitely not `CommonStrings`. A state is not a `CommonString`. Only certain values represent a valid state. For U.S. postal addresses, if we use abbreviations to represent the state, the abbreviation must appear on the U.S.

Postal Service’s official list of abbreviations. So the state should be declared as a data type called `State`. This data type can provide an appropriate validation mechanism. That mechanism can check to see that a string is in the official list, or it can supply a set of strings of all official abbreviations for use in a drop-down display box.

A U.S. Zip Code is not just a `CommonString` either. You can describe it as a `NumericString` data type (e.g., one with all digits), as a `FormattedString` data type (one with five digits plus a dash plus four digits), or as a `ZipCode` data type. If any combination of digits was valid, using `NumericString` or `FormattedString` might be appropriate. However, declaring the attribute as a `ZipCode` type allows us to abstract away its actual representation.

Do not combine classes simply for implementation purposes. You can define both `SocialSecurityNumbers` and `PhoneNumbers` as strings of digits with two dashes. That does not make them equivalent—that is just accidental cohesion. They are two distinctly different classes with different validation. A phone number in the U.S. cannot begin with a 1 or a 0. Certain ranges of Social Security numbers are not used. These numbers can use the same type of formatted string for input or display purposes, but the semantics of each class are entirely different. You would never send a Social Security number to be dialed, nor would you attempt to record payroll taxes against a phone number. (All right, someone at some time will come up with a counterexample, so perhaps I should never say “never.”)

Much data that might be a `CommonString` can be assigned its own data type. For example, filenames are usually typed as strings, but they cannot contain certain characters. In the Windows world, you cannot have any of the following characters in a filename: `\`, `/`, `:`, `*`, `"`, `?`, `<`, `>`, or `|`. A `FileName` data type can represent a filename and enforce this limitation. An advantage of using a data type becomes apparent in graphical user interface (GUI) development. For instance, the user interface code could recognize the `FileName` data type and automatically insert a browse button next to a text field.

On the Web, parameter values for the Hypertext Transfer Protocol (HTTP) commands `GET` and `PUT` use encoded strings. Characters that are not alphabetic or numeric are encoded using their hexadecimal values. The encoded string is sent to the web server. Although the unencoded string and the encoded string are both implemented with strings, they are different. You can have invalid encoded strings—ones with unencoded punctuation, such as a hacker might send to a server. You could use an `EncodedWebString` class to represent strings on a server. If an input were not validated as an `EncodedWebString`, it would be rejected.

---

## MOST STRINGS ARE MORE THAN JUST A STRING

**Treat `String` as a primitive data type. Describe attributes with abstract data types, instead of as `Strings`.**

---

## Constant Avoidance

Similar to the way in which most strings are more than strings, most constant values are more than just constants. A constant value can usually be assigned a name that denotes the meaning of that constant. Avoid using the explicit value in a specification or executable code.\* Declare the value as a constant and use the name of that constant in the document or the code.

If Sam mentions that the late fee for a rental is \$3, I create a constant:

```
Dollar RENTAL_LATE_FEE = 3.00;
```

When reading the relevant documents later on, I need not concentrate on the actual value, only on the assigned name. Suppose this value was not transformed into a constant and the value of 3.00 was used frequently in the documents for other purposes. If I went searching for it, I would have to examine each appearance carefully to see if it was a reference to the rental late fee or to some other value.

You might not get rid of every constant value. The value 0 often appears in initializing variables or setting the initial index for an array. There is little to be gained by creating a named constant for zero.

If the value that a name represents is subject to change, the value should be kept with a configuration mechanism. In that case, the code would use the symbolic name to look up the configured value. The configuration mechanism could use an XML configuration file, a database table, or another form of persistence to store the values. For example, RENTAL\_LATE\_FEE is probably something that should exist in a configuration file rather than a con-

---

### NEVER LET A CONSTANT SLIP INTO CODE

Use a symbolic name for all values.

---

## Prototypes Are Worth a Thousand Words

It is often said that a picture is worth a thousand words. A prototype is like a picture. A user interface described in text is often harder for the customer to visualize than the same interface described with a diagram or picture. Use cases can provide excellent textual descriptions. A prototype (or screen mockup) gives a more concrete perspective on a program's intended operation. The prototype can spark feedback from the client in both the program's operation and in missing requirements.

\* Michael Green, a reviewer, called this principle "No magic numbers!" after having to deal with numbers that could not be changed or removed (without negative side effects in apparently unrelated code). He could find no one who knew what they were for or why they were there.

## DON'T THROW AWAY INFORMATION

When I was presenting these prefactoring guidelines in a talk at the Software Development Conference, Jerry Weinberg made an interesting observation about some of the guidelines in this chapter. He stated that they revolve around the central principle of not throwing away information. For example, describing a price as a `double`, rather than as a `Dollar`, decreases the information about the price. Lumping a group of concepts into a single class, rather than splitting them into multiple classes, hides information. Now, if I only could have convinced my mother that my comic book collection was really information...

One of the dangers of making a perfect-looking GUI for a prototype is that the interface represents the program to the user. If the interface is complete, the user might expect that the system is almost complete. Some user interface experts suggest that interfaces be designed using whiteboards or Post-it notes. If you are programming in Java, you can use the Napkin Look and Feel (<http://napkinlaf.sourceforge.net/>). Tim and I created a rough-draft prototype of the screens for the uses cases we worked on with Sam (Figure 2-1). We went

The figure displays three screenshots of a GUI for 'Sam's CD Rental'. The top screenshot shows the main menu with 'CheckOut' and 'CheckIn' buttons. The middle screenshot shows the 'CheckOut' dialog with input fields for 'Enter Customer ID' and 'Enter CDDisc ID', and 'Done' and 'Cancel' buttons. The bottom screenshot shows the 'CheckIn' dialog with an input field for 'Enter CDDisc ID' and 'Done' and 'Cancel' buttons.

FIGURE 2-1. Rental screens

over it with Sam. The cases are simple, so he had no changes in its interface. He did note that the buttons should use a large font so that he could read them without his glasses.

---

## **PROTOTYPES ARE WORTH A THOUSAND WORDS**

**A picture of an interface, such as a screen, can be more powerful than just a description.**

---