# Imperative Programming 3

## Design Patterns

Peter Jeavons

Trinity Term 2019

# Average vs. Best Programmer

- Initial coding time
  - studies from 1968 to 2000 showed 20x ratio
- Debugging time
  - 20x ratio
- Program execution speed
  - 10x ratio
- 80% contributions from 20% of programmers
  - At the end of this course, you will be in the top-20%

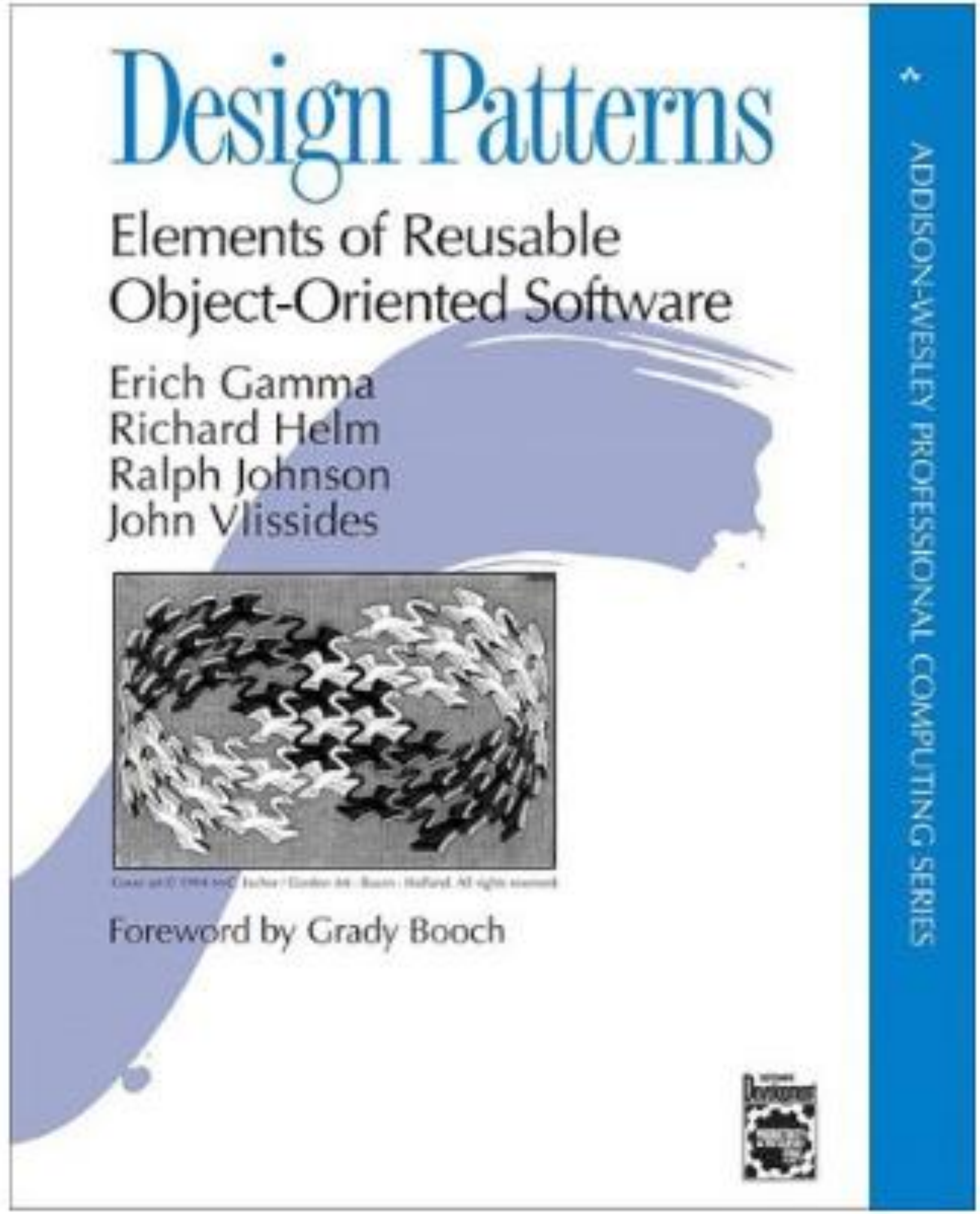
# Challenges in Software Engineering

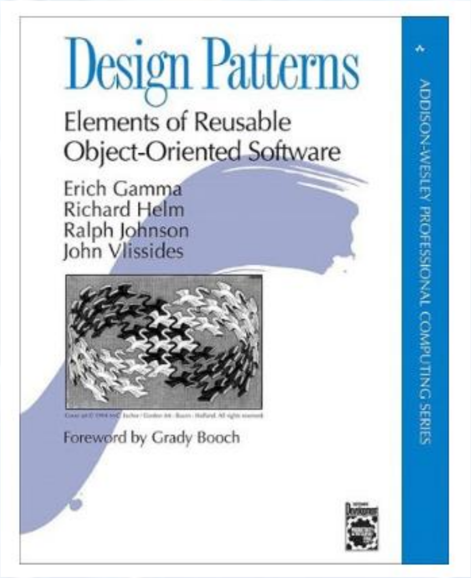
- Designing good object-oriented software
  - suitable classes? interfaces? inheritance? relationship?
  - experienced designers can get it right
  - novices spend lots of time and make mistakes
- Experience = toolbox of reusable solutions
  - classify problems and apply solution templates

# Design Patterns

Each pattern describes a problem which occurs over and over again in our environment, and then describes the core solution to that problem, in such a way that you can use the solution a million times over, without ever doing it the same way twice.

Christopher Alexander (1977)






***Creational Patterns***

Abstract Factory
Builder
Factory Method
Factory Object
Lazy Initialization
Prototype
Singleton

***Structural Patterns***

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy

***Behavioral Patterns***

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

***Architectural***

Model-View-Controller
Service-oriented Architecture

***Concurrency Patterns:*** Active Object
Monitor
Thread Pool

# Basic Design Patterns

- Encapsulation
- Inheritance
- Exceptions

# Encapsulation

- Problem = exposure can lead to…
  - violation of representation invariant
  - dependencies that hamper implementation changes
- Solution = hide components
- Consequences
  - interface may not provide all desired operations
  - indirection may reduce performance

# Inheritance

- Problem = similar abstractions…
  - have similar fields and methods
  - repeating them => tedious, error-prone, unmaintainable
- Solution = inherit default members
  - correct implementation selected via runtime dispatching
- Consequences
  - code for a subclass not contained all in one place
  - fragile base class problem
  - runtime dispatching introduces performance overhead

# Exceptions

- Problem = errors occur in one place…
  - but should be handled in another part of the code
  - shouldn’t clutter code with error recovery
  - shouldn’t mix return values with error codes
- Solution = specialised language structure
  - throw exception in one place, catch & handle in another

# Exceptions in Scala

- Exceptions in Scala are, naturally, objects
- To *raise* an exception: throw an object

```
def subSequence(start: Int, end: Int):
     CharSequence = {
 if (start < 0 || end < start || len < end)
    throw new IndexOutOfBoundsException()

 getString(start, end-start)
}
```

- Execution then continues at the nearest exception handler… (if none, program crashes)

# Exceptions in Scala

- Exception handling = enclose the code throwing an exception in a try/catch statement

```
try {
  // code that might throw an exception
}
catch {
  case ex: ExceptionType1 => {
    println(”Problem: ” + ex)
  }
  case ex: ExceptionType2 => {...}
}
```

# Exceptions in Scala

- `finally` blocks are always executed (cleanup)
- User-defined exceptions

```
case class UnderAgeException(message: String) extends
                                        Exception(message)

def buyCigarettes(customer: Customer): Cigarettes =
  if (customer.age < 16) throw
      UnderAgeException("Customer must be older than 16")
  else new Cigarettes
```

- Runtime and static checks: assert, require, assume

# Exceptions

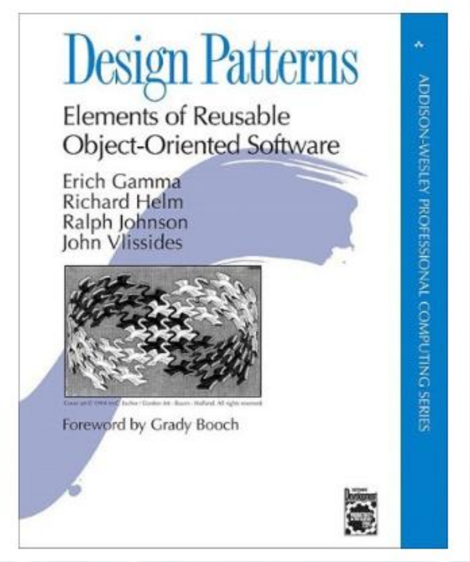
- Problem = errors occur in one place…
  - but should be handled in another part of the code
  - shouldn’t clutter code with error recovery
  - shouldn’t mix return values with error codes
- Solution = specialized language structure
  - throw exception in one place, catch & handle in another
- Consequences
  - hard to know layer at which exception will be handled
  - evil temptation to use this for normal control flow

### *Creational Patterns*

Abstract Factory
Builder
Factory Method
Factory Object
Lazy Initialization
Prototype
Singleton

### *Structural Patterns*

Adaptor
Bridge
Composite
Decorator
Façade
Flyweight
Proxy




### *Behavioral Patterns*

Chain of Responsibility
Command
Interpreter
Iterator
Mediator
Memento
Observer
State
Strategy
Template Method
Visitor

### *Architectural*

Model-View-Controller
Service-oriented Architecture

***Concurrency Patterns:*** Active Object
Monitor
Thread Pool

# Iterator pattern

- Problem = accessing all items in some collection
  - requires specialized traversal
  - exposes underlying details of how collection is stored
- Solution = implementation does traversal
  - results returned via standard interface
- Consequences
  - iteration order constrained by implementation

# The Iterator interface

```
trait Iterator[T] {
  def hasNext(): Bool
  def next(): T
}
```

Interface independent of what's being iterated over

- External iterators
  - client controls iteration by calling *hasNext()*, *next()*
  - default in most imperative languages like Java, C++

# The Iterable interface

```
trait Iterable[T] {
  def iterator(): Iterator[T]
}
```

# Internal iterators

- Accept a method to execute on all elements of a collection

```
someList.foreach(x => print(x))
```

- Mostly in languages with anonymous (lambda) functions and closures, like Scala, Ruby, ML, etc.

```
def foreach[U](f: Elem => U): Unit = {
  val it = iterator
  while (it.hasNext) f(it.next())
}
```

# Case Study: Text Editor

# Introducing the Case Study

- To really appreciate the power of OOP ideas for developing larger programs we need to look at a (fairly) large program;
- In this part of the course, we will examine a program to implement a simple text-editor, originally written by Mike Spivey
- This program contains around 2000 LoC, and illustrates many design principles…

# How do you start?

- When faced with the problem of designing a fairly large program, remember the slogans:

**“Separation of concerns”**

**“Stability under change”**

# How do you start?

- There is no ‘royal road’ to the ideal design, but it can be helpful to:

  - List the *concerns* of the program;

  - List the likely *changes*;

# So what are the concerns?

- A ***text editor*** is likely to be concerned with the following things:
  - Manipulating some text and keeping a current position within it;
  - Getting keyboard input from the user, and interpreting it as commands or new pieces of text;
  - Obeying these commands by moving the current position around, changing the text, and maybe saving it and loading it to/from files;
  - Displaying the text on the screen so the user can see the effect of the commands and the current state of the text;
  - Storing a list of the commands so they can be undone later

# So what are the concerns?

- A ***text editor*** is likely to be concerned with the following
  - Manip ent position within
  - Getting keyboard input from the user, and interpreting it as commands or new pieces of text;
  - Obeying these commands by moving the current position around, changing the text, and maybe saving it and loading it to/from files;
  - Displaying the text on the screen so the user can see the effect of the commands and the current state of the text;
  - Storing a list of the commands so they can be undone later

**Question:** What are the nouns in this description?

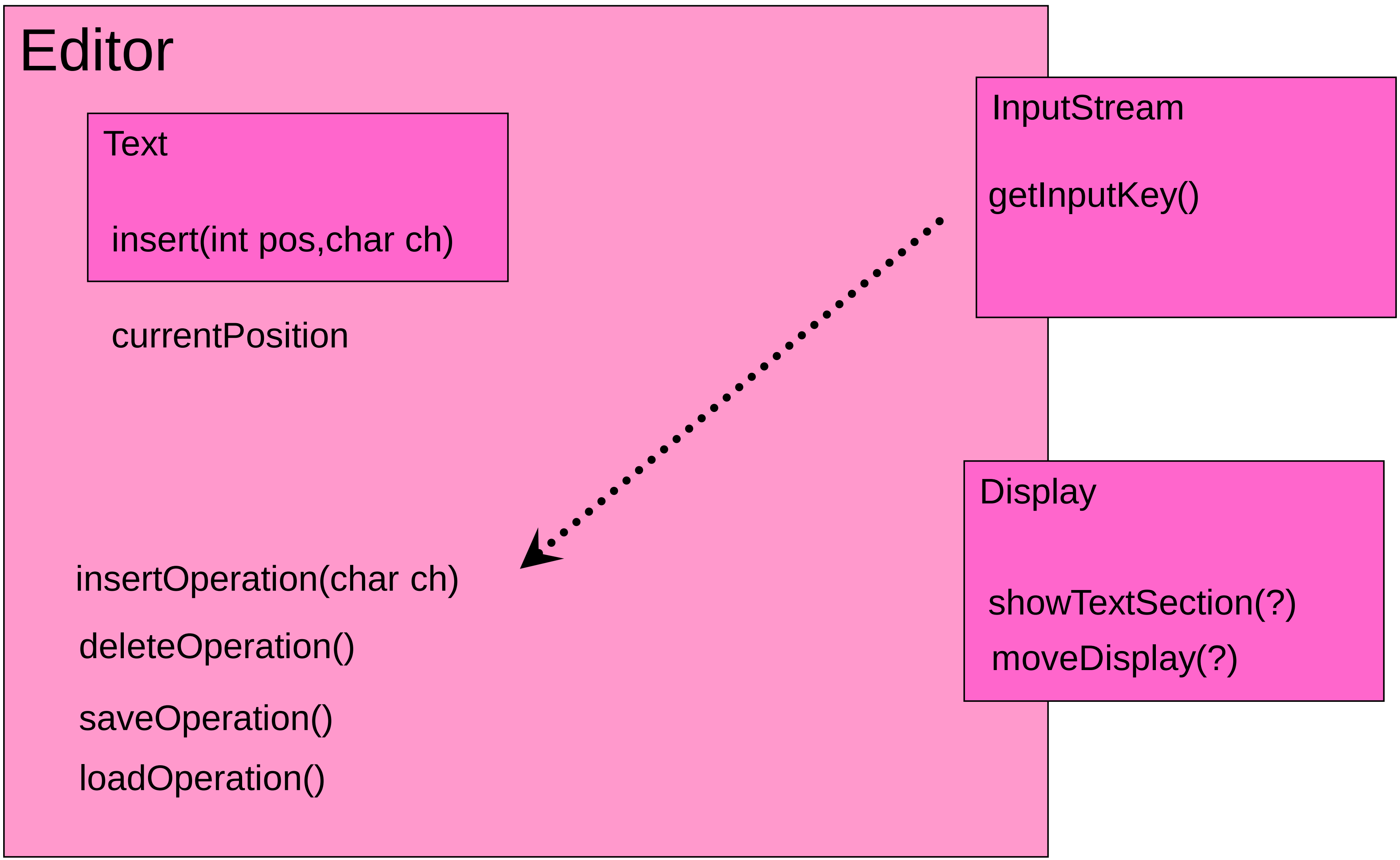
# Nouns

# So what are the likely changes?

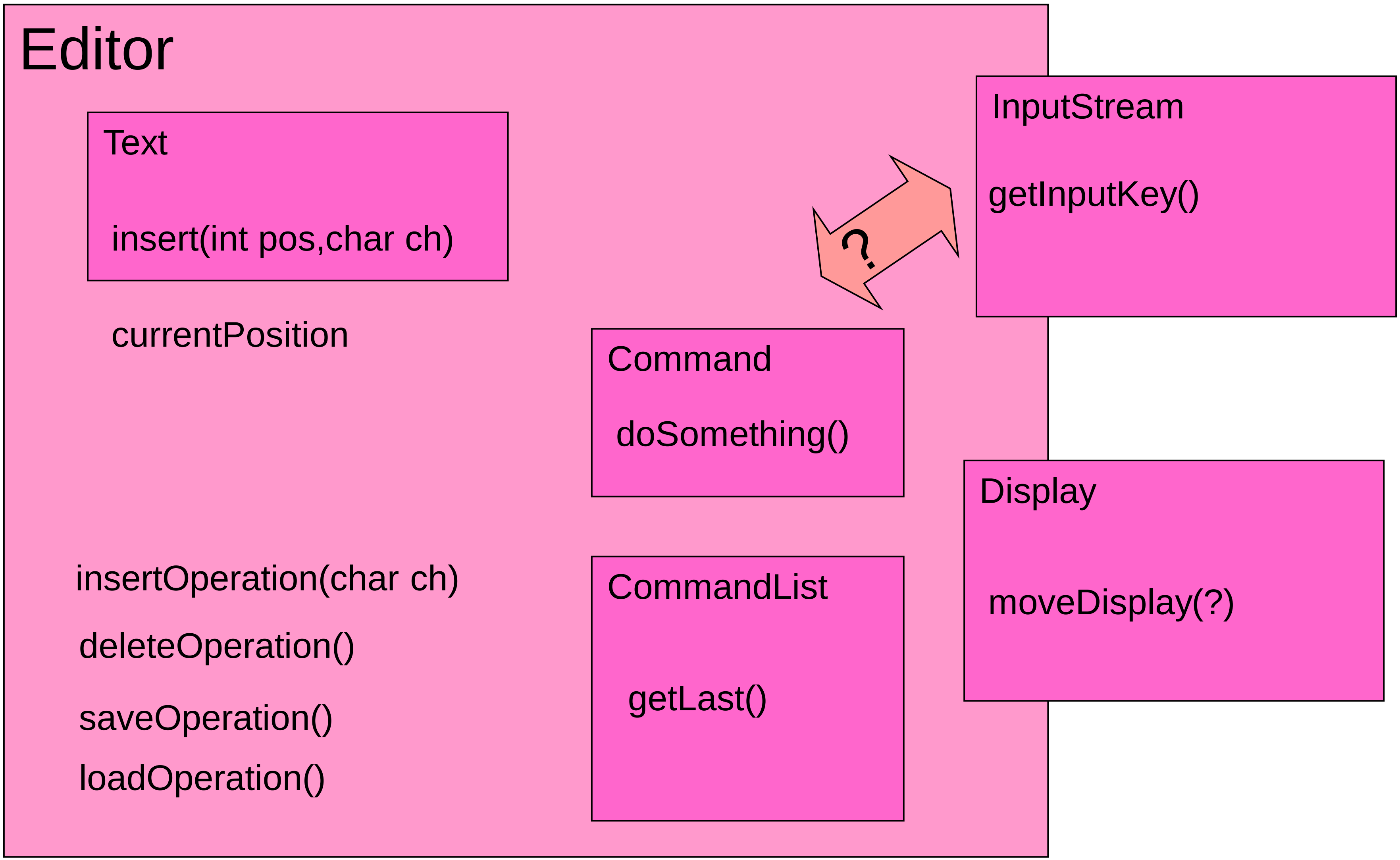
- A ***text editor*** is likely to experience the following changes during its lifetime:
  - Changes to the way text is stored – perhaps to allow larger texts, or faster access, maybe allowing more than one text;
  - Changes to the input source;
  - Changes to the way the input is interpreted – maybe giving new names/keys to commands;
  - Changes to the commands that are available;
  - Changes to the way text is displayed, perhaps to show larger or smaller portions, or move around differently when the editing position is changed;
  - Changes to the kind of screen or display device that the text is displayed on – maybe allowing more than one;

# Nouns *+ Changes*

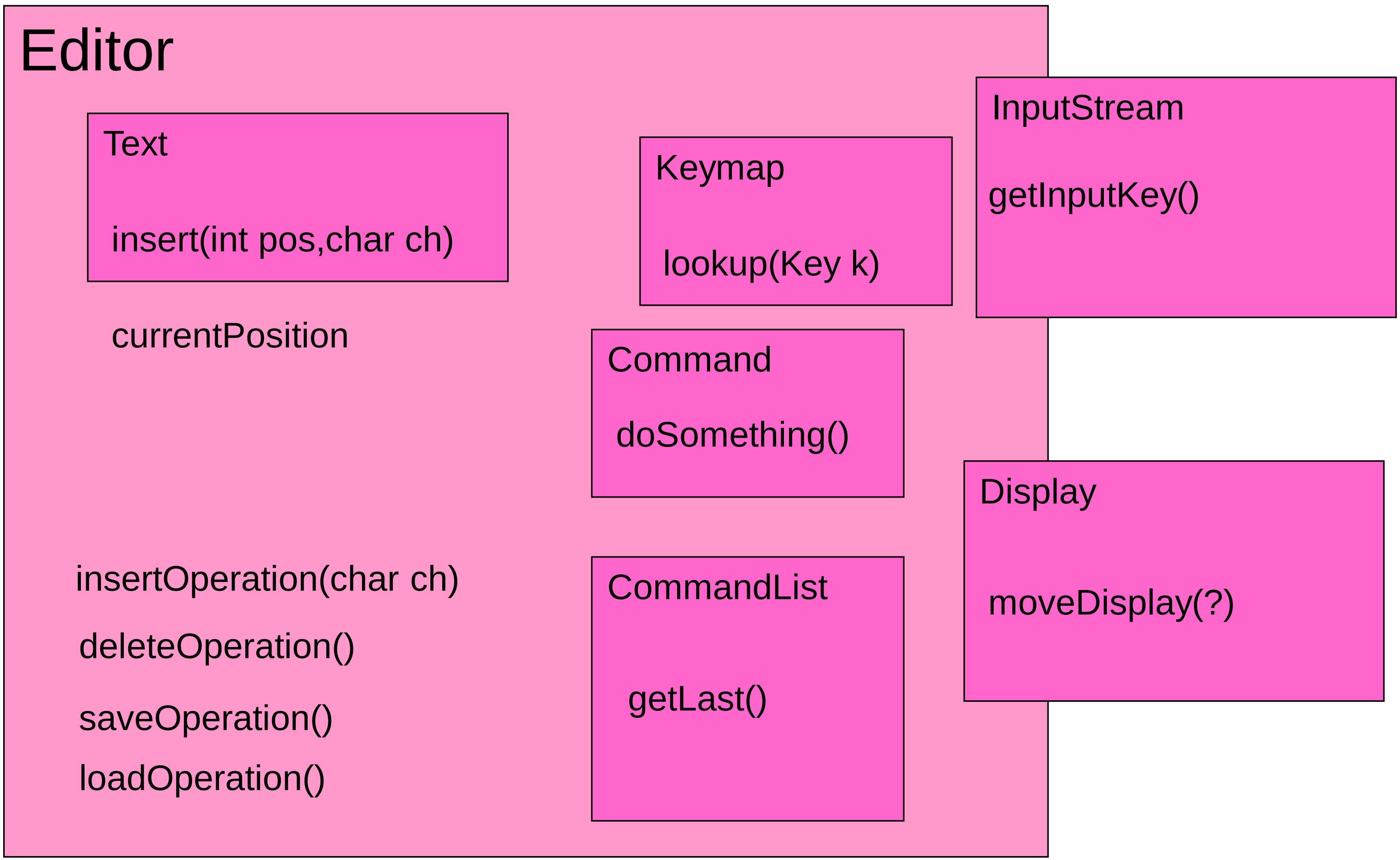
# Designing the Classes



# Designing the Classes



# Designing the Classes



# Summary

- Design patterns
  - can capture the experience of the best programmers
- Basic patterns:
  - encapsulation, inheritance, exceptions
- The Iterator pattern

- Designing an Editor…

See also *Head First Design Patterns*: Chapters 1 & 9