Considerations for Testing

A user registration function is basically a data entry function and should be tested as such. Test the entry and validation of all values. Test that validation is not performed only on an untrusted client device (a remote user's PC). Test that any values that are intended to be secret (such as passwords) are treated as such, and cannot be viewed using the system (such as via inquiries or reports or obtained from logs or from querying the database).

Testing that a user registration process is secure cannot be done solely by *using* the system. It's a specialized activity (as discussed in the user authentication requirement pattern).

11.2 User Authentication Requirement Pattern

Basic Details

Related patterns:	Accessibility
Anticipated frequency:	One or two requirements
Pattern classifications:	Functional: Yes

Applicability

Use the user authentication requirement pattern to specify that a person must make their identity known to the system before they can access anything non-public or anything for which they cannot remain anonymous (in short, anything for which they must log in).

Discussion

Authentication is the process by which a registered user declares who they are and proves this assertion to the system's satisfaction. This requirement pattern says little on the subject beyond what is of interest to requirements; it doesn't worry about the considerable challenges of implementing a secure system. The most widespread form of authentication in commercial systems is just asking the user to enter their user ID and password, and that's what this requirement pattern concentrates on. The common name for user authentication is "logging in." (This book prefers logging *in* to logging *on*—on the basis that *entering* a system is a better metaphor than *climbing atop* it. Let's also point out that, as a verb, to "log in" is two words: "to log someone in" sounds fine, and is only possible if there is a gap in which to squeeze "someone.")

There are three accepted ways to check that a person is who they claim to be: something they **know** (such as a password), something they **have** (such as an ID card), and something they **are** (using a part of their body that can be distinguished from that of everyone else—such as a finger or an eye). Depending on how secure you want your authentication process to be, you can ask for more than one (a password *and* an ID card *and* a fingerprint, if you insist). There are various factors that affect what's best in a particular situation, including:

Factor 1: The potential damage that an impostor could cause This depends on what the user has access to—not just which functions but also other assets (for example, a customer who has a large sum of money in their account), so you might want to consider stronger authentication for users who have the greatest powers (or valuables) in the system. Also consider possible *indirect* damage, such as to the company's reputation if unauthorized access became public knowledge.

Factor 2: The type of device on which the user self-authenticates This could preclude certain kinds of authentication. For example, you can't expect the average Internet user to have a card reader attached to their PC (at least, not at the time of writing).

Factor 3: The local environment of the user How trusted or untrusted is the user's device by virtue of where it resides? Is it snug inside a company office, or in the wilds outside? Is the user likely to have someone looking over their shoulder when self-authenticating? (Note that when using an ATM, you *don't* enter both a user ID and PIN; your bank card–which a spy can't see–says *who* you are.)

Authentication can be regarded as creating a **user session**, which grants the user access until the session ends. This might sound like an implementation contrivance, but it's not–and it makes some aspects easier to discuss (especially a few of the extra requirements that follow). It appears a good idea to prevent a user from having more than one session at the same time, but it can be problematic, due to technical unreliability (especially with communications). A user is liable to get frustrated if they "return" after a failure, attempt to log in, and are refused because they already have a session (which they cannot use). On the other hand, letting a user have many active sessions can cause problems, too. If you allow a user to have more than one session simultaneously, a history of their actions only makes sense if you show each session separately. One option is for logging in to terminate all previous sessions–but there are circumstances where that's awkward, too (for example, if a user is active and someone who has purloined their password then logs in).

Content

A user authentication requirement should contain:

1. **Class of users** Which users does this requirement apply to? "All users" is an acceptable class.
2. **Authentication mechanism(s)** How do you expect the users to identify themselves? You could be specific (entry of a user ID and password, for example), or you could just describe the level of security needed and leave the details up to the development team. Avoid mention of specific technologies as far as possible.

   You could also keep the login function for one class of user separate from that for other classes of users. For instance, you might not want Web customers anywhere near the login function used by employees.
3. **Initiated by** When do users need to authenticate themselves? You could force users to log in to gain *any* access, or you could let them wander around anonymously and invite them to log in only if they want to do something that demands it. Both approaches have their place, and you might choose to treat different classes of users differently. For example, force employees to log in before they can do anything, but let Web site visitors log in when they want to. Another factor to bear in mind is that knowing who someone is as early as possible lets you record their actions, though you might irritate users if you insist on them logging in earlier than strictly necessary.

   Alternatively, you could leave the matter open–though for completeness, you could state that users must have authenticated themselves before doing anything that depends on their identities being known.

Template(s)

Summary	Definition
«*User class*» authentication/ login	A «*User class*» shall be able to self-authenticate (log in) by «*Authentication step(s)*». [«*Initiated by description*».]

Example(s)

Don't simply copy any of the requirements given here or in the "Extra Requirements" section. None of them suits all situations.

Summary	Definition
User authentication	A user shall be able to self-authenticate (log in), and must do so before they can access any function or information that is not publicly or anonymously accessible. The level of security offered by the mechanism used to authenticate a particular user (or class of users) shall be appropriate to the extent and sensitivity of the access they have (that is, the amount of damage that a malicious impostor could inflict). It is acceptable to use different login mechanisms for different classes of users. Customers and employees shall be kept apart to the extent that a customer shall not be able to log in as an employee simply by entering an employee's user ID and password into a customer login screen.
Customer authentication	A customer shall be able to self-authenticate (log in) by entering their user ID and password. They can choose to log in at any time (by visiting the login page)—but if they have not logged in when they attempt an action for which their identity must be known, they shall be prompted to log in and not allowed to proceed with the action until they have done so. The identity of each customer must be determined before they may initiate or view transactions. This shall be achieved by the customer entering their user ID and password.

Observe that the first example doesn't mention any particular authentication mechanism.

Extra Requirements

A user authentication requirement looks straightforward and self-contained, but there are various related matters you can specify in extra requirements, mostly to limit the misuse of user accounts. These include the following (each of which is covered in its own subsection that follows—except accessibility, which is covered in the accessibility requirement pattern in Chapter 8, "User Function Requirement Patterns").

1. **Forgotten password handling** People forget things. They forget their passwords, especially if you force them to be long or to use strange combinations of characters. What happens then?

2. **User de-authentication** (Logging out.) If you let someone in, you should let them out again–to tell you they've finished for now.
3. **Ending user sessions** We can't allow a user session to last forever, so we need at least one way to bring it to an end. First, limit the duration of each session. It's also useful to let an operator terminate a selected user session (if a user appears to be up to no good, for instance) or all–or nearly all–sessions prior to shutting the system down.
4. **Absent user protection** If a user's done nothing for a while, they might have stepped away from their machine, so we might want to try to stop someone else nipping in and doing things in that user's name.
5. **Helping users spot breaches** Users themselves represent an army of people with a strong interest in discovering any abuse of their own accounts. So why not help them do it? Help them detect whether someone else has logged in with their user ID, and give them a way of signaling security breaches.
6. **Blocking users** (So they can't log in.) There are several reasons for denying a user access to the system, including repeated entry of the wrong password, because they're going away (an employee taking a vacation, say), or simply because they've misbehaved. If users can be blocked, we need a way to unblock them. Finally, the ability to prevent *anybody* from logging in is handy–shortly before shutting down the system, if at no other time.
7. **Viewing user sessions** Looking at current user sessions can tell you about what's going on in the system *now*. Storing session information and being able to study it *later* can help you investigate a breach of security and is a good source of statistics.
8. **Accessibility** If you're using biometrics to check a person's identity, you should provide an alternative for people who are unable to offer the required body part to the device that reads it (because they don't possess it, lack mobility, or some other reason). This is a provision of Section 508 of the U.S. Rehabilitation Act. See the accessibility requirement pattern in Chapter 8 for more, including an example requirement (in the "General Accessibility" subsection of its "Extra Requirements" section). Beware of opening a security gap if you allow biometric authentication to be bypassed. Don't allow just anyone to say they're incapable of using a biometric reader; only allow nominated users to do so.

Forgotten Password Handling We must be able to get a user back on track when they forget their password, for which occasion, we need another way for them to prove their identity. This is straightforward enough in an environment where the user is known personally to someone who can vouch for their identity (such as colleagues in an office): a trusted user can run a function into which the hapless user can enter a new password.

In the absence of a trusted second person who can verify a user's identity (for instance, when a customer wishes to use our Web site), we're likely to be forced to resort to more dubious means. A common stratagem is to ask the user for additional "secret" information when they register–either a "secret phrase" or one or more questions along with their answers. (Registration can suggest questions: what was the name of you first pet? The make of your first car?) This typically results in values that are easier to guess than a password, and often not secret at all. In recognition of this, it's best not to depend solely on this "secret" information to identify the user. Two extra steps are as follows. First, to ask the user to speak on the telephone to an operator, who can bring to bear their personal judgment about whether the user is genuine (and ask extra questions). Second, test whether this person is able to receive communications sent to the user (which can

use any of the user's contact details–such as their home address or telephone number, but most often their email address, because it's quick and the sending of an email can be automated). Both these steps are weak from a security standpoint.

Whenever you're thinking about involving an operator in the forgotten password process, consider the potential operational cost, especially if the number of users is large. For systems that don't handle forgotten passwords automatically, this is one of the most numerous types of calls to help desks.

Summary	Definition
Customer forgets password	If a customer forgets their password, they shall be able to log in by entering the "secret phrase" they supplied when they registered plus a verification code sent to their email address. This shall be achieved by using the following process: 1. The customer indicates via the login screen that they have forgotten their password. 2. The system emails a one-off verification code to their registered email address (to prove this person can receive emails sent to it). 3. Upon receipt of the verification code, the customer can log in by entering it plus their secret phrase. They are forced to change their password before they can proceed.
Customer forgets secret phrase	If a customer forgets both their password and secret phrase, there shall be a means of last resort whereby they can prove their identity and have a new password assigned to them. This shall be achieved by using the following process (or one equivalent to it): 1. The customer is asked to phone a customer support number to speak to a helpdesk operator. 2. The helpdesk operator establishes the customer's identity by asking questions about their registration information and recent transactions. 3. *The operator unblocks the customer's account and tells the* customer a new (computer generated) one-off password. 4. The system emails a new verification code to the customer. 5. The customer uses the new password and verification code to log in and is forced to change their password and secret phrase (so the helpdesk operator doesn't know them).

User De-Authentication (Logging Out) Give users the opportunity to log out explicitly, so they can prevent someone else from slipping into their seat after they've gone and performing actions in their guise. Do this even in environments where users rarely log out. For example, visitors usually leave a Web site simply by moving on to another one, or by exiting their browser–but someone in an Internet café won't want a complete stranger to be able to carry on where they left off. Logging out has the effect of ending the user's session. If you allow a user to have

multiple sessions simultaneously, you must decide whether logging out applies to all of them or just to the one in which the logout was performed (both of which have their down sides)—or you could let the user choose.

Summary	Definition
User de-authentication	A user shall be able to de-authenticate (log out). After de-authentication, the system shall treat them just as if they had not been authenticated in the first place and shall regard all subsequent actions as being performed by an unknown user.

Ending User Sessions It's good practice to limit the duration of a user session, to force a user to prove it's really them from time to time. Without such a limit, a user could log in once and continue to use the system indefinitely. When a session is about to end, ask the user to establish their identity again, and don't permit them to continue to use the system until they do so. If possible, do this in a natural break in what the user is doing (at the end of a transaction, say): be polite. It also helps to let the user know that we're doing this for their own security. The duration of a session should be somewhat longer than users are typically continuously active, so we might set it at twelve hours for an employee, or three hours for a Web site customer. Logging out also ends a user session (as described in the previous subsection). If you can reliably discern that a user is leaving, you can end their session; this depends on the technology being used.

Summary	Definition
User session timeout	Every user session shall automatically end (time out) a length of time after it began. The length of time shall be configurable per class of user. When a timeout is imminent, the user shall be invited to reauthenticate (log in again), which shall allow them to continue using the system without interrupting any functions they happen to be using at the time. If they do not reauthenticate, the system shall reject any further requests from this session.
Exiting client application ends user session	When a user exits the client application, any user session opened by that application shall be terminated.

Notice that the first preceding example carefully sidesteps the question of whether it *extends* the existing user session or *creates a new one*. Developers might have strong views on the subject, but it's solely a matter for them: it doesn't concern the requirements, so steer clear of it here.

Next, it's useful for an operator to be able to peremptorily terminate a selected user session: without it, you'd be hamstrung in halting a user who's discovered performing nefarious activities. Taking this one small step further gives us the ability to terminate all active user sessions (or all sessions for a particular class of users—such as all customers), which can be useful if we want to shut the system down. Oh, and remember not to terminate the killer operator's own session!

Summary	Definition
Terminate user session(s)	An employee shall be able to terminate a selected active user session, all active sessions for a selected class of users, or all active user sessions. They shall not be able to terminate *their own* user session.

If we can give a user advance notice that their session will end shortly, it's polite to do so. For example, if their session is nearing its time-out time, we could notify them, so they can log out and log in again with minimal inconvenience.

Absent User Protection If an active user suddenly stops doing anything, it could be because they've walked off and left their machine all alone. A Web site customer who simply goes to another site (or shuts down their browser) without logging out can leave an open session. In these situations, we don't want someone else to be able to wander up (metaphorically in the case of an Internet attacker) and, as far as the system's concerned, impersonate the original user. If you want to prevent this, you can either write a requirement that expresses this intent, or write one for a specific way to prevent it—the commonest being an inactivity time-out, after which we ask the user to type in their password upon their return before they are allowed to proceed. We can then take this one stage further and let the user tell the system they're leaving their machine temporarily: that is, effectively to trigger the inactivity time-out immediately.

One awkward feature of locking a session can be (depending on the technology used) that it prevents anyone else from using the machine—which can be irritating. One way around this is to allow someone else to break the lock (though not to take over the original user's session, nor to see what they were doing). A further refinement is to let a user locking their session to say whether they're prepared to let it be broken.

Summary	Definition
User inactivity time-out	If a user with an active session has had no interaction with the system for longer than a configurable length of time, then the next time they make a request, they shall be asked to enter their password before they can proceed. The motivation for this requirement is to protect against someone attempting to take over a user's session when the user has walked away from their machine.
Lock user session	A user shall be able lock their current session, to indicate that they are about to leave their machine unattended. They must enter their password before they can proceed. When initiating the session lock, the user shall be able to indicate whether another user can break the lock. (Ordinarily, they should allow it to be broken, but might want to prevent it if they're in the middle of an important task.)
Break locked session	It shall be possible to break a user session that has been locked as a result of an inactivity time-out or an explicit lock. The person wishing to break the lock must enter their user ID and password, which terminates the original user's session and creates a session for the new user.

Helping Users Spot Breaches A system cannot tell the difference between a legitimate user and someone else who's discovered their user ID and password, so a system can never detect a security breach of this kind. But if we tell the user a little about their last session (typically, the time

they last logged in), they're in a position to judge whether someone else has been impersonating them in the meantime. Here's a requirement for the commonest way:

Summary	Definition
Show last login time at login	When a user has been successfully authenticated, the time at which they last logged in shall be displayed. The purpose of this requirement is to give them the opportunity to discover whether an imposter has been acting in their name since they last used the system.

If we help users detect misuse of their account, we should provide them with a way to tell us about it. You might think existing means of communication suffice, but it can be worth doing a little more. Here's one suggestion:

Summary	Definition
Notify operator of security breach	A user shall have a way to notify an operator of a security breach. If a standard notification method is used, it shall distinguish security breaches from other communications, to facilitate dealing with it as a matter of high priority. If email is the notification medium, a special email address shall be used for notifying security breaches, and users shall have a way of discovering this email address.

Think about what an operator should do when notified of a suspected breach of this kind. Have you specified requirements for tools that enable them to investigate (such as to view user sessions–discussed shortly)? Bear in mind that false alarms will probably greatly outnumber genuine misuses.

Blocking Users By "blocking" a user, we mean preventing them from logging in to the system. There several situations in which we might want to block a user, including:

A. The user enters their password incorrectly too many times in a row when logging in–because this might suggest someone else guessing and trying to impersonate them. It might be convenient for everyone for such a block to last for a short period of time only (an hour, say). We can afford to be relatively generous in the number of attempts to give a user to enter their password (25 or 50, say), because the chances of an attacker guessing the password in this way is still tiny (despite what Hollywood might have you believe). Systems that give a user only three attempts merely irritate the user and lead to more requests to whoever looks after unblocking user accounts (plus, for employees, loss of productivity until it's done).

B. A user might wish to block their own access for a certain length of time, if they know they're going to be away (such as before leaving on vacation).

C. An operator might wish to block a user who has been acting improperly or suspiciously.

If user accounts can become blocked, we need a way to unblock them. A useful related function is the ability to prevent all users (or a whole class of users–such as all customers) from logging in. This can be invoked prior to shutting down the system, to avoid frustrating users by kicking them out soon after they log in.

Summary	Definition
Block after successive invalid passwords	If a user enters their password incorrectly a given number of times in a row in one session, their account shall be blocked. The tolerable number of times shall be configurable per user class. (Suggested values are 50 for customers and 20 for employees—on the basis that impersonation of an employee is likely to have more serious consequences, and the far higher number of customers makes unblocking them a much bigger burden.)
Block own access	An employee shall be able to block their own access to the system for a specified length of time. The motivation for this requirement is to prevent improper access to an employee's account during known absences (such as vacations). If they return earlier than expected, they must ask an authorized other user to remove the block.
Block/unblock selected user	An authorized employee shall be able to block or unblock a selected user.
Disable/enable login	It shall be possible to switch off and on the ability of all users of a selected class to log in.

Viewing User Sessions User sessions contain valuable information–both about what's currently happening in the system and about which users accessed the system when. It's wasteful to simply discard or ignore this potential treasure. So store it, and let it be exploited. It might help identify a breach of security (after the fact), if nothing else.

Summary	Definition
Active sessions inquiry	It shall be possible to list all user sessions that are currently active and then to display all details about a selected session.
Own session inquiry	A user shall be able to view details about their current session and previous recent sessions. The motivation for this requirement is to let a user see whether someone else has been accessing their account—by spotting sessions they don't know about.

Summary	Definition
Store user sessions	All details about each user session shall be recorded in some form of persistent storage (such as a database). These details shall include: ■ User ID. ■ Start date and time. ■ Identity of machine or terminal from which the user is accessing the system. The identity to record is liable to depend on the connection mechanism (for example, the machine name if over a local network or the IP address if via the Internet). ■ Authentication method (for example, password, secret phrase, or smartcard). ■ End date and time. ■ Limit date and time: when the session would time out. ■ Cause of session end. Causes include logging out, timing out, and termination by operator. Session details shall be stored at the time the session starts and be updated whenever any item of information changes.
Old sessions inquiry	It shall be possible to list all user sessions that were active at a nominated point in time and to display all details about a selected session.
User session statistics report	There shall be a report that shows, for each day in a selected date range, the number of user sessions that began in each hour during that day and their average duration. Separate figures shall be shown for each class of users (customers and employees).

Considerations for Development

Requirements related to user authentication merely specify necessary features; they don't worry about the complexities, of which there are many—too many to relate here.

Coding a login function is almost as easy as a "Hello World" program. But doing it securely is considerably more difficult. Take into account all the good practices described in this requirement pattern, even if they're not formally required. Turn the password entered by the user into an undecipherable form as soon as possible, and overwrite the clear form in memory.

Considerations for Testing

The testing of user authentication needs to be performed at two very different levels:

- **Functional** Does the authentication process work properly in all circumstances normally encountered in the running of the system? Test it just as you would any user function: the entry and validation of values, expected responses, and so on. Test that a user who has not logged in has no non-public privileges. Test that after a user logs in, they can access whatever they are authorized to.

  Similarly, test all features discussed in the "Extra Requirements" section: the action and effect of logging out, session time-out, and so on.

- **Security** Not only must user authentication work, it must also do so in a secure r must prevent anyone else from learning sensitive information (especially user pas It must prevent the authentication process from being bypassed, subverted, or otl tricked. This is a specialized field. It involves replicating the wide variety of techni potential attackers could exploit, most of which are very sophisticated and highly technical. For example, someone listening to traffic between a user's machine and a central system can attempt a "replay" attack; this can only be tested by undertaking this kind of attack. This subject is beyond the scope of this requirement pattern; refer to specialized resources and consider bringing in an expert.

Ask for features that will assist testing. These include a way to view details about user sessions (especially when they end, so you can test that they time-out properly). To achieve this, we need the beginning and end of every user session to be recorded.

11.3 User Authorization Requirement Patterns

The purpose of access control is to restrict users to doing and seeing only what they should. This section talks about "what a user can do and see" as a concise way of referring respectively to the functions and the information they can access. Collectively they are referred to as **privileges** that they confer upon a user (or "access rights" or "permissions," though this book calls them privileges throughout). It's also worth recognizing that as far as a system is concerned, a user never does anything directly, so we mean what various parts of the system do on a user's behalf.

Access control is used most widely to control which **functions** someone can use. For example, a customer service operator could be granted access to all customer service functions, but little else. Functional access control can be extended to individual actions within a function (such as the ability to *change*, but not to *add*)—and while greater granularity gives more control, it takes more work both to develop and to maintain. Access control requirements can also restrict the **information** a user can see and can be used for other purposes (the commonest of which are spelled out in the specific authorization requirement pattern). These ways can be applied in combination. For example, a user might be granted access to one set of functions for one company and a different set of functions for another company. But this can get complicated: if you get away with authorizing access to functions and to information *independently*, life is much simpler.

There are two types of user authorization requirements: the first defines **specific privileges** (according to some access rule, such as for a named set of users); the second demands a **general ability to configure** who can do what. Each has its own requirement pattern. You can use both in a specification, in which case a configurable authorization requirement guides the capabilities of the system, and to guide its configuration, you can use specific authorization requirements (acting as a useful place to put knowledge gathered, to make setting up the system easier). A configurable authorization requirement is in effect an instruction to implement an access control infrastructure.

Regardless of how you choose to define who can do what, you should start by stating what access is allowed in absence of any guidance to the contrary. After all, a requirement saying that "A can do X" has little force if there's nothing to prevent everyone else from doing X. The most sensible approach to take is **denial by default**. That is, *everything* is to be inaccessible to every user, unless permission is granted to them either explicitly or by defining some things as publicly accessible.