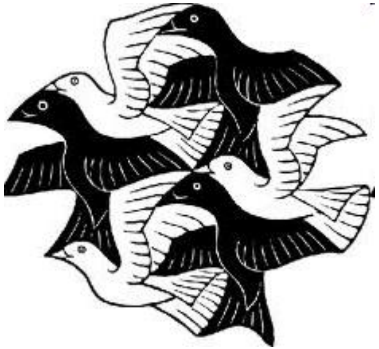


10. Best Practice Software Engineering

SW Design Patterns**,**
by Boyan Bontchev,
FMI - Sofia University
© 2006/2017

Annotation

- The Best Practice Software Engineering (BPSE) project
- Classification
- BPSE patterns
- BPSE relationships
- Conclusions

References

- Schatten, A., Biffl, S., Demolsky, M., Gostischa-Franta, E., Östreicher, Th., Winkler, D. (2010) Best Practice Software-Engineering: Eine praxiserprobte Zusammenstellung von komponentenorientierten Konzepten, Methoden und Werkzeugen, Spektrum Akademischer Verlag, Springer, Vienna, Austria.
- http://best-practice-software-engineering.ifs.tuwien.ac.at/

The Best Practice Software Engineering (BPSE) project

- Initiated in TU-Vienna, 2008-2010, by Alexander Schatten.
- The team said: “… we cannot "*teach*„ unexperienced developers *experience*, but what we can do is provide so called "best practices".
- Some of the good and bad decisions from concrete projects can be abstracted to scenarios. E.g., how to implement a persistence solution for a business application, how to implement a robust GUI that is also easy to maintain and so on”.

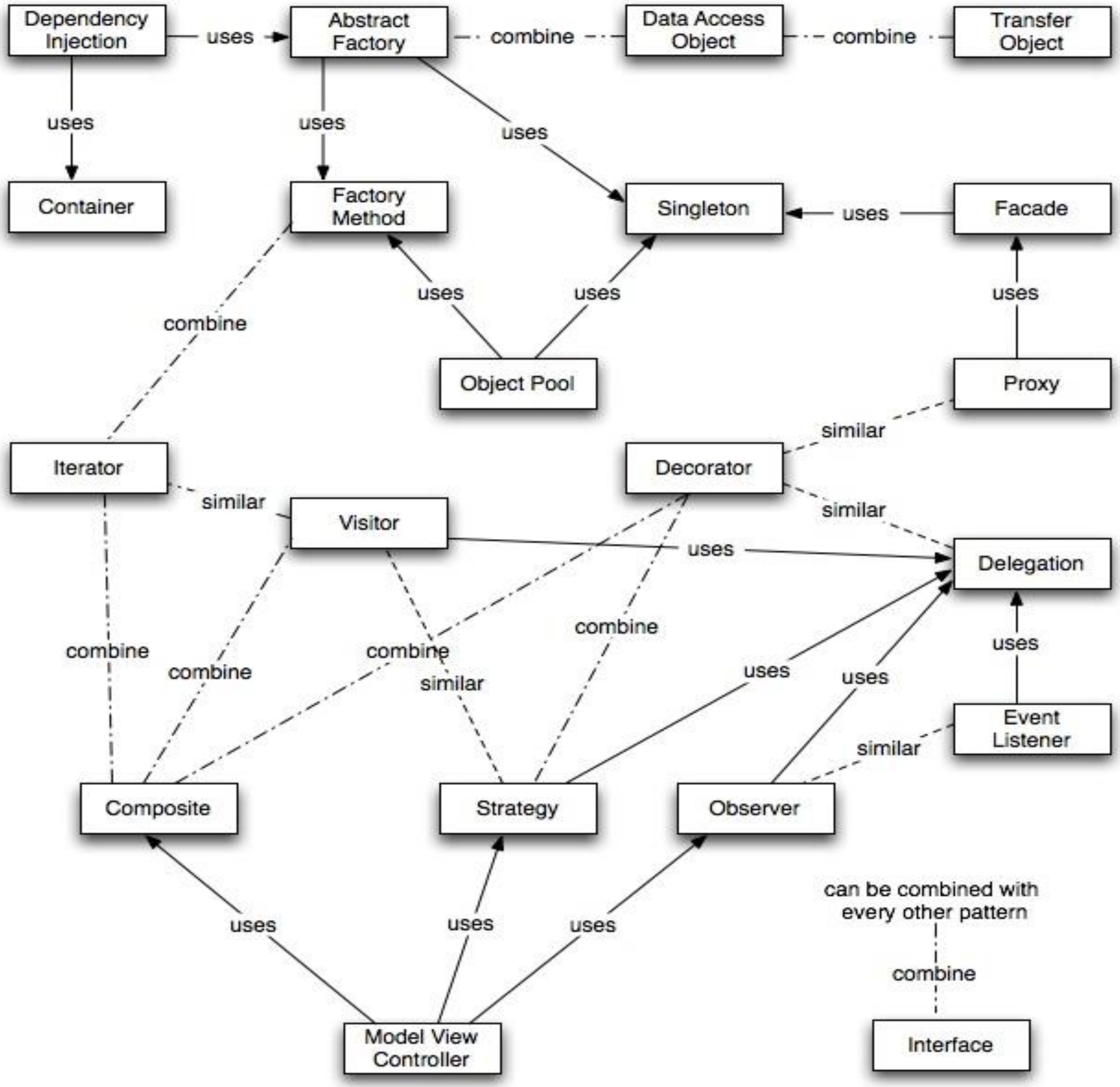
BPSE experience in two levels

- The first level is called **Software Patterns**:
  - □ **Software Patterns** describe how to solve typical problems that appear in many application scenarios in similar ways.
  - □ A "pattern" has been defined as *"an idea that has been useful in one practical context and will probably be useful in others."* [M. Fowler, "Analysis Patterns - Reusable Object Models", Addison Wesley, ISBN 0-201-89542-0].
  - □ These patterns can be seen as abstracted knowledge, that guides you in better solving your concrete problems.
- The second level is: **technical-implementation best-practices**

BPSE pattern categories

- The BPSE patterns are divided into five categories reinforcing and complementing each other:
  - *Fundamental*
  - *Architectural*
  - Creational
  - Structural, and
  - Behavioral
- Usually, patterns inside one category complement each other because they have the same underlying principles for structuring code.

BPSE pattern map



BPSE pattern catalogue: 1

1. Fundamental Design Patterns are general concepts, they are needed in most other patterns to accomplish their task.

- **Interface**
- **Container**
- **Delegation**

2. Architectural Patterns express a fundamental structural organization or schema for software systems.

3. Structural Design Patterns are concerned with how classes and objects are composed together to form larger structures. [GoF, "Design Patterns"]

4. Creational Design Patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. [GoF, "Design Patterns", Addison Wesley, ISBN 0201633612]

5. Behavioral Design Patterns are concerned with algorithms and the assignment of responsibilities between objects. [GoF, "Design Patterns"]

Fundamental Patterns: Interface

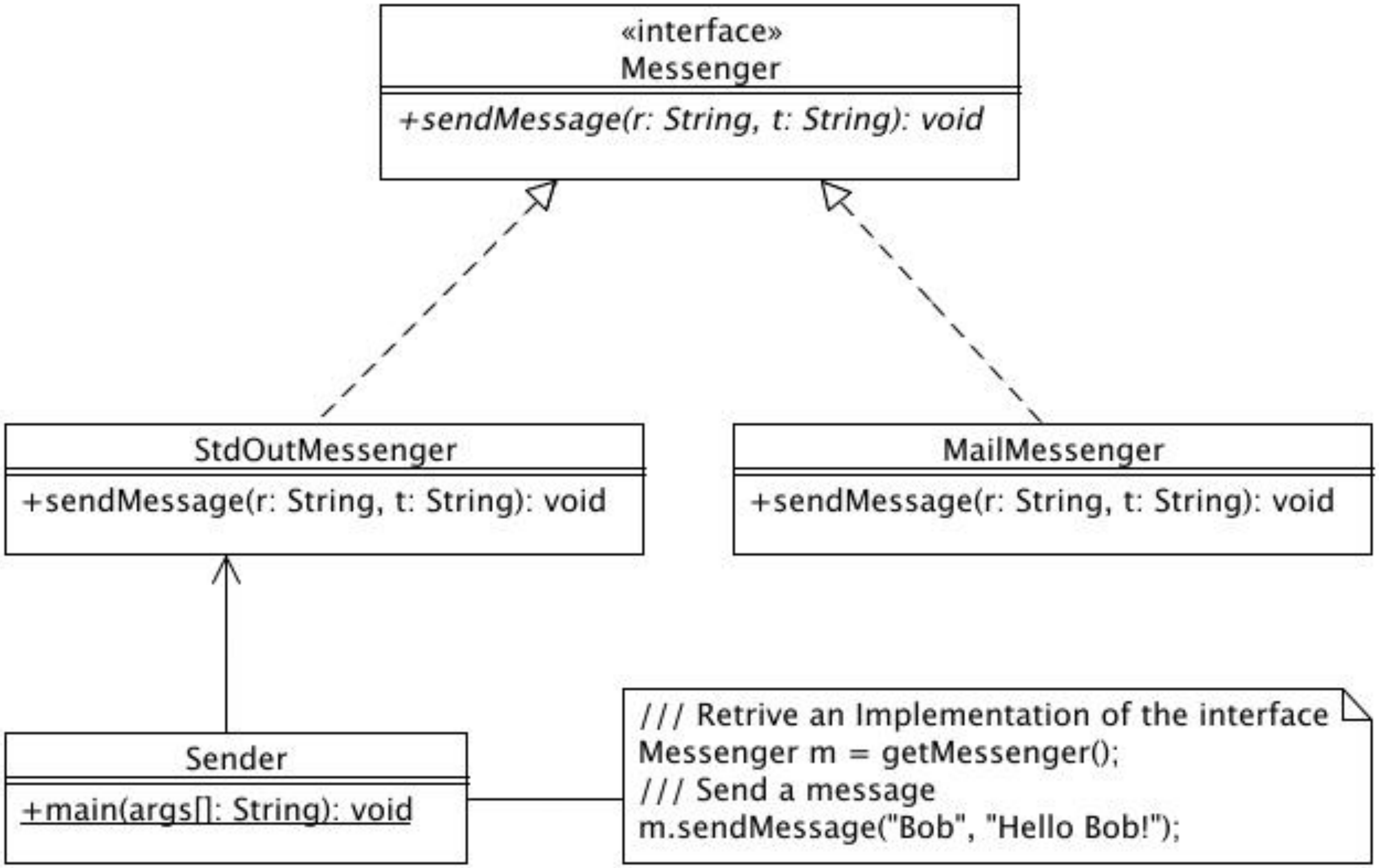
Definition:

- an abstraction defining the signature operations of an entity
- sets the communication boundary between two entities
- separates functions from implementations - a class can be exchanged easily without changing the code of the caller.

Applicability:

- you want to specify how classes exchange messages - every time, when a class should be reused, or used outside a specific context (package), declare the communication interface as an **Interface** type
- you have to switch the implementation of a module during run-time
- at design-time you don't yet know which implementation you will use at compile-time

Sample structure

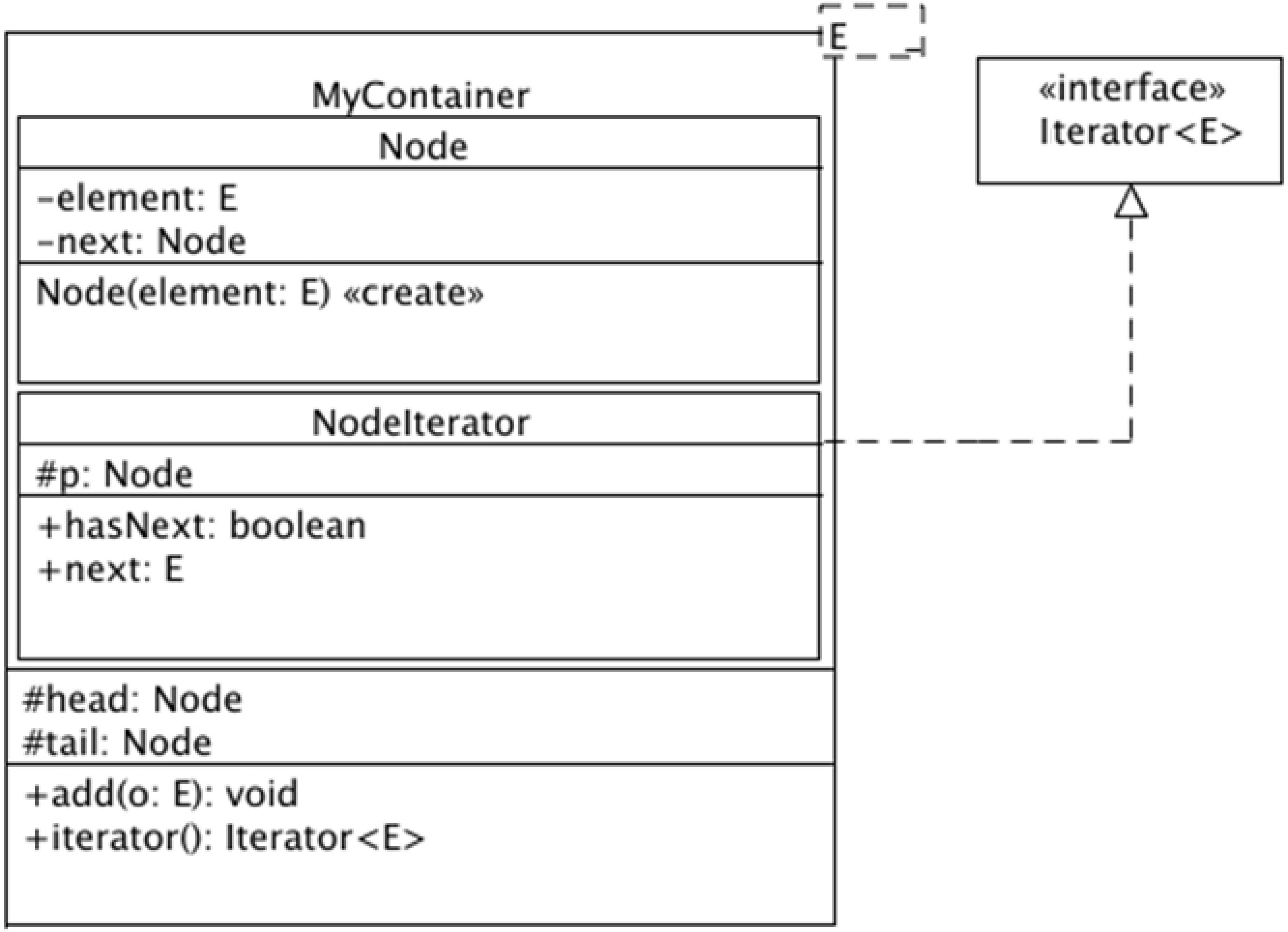

«interface»
Messenger
+sendMessage(r: String, t: String): void
StdOutMessenger
+sendMessage(r: String, t: String): void
MailMessenger
+sendMessage(r: String, t: String): void
Sender
+main(args[]: String): void

```
/// Retrive an Implementation of the interface
Messenger m = getMessenger();
/// Send a message
m.sendMessage("Bob", "Hello Bob!");
```

Fundamental Patterns: Container

- A Container is an object created to hold other objects that are accessed, placed, and maintained with the class methods of the container - queues, sets, lists, vectors, and caches all fit this description.
- These objects - the elements of the container - are usually allowed to be of any class and may be of the container class itself.
- Every Container should also have an associated Iterator type that can be used to iterate through the elements of the container.
- Java programmers tend to call these types of classes "collections" rather than "containers".
- Within the Spring Framework containers represent much more higher level concepts such as inversion of control

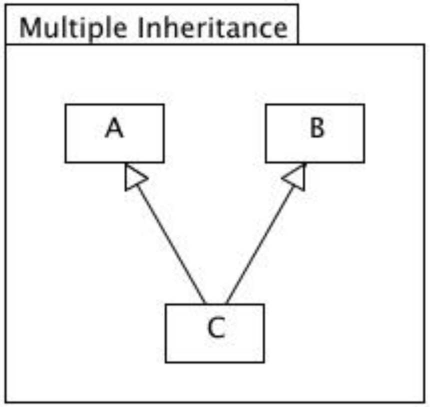
Sample structure

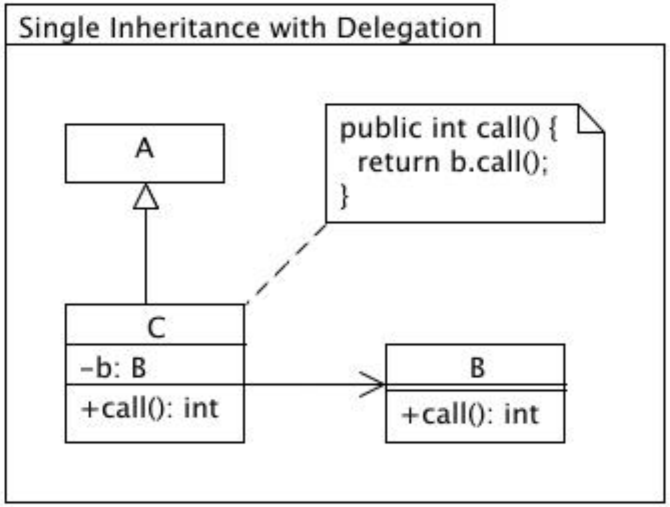


Code: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/container.html

Fundamental Patterns: Delegation

- "*Delegation is like inheritance done manually through object composition*.“





```
public int call() {
  return b.call();
}
```

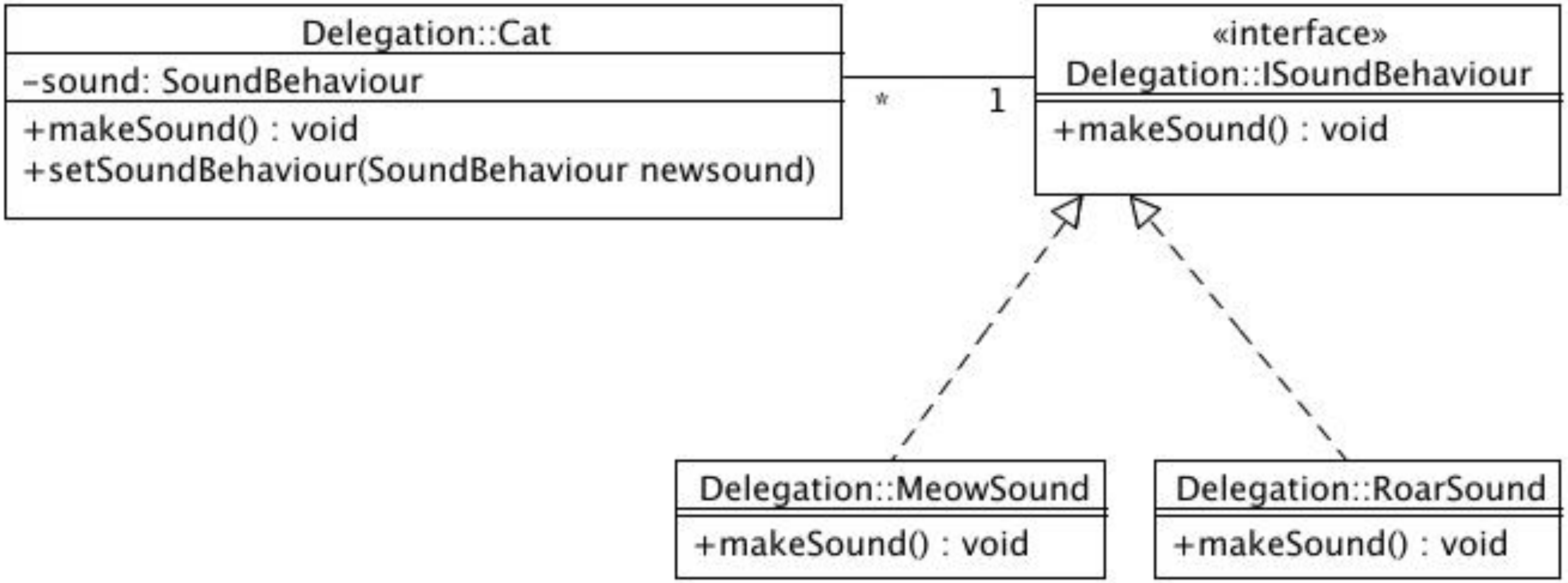
Applicability:

- to reduce the coupling of methods to their class
- you have components that behave identically, but realize that this situation can change in the future.

Related patterns:

- Decorator, Visitor, Observer, Strategy, …

Sample structure



How delegation makes it easy to compose behaviors at run-time?

Code: http://best-practice-software-engineering.ifs.tuwien.ac.at/patterns/delegation.html

BPSE pattern catalogue: 2

1. Fundamental Design Patterns are general concepts, they are needed in most other patterns to accomplish their task.

2. Architectural Patterns express a fundamental structural organization or schema for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

- **Model View Controller (MVC)**
- **Dependency Injection**

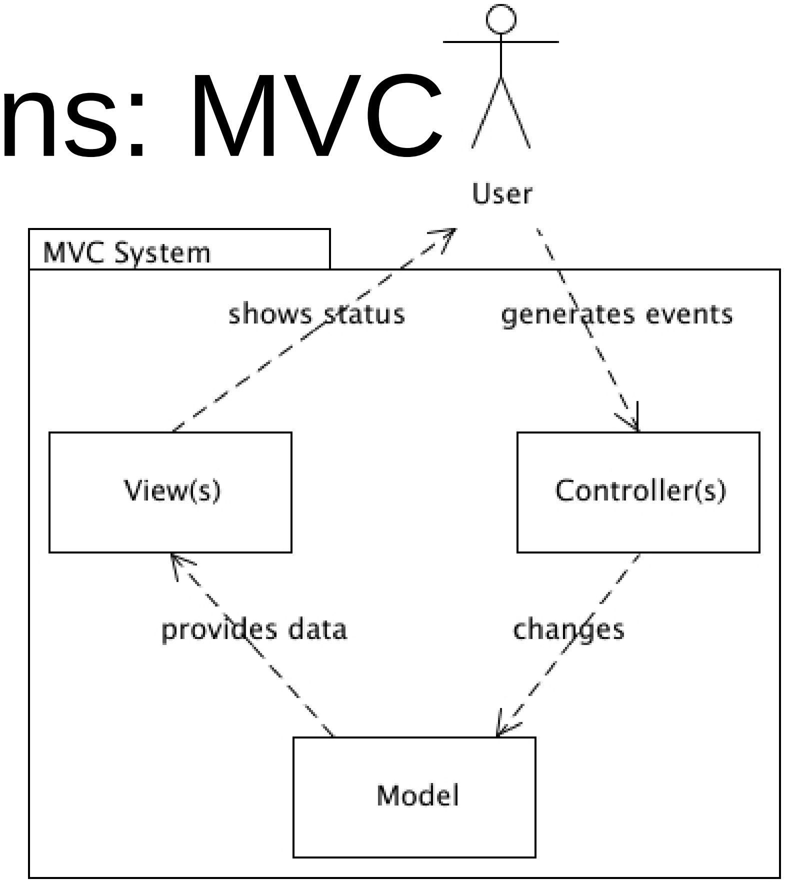
3. Structural Design Patterns are concerned with how classes and objects are composed together to form larger structures. [GoF, "Design Patterns"]

4. Creational Design Patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. [GoF, "Design Patterns"]

5. Behavioral Design Patterns are concerned with algorithms and the assignment of responsibilities between objects. [GoF, "Design Patterns"]

Architectural Patterns: MVC

- **Model** objects hold data and define the logic for manipulating that data. Model objects are not directly displayed. They often are reusable, distributed, persistent and portable to a variety of platforms.
- **View** objects represent something visible in the user interface, for example a panel or button.



The **Controller** communicates data back and forth between the Model objects and the View objects. A controller also performs all application specific tasks, such as processing user input or loading configuration data. There is usually one controller per application or window, in many applications the Controller is tightly coupled to the view.

More about MVC

Applicability

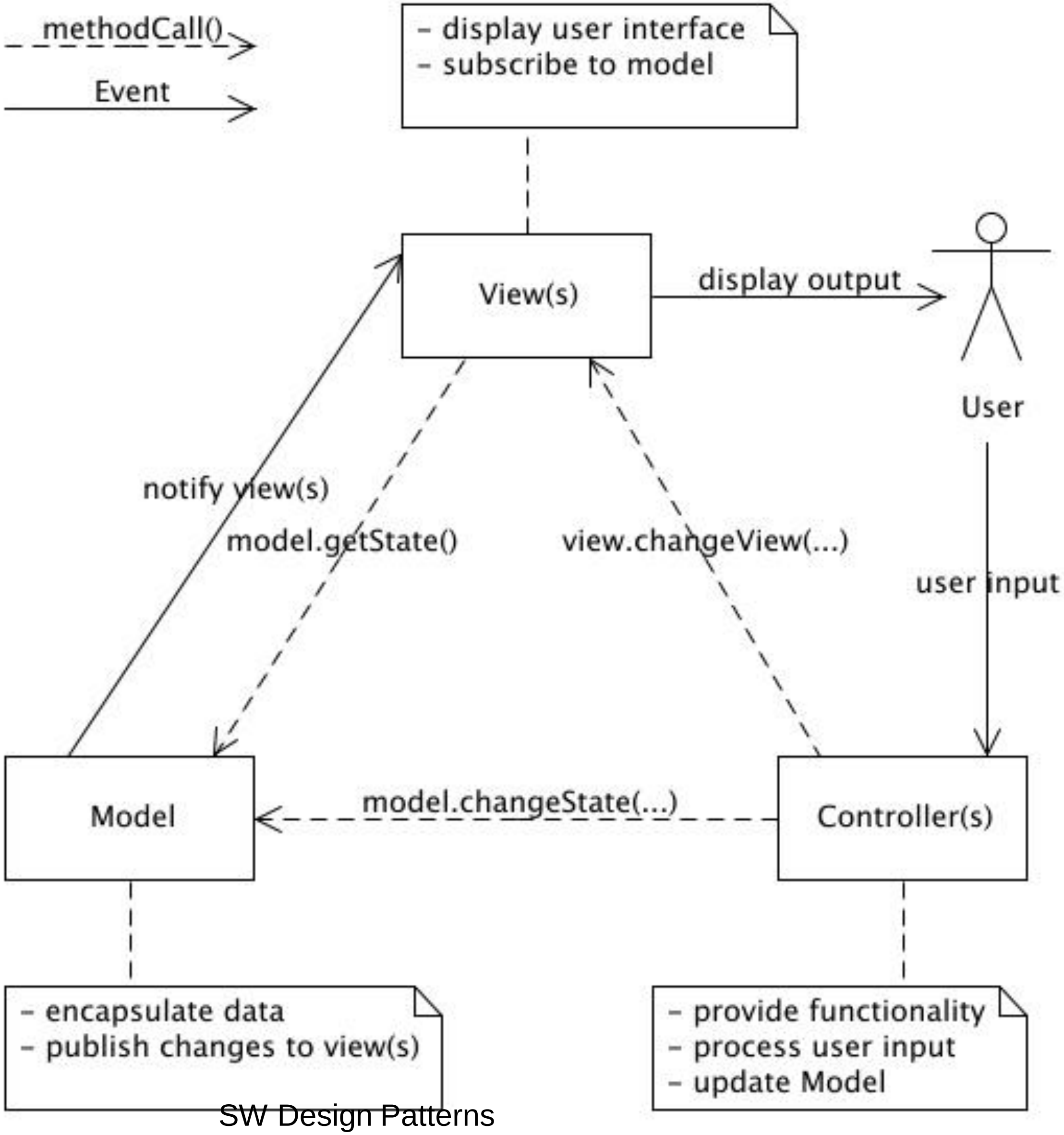
- almost in every application - depending on the application some classes might be coupled tighter than others, however it is generally always a good idea to structure your application according to MVC.

Related patterns

- Observer: used for loose coupling in Model and View. When the Model classes change, the View classes need to be notified and updated with the latest information.
- Strategy: used by the Model. The Data Access Object pattern is a form of the strategy pattern it is used primarily by the Model to access different form of data-sources (SQL, XML files, …)
- Composite: used by the View. There can be several different views within a MVC system, whose different implementations can be composed together and exchanged at run time as composites.

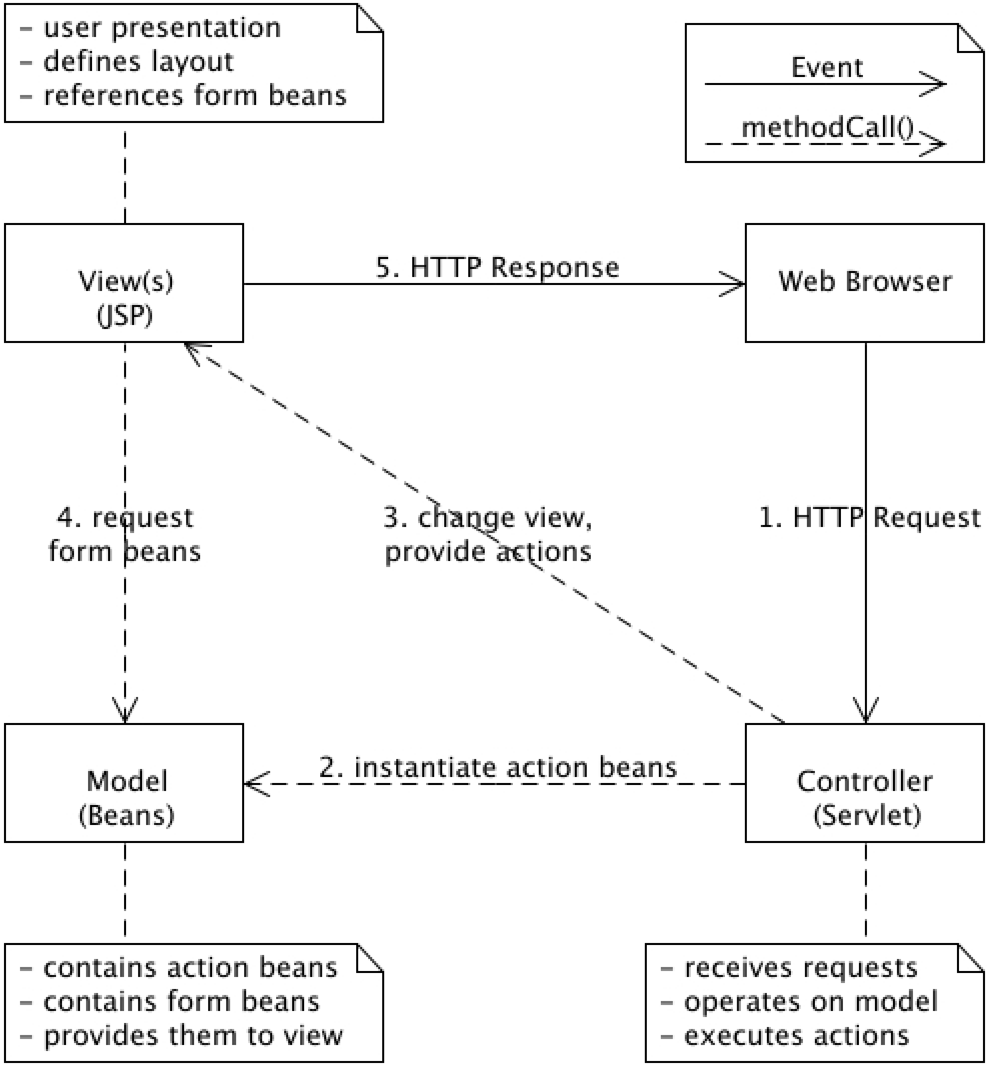
Sample structure 1:

Smalltalk



Sample structure 2:

Java Web App



Inversion of Control

- **Inversion of Control (IoC)** - a design principle in which custom-written portions of a computer program receive the flow of control from a generic framework (R. Martin and M. Fowler)
- Also known as the **Hollywood Principle** - "Don't call us, we'll call you"
- EJBs are a good example of this style of IoC. When you develop a session bean, you can implement various methods that are called by the EJB container at various lifecyle points. For example the Session Bean interface defines ejbRemove, ejbPassivate (stored to secondary storage), and ejbActivate (restored from passive state). You don't get to control when these methods are called, just what they do. The container calls us, we don't call it.

Architectural Patterns: Dependency Injection

- Based on IoC
- Relates to the way in which an object obtains references to its dependencies - the object is passed its dependencies through constructor arguments or after construction through setter methods or interface methods
- There are 3 forms of dependency injection: setter- (the recommended methodology using the Spring Framework), constructor- and interface-based injection
- Example: a business service will work with an interface of the Data Access Object (DAO). With the IoC pattern, as implemented in Spring, the developer defines which DAO in an external configuration file (in beans.xml). At runtime, the information from the configuration file are parsed and the dependency is *injected* into the service.

More about Dependency Injection

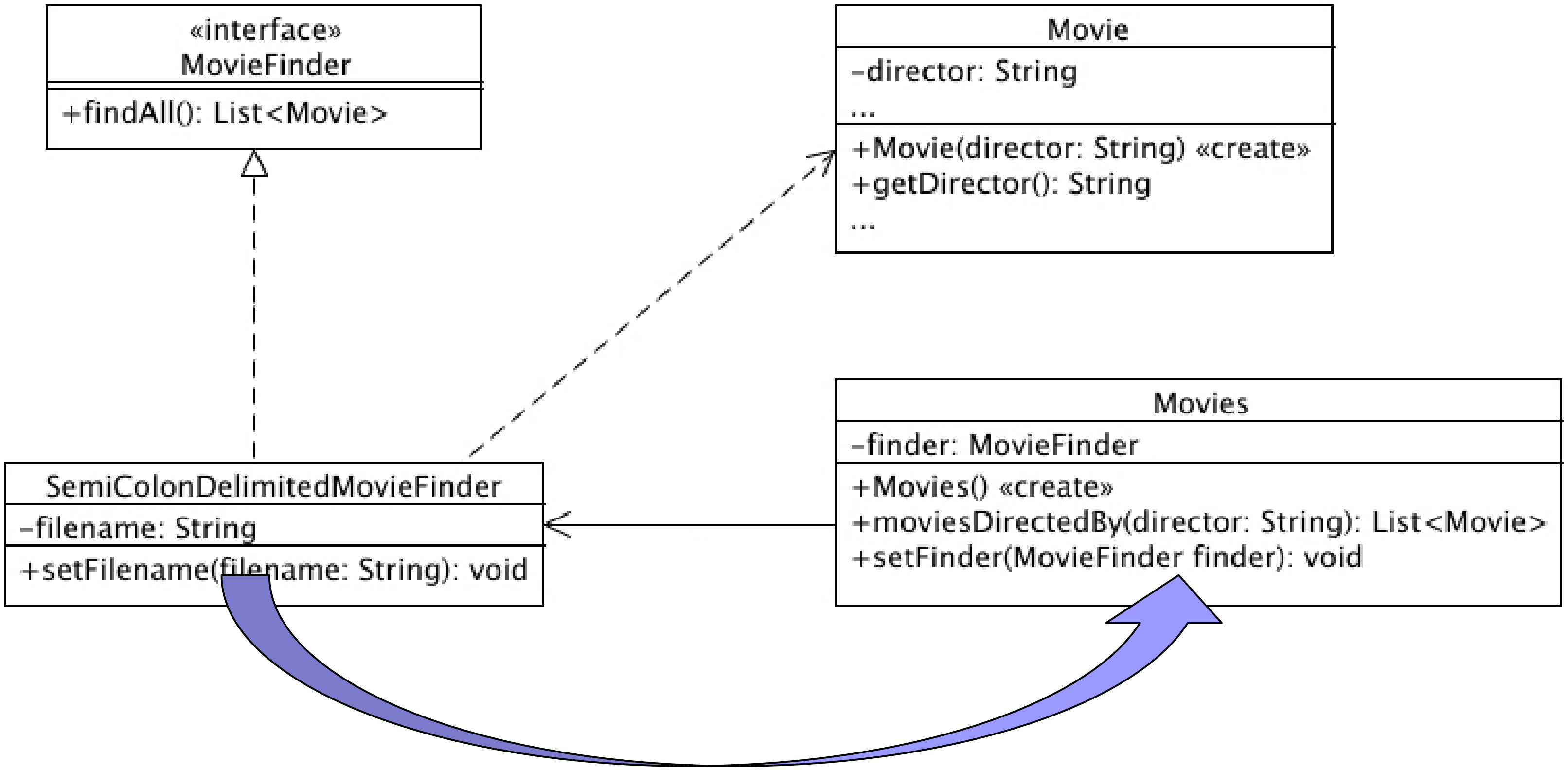
Applicability

- when the coupling between components needs to be reduced
- you want to save time in that you don't have to write *boilerplate* factory creation code over and over again

Related Patterns

- Abstract Factory: allows an application to acquire objects and components without exposing too much information about how to components fit together or what dependencies each component might have.
- Container: allows objects to be configured by the container instead of the client. A container can configure objects declaratively, e.g. using XML files.

Sample Structure



«interface»
MovieFinder

+findAll(): List<Movie>

Movie

-director: String
...

+Movie(director: String) «create»
+getDirector(): String
...

SemiColonDelimitedMovieFinder

-filename: String

+setFilename(filename: String): void

Movies

-finder: MovieFinder

+Movies() «create»
+moviesDirectedBy(director: String): List<Movie>
+setFinder(MovieFinder finder): void

```
<bean id="MovieFinder"
class="spring.ColonMovieFinder">
    <property name="filename">
              <value>movies1.txt</value>
    </property>
</bean>
```

BPSE pattern catalogue: 3

1. Fundamental Design Patterns are general concepts, they are needed in most other patterns to accomplish their task.

2. Architectural Patterns express a fundamental structural organization or schema for software systems.

3. Structural Design Patterns are concerned with how classes and objects are composed together to form larger structures. [GoF, "Design Patterns"]

- **Facade**
- **Decorator**
- **Proxy**
- **Data Access Object**
- **Transfer Object**

4. Creational Design Patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. [GoF, "Design Patterns"]

5. Behavioral Design Patterns are concerned with algorithms and the assignment of responsibilities between objects. [GoF, "Design Patterns"]

Structural Patterns: DAO

- Access to persistent data varies greatly depending on the type of storage (database, flat files, xml files, and so on) and it even differs from its implementation (for example different SQL-dialects).
- The goal is to abstract and encapsulate all access to the data and provide an interface. This is called the Data Access Object (DAO) pattern. The DAO "knows" which data source (that could be a database, a flat file or even a WebService) to connect to and is specific for this data source.
- From the applications point of view, it makes no difference when it accesses a relational database or parses XML files (using a DAO). The DAO is usually able to create an instance of a data object ("to read data") and also to persist data ("to save data") to the datasource.

DAO: when and how?

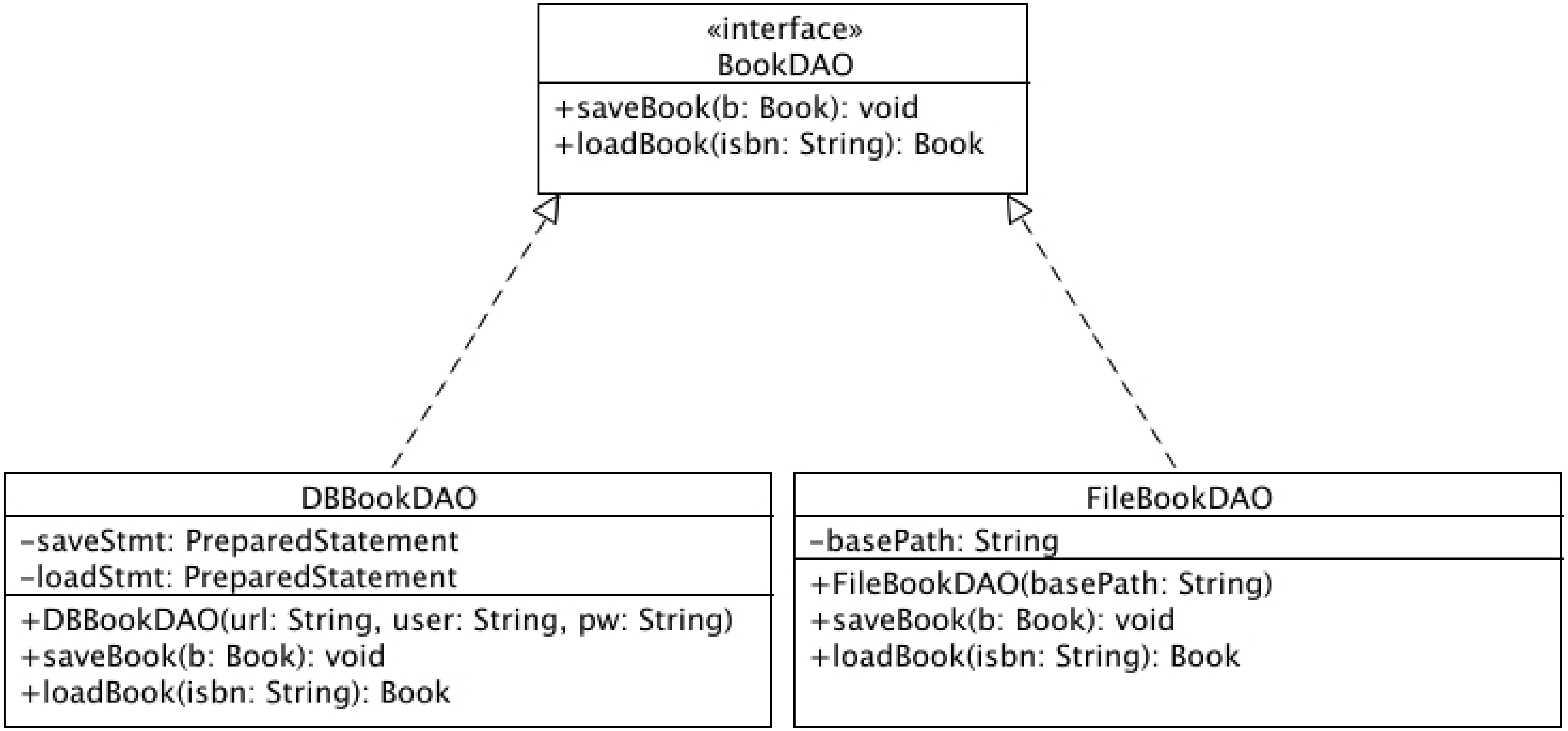
Applicability

- when you need to access a persistent storage more than one time, especially if you want to exchange the data source later.
- you want to separate a data resource's client interface from its data access mechanisms
- you want to adapt a specific data resource's access API to a generic client interface, with clean separation of concerns.

Related Patterns

- Abstract Factory: Applications often use a Factory to select the right DAO implementation at run time.
- Transfer Object: The DAO pattern often uses a Transfer Object to send data from the data source to its client and vice versa.

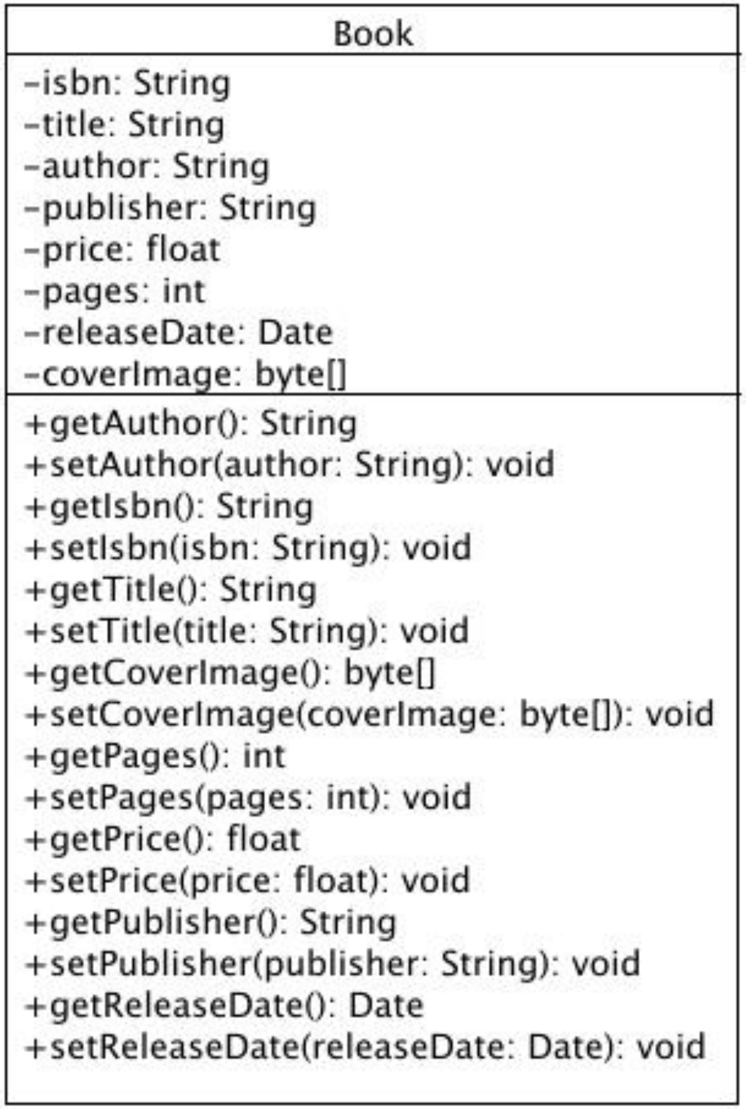
Sample Structure


«interface»
BookDAO
+saveBook(b: Book): void
+loadBook(isbn: String): Book
DBBookDAO
-saveStmt: PreparedStatement
-loadStmt: PreparedStatement
+DBBookDAO(url: String, user: String, pw: String)
+saveBook(b: Book): void
+loadBook(isbn: String): Book
FileBookDAO
-basePath: String
+FileBookDAO(basePath: String)
+saveBook(b: Book): void
+loadBook(isbn: String): Book

Structural Patterns: Transfer Object

- A Transfer Object is an object encapsulating data. A single method call is now sufficient to send and retrieve the Transfer Object and all included data.
- Applicability:
  - when the number of calls made by a client to a Data Access Object or Enterprise Bean impacts network performance
  - you want to reduce communication effort when dealing with a lot of small data entities
- Related Patterns
  - Data Access Object: A Transfer Object pattern is often used in combination with a Data Access Object pattern.

Sample Structure



```
Book
-isbn: String
-title: String
-author: String
-publisher: String
-price: float
-pages: int
-releaseDate: Date
-coverImage: byte[]
+getAuthor(): String
+setAuthor(author: String): void
+getIsbn(): String
+setIsbn(isbn: String): void
+getTitle(): String
+setTitle(title: String): void
+getCoverImage(): byte[]
+setCoverImage(coverImage: byte[]): void
+getPages(): int
+setPages(pages: int): void
+getPrice(): float
+setPrice(price: float): void
+getPublisher(): String
+setPublisher(publisher: String): void
+getReleaseDate(): Date
+setReleaseDate(releaseDate: Date): void
```

BPSE pattern catalogue: 4

1. Fundamental Design Patterns are general concepts, they are needed in most other patterns to accomplish their task.

2. Architectural Patterns express a fundamental structural organization or schema for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

3. Structural Design Patterns are concerned with how classes and objects are composed together to form larger structures. [GoF]

4. Creational Design Patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. [GoF, "Design Patterns"]

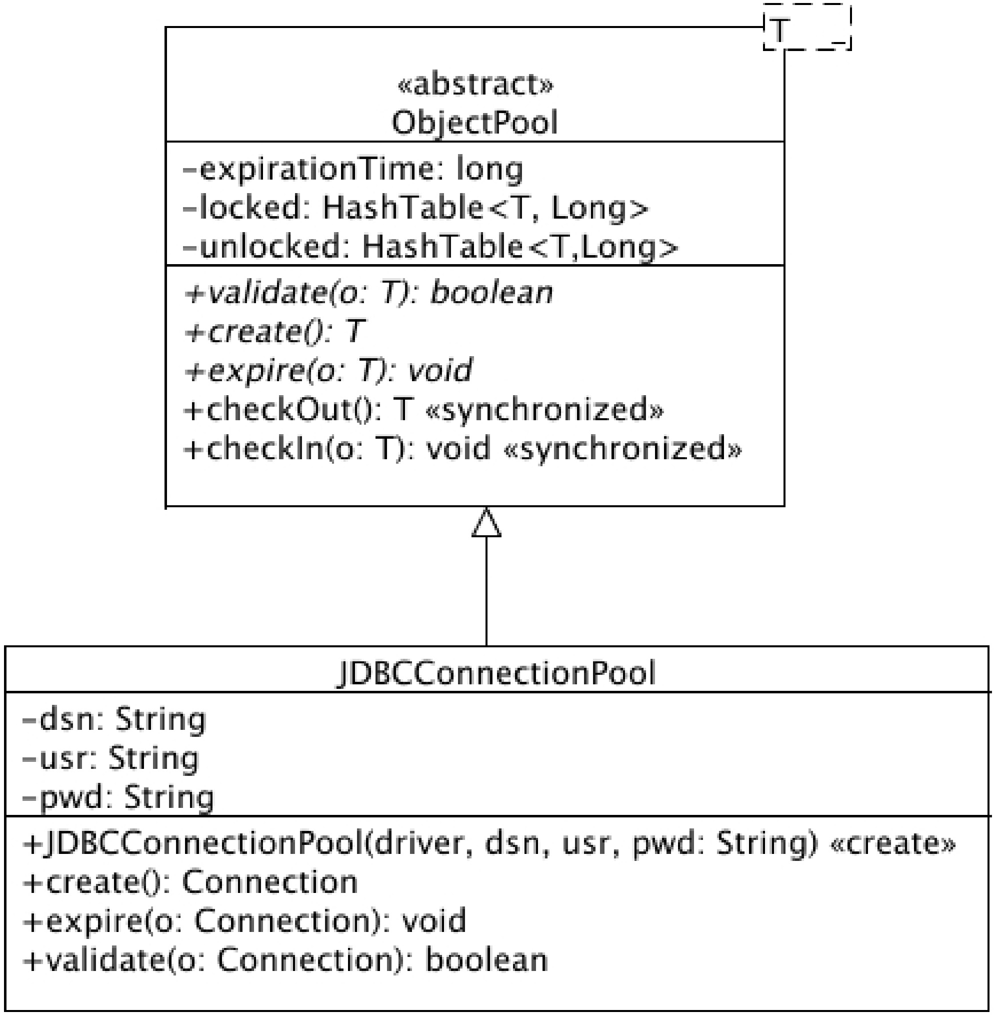
- **Factory Method**
- **Abstract Factory**
- **Object Pool**
- **Singleton**

5. Behavioral Design Patterns are concerned with algorithms and the assignment of responsibilities between objects. [GoF, "Design Patterns"]

Creational Patterns: Object Pool

- It is cheaper (in regards to system memory and speed) for a process to borrow an object rather than to instantiate it.
- The Object Pool lets others "check out" objects from its pool, when those objects are no longer needed by their processes, they are returned to the pool in order to be reused. However, we don't want a process to have to wait for a particular object to be released, so the Object Pool also instantiates new objects as they are required, but must also implement a facility to clean up unused objects periodically.
- Applicability
  - when your application sporadically requires objects which are "expensive" to create.
  - several parts of your application require the same objects at different times.
- Related Patterns
  - Factory Method: The Factory Method pattern can be used to encapsulate the creation logic for objects. However, it does not manage them after their creation, the object pool pattern keeps track of the objects it creates.
  - Singleton: Object Pools are usually implemented as Singletons.

Sample Structure



T
«abstract»
ObjectPool
-expirationTime: long
-locked: HashTable<T, Long>
-unlocked: HashTable<T,Long>
+validate(o: T): boolean
+create(): T
+expire(o: T): void
+checkOut(): T «synchronized»
+checkIn(o: T): void «synchronized»
JDBCConnectionPool
-dsn: String
-usr: String
-pwd: String
+JDBCConnectionPool(driver, dsn, usr, pwd: String) «create»
+create(): Connection
+expire(o: Connection): void
+validate(o: Connection): boolean

BPSE pattern catalogue: 5

1. Fundamental Design Patterns are general concepts, they are needed in most other patterns to accomplish their task.

2. Architectural Patterns express a fundamental structural organization or schema for software systems. They provide a set of predefined subsystems, specify their responsibilities, and include rules and guidelines for organizing the relationships between them.

3. Structural Design Patterns are concerned with how classes and objects are composed together to form larger structures. [GoF , "Design Patterns"]

4. Creational Design Patterns abstract the instantiation process. They help make a system independent of how its objects are created, composed, and represented. [GoF, "Design Patterns"]

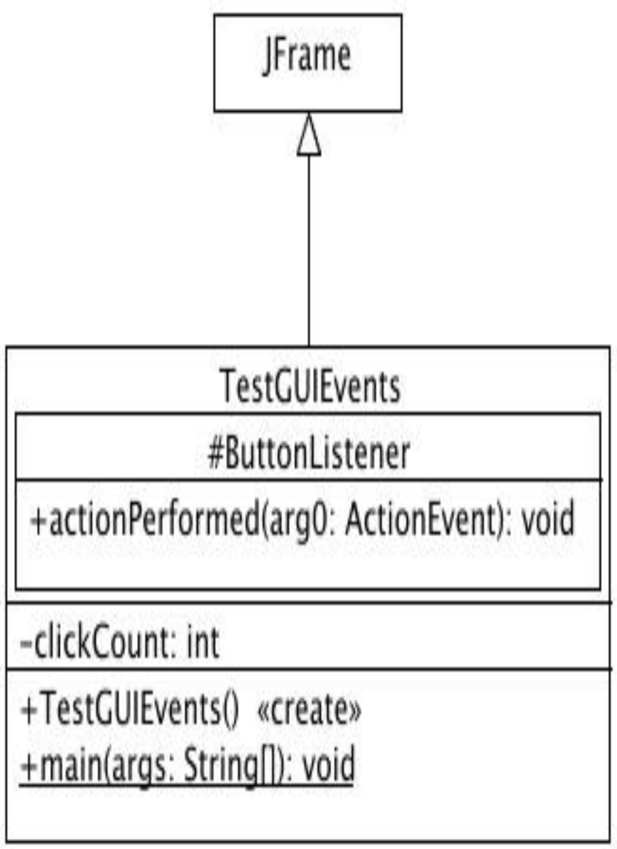
5. Behavioral Design Patterns are concerned with algorithms and the assignment of responsibilities between objects. [GoF, "Design Patterns"]

- **Iterator**
- **Observer**
- **Event Listener**
- **Strategy**

Behavioral Patterns: Event Listener

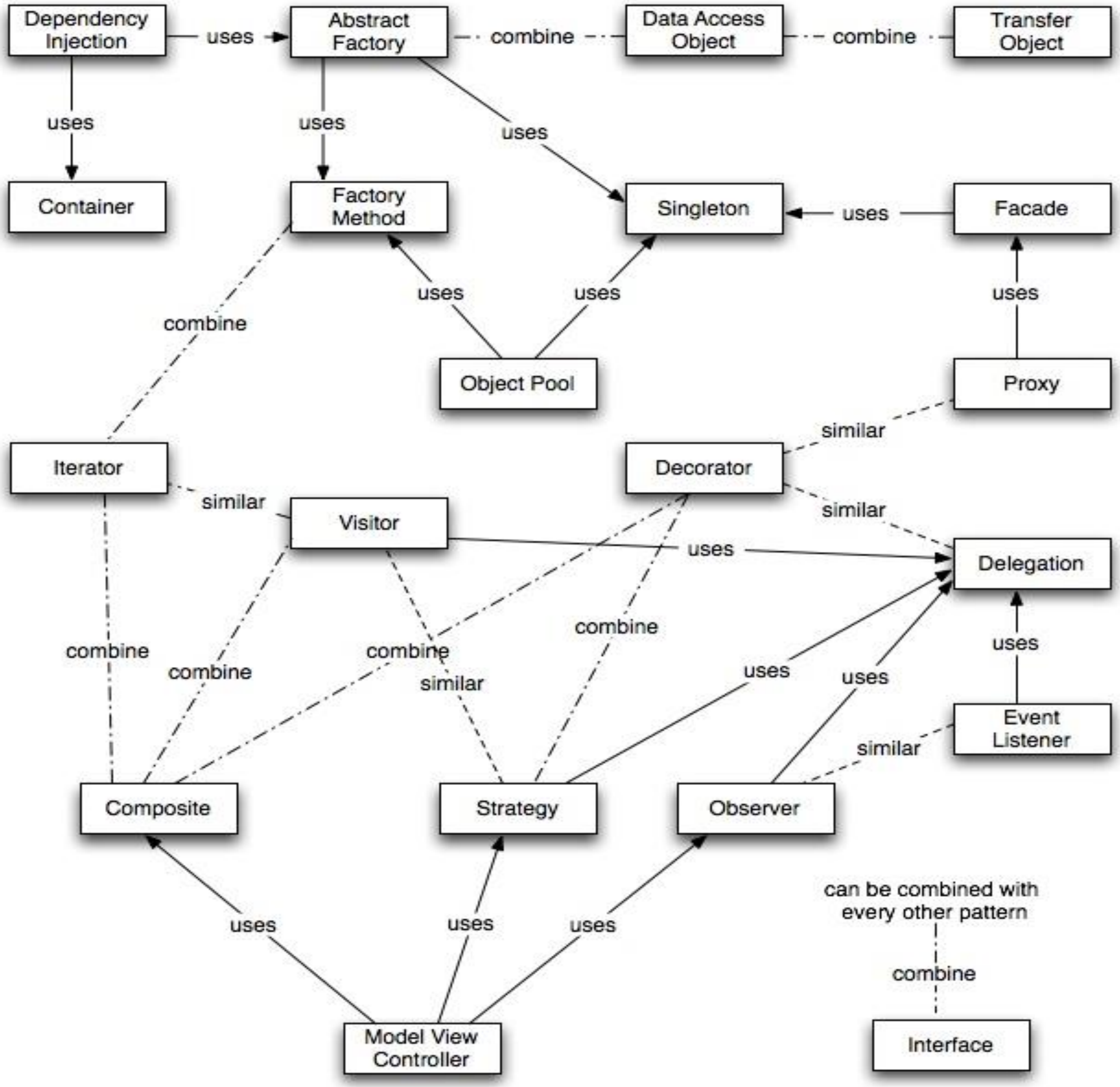
- Events are messages that are sent from one object to another. The component sending the event (aka firing the event) is the producer, the component receiving the event (aka handling the event) is the consumer. The producer of an event should have the ability to add and delete listeners for the events produced by itself.
- A predefined method of the listener is invoked by the producer when an event is fired.
- Applicability
  - □ Handling User-Interactions, such as clicking on a button, is realized through Events and Event-Listeners
- Related Patterns
  - □ Observer: The Event Listener pattern is a special flavor of Observer
  - □ Delegation: Event Listeners usually use delegation to accomplish their tasks.

Sample Structure


JFrame
TestGUIEvents
#ButtonListener
+actionPerformed(arg0: ActionEvent): void
-clickCount: int
+TestGUIEvents() «create»
+main(args: String[]): void

```
public class TestGUIEvents extends JFrame {
  int clickCount;
  public TestGUIEvents() {
    clickCount = 0;     setTitle("Click-Count: " + clickCount);
    JButton button = new JButton("Click me!");
    button.addActionListener(new ButtonListener());
    add(button);
    pack();
  }
  public static void main(String[] args) {
    new TestGUIEvents().setVisible(true);
  }
  protected class ButtonListener implements ActionListener {
    public void actionPerformed(ActionEvent arg0) {
      clickCount++;
      setTitle("Click-Count: " + clickCount);
    }
  }
}
```

BPSE pattern map again



Conclusions

- Some simple patterns can be directly transformed into code. However, do not expect all patterns to be complete solutions!
- Pattern make use of fundamental concepts of Object Oriented programming: Class, Object, Method, Message passing, Inheritance, Encapsulation, Abstraction and Polymorphism.
- Well-designed object-oriented systems have multiple patterns embedded in them.