Embedded Systems

Design Patterns for Embedded Systems

Design Patterns for Embedded Systems

- Categories of Design Patterns include:
  - Accessing hardware
  - Concurrency and Resource Management
  - State Machine implementation and usage
  - Safety and Reliability

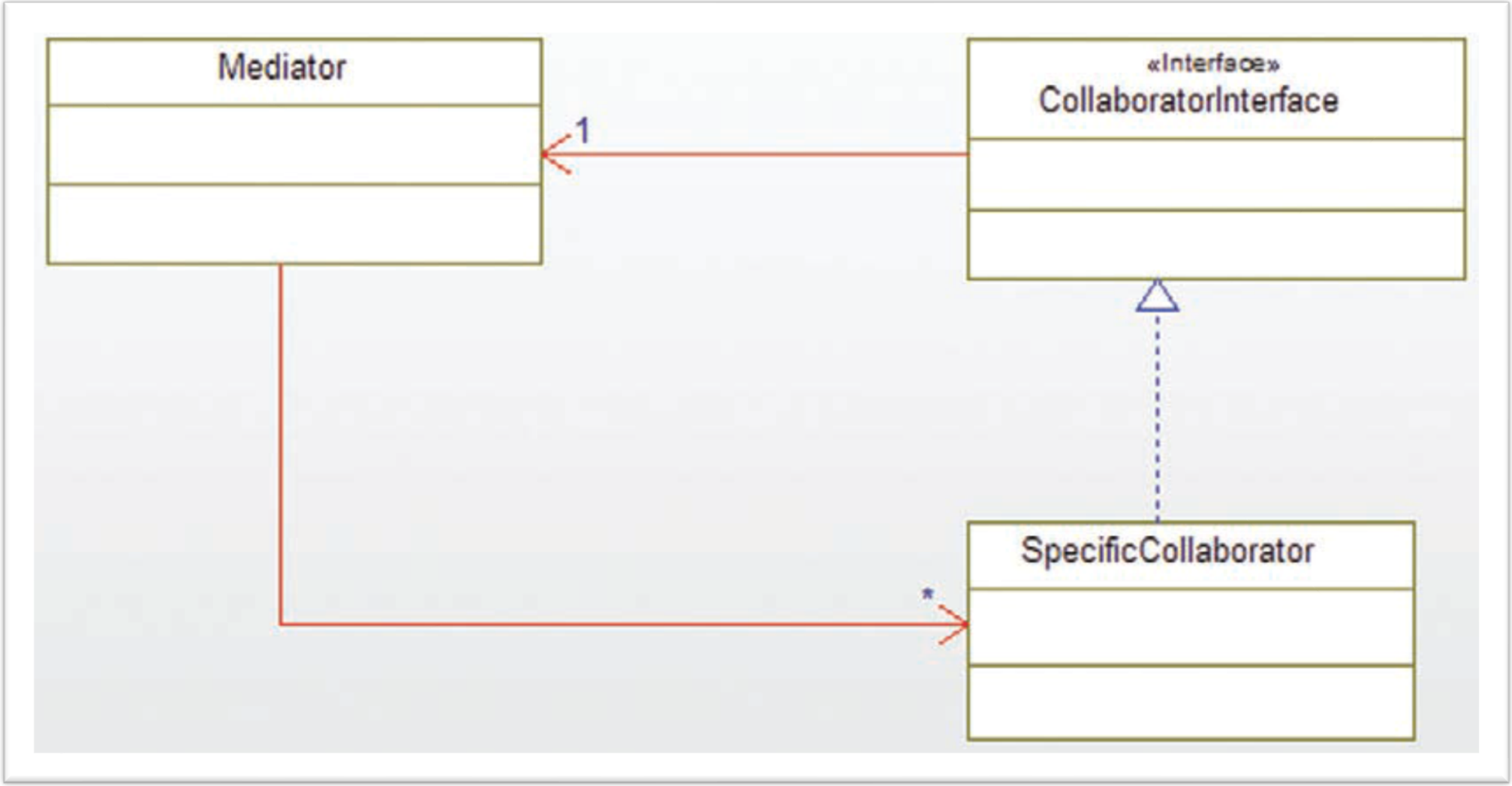
Mediator Pattern - Abstract

- Provides a means of coordinating a complex interaction among a set of elements.
- Useful for managing different hardware elements when their behavior must be coordinated in well-defined but complex ways.

Mediator Pattern - Problem

- Many embedded applications control sets of actuators that must work in concert to achieve the desired effect.
- For example, to achieve a coordinated movement of a multi-joint robot arm, all the motors must work together to provide the desired arm movement.

Mediator Pattern - Structure



Mediator Pattern - Structure

- It uses a mediator class to coordinate the actions of a set of collaborating devices to achieve the desired overall effect.
- It coordinates the control of the set of multiple Specific Collaborators (their number is indicated by the '*' multiplicity on the association between the Mediator and the Specific Collaborator)

Mediator Pattern - Structure

- The Collaborator Interface is a specification of a set of services common to all Specific Collaborators that may be invoked by the Mediator.
- For example, it is common to have reset(), shutdown, initialize() style operations for all the hardware devices to facilitate bringing them all to a known initial, recovery and/or shutdown state.

Mediator Pattern - Structure

- The Specific Collaborator represents one device and so may be a device driver or Hardware Proxy.
- It receives command messages from the Mediator and also sends messages to the Mediator when events of interest occur.

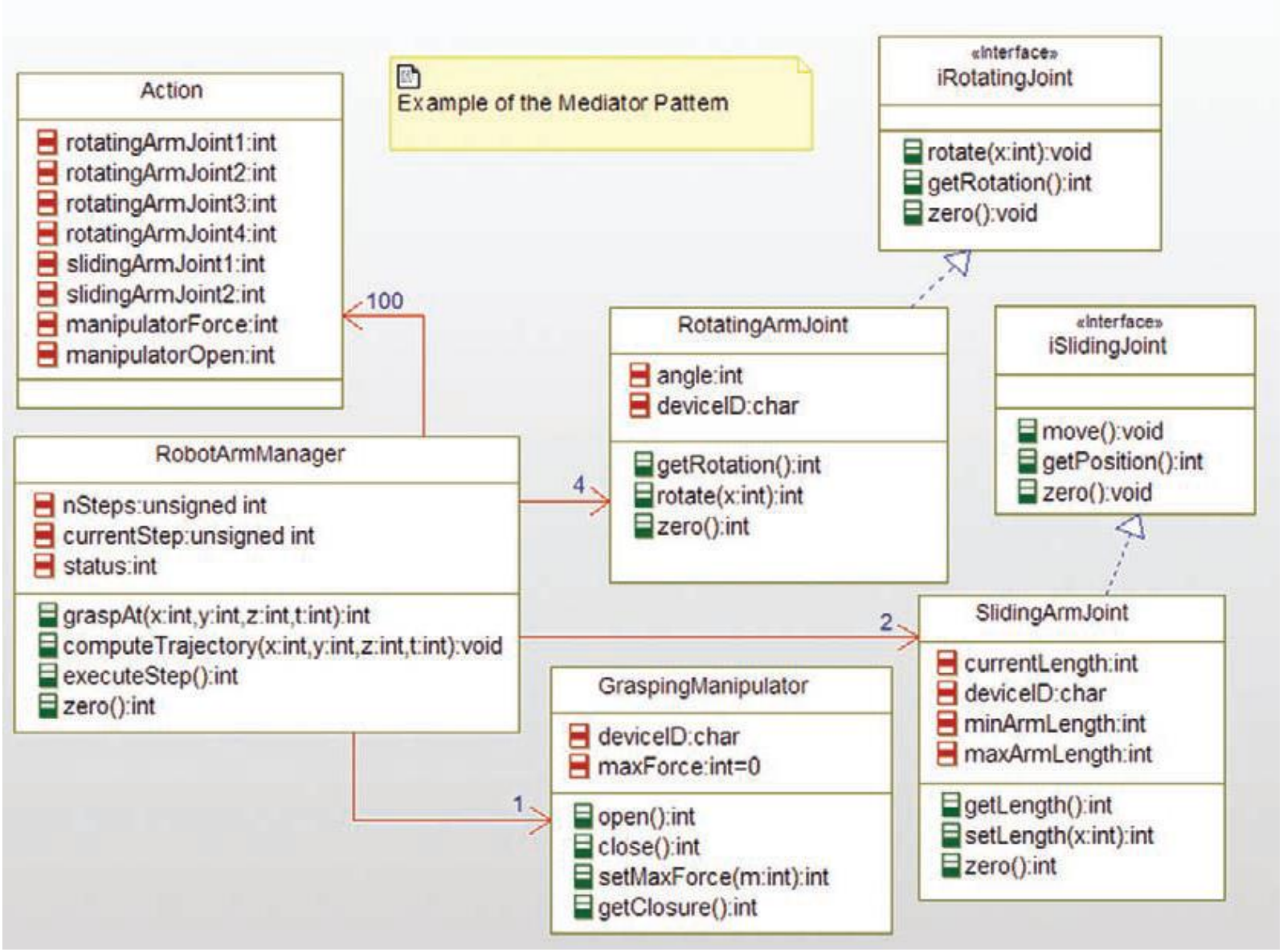
Mediator Pattern - Consequences

- This pattern creates a mediator that coordinates the set of collaborating actuators but without requiring direct coupling of those devices.
- This greatly simplifies the overall design by minimizing the points of coupling and encapsulating the coordination within a single element.
- Whenever the Collaborator would have directly contacted another Collaborator, instead it notifies the Mediator who can decide how to respond as a collective collaborative whole.

Mediator Pattern - Consequences

- Since many embedded systems must react with high time fidelity, delays between the actions may result in unstable or undesirable effects.
- It is important that the mediator class can react within those time constraints.

Mediator Pattern - Example



Mediator Pattern - Example

- The point of the example is to illustrate the value of the Mediator (RobotArmManager).
- Without its coordinating influences, all of the actuators would have to collaborate directly with their peers.
- The computation of allowable versus illegal movement paths of the arm would be divided up among the collaborators and would be very difficult to manage.

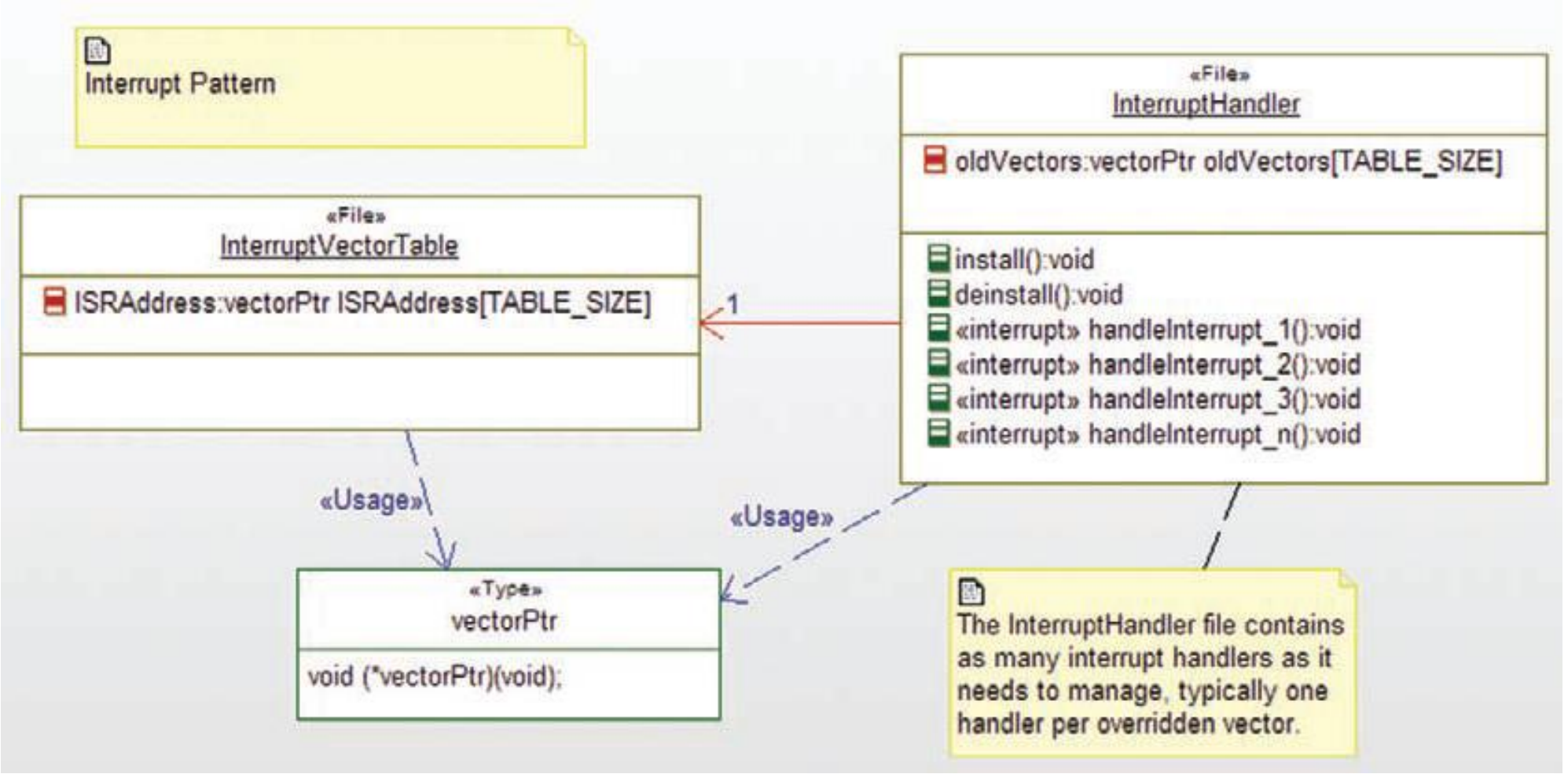
Interrupt Pattern - Abstract

- The Interrupt Pattern is a way of structuring the system to respond appropriately to incoming events.
- Once initialized, an interrupt will pause its normal processing, handle the incoming event, and then return the system to its original computations.

Interrupt Pattern – Problem

- The Interrupt Pattern is a way of structuring the system to respond appropriately to incoming events.
- In many systems, events have different levels of urgency.
- Most embedded systems have at least some events with a high urgency that must be handled even when the system is busy doing other processing.

Interrupt Pattern – Structure



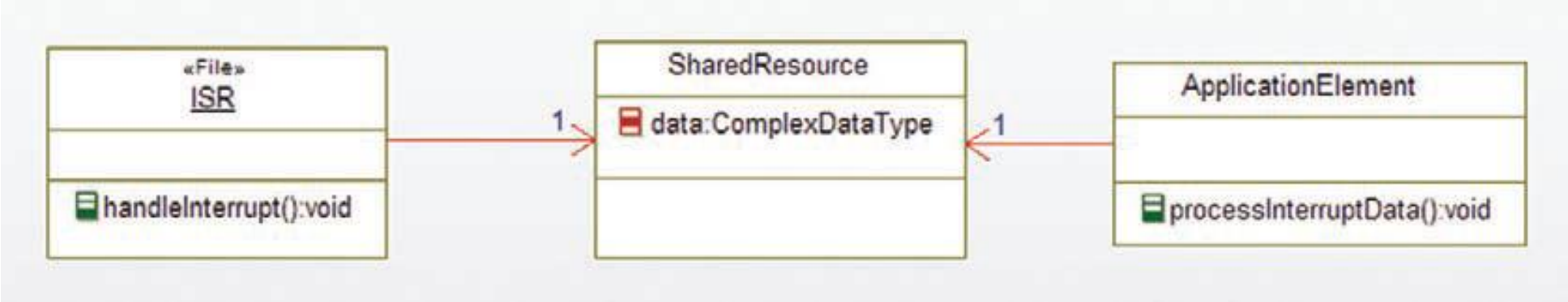
Interrupt Pattern - Consequences

- This pattern allows for highly responsive processing of events of interest.
- Normally, interrupts are disabled while an interrupt service routine is executing; this means that interrupt service routines must execute very quickly to ensure that other interrupts are not missed.

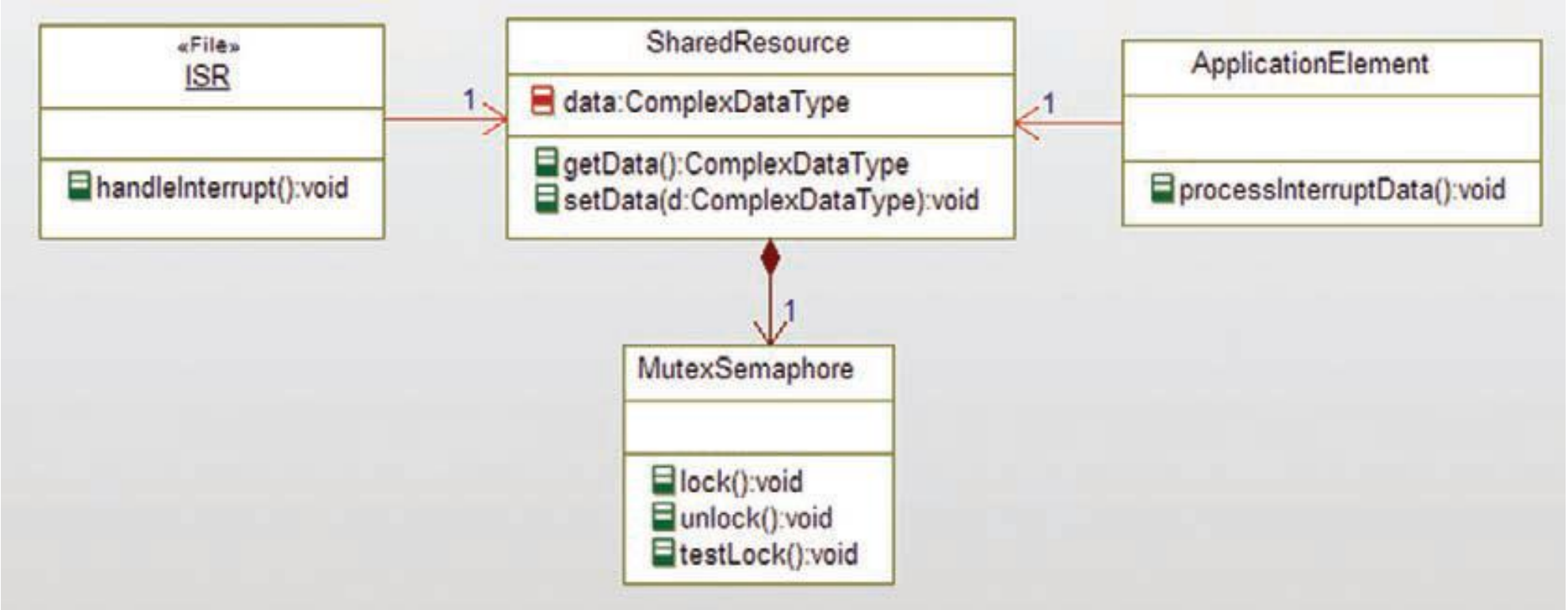
Interrupt Pattern - Consequences

- Problems arise with this pattern when the ISR processing takes too long, when an implementation mistake leaves interrupts disabled, or race conditions or deadlocks occur on shared resources.

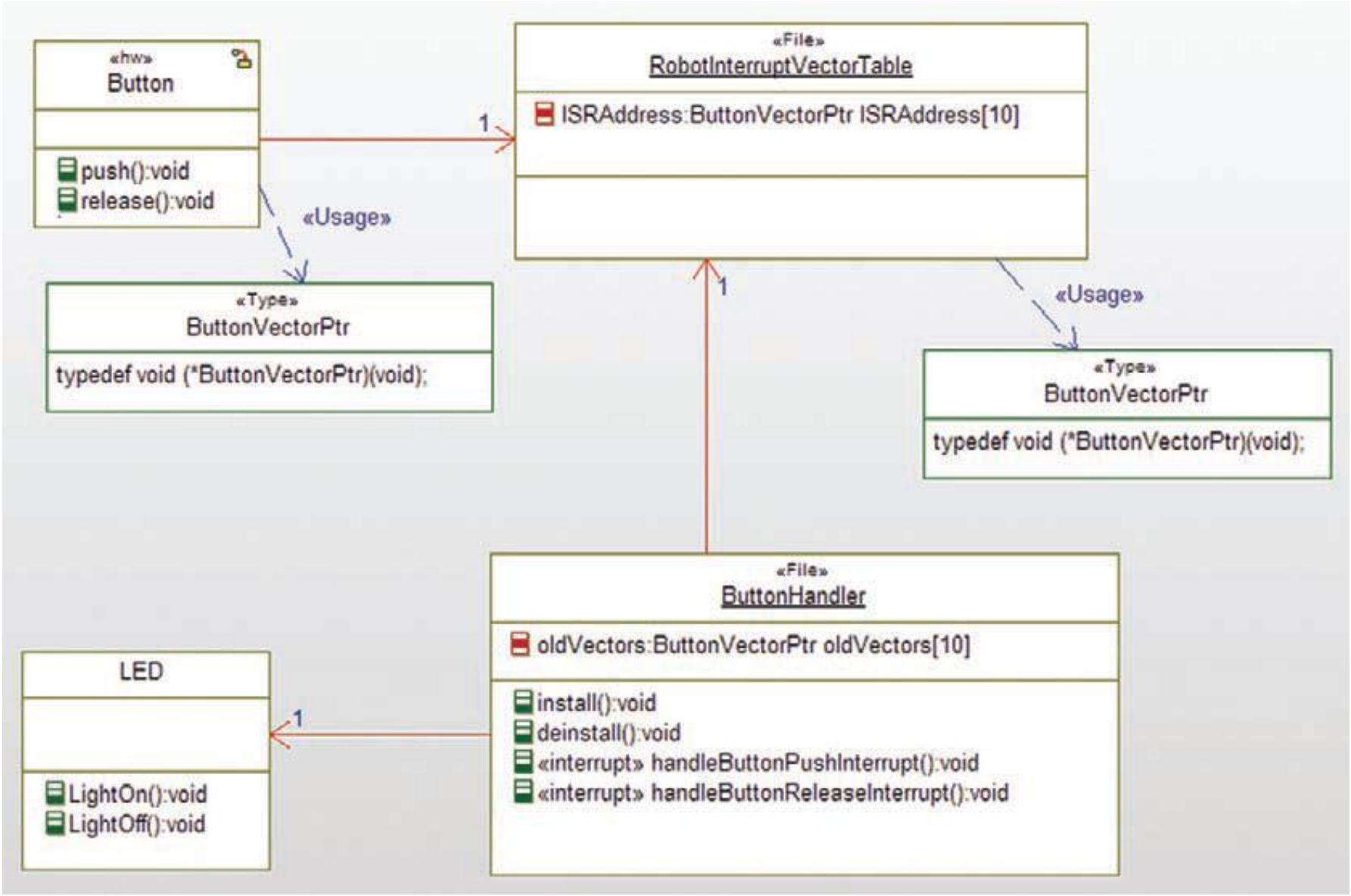
Interrupt Pattern - Potential Race Condition



Interrupt Pattern - Potential Deadlock Condition



Interrupt Pattern - Example



Polling Pattern - Abstract

- A common pattern for getting sensor data or signals from hardware is to check periodically, a process known as polling.
- Polling is useful when the data or signals are not so urgent that they can wait until the next polling period to be received or when the hardware isn't capable of generating interrupts when data or signals become available.

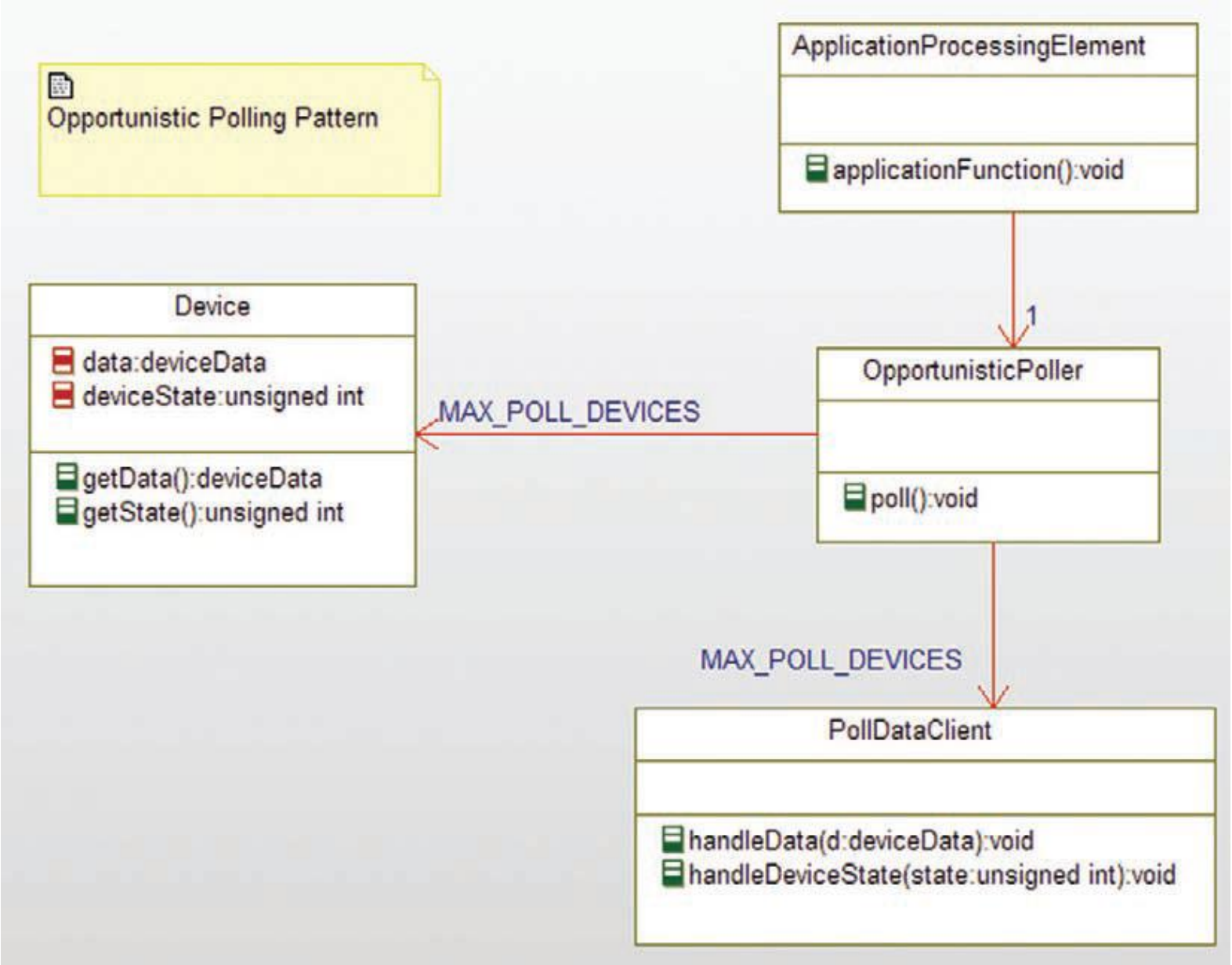
Polling Pattern - Abstract

- Polling can be periodic or opportunistic
- periodic polling uses a timer to indicate when the hardware should be sampled
- opportunistic polling is done when it is convenient for the system

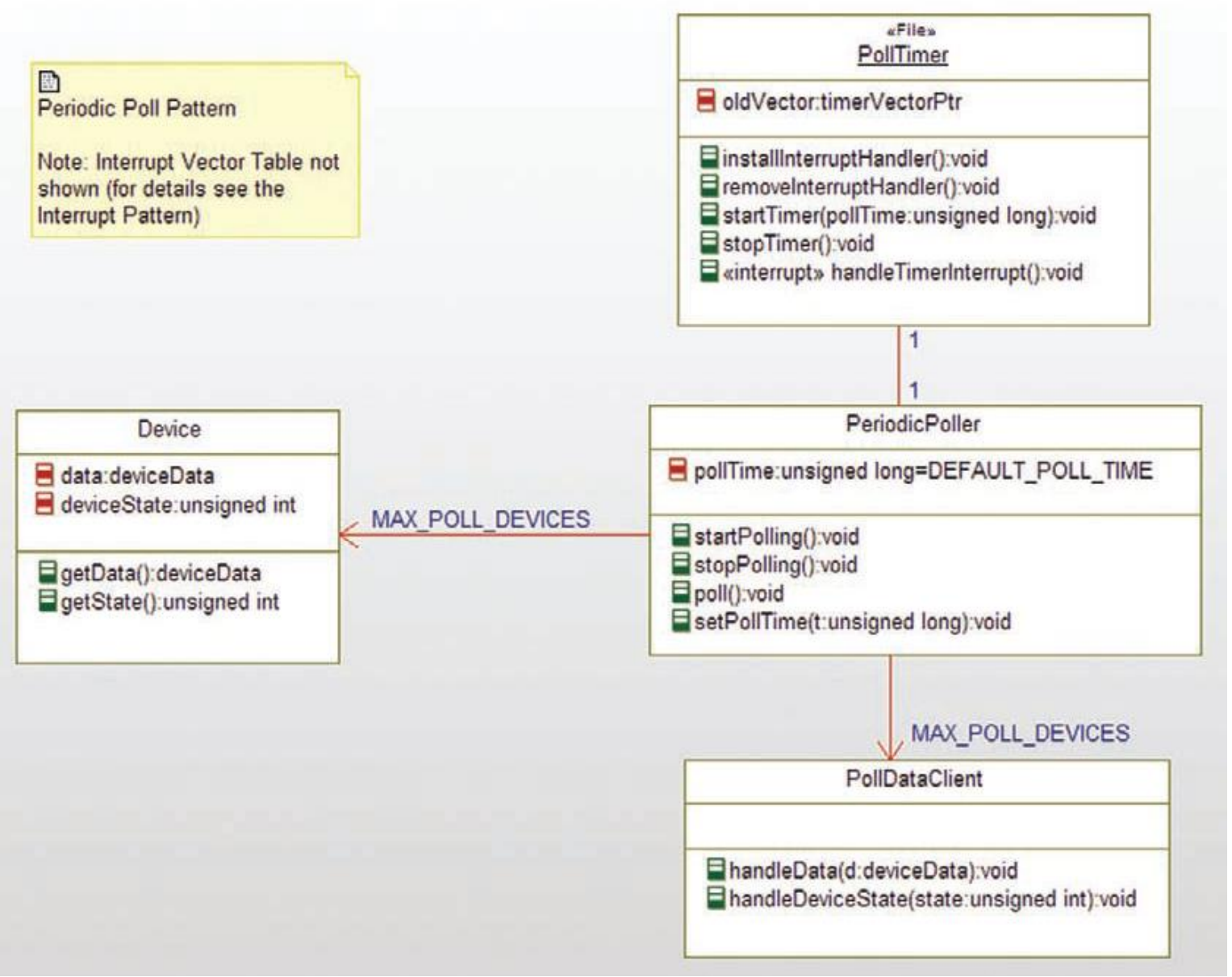
Polling Pattern - Problem

- The Polling Pattern addresses the concern of getting new sensor data or hardware signals into the system as it runs when the data or events are not highly urgent and the time between data sampling can be guaranteed to be fast enough.

Opportunistic Polling Pattern - Structure



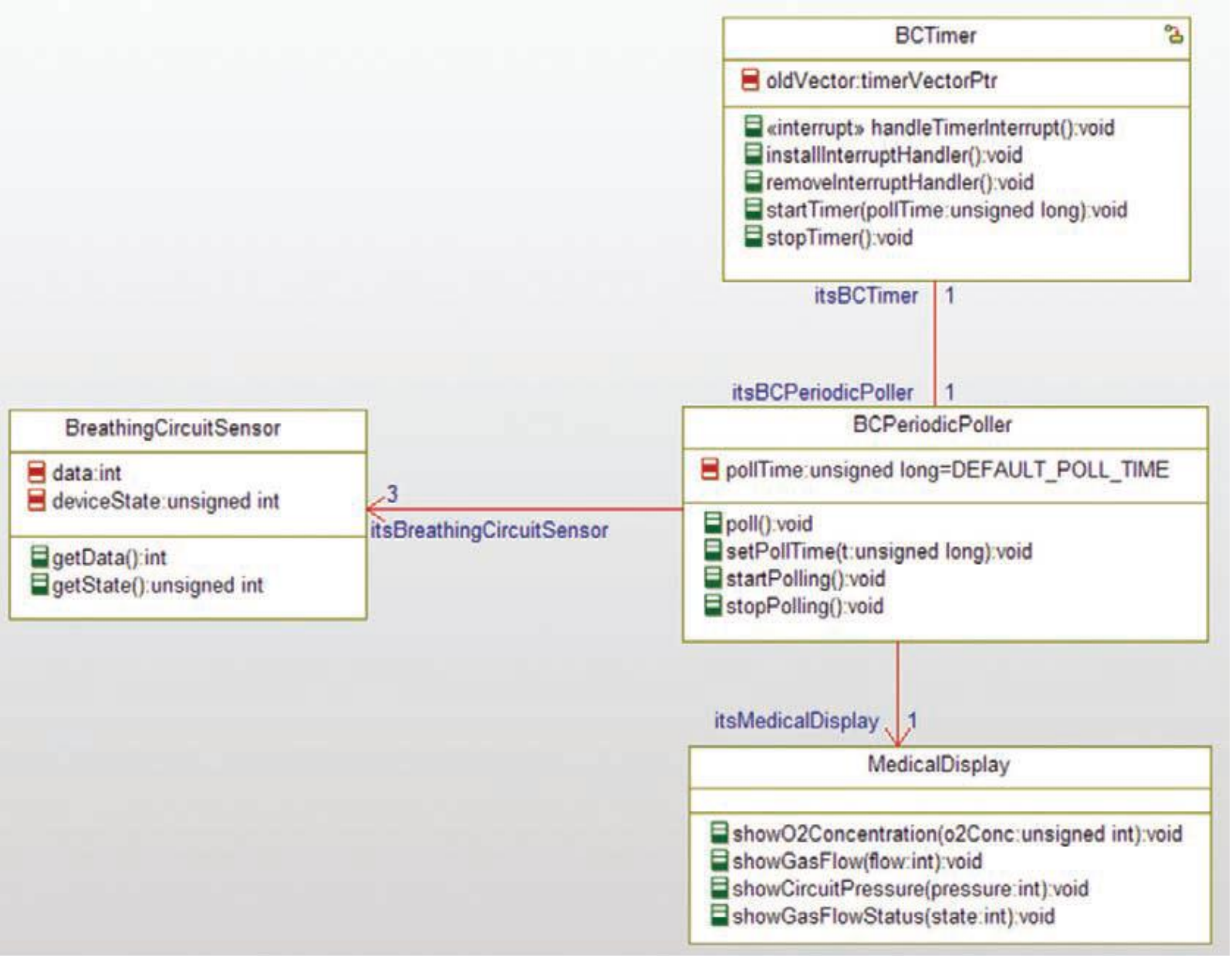
Periodic Polling Pattern - Structure



Polling Pattern - Consequences

- Polling is simpler than the setup and use of the ISRs, although periodic polling is usually implemented with an ISR tied to a poll timer.
- Polling can check many different devices at the same time for status changes but is usually less timely than interrupts.
- If data arrive faster than the poll time, then data will be lost.
- This is not a problem in many applications but is fatal in others.

Polling Pattern - Example



References

- **Chapter 3**: Douglass, Bruce Powel. **Design patterns for embedded systems in C: an embedded software engineering toolkit**. Elsevier, 2010.