AARHUS
UNIVERSITY
DEPARTMENT OF ENGINEERING

Architecture and Design of Embedded Real-Time Systems (TI-AREM)

GoF: Command Pattern
a Behavioral Pattern

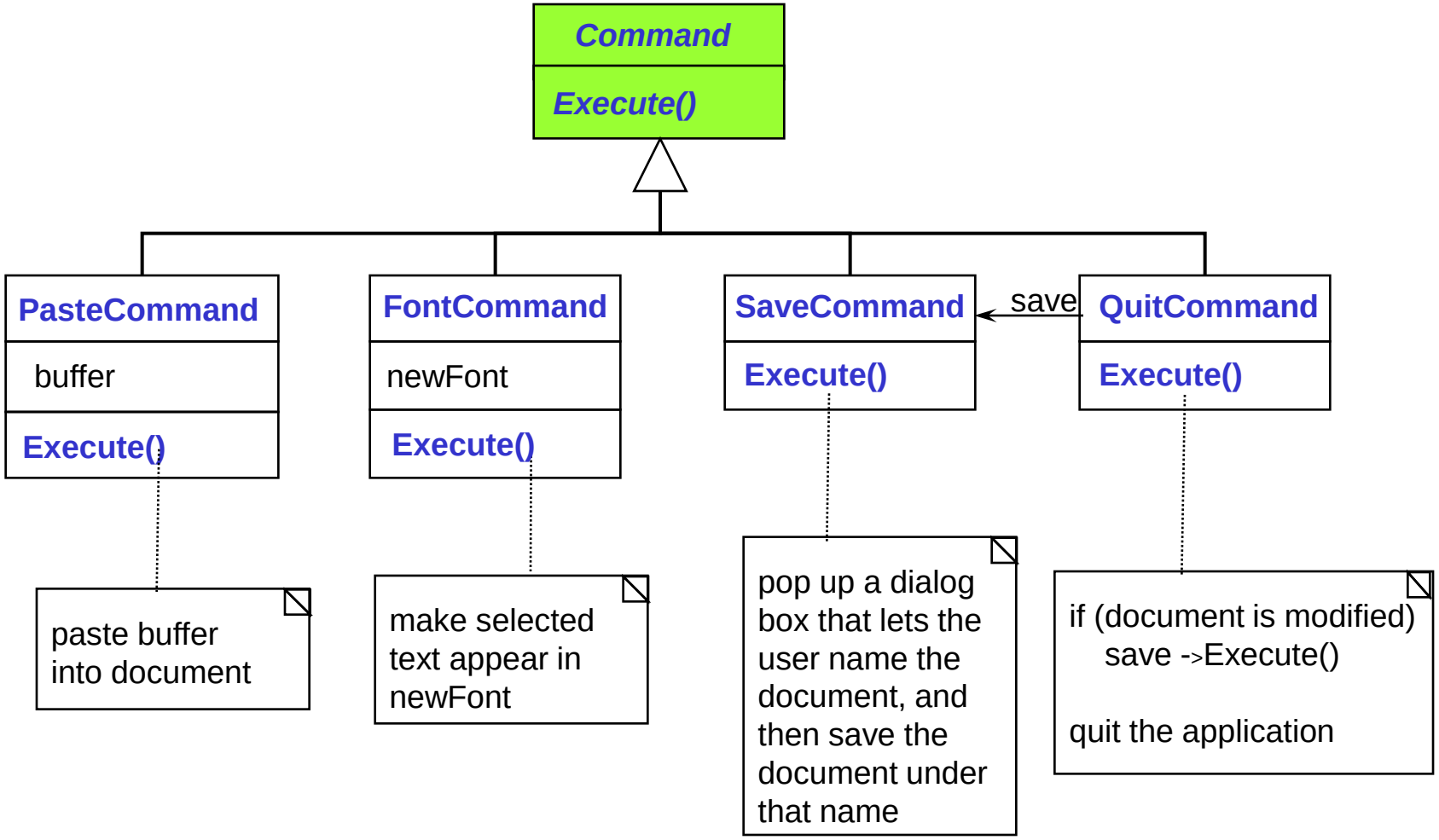
Version: 31-1-2015

Command Pattern (GoF) - Intent

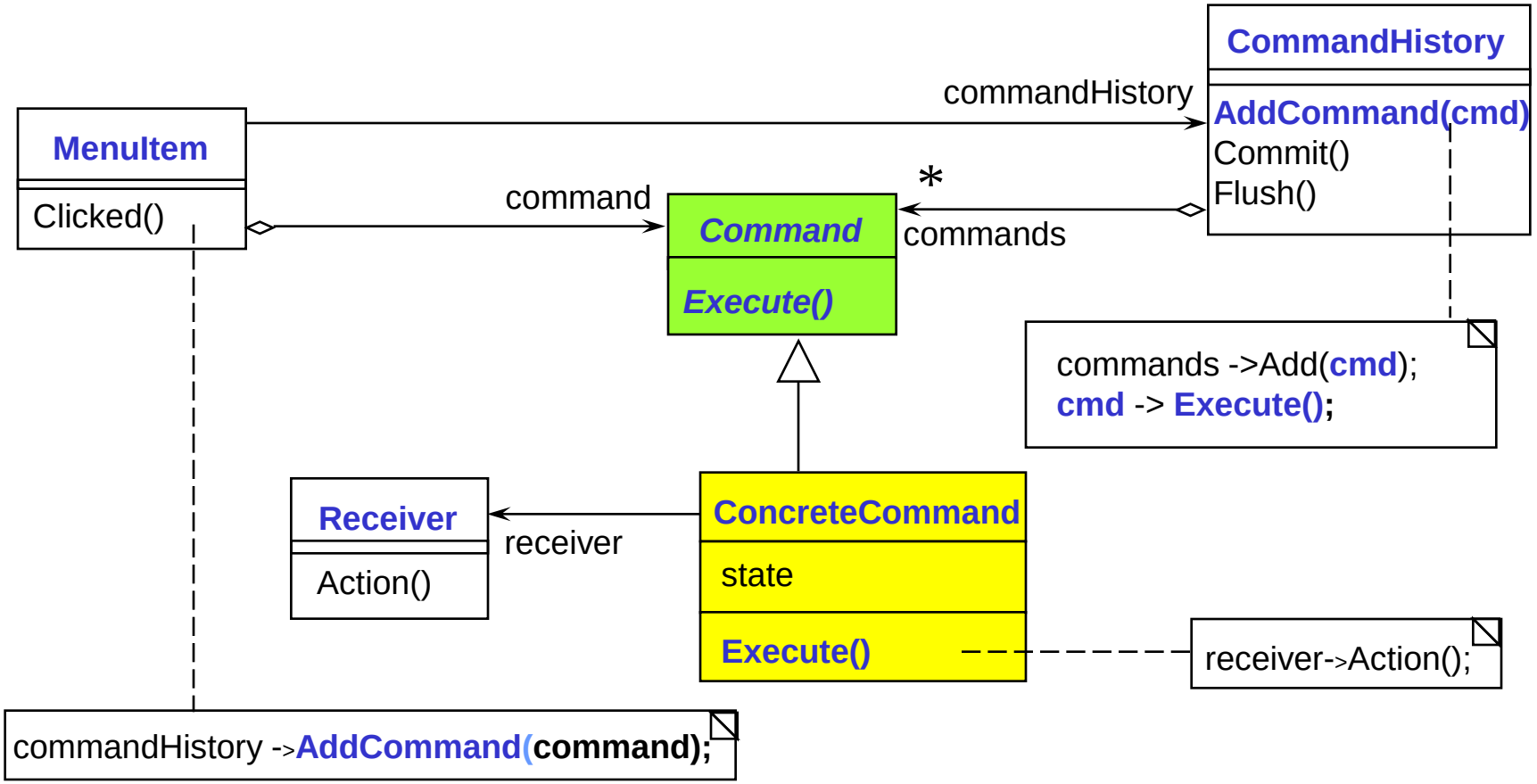
Encapsulate a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations

Ref. Gamma (GoF book p.233)

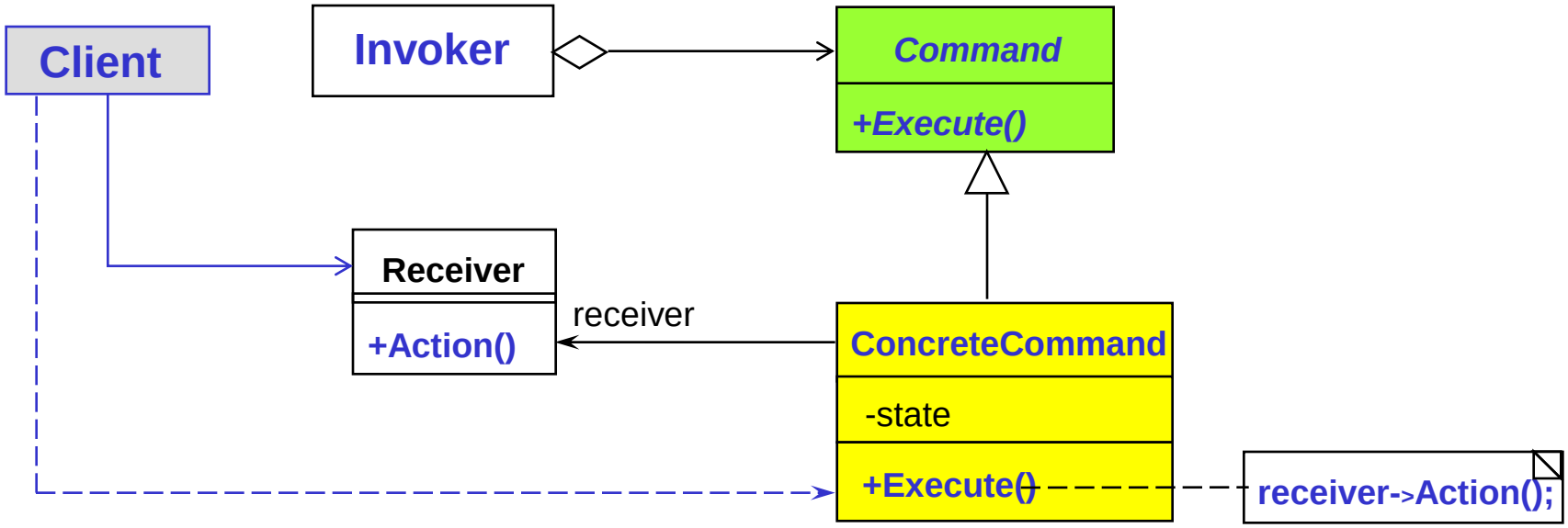
Command Pattern Example (1)



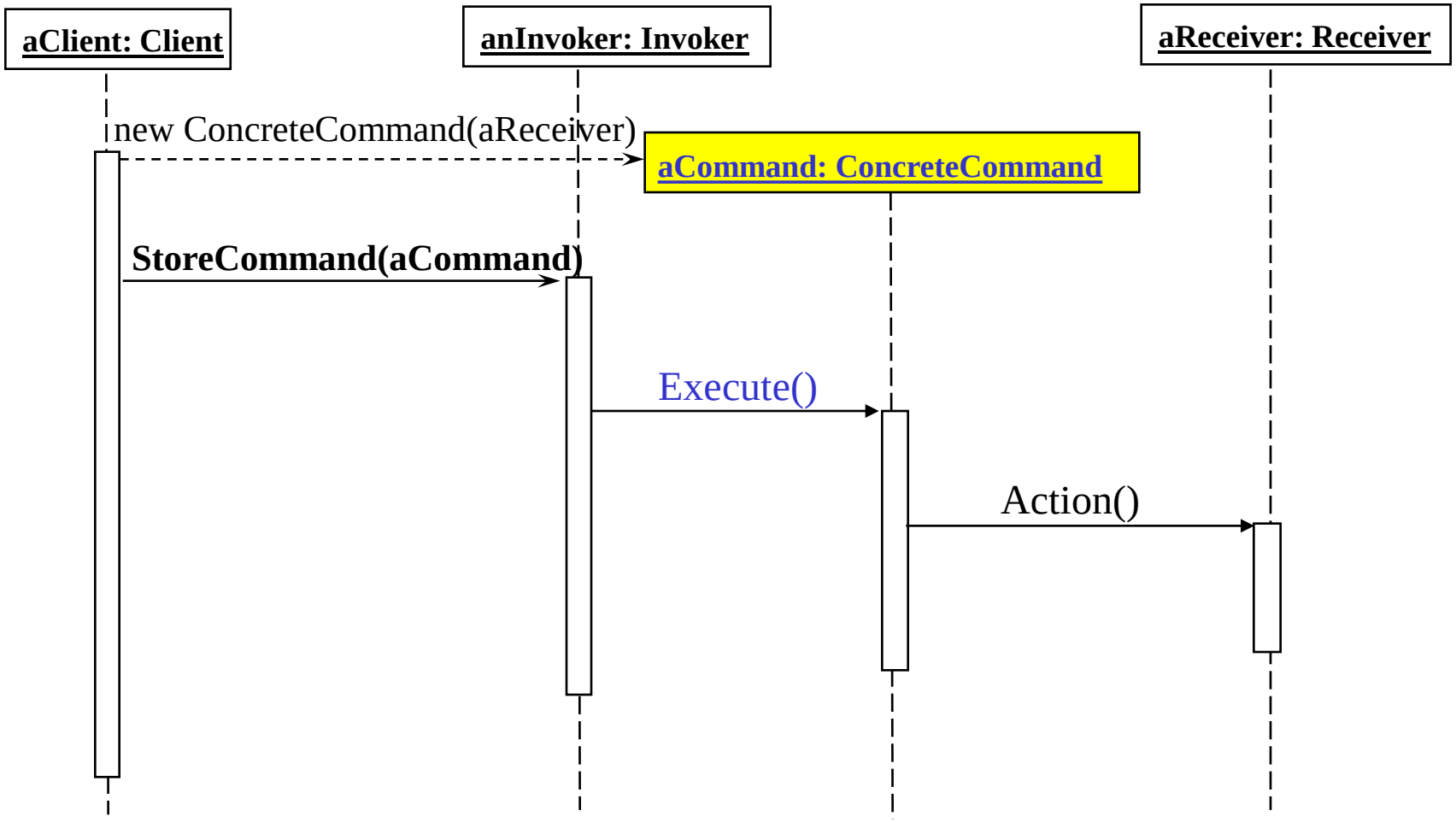
Command Pattern Example (2)



Command Pattern - GoF Structure



Command Pattern Sequence Diagram



Command Pattern Consequences

- Decouples the object that invokes the operation form the one that knows how to perform it
- Commands are first-class objects.
  - They can be manipulated and extended like any other object
- Commands can be assembled into a composite command
- It’s easy to add new commands
  - you don’t have to change existing classes

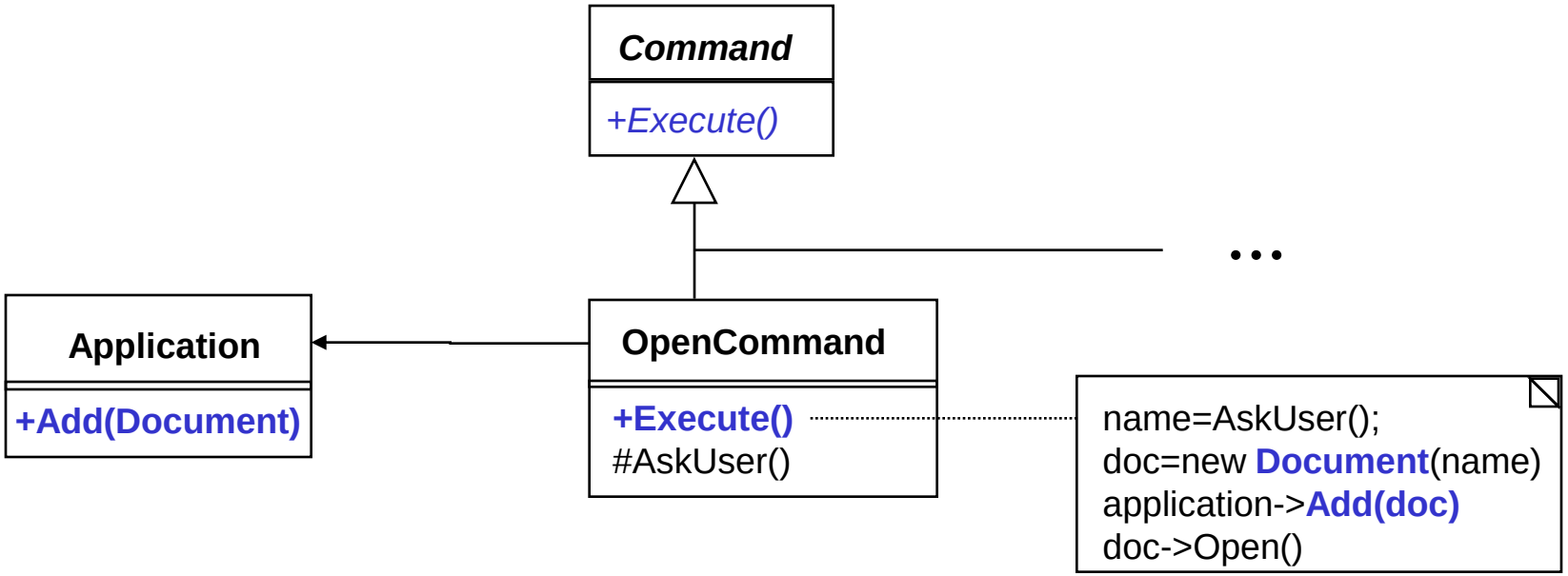
Implementation (1)

- How intelligent should a command be?
  - at one extreme it only defines a binding between a receiver and the actions that carry out the request
  - at the other extreme it implements everything itself without delegating to a receiver at all

Implementation (2)

- Using C++ templates
  - for commands that (1) aren't undoable and (2) don't require arguments
  - avoids creating a command subclass for every kind of action and receiver

Command Pattern - Example



Command C++ Example (1)

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

    virtual void Execute() = 0;
protected:
    Command();
};
```

```
class OpenCommand : public Command
{
public:
    OpenCommand(Application*);
    virtual void Execute();
protected:
    virtual const char* AskUser();
private:
    Application* _application;
    char* _response;
};
```

Command C++ Example (2)

```
OpenCommand::OpenCommand (Application* a) { _application = a; }

void OpenCommand::Execute()
{
    const char* name = AskUser();

    if (name != 0) {
        Document* document = new Document(name);
        _application->Add(document);
        document->Open();
    }
}
```

C++ Template Class Example (1)

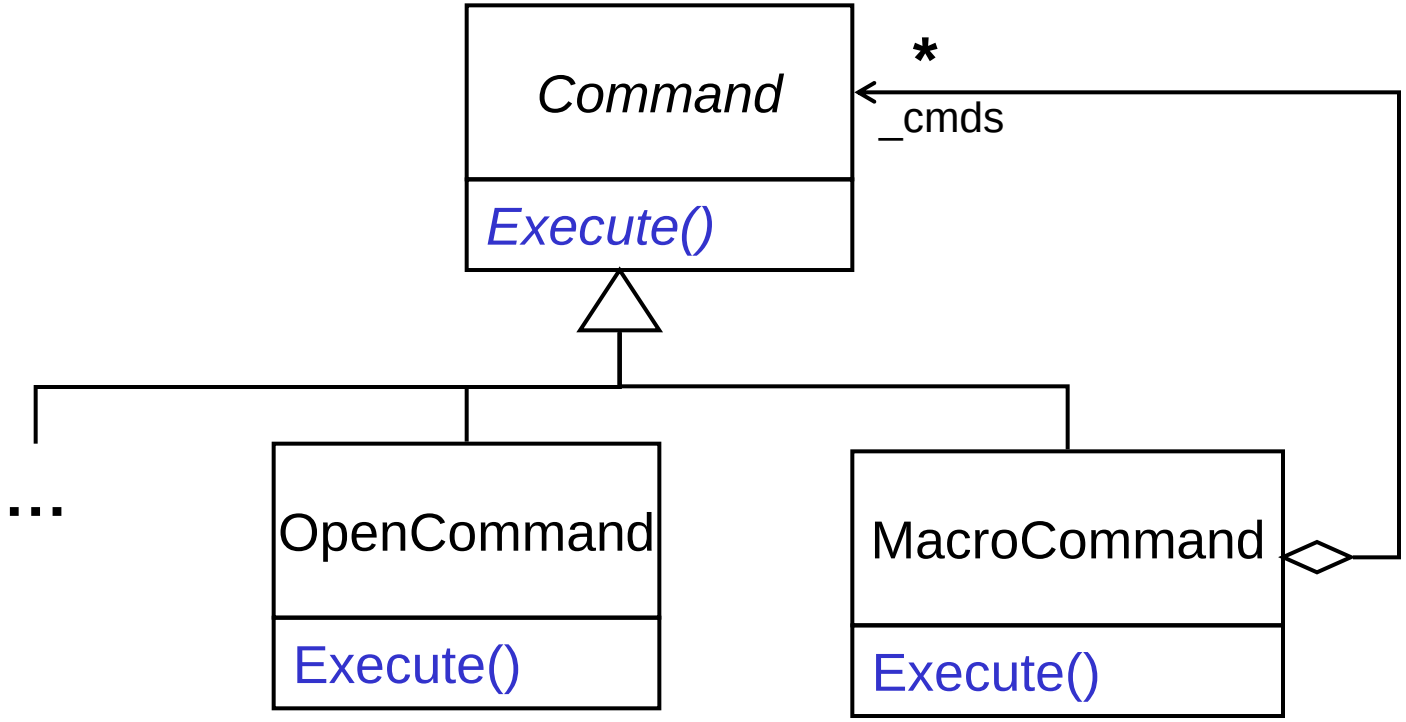
```
template <class Receiver>
class SimpleCommand : public Command
{
public:
   typedef void (Receiver::* Action) ();  // defines a function pointer

   SimpleCommand(Receiver* r, Action a) :
         _receiver(r), _action(a) { }

   virtual void Execute() { (_receiver->*_action) (); }
protected:
   virtual const char* AskUser();
private:
   Action _action;
   Receiver* _receiver;
};
```

C++ Template Class Example (2)

```
class MyReceiverClass
{
   public:
   void myAction1();
   void myAction2();
};
```

```
main()
{
   //
   MyReceiverClass* pReceiver = new MyReceiverClass;
   //
   Command* pCommand=
         new SimpleCommand<MyReceiverClass>
               (pReceiver,&MyReceiverClass::myAction1);
   //
   pCommand->Execute();
}
```

Macro Command



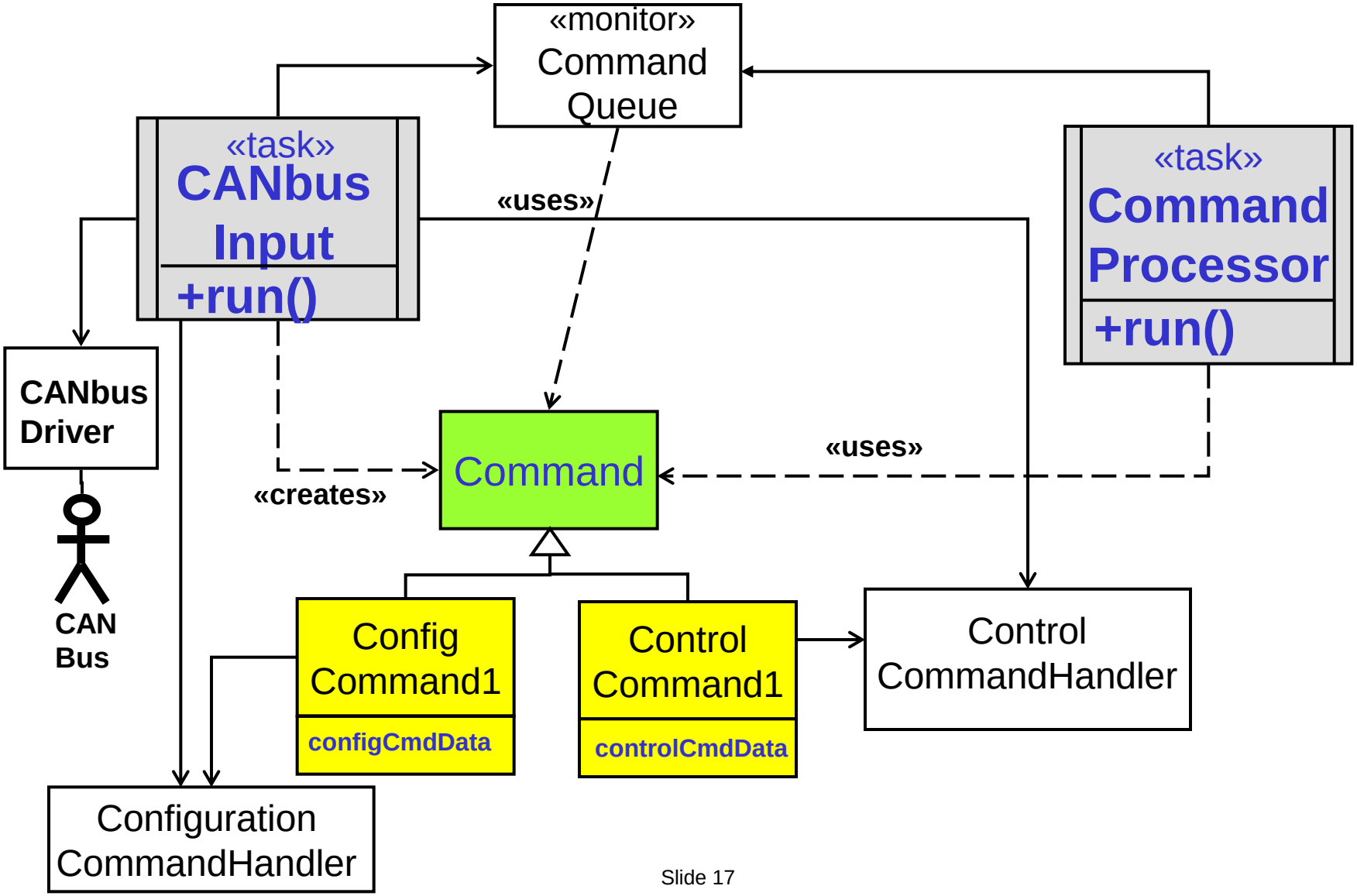
Command C++ Example (3)

```
class MacroCommand : public Command
{
public:
   MacroCommand();
   virtual ~MacroCommand();

   virtual void Add(Command*);
   virtual void Remove(Command*);
   virtual void Execute();
private:
   List<Command*>* _cmds;
};
```

```
void MacroCommand::Execute()
{
   ListIterator<Command*> i(_cmds);

   for (i.First(); !i.IsDone(); i.Next())
   {
      Command* c = i.CurrentItem();
      c->Execute();
   }
}
```

Iterator Pattern (GoF page 257)

Embedded System Example



Class Exercise

1. Identify the client, the invoker and the receiver roles on slide 17?
2. Add operations to the class diagram at slide 17.
3. Write “C++ pseudo code” for the “run() operation” in the CANbusInput task
4. Write “C++ pseudo code” for the “run() operation” in the CommandProcessor task.
5. Discuss the design – pros et cons.

Summary

Command Pattern
– Very useful