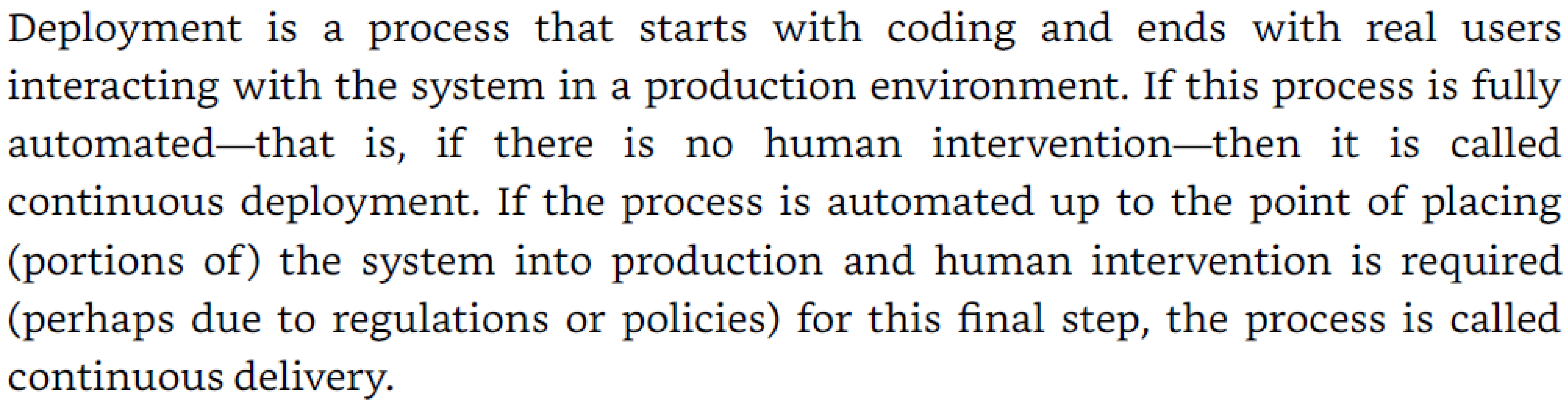
Software Systems Architecture

Quality Attributes

Poștaru Andrei

Deployability



Deployment is a process that starts with coding and ends with real users interacting with the system in a production environment. If this process is fully automated—that is, if there is no human intervention—then it is called continuous deployment. If the process is automated up to the point of placing (portions of) the system into production and human intervention is required (perhaps due to regulations or policies) for this final step, the process is called continuous delivery.

Continuous Deployment

To speed up releases, we need to introduce the concept of a *deployment pipeline*: the sequence of tools and activities that begin when you check your code into a version control system and end when your application has been deployed for users to send it requests. In between those points, a series of tools integrate and automatically test the newly committed code, test the integrated code for functionality, and test the application for concerns such as performance under load, security, and license compliance.

Continuous Deployment

Each stage in the deployment pipeline takes place in an environment established to support isolation of the stage and perform the actions appropriate to that stage. The major environments are as follows:

- Code is developed in a *development environment* for a single module where it is subject to standalone unit tests. Once it passes the tests, and after appropriate review, the code is committed to a version control system that triggers the build activities in the integration environment.
- An *integration environment* builds an executable version of your service. A continuous integration server compiles[1] your new or changed code, along with the latest compatible versions of code for other portions of your service and constructs an executable image for your service.[2] Tests in the integration environment include the unit tests from the various modules (now run against the built system) as well as integration tests designed specifically for the whole system. When the various tests are passed, the built service is promoted to the staging environment.

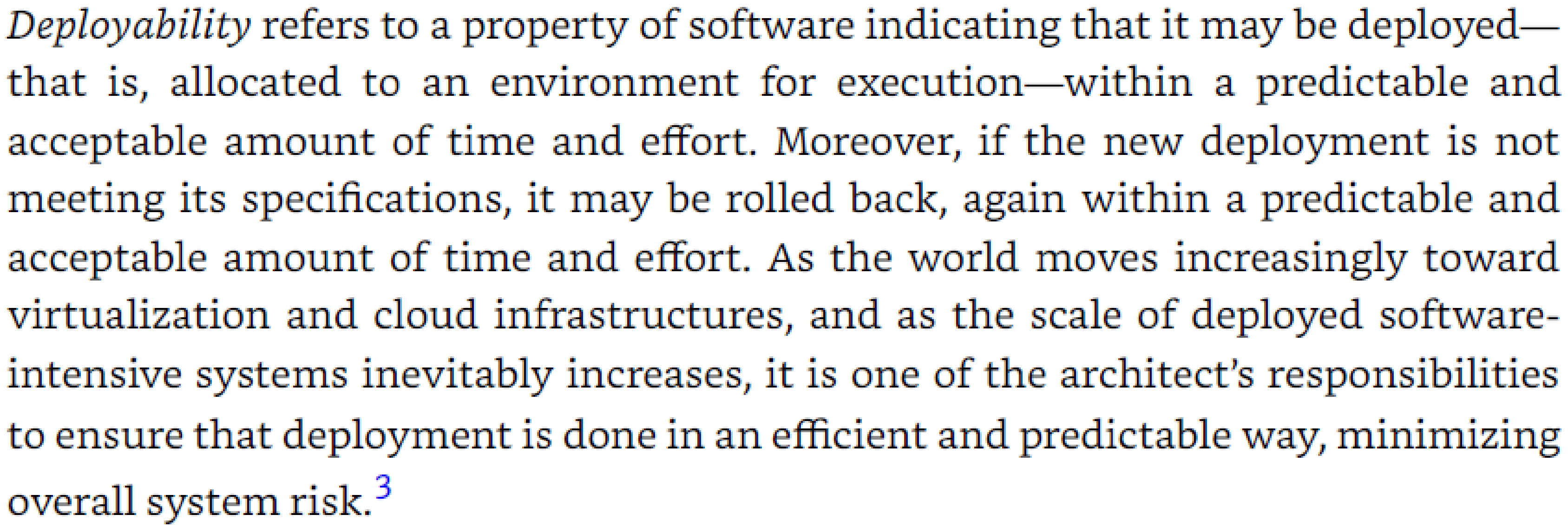
Continuous Deployment

- A *staging environment* tests for various qualities of the total system. These include performance testing, security testing, license conformance checks, and, possibly, user testing. For embedded systems, this is where simulators of the physical environment (feeding synthetic inputs to the system) are brought to bear. An application that passes all staging environment tests—which may include field testing—is deployed to the production environment, using either a blue/green model or a rolling upgrade .
- Once in the *production environment*, the service is monitored closely until all parties have some level of confidence in its quality. At that point, it is considered a normal part of the system and receives the same amount of attention as the other parts of the system.

Continuous Deployment

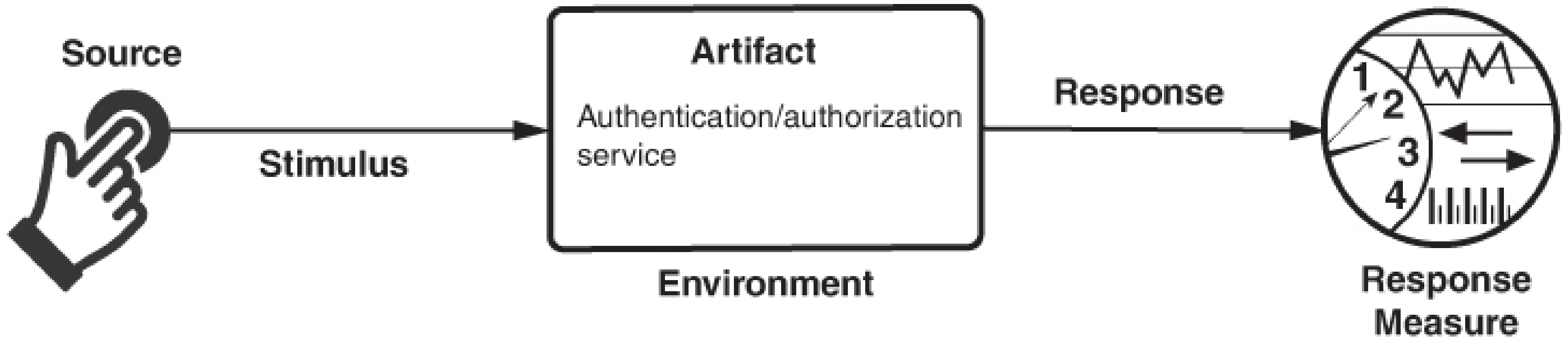
- A *staging environment* tests for various qualities of the total system. These include performance testing, security testing, license conformance checks, and, possibly, user testing. For embedded systems, this is where simulators of the physical environment (feeding synthetic inputs to the system) are brought to bear. An application that passes all staging environment tests—which may include field testing—is deployed to the production environment, using either a blue/green model or a rolling upgrade .
- Once in the *production environment*, the service is monitored closely until all parties have some level of confidence in its quality. At that point, it is considered a normal part of the system and receives the same amount of attention as the other parts of the system.

Deployability



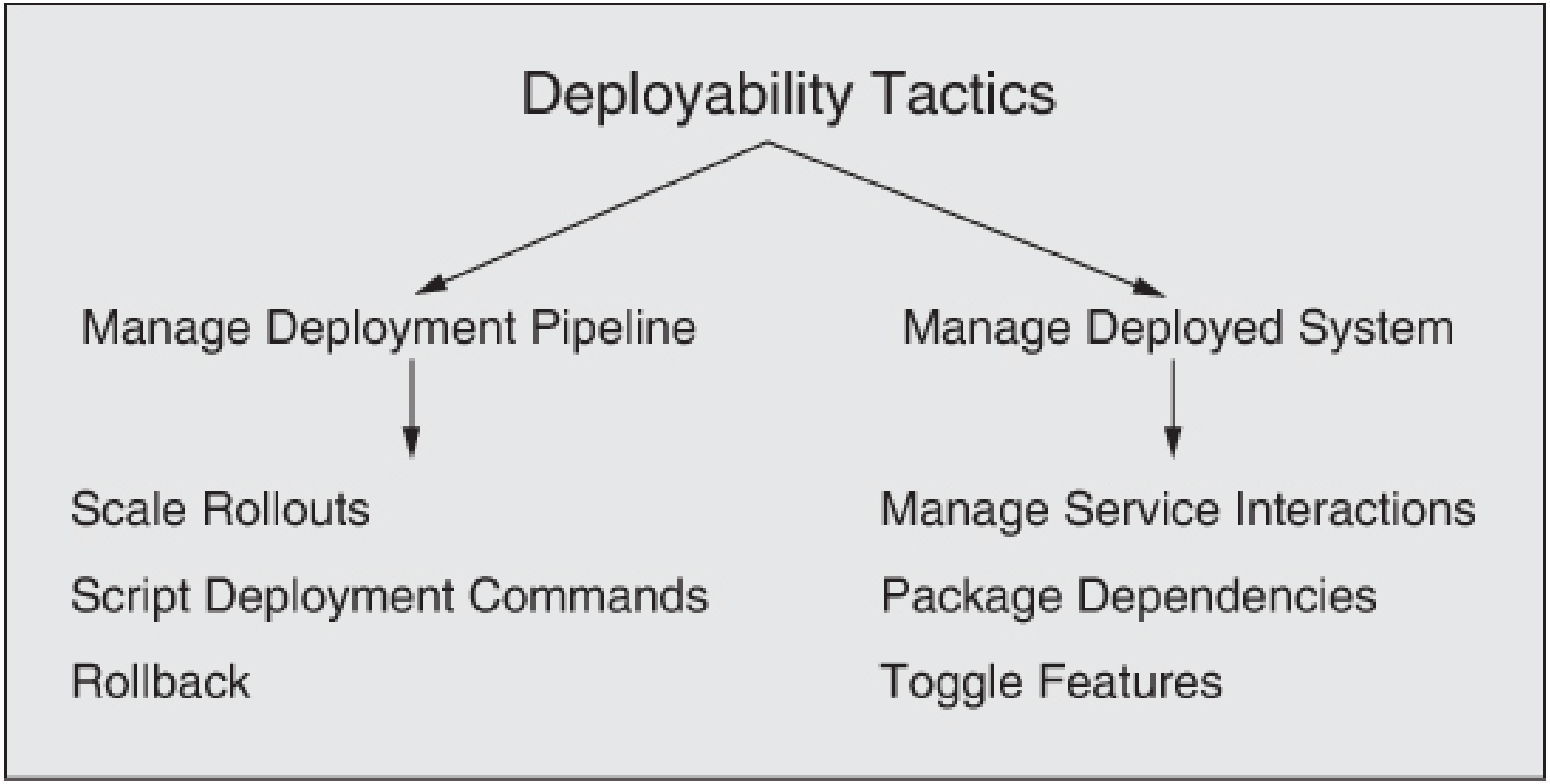
Deployability refers to a property of software indicating that it may be deployed—that is, allocated to an environment for execution—within a predictable and acceptable amount of time and effort. Moreover, if the new deployment is not meeting its specifications, it may be rolled back, again within a predictable and acceptable amount of time and effort. As the world moves increasingly toward virtualization and cloud infrastructures, and as the scale of deployed software-intensive systems inevitably increases, it is one of the architect's responsibilities to ensure that deployment is done in an efficient and predictable way, minimizing overall system risk.[3]

Deployability

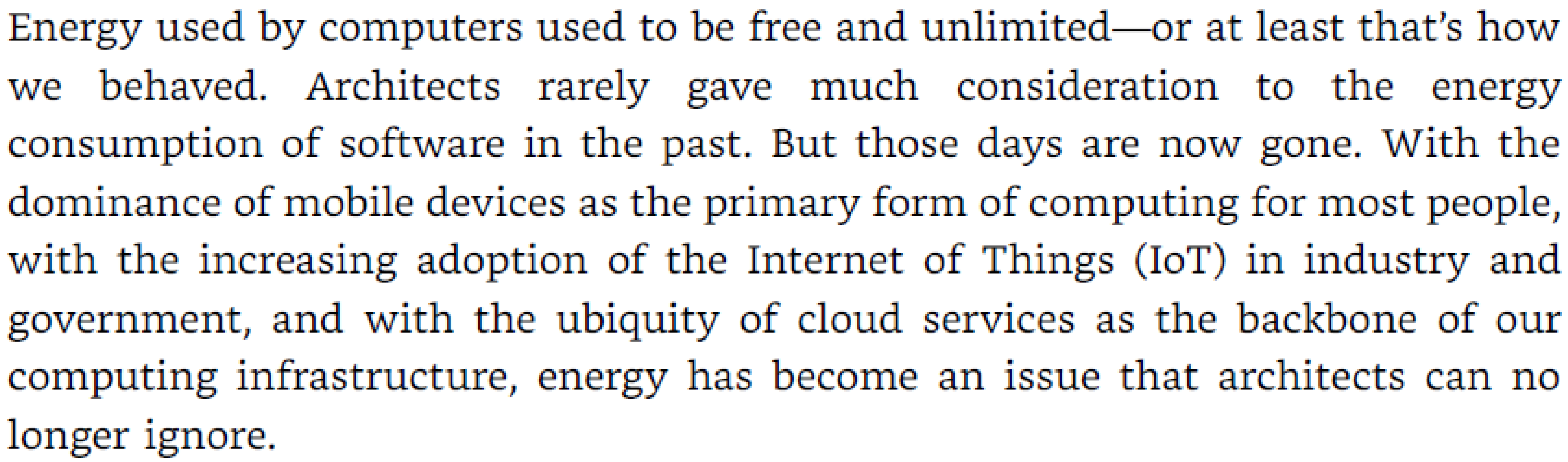


Source	Stimulus	Environment	Response	Response Measure
Component marketplace	New release of the authentication/authorization service is made available and the product owner decides to incorporate it	Production	The new service is tested in-house and deployed to production servers	Within 40 hours and no more than 120 person-hours of effort No defects introduced; no SLA violated

Deployability tactics

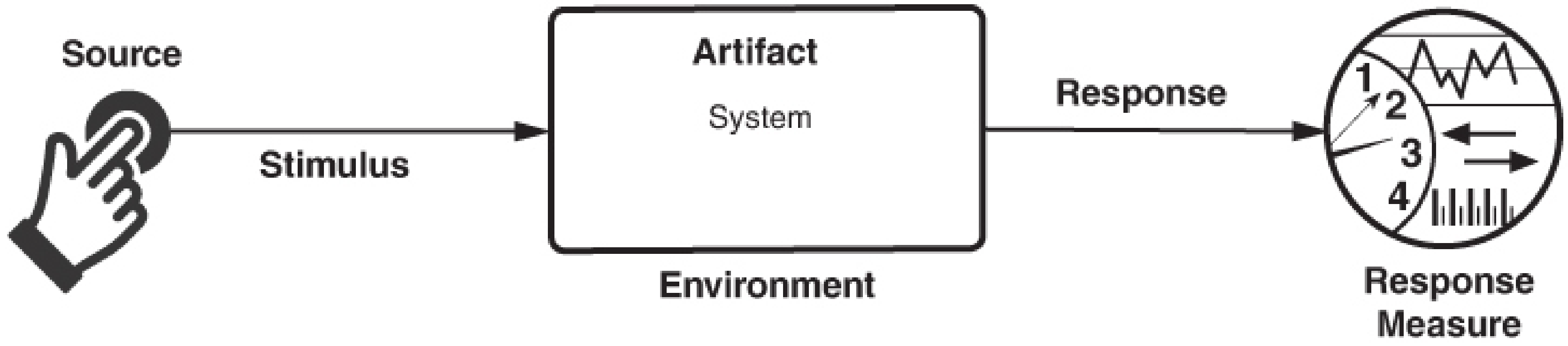


Energy Efficiency



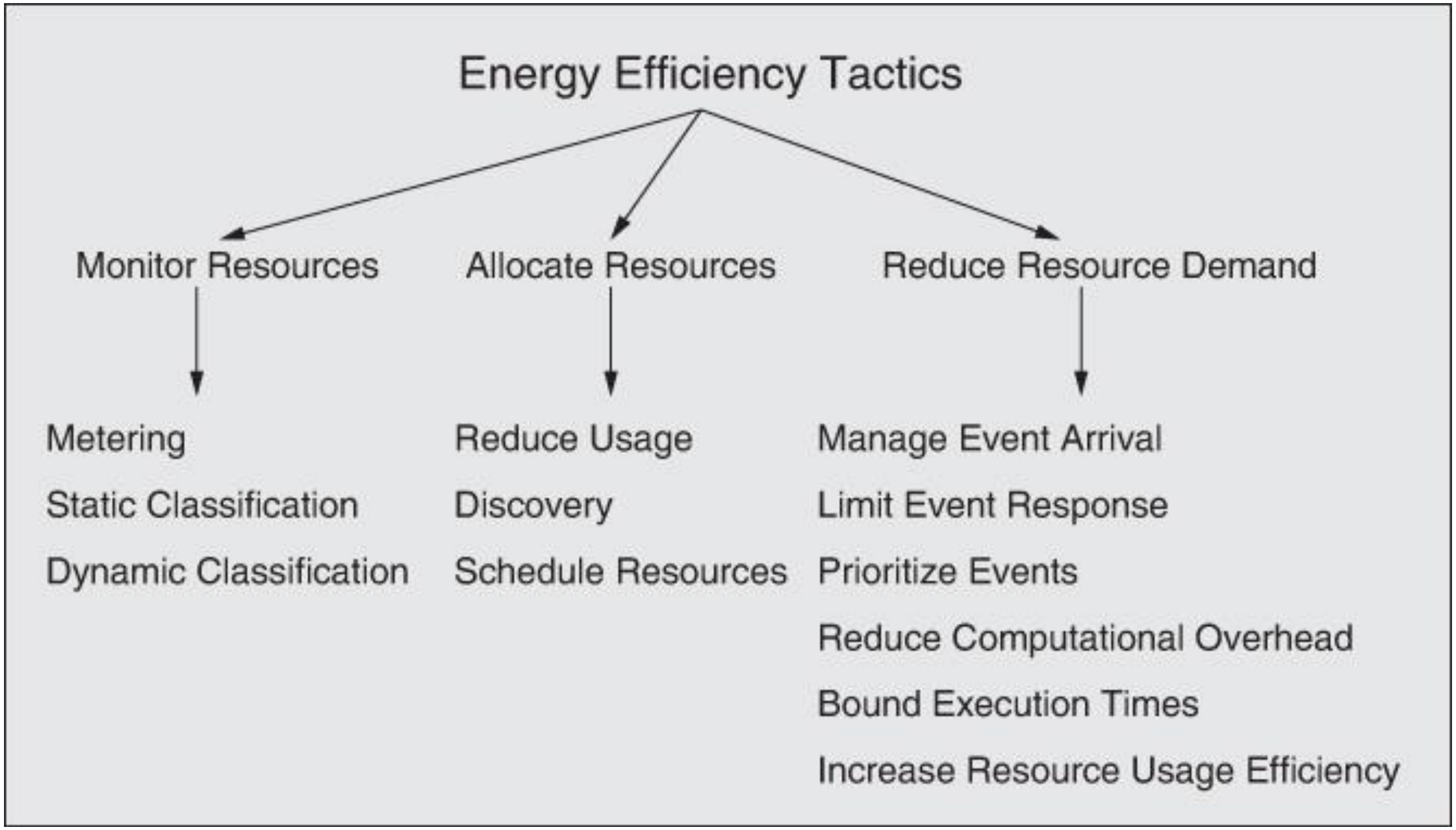
Energy used by computers used to be free and unlimited—or at least that's how we behaved. Architects rarely gave much consideration to the energy consumption of software in the past. But those days are now gone. With the dominance of mobile devices as the primary form of computing for most people, with the increasing adoption of the Internet of Things (IoT) in industry and government, and with the ubiquity of cloud services as the backbone of our computing infrastructure, energy has become an issue that architects can no longer ignore.

Sample energy efficiency scenario

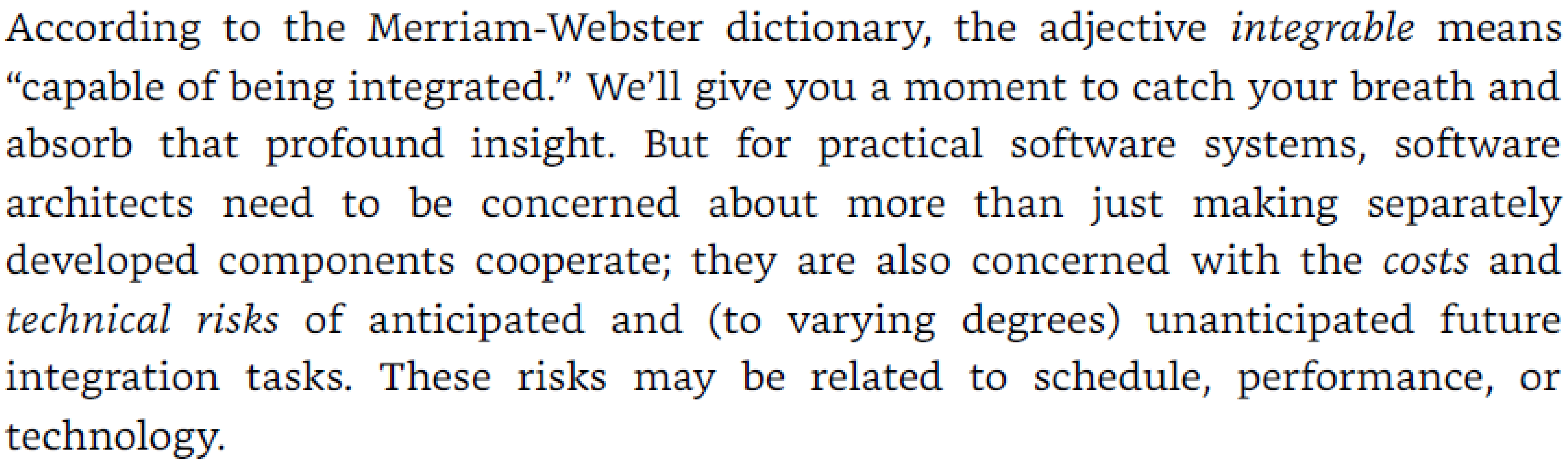


Manager	Wants to save energy at runtime by deallocating unused resources at non-peak periods	Non-peak period	The system deallocates resources	Maintaining worst-case latency of 2 seconds on database queries, saving on average 50% of the total energy required

Energy efficiency tactics

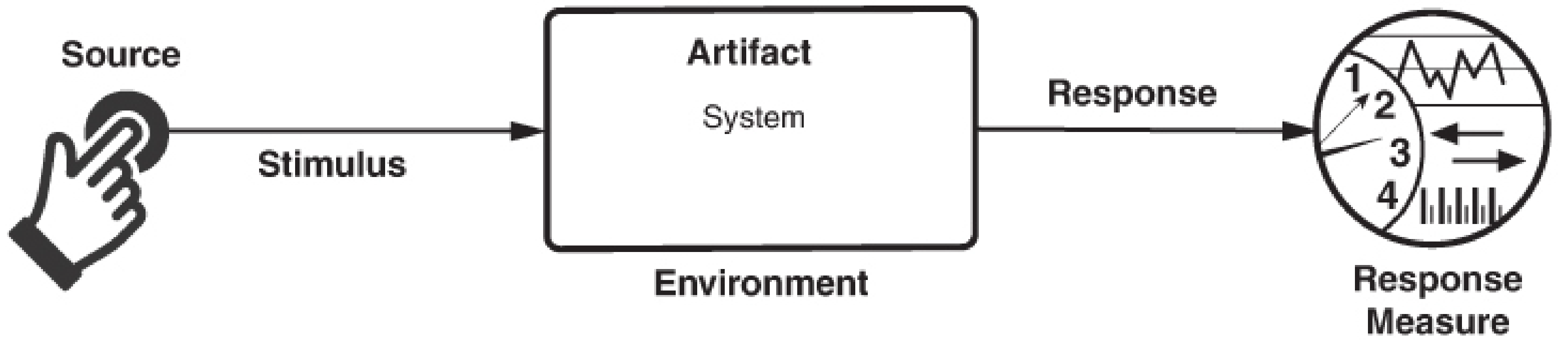


Integrability



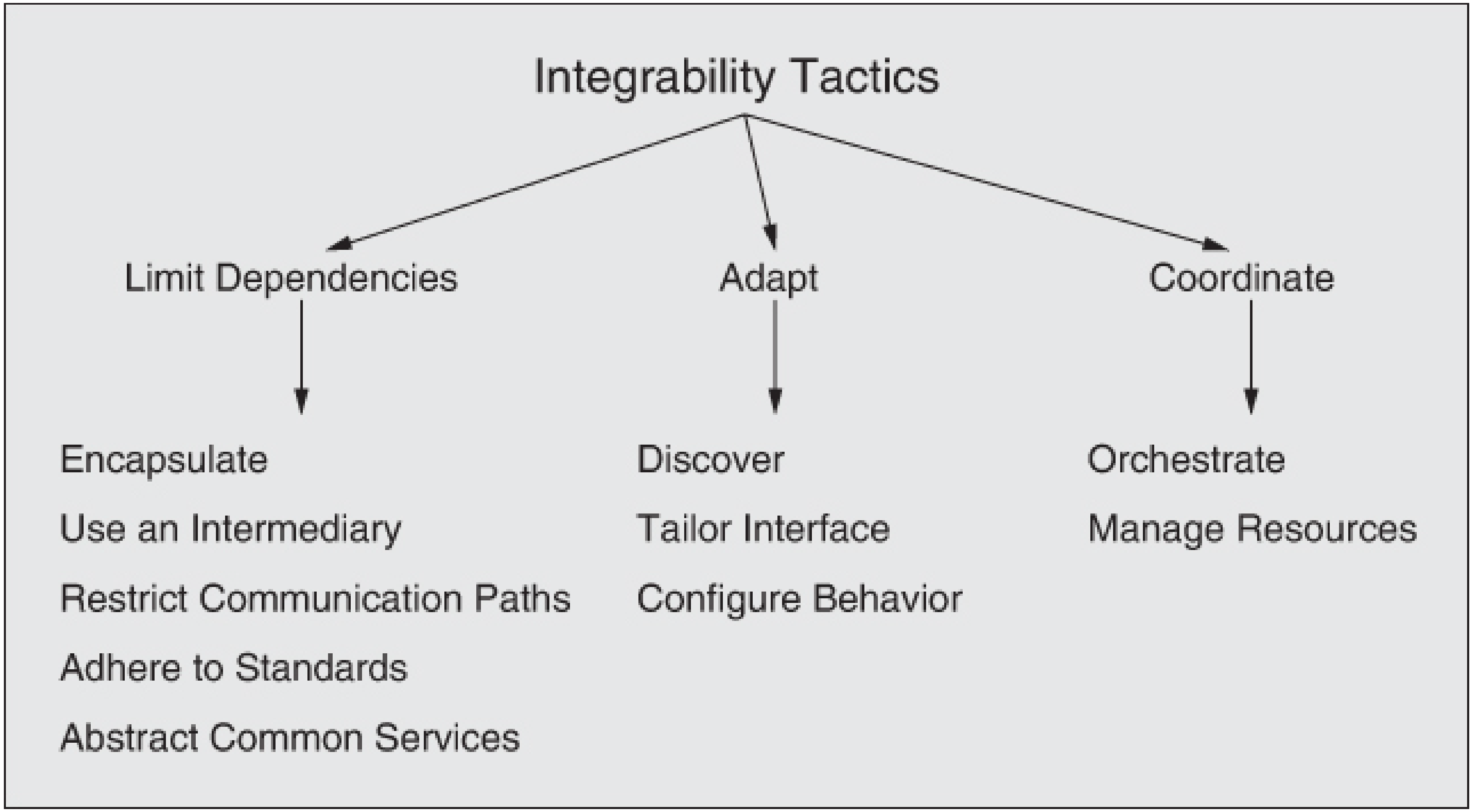
According to the Merriam-Webster dictionary, the adjective *integrable* means "capable of being integrated." We'll give you a moment to catch your breath and absorb that profound insight. But for practical software systems, software architects need to be concerned about more than just making separately developed components cooperate; they are also concerned with the *costs* and *technical risks* of anticipated and (to varying degrees) unanticipated future integration tasks. These risks may be related to schedule, performance, or technology.

Sample integrability scenario



Component marketplace	New data filtering component becomes available	Development	The new component is integrated and deployed	1 month, with no more than 1 person-month of effort

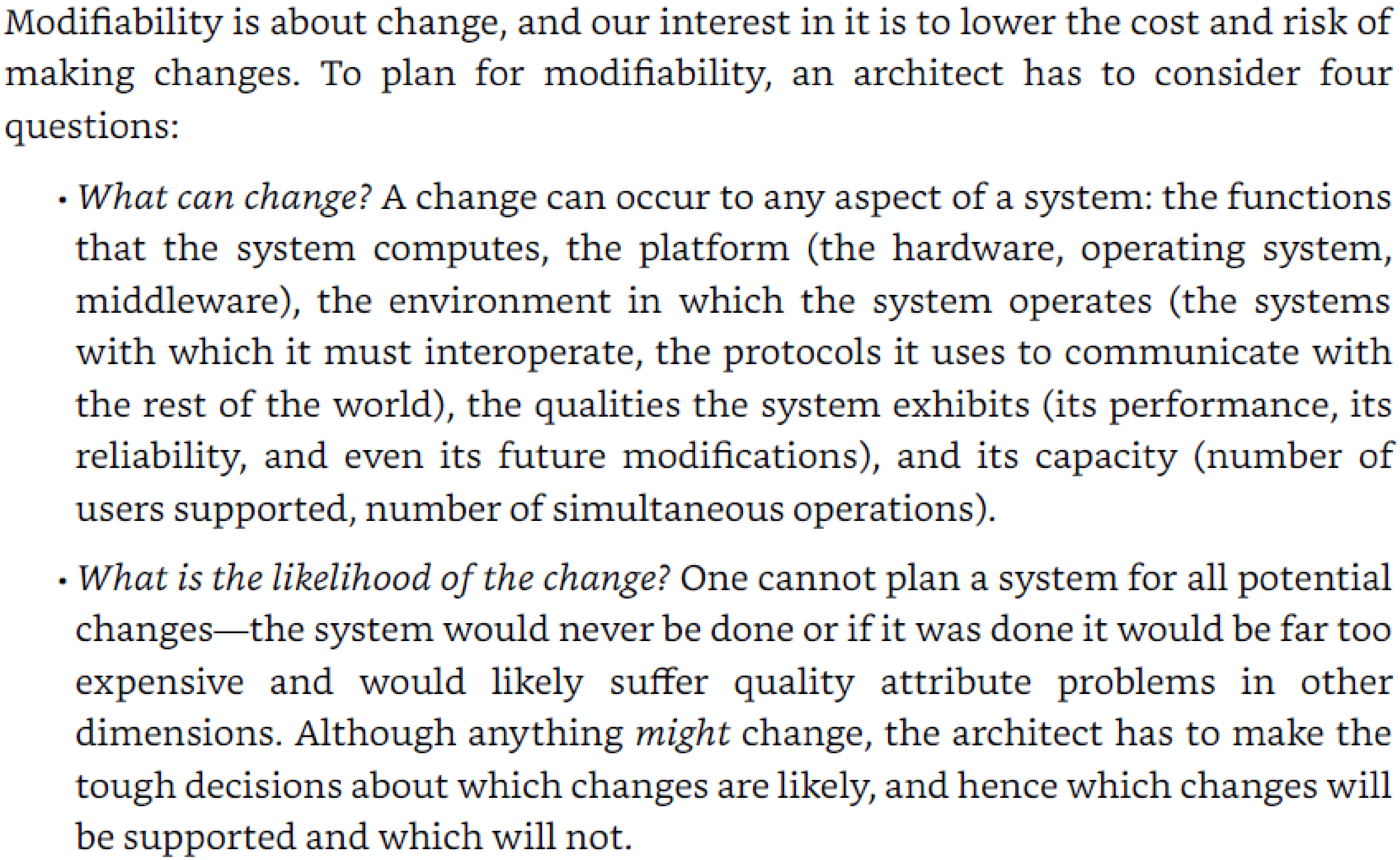
Integrability tactics



Modifiability

Study after study shows that most of the cost of the typical software system occurs *after* it has been initially released. If change is the only constant in the universe, then software change is not only constant but ubiquitous. Changes happen to add new features, to alter or even retire old ones. Changes happen to fix defects, tighten security, or improve performance. Changes happen to enhance the user's experience. Changes happen to embrace new technology, new platforms, new protocols, new standards. Changes happen to make systems work together, even if they were never designed to do so.

Modifiability



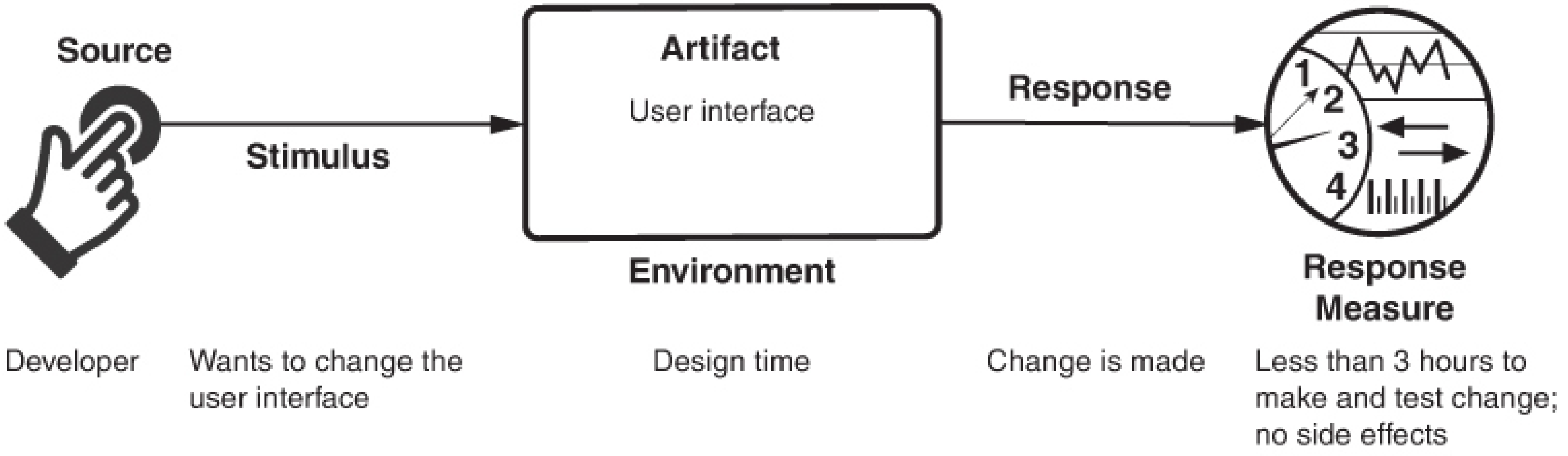
Modifiability is about change, and our interest in it is to lower the cost and risk of making changes. To plan for modifiability, an architect has to consider four questions:

- *What can change?* A change can occur to any aspect of a system: the functions that the system computes, the platform (the hardware, operating system, middleware), the environment in which the system operates (the systems with which it must interoperate, the protocols it uses to communicate with the rest of the world), the qualities the system exhibits (its performance, its reliability, and even its future modifications), and its capacity (number of users supported, number of simultaneous operations).
- *What is the likelihood of the change?* One cannot plan a system for all potential changes—the system would never be done or if it was done it would be far too expensive and would likely suffer quality attribute problems in other dimensions. Although anything *might* change, the architect has to make the tough decisions about which changes are likely, and hence which changes will be supported and which will not.

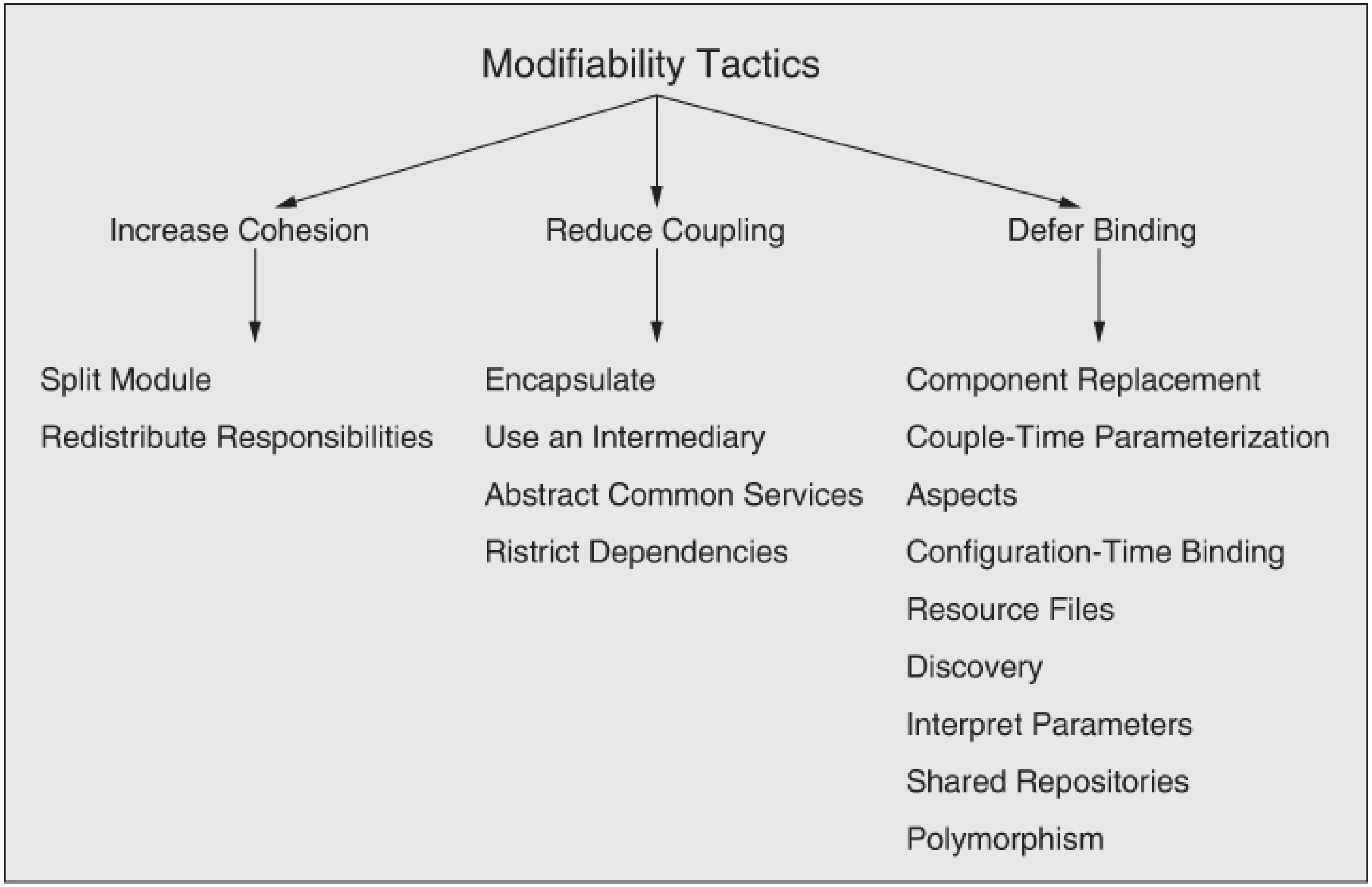
Modifiability

- *When is the change made and who makes it?* Most commonly in the past, a change was made to source code. That is, a developer had to make the change, which was tested and then deployed in a new release. Now, however, the question of when a change is made is intertwined with the question of who makes it. An end user changing the screen saver is clearly making a change to one aspect of the system. Equally clear, it is not in the same category as changing the system so that it uses a different database management system. Changes can be made to the implementation (by modifying the source code), during compilation (using compile-time switches), during the build (by choice of libraries), during configuration setup (by a range of techniques, including parameter setting), or during execution (by parameter settings, plug-ins, allocation to hardware, and so forth). A change can also be made by a developer, an end user, or a system administrator. Systems that learn and adapt supply a whole different answer to the question of when a change is made and "who" makes it—it is the system itself that is the agent for change.
- *What is the cost of the change?* Making a system more modifiable involves two types of costs:
  - The cost of introducing the mechanism(s) to make the system more modifiable
  - The cost of making the modification using the mechanism(s)

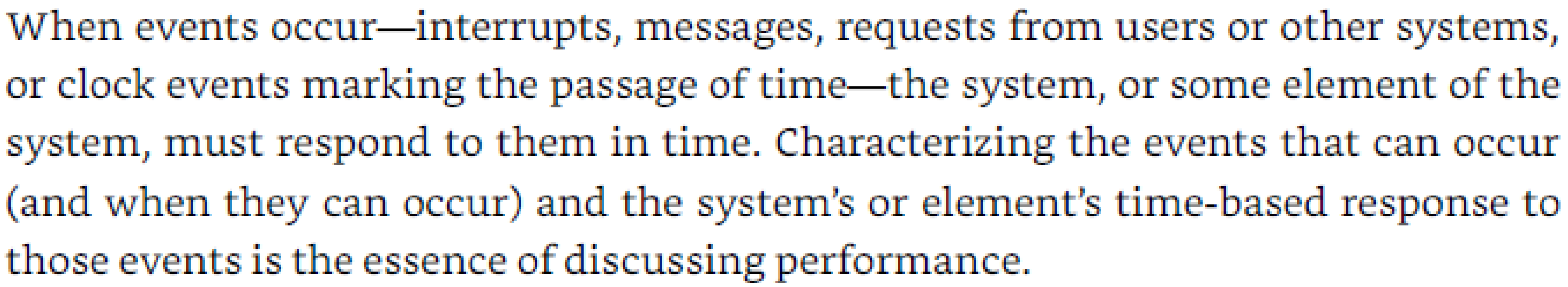
Sample concrete modifiability scenario



Modifiability tactics

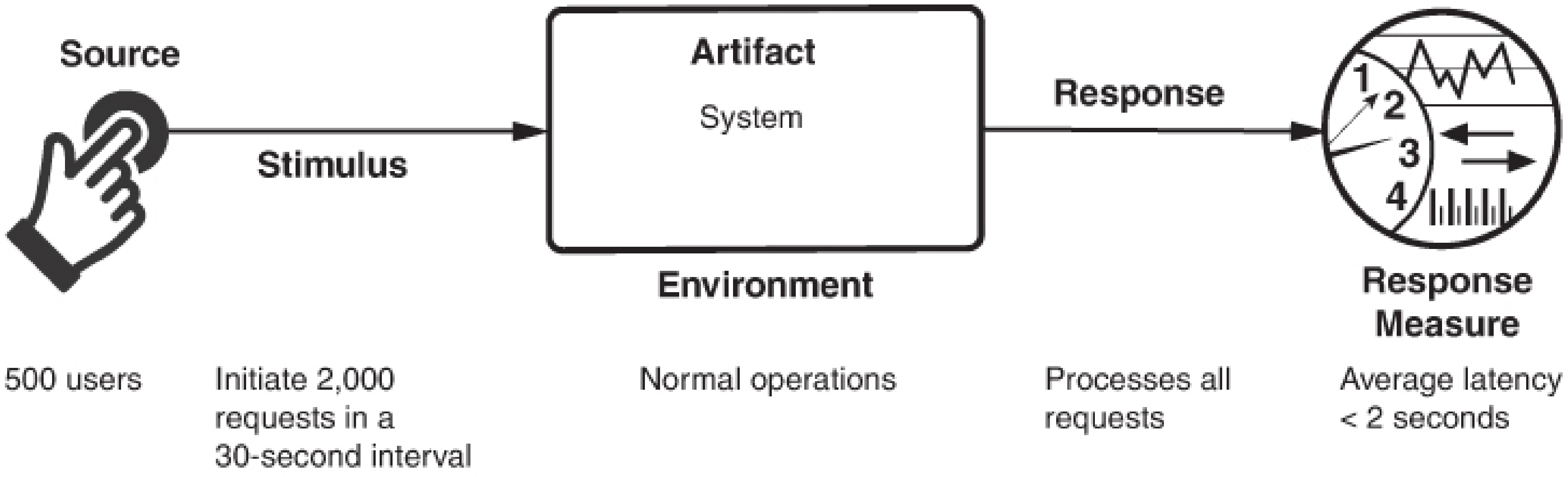


Performance

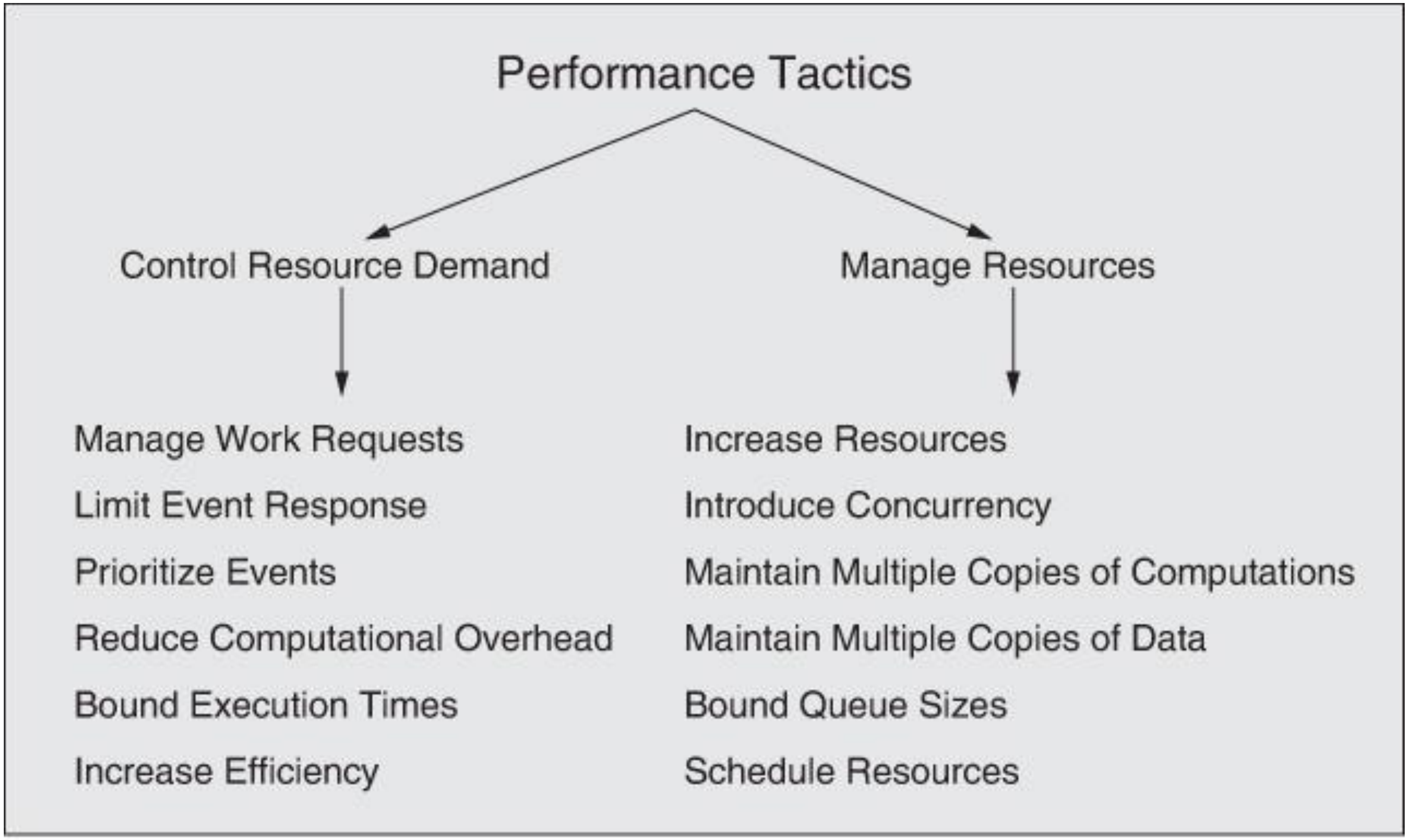


When events occur—interrupts, messages, requests from users or other systems, or clock events marking the passage of time—the system, or some element of the system, must respond to them in time. Characterizing the events that can occur (and when they can occur) and the system's or element's time-based response to those events is the essence of discussing performance.

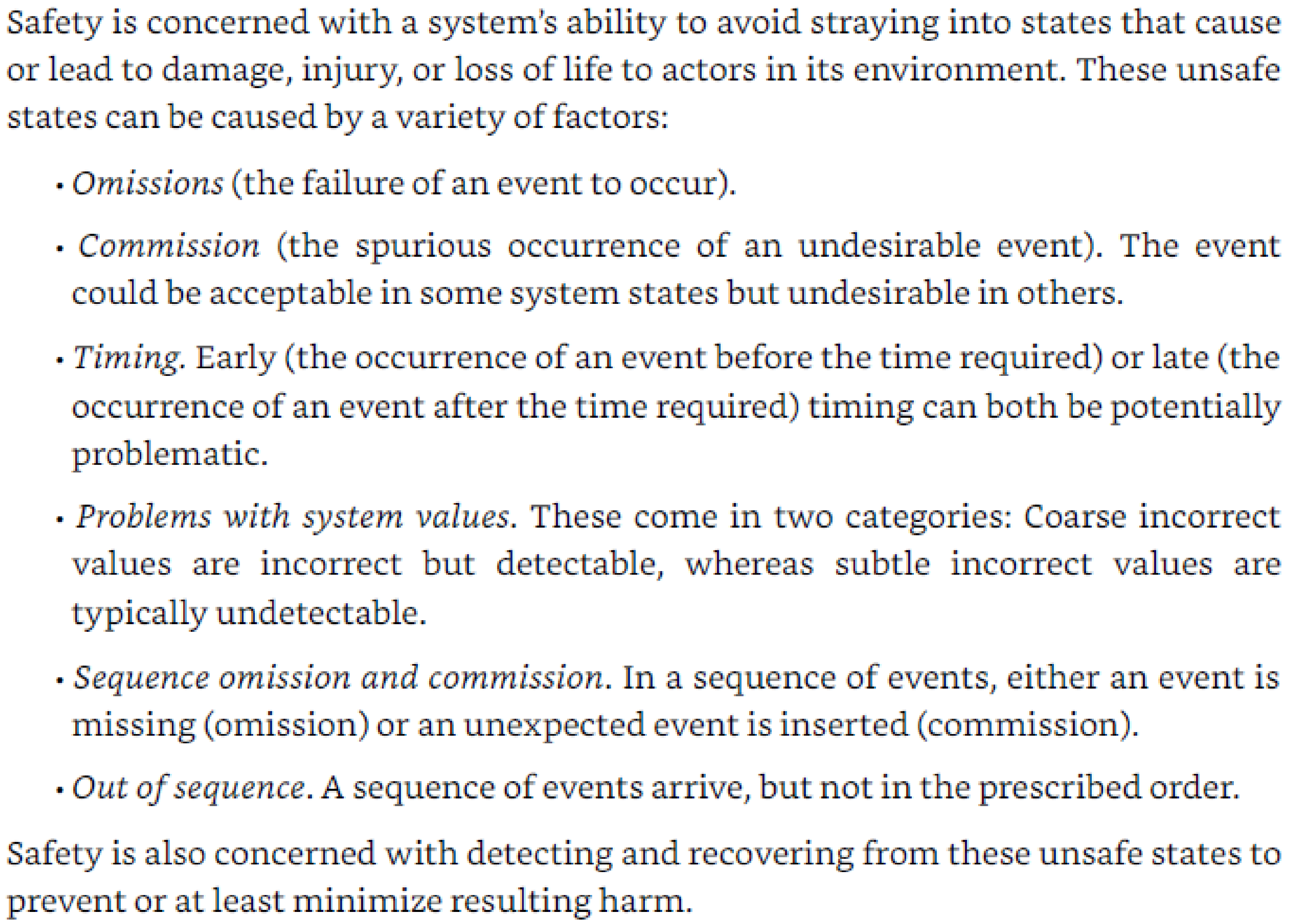
Sample performance scenario



Performance tactics



Safety

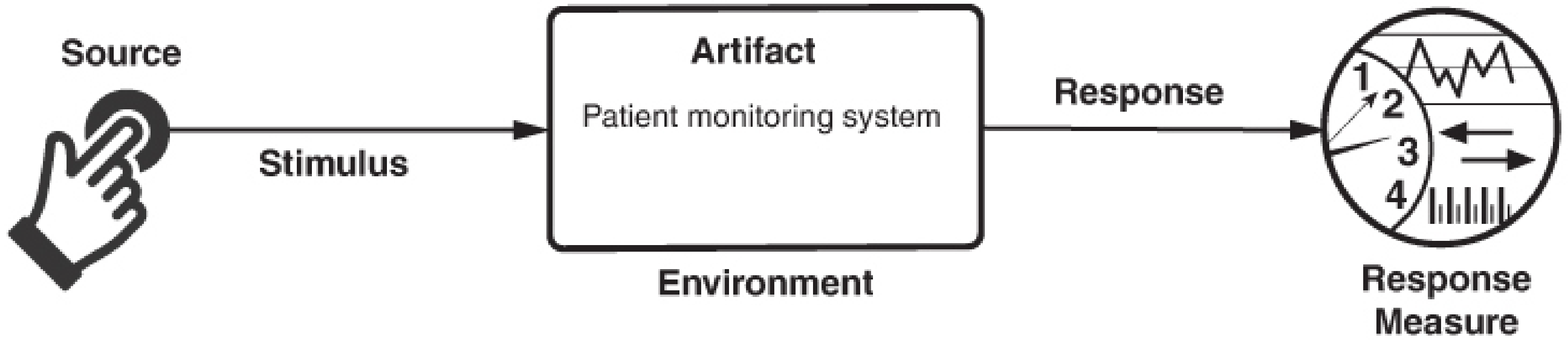


Safety is concerned with a system's ability to avoid straying into states that cause or lead to damage, injury, or loss of life to actors in its environment. These unsafe states can be caused by a variety of factors:

- *Omissions* (the failure of an event to occur).
- *Commission* (the spurious occurrence of an undesirable event). The event could be acceptable in some system states but undesirable in others.
- *Timing*. Early (the occurrence of an event before the time required) or late (the occurrence of an event after the time required) timing can both be potentially problematic.
- *Problems with system values*. These come in two categories: Coarse incorrect values are incorrect but detectable, whereas subtle incorrect values are typically undetectable.
- *Sequence omission and commission*. In a sequence of events, either an event is missing (omission) or an unexpected event is inserted (commission).
- *Out of sequence*. A sequence of events arrive, but not in the prescribed order.

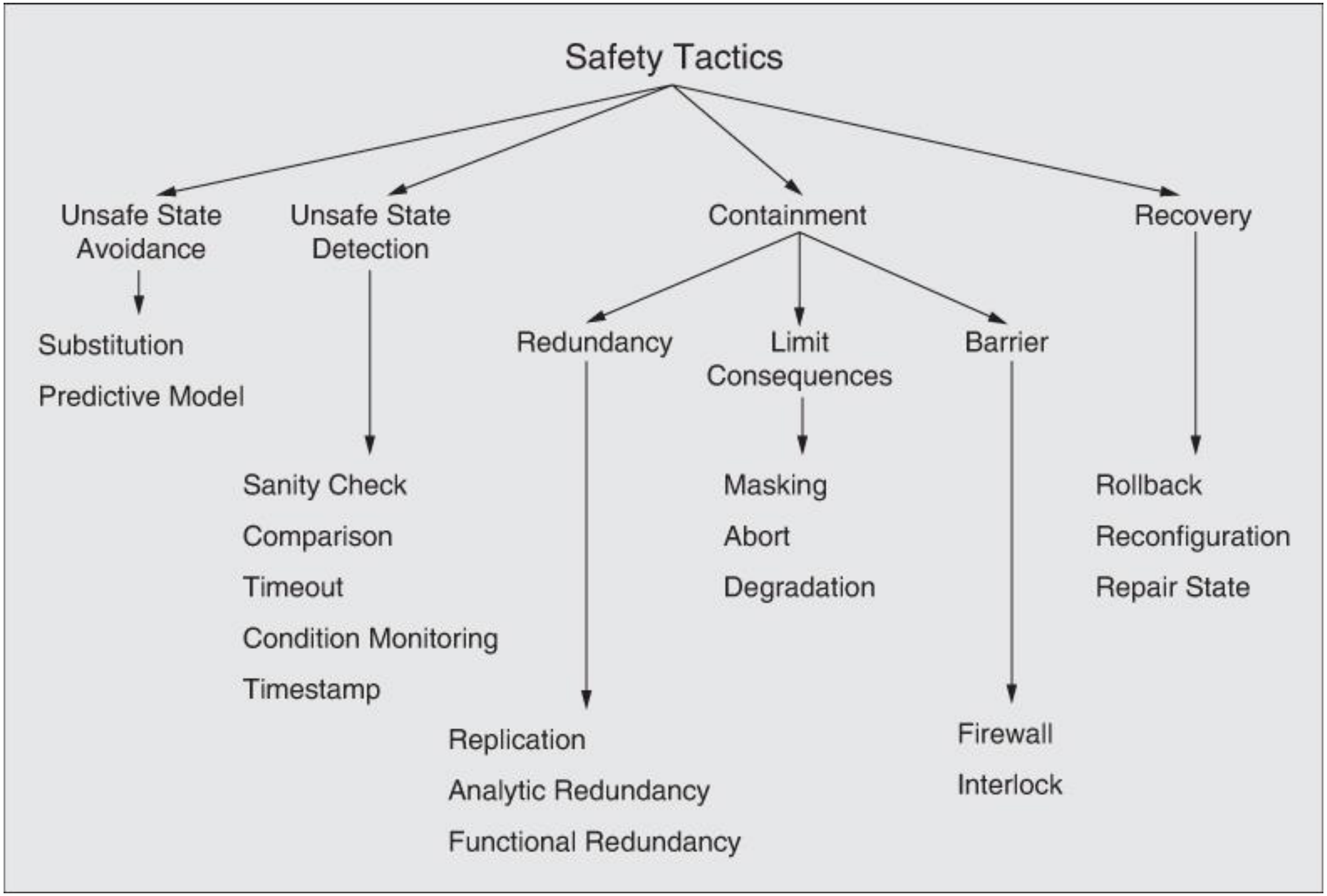
Safety is also concerned with detecting and recovering from these unsafe states to prevent or at least minimize resulting harm.

Sample concrete safety scenario

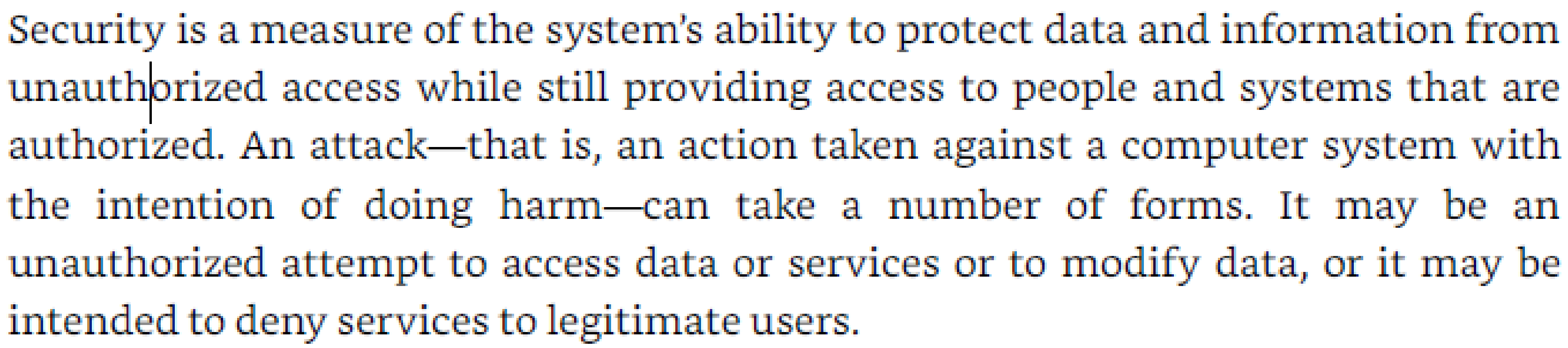


Source	Stimulus	Environment	Response	Response Measure
A sensor	Fails to report a life-critical value after 100 ms	Normal operations	The failure is logged, a warning light is illuminated on the console, and a backup (lower fidelity) sensor is engaged; the system monitors the patient using the backup sensor	After no more than 300 ms

Safety tactics

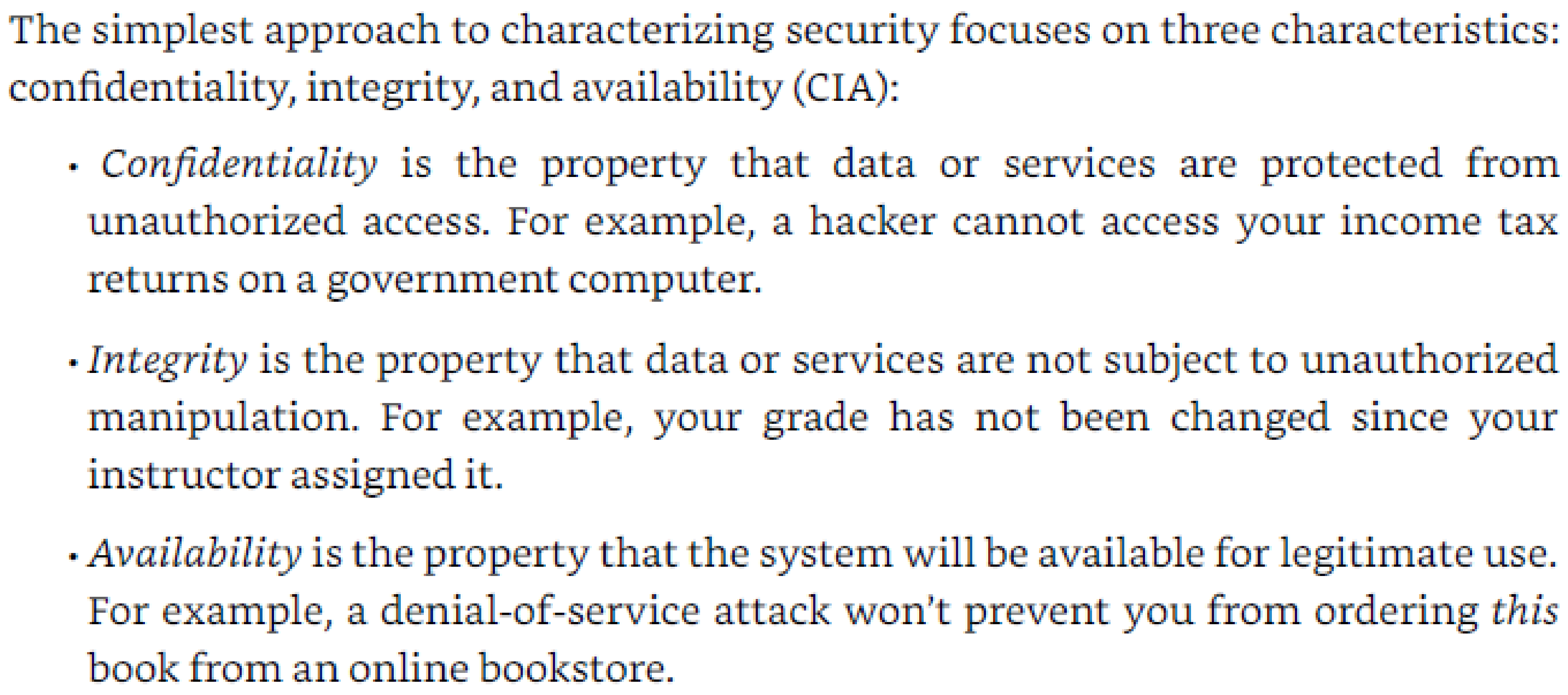


Security



Security is a measure of the system's ability to protect data and information from unauthorized access while still providing access to people and systems that are authorized. An attack—that is, an action taken against a computer system with the intention of doing harm—can take a number of forms. It may be an unauthorized attempt to access data or services or to modify data, or it may be intended to deny services to legitimate users.

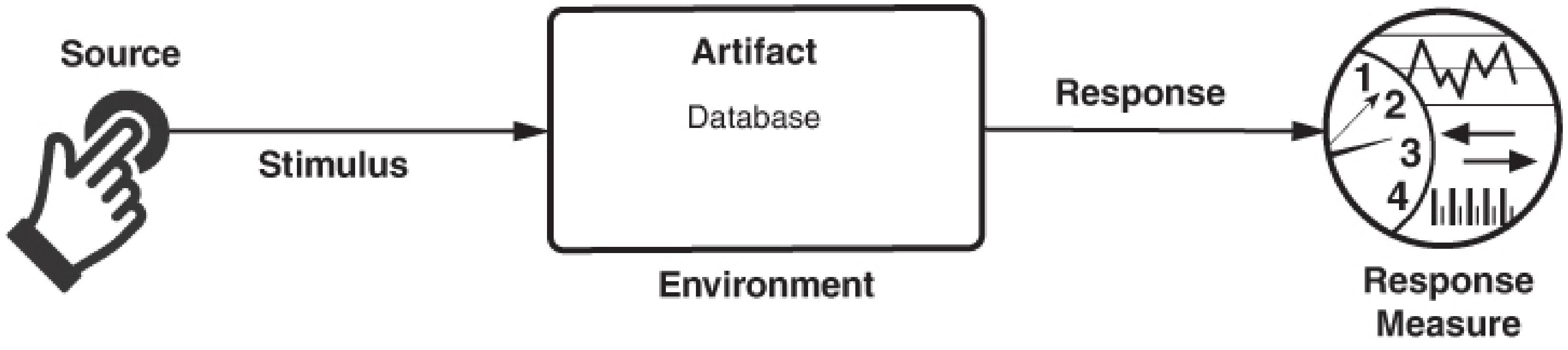
Security



The simplest approach to characterizing security focuses on three characteristics: confidentiality, integrity, and availability (CIA):

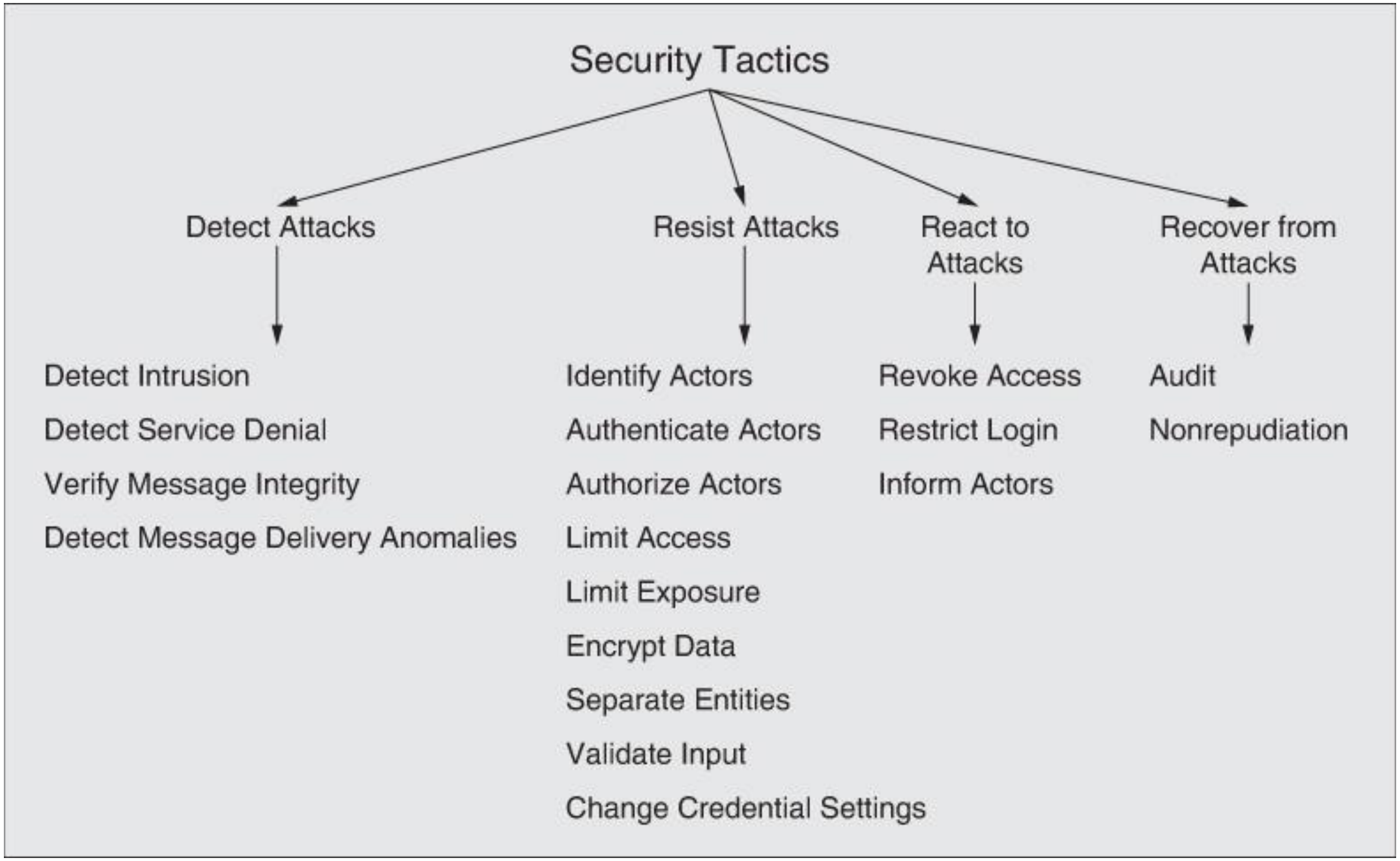
- *Confidentiality* is the property that data or services are protected from unauthorized access. For example, a hacker cannot access your income tax returns on a government computer.
- *Integrity* is the property that data or services are not subject to unauthorized manipulation. For example, your grade has not been changed since your instructor assigned it.
- *Availability* is the property that the system will be available for legitimate use. For example, a denial-of-service attack won't prevent you from ordering *this* book from an online bookstore.

Sample scenario for security



A disgruntled employee at a remote location	Attemps to improperly modify the pay rate table	Normal operations	The unauthorized access is detected and the system maintains an audit trail	Correct data is restored within one day

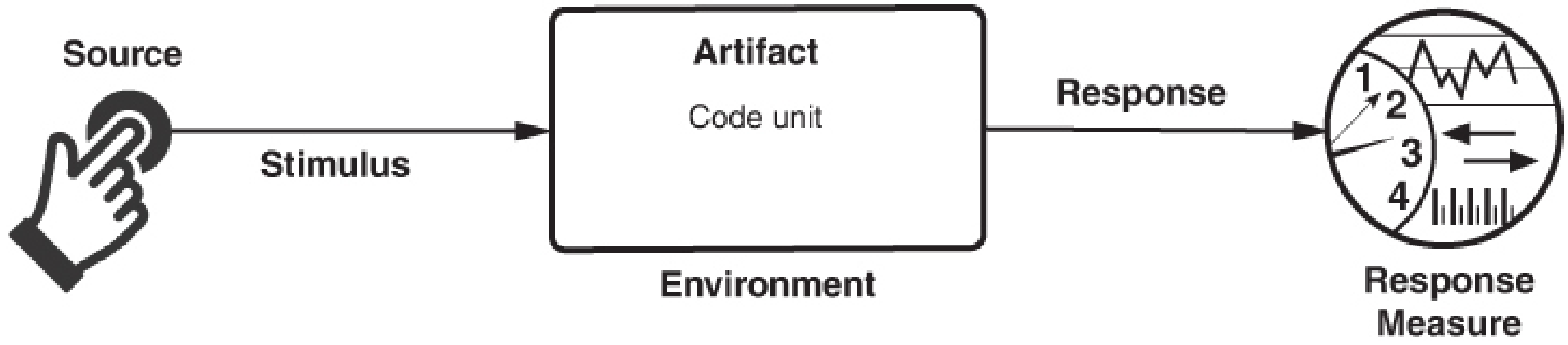
Security tactics



Testability

Software testability refers to the ease with which software can be made to demonstrate its faults through (typically execution-based) testing. Specifically, testability refers to the probability, assuming that the software has at least one fault, that it will fail on its next test execution. Intuitively, a system is testable if it "reveals" its faults easily. If a fault is present in a system, then we want it to fail during testing as quickly as possible. Of course, calculating this probability is not easy and—as you will see when we discuss response measures for testability—other measures will be used. In addition, an architecture can enhance testability by making it easier both to replicate a bug and to narrow down the possible root causes of the bug. We do not typically think of these activities as part of testability per se, but in the end just revealing a bug isn't enough: You also need to find and fix the bug!

Sample testability scenario



Developer

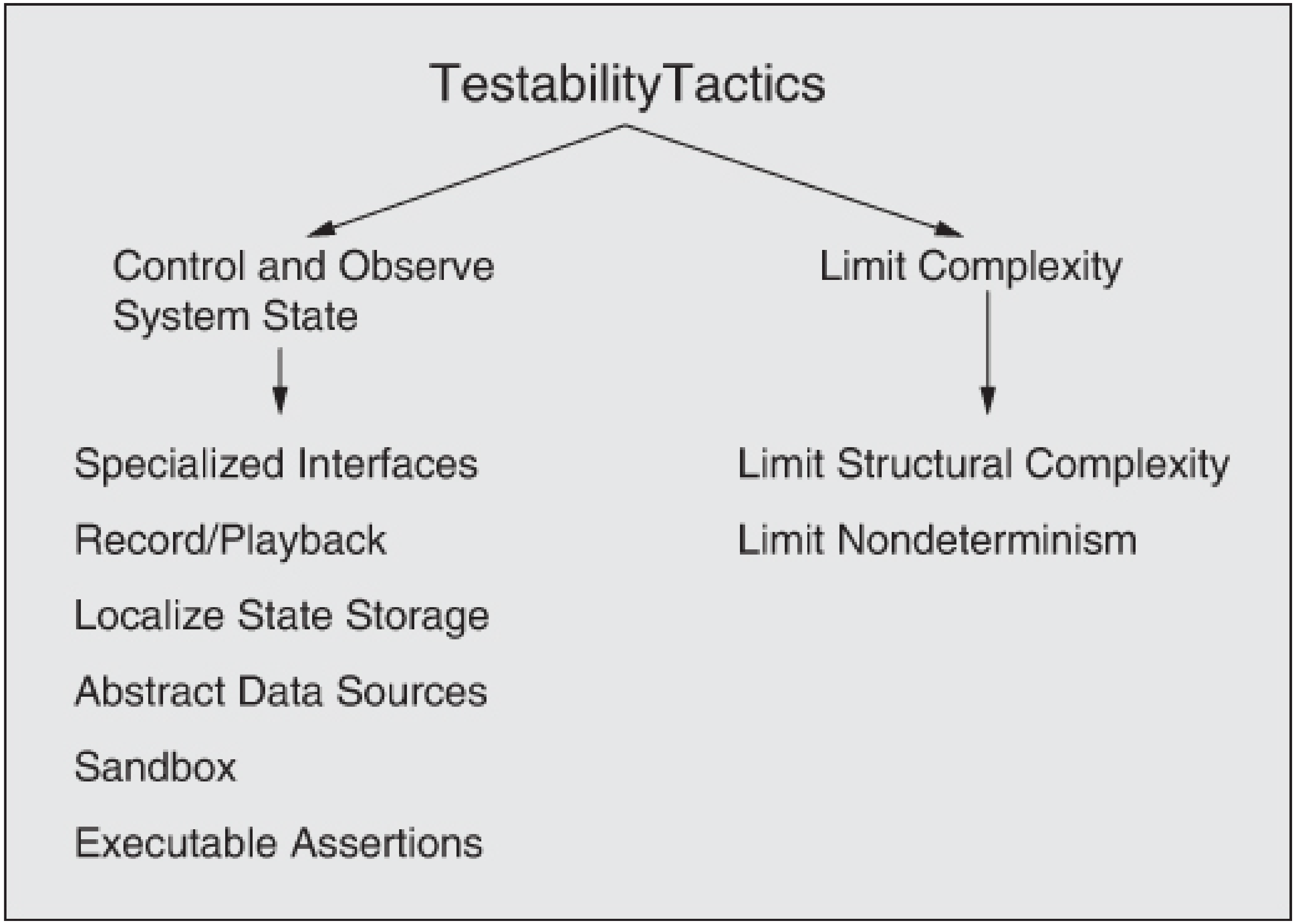
Completes a code unit

Development

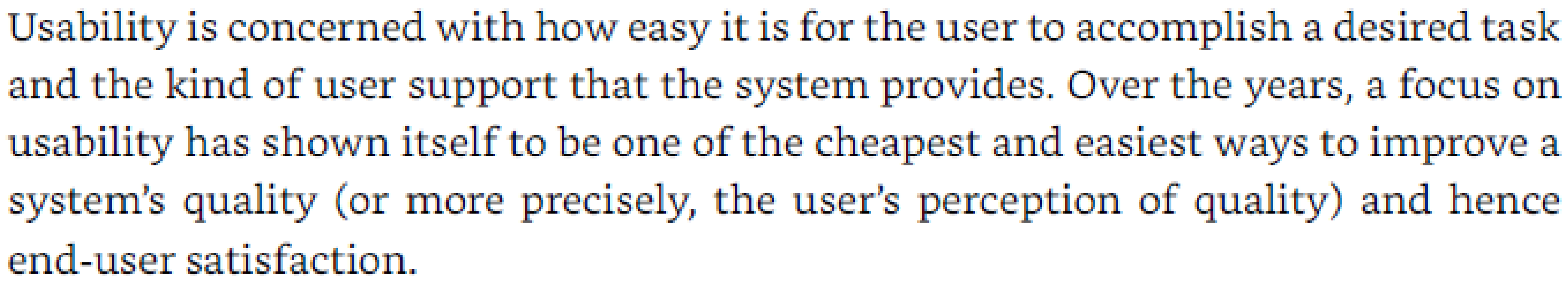
Performs a test sequence

85 % path coverage within 30 minutes

Testability tactics



Usability



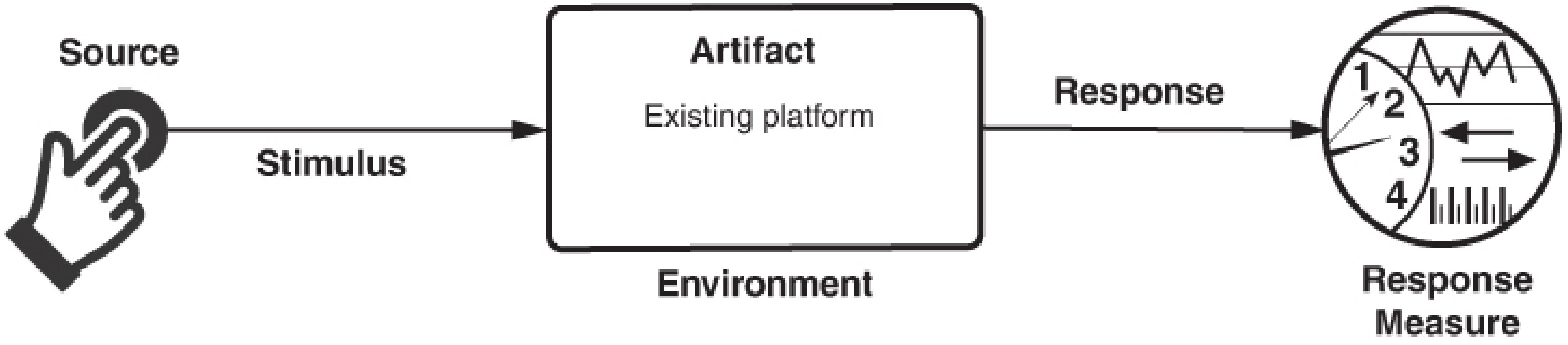
Usability is concerned with how easy it is for the user to accomplish a desired task and the kind of user support that the system provides. Over the years, a focus on usability has shown itself to be one of the cheapest and easiest ways to improve a system's quality (or more precisely, the user's perception of quality) and hence end-user satisfaction.

Usability

Usability comprises the following areas:

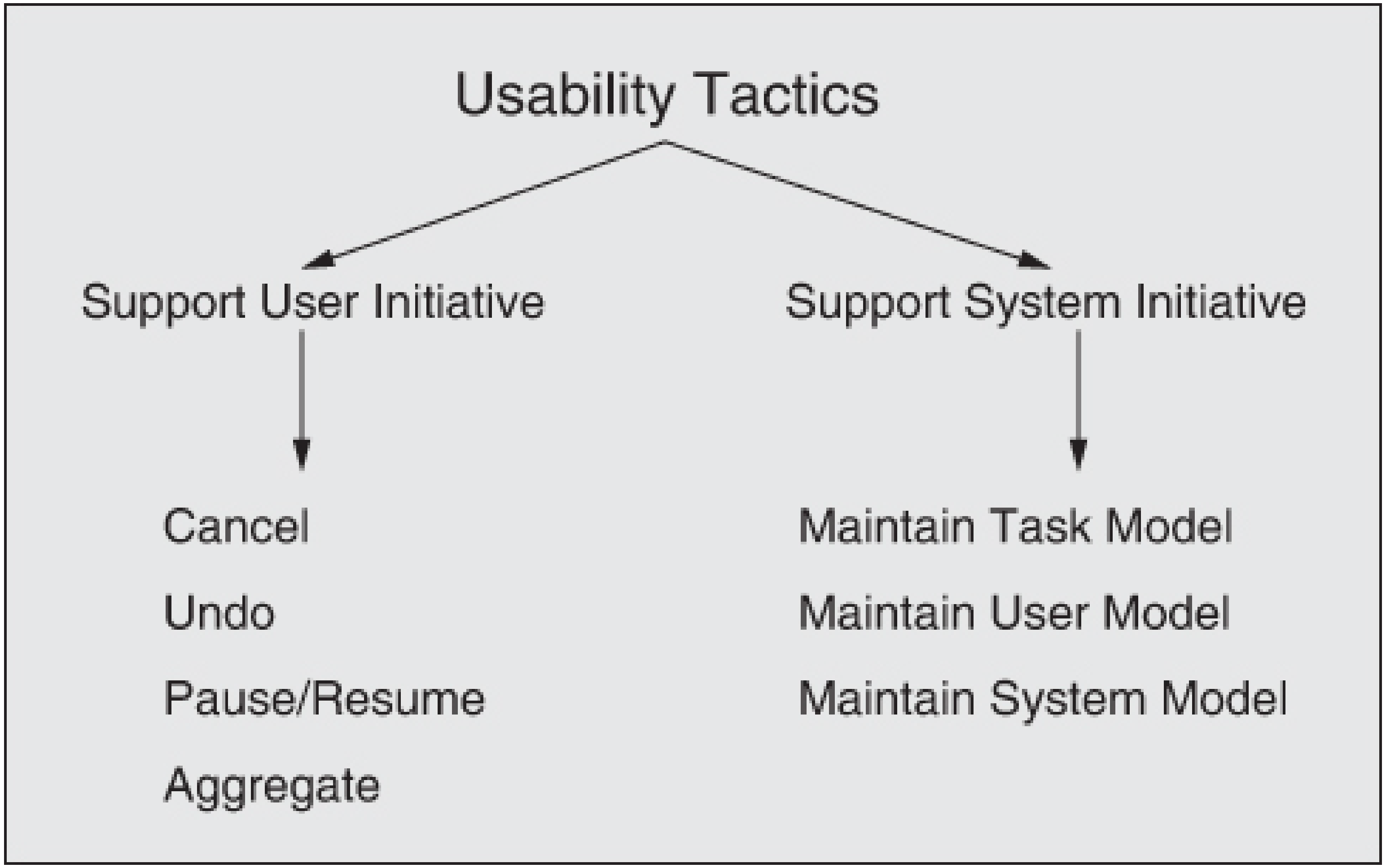
- *Learning system features*. If the user is unfamiliar with a particular system or a particular aspect of it, what can the system do to make the task of learning easier? This might include providing help features.
- *Using a system efficiently*. What can the system do to make the user more efficient in its operation? This might include enabling the user to redirect the system after issuing a command. For example, the user may wish to suspend one task, perform several operations, and then resume that task.
- *Minimizing the impact of user errors*. What can the system do to ensure that a user error has minimal impact? For example, the user may wish to cancel a command issued incorrectly or undo its effects.
- *Adapting the system to user needs*. How can the user (or the system itself) adapt to make the user's task easier? For example, the system may automatically fill in URLs based on a user's past entries.
- *Increasing confidence and satisfaction*. What does the system do to give the user confidence that the correct action is being taken? For example, providing feedback that indicates that the system is performing a long-running task, along with the completion percentage so far, will increase the user's confidence in the system.

Sample usability scenario



User	Downloads a new application	Runtime	Using it productively	>= 2 minutes of experimentation

Usability tactics


Usability Tactics
Support User Initiative
Support System Initiative
Cancel
Undo
Pause/Resume
Aggregate
Maintain Task Model
Maintain User Model
Maintain System Model