## 6. State-based testing

***State machine*: implementation-independent specification (model) of the dynamic behavior of the system**

§ ***state*:** abstract situation in the life cycle of a system entity (for instance, the contents of an object)

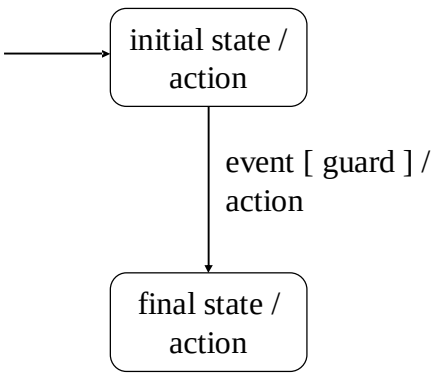
§ ***event*:** a particular input (for instance, a message or method call)

§ ***action*:** the result, output or operation that follows an event

§ ***transition*:** an allowable two-state sequence, that is, a change of state (”firing”) caused by an event

§ ***guard*:** predicate expression associated with an event, stating a Boolean restriction for a transition to fire





There are several types of state machines:

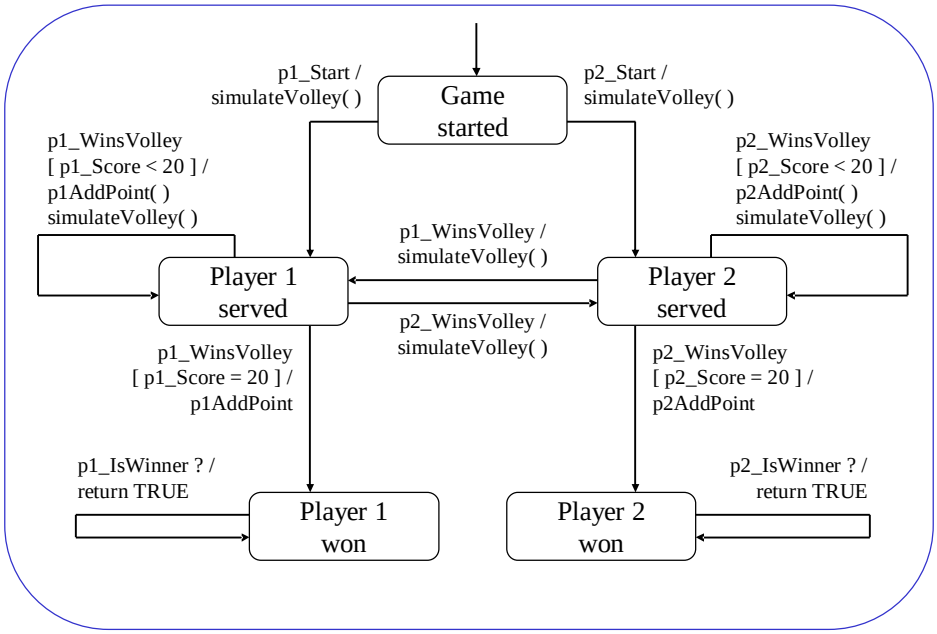
- finite automaton (no guards or actions)
- Mealy machine (no actions associated with states)
- Moore machine (no actions associated with transitions)
- statechart (hierarchical states: common superstates)

- state transition diagram: graphic representation of a state machine
- state transition table: tabular representation of a state machine



**Example**: Mealy model of a two-player video game

- each player has a start button
- the player who presses the start button first gets the first serve
- the current player serves and a volley follows:
  - if the server misses the ball, the server's opponent becomes the server
  - if the server's opponent misses the ball, the server's score is incremented and the server gets to serve again
  - if the server's opponent misses the ball and the server's score is at game point, the server is declared the winner (here: a score of 21 wins)






## General properties of state machines

- typically incomplete
  - just the most important states, events and transitions are given
  - usually just legal events are associated with transitions; illegal events (such as *p1_Start* from state *Player 1 served*) are left undefined
- may be deterministic or nondeterministic
  - deterministic: any state/event/guard triple fires a unique transition
  - nondeterministic: the same state/event/guard triple may fire several transitions, and the firing transition may differ in different cases
- may have several final states (or none: infinite computations)
- may contain empty events (default transitions)
- may be concurrent: the machine (statechart) can be in several different states at the same time

## The role of state machines in software testing

- Framework for *model testing*, where an executable model (state machine) is executed or simulated with event sequences as test cases, before starting the actual implemention phase
- Support for testing the system implementation (program) against the system specification (state machine)
- Support for automatic generation of test cases for the implementation
  - there must be an explicit mapping between the elements of the state machine (states, events, actions, transitions, guards) and the elements of the implementation (e.g., classes, objects, attributes, messages, methods, expressions)
  - the current state of the state machine underlying the implementation must be checkable, either by the runtime environment or by the implementation itself (*built-in tests* with, e.g., assertions and class invariants)



## Validation of state machines

Checklist for analyzing that the state machine is complete and consistent enough for model or implementation testing:

- one state is designated as the initial state with outgoing transitions
- at least one state is designated as a final state with only incoming transitions; if not, the conditions for termination shall be made explicit
- there are no equivalent states (states for which all possible outbound event sequences result in identical action sequences)
- every state is reachable from the initial state
- at least one final state is reachable from all the other states
- every defined event and action appears in at least one transition (or state)
- except for the initial and final states, every state has at least one incoming transition and at least one outgoing transition

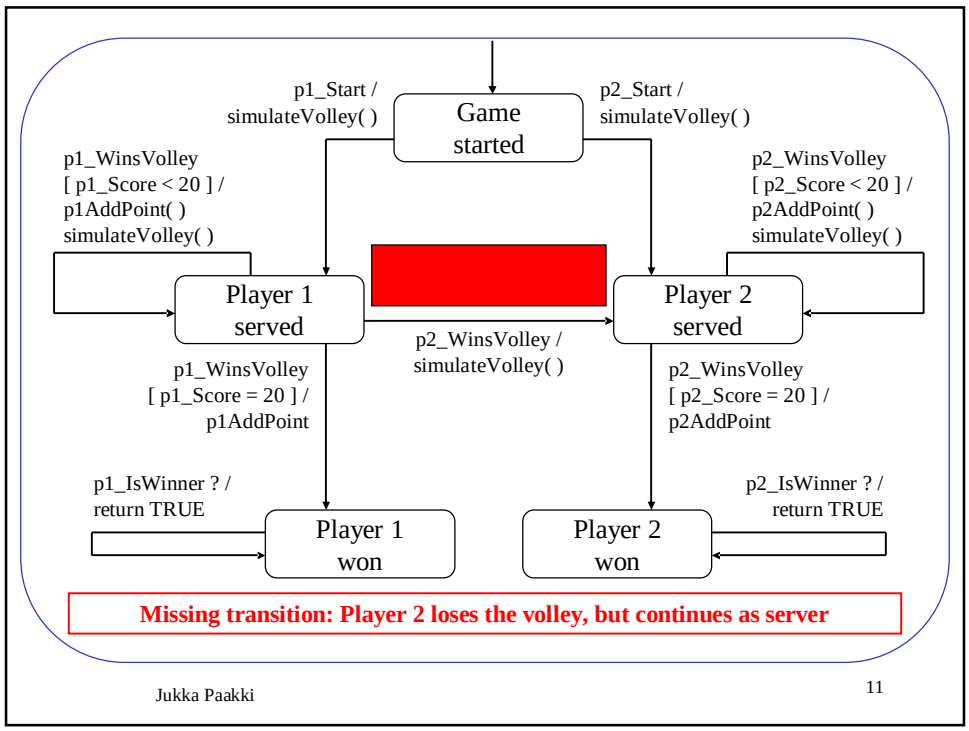
- for deterministic machines, the events accepted in a particular state are unique or differentiated by mutually exclusive guard expressions
- the state machine is completely specified: every state/event pair has at least one transition, resulting in a defined state; or there is an explicit specification of an error-handling or exception-handling mechanism for events that are implicitly rejected (with no specified transition)
- the entire range of truth values (true, false) must be covered by the guard expressions associated with the same event accepted in a particular state
- the evaluation of a guard expression does not produce any side effects in the implementation under test
- no action produces side effects that would corrupt or change the resultant state associated with that action
- a timeout interval (with a recovery mechanism) is specified for each state
- state, event and action names are unambiguous and meaningful in the context of the application



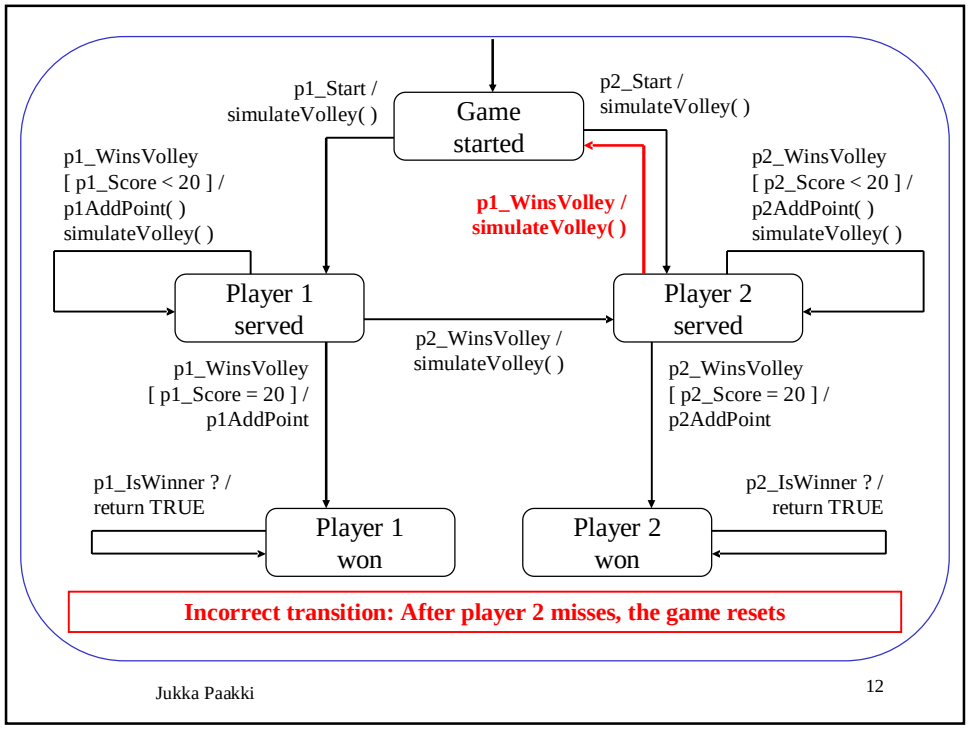
## Control faults

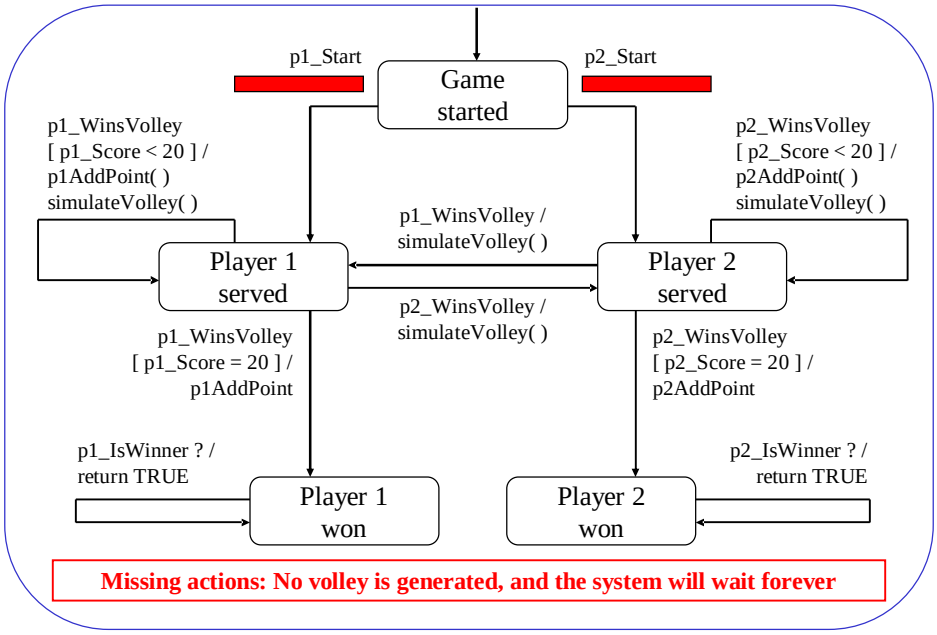
When testing an implementation against a state machine, one shall study the following typical control faults (incorrect sequences of events, transitions, or actions):

- missing transition (nothing happens with an event)
- incorrect transition (the resultant state is incorrect)
- missing or incorrect event
- missing or incorrect action (wrong things happen as a result of a transition)
- extra, missing or corrupt state
- sneak path (an event is accepted when it should not be)
- trap door (the implementation accepts undefined events)

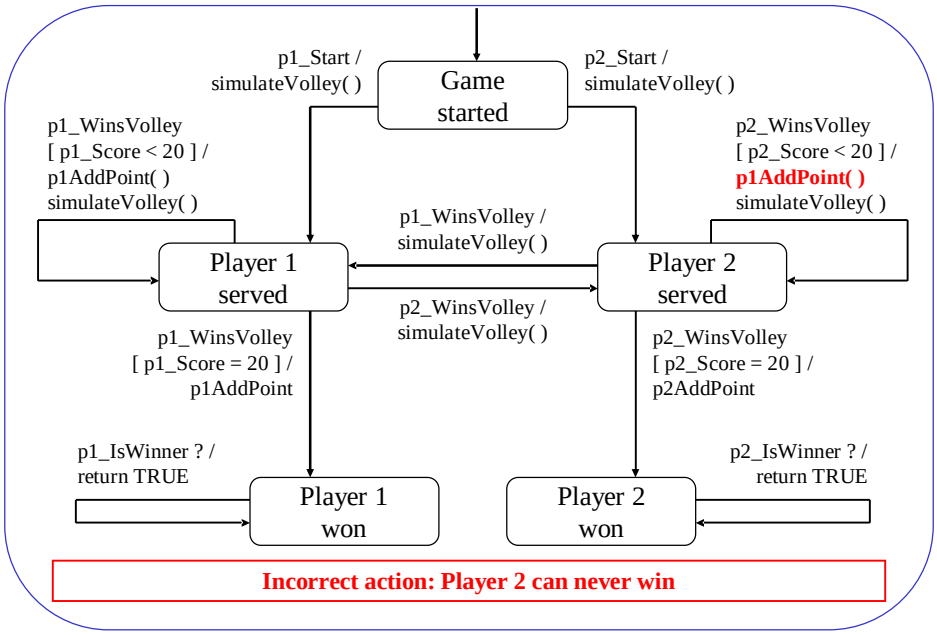

p1_Start /
simulateVolley( )
Game
started
p2_Start /
simulateVolley( )
p1_WinsVolley
[ p1_Score < 20 ] /
p1AddPoint( )
simulateVolley( )
p2_WinsVolley
[ p2_Score < 20 ] /
p2AddPoint( )
simulateVolley( )
Player 1
served
Player 2
served
p2_WinsVolley /
simulateVolley( )
p1_WinsVolley
[ p1_Score = 20 ] /
p1AddPoint
p2_WinsVolley
[ p2_Score = 20 ] /
p2AddPoint
p1_IsWinner ? /
return TRUE
p2_IsWinner ? /
return TRUE
Player 1
won
Player 2
won
Missing transition: Player 2 loses the volley, but continues as server

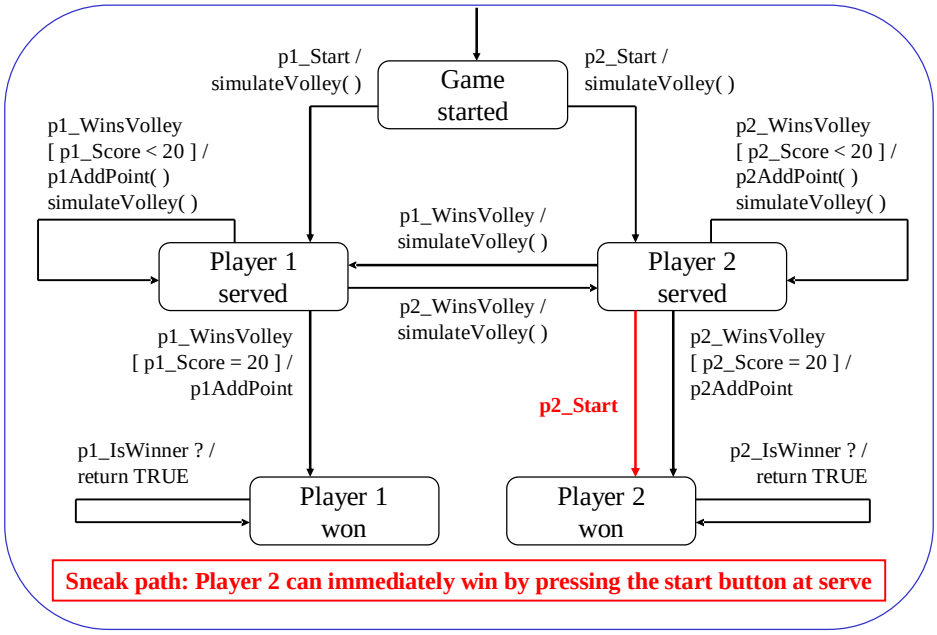



p1_Start /
simulateVolley( )
Game
started
p2_Start /
simulateVolley( )
p1_WinsVolley
[ p1_Score < 20 ] /
p1AddPoint( )
simulateVolley( )
p1_WinsVolley /
simulateVolley( )
p2_WinsVolley
[ p2_Score < 20 ] /
p2AddPoint( )
simulateVolley( )
Player 1
served
Player 2
served
p2_WinsVolley /
simulateVolley( )
p1_WinsVolley
[ p1_Score = 20 ] /
p1AddPoint
p2_WinsVolley
[ p2_Score = 20 ] /
p2AddPoint
p1_IsWinner ? /
return TRUE
p2_IsWinner ? /
return TRUE
Player 1
won
Player 2
won
Incorrect transition: After player 2 misses, the game resets

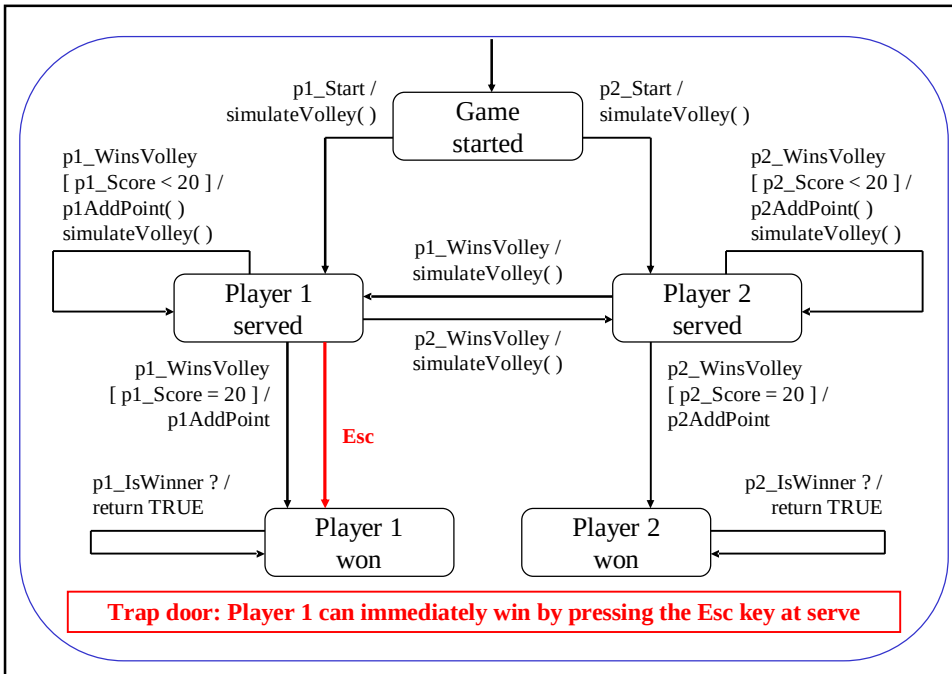

p1_Start
p2_Start
Game started
p1_WinsVolley
[ p1_Score < 20 ] /
p1AddPoint( )
simulateVolley( )
p2_WinsVolley
[ p2_Score < 20 ] /
p2AddPoint( )
simulateVolley( )
p1_WinsVolley /
simulateVolley( )
Player 1 served
Player 2 served
p2_WinsVolley /
simulateVolley( )
p1_WinsVolley
[ p1_Score = 20 ] /
p1AddPoint
p2_WinsVolley
[ p2_Score = 20 ] /
p2AddPoint
p1_IsWinner ? /
return TRUE
p2_IsWinner ? /
return TRUE
Player 1 won
Player 2 won
Missing actions: No volley is generated, and the system will wait forever




p1_Start /
simulateVolley( )
p2_Start /
simulateVolley( )
Game started
p1_WinsVolley
[ p1_Score < 20 ] /
p1AddPoint( )
simulateVolley( )
p2_WinsVolley
[ p2_Score < 20 ] /
p1AddPoint( )
simulateVolley( )
p1_WinsVolley /
simulateVolley( )
Player 1 served
Player 2 served
p2_WinsVolley /
simulateVolley( )
p1_WinsVolley
[ p1_Score = 20 ] /
p1AddPoint
p2_WinsVolley
[ p2_Score = 20 ] /
p2AddPoint
p1_IsWinner ? /
return TRUE
p2_IsWinner ? /
return TRUE
Player 1 won
Player 2 won
Incorrect action: Player 2 can never win


Game started
p1_Start / simulateVolley( )
p2_Start / simulateVolley( )
p1_WinsVolley [ p1_Score < 20 ] / p1AddPoint( ) simulateVolley( )
p2_WinsVolley [ p2_Score < 20 ] / p2AddPoint( ) simulateVolley( )
p1_WinsVolley / simulateVolley( )
p2_WinsVolley / simulateVolley( )
Player 1 served
Player 2 served
p1_WinsVolley [ p1_Score = 20 ] / p1AddPoint
p2_WinsVolley [ p2_Score = 20 ] / p2AddPoint
p2_Start
p1_IsWinner ? / return TRUE
p2_IsWinner ? / return TRUE
Player 1 won
Player 2 won
Sneak path: Player 2 can immediately win by pressing the start button at serve




Game started
p1_Start / simulateVolley( )
p2_Start / simulateVolley( )
p1_WinsVolley [ p1_Score < 20 ] / p1AddPoint( ) simulateVolley( )
p2_WinsVolley [ p2_Score < 20 ] / p2AddPoint( ) simulateVolley( )
p1_WinsVolley / simulateVolley( )
p2_WinsVolley / simulateVolley( )
Player 1 served
Player 2 served
p1_WinsVolley [ p1_Score = 20 ] / p1AddPoint
p2_WinsVolley [ p2_Score = 20 ] / p2AddPoint
Esc
p1_IsWinner ? / return TRUE
p2_IsWinner ? / return TRUE
Player 1 won
Player 2 won
Trap door: Player 1 can immediately win by pressing the Esc key at serve

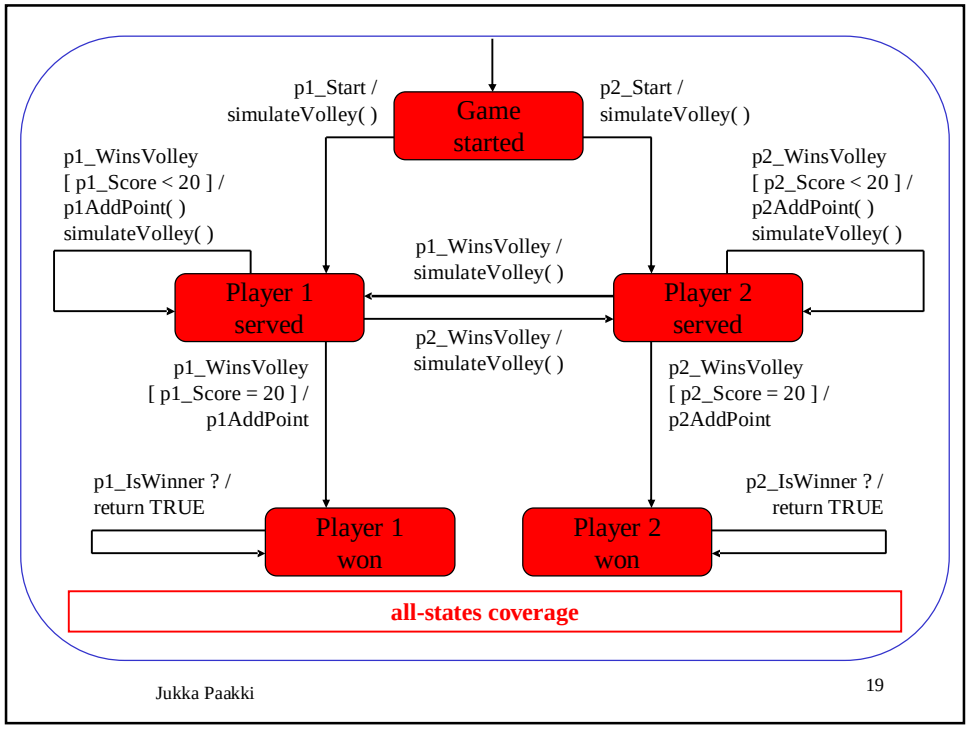
## Test design strategies for state-based testing

Test cases for state machines and their implementations can be designed using the same notion of *coverage* as in white-box testing:

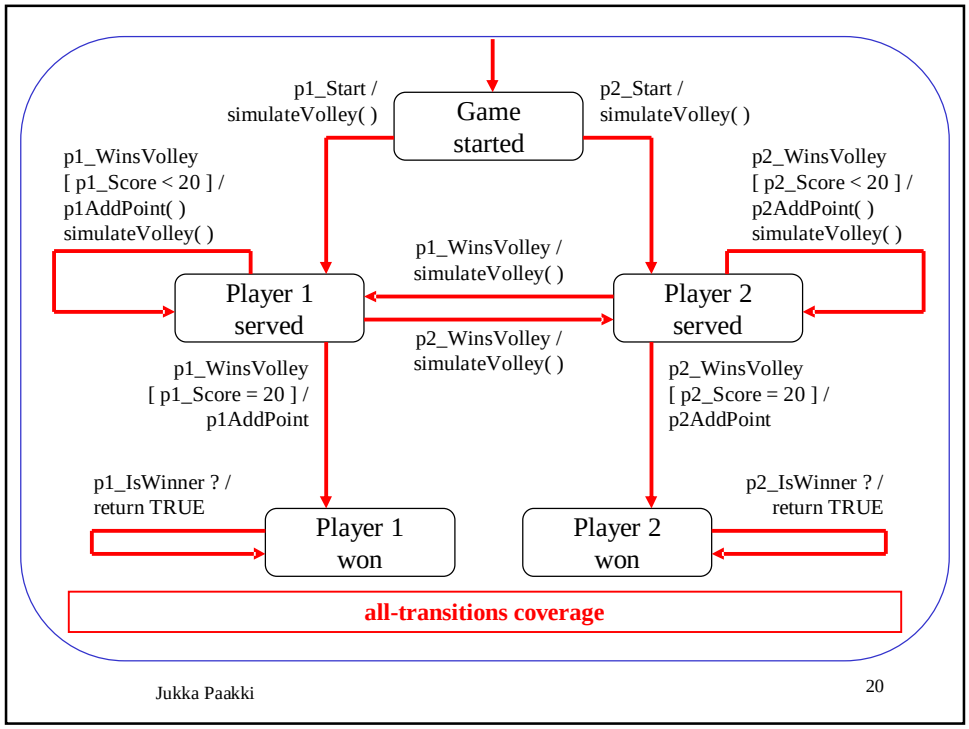
- ***test case*** = sequence of input events
- ***all-events coverage***: each event of the state machine is included in the test suite (is part of at least one test case)
- ***all-states coverage***: each state of the state machine is exercised at least once during testing, by some test case in the test suite
- ***all-actions coverage***: each action is executed at least once



- ***all-transitions***: each transition is exercised at least once
  - implies (subsumes) all-events coverage, all-states coverage, and all-actions coverage
  - ”minimum acceptable strategy for responsible testing of a state machine”
- ***all n-transition sequences***: every transition sequence generated by *n* events is exercised at least once
  - *all transitions = all 1-transition sequences*
  - *all n-transition sequences* implies (subsumes) *all* (*n-1*)*-transition sequences*
- ***all round-trip paths***: every sequence of transitions beginning and ending in the same state is exercised at least once
- ***exhaustive***: every path over the state machine is exercised at least once
  - usually totally impossible or at least unpractical


p1_Start /
simulateVolley( )
Game
started
p2_Start /
simulateVolley( )
p1_WinsVolley
[ p1_Score < 20 ] /
p1AddPoint( )
simulateVolley( )
p2_WinsVolley
[ p2_Score < 20 ] /
p2AddPoint( )
simulateVolley( )
p1_WinsVolley /
simulateVolley( )
Player 1
served
Player 2
served
p2_WinsVolley /
simulateVolley( )
p1_WinsVolley
[ p1_Score = 20 ] /
p1AddPoint
p2_WinsVolley
[ p2_Score = 20 ] /
p2AddPoint
p1_IsWinner ? /
return TRUE
p2_IsWinner ? /
return TRUE
Player 1
won
Player 2
won
all-states coverage




p1_Start /
simulateVolley( )
Game
started
p2_Start /
simulateVolley( )
p1_WinsVolley
[ p1_Score < 20 ] /
p1AddPoint( )
simulateVolley( )
p2_WinsVolley
[ p2_Score < 20 ] /
p2AddPoint( )
simulateVolley( )
p1_WinsVolley /
simulateVolley( )
Player 1
served
Player 2
served
p2_WinsVolley /
simulateVolley( )
p1_WinsVolley
[ p1_Score = 20 ] /
p1AddPoint
p2_WinsVolley
[ p2_Score = 20 ] /
p2AddPoint
p1_IsWinner ? /
return TRUE
p2_IsWinner ? /
return TRUE
Player 1
won
Player 2
won
all-transitions coverage

## 7. Testing object-oriented software

- The special characteristics of the object-oriented software engineering paradigm provide some advantages but also present some new problems for testing
  - advantages: well-founded design techniques and models (UML), clean architectures and interfaces, reusable and mature software patterns
  - problems: inheritance, polymorphism, late (dynamic) binding
- There is a need for object-oriented testing process, object-oriented test case design, object-oriented coverage metrics, and object-oriented test automation



## Object-oriented code defects

- buggy interaction of individually correct superclass and subclass methods
- omitting a subclass-specific override for a high-level superclass method in a deep inheritance hierarchy
- subclass inheriting inappropriate methods from superclasses (”fat inheritance”)
- failure of a subclass to follow superclass contracts in polymorphic servers
- omitted or incorrect superclass-initialization in subclasses
- incorrect updating of superclass instance variables in subclasses
- spaghetti polymorphism resulting in loss of execution control (the ”yo-yo” problem caused by dynamic binding and *self* and *super* objects)
- subclasses violating the state model or invariant of the superclass
- instantiation of generic class with an untested type parameter
- corrupt inter-modular control relationships, due to delocalization of functionality in mosaic small-scale and encapsulated classes
- unanticipated dynamic bindings resulting from scoping nuances in multiple and repeated inheritance

## Object-oriented challenges for test automation

- low controllability, observability, and testability of the system, due to a huge number of small encapsulated objects
- difficulties in analyzing white-box coverage, due to a large number of complex and dynamic code dependencies
- construction of drivers, stubs, and test suites that conform to the inheritance structure of the system
- reuse of superclass test cases in (regression) testing of subclasses
- incomplete applications (object-oriented frameworks)



## Object-oriented challenges for testing process

- need for a larger number of testing phases, due to an iterative and incremental software development process
- unclear notion of ”module” or ”unit”, due to individual classes being too small and too much coupled with other classes
- (UML) models being too abstract to support test case design
- need for more test cases than for conventional software, due to the inherent dynamic complexity of object-oriented code
- general and partly abstract reusable classes and frameworks, making it necessary to test them also for future (unknown) applications

## 7.1. UML and software testing

UML (Unified Modeling Language): a design and modeling language for (object-oriented) software

- widely used ”de facto” standard
- provides techniques to model the software from different perspectives
- supports facilities both for abstract high-level modeling and for more detailed low-level modeling
- consists of a variety of graphical diagram types that can be extended for specific application areas
- associated with a formal textual language, *OCL* (Object Constraint Language)
- provides support for model-based testing: (1) testing of an executable UML model, (2) testing of an implementation against its UML model



## UML diagrams in software testing

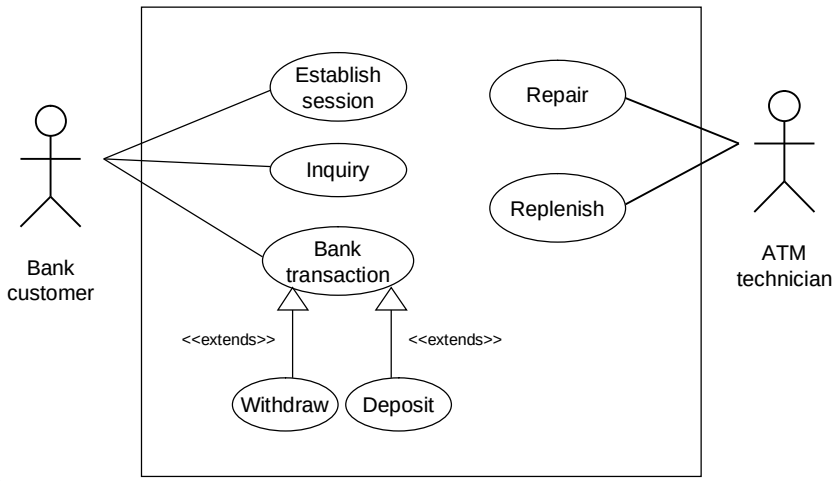
- *Use case diagrams*: testing of system-level functional requirements, acceptance testing
- *Class diagrams*: class (module / unit) testing, integration testing
- *Sequence diagrams*, *collaboration diagrams* : integration testing, testing of control and interaction between objects, testing of communication protocols between (distributed) objects
- *Activity diagrams*: testing of work flow and synchronization within the system, white-box testing of control flow
- *State diagrams* (statecharts): state-based testing
- *Package diagrams*, *component diagrams*: integration testing
- *Deployment diagrams*: system testing

## 7.2. Testing based on use cases

- *Use case*: a sequence of interactions by which the user accomplishes a task in a dialogue with the system
  - use case = one particular way to use the system
  - use case = user requirement
  - set of all use cases = complete functionality of the system
  - set of all use cases = interface between users (*actors*) and system
- *Scenario*: an instance of a use case, expressing a specific task by a specific actor at a specific time and using specific data
- UML: use case model = set of use case diagrams, each associated with a textual description of the user's task
  - for testing, the use case model must be extended with (1) the domain of each variable participating in the use cases, (2) the input/output relationships among the variables, (3) the relative frequency of the use cases, (4) the sequential (partial) order among the use cases



## Example: Automatic Teller Machine (ATM)



## Some ATM use cases and scenarios

| Use case / actor | Scenario |
|---|---|
| Establish session / Bank customer | 1) Wrong PIN entered. Display ”Reenter PIN”.<br>2) Valid PIN entered; bank not online. Display ”Try later”.<br>3) Valid PIN entered; account closed. Display ”Account closed, call your bank”. |
| Withdraw / Bank customer | 4) Requests 50 €; account open; balance 51 €. 50 € dispensed.<br>5) Requests 100 €; account closed. Display ”Account closed, call your bank”. |
| Replenish / ATM technician | 6) ATM opened; cash dispenser empty; 50000 € added.<br>7) ATM opened; cash dispenser full. |



## From use cases to test cases

1. Identification of the operational variables: explicit inputs and outputs, environmental conditions, states of the system, interface elements
   - *Establish session*: PIN on the card, PIN entered, response from the bank, status of the customer’s account
2. Domain definitions (and equivalence classes) of the variables
   - *Establish session*: PIN ∈ [ 0000 – 9999 ]
3. Development of an operational relation among the variables, modeling the distinct responses of the system as a decision table of variants
   - *variant*: combination of equivalence classes, resulting in a specific system action
   - variants shall be mutually exclusive
   - scenarios are represented by variants
4. Development of the test cases for the variants
   - at least one ”true” test case that satisfies all the variant’s requirements on variable values
   - at least one ”false” test case that violates the variant’s requirements
   - typically every ”false” test case is a ”true” test case for another variant

## *Establish session*: Operational relation

| | *Operational variables* | | | | *Expected action* | |
|---|---|---|---|---|---|---|
| **Vari-ant** | **Card PIN** | **Entered PIN** | **Bank response** | **Account status** | **ATM message** | **ATM card action** |
| 1 | Valid | Doesn't match card | – | – | Reenter PIN | None |
| 2 | Valid | Matches card | Does not acknowledge | – | Try later | Eject |
| 3 | Valid | Matches card | Acknowledges | Closed | Call bank | Eject |
| 4 | Invalid | – | – | – | Insert ATM card | Eject |
| 5 | Valid | Matches card | Acknowledges | Open | Select service | None |
| 6 | Revoked | – | Acknowledges | – | Card revoked | Retain |
| 7 | Revoked | – | Does not acknowledge | – | Invalid card | Eject |



## *Establish session*: Test cases

| **Vari-ant** | **Card PIN** | **Entered PIN** | **Bank response** | **Account status** | **ATM message** | **ATM card action** |
|---|---|---|---|---|---|---|
| 1T | 1234 | 1134 | – | – | Reenter | None |
| 1F (2T) | 1234 | 1234 | NACK | – | Try later | Eject |
| 2T | 9999 | 9999 | NACK | – | Try later | Eject |
| 2F (3T) | 9999 | 9999 | ACK | Closed | Call bank | Eject |
| 3T | 0000 | 0000 | ACK | Closed | Call bank | Eject |
| 3F (4T) | 000? | – | – | – | Insert ATM | Eject |
| 4T | %&+? | – | – | – | Insert ATM | Eject |
| 4F (5T) | 0001 | 0001 | ACK | Open | Select | None |
| 5T | 3210 | 3210 | ACK | Open | Select | None |
| 5F (2T) | 0001 | 0001 | NACK | – | Try later | Eject |
| 6T | 5555 | – | ACK | – | Revoked | Retain |
| 6F (1T) | 9998 | 9999 | – | – | Reenter | None |
| 7T | 5555 | – | NACK | – | Invalid card | Eject |
| 7F (1T) | 9999 | 0000 | – | – | Reenter | None |

## 7.3. Class-based testing

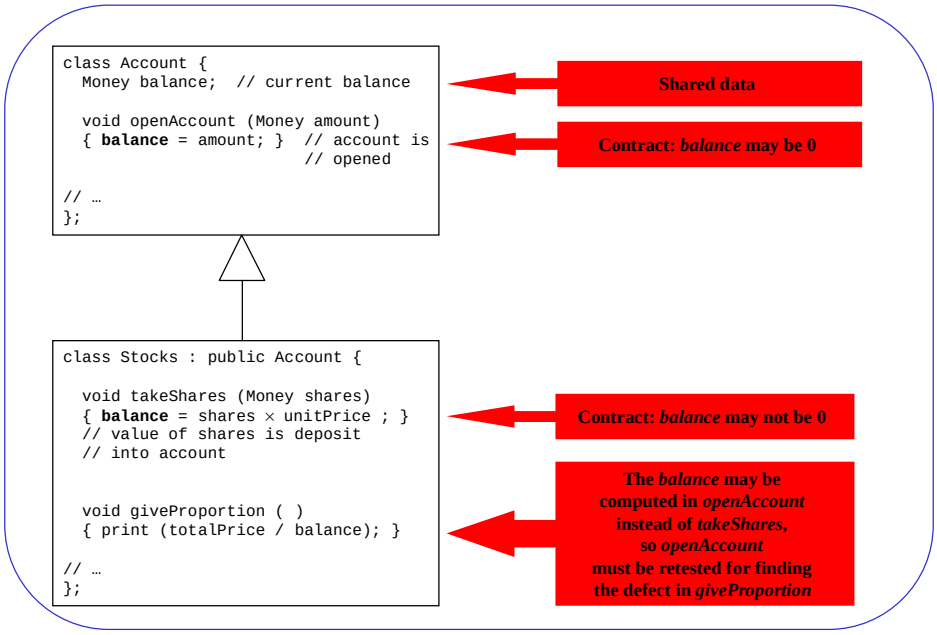
Some questions when designing the testing of subclasses:

- Should we test inherited methods?
  - *extension*: inclusion of superclass features in a subclass, inheriting method implementation and interface (name and arguments)
  - *overriding*: new implementation of a subclass method, with the same inherited interface as that of a superclass method
  - *specialization*: definition of subclass-specific methods and instance variables, not inherited from a superclass
- Can we reuse superclass tests for extended and overridden methods?
- To what extent should we exercise interaction among methods of all superclasses and of the subclass under test?



## Superclass-subclass development scenarios

1. *Superclass modifications*: a method is changed in superclass
   - the changed method and all its interactions with changed and unchanged methods must be retested in the superclass
   - the method must be retested in all the subclasses inheriting the method as extension
   - the ”super” references to the method must be retested in subclasses
2. *Subclass modifications*: a subclass is added or modified, without changing the superclasses
   - all the methods inherited from a superclass must be retested in the context of the subclass, even those which were just included by extension but not overridden






## Superclass-subclass development scenarios

3. *Reuse of the superclass test suite*: a method is added to a subclass, overriding a superclass method
   - the new implementation of the method must be tested
   - test cases for the superclass method can be reused, but additional test cases are also needed for subclass-specific behavior

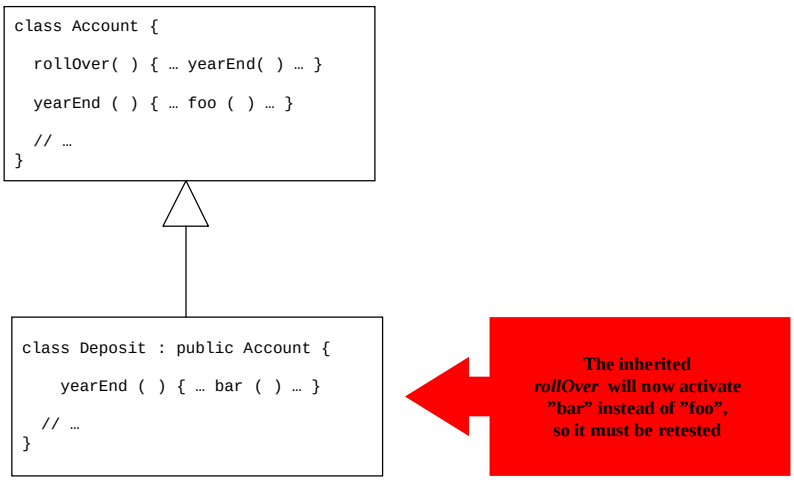
4. *Addition of a subclass method*: an entirely new method is added to a specialized subclass
   - the new method must be tested with method-specific test cases
   - interactions between the new method and the old methods must be tested with new test cases

## Superclass-subclass development scenarios

5. *Change to an abstract superclass interface*
   - ”abstract” superclass: some of the methods have been left without implementation; just the interface has been defined
   - all the subclasses must be retested, even if they have not changed

6. *Overriding a superclass method used by another superclass method*
   - the superclass method that is using the overridden method will behave differently, even though it has not been changed itself
   - so, the superclass method must be retested





## 8. Integration testing

*Integration testing*: search for module faults that cause failures in interoperability of the modules

– ”*module*”: generic term for a program element that implements some restricted, cohesive functionality

– typical modules: class, component, package

– ”*interoperability*”: interaction between different modules

– interaction may fail even when each individual module works perfectly by itself and has been module-tested

– usually related to the call interface between modules: a function call or its parameters are buggy



## Typical interface bugs

- missing, overlapping, or conflicting functions
- incorrect or inconsistent data structure used for a file or database
- violation of the data integrity of global store or database
- unexpected runtime binding of a method
- client sending a message that violates the server’s constraints
- wrong polymorphic object bound to message
- incorrect parameter value
- attempt to allocate too much resources from the target, making the target crash
- incorrect usage of virtual machine, object request broker, or operating system service
- incompatible module versions or inconsistent configuration

Interaction ***dependencies*** between modules:

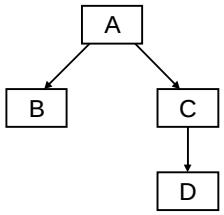
- function calls (the most usual case)
- remote procedure calls
- communication through global data or persistent storage
- client-server architecture
- composition and aggregation
- inheritance
- calls to an application programming interface (API)
- objects used as message parameters
- proxies
- pointers to objects
- dynamic binding



***Dependency tree***: hierarchical representation of the dependencies between modules

In integration testing: ***uses***-dependency

- usually: module *A* uses module *B* =
  (some function in) module *A* calls (some function in) module *B*
- can be generated by automated static dependency analyzers



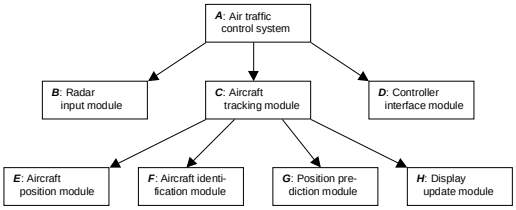

*A* uses (calls) *B*

*A* uses (calls) *C*

*C* uses (calls) *D*

## Integration testing strategies

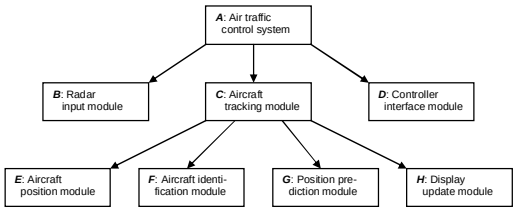
(1) *Big-bang integration*: all the modules are tested at the same time




– One test configuration: {A, B, C, D, E, F, G, H}
– Failure => where is the bug?
– OK for small and well-structured systems
– OK for systems constructed from trusted components



(2) *Top-down integration*: the system is tested incrementally, level by level with respect to the dependency tree, starting from the top level (root of the tree)




1. The main module *A* is tested by itself. ***Stubs***: *B*, *C*, and *D*.
2. The subsystem {*A*, *B*, *C*, *D*} is tested. ***Stubs***: *E*, *F*, *G*, and *H*.
3. Finally, the entire system is tested. No stubs.

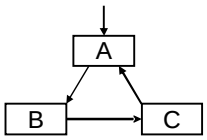
– Failure in step 3: the bug is in *E*, *F*, G, or *H*

**Advantages**:

– in case of failure, the suspect area is limited
– testing may begin early: when the top-level modules have been coded
– early validation of main functionality
– modules may be developed in parallel with testing

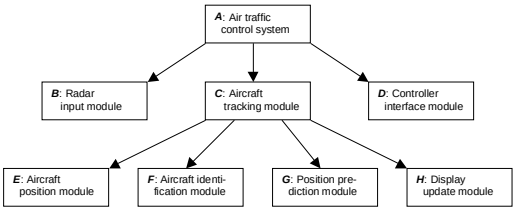
**Disadvantages**:

– lowest-level (most often used) modules are tested last, so performance problems are encountered late
– requires *stubs*: partial proxy implementations of called modules
– since stubs must provide some degree of real functionality, it may be necessary to have a set of test case specific stubs for each module
– dependency cycles must be resolved by testing the whole cycle as a group or with a special cycle-breaking stub






(3) *Bottom-up integration*: the system is tested incrementally, starting from the bottom level (leaves of the dependency tree)




1. The module *E* is tested alone. ***Driver***: *C*. [The driver may have been developed already in module testing of *E*.]
2. The module *F* is tested. (Extended) driver: *C*.
3. The module *G* is tested. (Extended) driver: *C*.
4. The module *H* is tested. (Extended) driver: *C*.
5. The module *B* is tested. Driver: *A*.
6. The module *D* is tested. (Extended) driver: *A*.
7. The modules {*C, E, F, G, H*} are tested together. (Extended) driver: *A*.
   Failure => bug in *C* or its downwards-*uses* interfaces.
8. Finally, the entire system is tested. No drivers.
   Failure => bug in *A* or its *uses* interfaces.

**Advantages**:

– in case of failure, the suspect area is very narrow: one module and its interfaces
– testing may begin early: as soon as any leaf-level module is ready
– initially, testing may proceed in parallel
– early validation of performance-critical modules

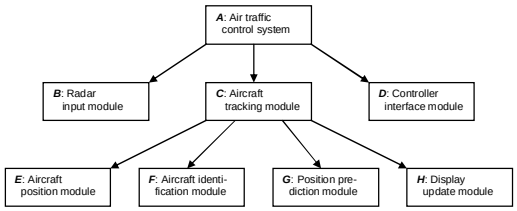
**Disadvantages**:

– main functionality (usability) and control interface of the system are validated late
– requires *drivers*: partial proxy implementations of calling modules
– dependency cycles must be resolved
– requires many tests, especially if the dependency tree is broad and has a large number of leaves



(4) *Backbone integration*: combination of big-bang, top-down, and bottom-up

1. The *backbone* (kernel) modules of the system are tested first, bottom-up.
2. The system control is tested next, top-down.
3. The backbone modules are big-banged together, bottom-up.
4. Top-down integration is continued, until the backbone has been included.




Backbone:
*E, F, G, H*

1. Each backbone module *E*, *F*, *G*, *H* is tested alone, in isolation. Drivers are needed.
2. The control subsystem {*A*} is tested. Stubs are needed.
3. The backbone and its controller {*E*, *F*, *G*, *H*, *C*} are tested. Driver for *C* is needed.
4. The subsystem {*A*, *B*, *C*, *D*} is tested. Stubs are needed.
5. The entire system is tested. No stubs or drivers.

***Usually bottom-up preferable*:**

+ Drivers are much easier to write than stubs, and can even be automatically generated.
+ The approach provides a greater opportunity for parallelism than the other approaches; that is, there can be several teams testing different subsystems at the same time.
+ The approach is effective for detecting detailed design or coding errors early enough.
+ The approach detects critical performance flaws that are generally associated with low-level modules.
+ The approach supports the modern software engineering paradigms based on classes, objects, and reusable stand-alone components.
– A prototype of the system is not available for (user) testing until at the very end.
– The approach is not effective in detecting architectural design flaws of large scale.
– May be too laborious for large systems.



## 9. Regression testing

*ANSI/IEEE Standard of Software Engineering Terminology:*

> *selective testing of a system or component to verify that **modifications** have not caused unintended effects and that the system or component still complies with its specifications*

Usually integrated with maintenance, to check the validity of the modifications:

§ *corrective maintenance* (fixes)

§ *adaptive maintenance* (porting to a new operational environment)

§ *perfective maintenance* (enhancements and improvements to the functionality)



**Differences between "ordinary" testing and regression testing:**

§ regression testing uses a (possibly) *modified* specification, a *modified* implementation, and an old test plan (to be updated)

§ regression testing checks the correctness of some *parts* of the implementation only

§ regression testing is usually not included in the total cost and schedule of the development

§ regression testing is done many times in the life-cycle of the system (during bug fixing and maintenance)

## General approach:

– *P*: program (module, component), already tested
– *P'*: modified version of *P*
– *T*: test suite used for testing *P*

1. Select $T' \subseteq T$, a set of tests to execute on *P'*.
2. Test *P'* with *T'*, establishing the correctness of *P'* with respect to *T'*.
3. If necessary, create *T''*, a set of new functional (black-box) or structural (white-box) tests for *P'*.
4. Test *P'* with *T''*, establishing the correctness of *P'* with respect to *T''*.
5. Create *T'''*, a new test suite and test history for *P'*, from *T*, *T'*, and *T''*: $T''' = T \cup T''$



§ naïve regression testing (re-running *all* the existing test cases) not cost-effective, although the common strategy in practice

§ principle 1: it is useless to test *unmodified* parts of the software again

§ principle 2: *modified* parts shall be tested with *existing* test cases (may not be possible, if the interface has been changed: *GUI capture-replay* problem!)

§ principle 3: *new* parts shall be tested with *new* test cases

§ database of test cases needed

§ additional automation support possible (data-flow analysis)

## When and how?

- *A new subclass has been developed*: rerun the superclass tests on the subclass, run new tests for the subclass
- *A superclass is changed*: rerun the superclass tests on the superclass and on its subclasses, rerun the subclass tests, test the superclass changes
- *A server* (*class*) *is changed*: rerun tests for the clients of the server, test the server changes
- *A bug has been fixed*: rerun the test that revealed the bug, rerun tests on any parts of the system that depend on the changed code
- *A new system build has been generated*: rerun the build test suite
- *The final release has been generated*: rerun the entire system test suite



## Selective regression test strategies

- *Risk-based heuristics*: rerun tests for (1) unstable, (2) complex, (3) functionally critical, or (4) frequently modified modules (classes, functions, and such)
- *Profile-based heuristics*: rerun tests for those use cases, properties, or functions that are the most frequently used
- *Coverage-based heuristics*: rerun those tests that yield the highest white-box (statement, branch, …) code coverage
- *Reachability-based heuristics*: rerun those tests that reach an explicitly or implicitly changed or deleted module
- *Dataflow-based heuristics*: rerun those tests that exercise modified or new definition-use pairs
- *Slice-based heuristics*: rerun those tests that generate a similar data-flow slice over the old and the new software version

## 10. Statistical testing

§ ***operational profile***: distribution of functions *actually used*

⇒ probability distribution of inputs

§ most ***frequently*** used functions / properties tested more carefully, with a larger number of test cases

§ more ***complex***, error-prone functions tested more carefully

§ ***central*** "kernel" functions tested more carefully

§ useful strategy, when in lack of time or testing resources

§ based on experience with and statistics over previous use

§ history data over existing systems must be available



## Example: file processing

– *create*: probability of use 0.5 (50 %)
– *delete*: probability of use 0.25 (25 %)
– *modify*: probability of use 0.25 (25 %)

ð *create*: 50 test cases
ð *delete*: 25 test cases
ð *modify*: 25 test cases

} In total: 100

## Improved strategy: relative probability of failure

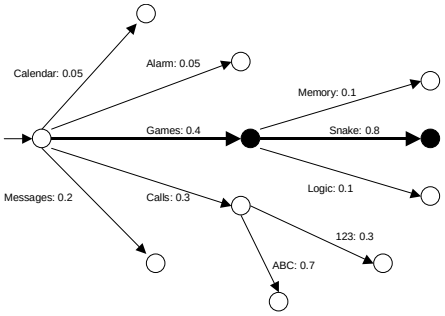
– *modify* twice as complex as *create* and *delete*

- ð *create*: 40 test cases
- ð *delete*: 20 test cases
- ð *modify*: 40 test cases

} In total: 100



**Profile of a mobile phone as a graph**




*Complexity ?!?*

– *Memory*: 0.4 ∗ 0.1 = 4 %
– *Snake*: 0.4 ∗ 0.8 = 32 %
– *Logic*: 0.4 ∗ 0.1 = 4 %
– *Calls*: 30 %

} *Games*: 40 %

## 11. Practical aspects of testing

**When to stop?**

§ all the ***planned*** test cases have been executed

§ required (white-box) ***coverage*** has been reached (e.g. all the branches have been tested)

§ all the (black-box) operations, their parameters and their ***equivalence classes*** have been tested

§ required percentage (e.g., 95%) of ***estimated*** total number of errors has been found (known from company's project history)

§ required percentage (e.g., 95%) of ***seeded*** errors has been found

§ ***mean time to failure*** (in full operation) is greater than a required threshold time (e.g. 1 week)



**Frequently made mistakes in software testing**

§ Most organizations believe that the purpose of testing is just to find bugs, while the key is to find the *important* bugs.

§ Test managers are reporting bug data without putting it into context.

§ Testing is started too late in the project.

§ Installation procedures are not tested.

§ Organizations overrely on beta testing done by the customers.

§ One often fails to correctly identify risky areas (areas that are used by more customers or would be particularly severe if not functioning perfectly).

§ Testing is a transitional job for new, novice programmers.

§ Testers are recruited from the ranks of failed programmers.

§ Testers are not domain experts.



§ The developers and testers are physically separated.

§ Programmers are not supposed to test their own code.

§ More attention is paid to running tests than to designing them.

§ Test designs are not reviewed / inspected.

§ It is checked that the product does what it's supposed to do, but not that it *doesn't* do what is *isn't* supposed to do.

§ Testing is only conducted through the user-visible interface.

§ Bug reporting is poor.

§ Only existing test cases (no new ones) are applied in regression testing of system modifications.

§ All the tests are automated, without considering economic issues.

§ The organization does not learn, but makes the same testing mistakes over and over again.



## Empirical hints for defect removal

B. Boehm, V.R. Basili: Software Defect Reduction Top 10 List. (*IEEE*) *Computer*, 34, 1 (January), 2001, 135-137.

1. Finding and fixing a software problem after delivery is often 100 times more expensive than finding and fixing it during the ***requirements and design phase***.
2. Current software projects spend about 40 to 50 percent of their effort on ***avoidable rework***.
3. About 80 percent of avoidable rework comes from 20 percent of the defects.
4. About 80 percent of the defects come from 20 percent of the modules, and about half of the modules are defect free.
5. About 90 percent of the downtime comes from, at most, 10 percent of the defects.

6. Peer ***reviews*** catch 60 percent of the defects.
7. Perspective-based reviews catch 35 percent more defects than nondirected reviews.
8. Disciplined ***personal practices*** can reduce defect introduction rates by up to 75 percent.
9. All other things being equal, it costs 50 percent more per source instruction to develop ***high-dependability software*** products than to develop low-dependability software products. However, the investment is more than worth it if the project involves significant operations and maintenance costs.
10. About 40 to 50 percent of user programs contain nontrivial defects.

***Conclusion***: *Try to avoid testing as much as possible. However, if you have to test, do it carefully and with focus.*



## ***Observations***

- Testing is not considered as a profession, but rather as an art.
  - Psychologically, testing is more "boring" than coding
- Too little resources are allocated to testing.
  - Exception: Microsoft has about as many testers as programmers
- Quality of testing is not measured.
  - A general test plan is usually written, but the process is not tracked
- The significance of testing tools is overestimated.
- The significance of inspections is underestimated.