POLYTECHNIQUE MONTRÉAL
UNIVERSITÉ D'INGÉNIERIE

LOG8430E: Quality attributes, part 4

Quality attributes – part 4

Definition of quality attributes

1. Availability
2. Interoperability
3. Modifiability
4. Performance
5. Security
6. **Testability**
7. **Usability**
8. **Other quality attributes**

Quality attribute 6 - Testability

The cost of tests of a well-designed software system can represent 30 to 50% of the total development costs, and sometimes more.

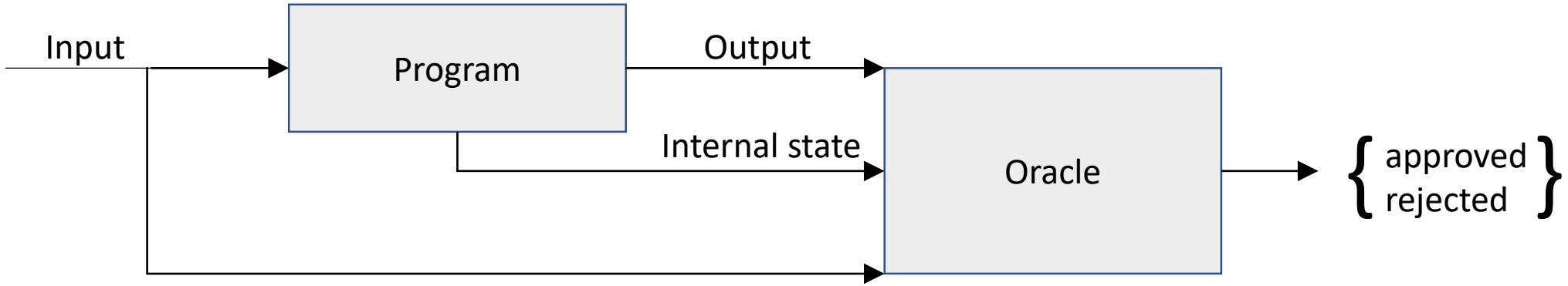
If the architecture can allow to reduce these costs, the return can be even more important.

The testability refers to the facility with which a software can be led to reveal its faults by tests.

The testability refers to the probability, if a fault exists in a software, that this fault will appear at the next execution of a test.

Quality attribute 6 - Testability

A model for the test of a program comprises of the following elements: input data, output results, internal state of the program and an oracle that decides if the achieved results are correct or not.



Quality attribute 6 - Testability

- The oracle is a human agent or a mechanism that decides if the result is correct or not by comparing the output to the specification of the program.
- The output are not only functional, but they can include measures derived from quality attributes, for example, the time necessary to produce a result.
- The internal state of the program can equally be exposed to the oracle, which will decide if the program is in a correct state.
  - Establishing and examining the internal state of a program is an aspect of tests very much present in the tactics for testability.

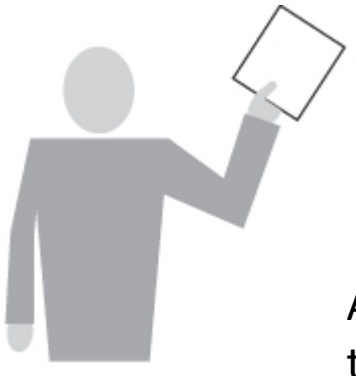
Quality attribute 6 – Testability
Test harness

For a system to be able to be correctly tested, we must be able to measure the input of every component and to observe the output.

It may also be necessary to manipulate and observe the internal state of the component.

This control and these observations are often implemented by a test harness, which permits the execution of procedures to test and register the results.

General scenario of testability



Source of stimulus

Unit tester, Integration tester, System tester, Acceptance and end-user tester, Human or automated system.

Stimulus

A set of executed tests following the completion of a code increment, a class or service layer, a subsystem integration, full system implementation or system delivery.

Artifact

A unit of code (a module), a subsystem or the entire system

Environment

The tests are performed during the development, at compilation, at deployment or at runtime. Can include the test harness.

Response

The system can be checked to perform the desired test and the results can be observed.

Measure of the response

Aiming at measuring how easy a system under test can divulge its faults. The measures can include: the necessary effort to find a fault or a class of faults, the required effort to test a given percentage of statements, the length of the longest test chain, the effort to perform the tests, to find faults, an estimation of the probability find new faults and the time and effort necessary to prepare the environment of testing.

Example of concrete scenario of testability

Source of stimulus

Unit tester

Stimulus
Completed unit of code

Artifact
Code unit

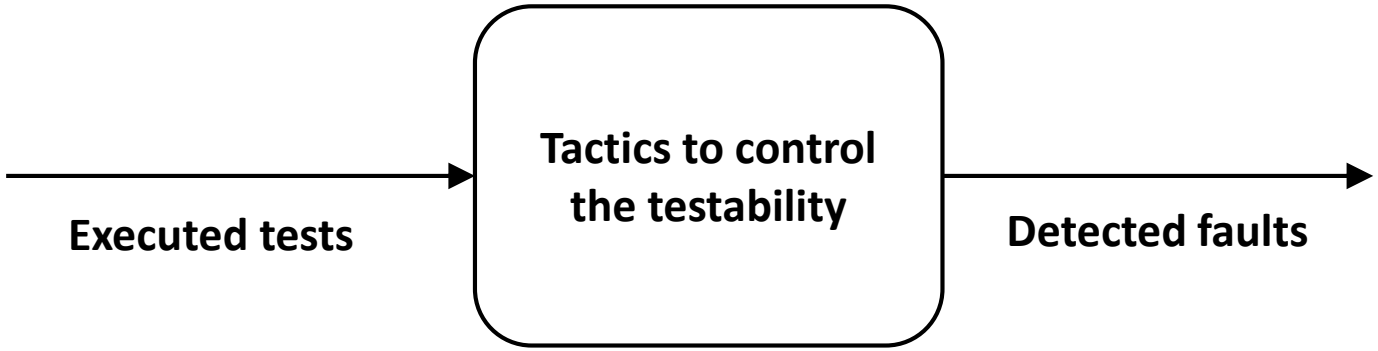
Environment
Development

Response
Registered results

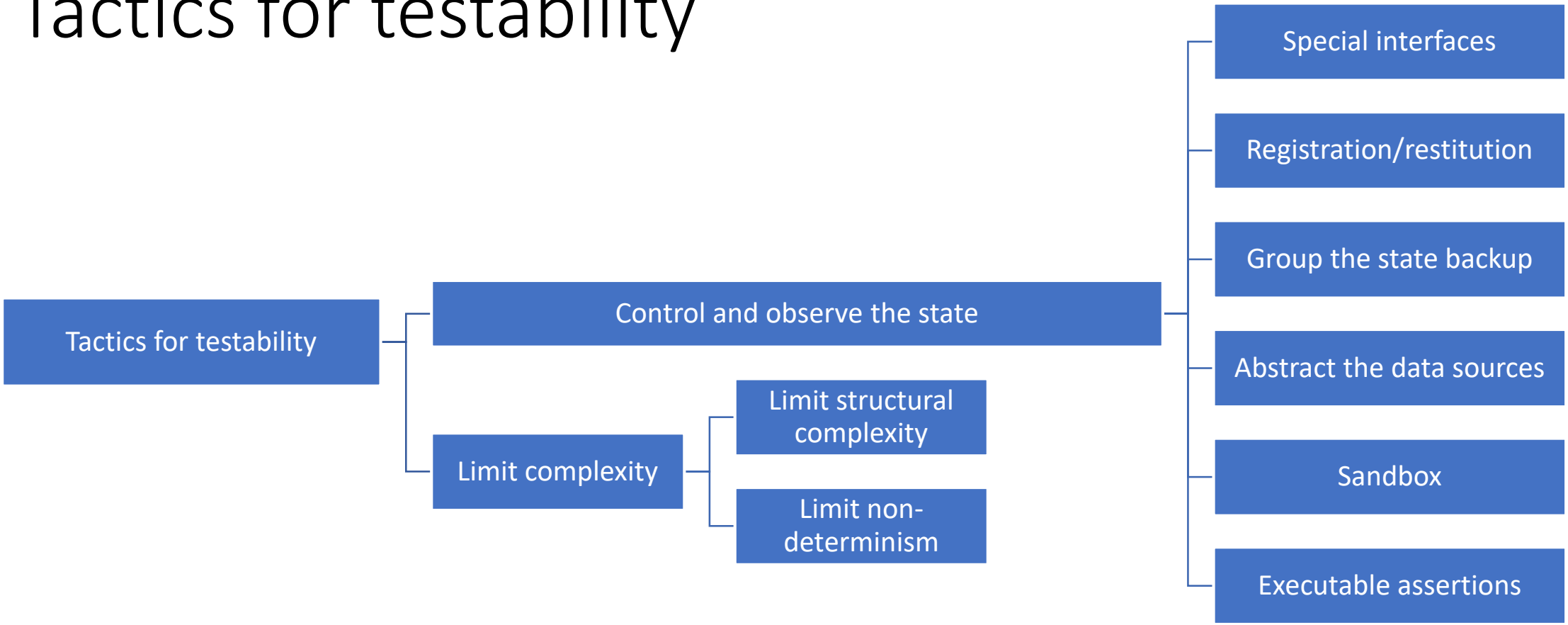
Measure of response
85% of paths covered in three hours

Quality attribute 6: Tactics for testability

The tactics of testability aim to render the tests easier to perform when a part of the software development is completed.



Quality attribute 6: Tactics for testability



Quality attribute 6: Tactics for testability: **Control and observe the state**

- Special interfaces: providing special interfaces for tests allows to control and query the values of variables of a component by a test harness or at normal runtime. For example:
  - Getters and setters for the important variables or the attributes (methods that may not necessarily be generally available, but only for the tests),
  - A method report to return the complete state of an object,
  - A method reset to initialize the internal state of an object,
  - A method to activate a “verbose” mode for output, different registration levels for events or for resource monitoring.
- These test interfaces should be clearly distinguished from standard interfaces provided to implement the functionality, so that they can be removed at will and on demand.

Quality attribute 6: Tactics for testability: **Control and observe the state**

- Registration/restitution: recovering the state of the system at the moment, when a fault occurred can be difficult. Registering the state when an interface is used allows to use this state to “replay” a part of the system to automatically reproduce the fault.
  - This tactic refers both to the registration the state and to the use of the state for testing.

Quality attribute 6: Tactics for testability: **Control and observe the state**

- Group the backup state: to allow for the launch of a system, subsystem, or module in a particular state, it is very practical that this state is stored in a single place. Conversely, if the state is nested or distributed, this becomes very difficult or impossible.
  - The state can be fine-grained (at the level of bits) or coarse-grained to represent global abstractions or operation modes.
  - The choice of granularity depends on what mode of registration of the state will be used in the tests.
  - A convenient way of outsourcing state storage (making the state manageable via an interface) is to use a state machine as a mechanism to track and report the current state.

Quality attribute 6: Tactics for testability: **Control and observe the state**

- Abstract the data sources: as in the state, easily checking the input data facilitates the tests. Abstracting the interfaces allows to easily substitute test data.
  - For example, in a system that uses a database for client transaction, the architecture can be designed in a way to easily use another database or even a file to perform the tests, without changing the functional code.

Quality attribute 6: Tactics for testability: **Control and observe the state**

- Sandbox: isolate an instance of the system to allow for an experiment unconstrained by the necessity to recover the consequences of the experiment.
  - The tests are facilitated by the possibility of operating the system without being concerned by permanent consequences or by a need to be able to return back.
  - Allows for the analysis of scenarios, training or simulation.
- A current form of sandbox consists of virtualizing the resources. To test a system, we must often interact with resources external to the system. The method of sandbox allows to better control the resources.

Quality attribute 6: Tactics for testability: **Control and observe the state**

- Executable assertions: manually place assertions in the code in strategic points to indicate if the program has a fault.
  - Verify that data values satisfy specified constraints.
  - The assertions are defined in terms of specific data declarations and should be placed there or where data are referenced or modified.
  - The assertions can be specified as pre or postconditions for a method or as class invariants.
  - The assertions increase observability.
  - The assertions are a way to incorporate the oracle directly in the code.

Quality attribute 6: Tactics for testability: **Control and observe the state**

All these tactics add capabilities or abstractions to the software that would not be present if we wouldn't like to test it. These additions can be made thanks to different mechanisms:

- Replacement of component, which switches a component with an instrumented version to facilitate testing. Generally during construction.
- Preprocessor macros to activate parts of the code or monitors that provide information or return the control to a testing console.
- Compilation options that activate assertions.
- Aspects that are responsible to report the state.

Quality attribute 6: Tactics for testability: **Limit complexity**

- Limit structural complexity: avoid or elimitate cyclomatique dependencies between components, isolate or encapsulate the dependencies to the external environment and generally reduce the dependencies between the components.
- At a class level, in order to increase testability, we can also:
  - Simplify the inheritance structure by limiting the number of classes that extend or are extended by other classes,
  - Limit the depth of the inheritance tree,
  - Limit polymorphism or dynamic calls,
  - Limit the metric RFC (response for a class) (number of class methods and of other classes called by methods of the class).

Quality attribute 6: Tactics for testability: **Limit complexity**

- Limit the non-determinism: the counterpart of limiting the structural complexity is to limit the behavioral complexity. From the point of view of tests, the non-determinism is a bad form of behavioral complexity. Finding and limiting sources of non-determinism increase the testability:
  - Unconstrained parallelism,
  - Dependencies on memory locations,
  - Stochastic approaches for task planning.

Verification list for testability
Allocation of responsibilities

- Determine what system responsibilities are the most critical ones and need to be tested in depth.
- Ensure that responsibilities have been added to:
  - Execute the test suites and capture the results,
  - Register the activity or activities that have resulted to a fault or to unexpected behavior, that is not necessary a fault,
  - Check the aspects of the system state relevant for testing.
- Ensure that the allocation of functionalities generates high cohesion, low coupling, great separation of responsibilities and low structural complexity.

Verification list for testability
Coordination model

- Ensure that the mechanisms for coordination and communication:
  - Support the execution of a test suite and allow for capturing the results within a system or between systems,
  - Support the registration of activities that have resulted in a fault in a system or between systems,
  - Support the injection or the monitoring statement in communication channels used for tests,
  - Do not introduce unnecessary non-determinism.

Verification list for testability
Data model

- Determine the principal data abstractions that should be tested to ensure the correct operation of the system.
  - Ensure that it is possible to capture the values of instances and of abstractions,
  - Ensure that the values of instances of data abstractions can be initialized when a state is injected in the system to reproduce the circumstances that have led to a fault,
  - Ensure that the creation, initialization, persistence, manipulation, translation and destruction of these instances of data abstractions can be invoked and registered.

Verification list for testability

Correspondence between architectural elements

- Determine how to test different possibilities for correspondences between architectural elements, like:
  - The correspondence of processes to processors,
  - The correspondence of execution queues to processes,
  - The correspondence of modules to components,

  to obtain the desired response in tests, and identify and eliminate the potential conditions of competition.
- Determine if it is possible to test the illegal correspondences between architectural elements.

Verification list for testability
Resource management

- Ensure that there are sufficient available resources to execute the test suites.
- Ensure that the test environment is representative of the execution environment.
- Ensure that the system provides the means to:
  - Test the limits of the resources,
  - Register the details of the resource utilization so that they can be analyzed in case of fault,
  - Inject new limits for the resources in the system for testing purposes.
  - Provide virtual resources for tests.

Verification list for testability
Choice of moment to establish a link

- Ensure that the components that are bound later than the compilation can be tested in the context of delayed binding.
- Ensure that delayed bindings can be registered in case they cause a fault, in order to recreate the state of the system that has led to the fault.
- Ensure that the set of all possible bindings can be tested.

Verification list for testability
Choice of technology

- Determine what technologies are available to help implement the scenarios of testability applicable to the architecture.
- Are technologies available to facilitate the performance of regression tests, the injection of faults, the registration and the restitutions?
- Determine how testable the chosen or considered technologies are and ensure that the chosen technologies support the level of testability required by the system.

Quality attribute 6: Testability
Two challenges: **1 – test data**

- Creating consistent and useful sets of test data can represent a considerable challenge.
- The data for unit and acceptance tests are often generated by hand.
- Generating data for integration tests, allowing to ensure that multiple components function correctly together, and for performance tests, is particularly difficult.

Quality attribute 6: Testability
Two challenges: **1 – test data**

- Two approaches to create data sets:
  - Design a system for data generation
    - Requires considerable effort,
    - Allows to cover all border cases.
  - Register execution data in a production environment
    - Simpler than generating data by hand,
    - Requires the verification of coverage of the entire system and of border cases,
    - Can be constrained by confidentiality and legal concerns.
- Can have an important impact on the architecture to register transactions at runtime or save data snapshots.

Quality attribute 6: Testability
Two challenges: **2 – automation of tests**

- Put in place an environment for automated tests
  - Daily execution (nightly to be more precise)
  - Execution at every change (continuous integration)
- Often neglected in the planning of grand projects
  - No budget or specific resources to put in place the testing infrastructure
  - Can be a complex infrastructure in itself – which requires testing…
- Testing the user interfaces is often very complicated and strongly subject to change.
- Use of specialised tools.

Quality attribute 7: Usability

The usability refers to how easy it is for a user to accomplish a desired task and the level of support offered by the system.

Improving the usability has been identified as one of the most efficient means of improving the quality of a system – or at least the perception of the users on the quality of the system.

Quality attribute 7: Usability

Usability comprises of the following aspects:

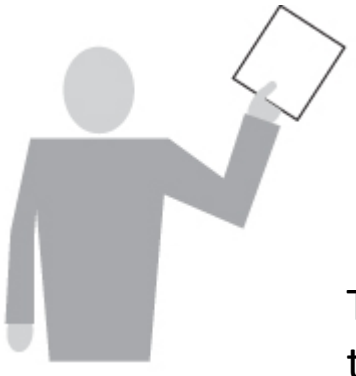
- **Learn the characteristics of the system**. For a user that is not familiar with a system or with certain aspects, how can the system facilitate learning? This can include help mechanisms.
- **Use the system efficiently**. How to make the user more efficient in her operations? This can include the possibility for the user to redirect the system after having sent a command. For example, suspend a task, perform multiple operations and then return to the task.
- **Minimize the impact of errors**. How to make a user error have minimal impact? For example, allow to cancel an incorrect command.

Quality attribute 7: Usability

Usability comprises of the following aspects:

- **Adapt the system to the user needs**. How to make the system to adapt itself in order to facilitate the user tasks? For example, by providing default values learnt dynamically as respond choices.
- **Increase trust and satisfaction.** What does the system do to confirm to the user that the right action is occurring? For example, providing feedback that indicates that a long task is in progress and indicating completed progress.

General scenario for usability



Source of stimulus

End-user, possibly in a special role

Stimulus

The user tries to use the system in an efficient manner, learn to use the system, minimize the impact of errors, adapt or configure the system.

Artifact

The system or specific part of the system with which the user interacts.

Environment

At runtime or at configuration

Response

The system should provide to the user the characteristics they need or anticipate the needs of the user.

Measure the response

One or more of the following:

- Time to perform a task,
- Number of errors,
- Number of completed tasks,
- User satisfaction,
- Increase of user knowledge,
- Ratio of successful operations over total number of operations,
- Time lost or data lost in case of error

Example of concrete scenario of usability

Source of stimulus

User

Stimulus
Download a new application

Artifact
System

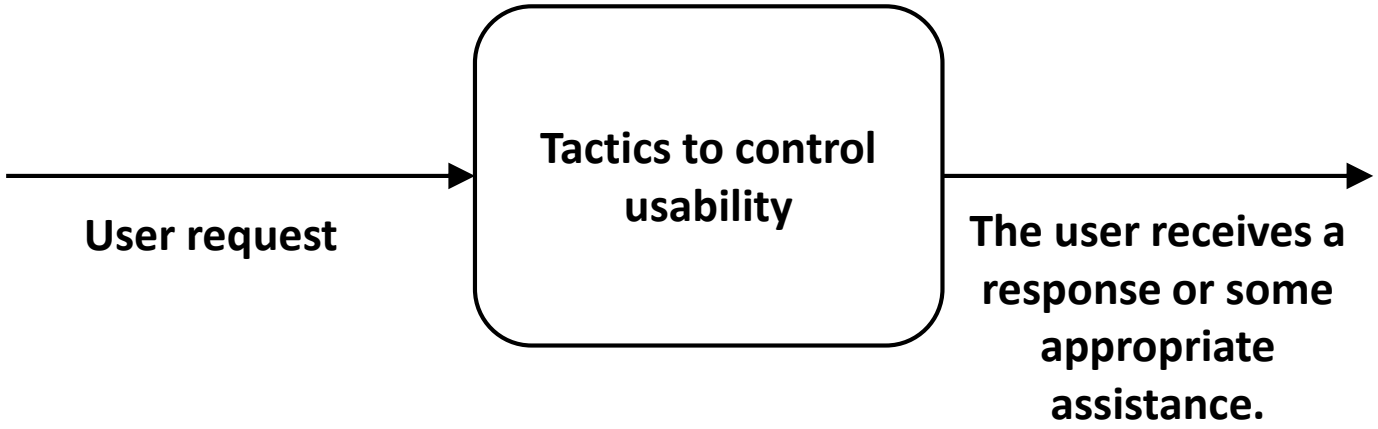
Environment
At runtime

Response
The user uses the application productively.

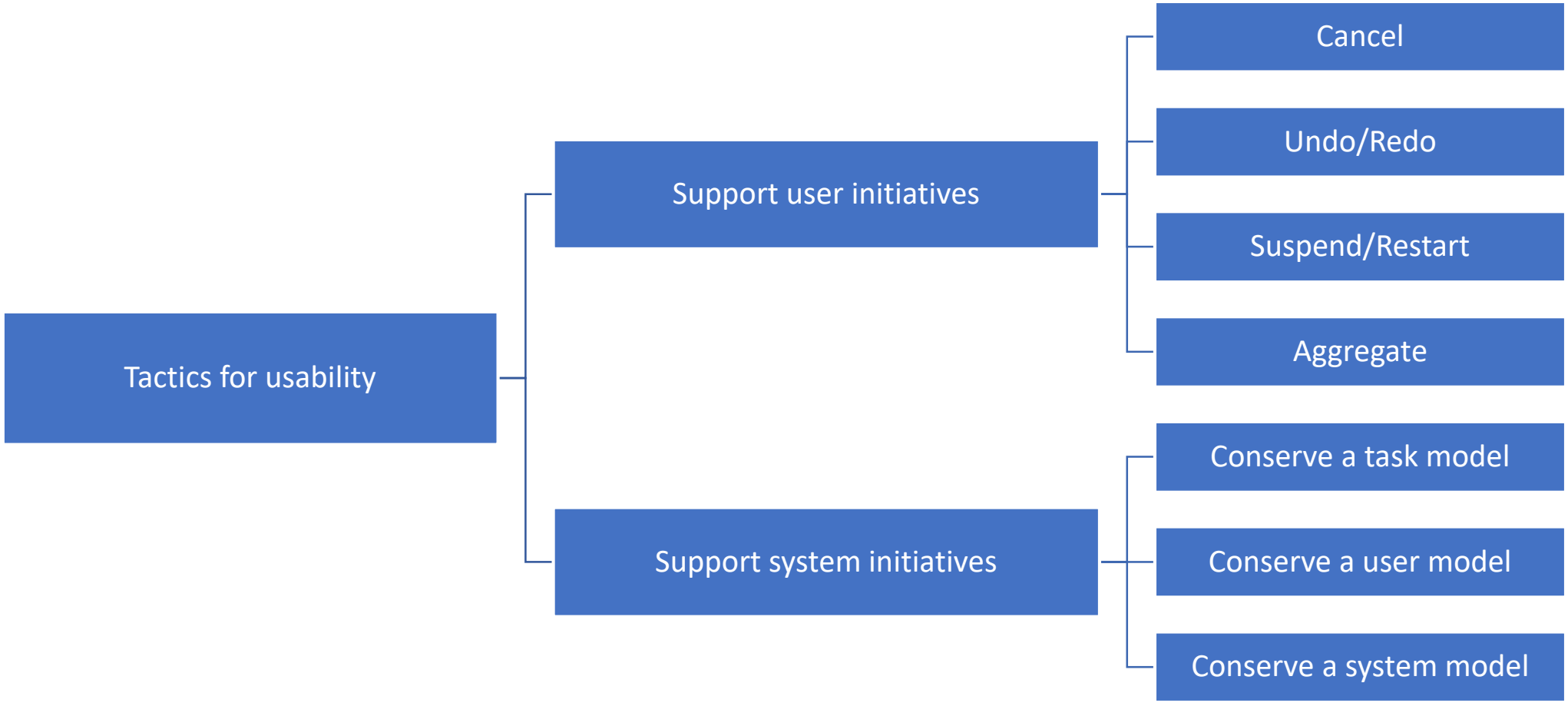
Measure the response
At least 2 minutes of experimentation

Quality attribute 7: Tactics for usability

The tactics for usability aim at rendering the system easier to use effectively and to learn.



Quality attribute 7: Tactics for usability



Quality attribute 7: Tactics for usability: **Support the user initiatives**

- Cancel: When a user sends a cancel command, the system should be listening in order to act immediately:
  - A system component needs to have the responsibility to constantly listen to user events and it must not blocke, whatever the command that should be canceled,
  - The command to be canceled should be terminated immediately and the used resources must be released,
  - Components that collaborate with the canceled command must be notified so that they take appropriate actions.

Quality attribute 7: Tactics for usability: **Support the user initiatives**

- Undo/Redo: to allow the undo or the redo of a command, the system must conserve sufficient information concerning its prior state to be able to restore it.
  - The registration of the state can be in the form of a snapshot, a verification point or a series of reversible operations.
  - Not all operations are reversible. For the non-reversible operations, the system needs to register more information concerning applied changes.
  - Certain operations cannot be simply undone. For example, the emission of a sound.

Quality attribute 7: Tactics for usability: **Support the user initiatives**

- Suspend/Restart: When a user starts a long operations – calculation, download, etc. – it is often useful to allow them to suspend and restart the operation.
  - Involves the capability to temporarily release resources that can be reallocated to other tasks.

Quality attribute 7: Tactics for usability: **Support the user initiatives**

- Aggregate: When the user performs repetitive tasks or tasks that affect multiple objects in a similar fashion, it is useful to provide a mechanism to reassemble the objects at the lower level in a group and apply the operation to the group.

Quality attribute 7: Tactics for usability: **Support the system initiatives**

- Conserve a task model: a task model is used to determine the context so that the system can have an idea of what the user tries to accomplish and be able to provide help.
  - For example, spell checking in text editors.

Quality attribute 7: Tactics for usability: **Support the system initiatives**

- Conserver a user model: this model represents explicitely the user's knowledge about the system, the behavior of the user in terms of the expected response and other aspects specific to a particular user or a class of users.
  - For example, the system can modify the movement of the mouse during a selection to allow for a precise selection in a text section.
  - The model can control the quantity of help or the offered suggestions to a user.
  - A particular case of this tactic is to provide to the user the possibility to customize the interface of the system, which acts directly on the user model.

Quality attribute 7: Tactics for usability: **Support the system initiatives**

- Conserver a system model: in this case, the system conserves a model specific to itself, so that it can predict its proper behavior and provide appropriate responses to the user.
  - For example, provide a progress indicator that shows the time that remains before completing an operation.

Verification list for the usability
Allocation of responsibilities

- Ensure that additional responsibilities have been allocation, on demand, in order to assist the user to:
  - Learn how to use the system,
  - Efficiently perform a task,
  - Adapt and configure the system,
  - Recover from user and system error.

Verification list for the usability
Coordination model

- Determine if the coordination properties of the system elements, such as:
  - the punctuality, the completeness, the exactness and the consistency

  affect the way that the user:
  - Learns to use the system,
  - Achieves their objectives or completes a task,
  - Adapts or configures the system,
  - Recovers from user or system errors,
  - Acquires more trust and satisfaction.
- For example:
  - Can the system respond to mouse events and provide a real-time response?
  - Can a task that requires a long execution time be interrupted with reasonable delay?

Verification list for the usability
Data model

- Determine the principal data abstractions that are involved in behaviors perceivable by the users.
- Ensure that these abstractions, their operations and their properties are designed in a way to assist the user to:
  - Accomplish tasks,
  - Adapt and configure the system,
  - Recover from user or system errors,
  - Acquire more trust and satisfaction.
- For example,
  - The abstractions should be designed in order to allow the undo/redo of an operation,
  - The granularity of transactions should not be too fine that makes it too long to undo an operation.

Verification list for the usability
Correspondence between architectural elements

- Determine which correspondences between architectural elements are visible to the user. For example:
  - Verify to what degree the user is aware of services that are local and those that are distant.
- For correspondences visible to the users, determine in what way or how easily the user can:
  - Learn to use the system,
  - Achieve their objective or complete a task,
  - Adapt or configure the system,
  - Recover from user or system errors,
  - Acquire more trust and satisfaction.

Verification list for the usability
Resource management

- Determine how a user can adapt or configure the utilization of resources by the system.
- Ensure that the limits on the resources for all configurations controlled by the user will not reduce the probability of achieving the objectives of the user. For example:
  - Avoid the configurations that will result in excessive response times.
- Ensure that the level of resources will not affect the capability of the user to learn to use the system or that it will reduce the their level of trust or satisfaction by the system.

Verification list for the usability
Choice of moment to establish a link

- Determine what binding decisions should be controlled by the user and ensure that users can make decisions that contribute to the usability.
- For example, if the user can choose at runtime the configuration of the system its communication protocols or its functionality thanks to plugins, ensure that these choices will not negatively affect the capability of the user to:
  - Learn to use the system,
  - Achieve their objectives or complete a task,
  - Adapt or configure the system,
  - Recover from user or system errors,
  - Acquire more trust and satisfaction.

Verification list for the usability
Choice of technology

- Ensure that the chosen technologies contribute to implement the usability scenarios that apply to the system. For example, do these technologies help in the creation of online help, in the production of training material and in the interpretation of user feedback?
- How easy are the chosen technologies for the user? Ensure that the chosen technologies will not negatively affect the usability of the system.

Other quality attributes

Besides the presented attributes, many other quality attributes can become important according to the specific characteristics of the system. Among these, we often find:

- The variability,
- The portabilitym
- The distribution of development,
- The scalability,
- The deployability,
- The mobility,
- The capacity to be monitored,
- The safety.

Variability

- A particular form of modifiability that aims to support a set of variants for the system, that differ between each other in a planned way.
- This quality is particularly important in the context of a product line that needs to support multiple products of the same family and the utilisation needs to be adapted for each product.
- The goal of variability is to make it easy to construct and maintain products of the product line for a given time.
- The scenarios of variability analyse the moments when links are established and the actors that establish the links.

Portability

- Also a particular form of modifiability that refers to ease with which a system constructed for a given platform can be changed to run on a different platform.
- The portability is obtained by minimizing the dependencies to the platform within the software, by isolating the dependencies in well-defined places, and by writing the software so that it runs on a virtual machine (like JVM) that encapsulates the dependencies to the platform.
- The scenarios of portability describe the migration of a software towards a new platform by investing a certain amount of maximum effort or by accounting for the number of places in the software that should be modified.

Distribution of development

- The distribution refers to the quality of a software architecture to support the development by distributed teams.
- One problem concerns the coordination of activities between the teams, a need that should be minimized.
- This minimization of the coordination needs to be realized both for the code and for the data model.
- Involves the negotiation between development teams over the interfaces of the modules that need to communicate between themselves.
- The scenarios of distribution analyse the compatibility of communication structures and of data models of the system to be developed and the coordination mechanisms of the involved organisations.

Scalability

- Two types of scaling:
  1. Horizontal scaling, where new resources are added to logical units, for example a new server in a cluster.
  2. Vertical scaling, where new resources are added to physical units, for example add memory to a given computer.
- Up to what point the architecture of the system allows to efficiently use new resources. Can a measurable improvement be observed, without considerable effort or unacceptable service interruption?
- In cloud environments, the capacity to horizontally scale is called elasticity.

Deployability

- How is an executable placed in a host computer and how is it invoked?
- Among the questions to examine:
  - How do updates happen? Push mode, without user intervention, or pull mode, where the user launches an update.
  - How are updates integrated? Is it possible to integrate updates at runtime?
  - How is bandwidth handled on mobile devices?
- The scenarios of deployability analyse the types of updates, their format, the integration in the existing system, the efficiency of the process and the associate risks.

Mobility

- Mobility examines problems related to moves and necessary accommodations to support a platform:
  - Size, type of screen, type of input device, availability and size of bandwidth, battery life.
- Requires an ability to reconnect in case of interruption of communication and different user interfaces to support multiple platforms.
- The scenarios examine the necessary accommodations to support the move and the variability of platforms.

Capacity to be monitored

- Capacity of the support staff to follow the operations of a system at runtime.
- Different types of relevant information:
  - The length and the number of events in queues,
  - The average processing time of transactions,
  - The state of different components of the system.
- The scenarios of monitoring analyse the potential problems, their visibility to the support staff and the potential corrective actions.

Safety

- Software safety refers to the capacity of a system to avoid entering a state that can cause or lead to damage, injuries or loss of lives, and to its capacity to recover and limit the damage when it enters a bad state.
- Safety concerns itself with the prevention and the recovery in case of a fault. Safety, then, borrows a lot of its scenarios and tactics from availability, with which is tightly related.
- Safety should not be confused with reliability. A system can be reliable if it respects its specifications, but not safe, if its specifications allow it to perform dangerous operations.