Wiselib School 2010

Session 3
Combining Algorithms Into Higher-Level Systems

Henning Hasemann

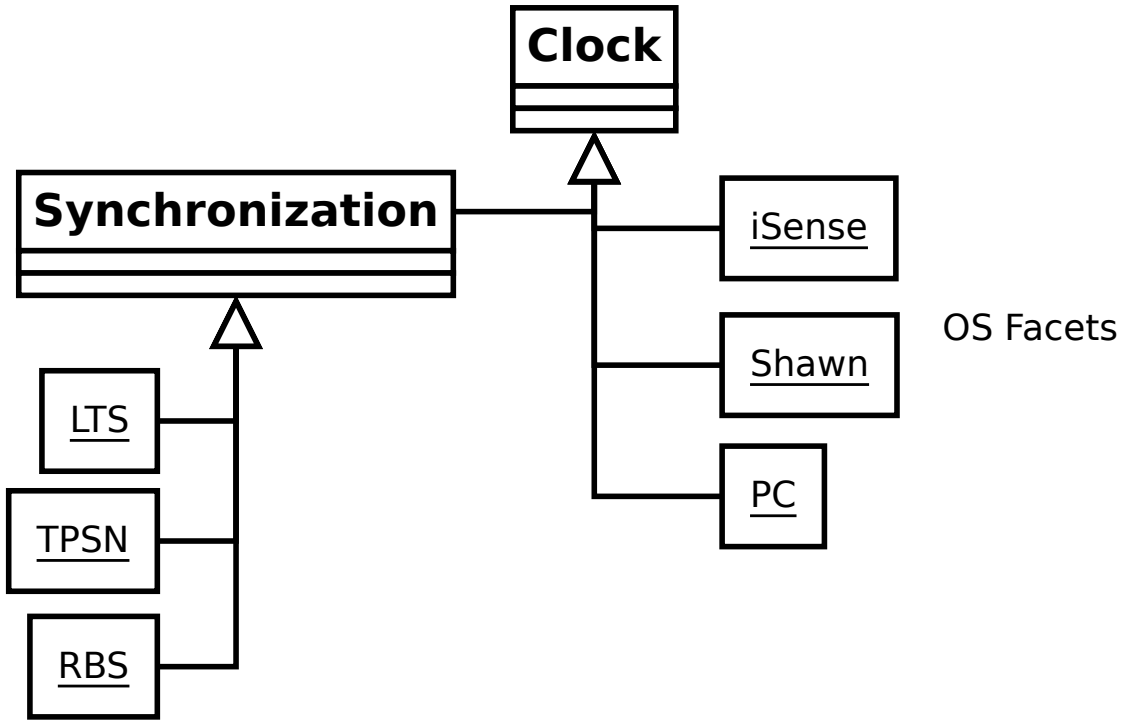
Institute of Operating Systems and Computer Networks
Handout on http://www.ibr.cs.tu-bs.de/alg/winterschool/

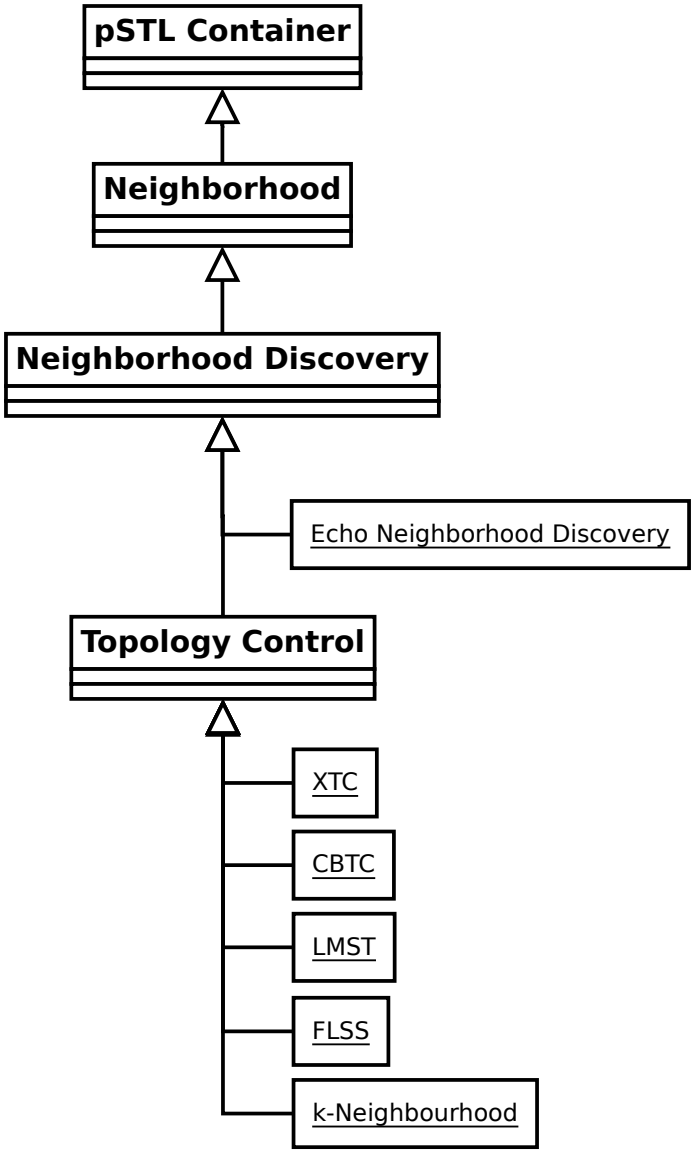
October 2010

Outline

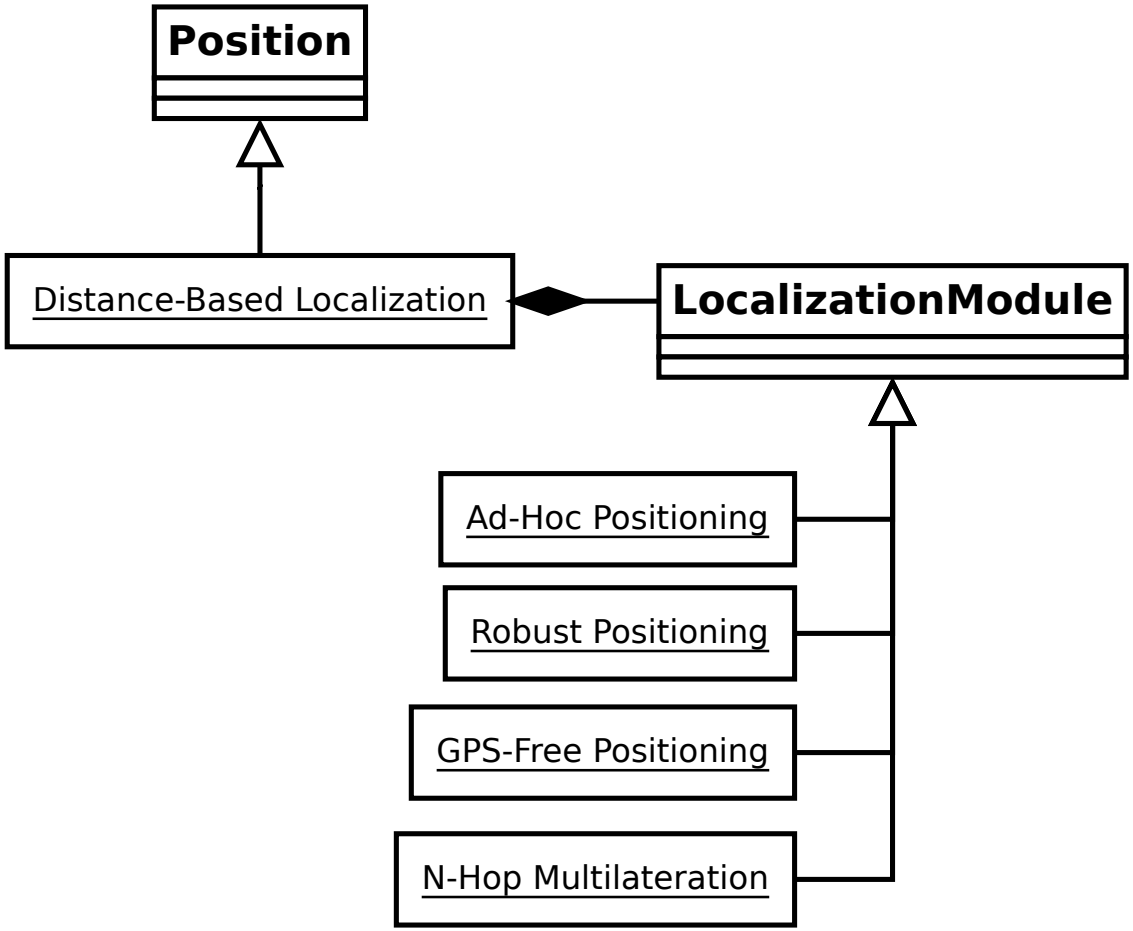
Algorithms in the Wiselib

- There are a lot already
- Contributed by different universities
- Cover a broad spectrum of fields
- A concept per algorithm category
- Partly these are OS facet concepts

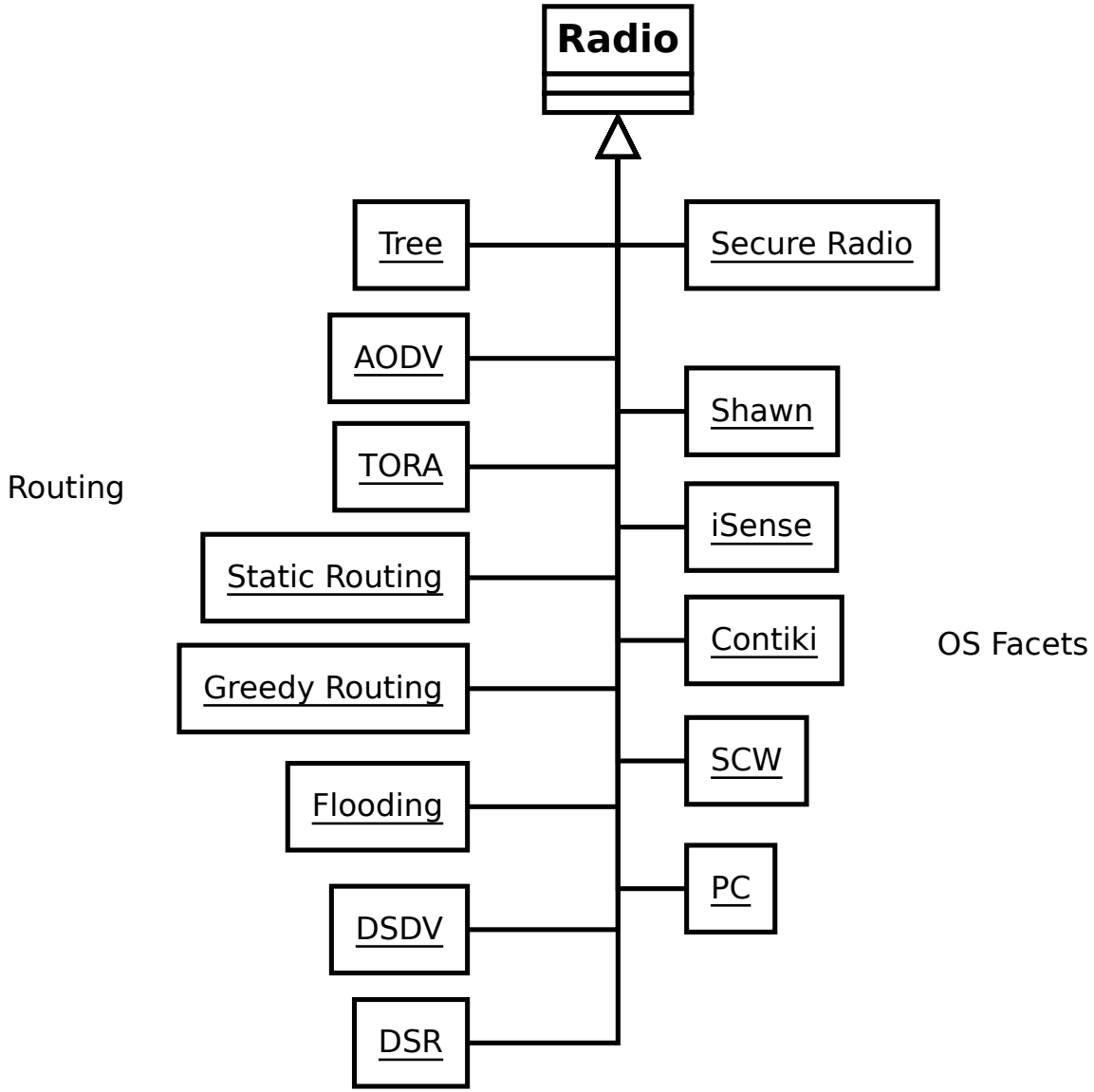

Clock
Synchronization
iSense
Shawn
PC
OS Facets
LTS
TPSN
RBS



- A neighborhood implements the container concept
- I.e. you can iterate over it
- Does not mean it is static
- Examples of implemented topology control algorithms which provide a neighborhood



- Distance-based localization implements Position
- Can use different localization modules for estimation


Radio
Tree
Secure Radio
AODV
Shawn
TORA
iSense
Routing
Static Routing
Contiki
OS Facets
Greedy Routing
SCW
Flooding
PC
DSDV
DSR

There is much more...

- We just scratched the surface
- Clustering (5)
- Graph Coloring (4)
- General graph algorithms (2)
- Ciphers (4)
- Duty Cycling (1)
- Metrics (2)
- Tracking (1)

Outline

Outline

Problem Statement

- OS facets are interfaces to the underlying operating system
- They reflect hardware components
- The exact **initialization depends on the platform**
- Hardware is **there all the time**, so should be its facet

- Algorithms use OS facets and other algorithms to accomplish a task
- They control facets and other algorithms
- The **initialization is different for each algorithm**
- I.e. different template parameters
- So it cannot be automated
- Algorithms should be able to **re-initialize** other algorithms

Algorithm Init Methods

```
1 template<typename OsModel_P,
2          typename DataStructure_P,
3          typename Radio_P = typename OsModel_P::Radio, ...>
4 class MyAlgorithm {
5   public:
6     typedef OsModel_P OsModel;
7     typedef Radio_P Radio;
8     typedef DataStructure_P DataStructure;
9
10    int init( Radio& radio, ...) {
11      radio_ = &radio_; ...
12    }
13
14    int init() { ... };
15    int destruct() { ... };
16
17  private:
18    Radio::self_pointer_t radio_;
19    DataStructure data_structure_;
20 };
```

- `init(...)` for initialization
- `init()` for re-initialization / “reset”
- `destruct()` for destruction (stopping all activity)
- Data structures held as private members

Passing Os Facets to an Algorithm

- Facets **can not** be constructed by algrithms, only **pointers** are held
- Use provided type `FACET::self_pointer_t`, **not** `FACET*`

```
1   Radio::self_pointer_t radio_;
```

- Use like ordinary pointer

```
1   radio_->id()
```

- Facets are passed as reference in `init(...)` method

```
1   init( Radio& radio, ...)
2   { radio_ = &radio_; ... }
```

- `init(...)` is called by constructing **application**
- Other algorithms do **not** call `init(...)` $\Rightarrow$ parameters unknown

Init and Destruct

- **Each** algorithm provides `init()` and `destruct()` method

```
1   int init();
2   int destruct();
```

- `init()` method
  - Set all internal states to initial values
  - Re-start algorithm
  - **Can be** called by other algorithms
- `destruct()` method
  - Stop algorithm: **No** timers, **no** message reception, ...
  - Turn off facets, if possible: E.g., `radio_->disable_radio()`
  - **Can be** called by other algorithms

Passing Data Structures

- Data structures are created by algorithm ⇒ **not** passed in `init(...)`
- Passed as template parameters, held as private data member

```
1 template<typename OsModel_P,
2          typename NodeList_P, ...>
3 class MyAlgorithm {
4   NodeList_P node_list_;
```

- Expected types for datastructures are part of **documentation**
- Example: Passing list of node ids
  - List of node ids from pSTL

```
1 #include "util/pstl/list_static.h"
2 typedef wiselib::list_static<Os, Os::Radio::node_id_t, 20> StaticNodeList;
3 typedef wiselib::MyAlgorithm<Os, StaticNodeList, ...> Algorithm;
```

  - List of node ids from STL

```
1 #include <list>
2 typedef std::list<Os::Radio::node_id_t> NodeListStl;
3 typedef wiselib::MyAlgorithm<Os, NodeListStl, ...> Algorithm;
```

Outline

Problem

When implementing e.g. a Radio, you have to provide these methods:

```
1 ...
2 template<class T, void (T::*TMethod)(node_id_t, size_t, block_data_t*)>
3 int reg_recv_callback(T *obj_pnt);
4
5 int unreg_recv_callback(int cid);
6 ...
```

- Provide method for registration of callbacks
- Provide method to unregister a callback
- Provide possibility to register multiple callbacks

→ Tedious, repeating work

Callback Simplification

→ Simplification

- We provide base classes where this is implemented
- Derive from such a base class
- Call `notify_xxx(...)`
- Callback registration is done automatically

- Only **some** base classes supported, but easy to add own/more ones

Example

- Radio base class

```
1 template<OsModel_P, node_id_t, size_t, block_data_t>
2 class RadioBase
3 {
4   template<class T, void (T::*TMethod)(node_id_t, size_t, block_data_t*)>
5   int reg_recv_callback( T *obj_pnt ) { ... }
6
7   int unreg_recv_callback( int ) { ... }
8
9   void notify_receivers( node_id_t from, size_t len, block_data_t *data ) { ... }
10 };
```

- Usage

```
1 template<OsModel_P, Radio_P, ...>
2 class MyRadioOrRouting
3   : RadioBase<OsModel_P, Radio_P::node_id_t, ...>
4 {
5   ...
6   int on_receive(node_id_t from, size_t len, block_data_t *data) {
7     notify_receivers( from, len, data );
8     return OsModel::SUCCESS;
9   }
10 }
```

Outline

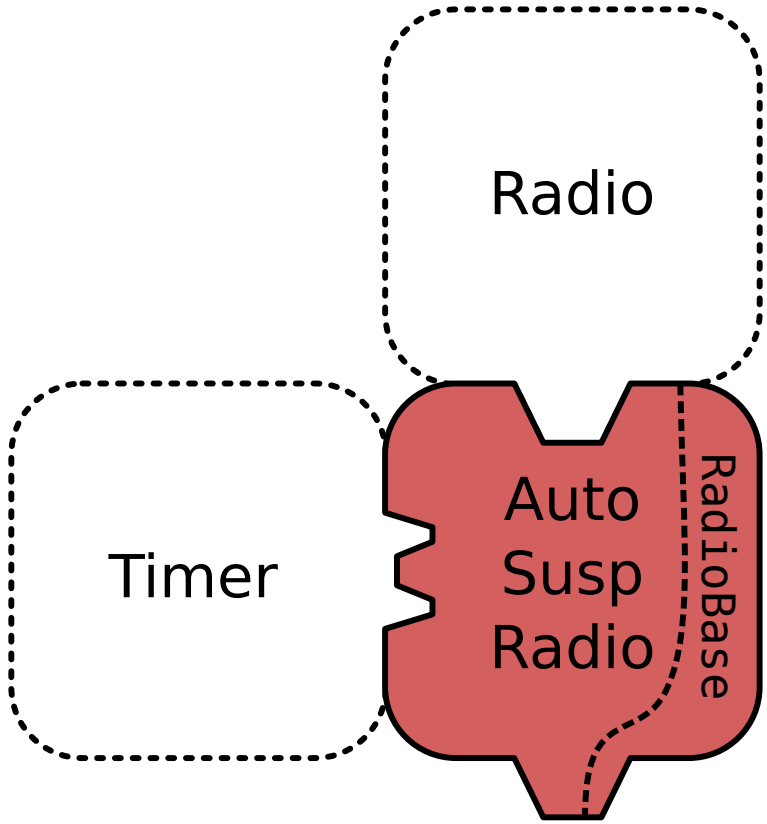
The Requirement

- Assume a node that sporadically outputs (sends) bursts of sensor data
- Also assume the node will not receive but only send data
- We want to save power when no data is being sent
- By turning off the radio when its idle for some time
- Which would naturally only make sense for certain nodes, which do not need to fulfill routing tasks

→ Build an "adaptor" on a radio

- Hand down all messages to the adapted radio
- When idle for a certain time, turn off

AutoSuspendRadio



- Our component will use a Radio and a Timer
- And behave like a radio
- We use `RadioBase` to make our lives easier

Idea

- Whenever a message is being sent, increase a value `packets_sent_`
- Also start a timer, to call `on_time`
- As userdata pass the value `packets_sent_` had when starting the timer
- If `userdata` $\neq$ `packets_sent_`, a new message has been sent in the meantime
  - $\rightarrow$ In this case, nothing should happen
- Else
  - `packets_sent_` has not increased
  - $\rightarrow$ The radio has been idle the whole time
  - $\rightarrow$ Disable the radio
- If disabled, and `send(...)` gets called, enable

The Concept We Have To Implement

- Implement the radio concept

```
1 concept RadioFacet {
2   typedef ... OsModel;
3   typedef ... node_id_t;
4   typedef ... block_data_t;
5   typedef ... size_t;
6   typedef ... message_id_t;
7
8   enum SpecialNodeIds { BROADCAST_ADDRESS = ..., NULL_NODE_ID = ...   };
9   enum Restrictions { MAX_MESSAGE_LENGTH = ...   };
10
11  int enable_radio();
12  int disable_radio();
13
14  int send(node_id_t receiver, size_t len, block_data_t *data );
15  node_id_t id();
16  template<class T, void (T::*TMethod)(node_id_t, size_t, block_data_t*)>
17  int reg_recv_callback( T *obj_pnt );
18  int unreg_recv_callback( int idx );
19 };
```

Implementation Example (1/7)

1 Template parameters

```
1 template<
2   typename OsModel_P,
3   typename Radio_P,
4   typename Timer_P = OsModel_P::Timer,
5   OsModel_P::size_t Interval_P = 10000
6>
7 class AutoSuspendRadio : RadioBase<OsModel_P, Radio_P::node_id_t, ...> {
8   public:
9     typedef OsModel_P OsModel;
10    typedef Radio_P Radio;
11    typedef Timer_P Timer;
12    static const OsModel::size_t interval = Interval_P;
```

- `OsModel_P` is always the first parameter
- Hold on to the naming convention
- Followed by data structures, then facets
- Note that you can also expect constant values like numbers
- Use default parameters, e.g. users will often use `OsModel::Timer` anyway
- But do not force them to
- `typedef` your parameters!

Implementation Example (2/7)

3. Implement types of the concept

```
1   typedef Radio::node_id_t node_id_t;
2   typedef Radio::block_data_t block_data_t;
3   typedef Radio::size_t size_t;
4   typedef Radio::message_id_t message_id_t;
5
6   typedef AutoSuspendRadio<OsModel_P, Radio_P, Timer_P, Interval_P> self_type;
7
8   enum SpecialNodeIds {
9     BROADCAST_ADDRESS = Radio::BROADCAST_ADDRESS,
10    NULL_NODE_ID = Radio::NULL_NODE_ID
11  };
12
13  enum Restrictions {
14    MAX_MESSAGE_LENGTH = Radio::MAX_MESSAGE_LENGTH
15  };
```

- Avoid mixing sources of the types/constants
  Having `block_data_t` from `Radio` but `size_t` from `OsModel` is inconsistent
- Prefer aliases of parameters as sources
- Naturally, if you implement a radio, prefer radio as source

Implementation Example (3/7)

4. Hold references to used facets

```
1   private:
2     Radio::self_pointer_t radio_;
3     Timer::self_pointer_t timer_;
```

   - Naming convention: lowercase, trailing underscore
   - Always use `pointer_t`, not `Facet*`

5. Other private variables.
   Here: Counter for sent messages & enabled flag

```
1     bool enabled_;
2     OsModel::size_t packets_sent_;
```

   - Use `OsModel::size_t` for counting

Implementation Example (4/7)

6. Provide `init(...)`

```
1   public:
2     int init(Radio& radio, Timer& timer) {
3       radio_ = &radio;
4       timer_ = &timer;
5
6       radio_->disable(); // turn of to save power immediatly
7       radio_->template reg_regv_callback<self_type, &self_type::on_receive>(this);
8
9       packets_sent_ = 0;
10      enabled_ = false;
11      return OsModel::SUCCESS;
12    }
```

- Parameters: Same order as in template parameters
- Expect references, not pointers or copies

Implementation Example (5/7)

7 Destruction & Reset

```
1     int destruct() {
2       enabled_ = false;
3       disable_radio();
4       return OsModel::SUCCESS;
5     }
6
7     int init() {
8       packets_sent_ = 0;
9       enabled_ = false;
10      disable_radio();
11      return OsModel::SUCCESS;
12    }
```

Implementation Example (6/7)

8. Enable/Disable

```
1     void enable_radio() {
2       enabled_ = true;
3     }
4
5     void disable_radio() {
6       radio_->disable_radio();
7       enabled_ = false;
8     }
```

9. Forward other calls to the given radio.

```
1     node_it_t id() { return radio_.id(); }
2
3     void on_receive(Radio::node_id_t sender, Radio::size_t size, Radio::
          block_data_t* buffer) {
4       notify_receivers(sender, size, buffer);
5     }
```

- `notify_receivers` is from `RadioBase`

Implementation Example (7/7)

9 Sending data

```
1    int send(node_id_t receiver, size_t len, block_data_t *data) {
2      if(!enabled_) {
3        return OsModel::ERR_UNSPEC;
4      }
5
6      radio_->enable_radio();
7      timer_->template set_timer<self_type, &self_type::on_time>(interval, this, (
           void*)packets_sent_);
8      return radio_.send(receiver, len, data);
9    }
```

10 The timer callback

```
1    void on_time(void* userdata) {
2      if(!enabled_) {
3        return;
4      }
5
6      if((size_t)userdata == packets_sent_) {
7        radio_->disable();
8      }
9    }
```

Outline

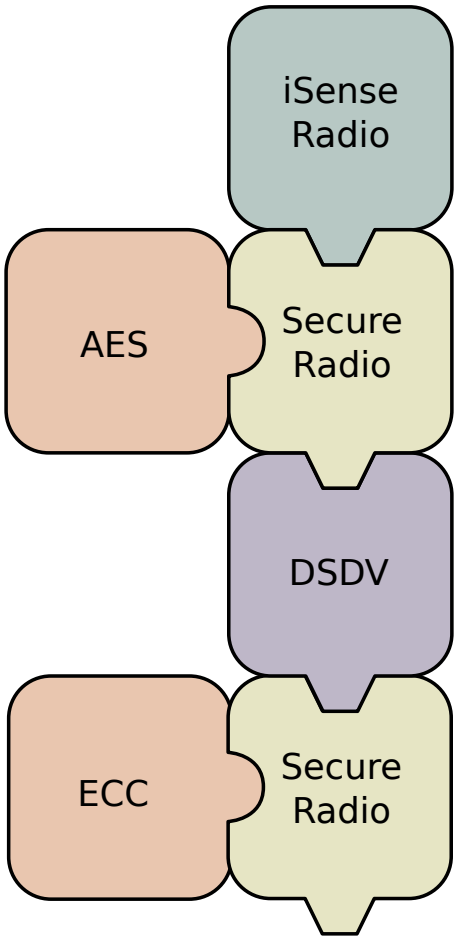
Motivation

- Very specialized and complex systems are hard to maintain
- ...or reuse
- Little changes in the application scenario can have drastic impact on the usability of a complex system

→ Make small components

- That fulfill a very specific task
- And do not rely on specifics regarding their surrounding
- Make these components combinable

Recall Of The Stackability Example



- Create arbitrary complex applications
- Just by plugging together algorithms

Here:

1. "Physical" radio by iSense
2. AES-Encrypted node-to-node radio
3. Routing, all packets AES-encrypted node-to-node
4. All packets AES-encrypted node-to-node,
   payload ECC encrypted end-to-end

How does this look in code?

- First, build the AES encrypted radio

```
1 typedef wiselib::SecureRadio<
2   Os, Os::Radio, wiselib::AES
3> aes_radio_t;
```

- Using that, we can build DSDV on top of AES encrypted radio

```
1 typedef wiselib::StaticArrayRoutingTable<
2   Os, aes_radio_t, 30,
3   wiselib::DsdvRoutingTableValue<Os, aes_radio>
4> routing_table_t;
5
6 typedef wiselib::DsdvRouting<
7   Os, routing_table_t, aes_radio_t
8> routing_t;
```

- ECC encrypted communication over DSDV Routing over AES encrypted iSense Radio

```
1 typedef wiselib::SecureRadio<
2   Os, routing_t, wiselib::ECC
3> ecc_t;
```

Initialization

- Declaration

```
1 Os::Radio::self_pointer_t radio_;
2 Os::Timer::self_pointer_t timer_;
3 Os::Debug::self_pointer_t debug_;
4
5 wiselib::AES aes_;
6 aes_radio_t aes_radio_;
7
8 routing_t routing_;
9
10 wiselib::ECC ecc_;
11 ecc_radio_t ecc_radio_;
```

- Initialization

```
1 radio_ = &wiselib::FacetProvider<Os, Os::Radio>::get_facet(amp);
2 timer_ = &wiselib::FacetProvider<Os, Os::Timer>::get_facet(amp);
3 debug_ = &wiselib::FacetProvider<Os, Os::Debug>::get_facet(amp);
4
5 aes_.init();
6 aes_.key_setup();
7 aes_radio_.init(*radio_, aes_);
8
9 routing_.init(aes_radio_, *timer_, *debug_);
10
11 ecc_.init();
12 ecc_.key_setup();
13 ecc_radio_.init(routing_, ecc_);
```