OS API

AGENDA

- What is an API?
  - Design considerations
- OSAL - OS Abstraction Layer
- Why OO?
- Using OO OS Api - A case
- From posix threads to OO OS Api threads
- Guidelines

WHAT IS AN API?

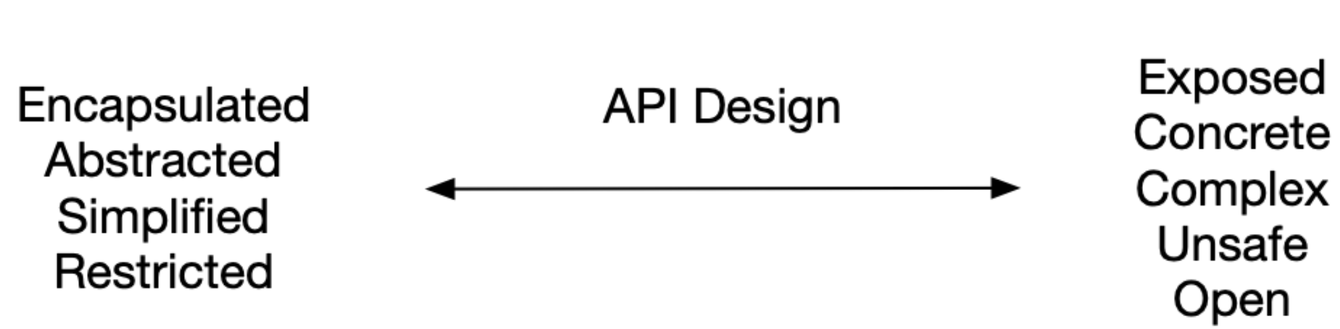
WHAT IS AN API?

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

DESIGN PLACEMENT

- API Design triggers conflicting ideals
  - Question: Where to be on the *scale*???



OSAL - OS ABSTRACTION LAYER

OSAL - OS ABSTRACTION LAYER

- What is an OSAL and why use it
- Encapsulation example
- Cross-development
- What should/could it cover

WHAT IS AN OSAL AND WHY USE IT

DIFFERENT OSES HAVE DIFFERENT APPROACHES (APIS) TO

- Create of threads
- Manipulate mutexes
- Allocate memory
- Open files
- Etc.

WHAT IS AN OSAL AND WHY USE IT

DIFFERENT OSES HAVE DIFFERENT APPROACHES (APIS) TO

- Create of threads
- Manipulate mutexes
- Allocate memory
- Open files
- Etc.

CREATING THREADS IN DIFFERENT OSES

```
01  //win32
02  HANDLE CreateThread(...);
```

```
01  //POSIX - Linux
02  void* pthread_create(...);
```

```
01  //VxWorks
02  void* pthread_create(...);
```

```
01  //FreeRTOS
02  portBASE_TYPE xTaskCreate(...);
```

WHAT IS AN OSAL AND WHY USE IT

- An OSAL is a library (might be framework) that abstracts
  - different OS concepts
  - presents a unified API independent of the underlying OS

WHAT IS AN OSAL AND WHY USE IT

- An OSAL is a library (might be framework) that abstracts
  - different OS concepts
  - presents a unified API independent of the underlying OS

- Consequence
  - Only need to learn ***ONE*** API to
    - Create threads
    - Manipulate mutexes
    - etc.

WHAT IS AN OSAL AND WHY USE IT

ARTICLE DISCUSSING OSAL

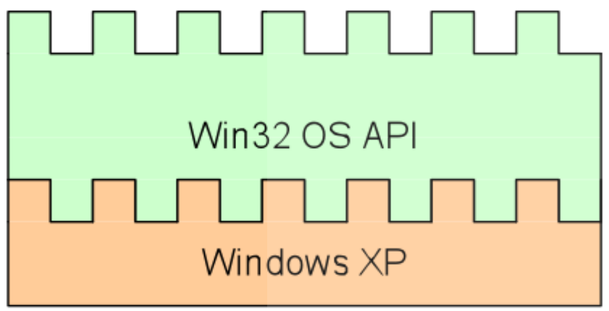
- An Operating System Abstraction Layer for Portable Applications in Wireless Sensor Networks (for the Mantis OS and FreeRTOS)
  - Why?
    - Faster development due to increase in portability
      - New platforms demand “only” implementation of OSAL (and drivers)
    - Support for different OS’s deployed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

WHAT IS AN OSAL AND WHY USE IT

- Exemplifying the concept
  - Remember: *Thin layer that encapsulates real OS API*

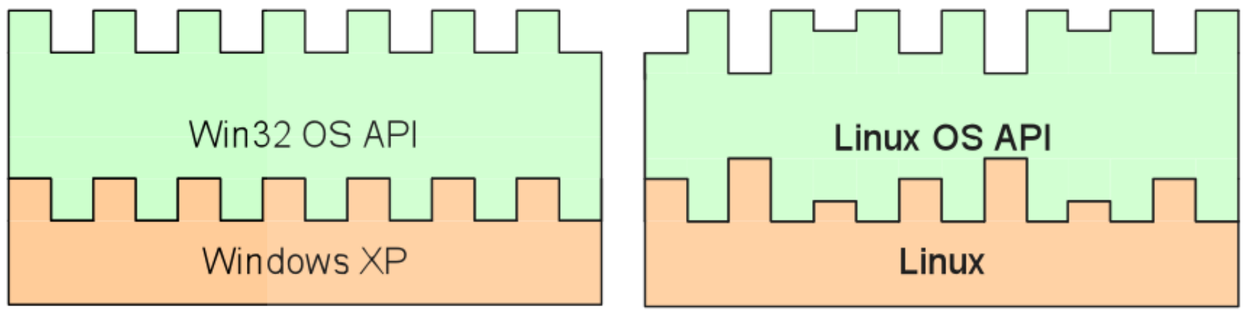
WHAT IS AN OSAL AND WHY USE IT

- Exemplifying the concept
  - Remember: *Thin layer that encapsulates real OS API*



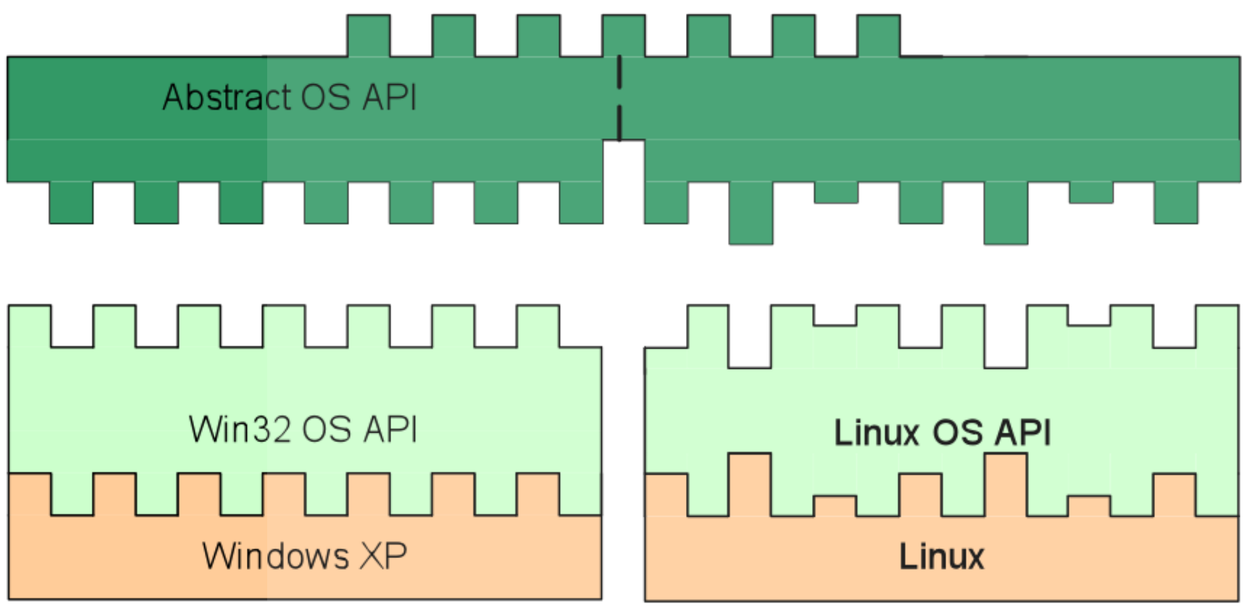
WHAT IS AN OSAL AND WHY USE IT

- Exemplifying the concept
  - Remember: *Thin layer that encapsulates real OS API*



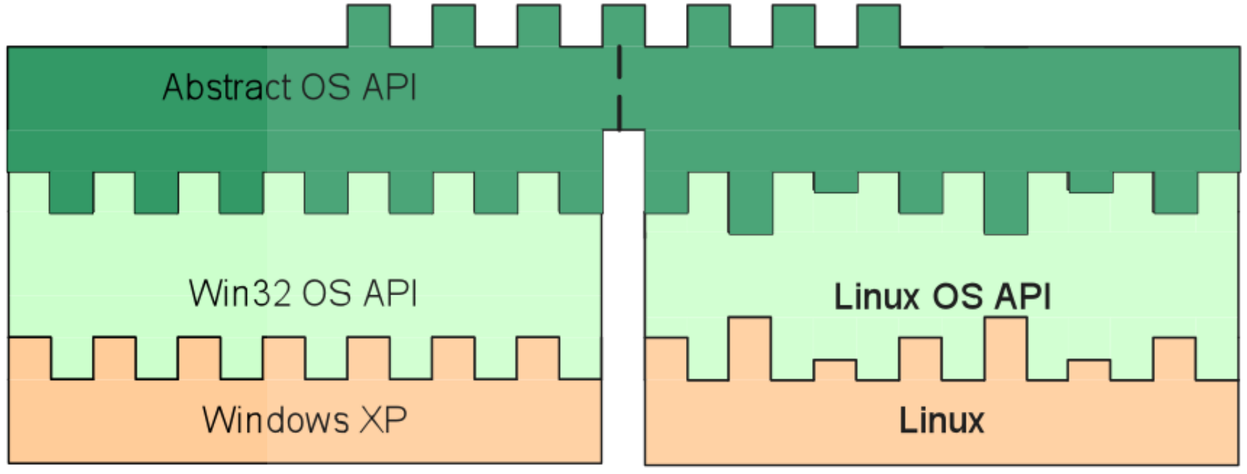
WHAT IS AN OSAL AND WHY USE IT

- Exemplifying the concept
  - Remember: *Thin layer that encapsulates real OS API*



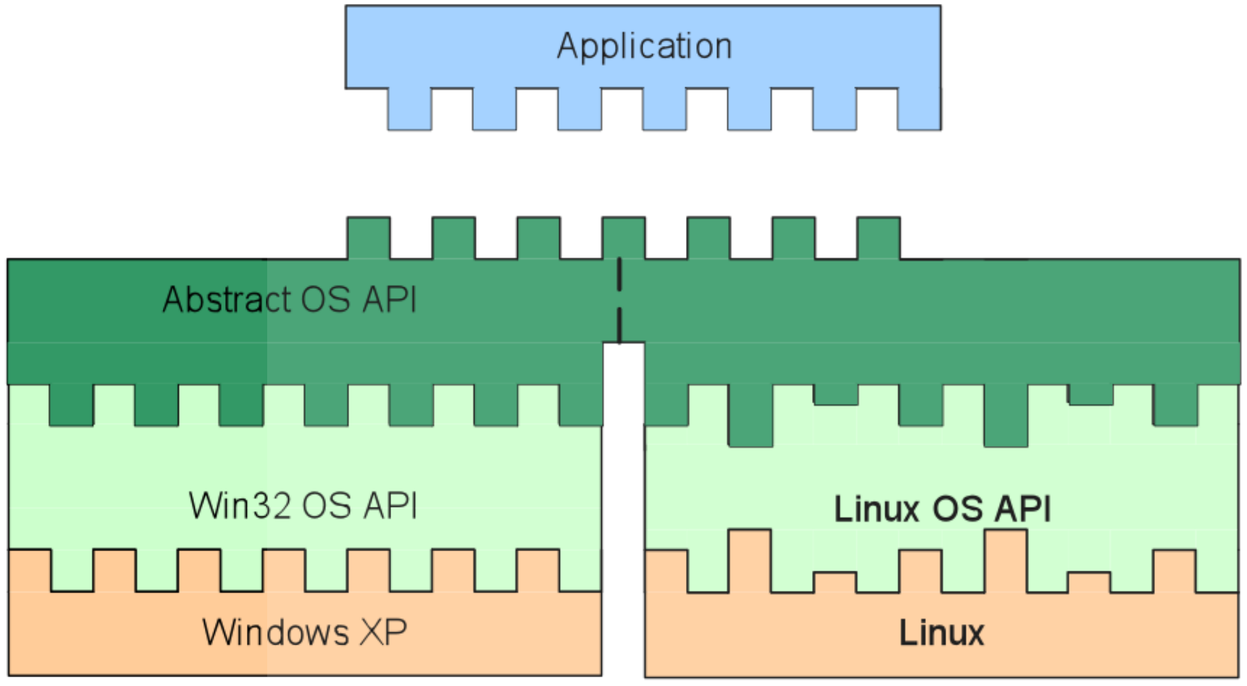
WHAT IS AN OSAL AND WHY USE IT

- Exemplifying the concept
  - Remember: *Thin layer that encapsulates real OS API*



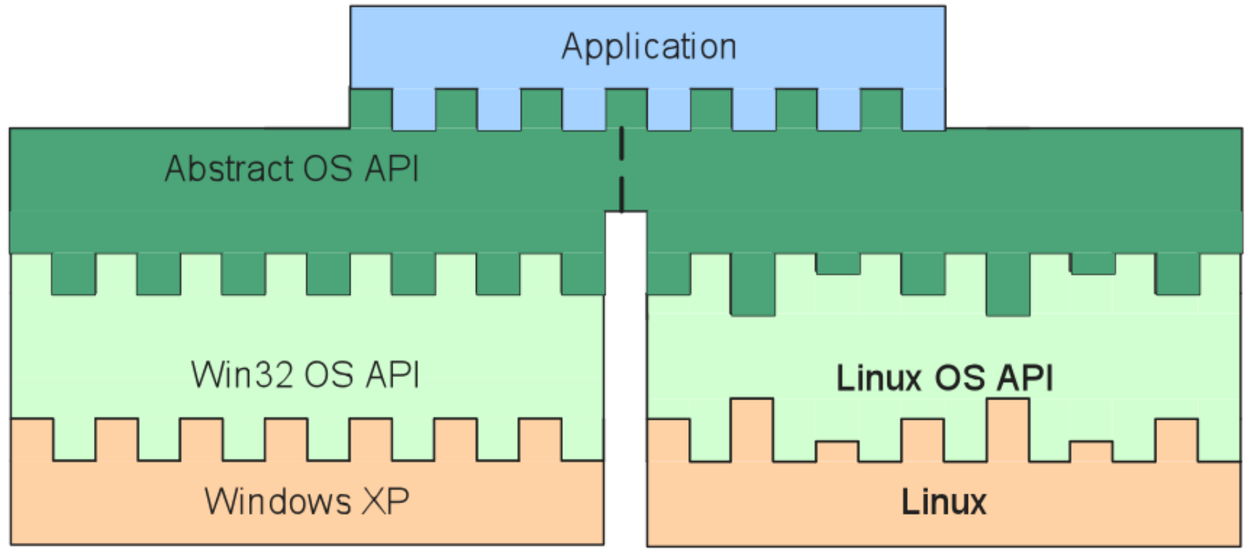
WHAT IS AN OSAL AND WHY USE IT

- Exemplifying the concept
  - Remember: *Thin layer that encapsulates real OS API*



WHAT IS AN OSAL AND WHY USE IT

- Exemplifying the concept
  - Remember: *Thin layer that encapsulates real OS API*



ENCAPSULATION EXAMPLE

SEMAPHORE IMPLEMENTATION IN LINUX

```cpp
// inc/osapi/linux/Semaphore.hpp
#include <semaphore.h>
#include <osapi/Utility.hpp>

namespace osapi
{
  class Semaphore : Notcopyable
  {
  public:
    Semaphore(unsigned int initCount);
    void wait();
    void signal();
    ~Semaphore();
  private:
    sem_t semId_;
  };
}
```

```cpp
// linux/Semaphore.cpp
#include <osapi/Semaphore.hpp>

namespace osapi
{
  Semaphore::Semaphore(unsigned int initCount)
  {
    if(sem_init(&semId_, 1, initCount) != 0)
      throw SemaphoreError();
  }

  void Semaphore::wait()
  {
    if(sem_wait(&semId_) != 0) throw SemaphoreError();
  }

  void Semaphore::signal()
  {
    if(sem_post(&semId_) != 0) throw SemaphoreError();
  }

  Semaphore::~Semaphore()
  {
    sem_destroy(&semId_);
  }

}
```

ENCAPSULATION EXAMPLE

SEMAPHORE IMPLEMENTATION IN LINUX

```
01  // inc/osapi/linux/Semaphore.hpp
02  #include <semaphore.h>
03  #include <osapi/Utility.hpp>
04
05  namespace osapi
06  {
07    class Semaphore : Notcopyable
08    {
09    public:
10      Semaphore(unsigned int initCount);
11      void wait();
12      void signal();
13      ~Semaphore();
14    private:
15      sem_t semId_;
16    };
17  }
```

- In namespace to signify cohesion

```
01  // linux/Semaphore.cpp
02  #include <osapi/Semaphore.hpp>
03
04  namespace osapi
05  {
06    Semaphore::Semaphore(unsigned int initCount)
07    {
08      if(sem_init(&semId_, 1, initCount) != 0)
09        throw SemaphoreError();
10    }
11
12    void Semaphore::wait()
13    {
14      if(sem_wait(&semId_) != 0) throw SemaphoreError();
15    }
16
17    void Semaphore::signal()
18    {
19      if(sem_post(&semId_) != 0) throw SemaphoreError();
20    }
21
22    Semaphore::~Semaphore()
23    {
24      sem_destroy(&semId_);
25    }
26
27  }
```

ENCAPSULATION EXAMPLE

SEMAPHORE IMPLEMENTATION IN LINUX

```cpp
// inc/osapi/linux/Semaphore.hpp
#include <semaphore.h>
#include <osapi/Utility.hpp>

namespace osapi
{
  class Semaphore : Notcopyable
  {
  public:
    Semaphore(unsigned int initCount);
    void wait();
    void signal();
    ~Semaphore();
  private:
    sem_t semId_;
  };
}
```

- Abstracted interface for a `Semaphore`
- All `Semaphore` *must* have this interface
  - Albeit the implementation may differ

```cpp
// linux/Semaphore.cpp
#include <osapi/Semaphore.hpp>

namespace osapi
{
  Semaphore::Semaphore(unsigned int initCount)
  {
    if(sem_init(&semId_, 1, initCount) != 0)
      throw SemaphoreError();
  }

  void Semaphore::wait()
  {
    if(sem_wait(&semId_) != 0) throw SemaphoreError();
  }

  void Semaphore::signal()
  {
    if(sem_post(&semId_) != 0) throw SemaphoreError();
  }

  Semaphore::~Semaphore()
  {
    sem_destroy(&semId_);
  }

}
```

ENCAPSULATION EXAMPLE

SEMAPHORE IMPLEMENTATION IN LINUX

```cpp
// inc/osapi/linux/Semaphore.hpp
#include <semaphore.h>
#include <osapi/Utility.hpp>

namespace osapi
{
  class Semaphore : Notcopyable
  {
  public:
    Semaphore(unsigned int initCount);
    void wait();
    void signal();
    ~Semaphore();
  private:
    sem_t semId_;
  };
}
```

- Single simply focus on implementing `wait()`
- Linux implementation of `wait()` called `sem_wait(...)`

```cpp
// linux/Semaphore.cpp
#include <osapi/Semaphore.hpp>

namespace osapi
{
  Semaphore::Semaphore(unsigned int initCount)
  {
    if(sem_init(&semId_, 1, initCount) != 0)
      throw SemaphoreError();
  }

  void Semaphore::wait()
  {
    if(sem_wait(&semId_) != 0) throw SemaphoreError();
  }

  void Semaphore::signal()
  {
    if(sem_post(&semId_) != 0) throw SemaphoreError();
  }

  Semaphore::~Semaphore()
  {
    sem_destroy(&semId_);
  }

}
```

CROSS-DEVELOPMENT

OTHER BENEFITS

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

CROSS-DEVELOPMENT

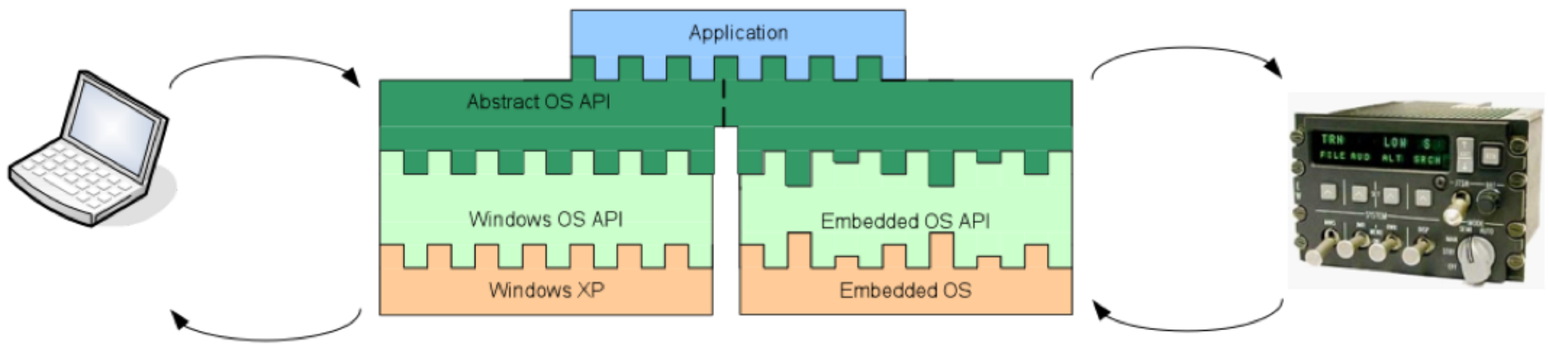
OTHER BENEFITS

- 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.)

CROSS-DEVELOPMENT

OTHER BENEFITS

- 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.)



x

WHAT SHOULD/COULD IT COVER

BASIC SYSTEM FUNCTIONALITY CONSIDERED IMPORTANT IN THE USES DEEMED IMPORTANT (INSPIRED BY FREERTOS API)

- Threads
- Mutexes/Semaphores
- Conditionals
- Time functions
- Message Queues
- Timers
- Input (keyboard)
- External connection handling such as TCP/IP etc.
- Further requirements are more than feasible, this is but a mere start
- Depends on the usage needs

WHAT OO OS API DOES COVER!

- An abstract ThreadFunctor & Thread class
  - For handling threads
  - 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

WHY OO?

WHY OO?

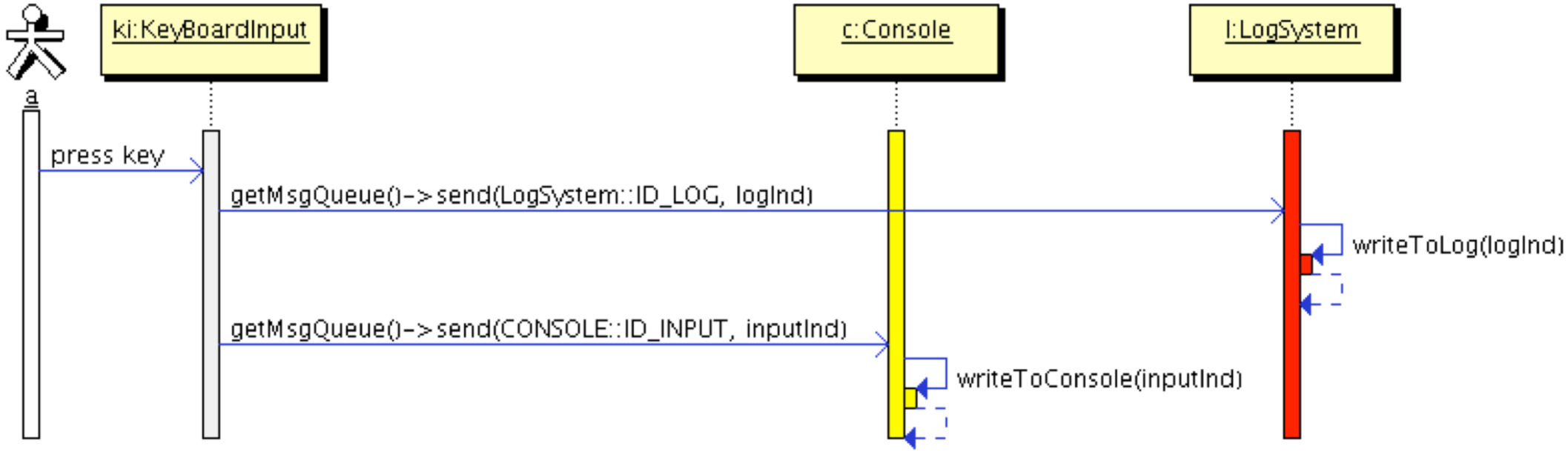
- Why should the abstract OS API be object oriented?
  - Easier to work with (if you're used to objects)
  - Cleaner code
  - Decreases the representational gap between design and implementation
- The representational gap
  - The "distance in representation" between the design and implementation of your application
  - Diagrams being sequences, classes etc. have meaning full representation in C++

USING OO OS API - A CASE

USING OO OS API - A CASE

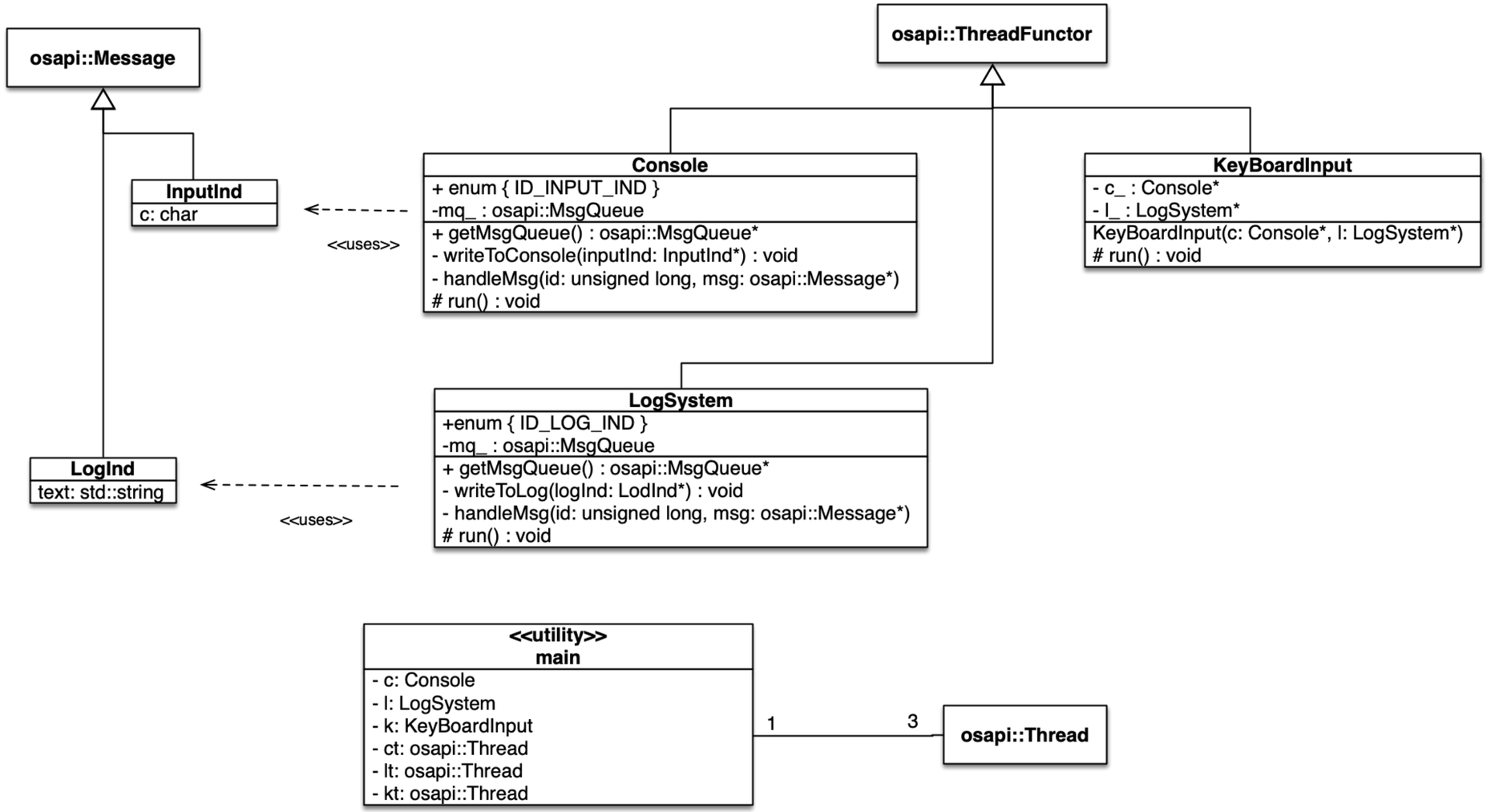
- I want a simple program that can (and is based on the OS Api)
  - Read keyboard input from stdin (thread)
    - Send it as an event/message to another thread
  - Write it out to a log file (thread)
    - Receive said event/message
  - Print it out to console (thread) *
    - *Used in design, but not implemented*

SEQUENCE DIAGRAM



- Keypress (actually a complete string)
  - Retrieve message queue from `LogSystem`
  - Send message `LogSystem::ID_LOG` containing string to `LogSystems` message queue
  - `LogSystem` receives message and writes the contents to the log file

CLASS DIAGRAM



IMPLEMENTATION OF MAIN.CPP

```cpp
// main.cpp
#include <osapi/Thread.hpp>
//#include <Console.hpp>
#include <osapi/example/LogSystem.hpp>
#include <osapi/example/KeyBoardInput.hpp>

int main()
{
    //Console c;
    LogSystem l;
    KeyBoardInput k(&l);

    //osapi::Thread ct(&c);
    osapi::Thread lt(&l);
    lt.start();
    osapi::Thread kt(&k);
    kt.start();

    //ct.join();
    lt.join();
    kt.join();

}
```

IMPLEMENTATION OF MAIN.CPP

- Various includes
  - Note that `osapi/Thread.hpp` is include

```
01  // main.cpp
02  #include <osapi/Thread.hpp>
03  //#include <Console.hpp>
04  #include <osapi/example/LogSystem.hpp>
05  #include <osapi/example/KeyBoardInput.hpp>
06
07  int main()
08  {
09      //Console c;
10      LogSystem l;
11      KeyBoardInput k(&l);
12
13      //osapi::Thread ct(&c);
14      osapi::Thread lt(&l);
15      lt.start();
16      osapi::Thread kt(&k);
17      kt.start();
18
19      //ct.join();
20      lt.join();
21      kt.join();
22
23  }
```

IMPLEMENTATION OF MAIN.CPP

- Various includes
  - Note that `osapi/Thread.hpp` is include
- Instantiate `LogSystem`

```cpp
// main.cpp
#include <osapi/Thread.hpp>
//#include <Console.hpp>
#include <osapi/example/LogSystem.hpp>
#include <osapi/example/KeyBoardInput.hpp>

int main()
{
    //Console c;
    LogSystem l;
    KeyBoardInput k(&l);

    //osapi::Thread ct(&c);
    osapi::Thread lt(&l);
    lt.start();
    osapi::Thread kt(&k);
    kt.start();

    //ct.join();
    lt.join();
    kt.join();

}
```

IMPLEMENTATION OF MAIN.CPP

- Various includes
  - Note that `osapi/Thread.hpp` is include
- Instantiate `LogSystem`
- Instantiate `KeyBoardInput`
  - Parsing a reference to `LogSystem` - Need access to `LogSystem` message queue

```
01  // main.cpp
02  #include <osapi/Thread.hpp>
03  //#include <Console.hpp>
04  #include <osapi/example/LogSystem.hpp>
05  #include <osapi/example/KeyBoardInput.hpp>
06
07  int main()
08  {
09      //Console c;
10      LogSystem l;
11      KeyBoardInput k(&l);
12
13      //osapi::Thread ct(&c);
14      osapi::Thread lt(&l);
15      lt.start();
16      osapi::Thread kt(&k);
17      kt.start();
18
19      //ct.join();
20      lt.join();
21      kt.join();
22
23  }
```

IMPLEMENTATION OF MAIN.CPP

- Various includes
  - Note that `osapi/Thread.hpp` is include
- Instantiate `LogSystem`
- Instantiate `KeyBoardInput`
  - Parsing a reference to `LogSystem` - Need access to `LogSystem` message queue
- Create and start `LogSystem` thread

```
01  // main.cpp
02  #include <osapi/Thread.hpp>
03  //#include <Console.hpp>
04  #include <osapi/example/LogSystem.hpp>
05  #include <osapi/example/KeyBoardInput.hpp>
06
07  int main()
08  {
09      //Console c;
10      LogSystem l;
11      KeyBoardInput k(&l);
12
13      //osapi::Thread ct(&c);
14      osapi::Thread lt(&l);
15      lt.start();
16      osapi::Thread kt(&k);
17      kt.start();
18
19      //ct.join();
20      lt.join();
21      kt.join();
22
23  }
```

IMPLEMENTATION OF MAIN.CPP

- Various includes
  - Note that `osapi/Thread.hpp` is include
- Instantiate `LogSystem`
- Instantiate `KeyBoardInput`
  - Parsing a reference to `LogSystem` - Need access to `LogSystem` message queue
- Create and start `LogSystem` thread
- Create and start `KeyBoardInput` thread

```
01  // main.cpp
02  #include <osapi/Thread.hpp>
03  //#include <Console.hpp>
04  #include <osapi/example/LogSystem.hpp>
05  #include <osapi/example/KeyBoardInput.hpp>
06
07  int main()
08  {
09      //Console c;
10      LogSystem l;
11      KeyBoardInput k(&l);
12
13      //osapi::Thread ct(&c);
14      osapi::Thread lt(&l);
15      lt.start();
16      osapi::Thread kt(&k);
17      kt.start();
18
19      //ct.join();
20      lt.join();
21      kt.join();
22
23  }
```

IMPLEMENTATION OF MAIN.CPP

- Various includes
  - Note that `osapi/Thread.hpp` is include
- Instantiate `LogSystem`
- Instantiate `KeyBoardInput`
  - Parsing a reference to `LogSystem` - Need access to `LogSystem` message queue
- Create and start `LogSystem` thread
- Create and start `KeyBoardInput` thread
- Finally remembering joining...

```
01  // main.cpp
02  #include <osapi/Thread.hpp>
03  //#include <Console.hpp>
04  #include <osapi/example/LogSystem.hpp>
05  #include <osapi/example/KeyBoardInput.hpp>
06
07  int main()
08  {
09      //Console c;
10      LogSystem l;
11      KeyBoardInput k(&l);
12
13      //osapi::Thread ct(&c);
14      osapi::Thread lt(&l);
15      lt.start();
16      osapi::Thread kt(&k);
17      kt.start();
18
19      //ct.join();
20      lt.join();
21      kt.join();
22
23  }
```

IMPLEMENTATION OF KEYBOARDINPUT

```
01 #ifndef KEYBOARD_INPUT_H_
02 #define KEYBOARD_INPUT_H_
03 #include <string>
04 #include <osapi/MsgQueue.hpp>
05 #include <osapi/ThreadFunctor.hpp>
06 #include <osapi/example/LogSystem.hpp>
07
08 class KeyBoardInput : public osapi::ThreadFunctor
09 {
10 public:
11   KeyBoardInput(LogSystem* l)
12     : l_(l) {}
13 private:
14   void run();
15   LogSystem*      l_;
16 };
17
18 #endif
```

```
01 // KeyBoardInput.cpp
02 #include <iostream>
03 #include <osapi/example/KeyBoardInput.hpp>
04
05 void KeyBoardInput::run()
06 {
07   for(;;)
08   {
09     std::string s;
10     std::cin >> s;
11     LogInd* logInd = new LogInd;
12     logInd->text = s;
13     l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14   }
15 }
```

IMPLEMENTATION OF KEYBOARDINPUT

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*

```
01  #ifndef KEYBOARD_INPUT_H_
02  #define KEYBOARD_INPUT_H_
03  #include <string>
04  #include <osapi/MsgQueue.hpp>
05  #include <osapi/ThreadFunctor.hpp>
06  #include <osapi/example/LogSystem.hpp>
07
08  class KeyBoardInput : public osapi::ThreadFunctor
09  {
10  public:
11    KeyBoardInput(LogSystem* l)
12      : l_(l) {}
13  private:
14    void run();
15    LogSystem*      l_;
16  };
17
18  #endif
```

```
01  // KeyBoardInput.cpp
02  #include <iostream>
03  #include <osapi/example/KeyBoardInput.hpp>
04
05  void KeyBoardInput::run()
06  {
07    for(;;)
08    {
09      std::string s;
10      std::cin >> s;
11      LogInd* logInd = new LogInd;
12      logInd->text = s;
13      l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14    }
15  }
```

IMPLEMENTATION OF KEYBOARDINPUT

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Constructor saving a reference to `LogSystem`

```
01 #ifndef KEYBOARD_INPUT_H_
02 #define KEYBOARD_INPUT_H_
03 #include <string>
04 #include <osapi/MsgQueue.hpp>
05 #include <osapi/ThreadFunctor.hpp>
06 #include <osapi/example/LogSystem.hpp>
07
08 class KeyBoardInput : public osapi::ThreadFunctor
09 {
10 public:
11   KeyBoardInput(LogSystem* l)
12     : l_(l) {}
13 private:
14   void run();
15   LogSystem*      l_;
16 };
17
18 #endif
```

```
01 // KeyBoardInput.cpp
02 #include <iostream>
03 #include <osapi/example/KeyBoardInput.hpp>
04
05 void KeyBoardInput::run()
06 {
07   for(;;)
08   {
09     std::string s;
10     std::cin >> s;
11     LogInd* logInd = new LogInd;
12     logInd->text = s;
13     l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14   }
15 }
```

IMPLEMENTATION OF KEYBOARDINPUT

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Constructor saving a reference to `LogSystem`
- Implementation of method `run()`

```
01 #ifndef KEYBOARD_INPUT_H_
02 #define KEYBOARD_INPUT_H_
03 #include <string>
04 #include <osapi/MsgQueue.hpp>
05 #include <osapi/ThreadFunctor.hpp>
06 #include <osapi/example/LogSystem.hpp>
07
08 class KeyBoardInput : public osapi::ThreadFunctor
09 {
10 public:
11   KeyBoardInput(LogSystem* l)
12     : l_(l) {}
13 private:
14   void run();
15   LogSystem*      l_;
16 };
17
18 #endif
```

```
01 // KeyBoardInput.cpp
02 #include <iostream>
03 #include <osapi/example/KeyBoardInput.hpp>
04
05 void KeyBoardInput::run()
06 {
07   for(;;)
08   {
09     std::string s;
10     std::cin >> s;
11     LogInd* logInd = new LogInd;
12     logInd->text = s;
13     l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14   }
15 }
```

IMPLEMENTATION OF KEYBOARDINPUT

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Constructor saving a reference to `LogSystem`
- Implementation of method `run()`
- Read line from *stdin*

```
01 #ifndef KEYBOARD_INPUT_H_
02 #define KEYBOARD_INPUT_H_
03 #include <string>
04 #include <osapi/MsgQueue.hpp>
05 #include <osapi/ThreadFunctor.hpp>
06 #include <osapi/example/LogSystem.hpp>
07
08 class KeyBoardInput : public osapi::ThreadFunctor
09 {
10 public:
11   KeyBoardInput(LogSystem* l)
12     : l_(l) {}
13 private:
14   void run();
15   LogSystem*      l_;
16 };
17
18 #endif
```

```
01 // KeyBoardInput.cpp
02 #include <iostream>
03 #include <osapi/example/KeyBoardInput.hpp>
04
05 void KeyBoardInput::run()
06 {
07   for(;;)
08   {
09     std::string s;
10     std::cin >> s;
11     LogInd* logInd = new LogInd;
12     logInd->text = s;
13     l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14   }
15 }
```

IMPLEMENTATION OF KEYBOARDINPUT

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Constructor saving a reference to `LogSystem`
- Implementation of method `run()`
- Read line from *stdin*
- Allocate message

```
01 #ifndef KEYBOARD_INPUT_H_
02 #define KEYBOARD_INPUT_H_
03 #include <string>
04 #include <osapi/MsgQueue.hpp>
05 #include <osapi/ThreadFunctor.hpp>
06 #include <osapi/example/LogSystem.hpp>
07
08 class KeyBoardInput : public osapi::ThreadFunctor
09 {
10 public:
11   KeyBoardInput(LogSystem* l)
12     : l_(l) {}
13 private:
14   void run();
15   LogSystem*      l_;
16 };
17
18 #endif
```

```
01 // KeyBoardInput.cpp
02 #include <iostream>
03 #include <osapi/example/KeyBoardInput.hpp>
04
05 void KeyBoardInput::run()
06 {
07   for(;;)
08   {
09     std::string s;
10     std::cin >> s;
11     LogInd* logInd = new LogInd;
12     logInd->text = s;
13     l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14   }
15 }
```

IMPLEMENTATION OF KEYBOARDINPUT

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Constructor saving a reference to `LogSystem`
- Implementation of method `run()`
- Read line from *stdin*
- Allocate message
- Copy read string to message

```
01 #ifndef KEYBOARD_INPUT_H_
02 #define KEYBOARD_INPUT_H_
03 #include <string>
04 #include <osapi/MsgQueue.hpp>
05 #include <osapi/ThreadFunctor.hpp>
06 #include <osapi/example/LogSystem.hpp>
07
08 class KeyBoardInput : public osapi::ThreadFunctor
09 {
10 public:
11   KeyBoardInput(LogSystem* l)
12     : l_(l) {}
13 private:
14   void run();
15   LogSystem*      l_;
16 };
17
18 #endif
```

```
01 // KeyBoardInput.cpp
02 #include <iostream>
03 #include <osapi/example/KeyBoardInput.hpp>
04
05 void KeyBoardInput::run()
06 {
07   for(;;)
08   {
09     std::string s;
10     std::cin >> s;
11     LogInd* logInd = new LogInd;
12     logInd->text = s;
13     l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14   }
15 }
```

IMPLEMENTATION OF KEYBOARDINPUT

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Constructor saving a reference to `LogSystem`
- Implementation of method `run()`
- Read line from *stdin*
- Allocate message
- Copy read string to message
- Get pointer to `LogSystem` message queue and send the message to it

```
01 #ifndef KEYBOARD_INPUT_H_
02 #define KEYBOARD_INPUT_H_
03 #include <string>
04 #include <osapi/MsgQueue.hpp>
05 #include <osapi/ThreadFunctor.hpp>
06 #include <osapi/example/LogSystem.hpp>
07
08 class KeyBoardInput : public osapi::ThreadFunctor
09 {
10 public:
11   KeyBoardInput(LogSystem* l)
12     : l_(l) {}
13 private:
14   void run();
15   LogSystem*      l_;
16 };
17
18 #endif
```

```
01 // KeyBoardInput.cpp
02 #include <iostream>
03 #include <osapi/example/KeyBoardInput.hpp>
04
05 void KeyBoardInput::run()
06 {
07   for(;;)
08   {
09     std::string s;
10     std::cin >> s;
11     LogInd* logInd = new LogInd;
12     logInd->text = s;
13     l_->getMsgQueue()->send(LogSystem::ID_LOG_IND, logInd);
14   }
15 }
```

INTERFACE FOR LOGSYSTEM

```
01 #ifndef LOG_SYSTEM_H_
02 #define LOG_SYSTEM_H_
03
04 #include <string>
05 #include <fstream>
06 #include <osapi/MsgQueue.hpp>
07 #include <osapi/ThreadFunctor.hpp>
08
09 struct LogInd : public osapi::Message
10 { std::string text; };
11
12 class LogSystem : public osapi::ThreadFunctor
13 {
14 public:
15     enum { ID_LOG_IND };
16     static const int MAX_QUEUE_SIZE = 10;
17     LogSystem()
18     : mq_(MAX_QUEUE_SIZE),
19       lf_("log.txt") { }
20
21     osapi::MsgQueue* getMsgQueue() { return &mq_; }
22
23 private:
24     void writeToLog(LogInd* l);
25     void handleMsg(unsigned long id, osapi::Message* msg);
26     void run();
27
28     osapi::MsgQueue mq_;
29     std::ofstream   lf_;
30 };
31
32 #endif
```

INTERFACE FOR LOGSYSTEM

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*

```
01  #ifndef LOG_SYSTEM_H_
02  #define LOG_SYSTEM_H_
03
04  #include <string>
05  #include <fstream>
06  #include <osapi/MsgQueue.hpp>
07  #include <osapi/ThreadFunctor.hpp>
08
09  struct LogInd : public osapi::Message
10  { std::string text; };
11
12  class LogSystem : public osapi::ThreadFunctor
13  {
14  public:
15      enum { ID_LOG_IND };
16      static const int MAX_QUEUE_SIZE = 10;
17      LogSystem()
18      : mq_(MAX_QUEUE_SIZE),
19        lf_("log.txt") { }
20
21      osapi::MsgQueue* getMsgQueue() { return &mq_; }
22
23  private:
24      void writeToLog(LogInd* l);
25      void handleMsg(unsigned long id, osapi::Message* msg);
26      void run();
27
28      osapi::MsgQueue mq_;
29      std::ofstream   lf_;
30  };
31
32  #endif
```

INTERFACE FOR LOGSYSTEM

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Message identifier and associated message/struct
- Note the id is part of the class interface

```
01 #ifndef LOG_SYSTEM_H_
02 #define LOG_SYSTEM_H_
03
04 #include <string>
05 #include <fstream>
06 #include <osapi/MsgQueue.hpp>
07 #include <osapi/ThreadFunctor.hpp>
08
09 struct LogInd : public osapi::Message
10 { std::string text; };
11
12 class LogSystem : public osapi::ThreadFunctor
13 {
14 public:
15     enum { ID_LOG_IND };
16     static const int MAX_QUEUE_SIZE = 10;
17     LogSystem()
18     : mq_(MAX_QUEUE_SIZE),
19       lf_("log.txt") { }
20
21     osapi::MsgQueue* getMsgQueue() { return &mq_; }
22
23 private:
24     void writeToLog(LogInd* l);
25     void handleMsg(unsigned long id, osapi::Message* msg);
26     void run();
27
28     osapi::MsgQueue mq_;
29     std::ofstream   lf_;
30 };
31
32 #endif
```

INTERFACE FOR LOGSYSTEM

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Message identifier and associated message/struct
- Note the id is part of the class interface
- Instantiate a message queue as part of the class
  - Can receive messages now!
- Pointer to queue via `getMsgQueue()`

```
01 #ifndef LOG_SYSTEM_H_
02 #define LOG_SYSTEM_H_
03
04 #include <string>
05 #include <fstream>
06 #include <osapi/MsgQueue.hpp>
07 #include <osapi/ThreadFunctor.hpp>
08
09 struct LogInd : public osapi::Message
10 { std::string text; };
11
12 class LogSystem : public osapi::ThreadFunctor
13 {
14 public:
15     enum { ID_LOG_IND };
16     static const int MAX_QUEUE_SIZE = 10;
17     LogSystem()
18     : mq_(MAX_QUEUE_SIZE),
19       lf_("log.txt") { }
20
21     osapi::MsgQueue* getMsgQueue() { return &mq_; }
22
23 private:
24     void writeToLog(LogInd* l);
25     void handleMsg(unsigned long id, osapi::Message* msg);
26     void run();
27
28     osapi::MsgQueue mq_;
29     std::ofstream   lf_;
30 };
31
32 #endif
```

INTERFACE FOR LOGSYSTEM

- Inherit from osapi::ThreadFunctor and implement `run()`
  - Method `run()` is the *thread*
- Message identifier and associated message/struct
- Note the id is part of the class interface
- Instantiate a message queue as part of the class
  - Can receive messages now!
- Pointer to queue via `getMsgQueue()`
- Open file for writing the log messages in the associated handler

```
01 #ifndef LOG_SYSTEM_H_
02 #define LOG_SYSTEM_H_
03
04 #include <string>
05 #include <fstream>
06 #include <osapi/MsgQueue.hpp>
07 #include <osapi/ThreadFunctor.hpp>
08
09 struct LogInd : public osapi::Message
10 { std::string text; };
11
12 class LogSystem : public osapi::ThreadFunctor
13 {
14 public:
15     enum { ID_LOG_IND };
16     static const int MAX_QUEUE_SIZE = 10;
17     LogSystem()
18     : mq_(MAX_QUEUE_SIZE),
19       lf_("log.txt") { }
20
21     osapi::MsgQueue* getMsgQueue() { return &mq_; }
22
23 private:
24     void writeToLog(LogInd* l);
25     void handleMsg(unsigned long id, osapi::Message* msg);
26     void run();
27
28     osapi::MsgQueue mq_;
29     std::ofstream   lf_;
30 };
31
32 #endif
```

IMPLEMENTATION OF LOGSYSTEM

```cpp
// LogSystem.cpp
#include <iostream>
#include <osapi/example/LogSystem.hpp>

void LogSystem::writeToLog(LogInd* l)
{
  lf_ << l->text << std::endl;
}

void LogSystem::handleMsg(unsigned long id, osapi::Message* msg)
{
  switch(id)
  {
    case ID_LOG_IND:
      writeToLog(static_cast<LogInd*>(msg));
      break;

    default:
      std::cout << "Unknown event..." << std::endl;
  }
}

void LogSystem::run()
{
  for(;;)
  {
    unsigned long id;
    osapi::Message* msg = mq_.receive(id);
    handleMsg(id, msg);
    delete msg;
  }
}
```

IMPLEMENTATION OF LOGSYSTEM

- Classic event loop
  - Receive
  - Handle
  - Delete

```
01  // LogSystem.cpp
02  #include <iostream>
03  #include <osapi/example/LogSystem.hpp>
04
05  void LogSystem::writeToLog(LogInd* l)
06  {
07    lf_ << l->text << std::endl;
08  }
09
10  void LogSystem::handleMsg(unsigned long id, osapi::Message* msg)
11  {
12    switch(id)
13    {
14      case ID_LOG_IND:
15        writeToLog(static_cast<LogInd*>(msg));
16        break;
17
18      default:
19        std::cout << "Unknown event..." << std::endl;
20    }
21  }
22
23  void LogSystem::run()
24  {
25    for(;;)
26    {
27      unsigned long id;
28      osapi::Message* msg = mq_.receive(id);
29      handleMsg(id, msg);
30      delete msg;
31    }
32  }
```

IMPLEMENTATION OF LOGSYSTEM

- Classic event loop
  - Receive
  - Handle
  - Delete
- Handler (dispatcher)
  - Cases out on the various ids
    - only one here
  - Handler placed in its own method `writeToLog()`

```
01  // LogSystem.cpp
02  #include <iostream>
03  #include <osapi/example/LogSystem.hpp>
04
05  void LogSystem::writeToLog(LogInd* l)
06  {
07    lf_ << l->text << std::endl;
08  }
09
10  void LogSystem::handleMsg(unsigned long id, osapi::Message* msg)
11  {
12    switch(id)
13    {
14      case ID_LOG_IND:
15        writeToLog(static_cast<LogInd*>(msg));
16        break;
17
18      default:
19        std::cout << "Unknown event..." << std::endl;
20    }
21  }
22
23  void LogSystem::run()
24  {
25    for(;;)
26    {
27      unsigned long id;
28      osapi::Message* msg = mq_.receive(id);
29      handleMsg(id, msg);
30      delete msg;
31    }
32  }
```

IMPLEMENTATION OF LOGSYSTEM

- Classic event loop
  - Receive
  - Handle
  - Delete
- Handler (dispatcher)
  - Cases out on the various ids
    - only one here
  - Handler placed in its own method `writeToLog()`
- Write the contents of the string received to the log file

```
01  // LogSystem.cpp
02  #include <iostream>
03  #include <osapi/example/LogSystem.hpp>
04
05  void LogSystem::writeToLog(LogInd* l)
06  {
07    lf_ << l->text << std::endl;
08  }
09
10  void LogSystem::handleMsg(unsigned long id, osapi::Message* msg)
11  {
12    switch(id)
13    {
14      case ID_LOG_IND:
15        writeToLog(static_cast<LogInd*>(msg));
16        break;
17
18      default:
19        std::cout << "Unknown event..." << std::endl;
20    }
21  }
22
23  void LogSystem::run()
24  {
25    for(;;)
26    {
27      unsigned long id;
28      osapi::Message* msg = mq_.receive(id);
29      handleMsg(id, msg);
30      delete msg;
31    }
32  }
```

THREADING IN OO OS

API

THREADING IN OO OS API

- How to create threads in OO OS Api
- What happens behind the scenes
  - Going from a C to C++ API

HOW TO CREATE THREADS IN OO OS API

```
01 // Shortened
02 class Thread
03 {
04 public:
05   Thread(ThreadFunctor* tf,
06          ThreadPriority p = PRIORITY_NORMAL,
07          const std::string& name="",
08          bool autoStart = false);
09   void start();
10 private:
11   ThreadFunctor*  tf_;
12 };
```

```
01 // Shortened
02 class ThreadFunctor
03 {
04 public:
05
06 protected:
07   virtual void run() = 0;
08   ~ThreadFunctor(){}
09 private:
10   static void* threadMapper(void* p);
11 };
```

```
01 class KeyBoardInput : public osapi::ThreadFunctor
02 {
03 public:
04   KeyBoardInput(LogSystem* l)
05     : l_(l) {}
06 protected:
07   virtual void run();
08 private:
09   LogSystem*      l_;
10 };
```

HOW TO CREATE THREADS IN OO OS API

```
01  // Shortened
02  class Thread
03  {
04  public:
05    Thread(ThreadFunctor* tf,
06           ThreadPriority p = PRIORITY_NORMAL,
07           const std::string& name="",
08           bool autoStart = false);
09    void start();
10  private:
11    ThreadFunctor*  tf_;
12  };
```

- Administrative part
- Needs a reference to a `ThreadFunctor` the actual thread

```
01  // Shortened
02  class ThreadFunctor
03  {
04  public:
05
06  protected:
07    virtual void run() = 0;
08    ~ThreadFunctor(){}
09  private:
10    static void* threadMapper(void* p);
11  };
```

```
01  class KeyBoardInput : public osapi::ThreadFunctor
02  {
03  public:
04    KeyBoardInput(LogSystem* l)
05      : l_(l) {}
06  protected:
07    virtual void run();
08  private:
09    LogSystem*      l_;
10  };
```

HOW TO CREATE THREADS IN OO OS API

```
01  // Shortened
02  class Thread
03  {
04  public:
05    Thread(ThreadFunctor* tf,
06           ThreadPriority p = PRIORITY_NORMAL,
07           const std::string& name="",
08           bool autoStart = false);
09    void start();
10  private:
11    ThreadFunctor*  tf_;
12  };
```

- Administrative part
- Needs a reference to a `ThreadFunctor` the actual thread

- The actual thread
- Method `threadMapper()` calls `run()`
  - Inheriting classes *must* implement `run()`

```
01  // Shortened
02  class ThreadFunctor
03  {
04  public:
05
06  protected:
07    virtual void run() = 0;
08    ~ThreadFunctor(){}
09  private:
10    static void* threadMapper(void* p);
11  };
```

```
01  class KeyBoardInput : public osapi::ThreadFunctor
02  {
03  public:
04    KeyBoardInput(LogSystem* l)
05      : l_(l) {}
06  protected:
07    virtual void run();
08  private:
09    LogSystem*      l_;
10  };
```

HOW TO CREATE THREADS IN OO OS API

```
01 // Shortened
02 class Thread
03 {
04 public:
05   Thread(ThreadFunctor* tf,
06          ThreadPriority p = PRIORITY_NORMAL,
07          const std::string& name="",
08          bool autoStart = false);
09   void start();
10 private:
11    ThreadFunctor*  tf_;
12 };
```

- Administrative part
- Needs a reference to a `ThreadFunctor` the actual thread

- The actual thread
- Method `threadMapper()` calls `run()`
  - Inheriting classes *must* implement `run()`

- Inherit from `osapi::ThreadFunctor` and implements `run()`
  - Method `run()` is the *thread*

```
01 // Shortened
02 class ThreadFunctor
03 {
04 public:
05
06 protected:
07   virtual void run() = 0;
08   ~ThreadFunctor(){}
09 private:
10   static void* threadMapper(void* p);
11 };
```

```
01 class KeyBoardInput : public osapi::ThreadFunctor
02 {
03 public:
04   KeyBoardInput(LogSystem* l)
05     : l_(l) {}
06 protected:
07   virtual void run();
08 private:
09   LogSystem*      l_;
10 };
```

WHAT HAPPENS BEHIND THE SCENES

GOING FROM A C TO C++ API

```
01 Thread::Thread(ThreadFunctor* tf,
02                Thread::ThreadPriority priority,
03                const std::string& name,
04                bool autoStart)
05 : tf_(tf), priority_(priority), name_(name), attached_(true)
06 {
07   if(autoStart)
08     start();
09 }
10
11 void Thread::start()
12 {
13   //...
14   if(pthread_create(&threadId_, &attr, ThreadFunctor::threadMapper,
15                     tf_) != 0) throw ThreadError();
16   //...
17 }
```

```
01 void* ThreadFunctor::threadMapper(void* thread)
02 {
03   ThreadFunctor* tf = static_cast<ThreadFunctor*>(thread);
04   tf->run();
05
06   tf->threadDone_.signal();
07   return NULL;
08 }
```

WHAT HAPPENS BEHIND THE SCENES

GOING FROM A C TO C++ API

- The `Thread` class administrates an associated thread
  - Namely a `ThreadFunctor`

```
01 Thread::Thread(ThreadFunctor* tf,
02                Thread::ThreadPriority priority,
03                const std::string& name,
04                bool autoStart)
05 : tf_(tf), priority_(priority), name_(name), attached_(true)
06 {
07   if(autoStart)
08     start();
09 }
10
11 void Thread::start()
12 {
13   //...
14   if(pthread_create(&threadId_, &attr, ThreadFunctor::threadMapper,
15                     tf_) != 0) throw ThreadError();
16   //...
17 }
```

```
01 void* ThreadFunctor::threadMapper(void* thread)
02 {
03   ThreadFunctor* tf = static_cast<ThreadFunctor*>(thread);
04   tf->run();
05
06   tf->threadDone_.signal();
07   return NULL;
08 }
```

WHAT HAPPENS BEHIND THE SCENES

GOING FROM A C TO C++ API

- The `Thread` class administrates an associated thread
  - Namely a `ThreadFunctor`
- Posix threads are threads created using C
  - Thus *impossible* to call C++ methods
    - `pthread_create` requires a free/global function

```
01 Thread::Thread(ThreadFunctor* tf,
02                Thread::ThreadPriority priority,
03                const std::string& name,
04                bool autoStart)
05 : tf_(tf), priority_(priority), name_(name), attached_(true)
06 {
07   if(autoStart)
08     start();
09 }
10
11 void Thread::start()
12 {
13   //...
14   if(pthread_create(&threadId_, &attr, ThreadFunctor::threadMapper,
15                     tf_) != 0) throw ThreadError();
16   //...
17 }
```

```
01 void* ThreadFunctor::threadMapper(void* thread)
02 {
03   ThreadFunctor* tf = static_cast<ThreadFunctor*>(thread);
04   tf->run();
05
06   tf->threadDone_.signal();
07   return NULL;
08 }
```

WHAT HAPPENS BEHIND THE SCENES

GOING FROM A C TO C++ API

- The `Thread` class administrates an associated thread
  - Namely a `ThreadFunctor`
- Posix threads are threads created using C
  - Thus *impossible* to call C++ methods
    - `pthread_create` requires a free/global function
- Solution
  - Method `ThreadFunctor::threadMapper` is `static` therefore free/global
    - Returns `void*` and takes a `void*`
  - `pthread_create()` is passed
    - `ThreadFunctor::threadMapper`
    - Pointer to an object which class inherits from `ThreadFunctor`
      - This class implements the `run()` method

```
01 Thread::Thread(ThreadFunctor* tf,
02                Thread::ThreadPriority priority,
03                const std::string& name,
04                bool autoStart)
05 : tf_(tf), priority_(priority), name_(name), attached_(true)
06 {
07   if(autoStart)
08     start();
09 }
10
11 void Thread::start()
12 {
13   //...
14   if(pthread_create(&threadId_, &attr, ThreadFunctor::threadMapper,
15                     tf_) != 0) throw ThreadError();
16   //...
17 }
```

```
01 void* ThreadFunctor::threadMapper(void* thread)
02 {
03   ThreadFunctor* tf = static_cast<ThreadFunctor*>(thread);
04   tf->run();
05
06   tf->threadDone_.signal();
07   return NULL;
08 }
```

WHAT HAPPENS BEHIND THE SCENES

GOING FROM A C TO C++ API

- The `Thread` class administrates an associated thread
  - Namely a `ThreadFunctor`
- Posix threads are threads created using C
  - Thus *impossible* to call C++ methods
    - `pthread_create` requires a free/global function
- Solution
  - `thread` is a `ThreadFunctor`
    - Thus enabling the static_cast< ThreadFunctor*>(thread);

```
01 Thread::Thread(ThreadFunctor* tf,
02                Thread::ThreadPriority priority,
03                const std::string& name,
04                bool autoStart)
05 : tf_(tf), priority_(priority), name_(name), attached_(true)
06 {
07   if(autoStart)
08     start();
09 }
10
11 void Thread::start()
12 {
13   //...
14   if(pthread_create(&threadId_, &attr, ThreadFunctor::threadMapper,
15                     tf_) != 0) throw ThreadError();
16   //...
17 }
```

```
01 void* ThreadFunctor::threadMapper(void* thread)
02 {
03   ThreadFunctor* tf = static_cast<ThreadFunctor*>(thread);
04   tf->run();
05
06   tf->threadDone_.signal();
07   return NULL;
08 }
```

WHAT HAPPENS BEHIND THE SCENES

GOING FROM A C TO C++ API

- The `Thread` class administrates an associated thread
  - Namely a `ThreadFunctor`
- Posix threads are threads created using C
  - Thus *impossible* to call C++ methods
    - `pthread_create` requires a free/global function
- Solution
  - Finally we can call `run()`
    - The inherited implemented thus called since its a virtual method

```
01 Thread::Thread(ThreadFunctor* tf,
02                Thread::ThreadPriority priority,
03                const std::string& name,
04                bool autoStart)
05 : tf_(tf), priority_(priority), name_(name), attached_(true)
06 {
07   if(autoStart)
08     start();
09 }
10
11 void Thread::start()
12 {
13   //...
14   if(pthread_create(&threadId_, &attr, ThreadFunctor::threadMapper,
15                     tf_) != 0) throw ThreadError();
16   //...
17 }
```

```
01 void* ThreadFunctor::threadMapper(void* thread)
02 {
03   ThreadFunctor* tf = static_cast<ThreadFunctor*>(thread);
04   tf->run();
05
06   tf->threadDone_.signal();
07   return NULL;
08 }
```

GUIDELINES

GUIDELINES

- Simple threading example
- Using messages
- Adding states

SIMPLE THREADING EXAMPLE

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

```
01 class MyThread : public osapi::ThreadFunctor
02 {
03 public:
04   MyThread() : running_(true) {}
05   virtual void run()
06   {
07     while (running_)
08     {
09       m_.lock();
10       // Do stuff
11       m_.unlock();
12       // Do stuff
13     }
14   }
15 private:
16   bool          running_;
17   osapi::Mutex m_;
18 };
```

```
01 int main(int argc, char *argv[])
02 {
03   MyThread myt;
04   osapi::Thread t(&myt);
05   t.start();
06
07   t.join();
08 }
```

USING MESSAGES

- Event loop
  - Receive
  - Handle
  - Delete
- Handler
  - Case out
    - Specific handler
      - Name it by the message it handles
- Ids
  - Part of interface, part of service thread presents

```
01 class MyThread : osapi::ThreadFunctor
02 {
03 public:
04   enum { ID_START, ID_.... };
05
06   MyThread();
07   osapi::MsgQueue* getMsgQueue() { return &mq_; }
08
09 protected:
10   virtual void run();
11 private:
12   void handle(unsigned long id, osapi::Message* msg);
13   void handleMsgIdStart(Start* start);
14
15   osapi::MsgQueue mq_;
16   bool            running_;
17 };
18
19 virtual void MyThread::run()
20 {
21   while(running_)
22   {
23     unsigned long id;
24     Message* msg = mq_.receive(id);
25     handle(id, msg);
26     delete msg;
27   }
28 }
29
30 void MyThread::handle(unsigned long id, osapi::Message* msg)
31 {
32   switch(id)
33   {
34     case ID_START:
35       handleMsgIdStart(static_cast<Start*>(msg));
36       break;
37       ....
38   }
39 }
```

ADDING STATES - HEADER

- Same as previous, now an added variable state
  - Containing which state the thread is in

```
01  class MyThread : public osapi::ThreadFunctor
02  {
03  public:
04    enum MsgID
05    {
06      ID_MSG,
07      ID_TERMINATE
08    }
09
10    // Other functions...
11    MyThread();
12  protected:
13    virtual void run();
14  private:
15    void handleMsg(unsigned long id,
16                   osapi::Message* msg);
17
18    enum State
19    {
20      ST_IDLE,
21      ST_RUNNING
22    };
23
24    osapi::MsgQueue mq_;
25    State           state_;
26  };
```

ADDING STATES - IMPLEMENTATION

```
01  void MyThread::run()
02  {
03    // Initial one-time setup
04    while (running_)
05    {
06      // Wait for message (e.g. mail)
07      unsigned long id;
08      osapi::Message* msg = mq_.receive(id);
09      handleMsg(id, msg);
10      delete msg;
11    }
12  }
```

```
01  void MyThread::handleMsg(unsigned long id,
02                           osapi::Message* msg)
03  {
04    switch (state_)
05    {
06      case ST_IDLE:
07        handleMsgStateIdle(id, msg);
08        break;
09      default:
10        break;
11    }
12  }
```

```
01  void MyThread::handleMsgStIdle(unsigned int long,
02                                  osapi::Message* msg)
03  {
04    switch(id)
05    {
06      case ID_MSG:
07        // handle firstType-messages
08        handleStIdleIdMsg(static_cast<Msg*>(msg));
09        break;
10
11      case ID_TERMINATE:
12        // handle secondType-messages
13        break;
14
15      ...
16
17      default:
18        // signal an error
19        break;
20    }
21
22  }
```

- Same approach as without
- Only inserted a layer handling states