Five Principles (revisited)

1. Single-Responsibility Principle
2. Open–Closed Principle
3. Liskov Substitution Principle
4. Depency-Inversion Principle
5. Interface-Segregation Principle

Design Patterns: Background

- Architectural design patterns
  - Christopher Alexander *et al.*: *A Pattern Language*, 1977
  - Christopher Alexander: *The Timeless Way of Building*, 1979
- World consists of repeating instances of various patterns
  - a pattern is (possibly hidden) design know-how that should be made explicit
  - 'a quality without name': not measurable but recognizable
- User-centred design
  - capture the quality in a pattern language
  - inhabitants should design their own buildings together with a professional using the patterns

Alexander's Patterns

- What do high-quality contructs have in common?
- Structures cannot be separated from the problems they are solving
- Similarities in the solution structures → a pattern
- Each pattern defines subproblems solved by other smaller patterns
- A pattern is a rule that expresses a relation between
  - a context
  - a problem and
  - a solution

Alexander's Patterns (cont'd)

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

– C. Alexander, *The Timeless Way of Building*, 1979

Software Design Patterns

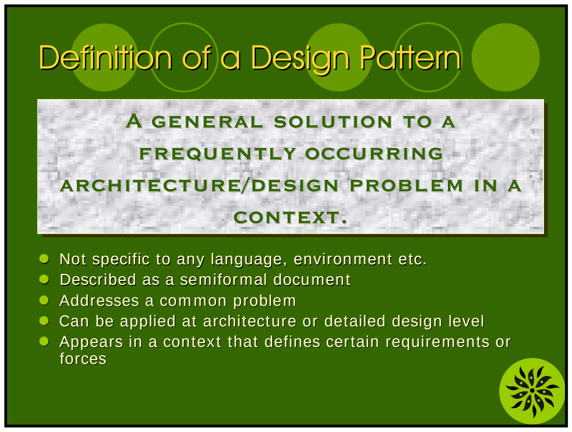
'[Patterns] are descriptions of communicating objects and classes that are customized to solve a general design problem in a particular context.'

'A design pattern names, abstracts, and identifies the key aspects of a common design structure that make it useful for creating a reusable object-oriented design.'

– E. Gamma (1995):

Software Design Patterns (cont'd)

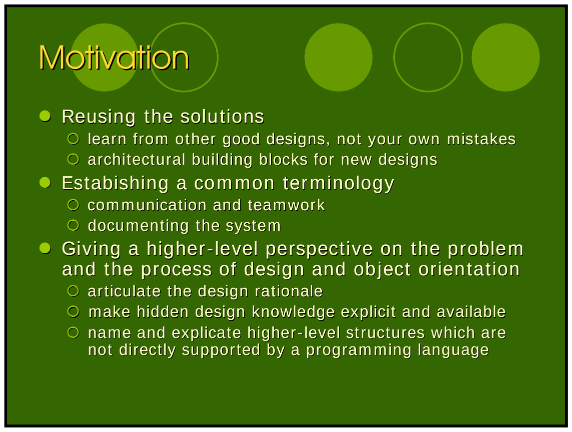
- Reusable solutions to general design problems
- Represent solutions to problems that arise when developing software within a particular context
  - design pattern = *problem–solution* pair in a *context*
  - basic steps remain the same but the exact way of applying a pattern is always different
- Capture well-proven experience in software development
  - static and dynamic structure
  - collaboration among the key participants
- Facilitate the reuse of successful software architectures and designs



Definition of a Design Pattern

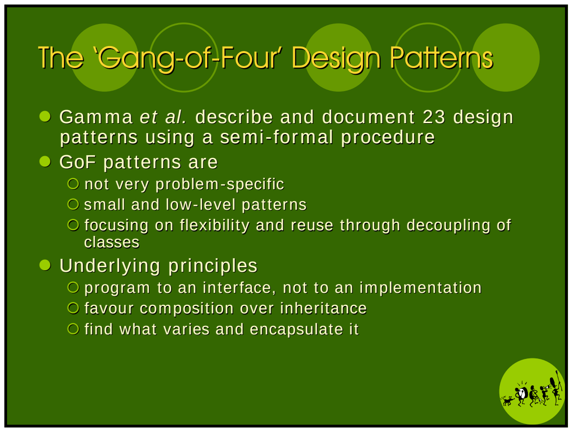
A GENERAL SOLUTION TO A FREQUENTLY OCCURRING ARCHITECTURE/DESIGN PROBLEM IN A CONTEXT.

- Not specific to any language, environment etc.
- Described as a semiformal document
- Addresses a common problem
- Can be applied at architecture or detailed design level
- Appears in a context that defines certain requirements or forces



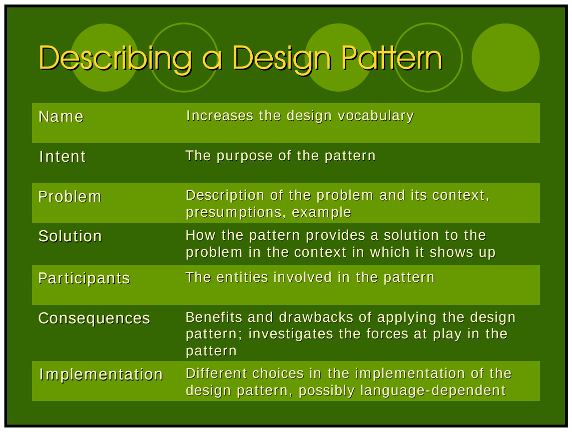
Motivation

- Reusing the solutions
  - learn from other good designs, not your own mistakes
  - architectural building blocks for new designs
- Estabishing a common terminology
  - communication and teamwork
  - documenting the system
- Giving a higher-level perspective on the problem and the process of design and object orientation
  - articulate the design rationale
  - make hidden design knowledge explicit and available
  - name and explicate higher-level structures which are not directly supported by a programming language



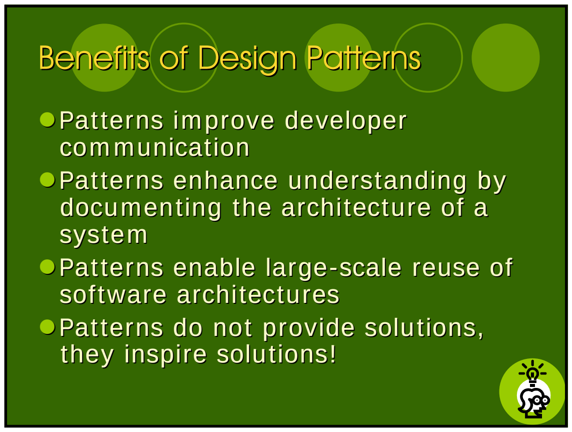
The 'Gang-of-Four' Design Patterns

- Gamma *et al.* describe and document 23 design patterns using a semi-formal procedure
- GoF patterns are
  - not very problem-specific
  - small and low-level patterns
  - focusing on flexibility and reuse through decoupling of classes
- Underlying principles
  - program to an interface, not to an implementation
  - favour composition over inheritance
  - find what varies and encapsulate it



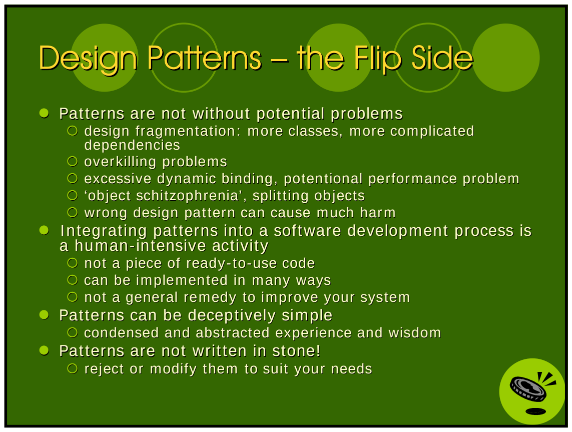
Describing a Design Pattern

Name	Increases the design vocabulary
Intent	The purpose of the pattern
Problem	Description of the problem and its context, presumptions, example
Solution	How the pattern provides a solution to the problem in the context in which it shows up
Participants	The entities involved in the pattern
Consequences	Benefits and drawbacks of applying the design pattern; investigates the forces at play in the pattern
Implementation	Different choices in the implementation of the design pattern, possibly language-dependent



Benefits of Design Patterns

- Patterns improve developer communication
- Patterns enhance understanding by documenting the architecture of a system
- Patterns enable large-scale reuse of software architectures
- Patterns do not provide solutions, they inspire solutions!

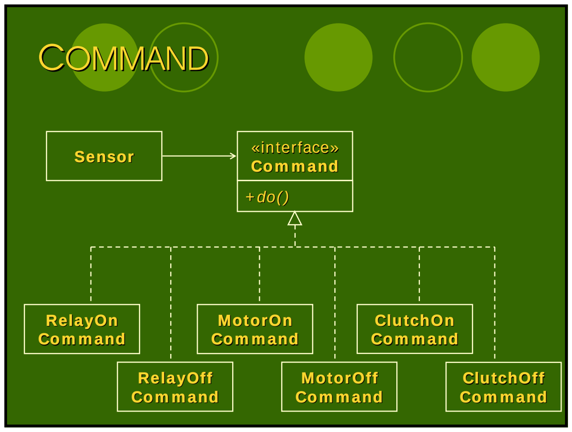


Design Patterns – the Flip Side

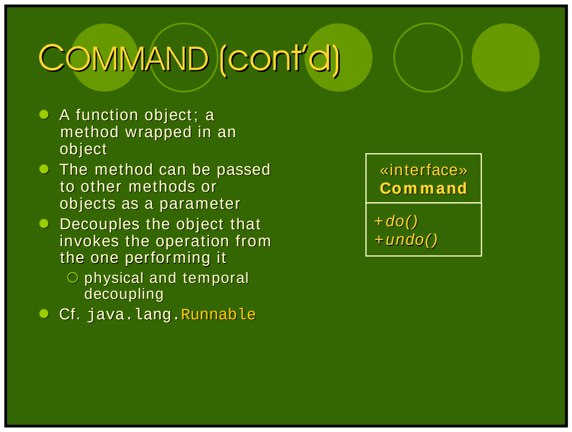
- Patterns are not without potential problems
  - design fragmentation: more classes, more complicated dependencies
  - overkilling problems
  - excessive dynamic binding, potentional performance problem
  - 'object schitzophrenia', splitting objects
  - wrong design pattern can cause much harm
- Integrating patterns into a software development process is a human-intensive activity
  - not a piece of ready-to-use code
  - can be implemented in many ways
  - not a general remedy to improve your system
- Patterns can be deceptively simple
  - condensed and abstracted experience and wisdom
- Patterns are not written in stone!
  - reject or modify them to suit your needs

Design Patterns: Set 1

- COMMAND and ACTIVE OBJECT
- TEMPLATE METHOD and STRATEGY
- FACADE and MEDIATOR
- SINGLETON and MONOSTATE
- NULL OBJECT



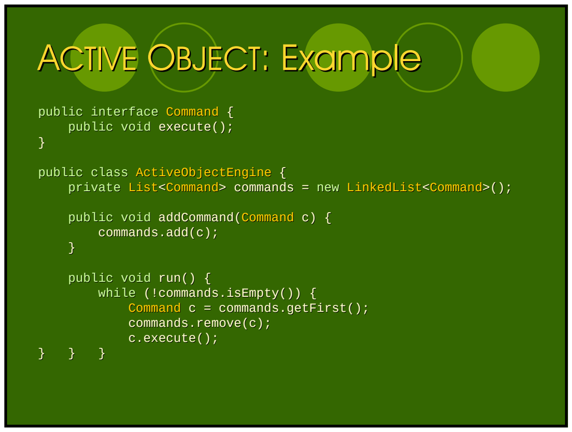
COMMAND



COMMAND (cont'd)

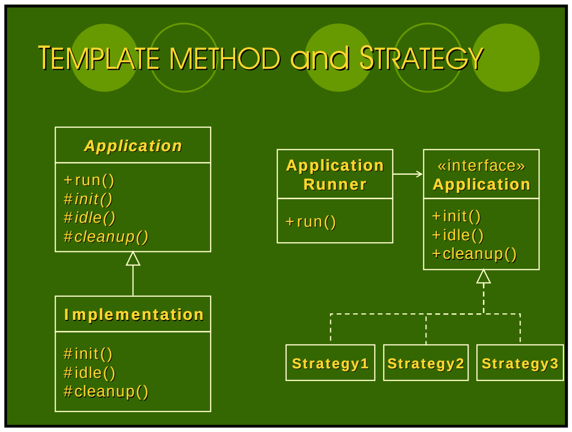
- A function object; a method wrapped in an object
- The method can be passed to other methods or objects as a parameter
- Decouples the object that invokes the operation from the one performing it
  - physical and temporal decoupling
- Cf. `java.lang.Runnable`

«interface» Command
+do()
+undo()

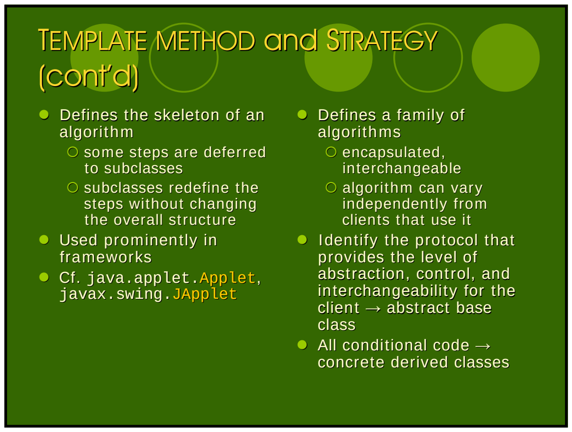


ACTIVE OBJECT: Example

```
public interface Command {
    public void execute();
}

public class ActiveObjectEngine {
    private List<Command> commands = new LinkedList<Command>();

    public void addCommand(Command c) {
        commands.add(c);
    }

    public void run() {
        while (!commands.isEmpty()) {
            Command c = commands.getFirst();
            commands.remove(c);
            c.execute();
}   }   }
```



TEMPLATE METHOD and STRATEGY

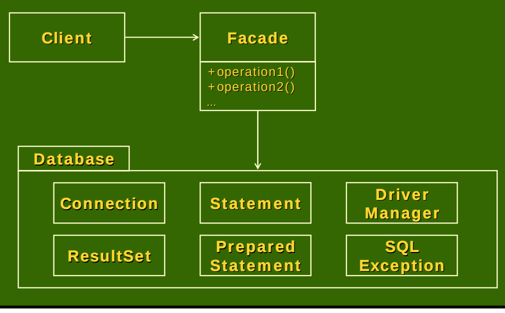


TEMPLATE METHOD and STRATEGY (cont'd)

- Defines the skeleton of an algorithm
  - some steps are deferred to subclasses
  - subclasses redefine the steps without changing the overall structure
- Used prominently in frameworks
- Cf. `java.applet.Applet`, `javax.swing.JApplet`

- Defines a family of algorithms
  - encapsulated, interchangeable
  - algorithm can vary independently from clients that use it
- Identify the protocol that provides the level of abstraction, control, and interchangeability for the client → abstract base class
- All conditional code → concrete derived classes

FACADE

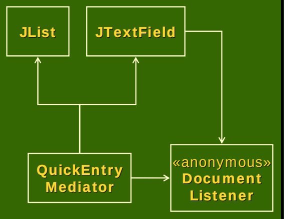


FACADE (cont'd)

- A unified interface to a set of interfaces in a subsystem
  - encapsulates a complex subsystem within a single interface object
  - makes the subsystem easier to use
- Decouples the subsystem from its clients
  - if it is the only access point, it limits the features and flexibility
- Imposes a policy 'from above'
  - everyone uses the facade instead the subsystem
  - visible and constraining

MEDIATOR

- Imposes a policy 'from below'
  - hidden and unconstraining
- Promotes loose coupling
  - objects do not have to refer to one another
  - simplifies communication
- Problem: monolithism
- Example: QuickEntryMediator
  - binds text-entry field to a list
  - when text is entered, the first element matching in the list is highlighted



SINGLETON: Example

```
public class Singleton {
    private static Singleton
                            theInstance = null;

    private Singleton() { /* nothing */ }

    public static Singleton create() {
        if (theInstance == null)
            theInstance = new Singleton();
        return theInstance;
}   }
```

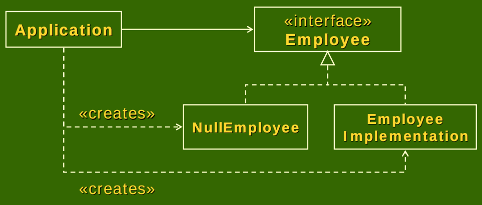
MONOSTATE: Example

```
public class Monostate<T> {
    private static T itsValue = null;

    public Monostate() { /* nothing */ }

    public void set(T value) {
        itsValue = value;
    }
    public T get() {
        return itsValue;
}   }
```

Comparison

- SINGLETON
  - applicable to any class
  - lazy evaluation: if not used, not created
  - not inherited: a derived class is not singleton
  - can be created through derivation
  - non-transparent: the user knows…
  - cf. java.lang.Integer.MAX_VALUE, java.util.Collections.EMPTY_SET
- MONOSTATE
  - inherited: a derived class is monostate
  - polymorphism: methods can be overridden
  - normal class cannot be converted through derivation
  - transparent: the user does not need to know…

NULL OBJECT



NULL OBJECT: Example

```
public interface Employee {
    public boolean isTimeToPay(Date payDate);
    public void pay();

    public static final Employee NULL =
                                 new Employee() {
        public boolean isTimeToPay(Date payDate) {
            return false;
        }
        public void pay() { /* nothing */ }
      };
}
```

Reading for the Next Week

- Section 4: Packaging the Payroll System
  - Chapter 20: Principles of Package Design
  - Chapter 21: FACTORY
  - Chapter 22: The Payroll Case Study (Part 2)