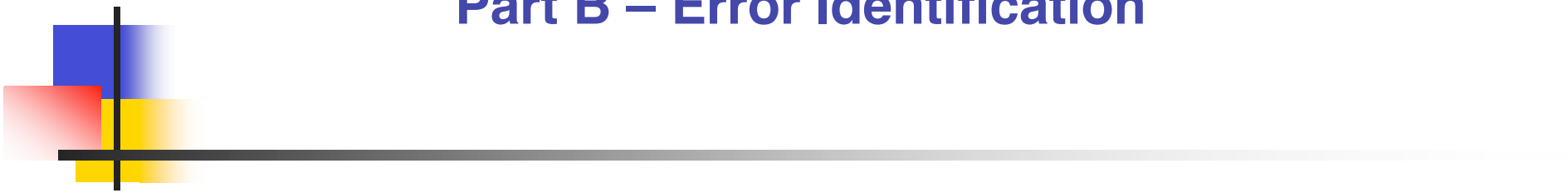
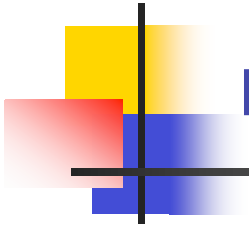


State-Based Testing
Part B – Error Identification

Generating test cases for complex behaviour

Reference: Robert V. Binder
Testing Object-Oriented Systems: Models, Patterns, and Tools
Addison-Wesley, 2000, Chapter 7



Flattening the statechart

- Statecharts are great for communication, reducing clutter etc.
- They might hide subtle bugs
  - **e.g. entering a sub-state rather than a super-state**
- We need to expand them to full transition diagrams for testing purposes
  - **Expansion makes implicit transitions explicit, so they are not lost**
  - **Expansion is a flat view**
    - **Includes everything from inheritance in OO and sub-states in statecharts**
- An automatable process

Concurrent statechart

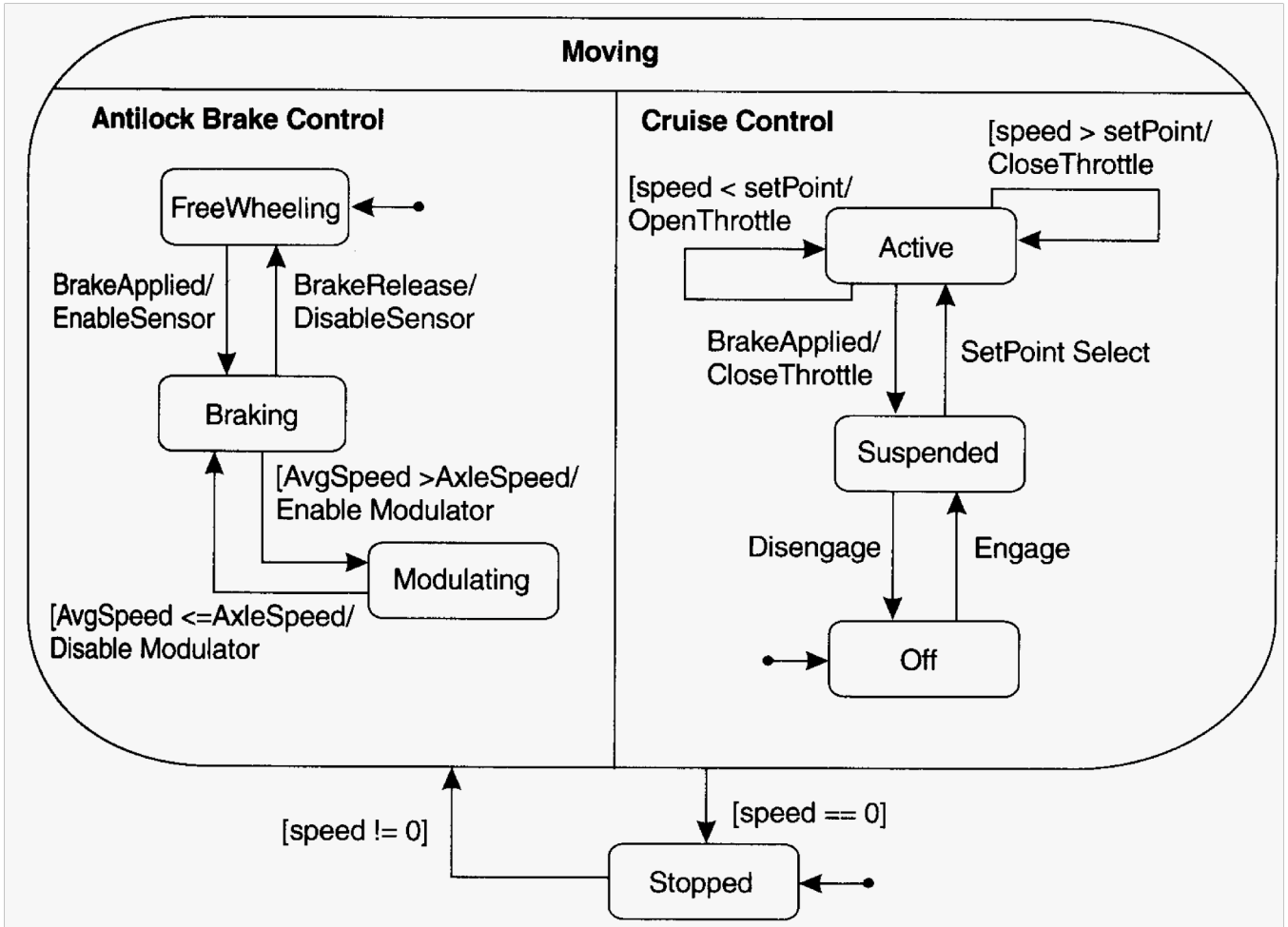
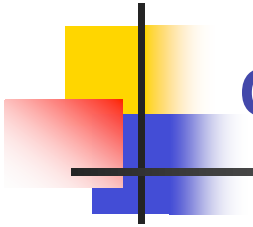


FIGURE 7.14 Automotive control statechart with orthogonal states.



Concurrency Hides Problems

- Concurrency hides implicit state combinations
  - **Hides potential serious defects**
  - **Arise from implicit state combinations**
- Explicit violations of implicit prohibitions should be tested

Expanding the Example

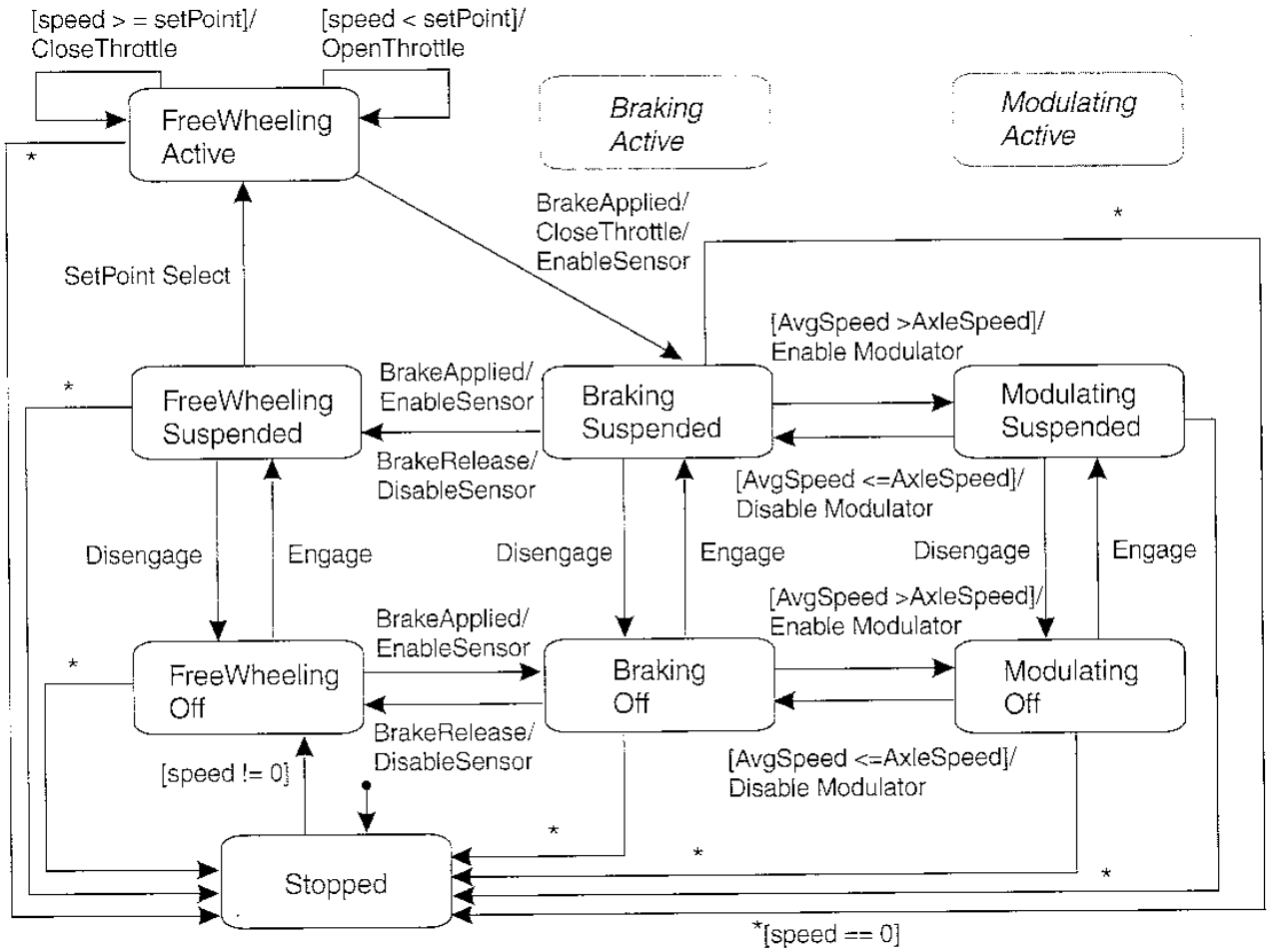
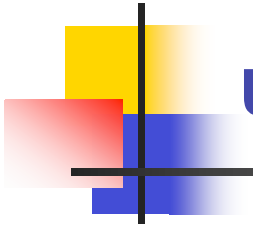


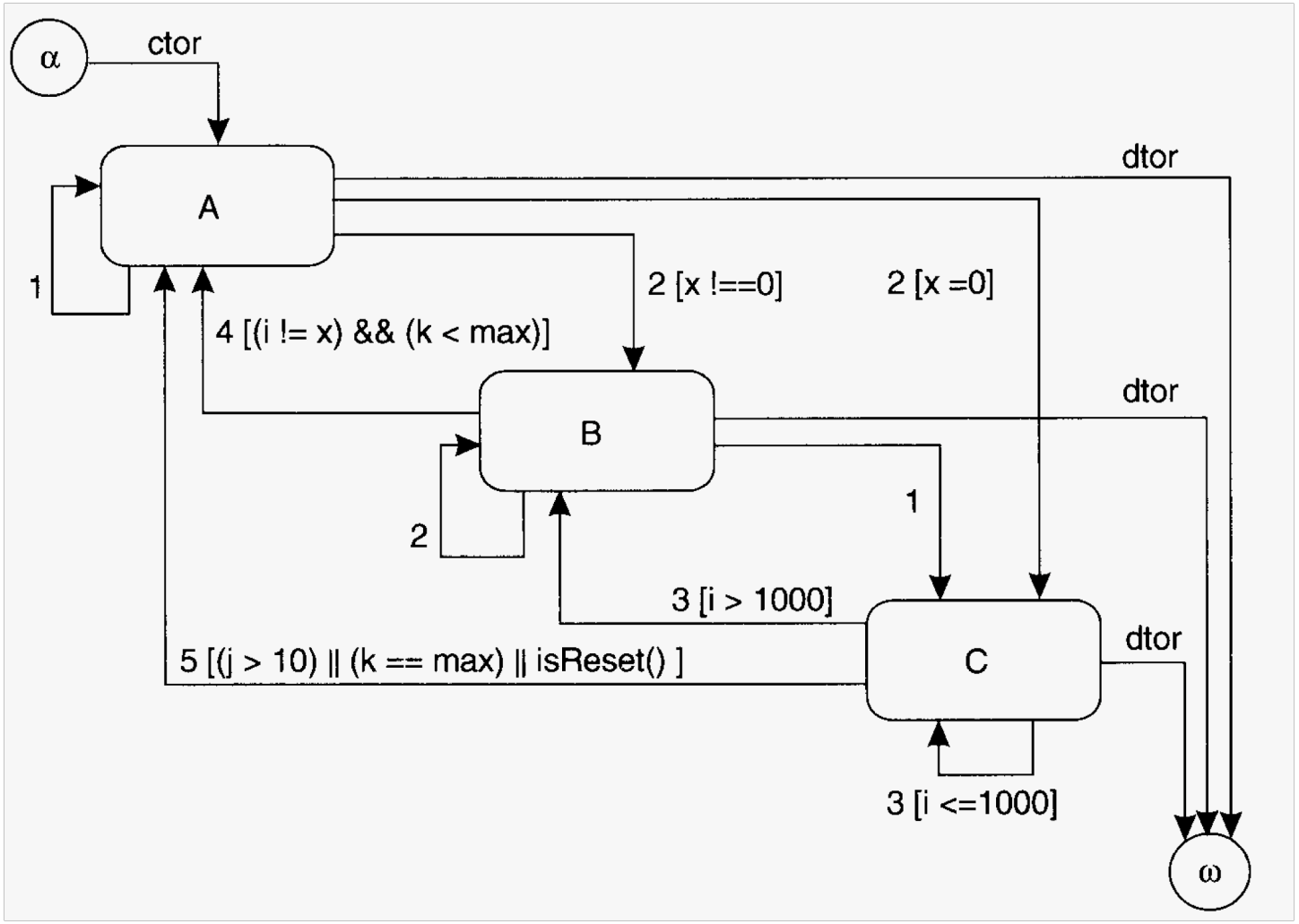
FIGURE 7.15 Automotive control state transition diagram.



Unspecified Event/State Pairs

- State machine models will not include all events for all states
- Implicit transitions may be illegal, ignored, or a specification omission
- Accepted illegal events lead to bugs called **sneak paths**
- For testing purposes, we cannot ignore implicit behaviour
  - **Develop a Response Matrix**

Example statechart

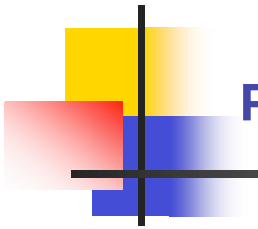


Response matrix

Events and Guards				Accepting State/Expected Response					
				α	A	B	. . .	C	
ctor				✓	6	6	6	6	6
Event 1				☒	✓	✓	. . .	2	6
Event 2	x == 0								
	DC			☒	☒	✓	. . .	2	6
	F			☒	✓	☒	. . .	☒	☒
	T			☒	✓	☒	. . .	☒	☒
Event 3	i <= 1000								
	DC			☒	2	2	. . .	☒	6
	Off			☒	2	2	. . .	✓	☒
	On			☒	2	2	. . .	✓	☒
Event 4	i != x	k < max							
	DC	DC		☒	2	☒	. . .	2	6
	F	F		☒	☒	1	. . .	☒	☒
	F	T		☒	☒	2	. . .	☒	☒
	T	F		☒	☒	2	. . .	☒	☒
	T	T		☒	☒	✓	. . .	☒	☒
Event 5	i > 10	k == max	isReset()						
	DC	DC	DC	☒	2	5	. . .	☒	6
	F	F	F	☒	☒	☒	. . .	5	☒
	F	F	T	☒	☒	☒	. . .	✓	☒
	F	T	F	☒	☒	☒	. . .	✓	☒
	F	T	T	☒	☒	☒	. . .	✓	☒
	T	F	F	☒	☒	☒	. . .	✓	☒
	T	F	T	☒	☒	☒	. . .	✓	☒
	T	T	F	☒	☒	☒	. . .	✓	☒
	T	T	T	☒	☒	☒	. . .	✓	☒
dtor				☒	✓	✓	. . .	✓	2

Key: T = true, F = false, DC = don't care; for Action codes, see Table 7.3

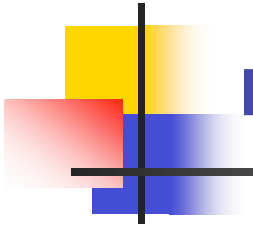
☐ Not applicable ☒ Excluded ☑ Explicitly specified transition



Possible responses to illegal events

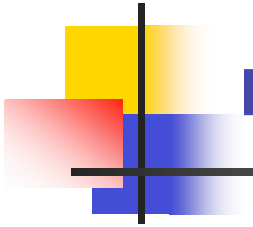
TABLE 7.3 Response Codes for Illegal Events

Response Code	Name	Response
0	Accept	Perform the explicitly specified transition
1	Queue	Place the illegal event in a queue for subsequent evaluation and ignore
2	Ignore	No action or state change is to be produced, no error is returned, no exception raised
3	Flag	Return a nonzero error code
4	Reject	Raise an `IllegalEventException`
5	Mute	Disable the source of the event and ignore
6	Abend	Invoke abnormal termination services (e.g., core dump) and halt the process



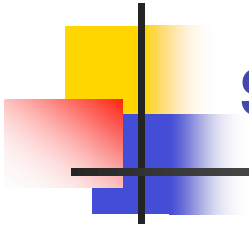
Designing responses to illegal events

- Abstract state should not change
  - **Concrete state may change due to exception handling**
- Illegal event design question
  - **Handle with defensive programming**
    - **Defensive systems**
  - **Handle with precondition contracts**
    - **Cooperative systems**



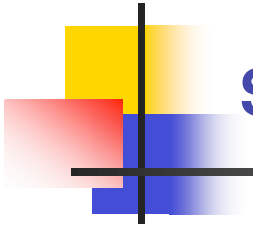
Designing responses to illegal events – 2

- Possible responses
  - **Raise exception**
  - **Treat message as a noop**
  - **Attempt error recovery**
  - **Invoke abnormal termination**
- Tester needs to decide expected responses so actual responses can be evaluated



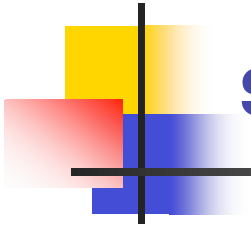
State model validation

- A state model must be complete, consistent, and correct before it is used to generate test cases
- We will look at four validation checklists
  - **Structure checklist**
  - **State name checklist**
  - **Guarded transition checklist**
  - **Well-formed subclass behaviour checklist**
  - **Robustness checklist**



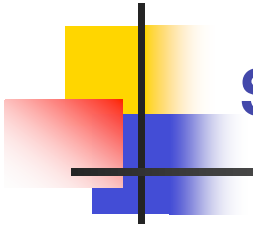
Structure checklist

- There is an initial state with only outbound transitions
- There is a final state with only inbound transitions (if not, explicit reason is needed)
- No equivalent states
- Every state is reachable from the initial state
- The final state is reachable from all states
- Every defined event and every defined action appears in at least one transition



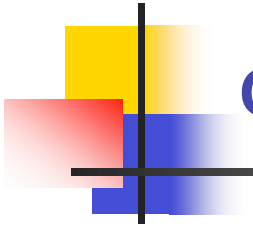
Structure checklist

- Except for the initial and final states, every state has at least one incoming and one outgoing transition
- The events accepted in a particular state are unique or differentiated by mutually exclusive guards
- Complete specification: For every state, every event is accepted or rejected (either explicitly or implicitly)



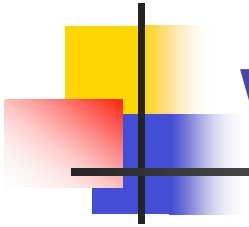
State name checklist

- Poor names are often indications of incomplete or incorrect design
- Names must be meaningful in the context of the application
- If a state is not necessary, leave it out
  - **“Wait states” are often superfluous**
- State names should be passive
- Adjectives are best, past participles are OK



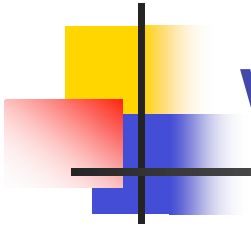
Guarded transition checklist

- The entire range of truth values for a particular event is covered
- Each guard is mutually exclusive of all other guards
- Guard variables are visible
- Guards with three or more variables are modeled with a decision table
- The evaluation of a guard does not cause side effects



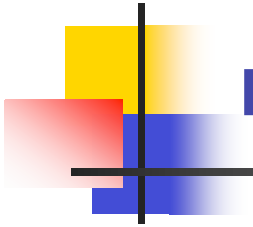
Well-Formed Subclass Behaviour Checklist

- Does not remove any superclass states
  - **All transitions accepted in the superclass are accepted in the subclass**
- Subclass does not weaken the state invariant of the superclass
- Subclass may add an orthogonal state defined with respect to its locally introduced instance variables
- All guards on superclass transitions are the same or weaker for subclass transitions



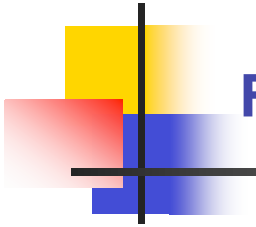
Well-Formed Subclass Behaviour Checklist – 2

- All inherited actions are consistent with the subclass's responsibilities
  - **Verify name-scope sensitive or dynamic binding of intraclass messages is correct**
- All inherited accessor events are appropriate in the context of the subclass
- Messages sent to objects that are variables in a guard expression do not have side effects on the class under test



Robustness checklist

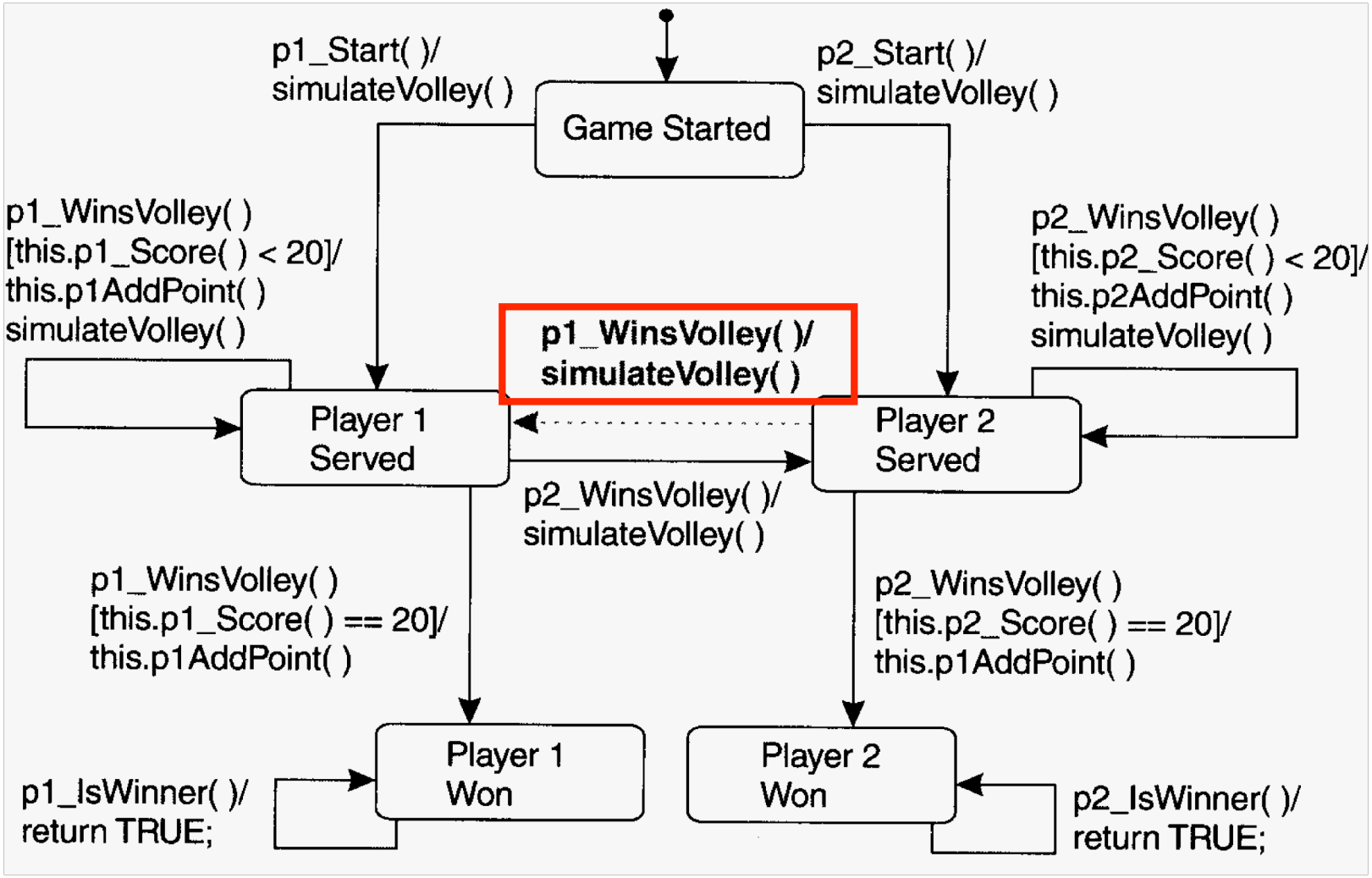
- There is an explicit spec for an error-handling or exception-handling mechanism for implicitly rejected events
- Illegal events do not corrupt the machine (preserve the last good state, reset to a valid state, or self-destruct safely)
- Actions have no side effects on the resultant state
- Explicit exception, error logging, and recovery mechanisms are specified for contract violations

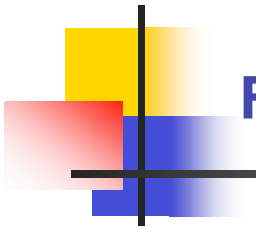


Fault model for state machines

- Control faults: An incorrect sequence of events is accepted, or an incorrect sequence of outputs is produced
  - **Missing transition**
    - **Implementation does not respond to a valid event-state pair**
    - **Resultant state is incorrect but not corrupt**

Missing transition

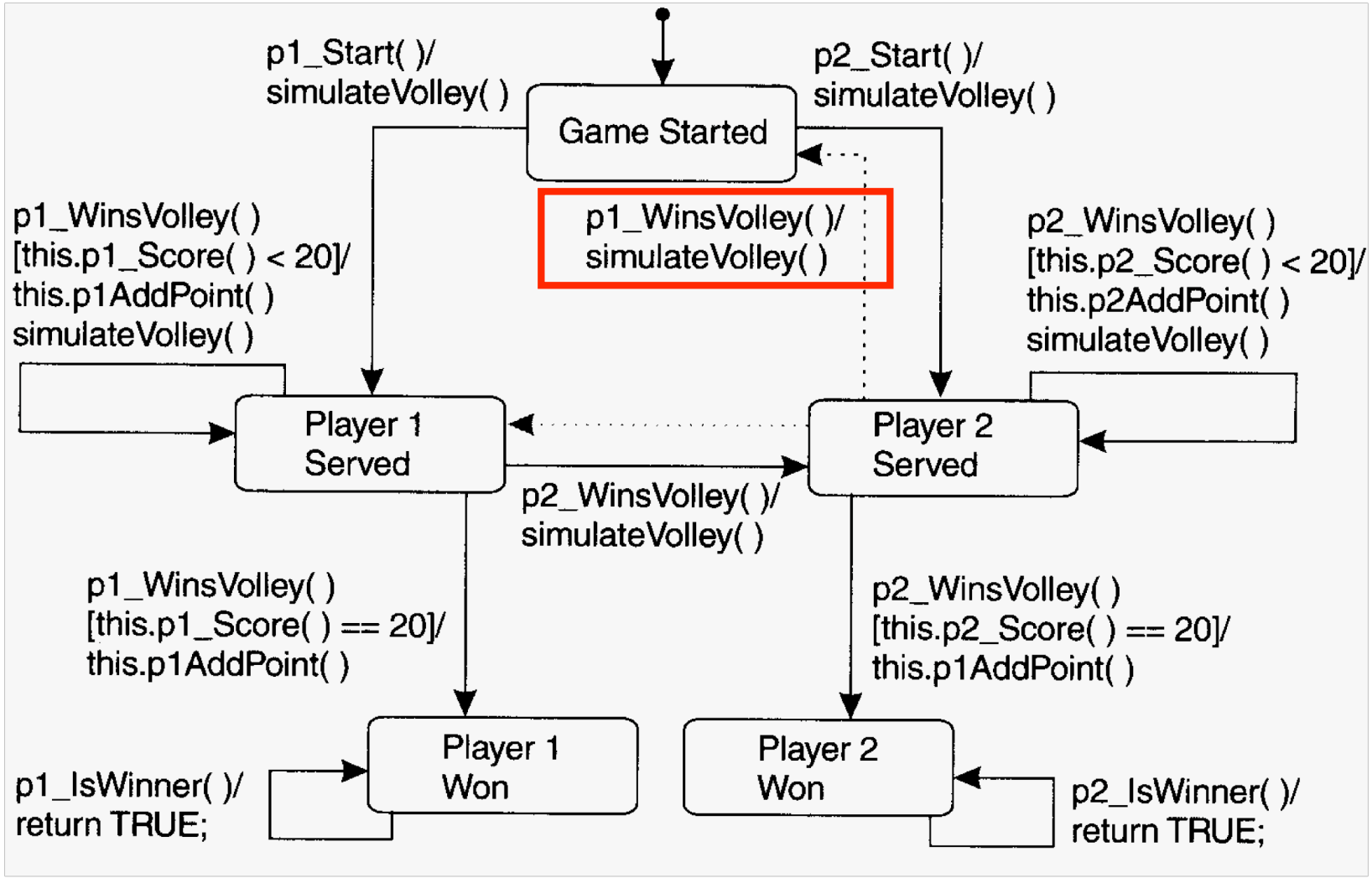


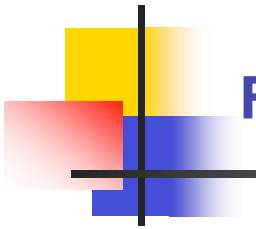


Fault model for state machines – 2

- Incorrect transition
  - Implementation behaves as if an incorrect resultant state has been reached
  - Resultant state is incorrect but not corrupt

Incorrect transition

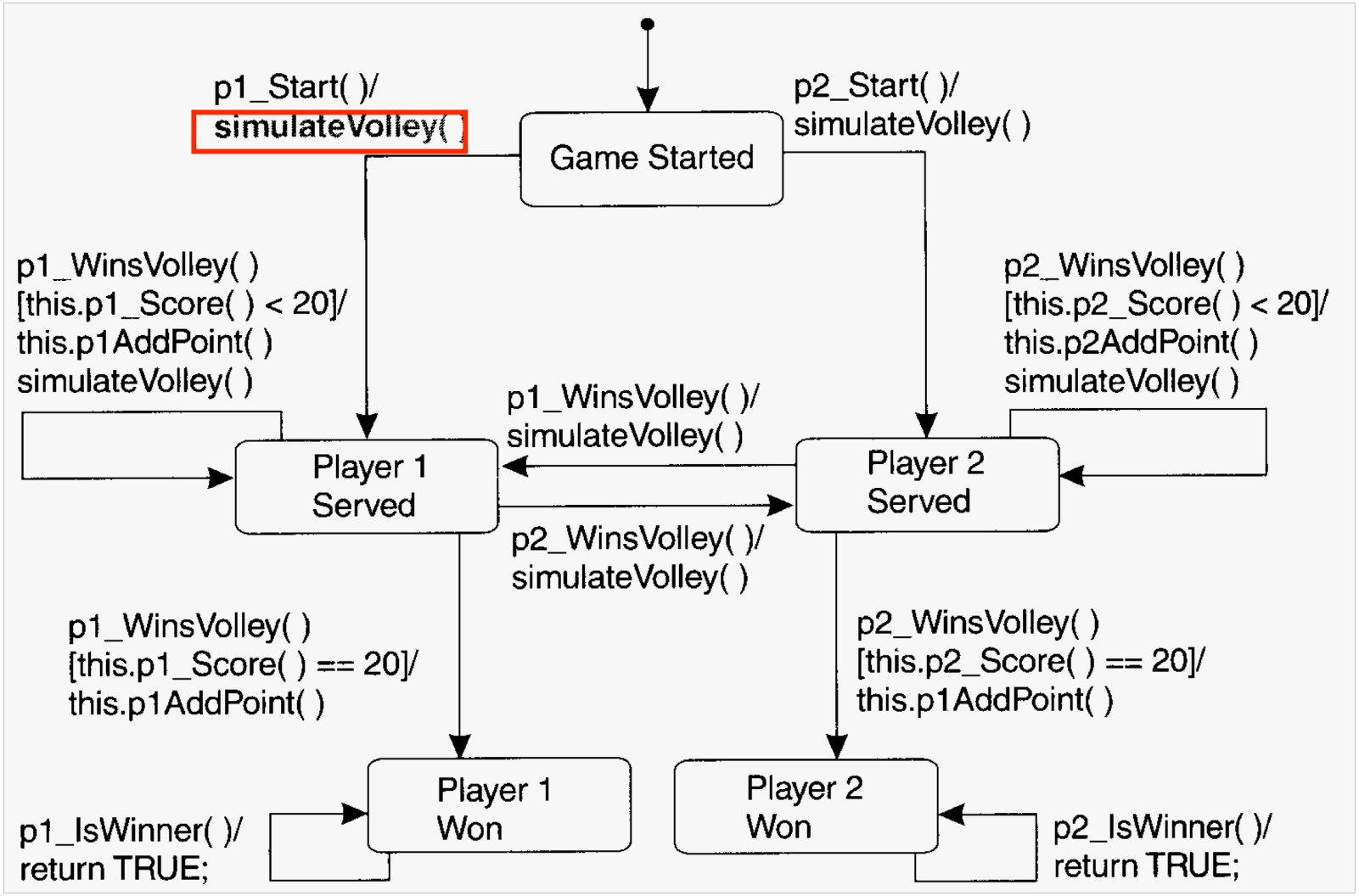


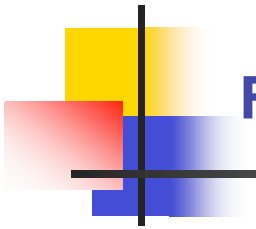


Fault model for state machines – 3

- Missing action
  - Implementation does not produce any action for a transition

Missing action

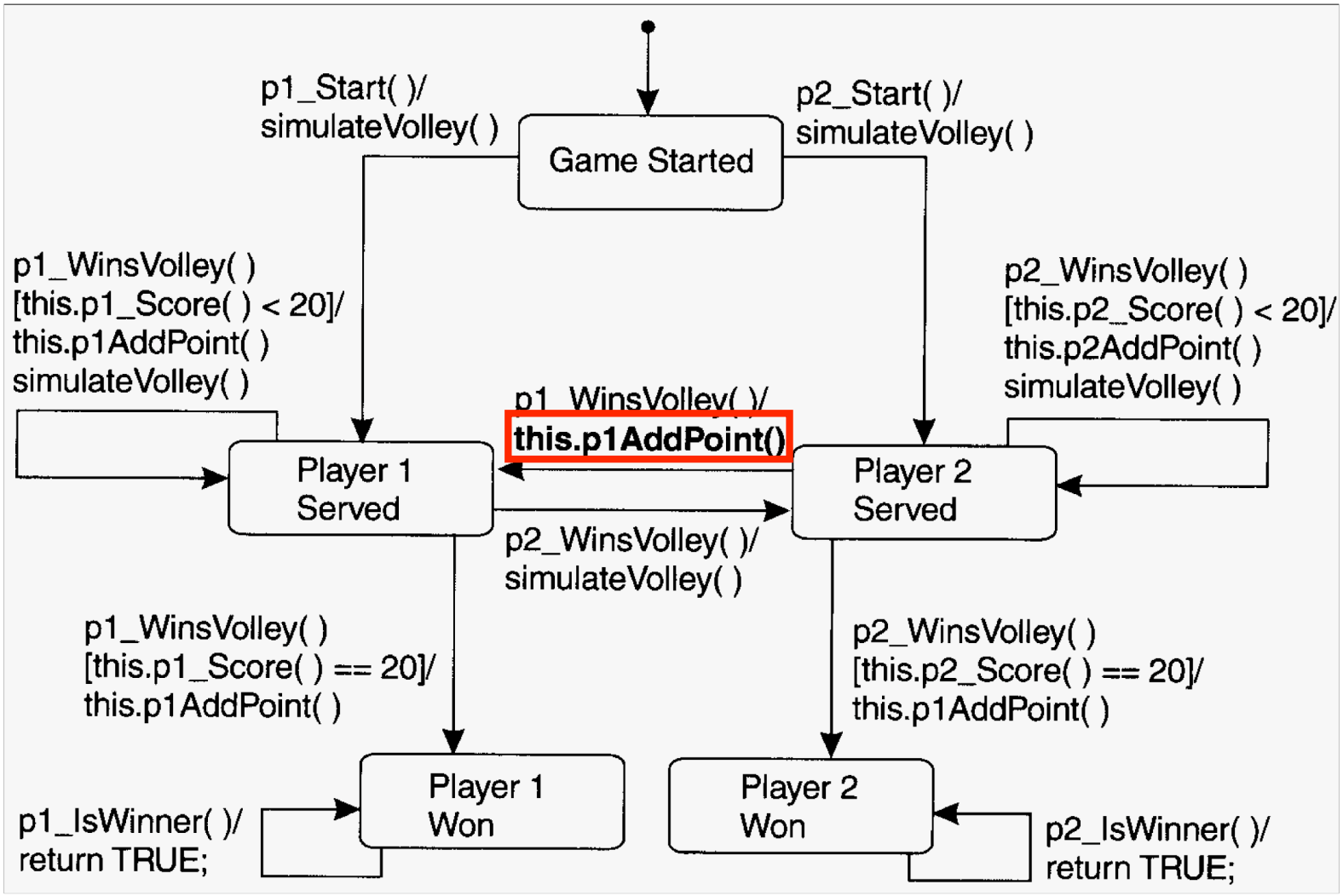


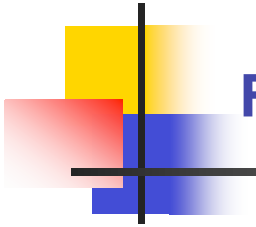


Fault model for state machines – 4

- Incorrect action
  - Implementation produces the wrong action for a transition

Incorrect action

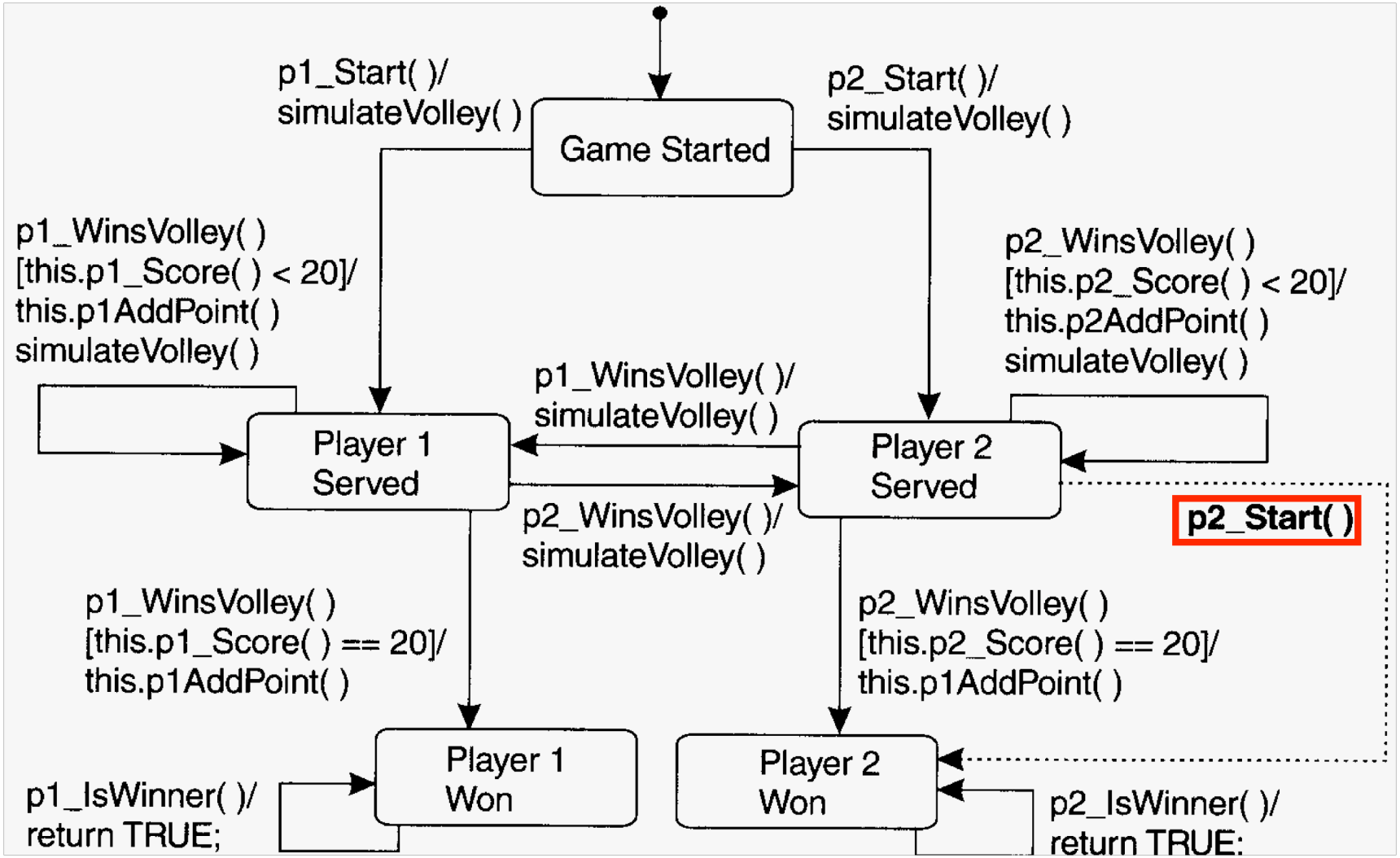


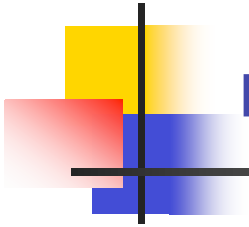


Fault model for state machines – 5

- Sneak path
  - Implementation accepts an event that is illegal or unspecified for a state

Sneak path

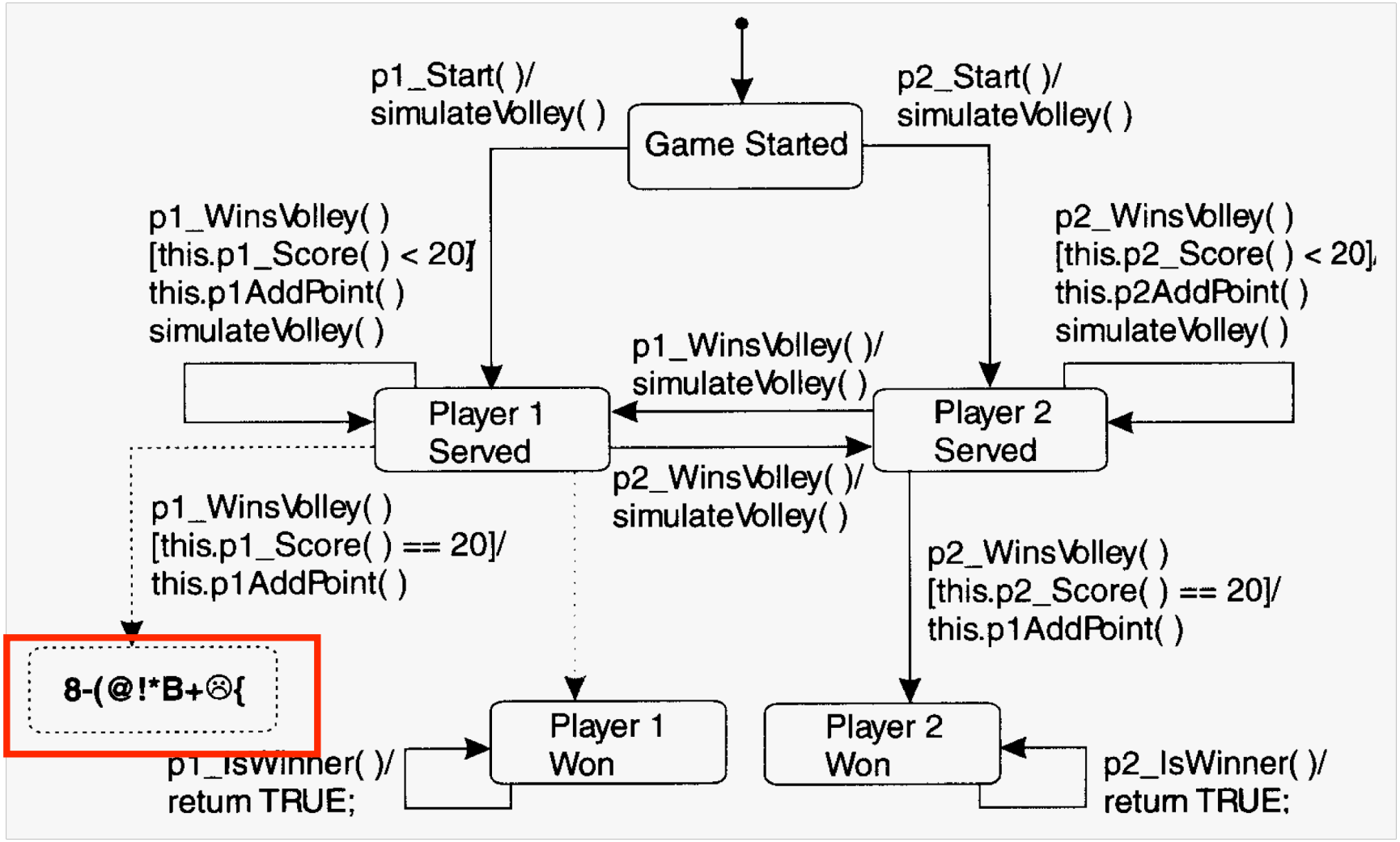

p1_Start( )/
simulateVolley( )
Game Started
p2_Start( )/
simulateVolley( )
p1_WinsVolley( )
[this.p1_Score( ) < 20]/
this.p1AddPoint( )
simulateVolley( )
p2_WinsVolley( )
[this.p2_Score( ) < 20]/
this.p2AddPoint( )
simulateVolley( )
p1_WinsVolley( )/
simulateVolley( )
Player 1
Served
Player 2
Served
p2_WinsVolley( )/
simulateVolley( )
p2_Start( )
p1_WinsVolley( )
[this.p1_Score( ) == 20]/
this.p1AddPoint( )
p2_WinsVolley( )
[this.p2_Score( ) == 20]/
this.p1AddPoint( )
Player 1
Won
Player 2
Won
p1_IsWinner( )/
return TRUE;
p2_IsWinner( )/
return TRUE:

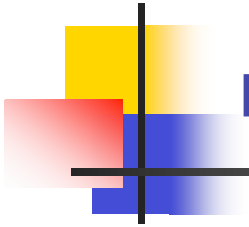


Fault model for state machines – 6

- Corrupt state
  - Implementation computes a state that is not valid
  - Either the class invariant of state invariant is violated

Corrupt state

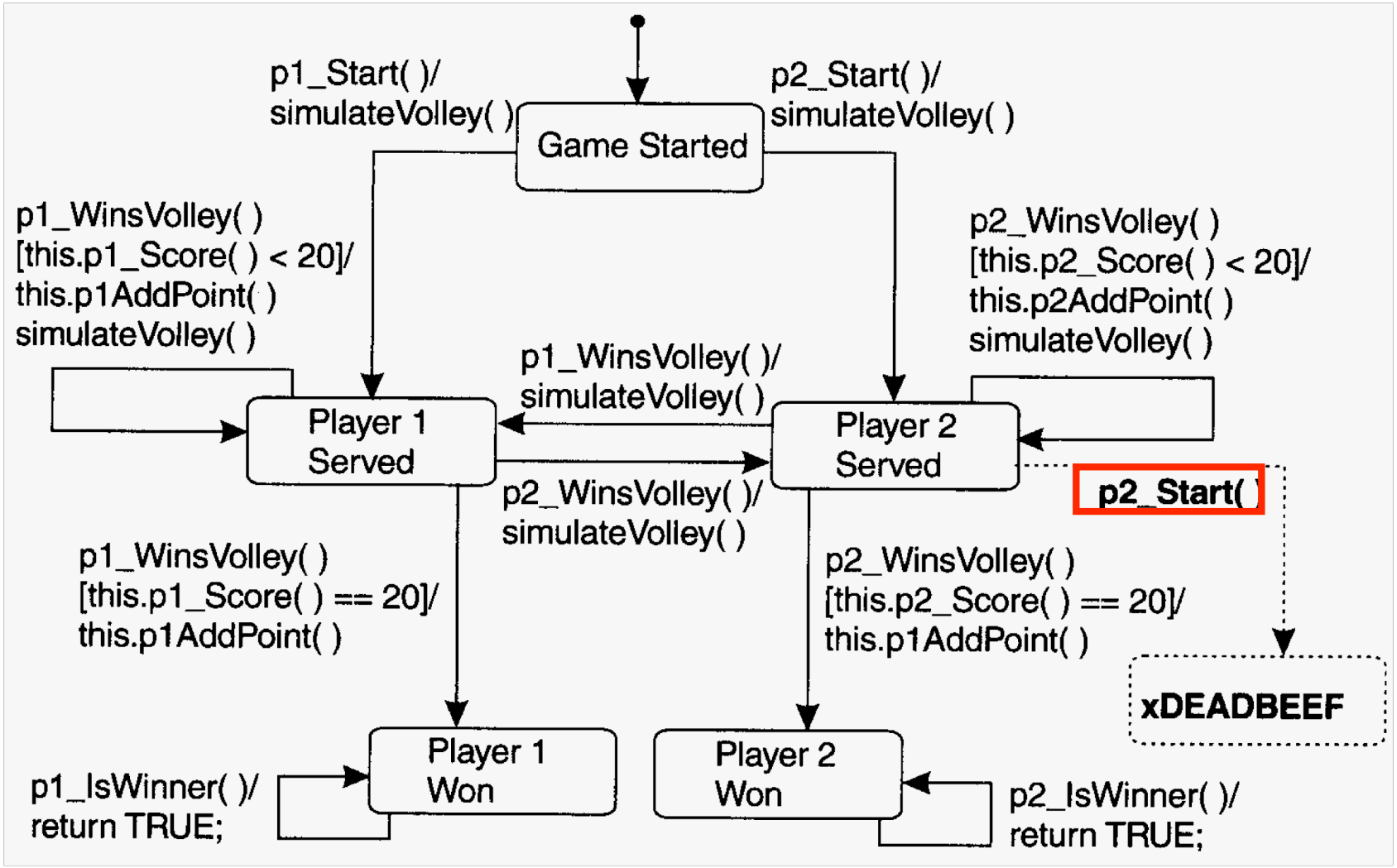


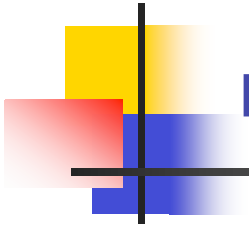


Fault model for state machines – 7

- Illegal message failure
  - Implementation fails to handle and illegal message or unspecified message correctly
  - Incorrect output is produced, the state is corrupted, or both

Sneak path to corrupt state

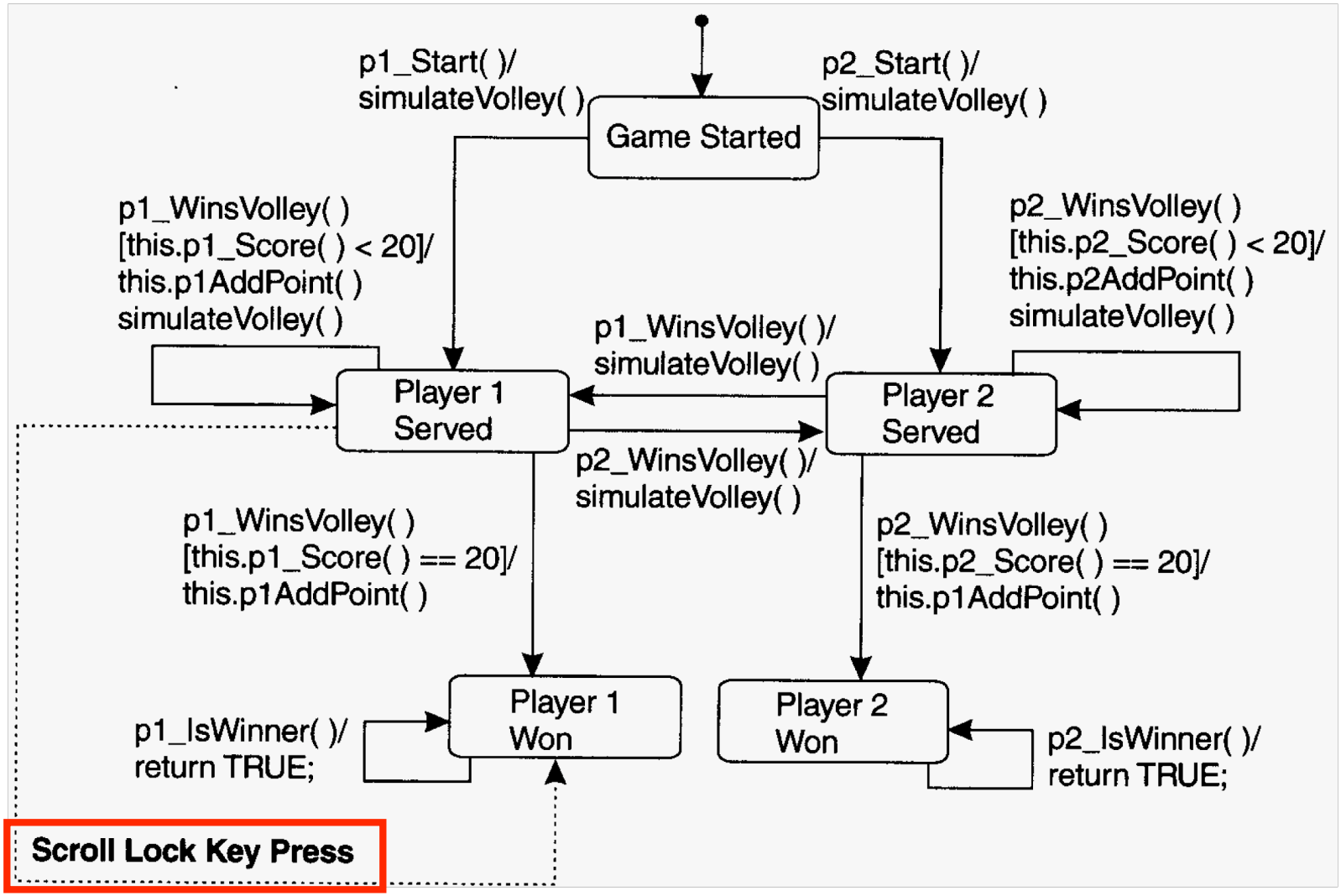


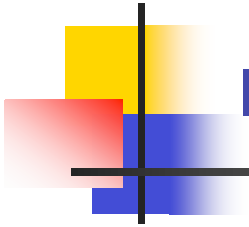


Fault model for state machines – 8

- **Trap door – undefined message/events**
  - **Implementation accepts an event that is not defined in the specification**
  - **Can result from**
    - Obsolete features that were not removed
    - Inherited features that are inconsistent with the requirements of the subclass
    - "Undocumented" features added by the developer for debugging purposes
    - Sabotage for criminal or malicious purposes

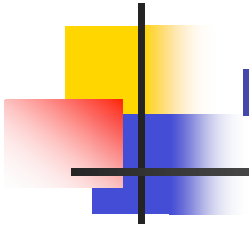
Trap door


p1_Start( )/
simulateVolley( )
p2_Start( )/
simulateVolley( )
Game Started
p1_WinsVolley( )
[this.p1_Score( ) < 20]/
this.p1AddPoint( )
simulateVolley( )
p2_WinsVolley( )
[this.p2_Score( ) < 20]/
this.p2AddPoint( )
simulateVolley( )
p1_WinsVolley( )/
simulateVolley( )
Player 1
Served
Player 2
Served
p2_WinsVolley( )/
simulateVolley( )
p1_WinsVolley( )
[this.p1_Score( ) == 20]/
this.p1AddPoint( )
p2_WinsVolley( )
[this.p2_Score( ) == 20]/
this.p1AddPoint( )
Player 1
Won
Player 2
Won
p1_IsWinner( )/
return TRUE;
p2_IsWinner( )/
return TRUE;
Scroll Lock Key Press



Incorrect Composite Behaviour

- Misuse of inheritance with modal classes can lead to state control bugs
  - **Subclasses can conflict with sequential requirements for a superclass**
  - **Need to test beyond the scope of one class**



Incorrect Composite Behaviour – 2

- Bugs occur for the following reasons
  - Missing or incorrect redefinition of a method
  - Subclass extension of the local state conflicts with a superclass state
  - Subclass fails to retarget a superclass transition
    - Switches to an incorrect or undefined superclass state
  - Order of evaluation of guards and preconditions is incorrect or sensitive to the order of evaluation
    - Guards behave as if an extra state exists
  - Order of guard evaluation produces a side effect in the subclass that is not present in the superclass
  - Default name scope resolution results in guard parameters being bound to the wrong subclass or superclass methods