Embedded Software

Abstract Object-Oriented OS APIs

Agenda

- What is an API and why use it?
- What is an OS API?
- Concrete examples using OS API
- Guidelines for writing event based thread oriented programs

API and OS API - What and Why?

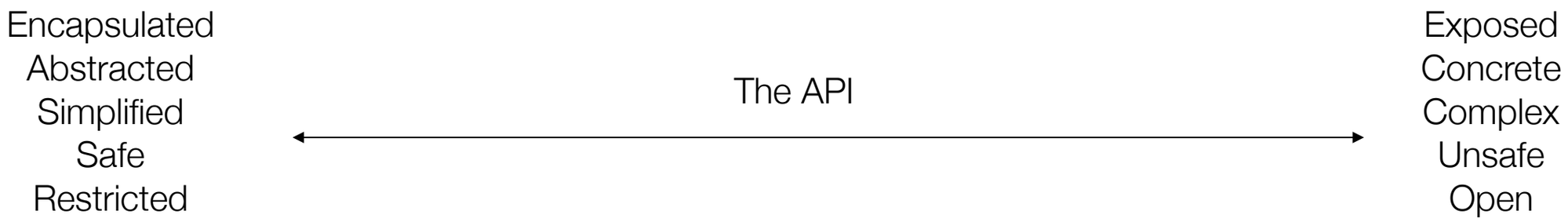
What is an API?

What is an API?

- Why use an API?
  - Encapsulation – the API may hide some of the system
  - Abstraction – only the system interface is revealed
  - Simplification – the API may restrict access to the system

What is an API?

- Why use an API?
  - Encapsulation – the API may hide some of the system
  - Abstraction – only the system interface is revealed
  - Simplification – the API may restrict access to the system



ENGINEERING COLLEGE OF AARHUS

The OS API

- Operating systems have extensive APIs to access OS resources
  - Threads, semaphores, timers, pipes…
  - Example: Thread creation

```
//win32
HANDLE CreateThread(…);
```

```
//POSIX - Linux
void* pthread_create(…);
```

```
//VxWorks
void* pthread_create(…);
```

The abstract OS API

The abstract OS API

- Why is the native OS' API not enough for us
  - why further abstract it?

The abstract OS API

- Why is the native OS' API not enough for us
  - why further abstract it?

- ***What if we wish to switch the OS itself?***

The abstract OS API

- Why is the native OS' API not enough for us
  - why further abstract it?

- ***What if we wish to switch the OS itself?***

Why would we ever want to switch the OS itself?

The abstract OS API

- Why is the native OS' API not enough for us
  - why further abstract it?

- ***What if we wish to switch the OS itself?***

Why would we ever want to switch the OS itself?

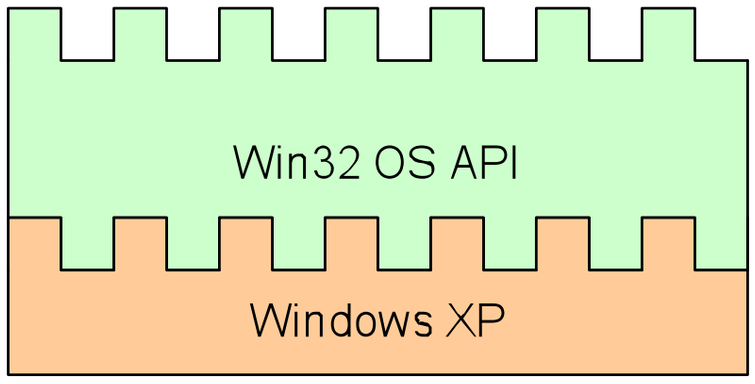
Grp 2 & 2 - 3mins

Your input....

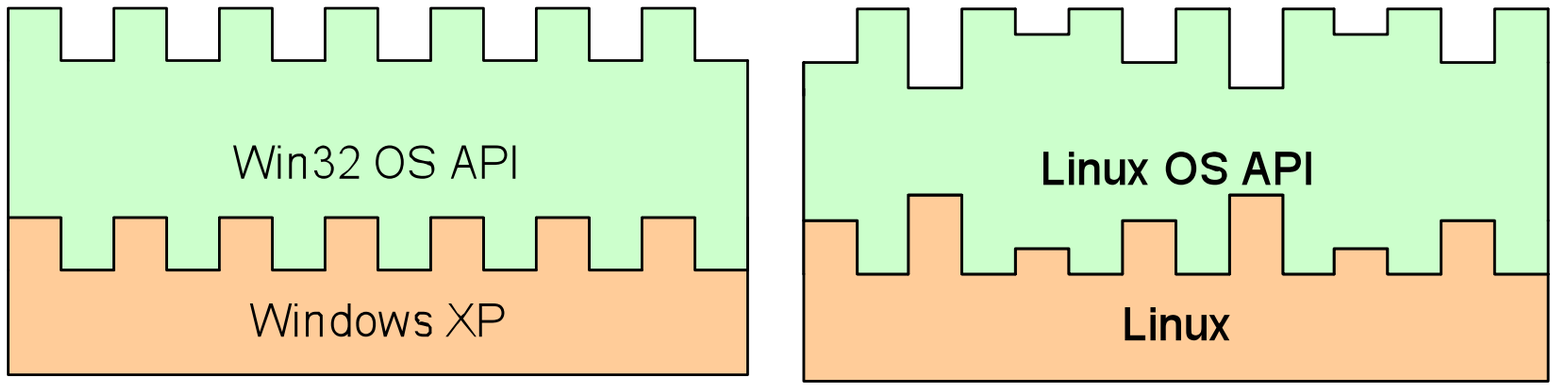
Concrete example - Article on OSAL

- *An Operating System Abstraction Layer for Portable Applications in Wireless Sensor Networks* (for the Mantis OS and FreeRTOS)
  - Why?
    - Faster development due to increase in portability
      - New platforms demand "*only*" implementation of OSAL (and drivers)
    - Support for different OS's deployed on different platforms
    - Same API used again and again - Only one API to learn
  - How?
    - Thin layer introduced between Application layer and OS layer

The abstract OS API

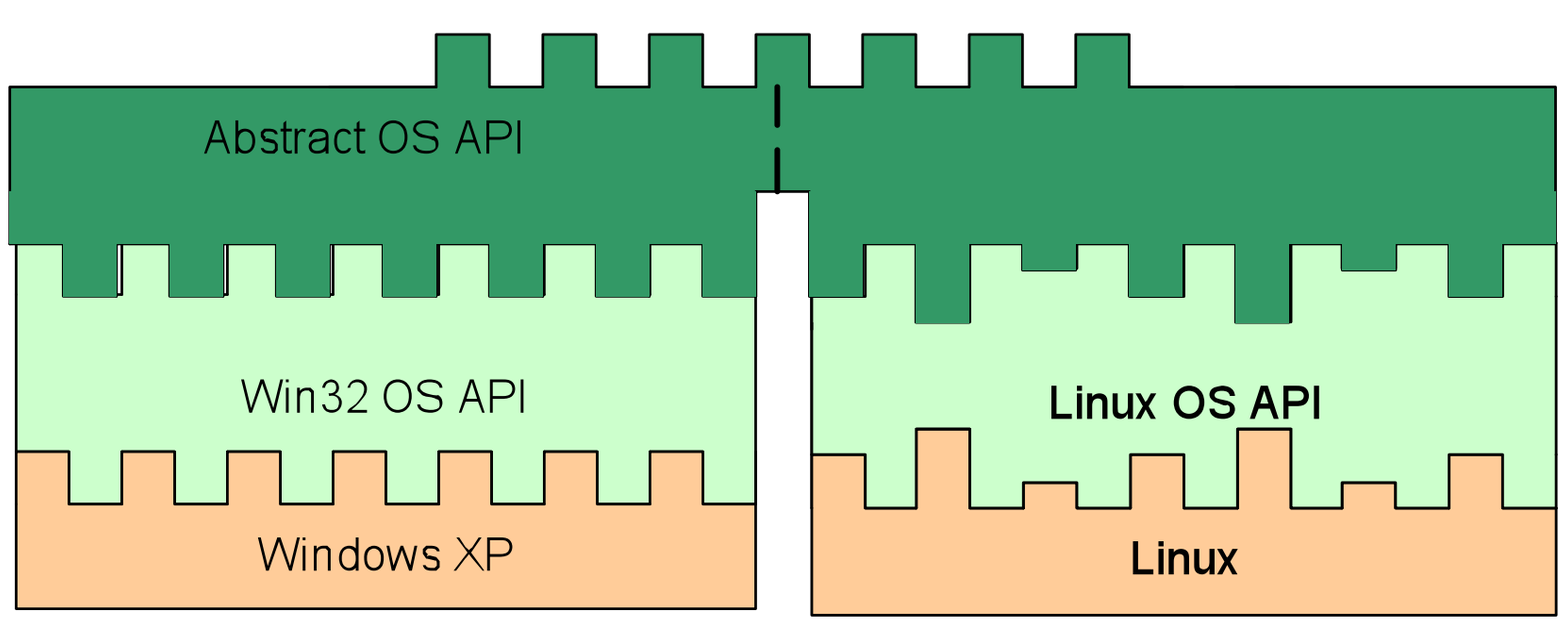


The abstract OS API



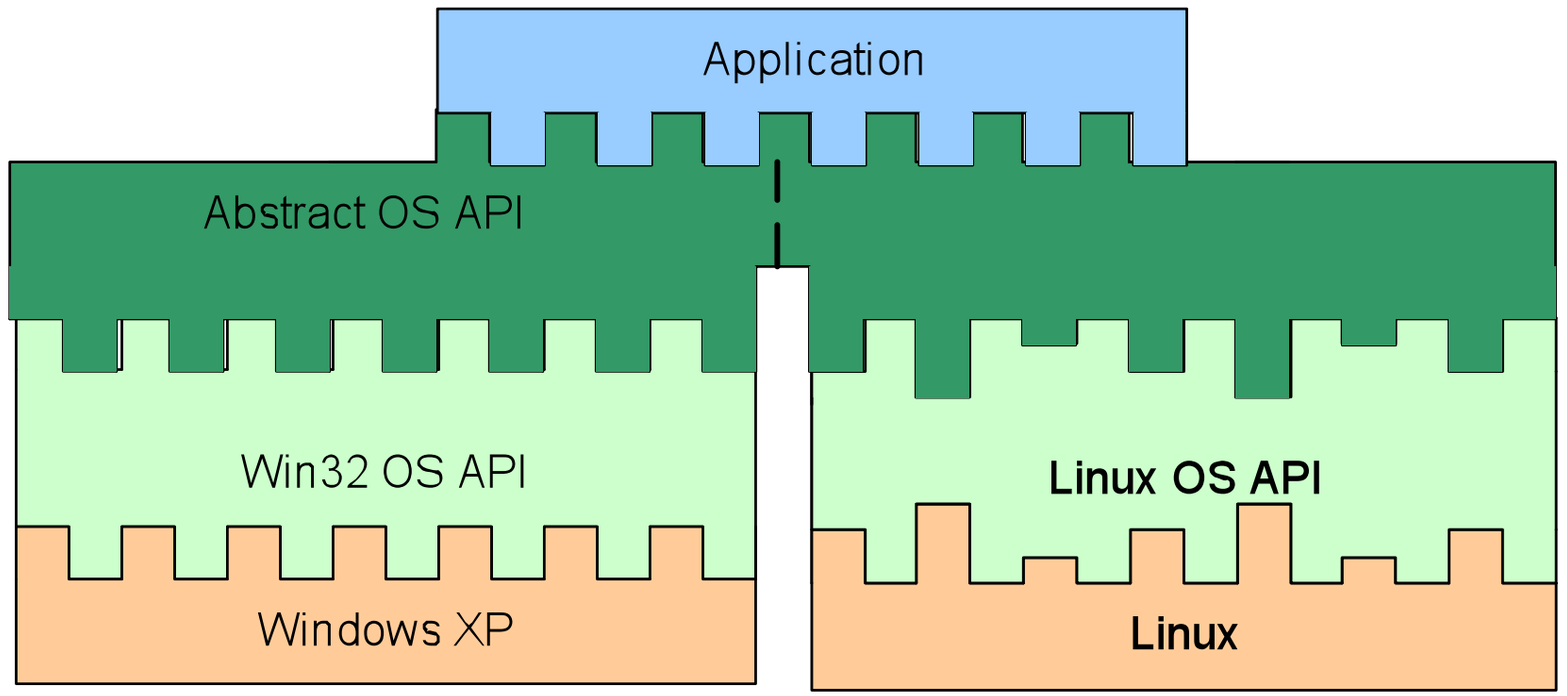
ENGINEERING COLLEGE OF AARHUS

The abstract OS API



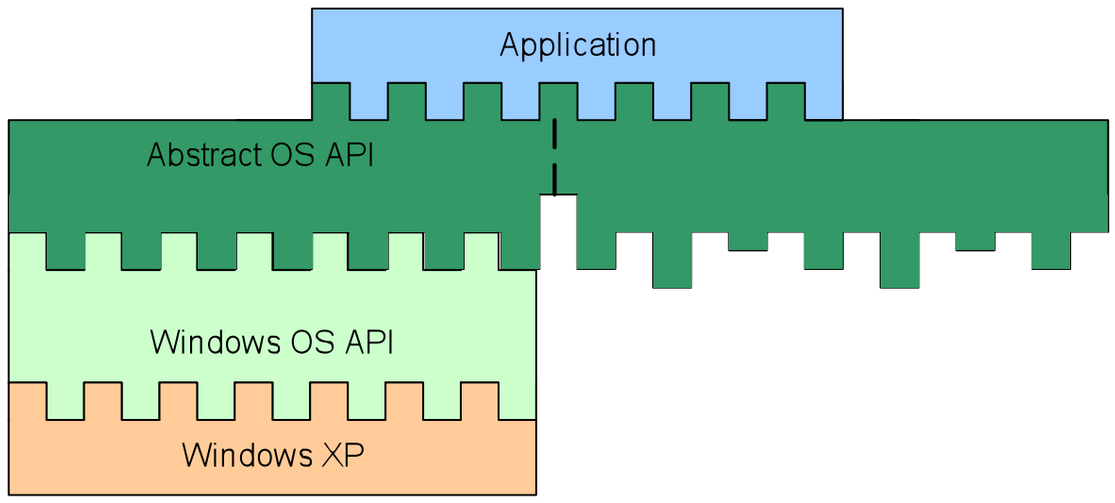
ENGINEERING COLLEGE OF AARHUS

The abstract OS API



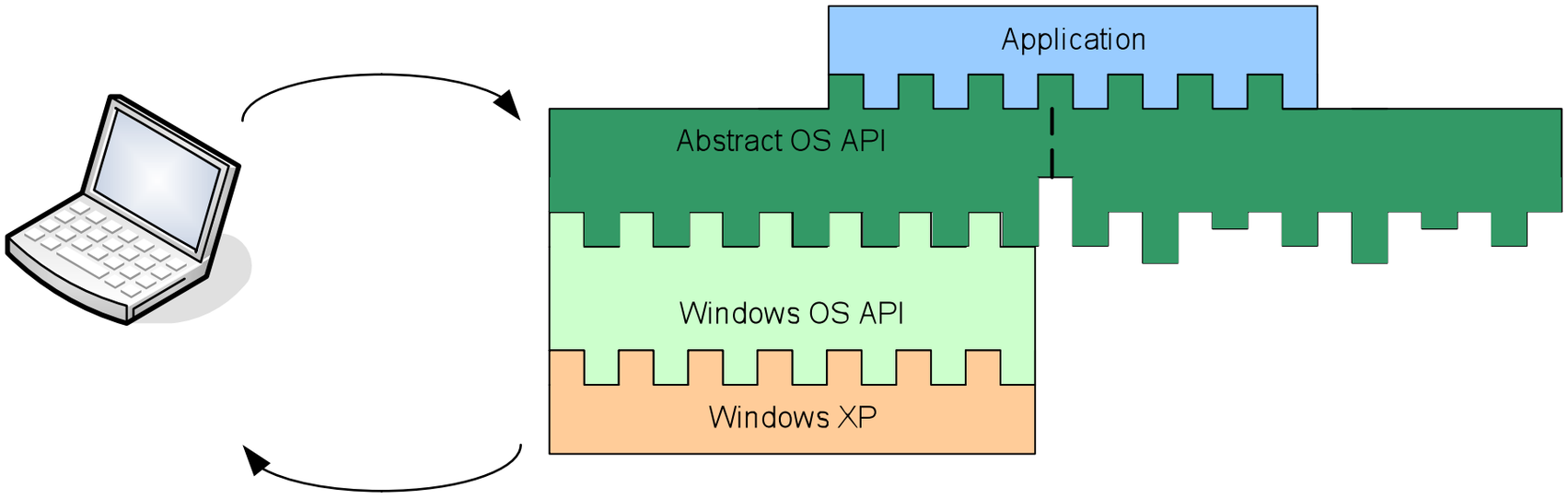
Abstract OS API: Cross-development

- Develop the system for the host platform
  - Debug the system until no errors are left
  - Use stubs for real-life peripherals (GoF Strategy)



Abstract OS API: Cross-development

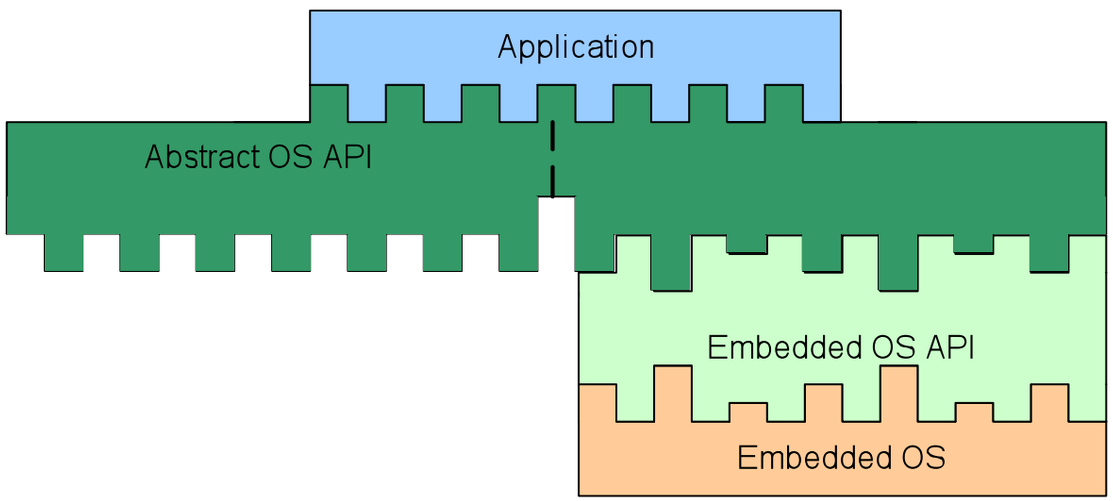
- Develop the system for the host platform
  - Debug the system until no errors are left
  - Use stubs for real-life peripherals (GoF Strategy)



ENGINEERING COLLEGE OF AARHUS

Abstract OS API: Cross-development

- Develop the system for the host platform
  - Debug the system until no errors are left
  - Use stubs for real-life peripherals (GoF Strategy)
- Now develop the same system for target platform
  - Little or no change to application
  - Now debug target-specific problems (timing, real peripherals, etc.)



An abstract object-oriented OS API

An abstract object-oriented OS API

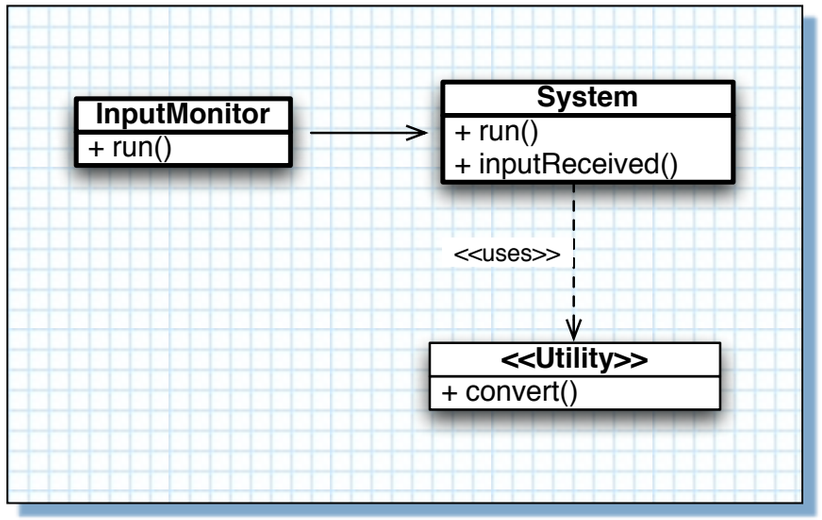
- Why should the abstract OS API be object oriented?
  - Easier to work with (if you're used to objects)
  - Cleaner code
  - Decreases the representational gap between design and implementation

An abstract object-oriented OS API

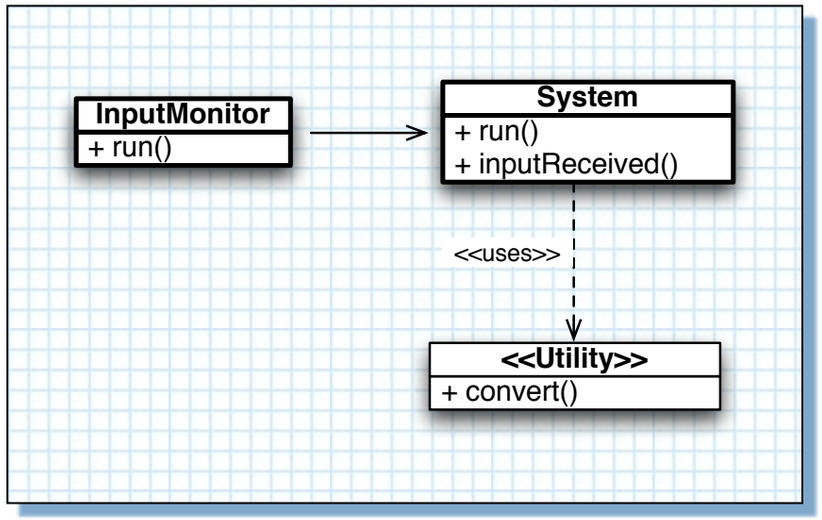
- Why should the abstract OS API be object oriented?
  - Easier to work with (if you're used to objects)
  - Cleaner code
  - Decreases the representational gap between design and implementation

- The representational gap
  - The ”distance in representation” between the design and implementation of your application

The representational gap

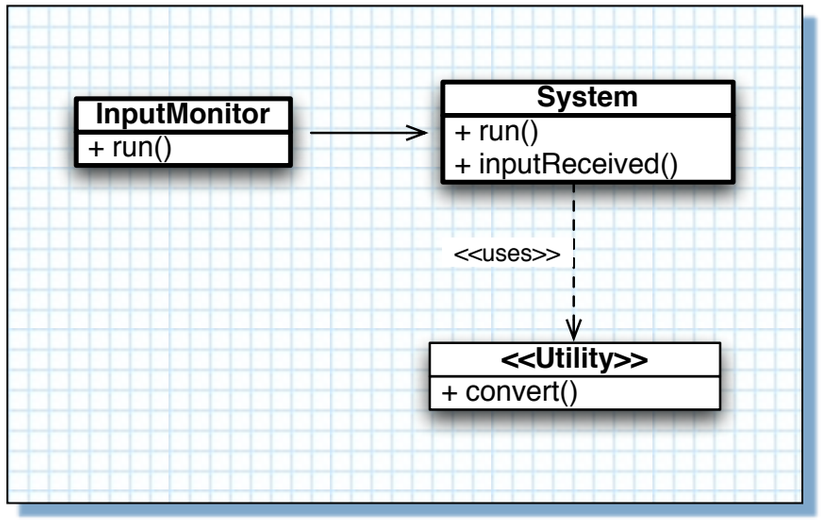


The representational gap




```
// system.h
class System : public Thread
{
public:
    enum { ID_INPUT };
    System() : mq_(MAX_QUEUE_SIZE) {}
    inputReceived(InputMsg* msg);
    MsgQueue mq_;
    ...
};

// system.cpp
void System::run()
{
    unsigned long id;
    Message* msg = mq_.receive(id);
    switch (id) {
            case ID_INPUT:
                        inputReceived(static_cast<InputMsg*> (msg));
                    break;
            default:
                    break;
    }
    delete msg;
}

void System::inputReceived(InputMsg* msg)
{
    convInput = convert(msg->value_);
}
```

The representational gap

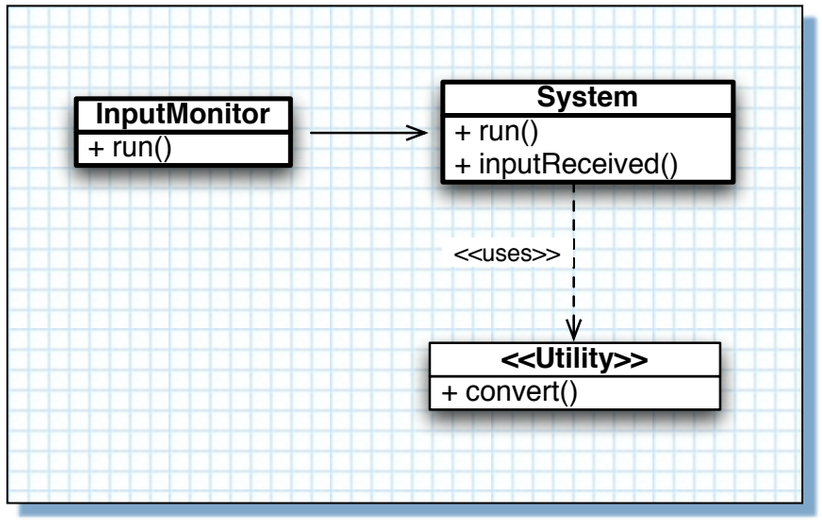



```
// inputmonitor.h
class InputMonitor
{
public:
    InputMonitor(MsgQueue* sysMsgQueue);
    ...
};


// inputmonitor.cpp
...
InputMonitor::inputReady()
{
    value = gpio_read_16bit(0x22);
    ...
    sysMsgQueue_->send(System::ID_INPUT, inputMsg);
}
```

```
// system.h
class System : public Thread
{
public:
    enum { ID_INPUT };
    System() : mq_(MAX_QUEUE_SIZE) {}
    inputReceived(InputMsg* msg);
    MsgQueue mq_;
    ...
};

// system.cpp
void System::run()
{
    unsigned long id;
    Message* msg = mq_.receive(id);
    switch (id) {
            case ID_INPUT:
                        inputReceived(static_cast<InputMsg*> (msg));
                    break;
            default:
                    break;
    }
    delete msg;
}

void System::inputReceived(InputMsg* msg)
{
    convInput = convert(msg->value_);
}
```


ENGINEERING COLLEGE OF AARHUS

The representational gap




```
// inputmonitor.h
class InputMonitor
{
public:
    InputMonitor(MsgQueue* sysMs
    ...
};


// inputmonitor.cpp
...
InputMonitor::inputReady()
{
    value = gpio_read_16bit(0x22);
    ...
    sysMsgQueue_->send(System::ID_INPUT, inputMsg);
}
```

```
// main.cpp
void main()
{
    System sys;
    InputMonitor inputMonitor(sys.getMsgQueue());
    sys.start();
    inputMonitor.start();

    while(true) sleep(1000);
}
```

```
// system.h
class System : public Thread
{
public:
    enum { ID_INPUT };
    System() : mq_(MAX_QUEUE_SIZE) {}
    inputReceived(InputMsg* msg);
    MsgQueue mq_;
    ...
};

// system.cpp
void System::run()
                                    ve(id);

                                    nputReceived(static_cast<InputMsg*> (msg));
    delete msg;
}

void System::inputReceived(InputMsg* msg)
{
    convInput = convert(msg->value_);
}
```


ENGINEERING COLLEGE OF AARHUS

Usage & Guidelines

OO OS Api - Example

- Simple example
  - `MyThread` inherits and implements function *run* from `Thread`
  - `osapi::Mutex` is part of `MyThread` and is default appropriately initialized
  - `MyThread` is created on the stack in function `main()`
    - Started via `start()`
    - Waited upon via `join()`

```
class MyThread : public osapi::Thread
{
public:
    MyThread() : running_(true) {}
    virtual void run()
    {
        while (running_) {
            m_.lock();
            // Do stuff
            m_.unlock();
            // Do stuff
        }
    }
private:
    bool          running_;
    osapi::Mutex m_;
};
```

```
int main(int argc, char *argv[])
{
    MyThread myt;
    myt.start();

    myt.join();
}
```

Typical task structure in event-based system

```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Typical task structure in event-based system

Perform setup here that does not belong in constructor

```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Typical task structure in event-based system

```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Perform setup here that does not belong in constructor

Get a message, e.g. msgQueue->receive()

Typical task structure in event-based system

```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Perform setup here that does not belong in constructor

Get a message, e.g. msgQueue->receive()

Handle each ”type” of message separately

Typical task structure in event-based system

```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Perform setup here that does not belong in constructor

Get a message, e.g. msgQueue->receive()

Handle each ”type” of message separately

Signal design error: Thread received something it did not expect

Typical task structure in event-based system

```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Perform setup here that does not belong in constructor

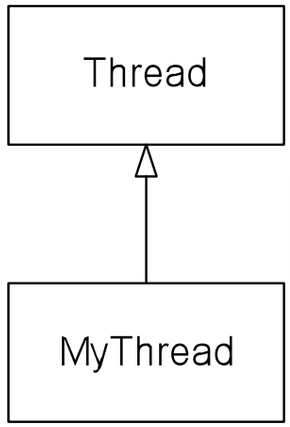
Get a message, e.g. msgQueue->receive()

Handle each ”type” of message separately

Signal design error: Thread received something it did not expect

Execution should never reach this point

Typical task structure in event-based system




```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Perform setup here that does not belong in constructor

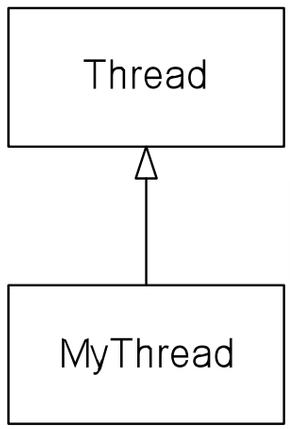
Get a message, e.g. msgQueue->receive()

Handle each ”type” of message separately

Signal design error: Thread received something it did not expect

Execution should never reach this point

Typical task structure in event-based system




```
void MyThread::run()
{
   // get message from message queue
   while(running_)
   {
        switch (on state) {
            case ST_IDLE:

                switch (on event) {
                    case ID_MSG:
                        // Handle event.
                        break;
                    default:
                        break;
                }

                break;
            default:
                break;
        }
   }

}
```

Perform setup here that does not belong in constructor

Get a message, e.g. msgQueue->receive()

Switch on state, each state has separate handlers

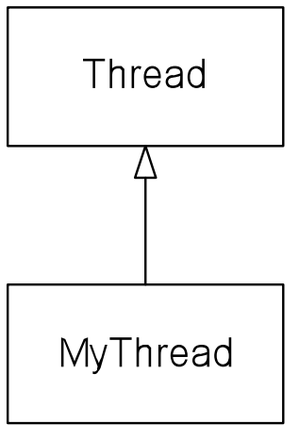
Handle each ”type” of message separately

Signal design error: Thread received something it did not expect

Execution should never reach this point

OS Api used with MsgQueues

- Thread class using the MsgQueue concept

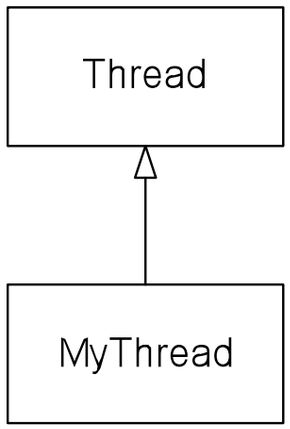



```
class MyThread : public osapi::Thread
{
public:
    enum MsgID {
        ID_MSG,
        ID_TERMINATE
    }
    // Other functions
    MyThread();
private:
    void handleMsg(unsigned int id, osapi::Message* msg);
    virtual void run();

    enum State {
        ST_IDLE,
        ST_RUNNING
    };

    osapi::MsgQueue mq_;
    State           st_;
};
```


ENGINEERING COLLEGE OF AARHUS

OS Api used with MsgQueues

- Thread class using the MsgQueue concept

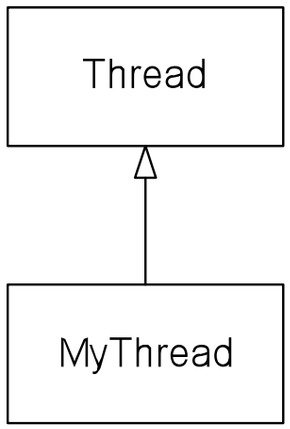


Messages that can be received = Class Interface

```
class MyThread : public osapi::Thread
{
public:
    enum MsgID {
        ID_MSG,
        ID_TERMINATE
    }
    // Other functions
    MyThread();
private:
    void handleMsg(unsigned int id, osapi::Message* msg);
    virtual void run();

    enum State {
        ST_IDLE,
        ST_RUNNING
    };

    osapi::MsgQueue mq_;
    State           st_;
};
```

OS Api used with MsgQueues

- Thread class using the MsgQueue concept



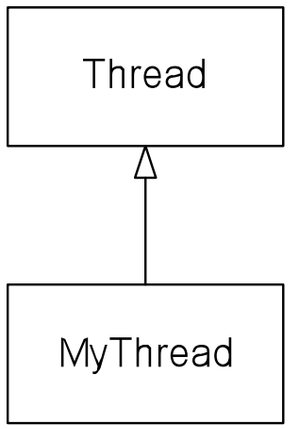
Messages that can be received = Class Interface

Thread function and message handler

```
class MyThread : public osapi::Thread
{
public:
    enum MsgID {
        ID_MSG,
        ID_TERMINATE
    }
    // Other functions
    MyThread();
private:
    void handleMsg(unsigned int id, osapi::Message* msg);
    virtual void run();

    enum State {
        ST_IDLE,
        ST_RUNNING
    };

    osapi::MsgQueue mq_;
    State           st_;
};
```

OS Api used with MsgQueues

- Thread class using the MsgQueue concept



Messages that can be received = Class Interface

Thread function and message handler

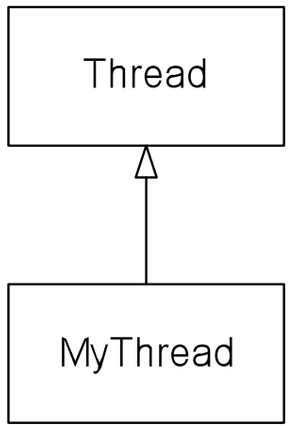
Thread states

```
class MyThread : public osapi::Thread
{
public:
    enum MsgID {
        ID_MSG,
        ID_TERMINATE
    }
    // Other functions
    MyThread();
private:
    void handleMsg(unsigned int id, osapi::Message* msg);
    virtual void run();

    enum State {
        ST_IDLE,
        ST_RUNNING
    };

    osapi::MsgQueue mq_;
    State           st_;
};
```


ENGINEERING COLLEGE OF AARHUS

Typical task structure in event-based system

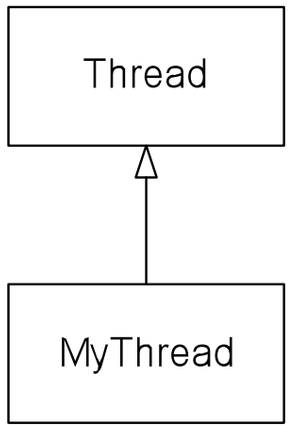
Typical task structure in event-based system




```
void MyThread::run()
{
    // Initial one-time setup
    while (running_) {
        {
            // Wait for message (e.g. mail)
            unsigned int id;
            osapi::Message* msg = mq_.receive(id);
            handleMsg(id, msg);
            delete msg;
        }
    }
}
```

Thread loop

Typical task structure in event-based system



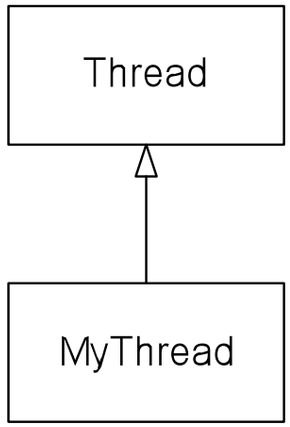

```
void MyThread::run()
{
    // Initial one-time setup
    while (running_) {
        {
            // Wait for message (e.g. mail)
            unsigned int id;
            osapi::Message* msg = mq_.receive(id);
            handleMsg(id, msg);
            delete msg;
        }
    }
}
```

Thread loop

```
void MyThread::handleMsg(unsigned int id,
                         osapi::Message* msg)
{
    switch (state) {
        case ST_IDLE:
            handleMsgStateIdle(id, msg);
            break;
        default:
            break;
    }
}
```

Switch on state and handle it

Typical task structure in event-based system




```
void MyThread::run()
{
    // Initial one-time setup
    while (running_) {
        {
            // Wait for message (e.g. mail)
            unsigned int id;
            osapi::Message* msg = mq_.receive(id);
            handleMsg(id, msg);
            delete msg;
        }
    }
}
```

Thread loop

```
void MyThread::handleMsg(unsigned int id,
                         osapi::Message* msg)
{
    switch (state) {
        case ST_IDLE:
            handleMsgStateIdle(id, msg);
            break;
        default:
            break;
    }
}
```

Switch on state and handle it

Switch on event and handle it

```
void MyThread::handleMsgStIdle(unsigned int id,
                               osapi::Message* msg)
{
    switch(event)
    {
        case ID_MSG:
            // handle firstType-messages
            handleStIdleIdMsg(msg);
            break;

        case ID_TERMINATE:
            // handle secondType-messages
            break;

            ...

        default:
            // signal an error
            break;
    }

}
```

The abstract OO OS API

- The OS API includes the following resources:
  - An abstract Thread class
  - sleep
  - A Timer class (for timeouts)
  - A Time class (simple time arithmetic)
  - Semaphore class (counting)
  - Mutex class
  - Conditional class
  - A ScopedLock class
  - A Completion class
  - A Log System
  - A Message Queue class
- Use (or extend) this to build generic, object-oriented applications