Embedded Software

A Message Distribution System

AARHUS
UNIVERSITY

Agenda

- Current design
- Decoupling current setup
  - Specific receiver
  - Broadcasting - who the receiver is *is* irrelevant
- Tools to use
  - Publisher/Subscriber (or Observer) pattern (& Signal/Slots)
  - Singleton pattern
  - Mediator pattern

Current design and next step

Current design

- A thread has a message queue, through which other threads pass it messages
  - Consequence is that “other” threads need to have access to its message queue.
    - Also need to know how that particular thread (message queue) wants its data
  - At application start these pointers (or references) must be passed around
- Problems - Potential Couplings issues
  - Challenges during creation - *chicken and the egg*
  - Leading to cyclic includes
  - Close relationships that are not needed

Next step

- 2 Overall different communication forms
  - **Communication via requests and confirms, two-way communication**
    - Knowledge of, or access to, message queue is relevant
    - Higher coupling, shared information
  - **Status information - indication**
    - One way communication
    - Knowledge of each other may be irrelevant
    - Anonymous system design may be used
    - Lower coupling

Next step

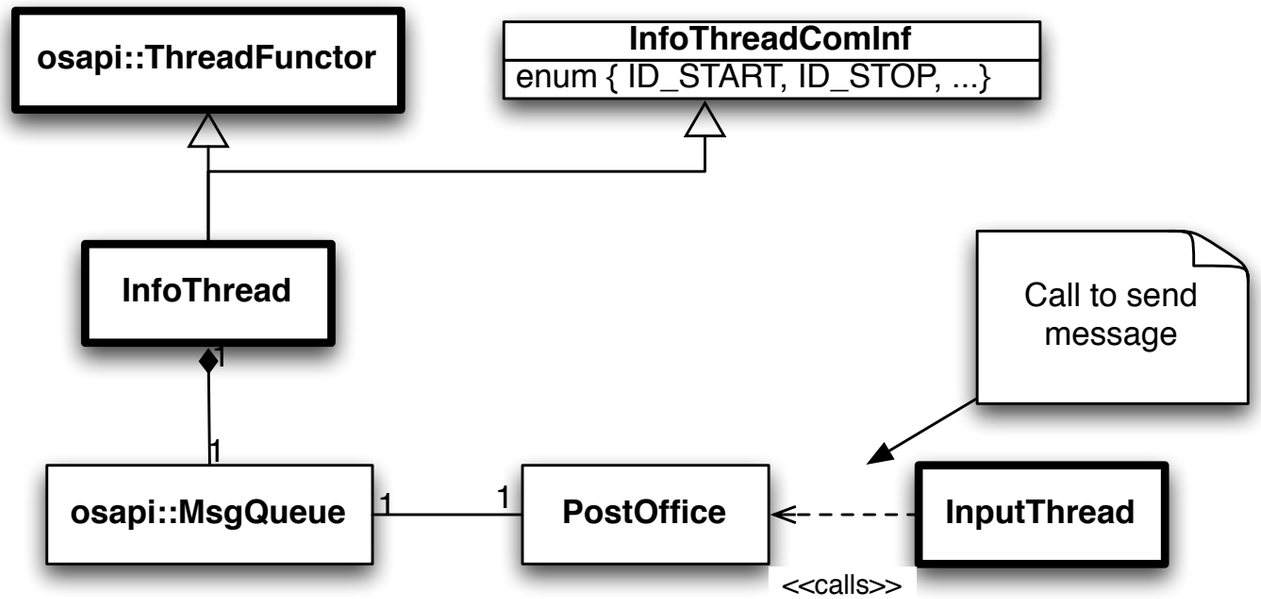
- 2 Overall different communication forms
  - **Communication via requests and confirms, two-way communication**
    - Knowledge of, or access to, message queue is relevant
    - Higher coupling, shared information
  - **Status information - indication**
    - One way communication
    - Knowledge of each other may be irrelevant
    - Anonymous system design may be used
    - Lower coupling

Our focus in todays exercise

Design: Specific receiver

Design: Specific receiver

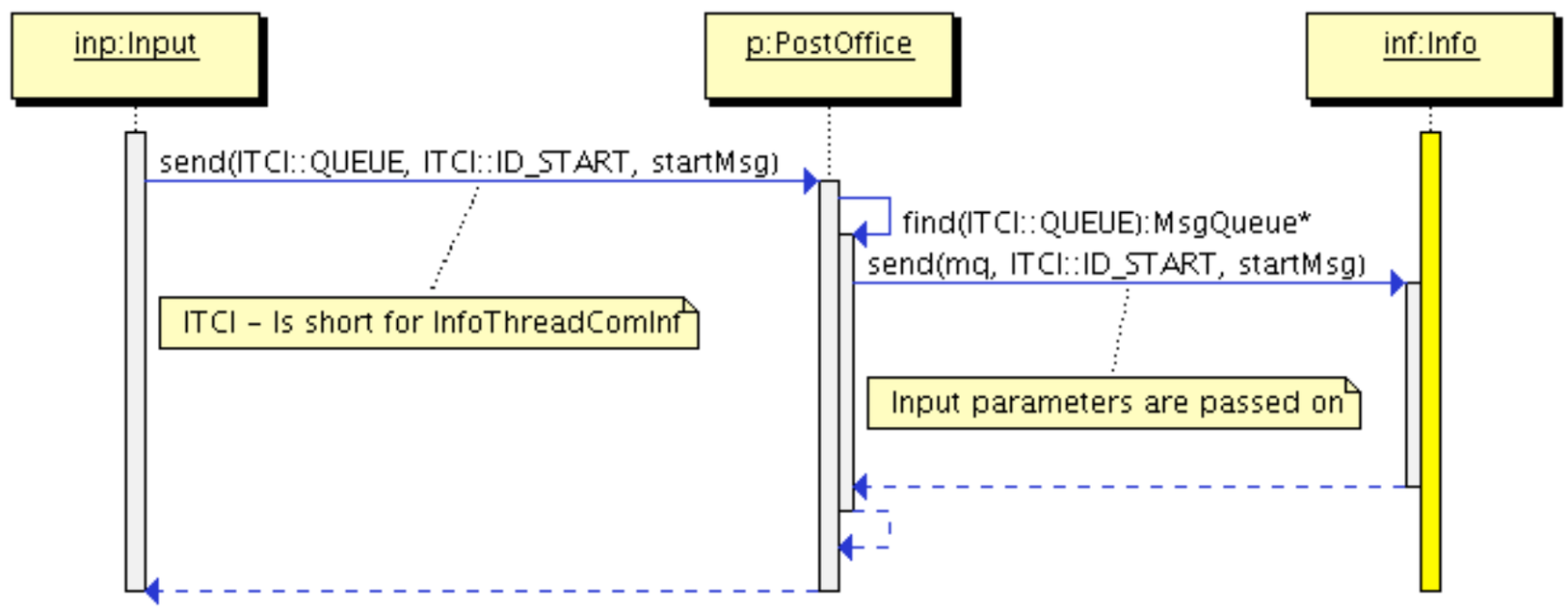
- Improve upon the include
  - Introduce another level (Mediator)
- Create a central postoffice
  - Send messages by *naming* (string format) the recipient
  - Or acquire a handle (speed up :-))
- Achieves
  - Low coupling since sender does not need to know receiver
  - Singleton usage or parsing around pointer/reference
  - Two-way communication possible



- Requires
  - A postoffice is up and running prior to use
  - Using a singleton or parsing around pointer/reference

Design: Specific receiver - Example

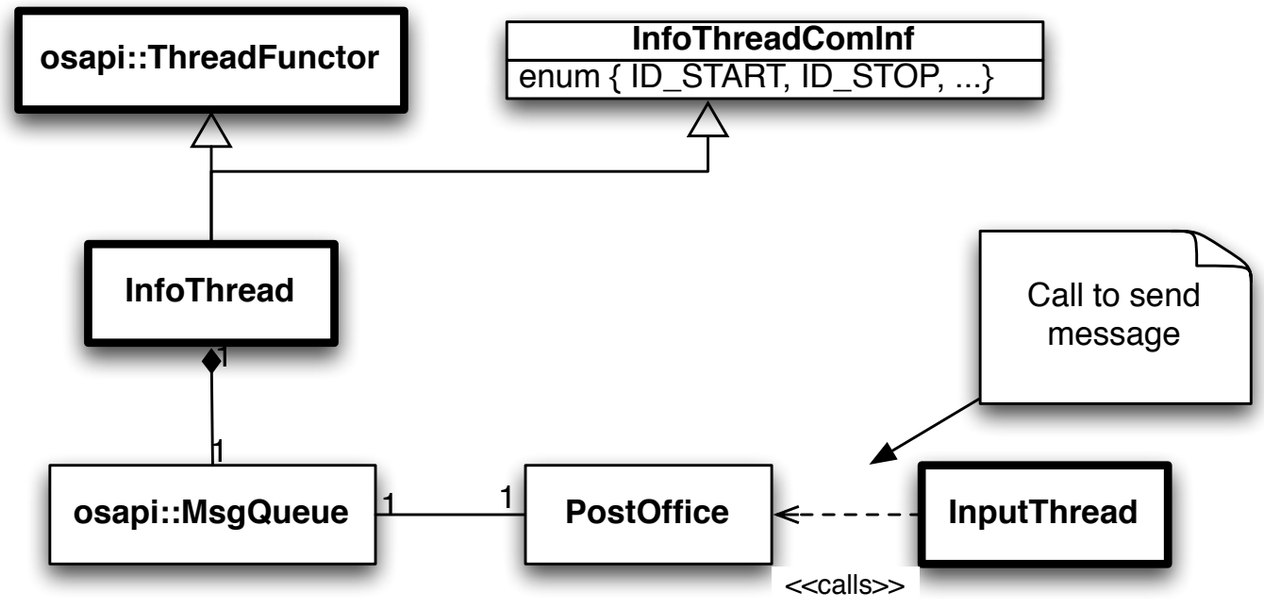
- Sequence diagram to illustrate
  - *Most importantly, the receiving party is denoted by a string*



AARHUS UNIVERSITY

Design: Specific receiver - Example

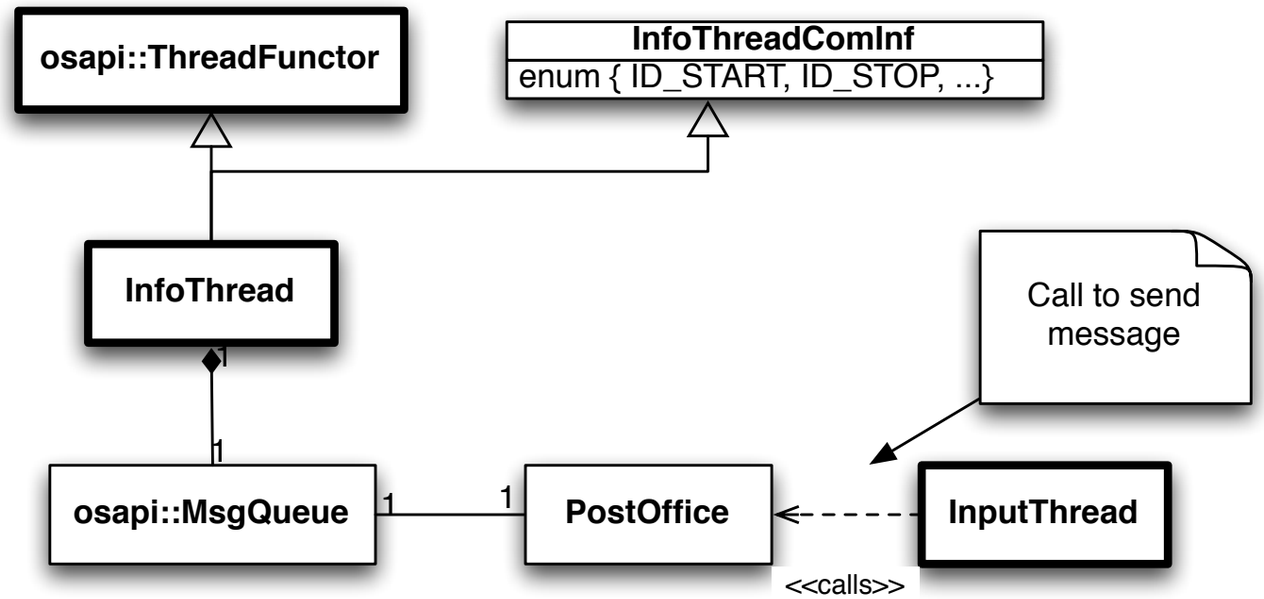
- Communication identification is done using a separate header file



Design: Specific receiver - Example

- Communication identification is done using a separate header file

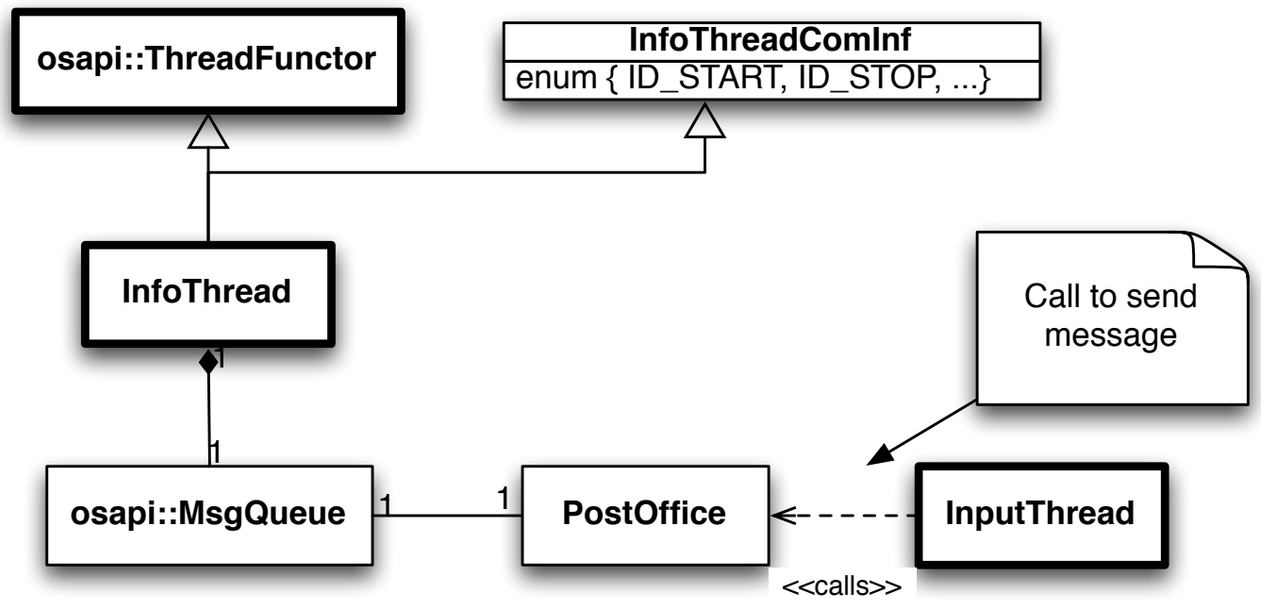
```
// InfoThreadComInf.hpp
struct InfoThreadComInf
{
     static const std::string QUEUE;
     enum { ID_START, ID_STOP, ... }
};

struct StartMsg : public osapi::Message
{ ... };
```



Design: Specific receiver - Example

- Communication identification is done using a separate header file

```
// InfoThreadComInf.hpp
struct InfoThreadComInf
{
     static const std::string QUEUE;
     enum { ID_START, ID_STOP, ... }
};

struct StartMsg : public osapi::Message
{ ... };
```

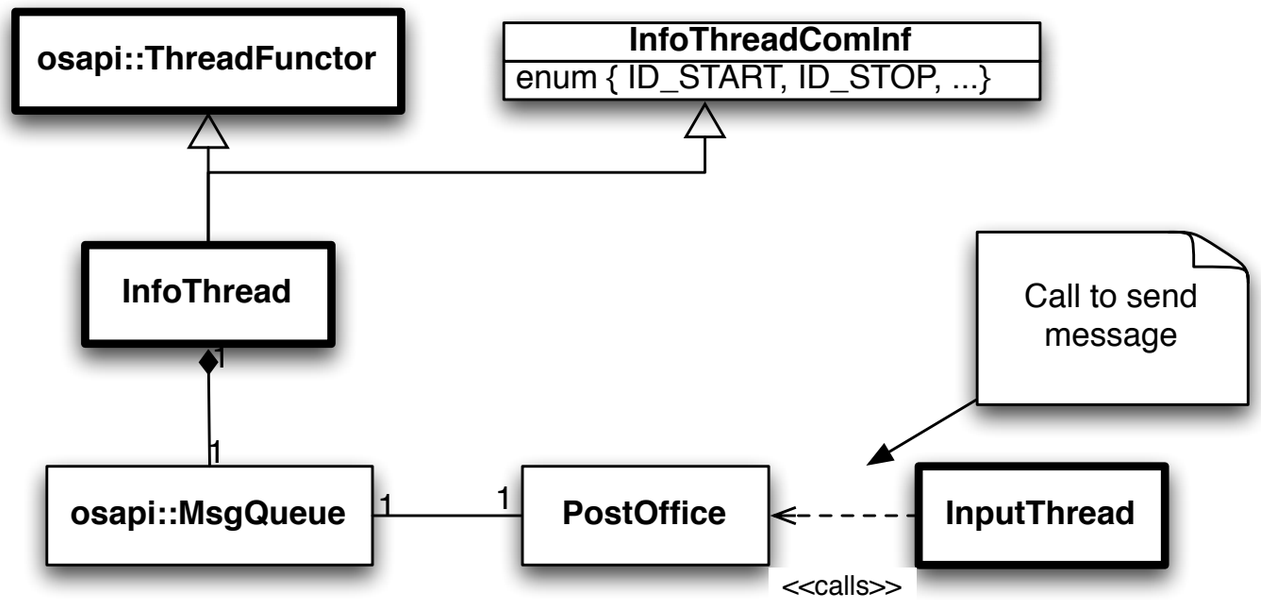



```
// Info thread header...
#include "InfoThreadComInf.hpp"
#include "PostOffice.hpp"

class InfoThread : public InfoThreadComInf,
                   public osapi::ThreadFunctor
{
    ...
private:
    // Receives Start message from “some”
    //  thread
    void handleStartMsg(StartMsg* sm);
    void handleMsg(...);
};
```

Design: Specific receiver - Example

- Communication identification is done using a separate header file

```
// InfoThreadComInf.hpp
struct InfoThreadComInf
{
     static const std::string QUEUE;
     enum { ID_START, ID_STOP, ... }
};

struct StartMsg : public osapi::Message
{ ... };
```



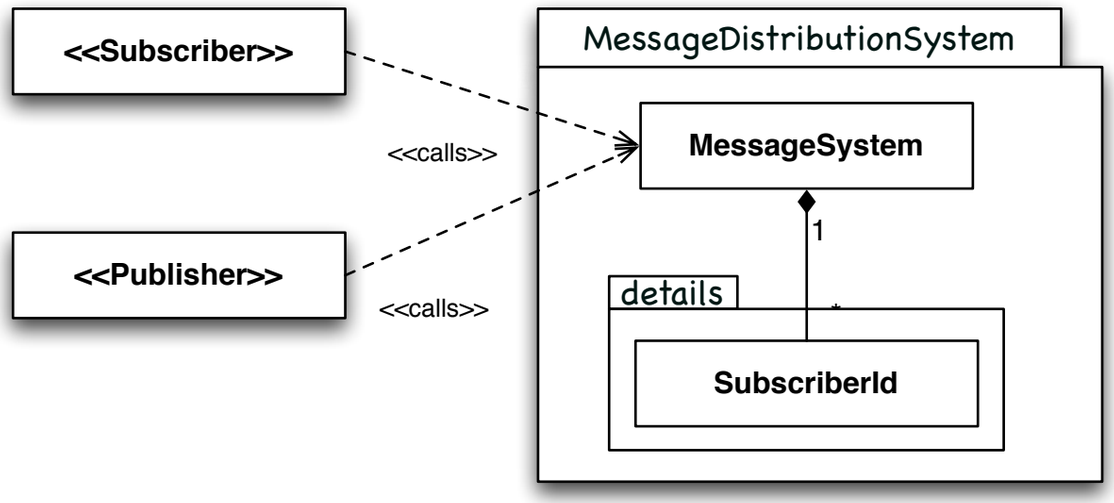

```
// In “some” thread about to send message...
#include "InfoThreadComInf.hpp"
#include "PostOffice.hpp"

void InputThread:HandleSomeMsg(...)
{
    StartMsg* startMsg = new StartMsg;

    PostOffice::send(InfoThreadComInf::QUEUE,
                     InfoThreadComInf::ID_START,
startMsg);
}
```

```
// Info thread header...
#include "InfoThreadComInf.hpp"
#include "PostOffice.hpp"

class InfoThread : public InfoThreadComInf,
                   public osapi::ThreadFunctor
{
    ...
private:
    // Receives Start message from “some”
    //  thread
    void handleStartMsg(StartMsg* sm);
    void handleMsg(...);
};
```

Design: Broadcasting - who the receiver is *is* irrelevant

Broadcasting - who the receiver is *is* irrelevant

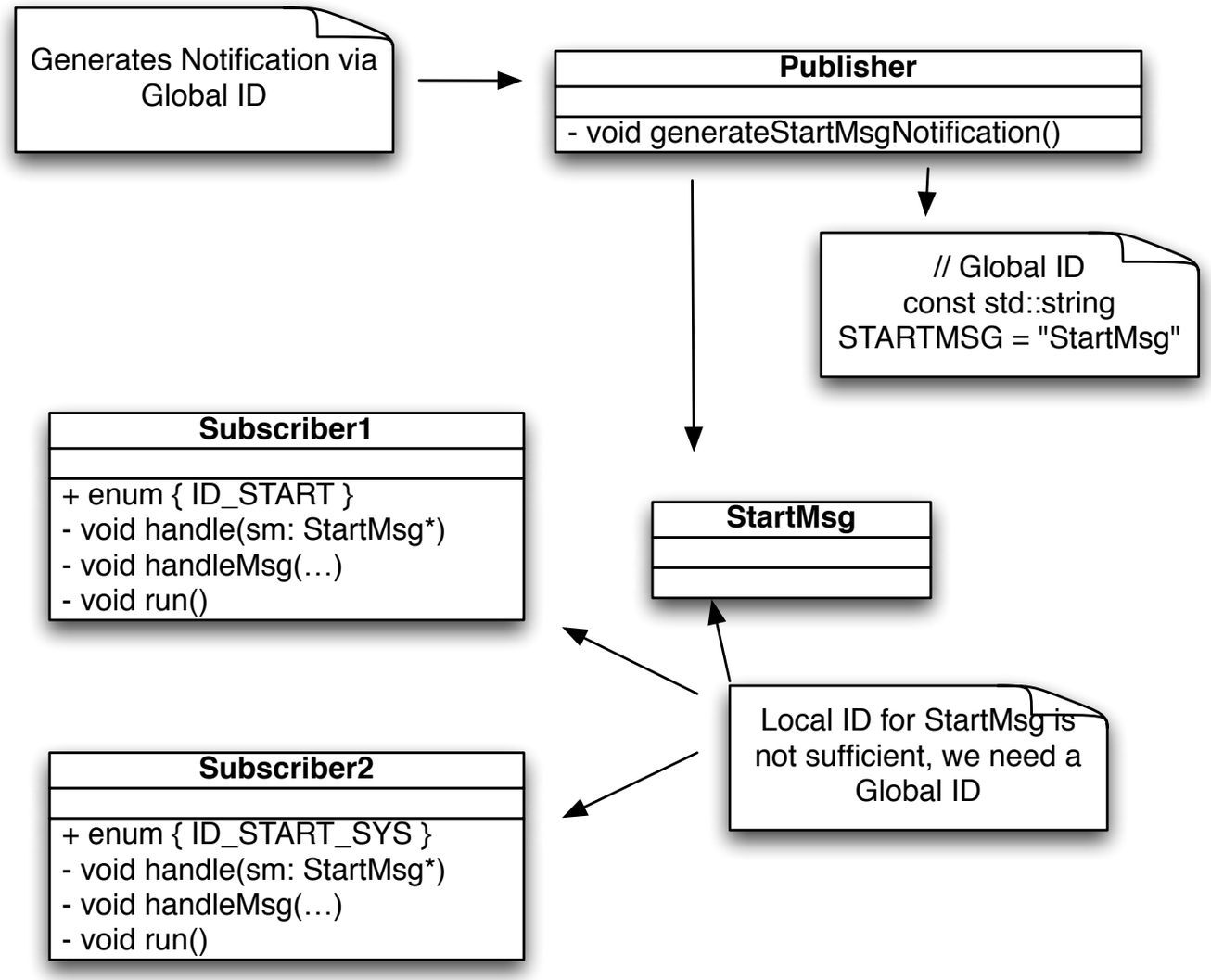
- Subscriber = Receiver
- Subscribes to a named message (std::string)
  - Who - By providing message queue pointer
  - How - By providing ID to receive when a message is ready



- Publisher
  - Notifies all subscribers (if any), each will receive the message being distributed with their own desired ID
- Achieves
  - Eliminates the need for the publisher to handle subscribers(s) (adding, removing)
  - Multiple subscribers may get the same message
- Requires
  - A MessageDistributionSystem is up and running prior to use
  - Using a singleton usage or parsing around pointer/reference
  - Messages must be ***Globally*** identifiable by strings
  - One way communication

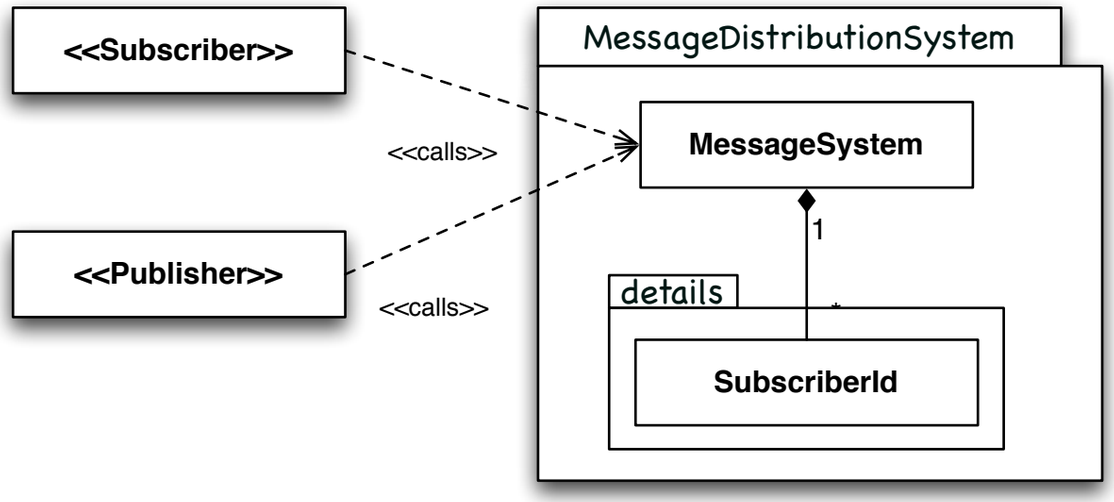
Broadcasting - who the receiver is *is* irrelevant

- Each recipient has a local ID
- We then need a global unique ID
  - Use the fully qualified name as a string
- Subscriber
  - Subscribes using own ***MsgQueue*** and ***Local ID*** when receiving a new Message by the name of ***Global ID***
- Publisher
  - Notifies by passing a new Message and associated ***Global ID***



Design: Broadcasting - who the receiver is *is* irrelevant Example

- Simple example using the MessageDistributionSystem directly




```
// Subscriber
MessageDistributionSystem::getInstance().subscribe(START_MSG, &mq_, ID_START);
...

void handleMsg(id, msg)
{
    switch(id_)
    {
        case ID_START:
            handleIdStart(msg);
    }
}

// Publisher
MessageDistributionSystem::getInstance().notify(START_MSG, startMsg);
```

Design: Broadcasting - who the receiver is *is* irrelevant Example

- *Common* header file(s) contains message structures & declaration of global string message ids
- Source file(s) contains the actual definition

Design: Broadcasting - who the receiver is *is* irrelevant Example

- *Common* header file(s) contains message structures & declaration of global string message ids
- Source file(s) contains the actual definition

```cpp
// cpp - file
const std::string START_MSG = "StartMsg";
const std::string LOG_ENTRY_MSG =
"LogEntryMsg";
```

```cpp
// hpp - file
struct StartMsg : public osapi::Message
{
    int x;
    int y;
};
extern const std::string START_MSG;

struct LogEntry : public osapi::Message
{
    char* filename_;
    int lineno_;
    std::string logStr_;
};
extern const std::string LOG_ENTRY_MSG;
```

Design: Broadcasting - who the receiver is *is* irrelevant Example

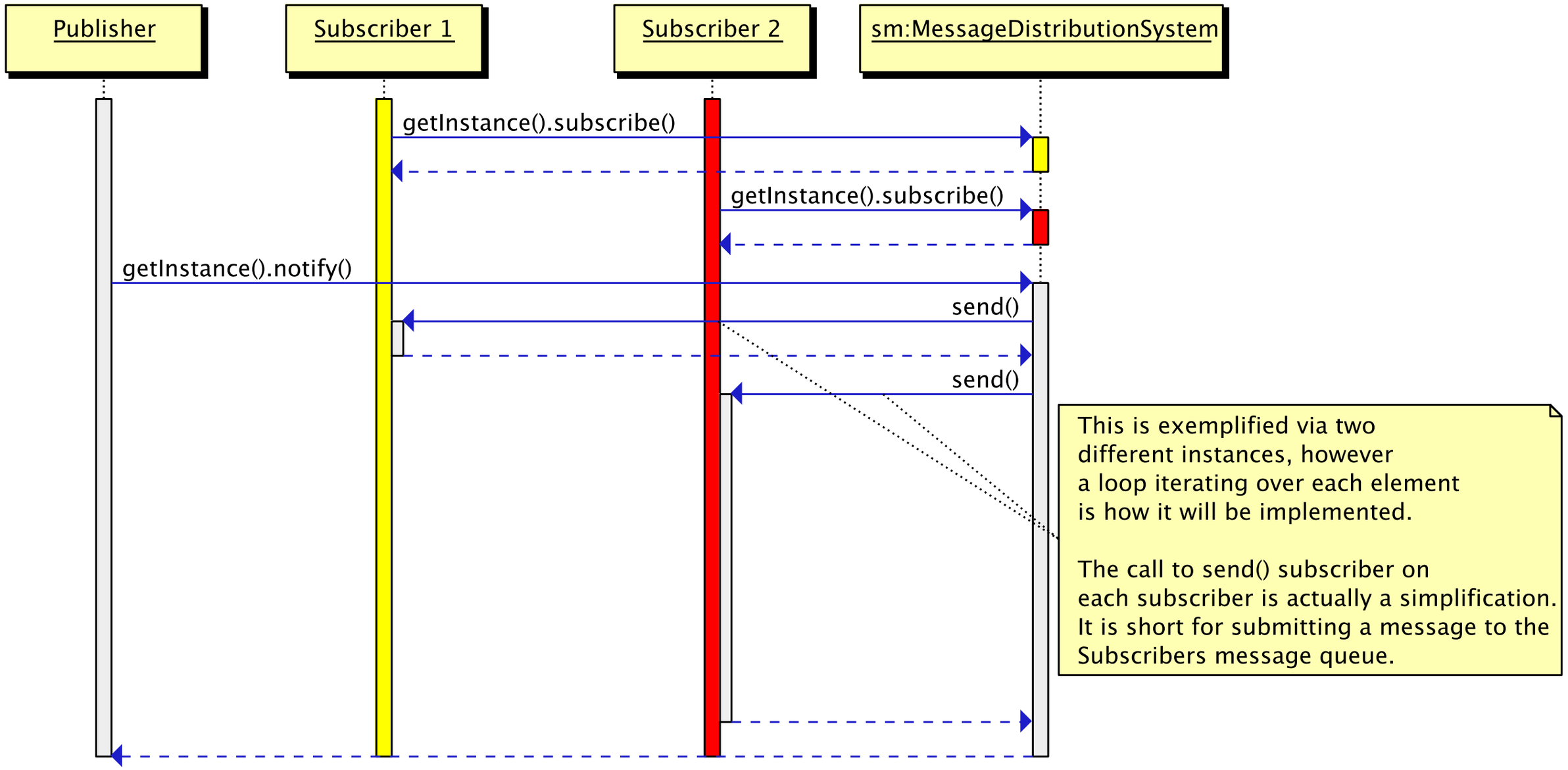
- *Common* header file(s) contains message structures & declaration of global string message ids
- Source file(s) contains the actual definition

```cpp
// cpp - file
const std::string START_MSG = "StartMsg";
const std::string LOG_ENTRY_MSG =
"LogEntryMsg";
```

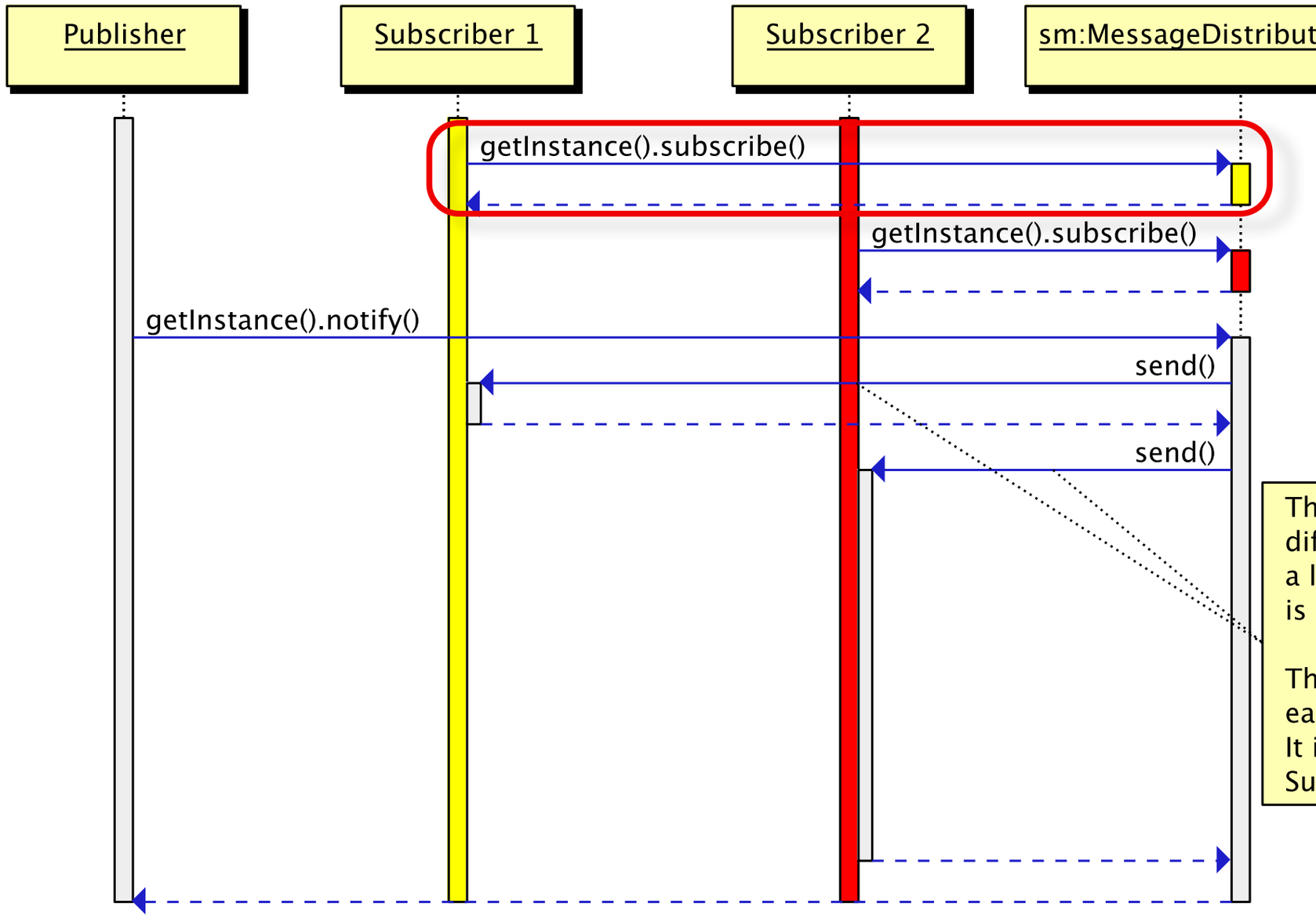
```cpp
// hpp - file
struct StartMsg : public osapi::Message
{
    int x;
    int y;
};
extern const std::string START_MSG;

struct LogEntry : public osapi::Message
{
    char* filename_;
    int lineno_;
    std::string logStr_;
};
extern const std::string LOG_ENTRY_MSG;
```

```cpp
// Subscriber
MessageDistributionSystem::getInstance().subscribe(START_MSG, &mq_, ID_START);
...
void Subscriber::handleMsg(id, msg)
{
    switch(id)
    {
        case ID_START:
            handleIdStart(static_cast<StartMsg*>(msg));
    }
}

// Publisher
MessageDistributionSystem::getInstance().notify(START_MSG, startMsg);
```

Message Distribution System in action



Message Distribution System in action

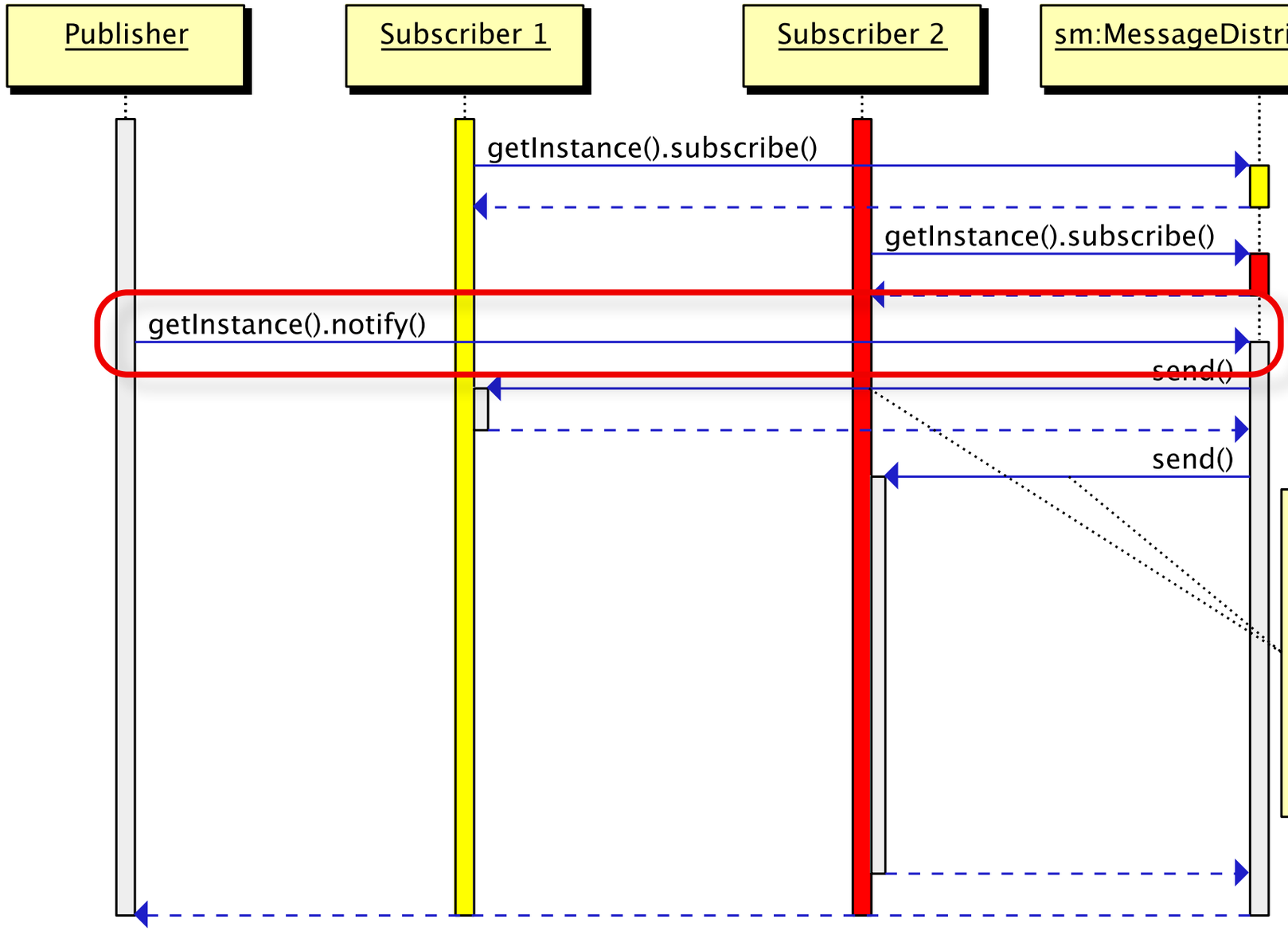


Subscribe on message

This is exemplified via two different instances, however a loop iterating over each element is how it will be implemented.

The call to send() subscriber on each subscriber is actually a simplification. It is short for submitting a message to the Subscribers message queue.

Message Distribution System in action

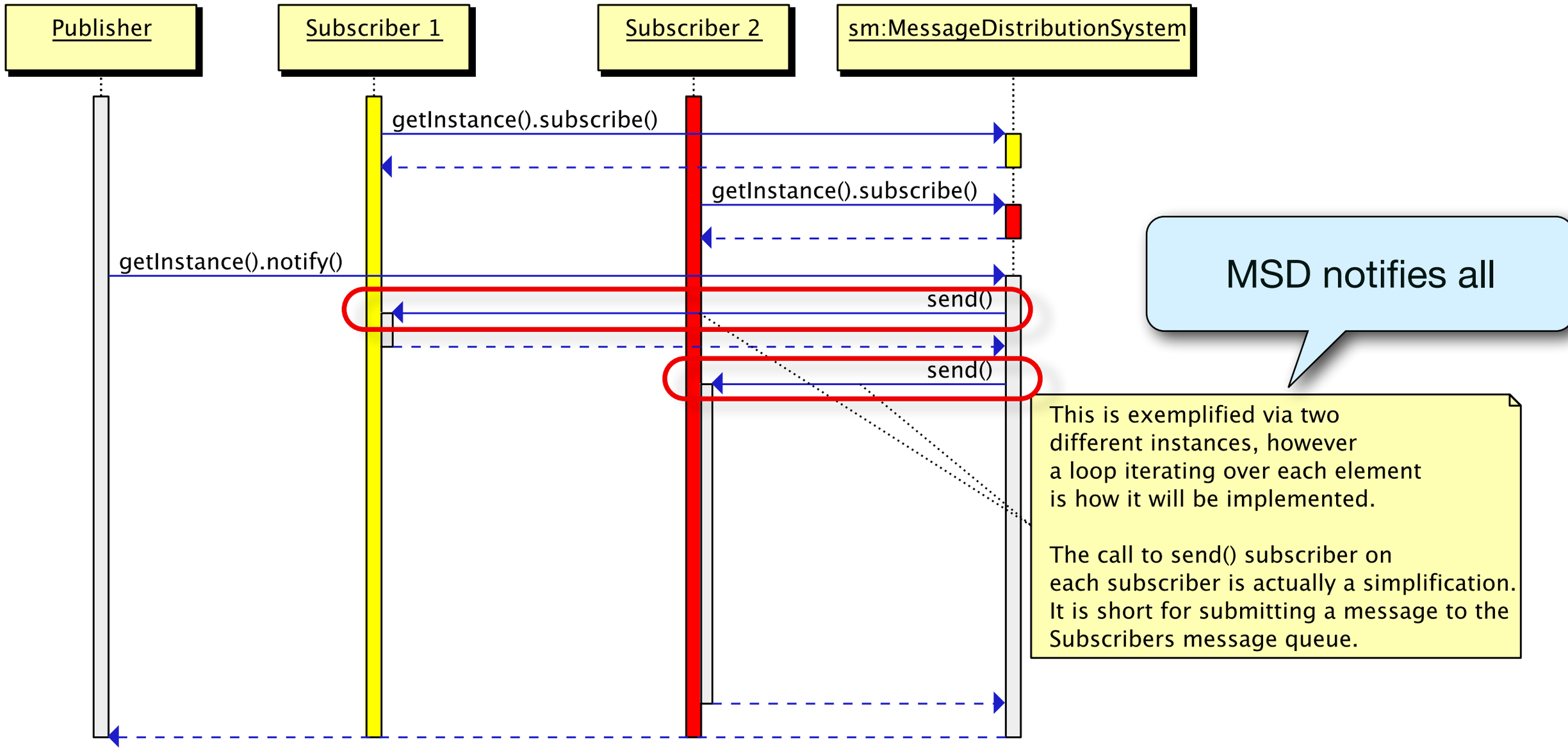


Publish message

This is exemplified via two different instances, however a loop iterating over each element is how it will be implemented.

The call to send() subscriber on each subscriber is actually a simplification. It is short for submitting a message to the Subscribers message queue.

Message Distribution System in action



Summary MDS

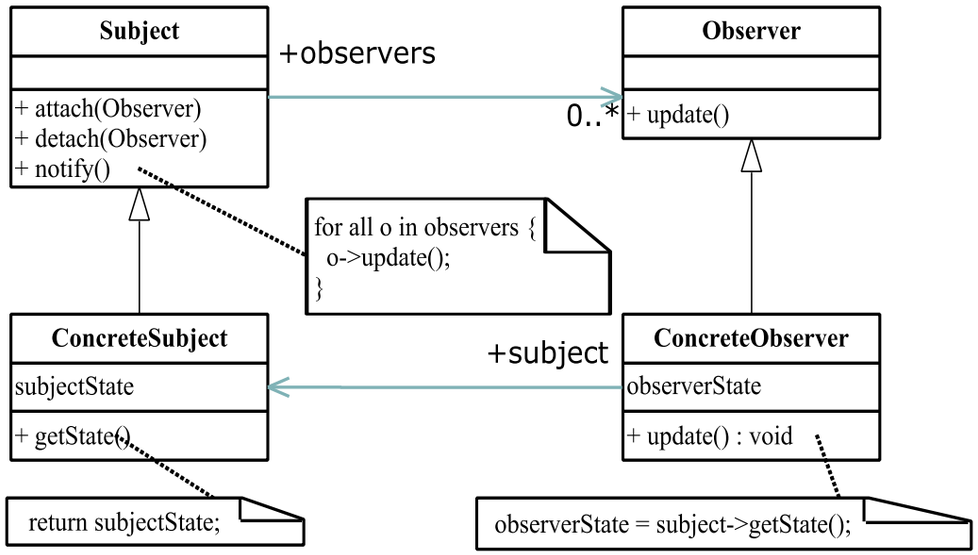
- **Broadcasting - Who the receiver is *is* irrelevant**
  - One way communication
  - Knowledge of each other irrelevant
  - Lower coupling
- Usage scenarios
  - Indication that something has happened
    - Log entry
    - New temperature value

Patterns

Publisher/Subscriber (or Observer)

Publisher/Subscriber (or Observer) pattern

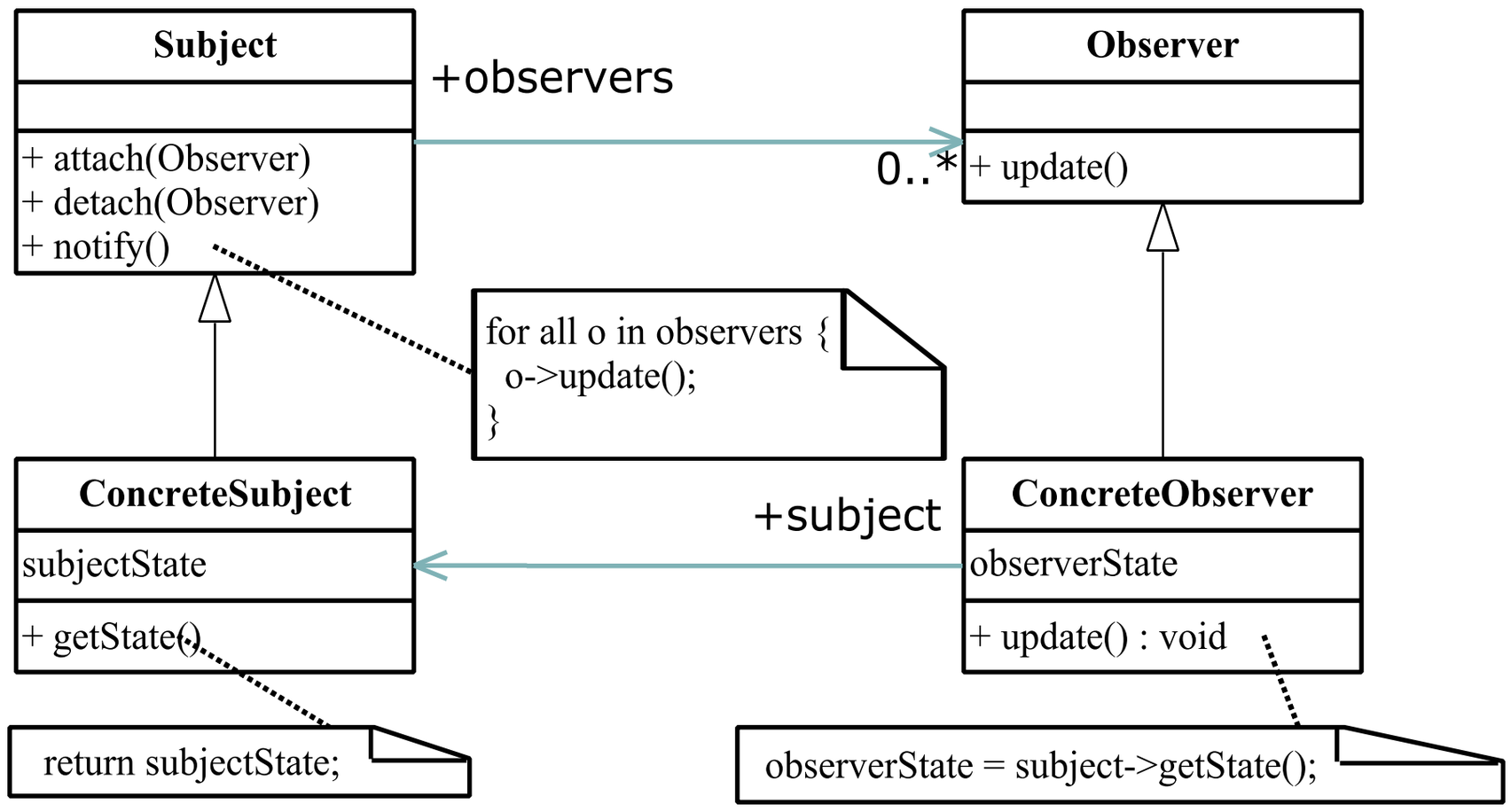
- Challenge
  - Needs notification when change occur (We do not want to *poll*)
  - One-to-many relation - Broadcasting
- Possible solution
  - ***Publisher/Subscriber (or Observer)***
- Usage could be
  - Message Distribution System
  - Button pushed in GUI -> Chain reaction (closing down + exiting program)
  - Sensor changes value -> various entities want to know



- Downsides
  - Updates cost throughout the system
  - A subscriber may take “long” time to handle incoming notification affecting the publisher

Publisher/Subscriber (or Observer) pattern

- UML Class diagram



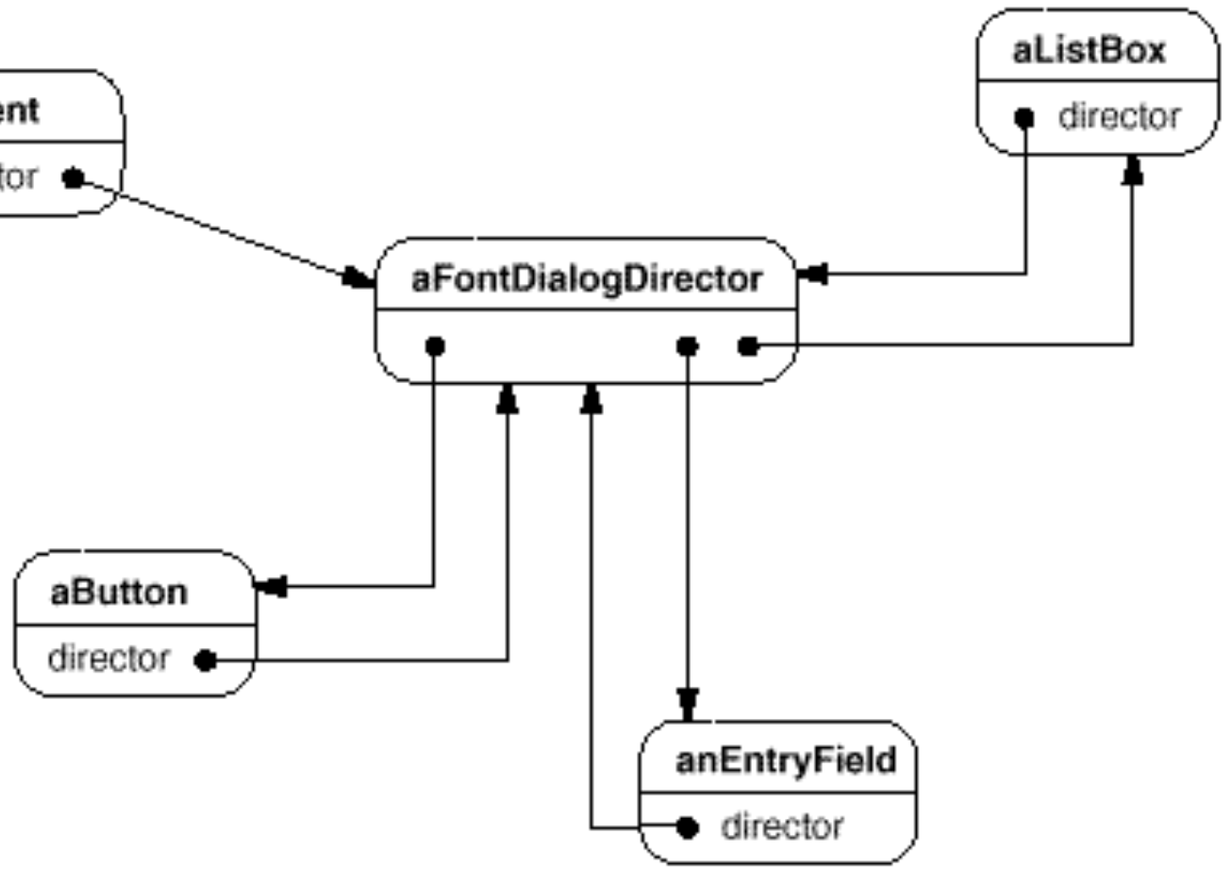
Mediator pattern

Mediator pattern

- Challenge
  - Need loose coupling and remove the need for objects (MsgQueues) to know each other
- Possible solution
  - ***Mediator***
- Usage could be
  - Message Distribution System
  - Graphics system - A draw() call is propagated to interesting parties
  - PostOffice
- Ups
  - Centralizes control
  - Focuses on how objects interact and not on behavior
  - Entities need not know about one another
- Downsides
  - Centralizes control

Mediator pattern

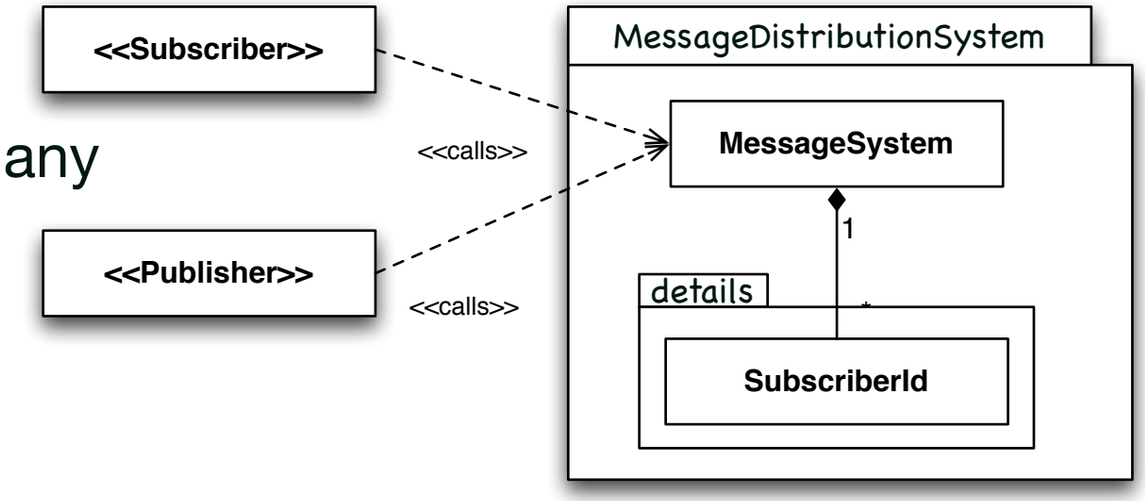
- UML Class diagram



Singleton pattern

Singleton pattern

- Challenge
  - System wide access to a given object ⇒Many pointers and/or references to be passed around
- Possible solution
  - ***Singleton***
- Usage could be
  - Message Distribution System
  - Config service
  - Log service
  - Any kind of application wide service



- Downsides
  - Global variable like
    - Serialized access needed
  - Lifetime
    - Who creates?
    - Who destroys and when?

Singleton pattern - Example

- Simple code example using the *static block initialization* approach
- Good
  - First access creates
  - Extremely easy to code and understand
  - No locks (in our approach)
- Downsides
  - First access creates - Multithreaded challenge
- Beware of “The double-checked locking” idiom
  - ***IT*** does not work!

Singleton pattern - Example

- Simple code example using the *static block initialization* approach
- Good
  - First access creates
  - Extremely easy to code and understand
  - No locks (in our approach)
- Downsides
  - First access creates - Multithreaded challenge
- Beware of “The double-checked locking” idiom
  - ***IT*** does not work!

```
// Singleton
class MessageDistributionSystem : osapi::NotCopyable
{
public:
    void subscribe(const std::string& msgId,
          osapi::MsgQueue* mq, unsigned long id);
    void unSubscribe(const std::string& msgId,
                     osapi::MsgQueue* mq, unsigned long id);

    static MessageDistributionSystem& getInstance()
    {
        static MessageDistributionSystem mds;
        return mds;
    }

private:
    MessageDistributionSystem() {}
};

// Subscriber
MessageDistributionSystem::
       getInstance().subscribe(START_MSG, &mq_, ID_START);
```