Considerations for Development

If access control logic is simple and fixed, you can hard-code it. If it's not simple, or if it's liable to change, it's better to build a general, configurable mechanism–regardless of whether the requirements say you should. However, developing an underlying mechanism for access control is difficult and not for the faint-hearted. Apparently sensible and logical approaches can turn out to be a nightmare to manage in practice, because of the sheer quantity of configuration data. If you think a general access control mechanism will be worthwhile, bite the bullet early. Trying to retrofit it later is likely to be unpleasant.

Considerations for Testing

First of all, recognize that you're testing the requirements, not the privileges of whatever roles might have been set up: testing isn't responsible for checking that every user can do what they need to do their job. Having said that, it's always sensible to configure a test system as closely as possible to the live environment–which means defining roles and their privileges as realistically as possible.

When testing access to functions, for each function, test two things: first, that access to the function is controlled; and second, that the correct privilege is checked. This is exceedingly tedious to do manually. It helps if the system lets you observe each authorization check (down to the name of the privilege being checked); then you can see directly whether the correct privilege is used for each function. One way to observe authorization checks is to record them in a chronicle, and there is an example requirement for doing this in the "Chronicling Authorization Changes" subsection of the "Extra Requirements" section in this pattern.

A simpler test is to create a user role that can do everything, and verify that a user with that role can indeed do everything. Also create a user role that can do nothing, and verify that a user with that role can do nothing but publicly-accessible things.

One difficulty is that the number of privileges could keep growing. You'd have to be inhumanly diligent to insist on trying to test every new one that comes along.

11.6 Approval Requirement Pattern

Basic Details

Related patterns:	None
Anticipated frequency:	Often no requirements; rarely more than half a dozen requirements
Pattern classifications:	Functional: Yes; Affects database: Yes

Applicability

Use the approval requirement pattern to specify that a particular action (or set of actions) must be approved (or, in *some circumstances* approved) by a second person before it takes place.

Discussion

Any function in a system can be constructed so as to require approval before it takes effect. We usually think of one person approving an action requested by another, but it is also possible to allow a person to approve actions suggested by the system itself. ("An upgrade is available for the Whizzo Happiness Calculator. Would you like to download it now?" is an example.)

Imagine a bank in which a teller must have any large cash withdrawal approved by a supervisor. This sounds straightforward enough. But what constitutes *large*? Does it vary from teller to teller: are some more trusted than others? What is meant by *supervisor*? Can any supervisor approve it, or just the teller's direct superior? If a supervisor performs a withdrawal, must it be approved by another supervisor–and does that mean someone more senior still? Approving an action is invariably significantly trickier than it at first appears (or if you imagine that it will be tricky, you'll be proven right). It's also commonly neglected in both requirements specifications and software.

And when a *system* has to handle approval of actions, there are further complexities, because every system lacks the common sense possessed by every human being. How do we bring something needing approval to the attention of someone who can approve it? What if more than one person is able to approve it: do we bring it to the attention of *all* of them? If not, how do we decide which? What if a usual approver is absent? What if someone else wants to perform an action that affects the pending action (for example, if we're waiting for approval to debit a bank account, and another debit to the same account comes along); there are no easy answers to this one. Every little detail must be figured out and specified–and commonly they aren't, resulting in systems that don't behave sensibly and are awkward to use.

Content

The approval of an action is best specified as an extra requirement following that for the function that carries out the action. Approval requirements should answer some or all of the following questions, as appropriate:

1. **Which action(s) require approval?** That is, to what does this approval requirement apply? This is what happens when approval is granted. If it's not simply a matter of performing an action specified somewhere else, explain what else is involved.
2. **Under what circumstances is approval needed?** It could be *always*. If not, specify the conditions–such as up to some limit set for the user, or class of user (up to a specified monetary value, say).
3. **Who can approve, and under what circumstances?** Answering this question involves several steps:
   - ❑ Step 1: Identify all the users (or classes of users) who can approve this particular type of action. Also identify the circumstances under which they can approve–which can involve similar conditions to those in the previous question (on when approval is needed). For example, we might need some sort of limit up to which a user (or class of user) can *approve* a particular action (distinct from the limit up to which the user can *perform* the action without approval).
   - ❑ Step 2: Identify the prospective approver (or approvers) for each user, or a way of working out who they are. For example, only people in the same department might be eligible. And we could nominate a supervisor for each user and state that they're to be the approver of first resort. When a bank teller needs approval for a large withdrawal, confine approval to the same bank branch (or havoc will reign!).
   - ❑ Step 3: Consider what happens if a prospective approver is absent. If you identify only one person as approver, you're in trouble. You need some fallback position. If someone else is performing this job temporarily, how does the system know? Getting a system

to behave smartly based on the information already at its disposal (such as it knowing who is logged in at the moment) takes a lot of thought.

- ❑ Step 4: Prevent a user from approving *their own* actions. This is covered in the "Extra Requirements" section in this pattern.

4. **How promptly is approval needed?** An approval requirement doesn't itself need to answer to this question, although it has a major bearing on the best way of bringing something needing approval to a potential approver's attention (which is the next question). If approvals are only needed by the end of the day, a relatively simple implementation will suffice; if an action must be approved within two minutes, it's a very different matter.

5. **How is something awaiting approval to be brought to an approver's attention?** There's no universal answer to this question: every way has its drawbacks. Which is the most suitable depends on the answers to the previous two questions. Decide how prescriptive you wish to be: either choose a specific mechanism, or merely specify the characteristics you want. Possible approval mechanisms (along with a few strengths and weaknesses) include:

   a. **An approve subfunction within the data entry function itself** Where the user calls an approver over to their machine, who then self-authenticates and signifies approval (or rejection). This is the easiest mechanism to implement–and perhaps the most reliable, too, because it can take advantage of the user's human common sense. But it involves people walking around (how low tech!), and the risk of the approver's password being observed (plus the temptation for an approver to give their password to someone else to save all this trouble).

   b. **A stand-alone approval function** This deserves a requirement of its own, which is why it's described in the "Extra Requirements" section of this pattern.

   c. **An electronic communication medium** Email, SMS, or pager–or some other way of getting a message to a prospective approver. But that's just half the story: we still need a function by which they can grant approval. Use this sort of messaging mechanism only if the number of approval requests received by one person is relatively small.

   The last two mechanisms need us to decide which approver(s) are to be informed of something awaiting approval. Typically just a subset of the people authorized to approve are informed. If using a notification mechanism that expects the recipient to respond, send a message to only one person: it would be frustrating to respond only to find that someone else has already approved the action.

6. **What happens if an approver denies approval?** You must always ask this question. It's surprising how often specifications fail to say what is to happen in this case, and as a result, systems sometimes struggle to deal with rejection (a surprisingly human trait for a system!). Actually, there are three distinct situations here:

   a. An approver says, "**I reject this. It must not be approved.**" Usually rejection is the end of the matter, but you might want to consider providing a way for the original user to challenge it: to let them pass it to someone else who might approve it.

   b. An approver says, "**I can neither approve nor reject this.**" This represents a firm and final decision on their part. They would do this if, for example, they believe they're not in a position to decide. The item needs to be passed on to another prospective approver for their consideration. You might decide that in your system, this situation will never arise–that every approver will always know what to do–but if so, you should make this a conscious decision; otherwise a user might be forced into making a judgment they don't want to.

c. An approver says, "**I need more information before I can decide.**" In this case, consider whether the system needs to provide the prospective approver with the ability to explain what else they need. If so, specify that feature.

This assumes that just a single extra person needs to give their approval. It can be extended to cover two (or even more) people needing to approve. Sometimes this is best done as a "chain" of approvals (one person after another); sometimes they can each approve independently in any order. An example of the latter might be a system that allows the members of a committee to sign off on documents.

Template(s)

Summary	**Definition**
«*Action name*» approval	A «*Action description*» must «*Approval circumstances*» be approved by «*Approver description*». [«*Approval promptness statement*».] [«*Approval mechanism statement*».] [If approval is denied «*Rejection action description*».]

Example(s)

Summary	**Definition**
Large cash withdrawal approval	Any cash withdrawal larger than a teller's withdrawal limit must be approved by another employee within whose withdrawal limit it falls. Approval is granted by the withdrawal function on the teller's machine allowing the second employee to enter their user ID and password and then signify approval. The second employee shall also have the ability to reject the withdrawal, which results in it being canceled.
Employee vacation subject to approval	Every request for vacation of more than two days must be approved by the human resources department.

These examples demonstrate that while there's a large number of aspects to consider when specifying approval, the end result can be straightforward.

Extra Requirements

By now it should be clear that the business of approval is tricky and complicated. (That's why you still see bank tellers running around looking for supervisors to approve things, despite banks automating almost everything else in sight. Go and watch them before they end up in a museum.) Properly defining your approval workflow could take a variety of extra requirements, a couple of topics for which are:

1. Preventing a user from approving an action that they have initiated.
2. An approval function–either a special one for a particular kind of approval, or a general approval mechanism for managing *anything* that's waiting to be approved.

Each of these is discussed in its own section that follows.

Cannot Approve Own Action It is unacceptable to allow a user to approve an action that they have performed. Apply this rule to every kind of approval. There is nothing to be gained by not having it: simply let users perform the action without approval being necessary if that's what you want. But it's important to state this restriction explicitly in a requirement–or be prepared for it to be forgotten. Note that an approval step is sometimes used simply to involve a second person, who could be at the same level as the first user (not a supervisor)–hence the elegantly descriptive German term *Vieraugen*, four eyes. In such a case, both users might have the same permissions, and an explicit restriction of this sort is the only way to prevent one person from both entering and approving. Insisting that a user be unable to approve their own action is an example of an operational rule as per the specific authorization requirement pattern.

Summary	Definition
Cannot approve own action	A user shall not be able to approve an action that they initiated. For any action that requires more than one approval, a user shall not be able to approve an action they have already approved. This requirement applies to every type of action that is subject to approval. It does not, however, preclude one user from being granted the ability to perform a function without approval (even if another user must obtain approval).

Approval Mechanism An approval mechanism lets you see and pick items awaiting approval and go ahead and act on them. The simplest case looks after just a single kind of item (or a small number of fixed kinds).

If your system has more than one or two functions for which approval is needed, it could be worthwhile building or buying (and hence specifying) a *general* approval mechanism that can manage everything that's awaiting approval, to save duplicating the messy parts. This might include, for example, a function to display all items pending approval by the current user, regardless of what sorts they are. It could incorporate some kind of *workflow* support, and it could involve an infrastructure deserving of its own requirements specification. You would need to specify requirements for the pieces that make up a general mechanism of this sort. Such requirements could be wide-ranging and complex.

If prompt approval is needed, approvers must have an approval mechanism running all the time (or utilize some kind of notification mechanism). But what if they don't have it running? To work well for prompt response, this function must refresh itself (which is harder with some technologies than others).

Here's an example of a specific approval function:

Summary	Definition
Employee vacation request approval	There shall be a function (intended to be used by the human resources department) that lists all pending vacation requests by employees in departments that are the responsibility of the current user. This function shall allow the user to select and view the details of any vacation request (as well as the employee's accrued vacation) and to approve or reject the request.

Considerations for Development

If more than a couple of types of action need approval (or might in the future), consider building a generalized mechanism for storing any kind of unapproved action. This must be able to store anything that's yet to be approved, for which it needs to be flexible. Depending how sophisticated you want this mechanism to be, it could also provide a means for other functions to find out whether anything of interest to them is awaiting approval. For example, if a large bank account debit is awaiting approval, another function that wishes to know the balance of that account probably needs to take the pending debit into account. Also take care to protect this pending-action store against tampering.

Note that if we build a place to store pending (yet-to-be-approved) actions, this can also be used as the basis for handling timed and co-ordinated changes, as described in the introduction to the "Data Entity Requirement Patterns" chapter. So if you build a mechanism for one, you have a head start towards the other.

Considerations for Testing

Concentrate on what sorts of things could happen between the action being initiated and its being approved. Has someone already figured out all the extra actions that users could take that could interfere with the action awaiting approval? A workflow (or equivalent) diagram is a particularly helpful form. If not, produce one—and be suspicious that the system might be incomplete. Identify all the possible states an approval request can be in, and the ways it can change state. Figure out all the actions that any of the people involved could take at any stage. Then test these possibilities and verify that the system handles them satisfactorily.