Patterns

- Observer Pattern

Accessing Hardware Patterns

- Hardware Proxy Pattern
- Hardware Adapter Pattern
- Mediator Pattern
- Interrupt Pattern
- Polling Pattern

Concurrency and Resource Management

- Cyclic Executive Pattern
- Critical Region Pattern

Lecture 6

● Pattern Hatching – Locating the Right Patterns

● Pattern Mining – Rolling Your Own Patterns

● Pattern Instantiation – Applying Patterns in Your Designs

● Design pattern is a generalized solution to a commonly occurring problem.

The solution must be general enough to be applied in a wide set of application domains.

● A good design is composed of a set of design patterns applied to a piece of functional software, achieves a balanced optimization of the design criteria, incurs an acceptable cost

● Basic Structure of a Design Pattern

- ● Name
  - ● provides a "handle" or means to reference the pattern.
- ● Purpose
  - ● provides the problem context and the QoS aspects the pattern seeks to optimize.
  - ● specifies under which situations the pattern is appropriate and under which situations it should be avoided
- ● Solution
  - ● structure and behavior of the pattern.
  - ● elements of the pattern and their roles in the pattern context.
- ● Consequences
  - ● set of pros and cons of the use of the pattern.

Pattern Hatching

Familiarize yourself with patterns -> Identify Design Criteria -> Rank Design Criteria by Criticality -> Identify Potential Design Alternatives -> Evaluate Design Alternatives -> Select and Apply Design Alternatives -> Verify Design Functionality, if functionality broken, go back to identify design alternatives. Else -> Verify Design Goals are met, if not, go back to identify design alternatives, else -> End.

Some Common Design Optimization Criteria

- Performance
  - Worst case
  - Average case
- Predictability
- Schedulability
- Throughput: number of tasks to be executed per second
  - Average: Average time to execute task
  - Sustained: not working average, but always working at peek potential (ثبات الأداء مع زيادة التاسكات)
  - Burst: القدرة على التعامل مع فترات الضغط المفاجئة زي وقت نزول النتيجة
- Reliability
  - With respect to errors or failures

Design Tradeoff Spreadsheet

We list Design criteria and their weights in rows and Design solution in columns,

Filling cells of each criteria and solution with a score, total weighted score for every design solution is calculated by: Sum of Design criteria weight * Design solution score. The highest Design Solution's total weighted score is the answer

Design Tradeoff Spreadsheet

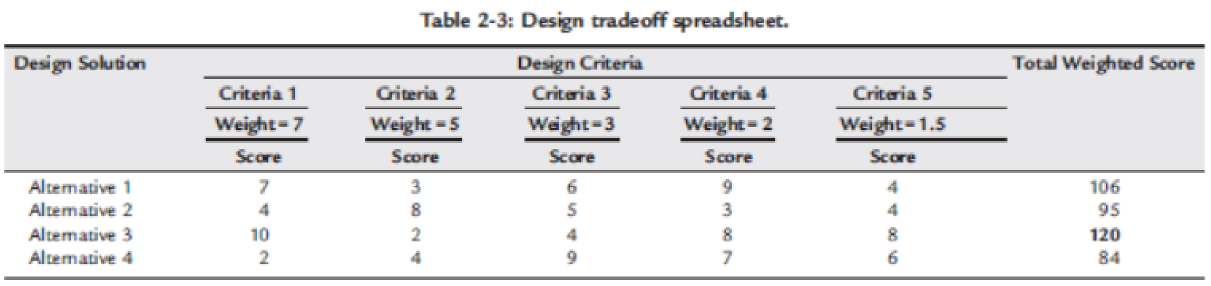


Table 2-3: Design tradeoff spreadsheet.

Design Solution	Design Criteria					Total Weighted Score
	Criteria 1	Criteria 2	Criteria 3	Criteria 4	Criteria 5	
	Weight = 7	Weight = 5	Weight = 3	Weight = 2	Weight = 1.5	
	Score	Score	Score	Score	Score	
Alternative 1	7	3	6	9	4	106
Alternative 2	4	8	5	3	4	95
Alternative 3	10	2	4	8	8	**120**
Alternative 4	2	4	9	7	6	84

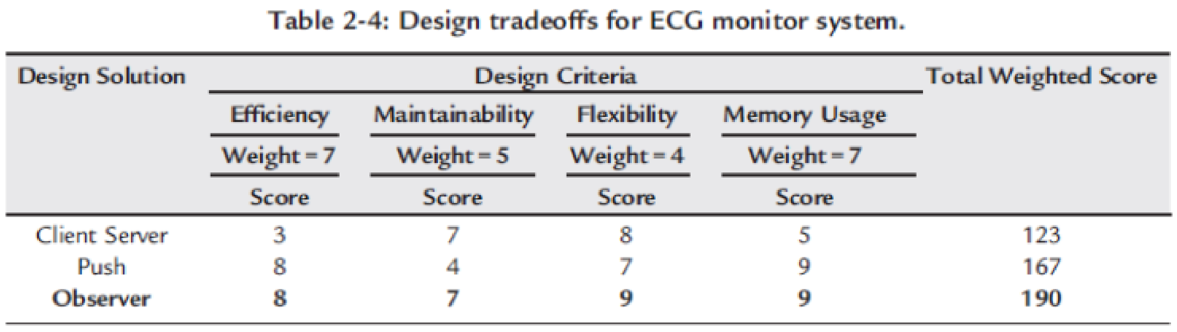


Table 2-4: Design tradeoffs for ECG monitor system.

Design Solution	Design Criteria				Total Weighted Score
	Efficiency	Maintainability	Flexibility	Memory Usage	
	Weight = 7	Weight = 5	Weight = 4	Weight = 7	
	Score	Score	Score	Score	
Client Server	3	7	8	5	123
Push	8	4	7	9	167
Observer	**8**	**7**	**9**	**9**	**190**

Observer Design Pattern

Provides a means for objects to "listen in" on others while requiring no modifications to the data servers.

From Embedded Perspective, sensor data can be easily shared to other elements.

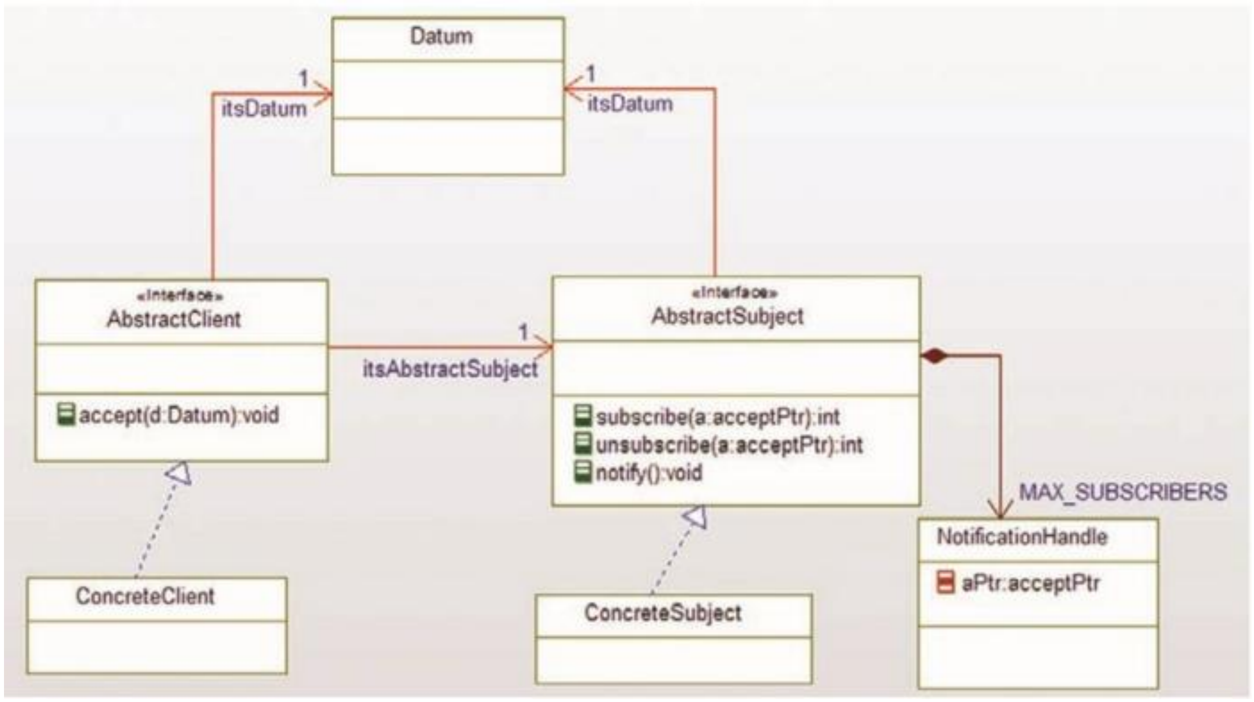
Abstract

- known as the "Publish-Subscribe Pattern"
- Provides notification to a set of interested clients that relevant data have changed.
- It does this without requiring the data server to have any a priori knowledge about its clients.
- Clients(Sensors) can use Subscribe function to add themselves to the notification list.
- The data server can then enforce whatever notification policy it desires.

Problem

- Each client can request data periodically from a data server in case the data have changed.
- If the data server pushes the data out, then it must know who all of its clients are.
- subscription and un-subscription services to data servers are allowed to clients.
- The pattern allows dynamic modification of subscriber lists.
- The server can enforce the appropriate update policy to the notification of its interested clients.

Structure



- Abstract Client Interface: has an accept function to accept data from the datum
- Abstract Subject Interface: The server, which has two functions to subscribe and unsubscribe
- Datum: the data shared among clients
- Notification Handle: Class handles the notification list and subscribers (add or remove subscribers)

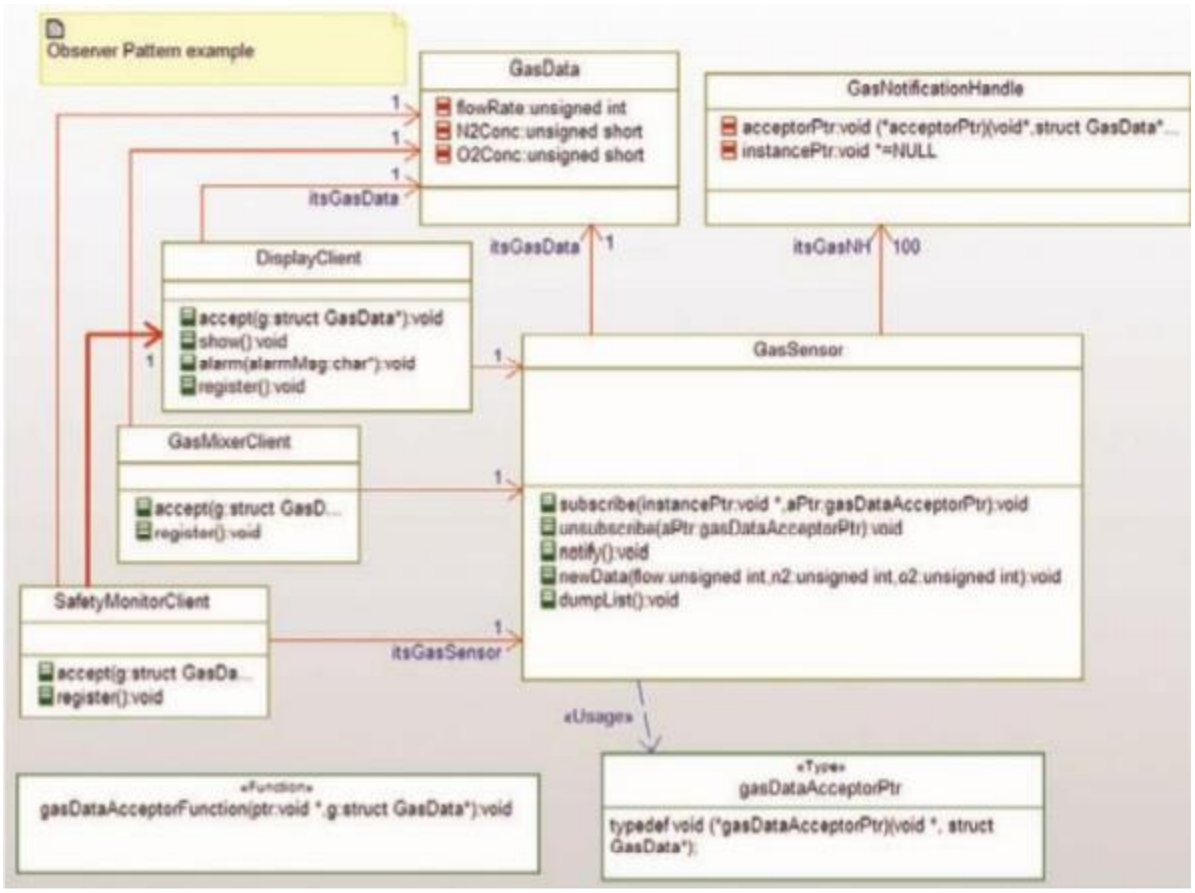
Consequences

- Simplifies the process of distributing data to a set of clients.
- Maintains the fundamental client-server relation while providing run-time flexibility of adding clients.
- Compute efficiency is maintained since clients can only know updates when data is changed.

Implementation Complexities

- The most complex aspects of this pattern are the implementation of the notification handle and the management of the notification handle list.
- The easiest approach for the notification list is to declare an array big enough to hold all potential clients. This wastes memory in highly dynamic systems with many potential clients.
- Another approach is to construct a linked list of all clients.

Example



Lecture 7

- Categories of Design Patterns include:
    - Accessing hardware
    - Concurrency and Resource Management
    - State Machine implementation and usage
    - Safety and Reliability
- Hardware Proxy Pattern (الوسيط)

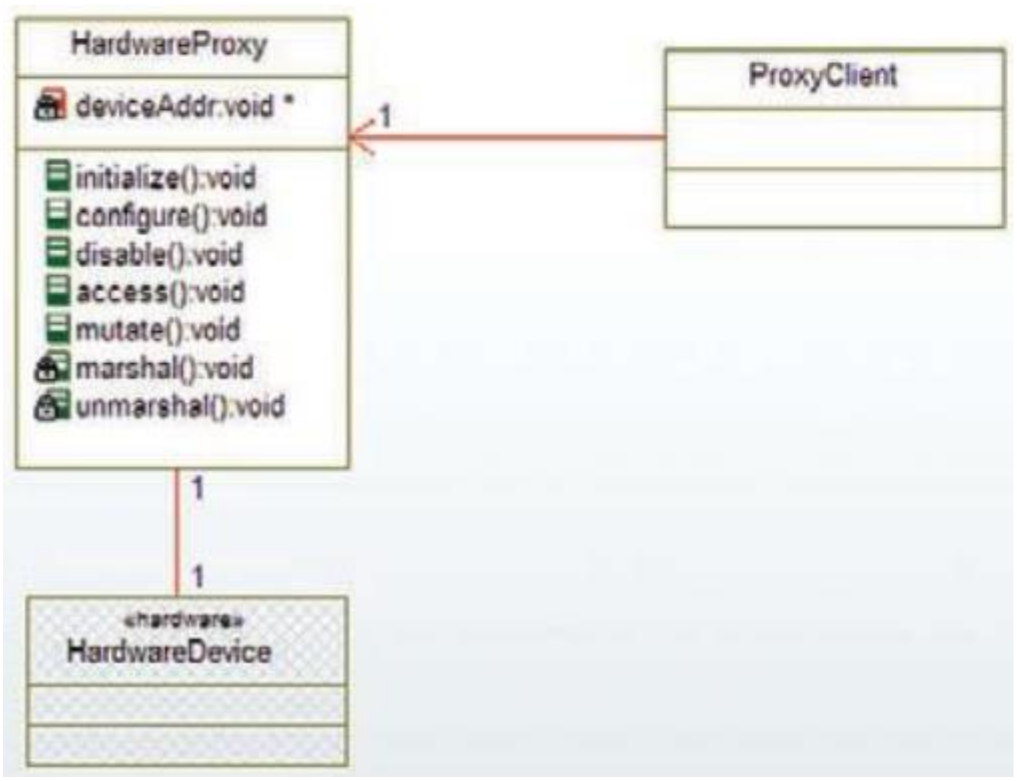
Abstract

- Creates a software element responsible for access to a piece of hardware and encapsulation of hardware compression and coding implementation.
- Uses a class (or struct) to encapsulate all access to a hardware device, regardless of its physical interface.
- Provides an encoding and connection-independent interface for clients and so promotes easy modification should the nature of the device interface or connection change

Problem

- By providing a proxy to sit between the clients and the actual hardware, the impact of hardware changes is greatly limited, easing any modifications in the hardware.

Structure



- marshal() – converts presentation (client) data format to native (motor) format
- unmarshal() – converts native (motor) data format to presentation (client) format

Consequences

- It provides flexibility for the actual hardware interface to change with absolutely no changes in the clients.
- This means that the proxy clients are usually unaware of the native format of the data and manipulate them only in presentation format.
- This can, however, have a negative impact on runtime performance.
- It may be more time efficient for the clients to know the encoding details and manipulate the data in their native format.

● Hardware Adapter Pattern (Like a convertor)

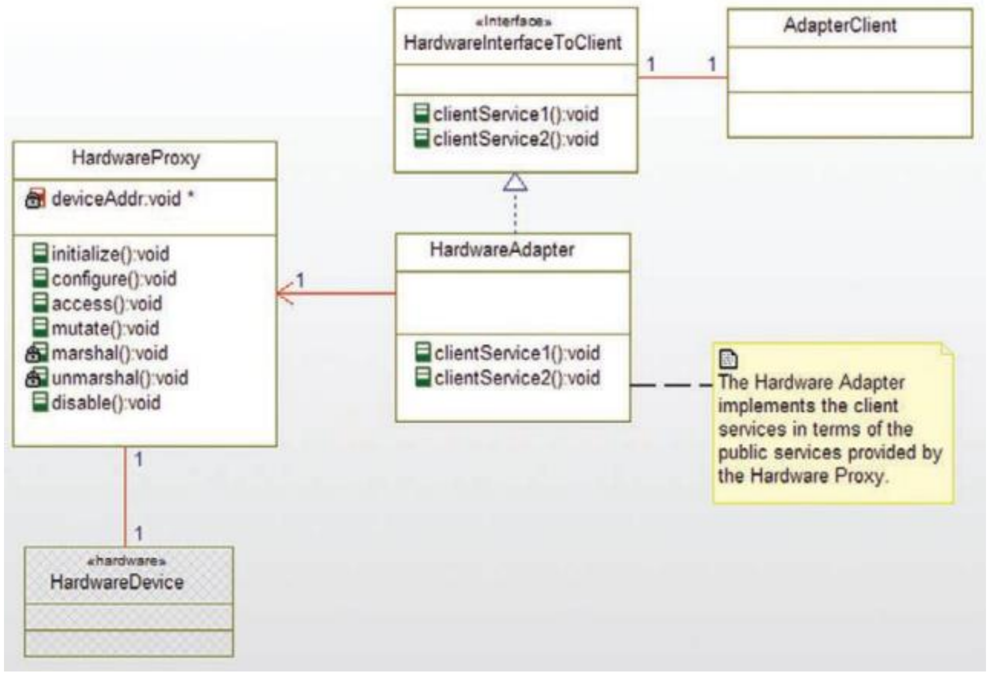
Abstract

- Provides a way of adapting an existing hardware interface into the expectations of the application.
- It is useful when the application requires or uses one interface, but the actual hardware provides another.
- Creates an element that converts between the two interfaces.

Problem

- It is useful when you have hardware that meets the semantic need of the system but that has an incompatible interface.
- The goal of the pattern is to minimize the reworking of code when one hardware design or implementation is replaced with another.

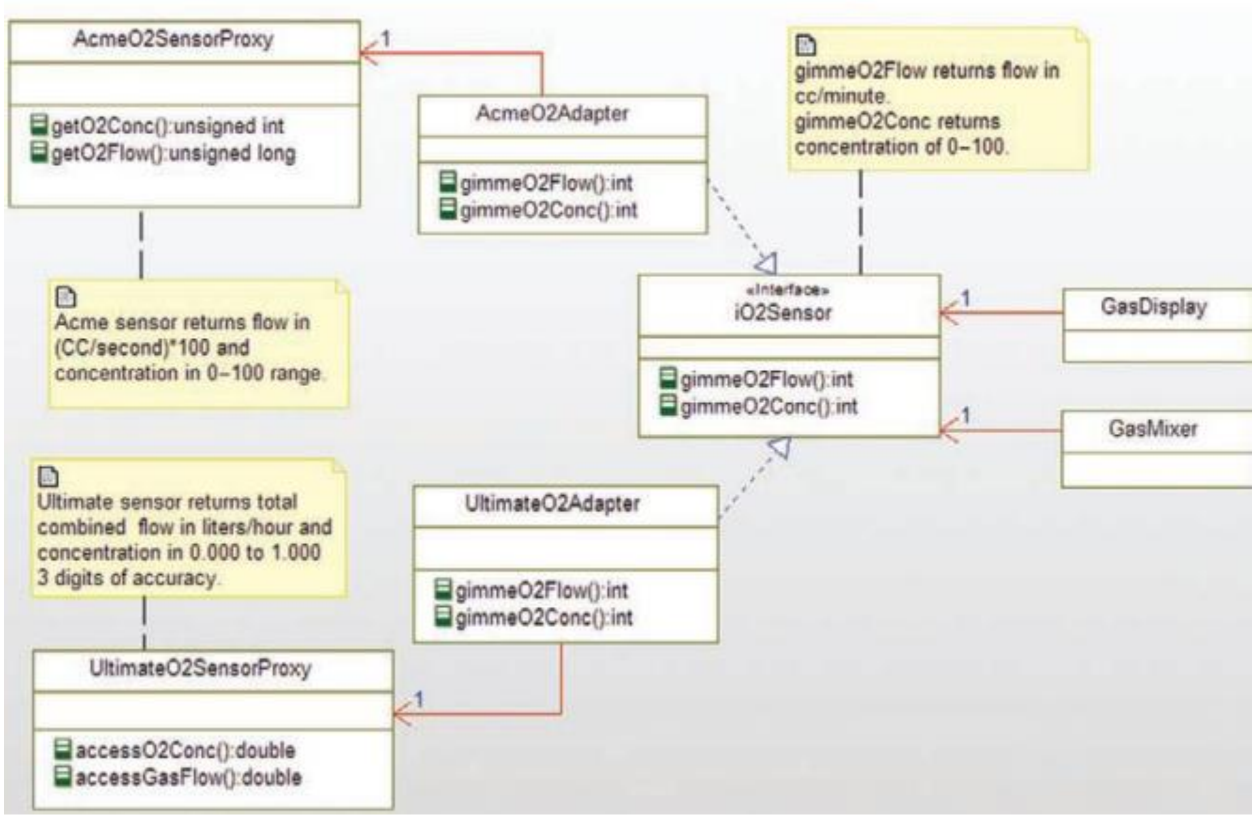
Structure



Consequences

- It allows various Hardware Proxies and their related hardware devices to be used as-is in different applications, while at the same time allowing existing applications to use different hardware devices without change.
- However, this pattern adds a level of indirection and therefore decreases runtime performance.

Example


AcmeO2SensorProxy
getO2Conc():unsigned int
getO2Flow():unsigned long
1
AcmeO2Adapter
gimmeO2Flow():int
gimmeO2Conc():int
gimmeO2Flow returns flow in cc/minute.
gimmeO2Conc returns concentration of 0-100.
Acme sensor returns flow in (CC/second)*100 and concentration in 0-100 range.
«Interface»
iO2Sensor
gimmeO2Flow():int
gimmeO2Conc():int
1
GasDisplay
1
GasMixer
Ultimate sensor returns total combined flow in liters/hour and concentration in 0.000 to 1.000 3 digits of accuracy.
UltimateO2Adapter
gimmeO2Flow():int
gimmeO2Conc():int
UltimateO2SensorProxy
accessO2Conc():double
accessGasFlow():double
1

Lecture 8

● Mediator Pattern (شبه ظابط المرور)

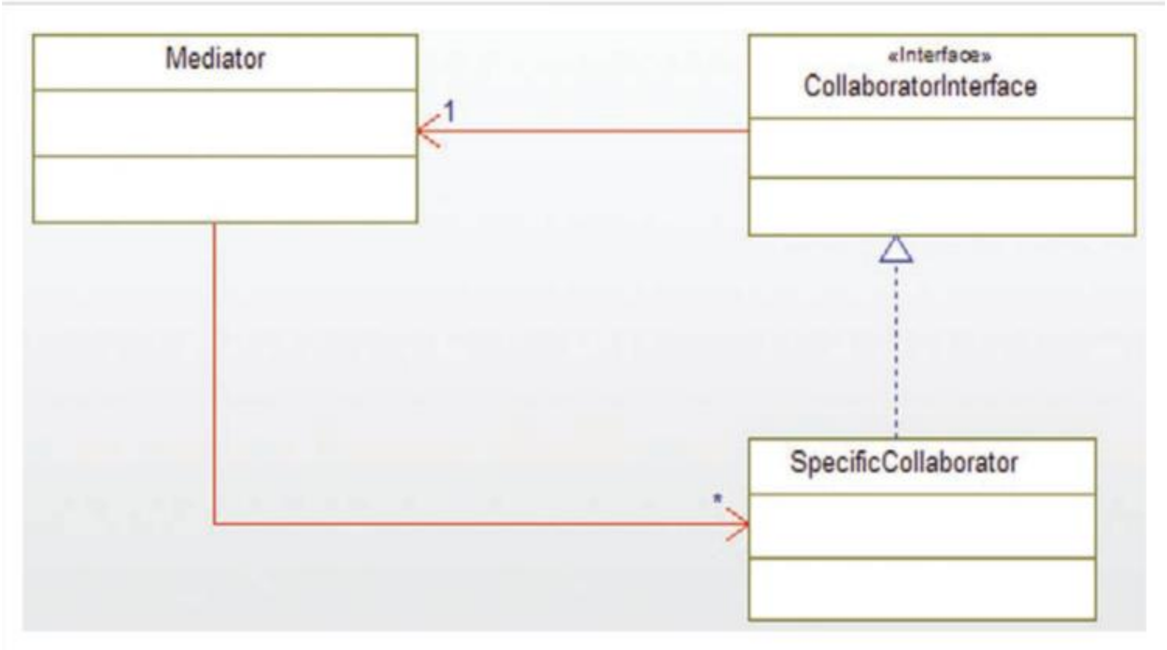
Abstract

- 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.

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.

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)
- 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.
- 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.

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.
- 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.

● Interrupt Pattern

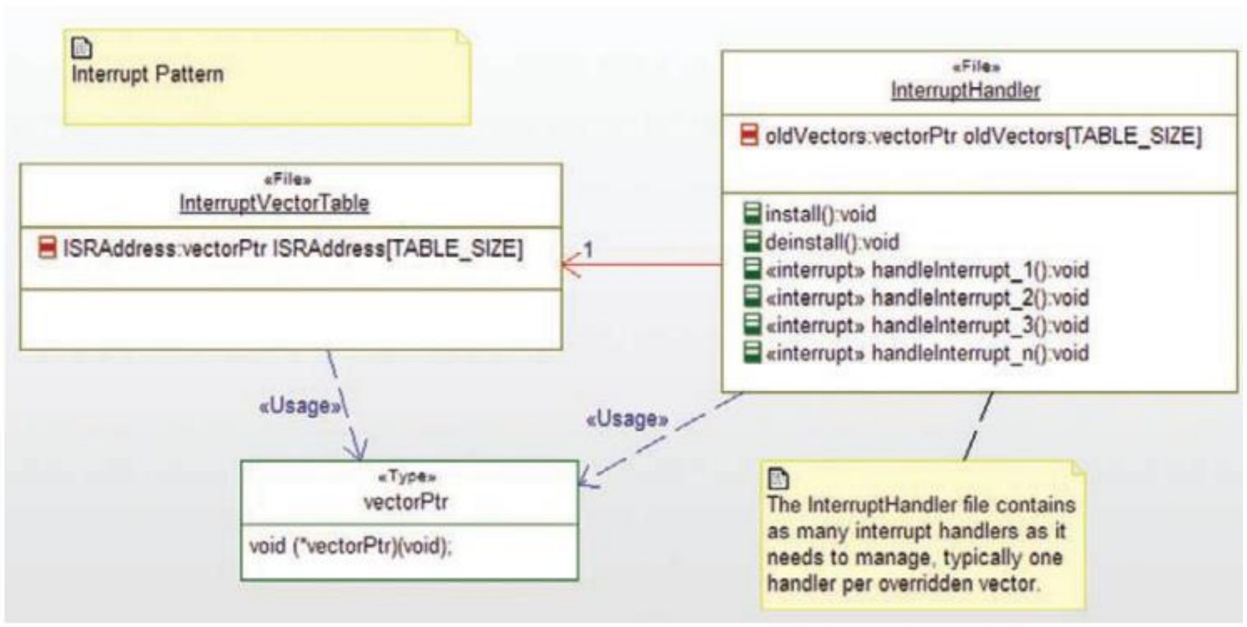
Abstract

- 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.

Problem

- 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.

Structure


Interrupt Pattern
«File»
InterruptHandler
oldVectors:vectorPtr oldVectors[TABLE_SIZE]
install():void
deinstall():void
«interrupt» handleInterrupt_1():void
«interrupt» handleInterrupt_2():void
«interrupt» handleInterrupt_3():void
«interrupt» handleInterrupt_n():void
«File»
InterruptVectorTable
ISRAddress:vectorPtr ISRAddress[TABLE_SIZE]
1
«Usage»
«Usage»
«Type»
vectorPtr
void (*vectorPtr)(void);
The InterruptHandler file contains as many interrupt handlers as it needs to manage, typically one handler per overridden vector.

Interrupt Vector Table: a table contains interrupt service routines addresses of interrupt handlers

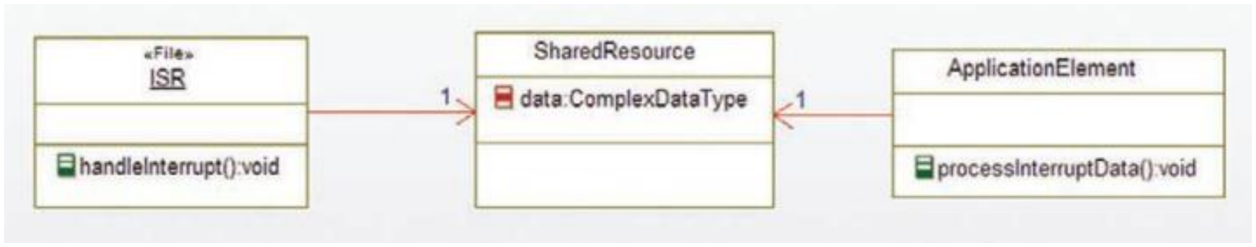
Interrupt Handler: a file contains all interrupts

VectorPtr: a pointer to point an interrupt handler to its address in the table

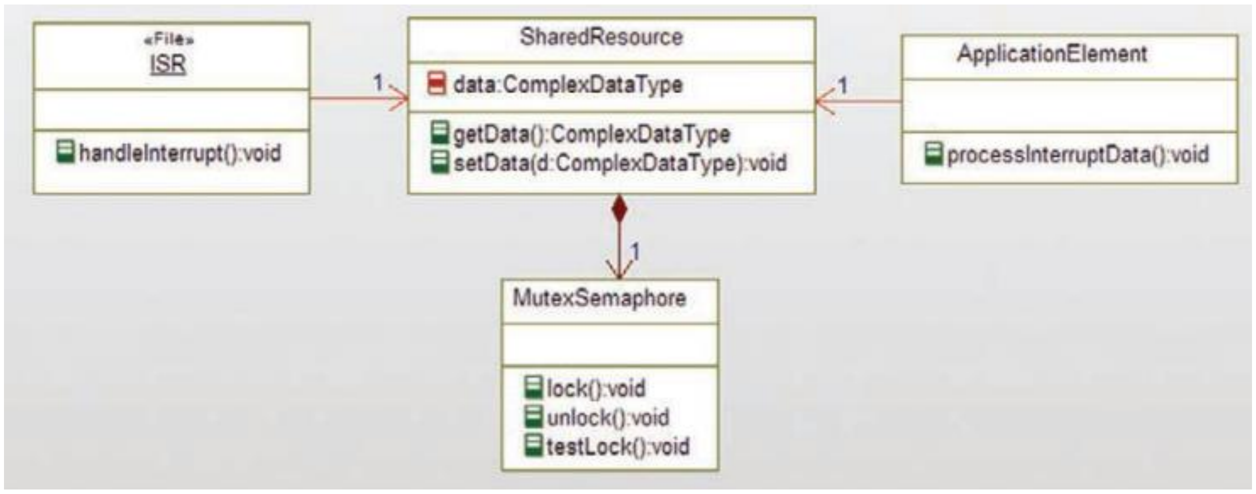
Consequences

- 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.
- 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.

Potential Race Condition



Potential Deadlock Condition



Polling Pattern

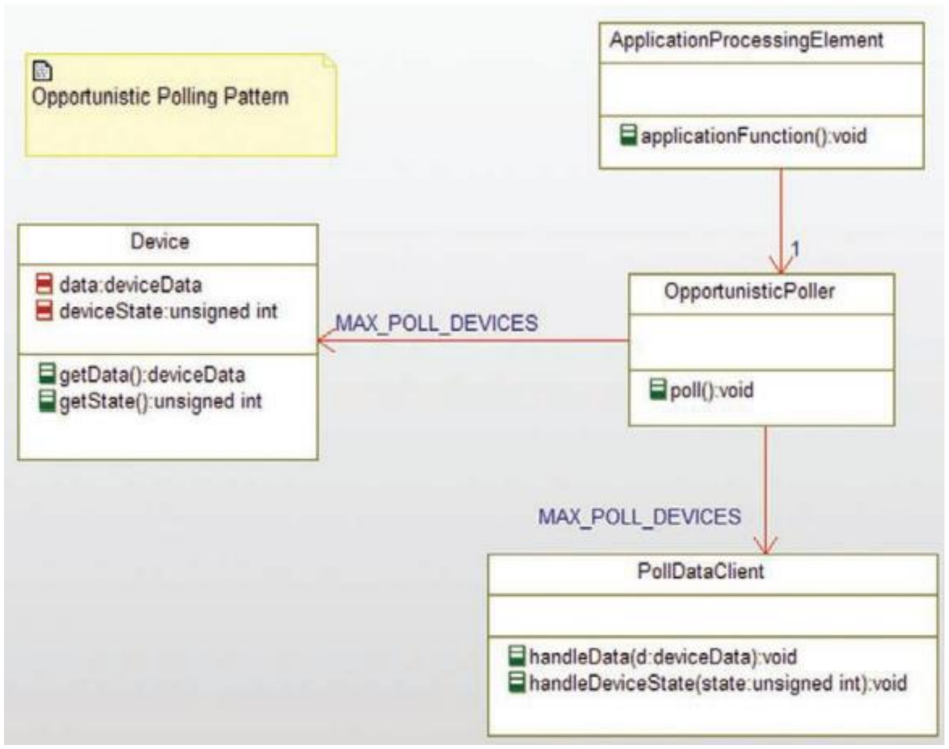
Abstract

- 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 can be periodic or opportunistic (waiting for the right moment).
- periodic polling uses a timer to indicate when the hardware should be sampled
- opportunistic polling is done when it is convenient for the system

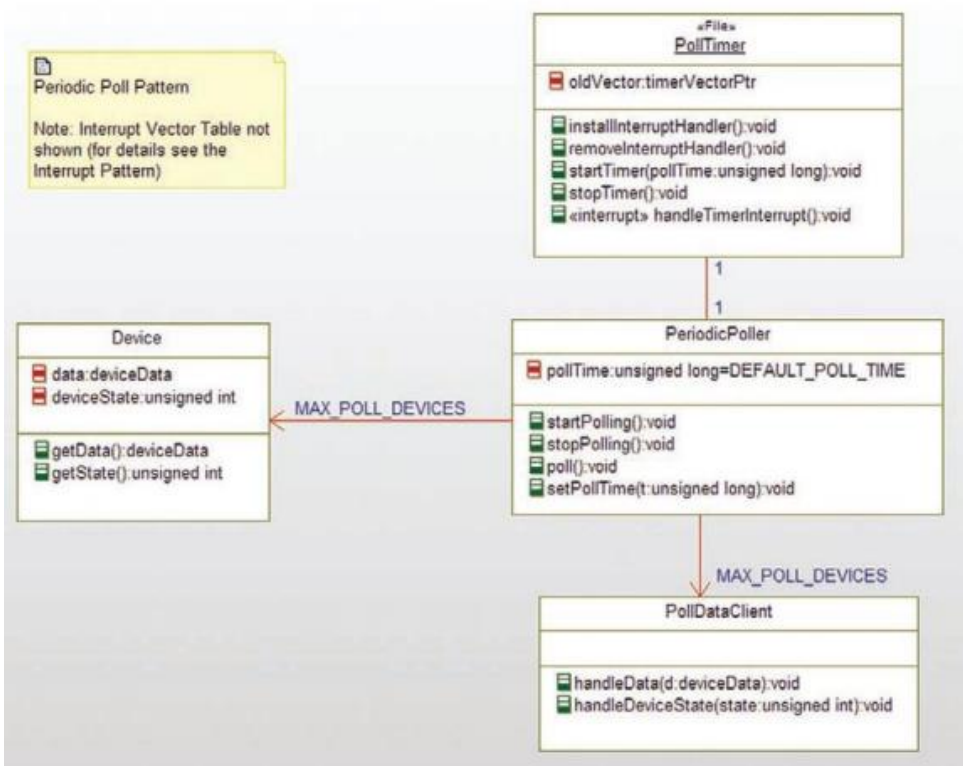
Problem

- 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



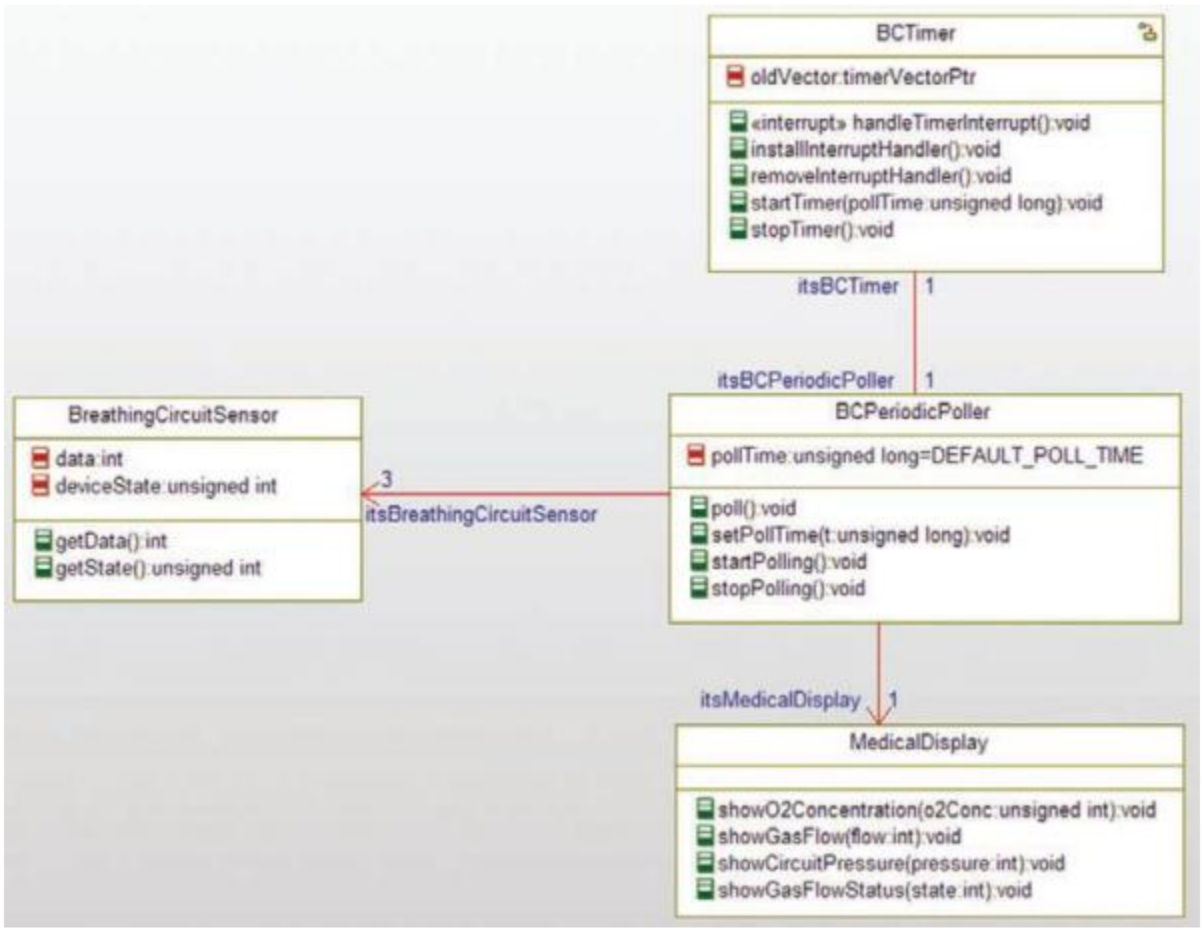
Periodic Polling Pattern Structure


Periodic Poll Pattern
Note: Interrupt Vector Table not shown (for details see the Interrupt Pattern)
«File»
PollTimer
oldVector:timerVectorPtr
installInterruptHandler():void
removeInterruptHandler():void
startTimer(pollTime:unsigned long):void
stopTimer():void
«interrupt» handleTimerInterrupt():void
1
1
Device
data:deviceData
deviceState:unsigned int
getData():deviceData
getState():unsigned int
MAX_POLL_DEVICES
PeriodicPoller
pollTime:unsigned long=DEFAULT_POLL_TIME
startPolling():void
stopPolling():void
poll():void
setPollTime(t:unsigned long):void
MAX_POLL_DEVICES
PollDataClient
handleData(d:deviceData):void
handleDeviceState(state:unsigned int):void

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.

Example


BCTimer
oldVector:timerVectorPtr
«interrupt» handleTimerInterrupt():void
installInterruptHandler():void
removeInterruptHandler():void
startTimer(pollTime:unsigned long):void
stopTimer():void
itsBCTimer
1
itsBCPeriodicPoller
1
BCPeriodicPoller
pollTime:unsigned long=DEFAULT_POLL_TIME
poll():void
setPollTime(t:unsigned long):void
startPolling():void
stopPolling():void
BreathingCircuitSensor
data:int
deviceState:unsigned int
getData():int
getState():unsigned int
3
itsBreathingCircuitSensor
itsMedicalDisplay
1
MedicalDisplay
showO2Concentration(o2Conc:unsigned int):void
showGasFlow(flow:int):void
showCircuitPressure(pressure:int):void
showGasFlowStatus(state:int):void

Lecture 9

● Cyclic Executive Pattern

Abstract

● used for very small embedded applications where it allows multiple tasks to be run pseudo-concurrently without the overhead of a complex scheduler or RTOS.

Problem

● Many embedded systems are tiny applications that have extremely tight memory and time constraints and cannot possibly host even a scaled-down microkernel RTOS.

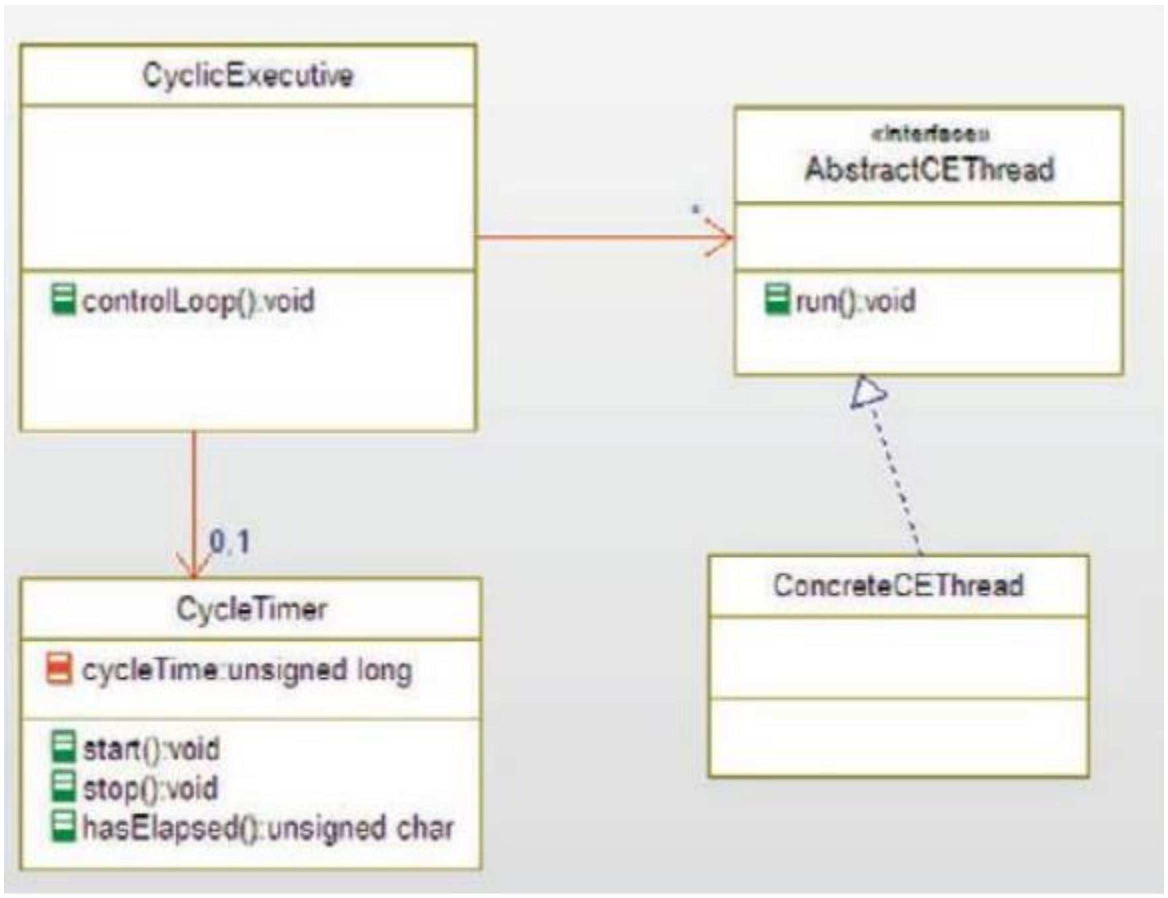
● The Cyclic Executive Pattern provides a low-resource means to accomplish Schedulability for a set of tasks.

- D_j is the deadline for task j
- C_j is the worst-case execution time for task j
- K is the cyclic executive loop overhead, including the overhead of the task invocation and return

for all tasks i, $D_i \geq \sum_{j=1}^{n} C_j + K$

Equation 4-1: Schedulability for Cyclic Executive with *n* tasks

Structure

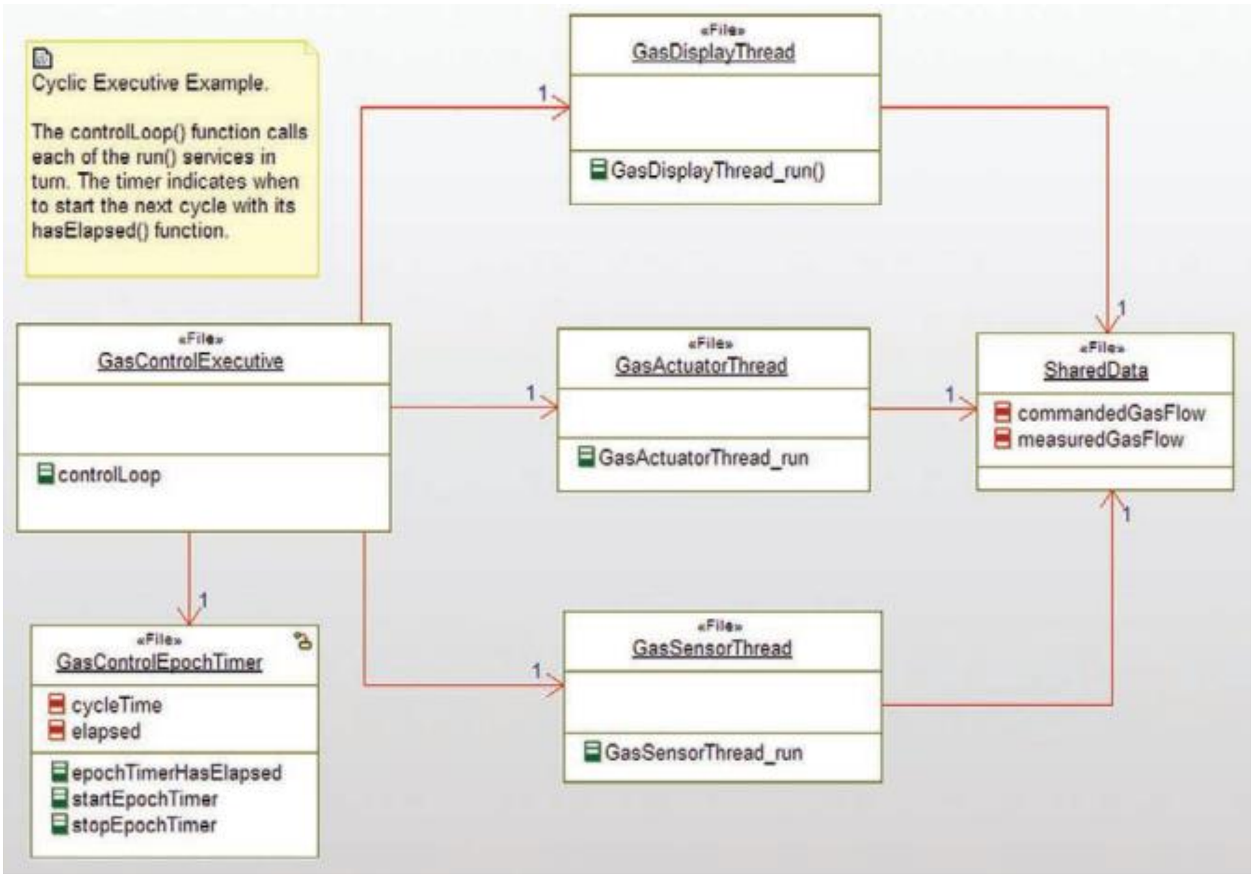


- The AbstractCEThread provides a standard interface for the threads by declaring a run() function.
- It is this function that the Cyclic Executive will invoke to execute the task.
- When this function completes, the CyclicExecutive runs the next task in the list.
- The CyclicExecutive contains the infinite loop that runs the tasks each in turn.
- The CyclicExecutive contains global stack and static data that are needed either by the tasks.
- The time-triggered Cyclic Executive will set up and use the CycleTimer to initiate each epoch (execution of the task list).
- because of the lack of preemption, unbounded blocking is not a concern.

Consequences

- The pattern is less responsive to high-urgency events, and this makes the pattern inappropriate for applications that have high responsiveness requirements.
- A downside of using the pattern is that the interaction of threads is made more challenging than other patterns.
- If an output of one task is needed by another, the data must be retained in global memory or a shared resource until that task runs

Example



● Critical Region Pattern

Abstract

- The critical region pattern used for task coordination around.
- This happens by disabling task switching during a region where it is important that the current task executes without being preempted.
- This can be because it is accessing a resource that cannot be simultaneously accessed safely, or because it is crucial that the task executes for a portion of time without interruption.

Problem

- There are two primary circumstances under which a task should not be interrupted or preempted.
- First, it may be accessing a resource that may not be safely accessed by multiple clients simultaneously.
- Secondly, the task may be performing actions that must be completed within a short period of time or that must be performed in a specific order.
- This pattern allows the active task to execute without any potential interference from other tasks.

Structure

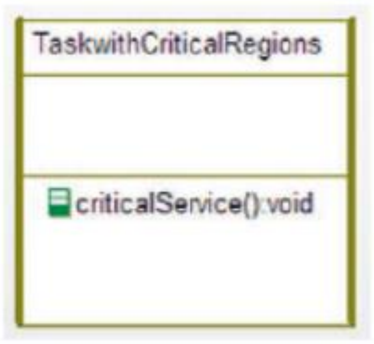
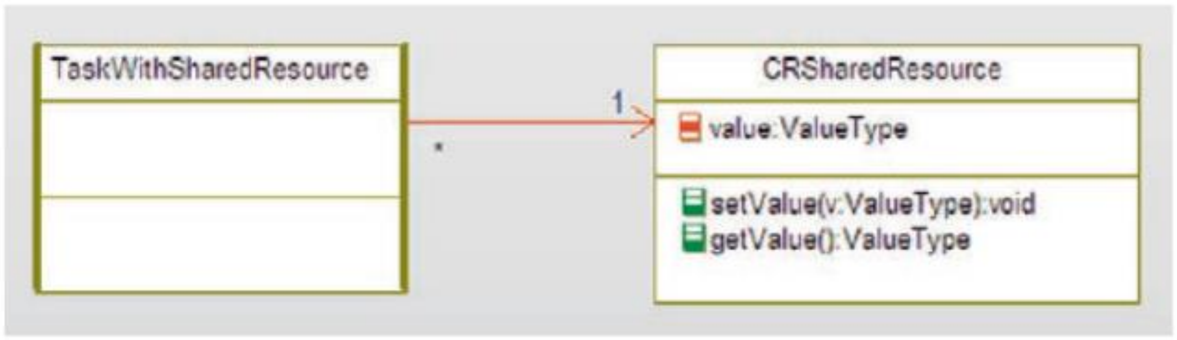


Figure 4-12: Critical Region Pattern



Consequences

- This pattern enforces the critical region policies well and can be used to turn off the scheduled task switches or, disable interrupts all together.
- Care must be taken to reenable task switching once the element is out of its critical region.
- Additionally, the use of this pattern can affect the timing of other tasks. Critical regions are usually short in duration for this reason.
- There is no problem of unbounded priority inversion because all task switching is disabled during the critical region.
- Care must be taken when nesting calls to functions with their own critical regions.

● If such a function is called within the critical region of another, the nest function call will reenable task switching and end the critical region of the caller function.