MESSAGE SYSTEM

AARHUS
UNIVERSITY
AARHUS UNIVERSITY SCHOOL OF ENGINEERING

AGENDA

- Current design and next step
- Decoupling current setup (2 different designs)
  - Specific receiver
  - Broadcasting - who the receiver is *is* irrelevant
- Patterns to use
  - Publisher/Subscriber (also called Observer)
    - (or Signal/slots)
  - Singleton
  - Mediator

CURRENT DESIGN AND NEXT STEP

CURRENT DESIGN AND NEXT STEP

- 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

CURRENT DESIGN AND NEXT STEP

- 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

1. **Communication via requests and confirms, two-way communication**
    - Knowledge of, or access to, message queue is relevant
    - Higher coupling, shared information
2. **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 - POSSIBLE DESIGN SOLUTIONS

1. Communication via requests and confirms, two-way communication
    - Possible approach - *Specific receiver design*
2. **Status information - indication**
    - Possible approach - *Broadcasting design*
        - Also called *Message Distribution System*

SPECIFIC RECEIVER

SPECIFIC RECEIVER

CASE

INPUT THREAD

- Send Messages to *InfoThread*

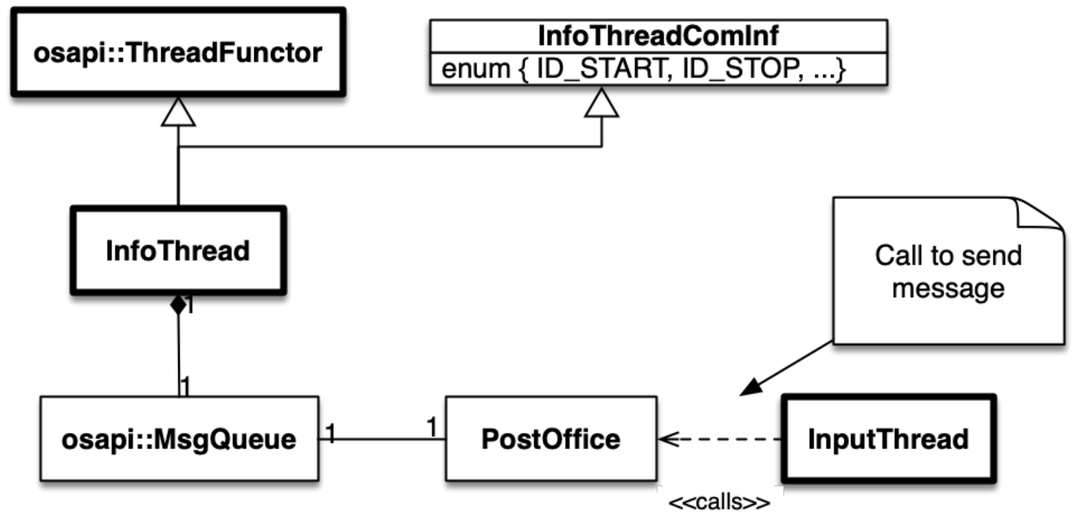
INFO THREAD

- Receive messages from amongst others *InfoThread*

SPECIFIC RECEIVER

DESIGN SOLUTION

- Improve upon the files included
  - Introduce another level (Mediator)
- Create a central postoffice
  - Send messages by naming (string format) the recipient
    - NO need to know the recepient's message queue
  - Or acquire a handle (speed up :-))



SPECIFIC RECEIVER

DESIGN

PSEUDO CODE FOR *INFO THREAD*

- Register at PostOffice using
  - Name/ID - "InfoThread" and associated *Message Queue*
  - Calls `PostOffice::register(std::string name, osapi::MsgQueue* mq)`
    - `PostOffice::register(InfoThreadComInf::QUEUE, &mq);`
    - Where InfoThreadComInf::QUEUE == "InfoThread"

SPECIFIC RECEIVER

DESIGN

PSEUDO CODE FOR *INFO THREAD*

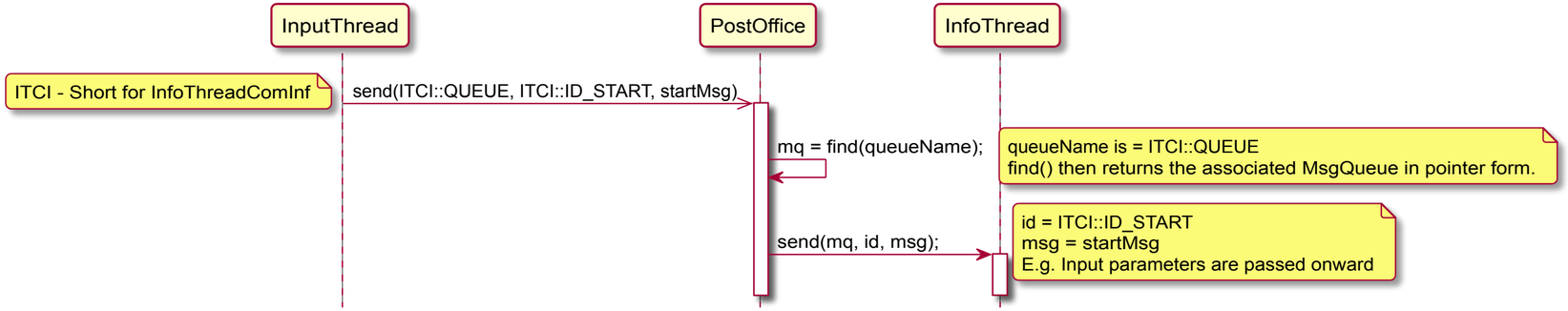
- Register at PostOffice using
  - Name/ID - "InfoThread" and associated *Message Queue*
  - Calls `PostOffice::register(std::string name, osapi::MsgQueue* mq)`
    - `PostOffice::register(InfoThreadComInf::QUEUE, &mq);`
    - Where InfoThreadComInf::QUEUE == "InfoThread"

PSEUDO CODE FOR *INPUT THREAD*

- Send data to *InfoThread* at some point
  - Uses PostOffice to pass on the message to recipient thread by it's ***Name/ID!***
  - Calls `PostOffice::send(std::string name, unsigned long id, osapi::Message* msg)`
    - `PostOffice::send(InfoThreadComInf::QUEUE, InfoThreadComInf::ID_START, startMsg)`

SPECIFIC RECEIVER

DESIGN - SEQUENCE DIAGRAM



SPECIFIC RECEIVER

DESIGN

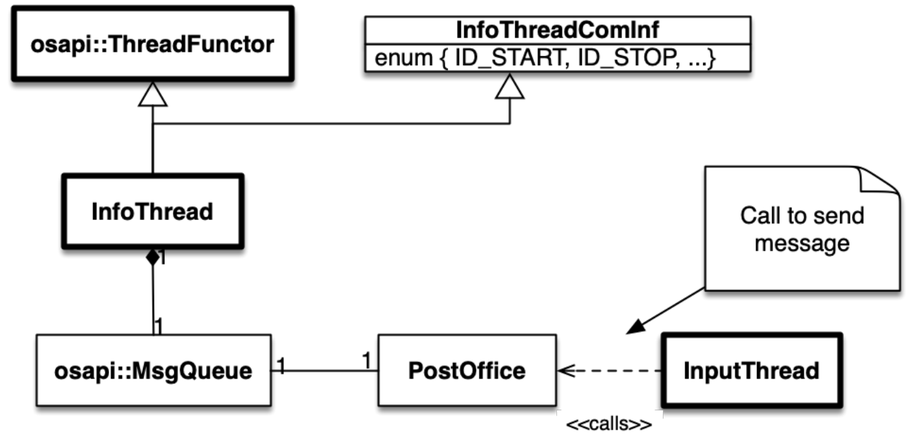
- Achieves
  - Low coupling since sender does not need to know receiver
  - Singleton usage or parsing around pointer/reference
  - Two-way communication possible
- Requires
  - A postoffice is up and running prior to use
  - Using a singleton or parsing around pointer/reference

SPECIFIC RECEIVER

SNIPPET OF POSSIBLE USAGE

- Communication identification is done using a separate header file

```
01  // InfoThreadComInf.hpp
02  struct InfoThreadComInf
03  {
04    static const std::string QUEUE;
05    enum { ID_START, ID_STOP, ... }
06  };
07
08  struct StartMsg : public osapi::Message
09  { ... };
```

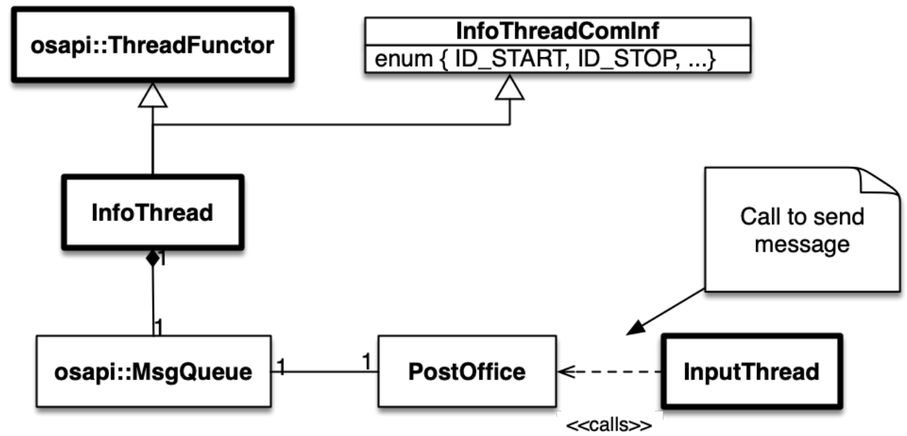


SPECIFIC RECEIVER

SNIPPET OF POSSIBLE USAGE

- Communication identification is done using a separate header file

```
01  // InfoThreadComInf.hpp
02  struct InfoThreadComInf
03  {
04    static const std::string QUEUE;
05    enum { ID_START, ID_STOP, ... }
06  };
07
08  struct StartMsg : public osapi::Message
09  { ... };
```




```
01  // In “some” thread about to send message...
02  #include "InfoThreadComInf.hpp"
03  #include "PostOffice.hpp"
04
05  void InputThread:HandleSomeMsg(...)
06  {
07    StartMsg* startMsg = new StartMsg;
08
09    PostOffice::send(InfoThreadComInf::QUEUE,
10                     InfoThreadComInf::ID_START,
11                     startMsg);
12  }
```

```
01  // Info thread header...
02  #include "InfoThreadComInf.hpp"
03  #include "PostOffice.hpp"
04
05  class InfoThread : public InfoThreadComInf,
06                     public osapi::ThreadFunctor
07  {
08      ...
09  private:
10    // Receives Start message from “some”
11    //  thread
12    void handleStartMsg(StartMsg* sm);
13    void handleMsg(...);
14  };
```

BROADCASTING - IRRELEVANT RECEIVER

BROADCASTING - IRRELEVANT RECEIVER

CASE

DIGITAL THERMOMETER THREAD

- Every second a new temperature is send to those who are interested

LOGGER THREAD

- Writes the value to the log

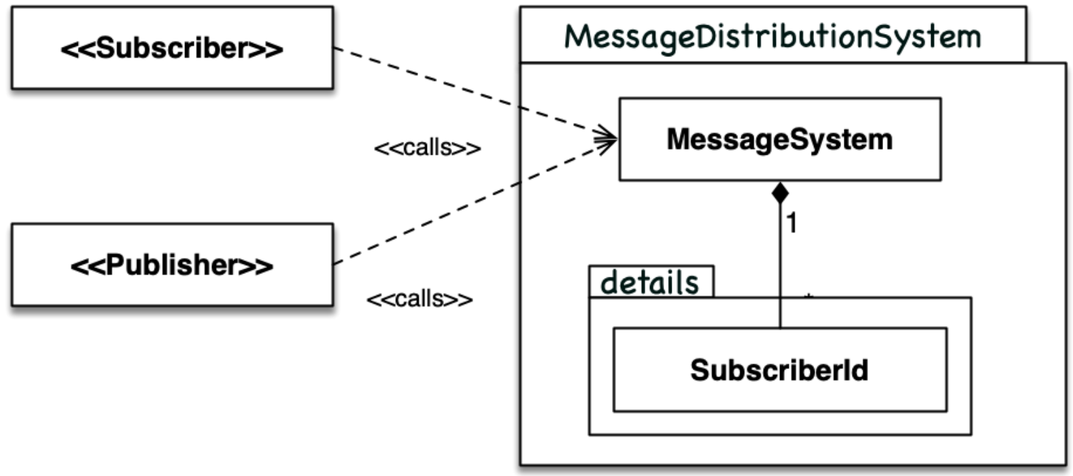
ALARM THREAD

- Check whether the value exceeds some threshold

BROADCASTING - IRRELEVANT RECEIVER

DESIGN

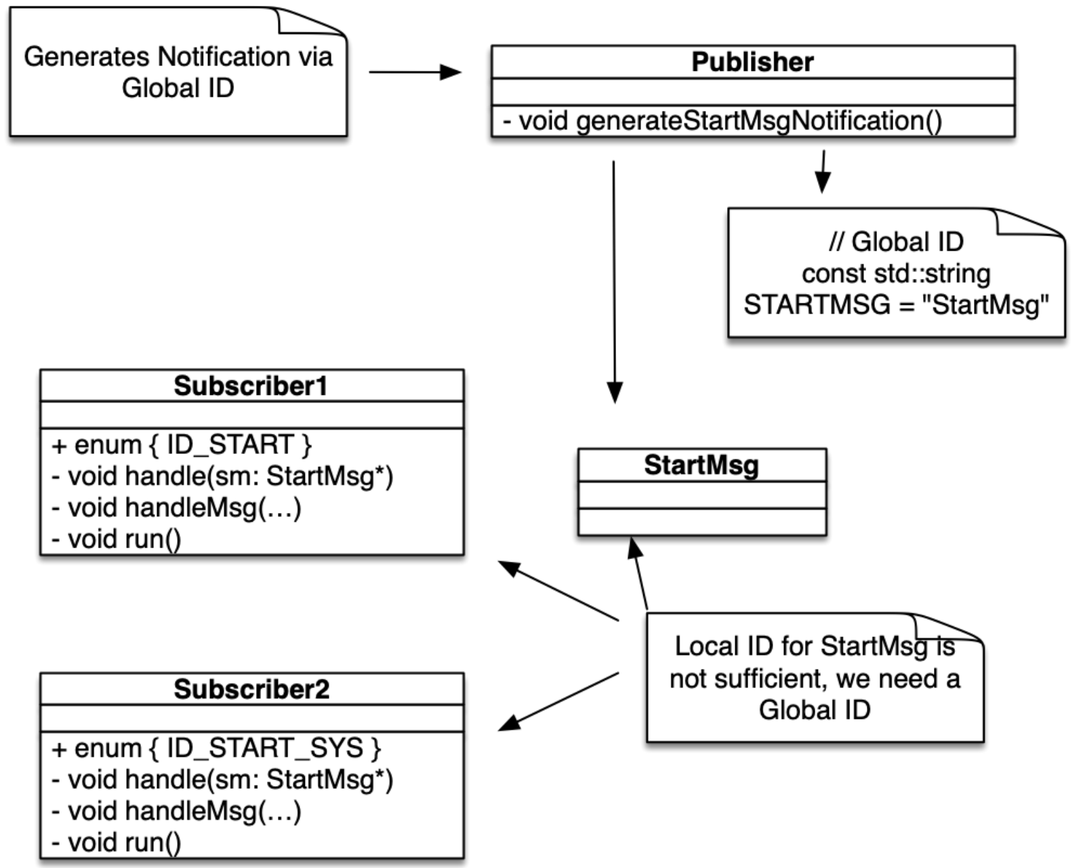
- Subscriber = Receiver
- Subscribes to a named message (std::string)
  - Who - By providing message queue pointer
  - How - By providing ID to receive when a message is ready
- Publisher
  - Notifies all subscribers (if any), each will receive the message being distributed with their own desired ID



BROADCASTING - IRRELEVANT RECEIVER

DESIGN

- 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



BROADCASTING - IRRELEVANT RECEIVER

DESIGN

PSEUDO CODE FOR *LOGGER THREAD* (*ALARM THREAD* IS SIMILAR)

- Subscribe at the MDS for temperature updates
  - Get access to the single instance of the MDS
  - Relevant parameters
    - Global ID of message to subscribe to
    - Pointer to own message queue
    - Id of message to receive in own message queue
      - NB! Receiver has their own separate IDs
  - Calls `MDS::subscribe(NEW_TEMP_VALUE_GLOBAL_ID, &mq, NEW_TEMP_VALUE_LOCAL_ID);`
- Upon receiving NEW_TEMP_VALUE_LOCAL_ID handle message appropriately
  - Write to log

BROADCASTING - IRRELEVANT RECEIVER

DESIGN

PSEUDO CODE FOR *LOGGER THREAD* (*ALARM THREAD* IS SIMILAR)

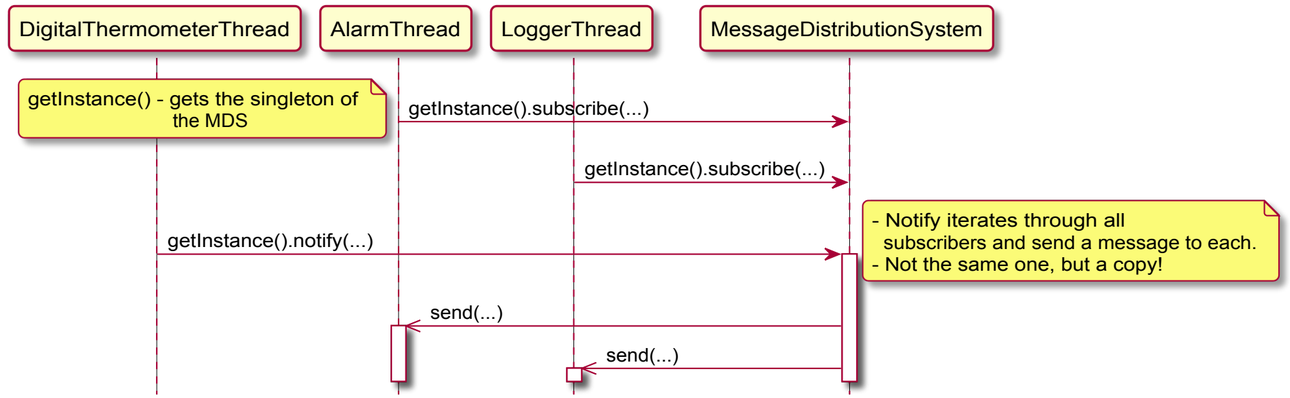
- Subscribe at the MDS for temperature updates
  - Get access to the single instance of the MDS
  - Relevant parameters
    - Global ID of message to subscribe to
    - Pointer to own message queue
    - Id of message to receive in own message queue
      - NB! Receiver has their own separate IDs
  - Calls `MDS::subscribe(NEW_TEMP_VALUE_GLOBAL_ID, &mq, NEW_TEMP_VALUE_LOCAL_ID);`
- Upon receiving NEW_TEMP_VALUE_LOCAL_ID handle message appropriately
  - Write to log

PSEUDO CODE FOR *DIGITAL THERMOMETER THREAD*

- Every second read temperature value from HW sensor
- Create message
- Broadcase message
  - Get access to the single instance of the MDS
  - Calls `MDS::send(NEW_TEMP_VALUE_GLOBAL_ID, tempMsg);`

BROADCASTING - IRRELEVANT RECEIVER

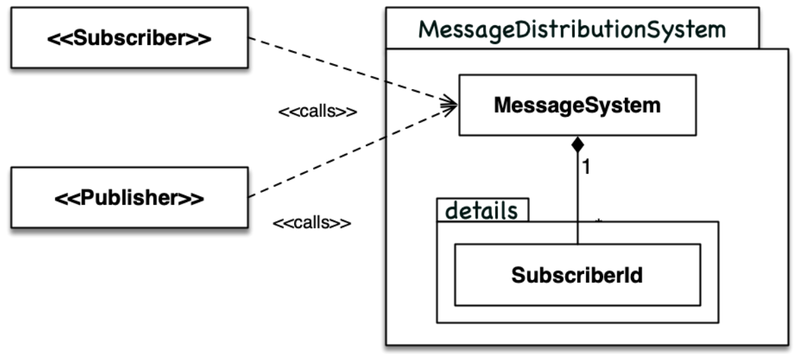
DESIGN



BROADCASTING - IRRELEVANT RECEIVER

DESIGN+IMPLEMENTATION

Simple example using the MessageDistributionSystem directly




```
01 // Subscriber
02 MessageDistributionSystem::getInstance().subscribe(START_MSG, &mq_, ID_START);
03 ...
04
05 void handleMsg(id, msg)
06 {
07   switch(id_)
08   {
09     case ID_START:
10       handleIdStart(msg);
11   }
12 }
13
14 // Publisher
15 MessageDistributionSystem::getInstance().notify(START_MSG, startMsg);
```

BROADCASTING - IRRELEVANT RECEIVER

DESIGN+IMPLEMENTATION

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

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

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

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

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

BROADCASTING - IRRELEVANT RECEIVER

DESIGN

- 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 - IRRELEVANT RECEIVER

SUMMARY

- 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

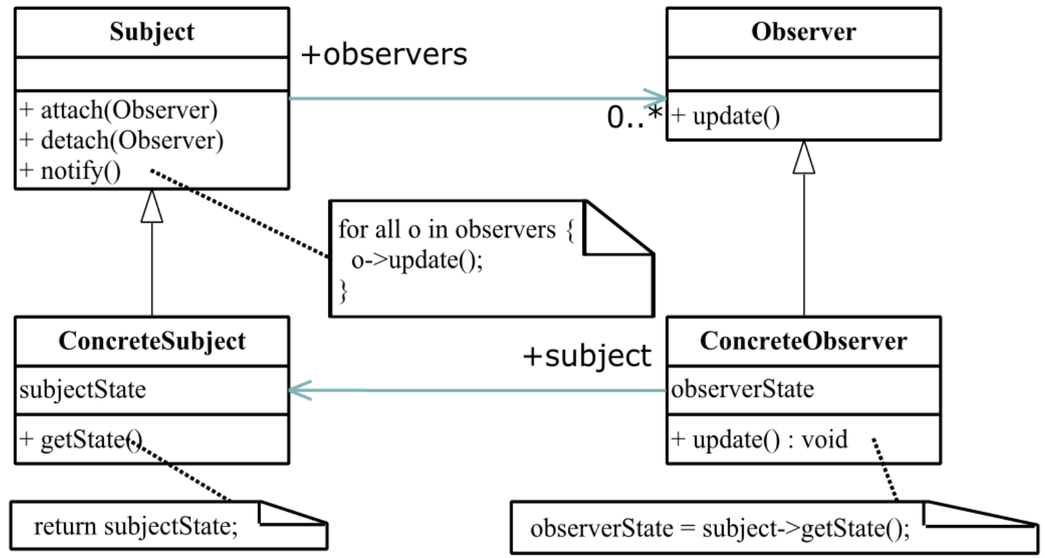
DESIGN PATTERNS

DESIGN PATTERNS

- Publisher/Subscriber
- Mediator
- Singleton

PUBLISHER/SUBSCRIBER

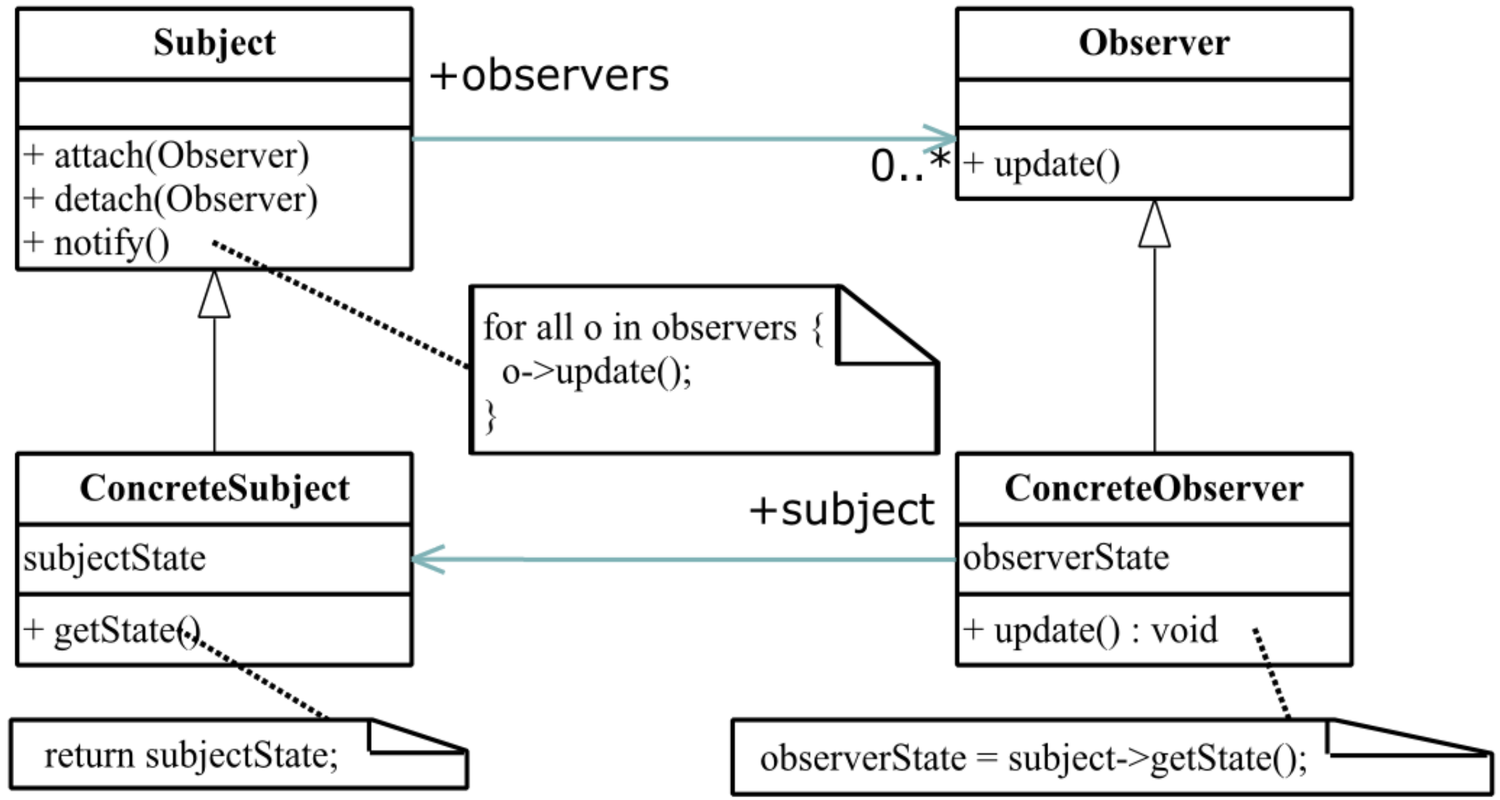
- Challenge
  - Needs notification when change occur (We do not want to poll)
  - One-to-many relation - Broadcasting
- Possible solution
  - *Publisher/Subscriber (or Observer)*
- Usage could be
  - Message Distribution System
  - Button pushed in GUI -> Chain reaction (closing down + exiting program)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

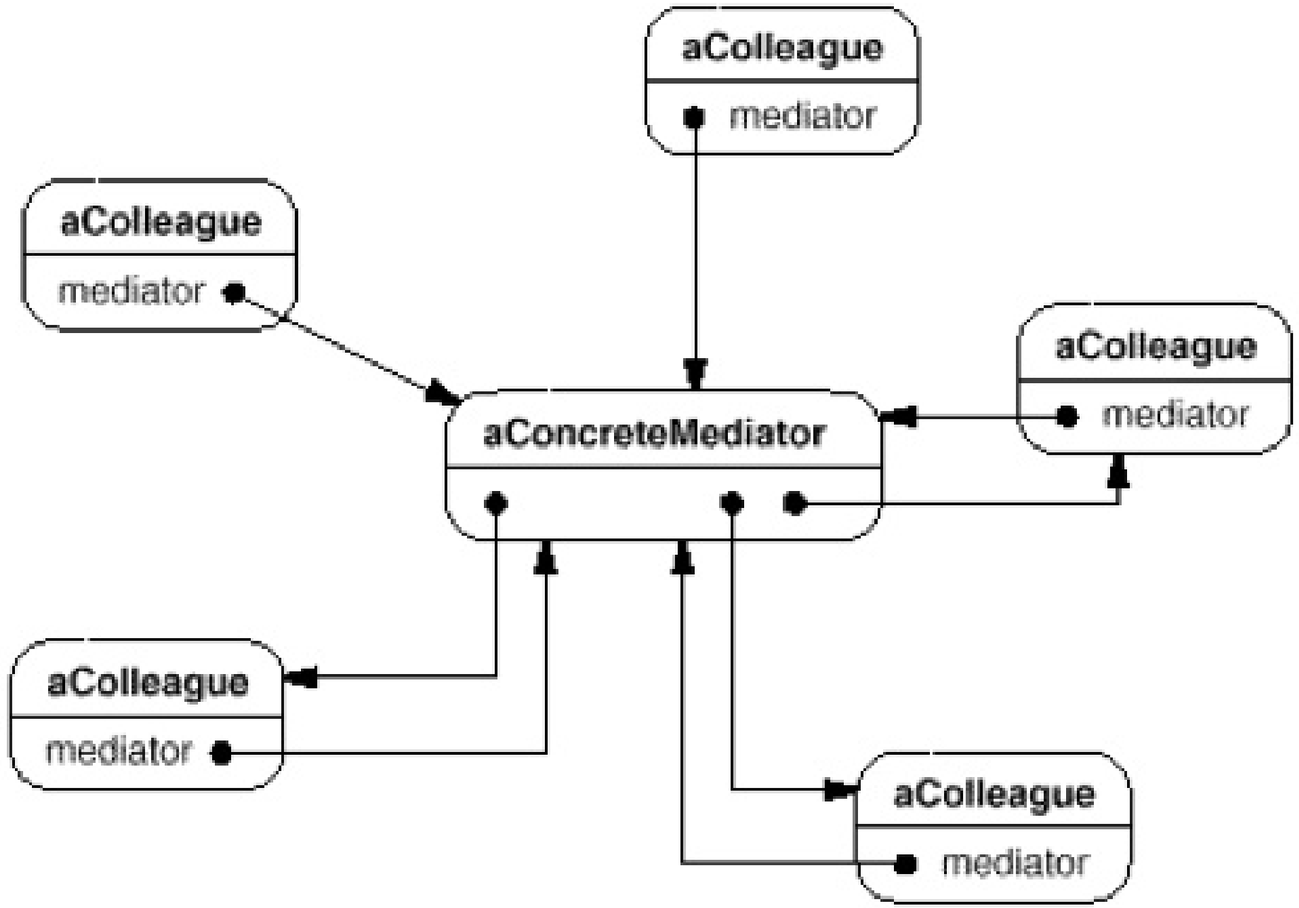
UML CLASS DIAGRAM



MEDIATOR

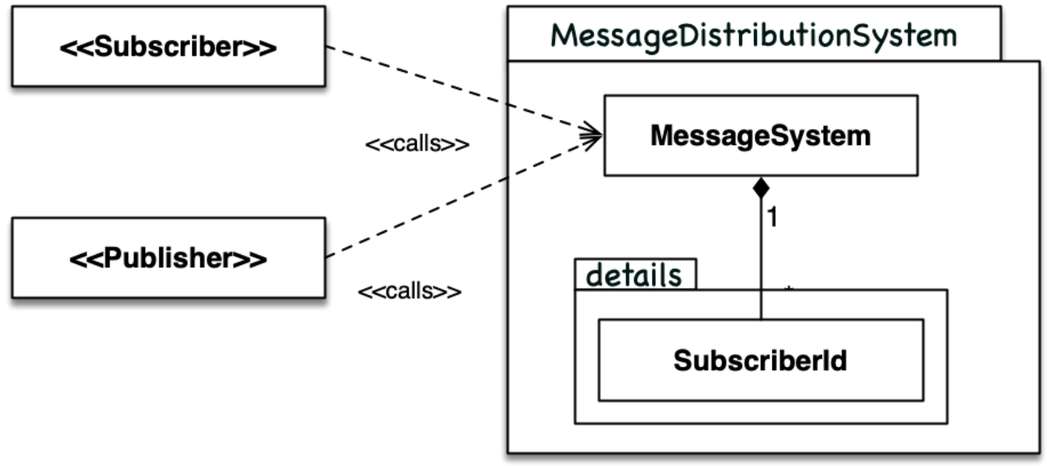
- Challenge
  - Need loose coupling and remove the need for objects (MsgQueues) to know each other
- Possible solution
  - *Mediator*
- Usage could be
  - Message Distribution System
  - Graphics system - A draw() call is propagated to interesting parties
  - PostOffice
- Ups
  - Centralizes control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

MEDIATOR


aColleague
mediator
aColleague
mediator
aConcreteMediator
aColleague
mediator
aColleague
mediator
aColleague
mediator

SINGLETON

- Challenge
  - System wide access to a given object
    ⇒Many pointers and/or references to be passed around
- Possible solution
  - *Singleton* - e.g. only one instance in the entire system!
- 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!

```
01 // Implemented using Meyers Singleton
02 class MessageDistributionSystem : osapi::NotCopyable
03 {
04 public:
05   void subscribe(const std::string& msgId,
06                  osapi::MsgQueue* mq,
07                  unsigned long id);
08
09   void unSubscribe(const std::string& msgId,
10                    osapi::MsgQueue* mq,
11                    unsigned long id);
12
13   static MessageDistributionSystem& getInstance()
14   {
15     static MessageDistributionSystem mds;
16     return mds;
17   }
18
19 private:
20   MessageDistributionSystem() {}
21 };
22
23 // Subscriber
24 MessageDistributionSystem::
25    getInstance().subscribe(START_MSG, &mq_, ID_START);
```