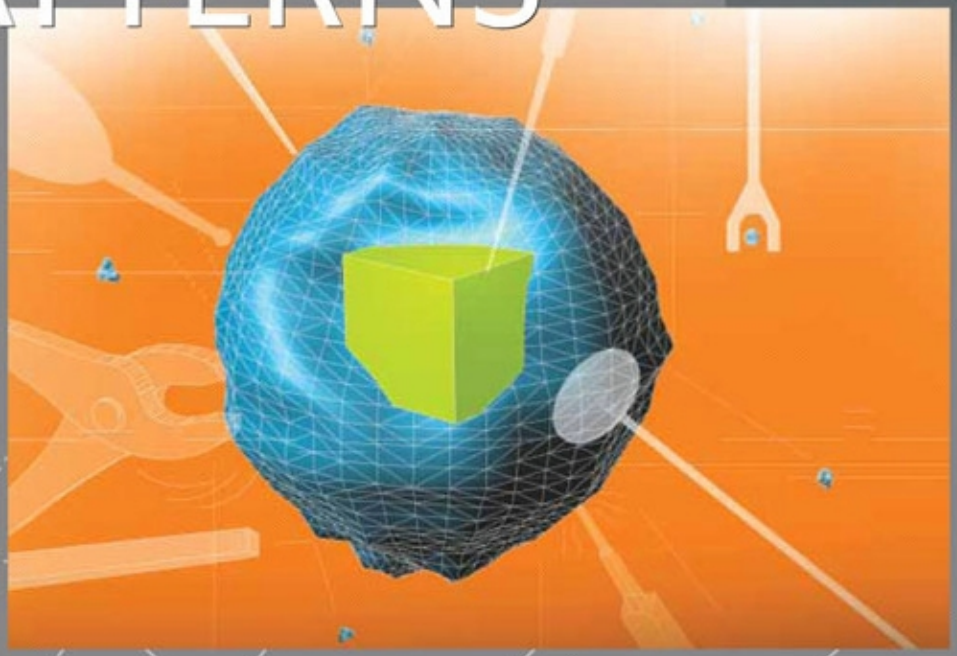
Microsoft

BEST PRACTICES

SOFTWARE REQUIREMENT PATTERNS



Stephen Withall

Foreword by Karl E. Wiegers
Author of Software Requirements

Microsoft®

Software Requirement Patterns

Stephen Withall

Factor 5: Simplicity Inflicting as few distinct parts on people as possible. This isn't the same thing as avoiding multipart IDs—as the discussion on that subject demonstrated (in the big paragraph preceding). No, it means needing to *make visible* as few parts as possible.

Factor 6: Who allocates A person or a machine? It's easier for a person to devise a name and easier for a machine to allocate a number.

Factor 7: Connection to other IDs Basing an ID on another ID that's already known. This is convenient when the base ID is known to the user. For example, banks don't inflict on their customers huge transaction numbers that are unique within their system. Rather, they allocate sequential numbers for each customer that are unique within the bank when tacked on to the customer ID.

Factor 8: Flexibility Being able to handle IDs of unknown type or several different types. This might occur if we must accommodate IDs supplied by an external system or by several different external systems.

Some of these factors go hand-in-hand; others conflict with one another. In particular, the uniqueness factor tends to conflict with the others.

Content

An ID requirement should contain the following:

1. **Owner entity name** To what are we allocating IDs of this kind? Examples might be *customer* and *employee*. In rare situations, we may want more than one type of entity to share the same ID scheme. For example, we could have transaction IDs that are used for both customer orders and defective item returns. Do this if there's a genuine business need, but avoid it otherwise.
2. **ID name** If the owner entity has more than one ID, to clearly distinguish one from another.
3. **ID form** Restrict an ID requirement to what's actually needed; don't make decisions about the form of IDs if you don't have to. Say what you need to achieve, and no more. One reason is that allocating IDs can have performance impact, especially if all IDs must be allocated by a single allocator (which would represent a bottleneck). Give developers the freedom to allocate IDs in the most efficient way if you can.

   For IDs that contain characters, state whether they're case-sensitive. Is "MAC" the same as "Mac"? IDs that *aren't* case-sensitive are usually better—to avoid confusion.
4. **Scope of uniqueness** What are the bounds of the context within which an ID of this kind is unique? It's important that you give some thought to this and not make the scope of uniqueness too small, because it might make life awkward later in the life of the system. For example, if you install multiple instances of your system that all allocate sequential customer numbers and later you want to merge them all into a single database, you'll need something in addition to the customer numbers to distinguish the customers. You could include this something (perhaps a company ID or office ID) in the scope of uniqueness from the outset.

   If the scope of uniqueness isn't stated in a requirement, assume that the ID is to be unique for entities of this type within the scope of the system but not beyond it. Take note of that "of this type" qualifier, because it means we can have the same ID for different things—which is usually no problem, because they're unlikely to collide. For example, we could use "JPY" as the ID of currency details about the Japanese Yen and also as the ID of an employee with

these initials. If you want to have only one entity with the ID "JPY," you must define the scope of uniqueness such that it spans every type of entity affected.

5. **How allocated** IDs don't appear magically out of thin air, so where do you want them to come from? There are three main ways IDs can be allocated, each with its own advantages and drawbacks:
   a. **Automatically by the system.** For this way, you could describe how the system is to allocate IDs, or you could leave that as a design question. The most common approach is sequential numbers—in which case you should consider the first value (start at one?), whether it should be reset (and, if so, when), how big it must be to avoid the risk of it overflowing, and what should happen if it does overflow (it mightn't bear thinking about!).
   b. **User choice.** Let the user type in the value they'd like to use as the ID. The system must cater for them entering a value that's already used as an ID. When this happens in the case of a *user ID*, it's common practice to ask them to choose a password at the same time and to say there's a problem with either the user ID or password, so as not to alert the user if they stumble upon someone else's user ID. (It doesn't fool an alert user, though.)
   c. **From an external source, such as another system.** A value a user possesses and is asked to type in counts as from an external source (for example, a credit card number), but there is the extra risk of them entering it incorrectly. So validate it thoroughly before using it as an ID.

   There are variations that look like combinations of these ways, but they usually boil down to one of the three. For example, the system could generate a *suggested* ID and then let the user modify it, thereby boiling down to user choice. Or the system might create an ID based on the user's choice—if it's already used, say—thereby boiling down to one of the first two, depending on whether we give the user the chance to modify the ID the system created.

   Just for good measure, there are a couple of obscure ID allocation factors to keep an eye open for. First, *hidden significance*: might an ID's value reveal something you'd rather it didn't? For example, if a customer is allocated customer number 0000012, you're revealing you've done little business and the customer might trust you less. Second, *continuity*: if an old system allocates IDs in a particular way, doing it differently in a new system might have unexpected consequences, even if it functions perfectly. For example, veteran employees in some companies take pride in having a low employee number; a new system that allocates employee numbers in a different way could be resented. Your ID requirement might respond by demanding that each ID allocated must be higher than all previous IDs.
6. **Display format** As per the data type requirement pattern. This is usually only warranted for complex ID forms, but you could use it to state, say, that numbers are to be shown with leading zeroes removed (or present, up to a specified number of digits, if you're so inclined). If an ID has multiple parts, how should it be displayed to people, to make each of the parts clear? For example, we could add separators like dashes and dots. Also consider describing how a multipart ID is to be typed in on a screen: should it be divided into multiple input fields?
7. **Sort order** If it's different from the obvious, or if you want to prevent the development team being silly. This applies especially with multipart IDs. For example, you may want "61-001" to come before "123-001," which is likely to happen only if sorting treats the ID as two separate numbers. Avoid sort orders that aren't intuitive or that involve interpreting the meaning of any part (such as extracting the date meaning from a form like, say, "21AUG12").

When sorting on textual values, consider whether you need to worry about the sort order of characters used in languages other than yours. For example, if it's important that "é" and "ë" are sorted as if they are next to "e" (or the same as "e"), say so.

8. **Reuse conditions, if necessary** When an entity expires, you may want to be able to reuse its ID. For example, if an employee identified by their email address "chris@ourco.com" departs, a new employee called Chris might want this email address. Avoid reusing IDs if possible, because it opens the door to ambiguity (and consequently mistakes): the new Chris might carry the can for the mistakes of the old one. But if the system must permit reuse, say so, and impose conditions to minimize the chances of the wrong entity being identified. If no reuse conditions are stated, it's reasonable to assume that IDs of this type won't be reused.

Template(s)

Summary	Definition
«*Owner entity name*» [«*ID name*»] ID	Each «*Owner entity name*» shall have a unique ID that is in the form of «*ID form*» allocated by «*How allocated*». [«*Display format*».] [Each «*ID name*» shall be unique «*Scope of uniqueness*».] [«*Sort order statement*».] [«*Reuse conditions statement*».]

Example(s)

Summary	Definition
Customer number, with check digit	Each customer shall be uniquely identified by a customer ID that is in the form of a number allocated sequentially plus a check digit calculated over that sequential number using the «*algorithm name*» algorithm (as explained at «*algorithm location*»).
Order ID	Each order shall be uniquely identified by an order ID that is in the form of the number of the customer that placed it plus an order number allocated sequentially for that customer, starting at one for the customer's first order. Order IDs shall be displayed in the form "«*Customer number*»-«*Order number*»" (for example, "10762-1").
Employee ID	Each employee shall have an employee ID that is a five-digit number allocated externally (and entered manually when an employee's details are first entered). Each employee ID shall be unique within the scope of the system; even if an employee departs, their employee ID shall not be reused for another employee.
Loan approval decision rule ID	Each loan approval decision rule shall have an ID by which it can be referred. Rule IDs shall be allocated in such a manner as to not become invalid or incorrect when another rule is added or removed. For example, a rule's sequence in a list of rules cannot be used because that will change if an extra rule is inserted before it.

Extra Requirements

An ID requirement is usually self-contained. But extra requirements might be needed for the following couple of topics:

1. Rules to be followed for every "**invisible**" ID scheme that developers choose to add. An "invisible" ID is one that is used internally by the system in order to function but that is not normally visible to users. For example, if we wanted to give customers the option of changing their customer ID, we might assign them a second, invisible customer ID that never changes. We can't demand anything for a particular ID scheme that's not itself mentioned in requirements, but we can write requirements to apply to all invisible IDs (or more widely to all IDs).
2. Continuity of IDs from a system we're replacing. If our system is taking over from a previous system, we'll have to import all its old data—including all its old IDs. Once our system goes live, it needs to take over from where the old system left off (or, at least, it must start allocating new IDs sensibly, without breaking anything). What must we do to achieve this? What pitfalls do we want to avoid? This can be a significant issue if we're replacing multiple old systems that used different ID schemes.

Here's an example pervasive requirement of the first kind, that cover *all* IDs (not just invisible IDs):

Summary	Definition
All IDs viewable	Every ID that can be used to identify an entity shall be viewable by some means, regardless of whether that ID is intended to be seen by normal users. The purpose of this requirement is to assist developers, testers, and auditors in examining the workings of the system.

Considerations for Development

It's possible to add extra ID schemes beyond those specified in requirements, and it's sometimes necessary. In particular, it can be useful to add "invisible" IDs that users need never see. If you use these "invisible" IDs for most processing, less reliance is placed on "visible" IDs, which makes it easier to change them. Conversely, if it **is** possible to change an ID, you're likely to need unchanging "invisible" IDs to hold everything together.

Martin Fowler's *Analysis Patterns* (1996) has an identification scheme analysis pattern.

Considerations for Testing

Test the overflow of sequentially allocated numbers. That is, contrive to reach the highest possible ID and then see what happens when you attempt to go past it.

Is it possible for unusual or special values to be used as IDs, such as zero or all spaces? If so, are they properly handled? If not, what happens if you *try* to these values as IDs?

If an ID allocated to an expired entity can later be reallocated to a different entity, scrutinize this situation closely. Do the two entities ever get mixed up? Make sure information about one isn't presented as if it belonged to the other.

6.4 Calculation Formula Requirement Pattern

Basic Details

Related patterns:	Data type
Anticipated frequency:	Between zero and a dozen requirements, depending on the nature of the system
Pattern classifications:	Pervasive: Maybe

Applicability

Use the calculation formula requirement pattern to specify how to calculate a particular kind of value, or how to determine a value via a process of logical steps.

Do not use the calculation formula requirement pattern directly for unduly complex formulae or logic. A mathematical treatise doesn't lend itself to squeezing into a set of requirements. In such cases, write a requirement that refers to an authoritative source that defines the calculations; don't attempt to duplicate such a source. Nevertheless, the calculation formula requirement pattern may still be useful in suggesting points to mention in the requirement.

Discussion

Getting calculations correct is vital, especially when they affect the revenue of a business. The way a calculation is to be performed deserves to spelled out clearly so that everyone can see and scrutinize it. It's made more important because computers don't let us observe the act of calculating and formulae embedded in software cannot be observed by nondevelopers. It's surprising, then, that critical calculations are often not mentioned anywhere in a system's documentation, leaving the business to trust that unguided developers do the right thing. That's not good enough—and it's the requirements' responsibility to see it doesn't happen.

Write a requirement for every significant calculation the system must perform. Don't be afraid to spell out even the most obvious formula—first, because no formula is so obvious that every reader can be sure to know it, and second, because it might turn out to be not so obvious after all when you sit down to specify it properly.

Formulae for calculations need to be stated precisely, or else they risk being misinterpreted and implemented incorrectly. A calculation that's *close* and *looks about right* to the naked eye is still wrong. Expose as much as possible how the system works, and what it's doing. Let users understand, and they'll be more likely to spot when something's not working properly and they'll have more faith in the system (because it becomes less of an inscrutable monster).

For convenience, this requirement pattern uses the term "calculation" to mean a formula that yields a single result value from a number of variables and "determination" to mean a list of instructional steps to follow to yield a result value. A determination step can itself contain a calculation formula.

First get your facts straight. Don't just write what you think is common sense off the top of your head, because it may be more complicated than that. Do your homework: find a reliable source that explains your formula, if you can. It might contain an unexpected twist or two. For instance, if you think calculating simple interest for a bank account is . . . well . . . simple, you need to think again, because there are subtly different alternatives (which are spelled out in the example requirements

later in this section). It's not until you discover something unexpected that you even appreciate the obvious formula might be inadequate.

Be alert to possible geographic variations (for example, again, how interest is calculated), or variations from one industry to another, or from one company to another. Don't be parochial. If you intend to support only one environment (or some other subset of all possibilities), say so explicitly. Allow variations to be accommodated via configuration where possible.

Content

A calculation formula requirement should contain these items:

1. **Value description** What is it that this formula is used to calculate?
2. **The formula itself, in the form "«*Value name*» = ..."** Choose the name of the value carefully, because the formula constitutes a definition of this name. Make the name clear and unique within the scope of the system. Split it into more than one formula if it makes it easier to explain.
3. **Explanations of all the variables used** For each variable state all the following that are relevant:
   a. **Variable name**. Capitalize the first letter to make it easier to distinguish in the formula. Use names consistently, especially when a value is the result of another calculation.
   b. **Origin**. Where does the variable come from? It might be the result of another calculation formula.
   c. **Data type** (if relevant). If a logical data type has already been defined for this value, refer to it. Otherwise, describe what kind of value it is (a whole number, a percentage, and so on) if it's not clear.
   d. **Allowed values or range**, if relevant. For example, if the variable is a percentage, perhaps it must be in the range 0–100%.
   e. **Number of decimal places**, if relevant.
4. **Calculation refinements** If the result must be calculated to a particular level of precision or rounded in a particular way, say so. It's usually reasonable to assume that calculations will be performed accurately, but software finds innumerable ways to trip the unwary—so if you're worried that 2 + 2 = 4.000012, explicitly insist on perfect accuracy.
5. **Applicability limitations** If this formula is suitable only in certain circumstances, say what they are (for example, for use in the United States only) so that if the system is used outside these bounds, it's clear there might be problems.
6. **Reference** Few calculation formulae used in systems are novel. Whenever possible, cite an external reference (or more than one) that explains the formula. This is important when there is a standard, law, or government or industry regulation that defines precisely must be done, especially for monetary calculations. Deviating from a prescribed way to calculate something might have unpleasant consequences.
7. **An example** If you think it makes things clearer. Or more than one.

Split a formula into more than one requirement if a single requirement would be large and unwieldy or if a subcalculation is used in more than one formula. Note that if you're using subcalculations, the variables listed in the original requirement might not be all you need, because a subcalculation might use additional variables. For example, the interest calculation formula that follows has a "number of days" variable, but the requirement specifying it uses "start date" and "end date" (among other variables).