Carlo Pescio

An overly simple, C++ idiomatic pattern language for message-based product families

Time-proven

- Used many times over ~20y
- Used in small (PIC 32 w/ KB of RAM) embedded systems
- Used in “big” (i7 w/ GB of RAM) embedded systems
- N x 100K units around the world using this stuff ☺

good design??

- ~~Principles~~
- ~~Patterns~~
- ~~Dogma~~
  - Paradigm
  - Method
  - Technology

Forces -> Response

Product family

Product

a subset of a “feature set”

Defined extensionally : $P = \{ F_1, \ldots, F_p \}$

Most likely dynamic over time

- Products can be standard or custom

- | P | is “relevant”

Product family

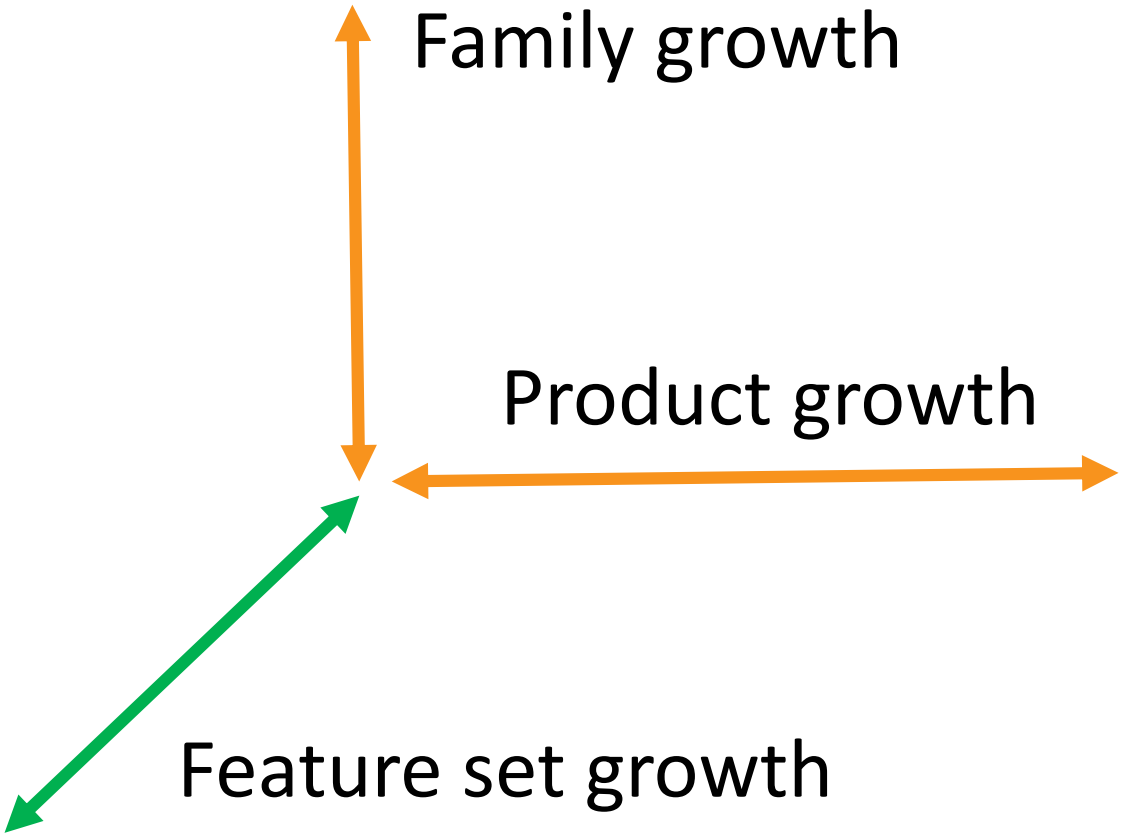
- a set of products
  - Defined extensionally : $F = \{ P_1, \ldots, P_f \}$
  - Dynamic over time
  - $| P_i \cap P_j | \approx \frac{\sum | P_i |}{f}$

- $| F |$ is “relevant”

Approaches

- Many, but mostly:
  - Full-blown + configuration
  - Selected features only

“expansive force”



“good response”

- New feature -> New code
- Grow / shrink product
  - include / remove file
- Add product to family
  - Choose file set

“bad response”

Custom code, “feature aware” code for each product

|F| “relevant” => lots of custom artifacts

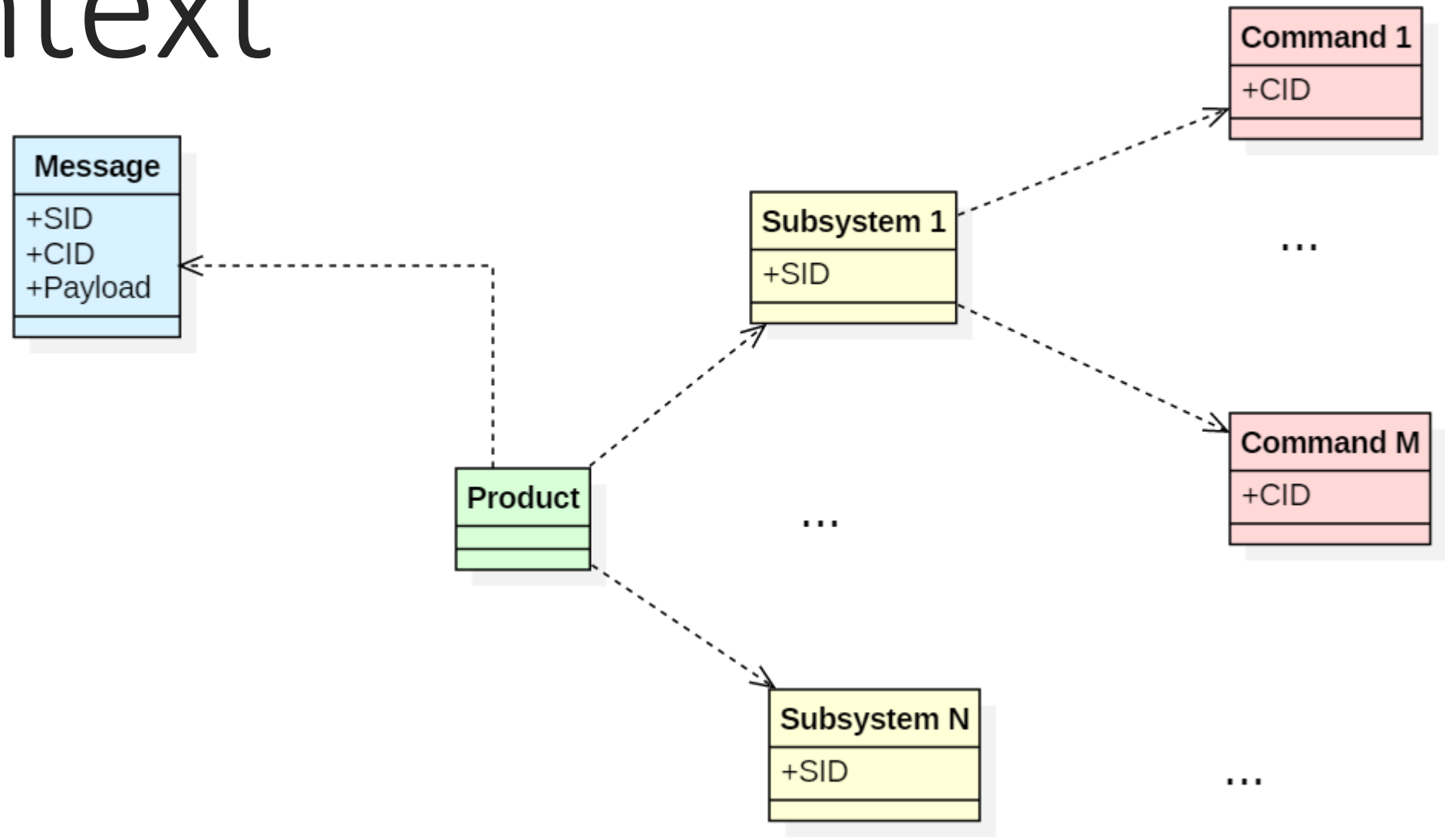
|P| “relevant” => each artifact is “relevant”

$$| P_i \cap P_j | \approx \frac{\sum | P_i |}{f}$$ => custom yet highly similar

|F| and |P| unstable and usually growing => maintenance cost

Common traps: switch/case, pattern matching, case classes, sum types, builders, … (anything “enumerative” in nature)

Context


Message
+SID
+CID
+Payload
Product
Subsystem 1
+SID
Subsystem N
+SID
Command 1
+CID
Command M
+CID
...
...
...

The ugliness

```
class MessageHandler // or Dispatcher
  {
  void handle( const Message& m )
    {
    switch( m.getSubsystem() )
      {
      case SS1 :
        {
        switch( m.getCommand() )
          {
          case CMD1:
            {
            // now what?
            }
          }
        }
      }
    }
  }
```

- Ugly naming
- Can’t share among products
- Maintenance black hole
- Same issue with pattern matching

Half-hearted

I know, I'll use a map of commands!

```
MessageHandler
   Message -> SID,CID,params
   CmdMap : (SID,CID) -> Command
   Command -> Execute( params )
```

Ok, **who** is filling the map?

- A "main"
- A "factory"

  still the same problem / shape

- A "configuration file"
  - not idiomatic (reflection?)
  - better for "full-blown"

- [dynamic loading not available everywhere, e.g. PIC 32]

A 1st idiomatic notion

Self-instantiating object

Avoid the common trap of a single artifact which knows many objects.

Add a class (artifact) -> Get an object (run-time)

The object is born before main is executed. Avoids threading issues as well.

Idiomatic: can't do this in Java or C#, they only got lazy statics.

Code: overly simple

```
I'm the magical self-creating object
I'm the main function
```

```
class SelfCreating
{
private:
  SelfCreating()
  {
  std::cout << "I'm the magical self-creating object" << std::endl;
  }
  static SelfCreating instance;
};
```

```
#include "SelfCreating.h"

SelfCreating SelfCreating::instance;
```

```
int main()
{
  cout << "I'm the main function" << endl;
  return 0;
}
```

The self-creating singleton

Useful outside this restricted application.

The C++ idiomatic solution to the real hard question:

who creates the creator? << it creates itself (seems just right too)

See:

prototype

abstract factory

factory method

etc.

Bring in some domain

A command is a self-instantiating, stateless singleton

- Self instantiating: no one needs to “know all the commands”
- Commands register themselves with a catalog

  reversing the dependency
- Being stateless is not strictly necessary, but it’s often the right choice

  commands are ok with being stateless anyway

A singleton (booooo!!!)

Stateless, immutable, single function -> **it's a function (ooohhhhh!!!!!)**

Added Value: **the singleton constructor will register the function in a catalog**

that's the magic functions can't do

But the catalog must be a singleton too, and **born before** (ain't that difficult?)

Commands & subsystem

Being realistic:
same command code in different susbsystems should be ok

Therefore: a command catalog for each subsystem

Therefore: the [concrete] command should know its own subsystem

it's OK, it's the opposite that we do not want (ss->cmd)

Code: simple enough

```
class Command
{
public:
  virtual ~Command()
  {
  }

  virtual void Process(
    const char* payload) = 0;
};
```

CRTP
(2nd Idiomatic notion)

```
template<class SS, class CMD> class
RegisteredCommand : public Command
{
protected:
  RegisteredCommand()
  {
    CommandCatalog<SS::SID>::
    RegisterCommand(CMD::CID, instance);
  }
private:
  // eager singleton - MUST be eager
  static CMD instance;
};

template<class SS, class CMD> CMD
RegisteredCommand<SS,CMD>::instance;
```

Code: simple enough

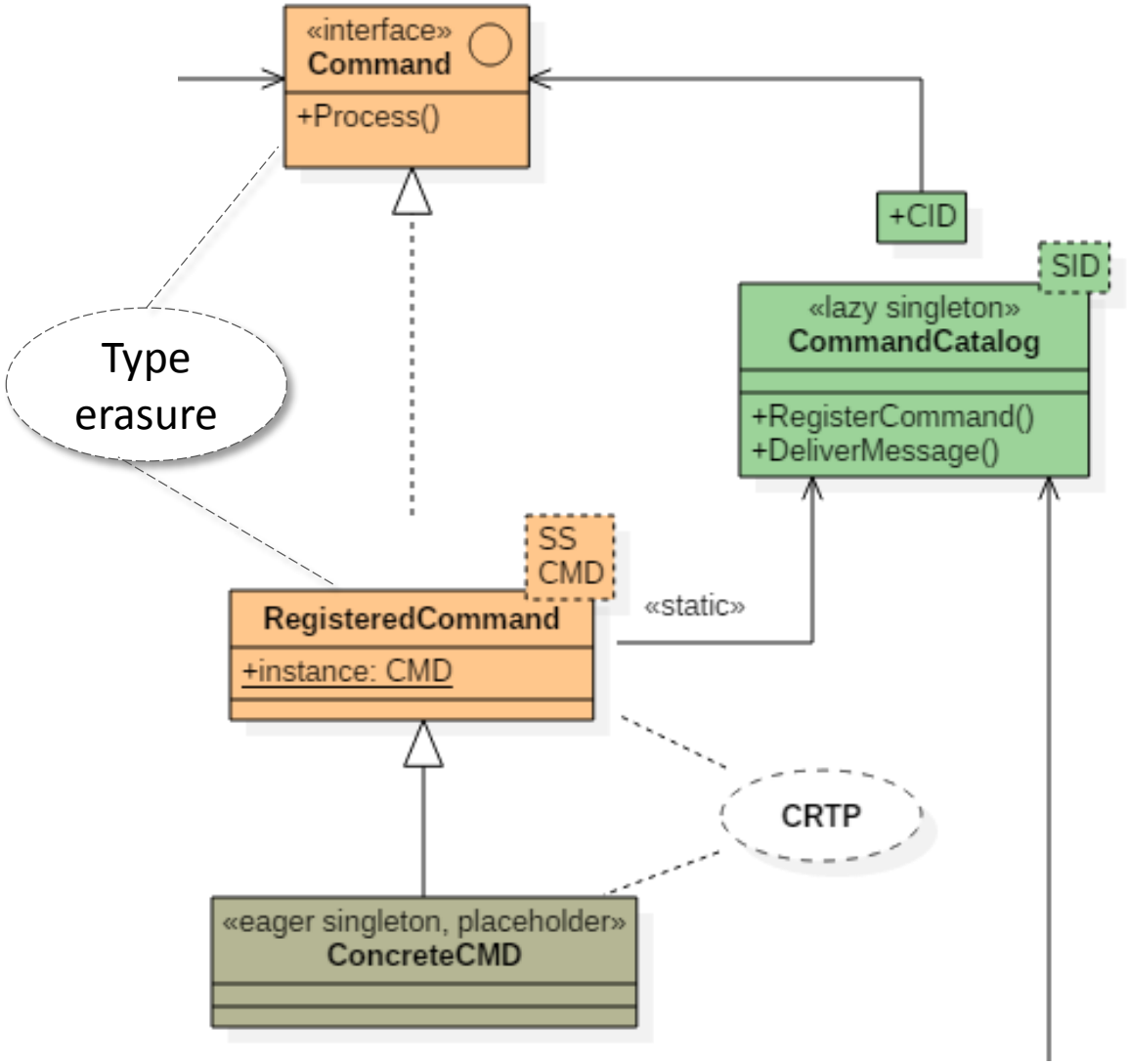
```
class StartEngineCMD : public
      RegisteredCommand< EngineSS, StartEngineCMD >
{
public:
  static const int CID = 2;

  void Process(const char* payload) override;

private:
  friend class RegisteredCommand< EngineSS, StartEngineCMD >;

  StartEngineCMD();
};
```

UML diagram (in 2017 ?! :-)



A 3rd idiomatic notion

A command *catalog* is a stateful, *semi-immutable* **lazy** singleton

- Lazy: simplest way to solve the initialization order problem

  commands are eagerly created before main is started,

  the first command creates the catalog as well

- The catalog is read-only / immutable when the main is started

  all mutations happen in a single thread before main is called

  thread-safe, read-only access after that (don't worry – live happy)

Code: again, overly simple

```
template< int SID > class CommandCatalog
{
public:
  static void RegisterCommand(int CID, Command& c)
  {
    auto& r = Recipients();
    assert(r.find(CID) == r.end());
    std::pair< int, Command& > p(CID, c);
    r.insert(p);
  }

private:
  static std::map< int, Command& >& Recipients()
  {
    // lazy singleton - MUST be lazy
    static std::map< int, Command& > recipients;
    return recipients;
  }
};
```

```
// later
// static bool DeliverMessage(
//    const Message& m)
```

The haiku

The **lazy** singleton

is **fully** constructed

before the **eagerest** singleton

Mocking commands?

Assuming it makes sense – depends on the application
    sometimes mocking hardware resources is a better option

If you want a “test build” vs. a “standard build”
    trivial, just define mock and std commands, use the makefile

If you want to choose at runtime
    create both, register both, enable one [with a command]
    one more template parameter will help ☺

Rinse and repeat

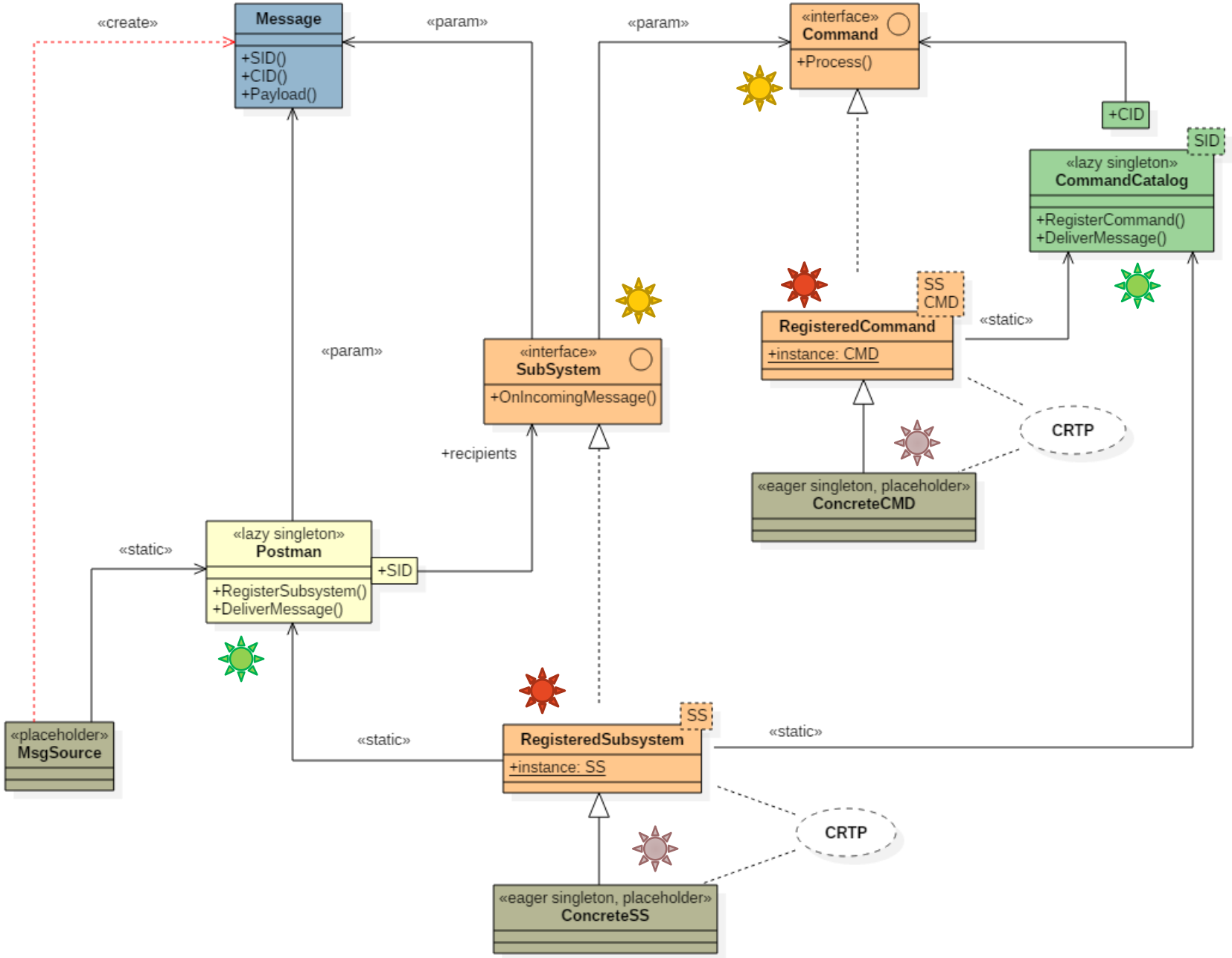
- Subsystems are self-creating singleton[s] too

- Subsystems register themselves with a postman / catalog
  no one needs to “know all the subsystems”

- The postman is a lazy singleton as well

- See the symmetry: Command<->CommandCatalog, Subsystem<->Postman
  sort of, there is 1 catalog per subsystem, but ok

there :)


«create»
Message
+SID()
+CID()
+Payload()
«param»
«param»
«interface»
Command
+Process()
+CID
SID
«lazy singleton»
CommandCatalog
+RegisterCommand()
+DeliverMessage()
SS
CMD
RegisteredCommand
+instance: CMD
«static»
«param»
«interface»
SubSystem
+OnIncomingMessage()
CRTP
+recipients
«eager singleton, placeholder»
ConcreteCMD
«static»
«lazy singleton»
Postman
+SID
+RegisterSubsystem()
+DeliverMessage()
«placeholder»
MsgSource
«static»
SS
RegisteredSubsystem
+instance: SS
«static»
CRTP
«eager singleton, placeholder»
ConcreteSS

Subsystems responsibilities

- Defines its own address (SID, this is in the sample too)
- Owns exclusive HW resources (e.g. a RS485, some I/O, etc.)
- Usually offers some logic to its own commands
- Is usually **stateful / mutable**: mirrors the state of hw devices

A mutable singleton!

- In the sample code, it's all synchronous, but in a real system, **a SS will probably have its own message queue and a thread**. N SS could share 1 thread if reasonable.

- **Commands belonging to a SS will be executed in the SS thread**.
  So even if they call back into the SS, it's still single-threaded.
  We never need to synchronize (except [maybe] the command queue).

- CMD1 in SS1 can send messages to SS2, using the Postman
  So it's also an internal communication channel.

- So, a subsystem **is an actor** (oooohhh).

- What about mocking?? (same as commands…)

A little more code

```
bool Postman::Deli
{
  auto r = Recipie
  auto i = r.find(

  if (i != r.end()
    return i->seco

return false;
}
```

```
Delivering StartEngineCMD
StartEngineCMD::Process
1

Delivering DisableKeyboardCMD
DisableKeyboardCMD::Process
1

Delivering EnableKeyboardCMD
EnableKeyboardCMD::Process
1

Delivering invalid command (sid ok cid ko)
0

Delivering invalid command (sid ko [cid ko])
0
```

```
sage(const Message& m)
);
D());
m.Payload());
```

Variations

Protocol: encoding, ack/nak, synchronous vs asynchronous answer, etc.

Threading: synchronous, 1 thread per SS, 1 thread * N SS, N thread * 1 SS

Queuing: selective coalescence or not.

Execution: 1 active command, N active commands, background activity

Substitution: Mocking commands, Activable commands, etc.

“Details”: splitting payload parsing and execution (stateful commands, cloning).

Why “a pattern language”

“Many patterns form a language”

File Name	Lines	Statements	Average Statements per Method
Command.h	12	3	0.0
CommandCatalog.h	41	20	4.0
Message.h	31	10	0.8
Postman.cpp	33	17	3.7
Postman.h	15	7	0.0
RegisteredCommand.h	18	7	1.0
RegisteredSubsystem.h	24	10	1.0
Subsystem.h	14	4	0.0

Trying to generalize will turn this short code into the usual abstract monster

The abstract monster

- Policies everywhere
- Parameters everywhere
- N times longer to account for any possible variability
- Names far removed from domain terminology
- Steep learning curve
- Cross maintenance

VS.

understand the Pattern Language ad adapt on your system

PhysicsOfSoftware.com

Notes on Software Design, Chapter 12: Entanglement

http://www.carlopescio.com/2010/11/notes-on-software-design-chapter-12.html

Notes on Software Design, Chapter 13: On Change

http://www.carlopescio.com/2011/02/notes-on-software-design-chapter-14.html

Notes on Software Design, Chapter 14: the Enumeration Law

http://www.carlopescio.com/2011/02/notes-on-software-design-chapter-14.html

On Growth and Software (nCrafts, Paris, May 2016)

Video: http://videos.ncrafts.io/video/167699028

Thanks to the sponsors!

Brain-Driven Design

Design your software by squeezing your brain, not by following trends

carlo.pescio@gmail.com