Design Patterns: Software Design Libraries

Andreas Roth, Richard Bubel

November 19, 2004
Design patterns are software design libraries.

Design patterns offer solutions for common software design problems on an \textit{abstract} level.

Design patterns are defined using a strict description scheme.

\textbf{Motivation} Where does the need for this pattern come from?

\textbf{Structure} What is the concrete (class/static) structure of the pattern?

\textbf{Applicability} Under which circumstances is it applicable? Which are the forces it obeys?

\textbf{Consequences} What are the effects when using this pattern?
Design patterns are software design libraries.

Design patterns offer solutions for common software design problems on an *abstract* level.

Design patterns are defined using a strict description scheme.

**Motivation**  Where does the need for this pattern come from?

**Structure**  What is the concrete (class/static) structure of the pattern?

**Applicability**  Under which circumstances is it applicable? Which are the forces it obeys?

**Consequences**  What are the effects when using this pattern?
Design patterns are software design libraries.

Design patterns offer solutions for common software design problems on an *abstract* level.

Design patterns are defined using a strict description scheme.

**Motivation**  Where does the need for this pattern come from?

**Structure**  What is the concrete (class/static) structure of the pattern?

**Applicability**  Under which circumstances is it applicable? Which are the forces it obeys?

**Consequences**  What are the effects when using this pattern?
Design patterns are software design libraries.

Design patterns offer solutions for common software design problems on an *abstract* level.

Design patterns are defined using a strict description scheme.

**Motivation** Where does the need for this pattern come from?

**Structure** What is the concrete (class/static) structure of the pattern?

**Applicability** Under which circumstances is it applicable? Which are the forces it obeys?

**Consequences** What are the effects when using this pattern?
Use of design patterns supports ...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
- documentation of software systems

Design Patterns: Software Design Libraries
November 19, 2004
Use of design patterns supports...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
- documentation of software systems

Design Patterns: Software Design Libraries
November 19, 2004
Use of design patterns supports...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
  - same vocabulary, focus on important details
- documentation of software systems
  - improved maintainability
  - design decisions are made (more) explicit
Use of design patterns supports...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
  - same vocabulary, focus on important details
- documentation of software systems
  - improved maintainability
  - design decisions are made (more) explicit
Use of design patterns supports ...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
  - same vocabulary, focus on important details
- documentation of software systems
  - improved maintainability
  - design decisions are made (more) explicit
Use of design patterns supports . . .

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
  - same vocabulary, focus on important details
- documentation of software systems
  - improved maintainability
  - design decisions are made (more) explicit
Use of design patterns supports ...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
  - same vocabulary, focus on important details
- documentation of software systems
  - improved maintainability
  - design decisions are made (more) explicit
Use of design patterns supports...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
  - same vocabulary, focus on important details
- documentation of software systems
  - improved maintainability
  - design decisions are made (more) explicit
Use of design patterns supports...

- knowledge transfer between software designers
  - better education, orientation for novices
- reuse of already found solutions
  - prevents from reinventing the wheel
- communication between developers
  - same vocabulary, focus on important details
- documentation of software systems
  - improved maintainability
  - design decisions are made (more) explicit
Classification of patterns

Patterns can be classified in

- structural
- behavioural
- creational

Here we will have a closer look on

- the observer and
- the commando

pattern.
Classification of patterns

Patterns can be classified in
- structural
- behavioural
- creational

Here we will have a closer look on
- the observer and
- the commando pattern.
Keep “Model, View and Control” (MVC) separate.

Advantages:
- improves reusability
- adding views possible without changing current code
- response can be dynamically chosen
- easier to guarantee consistence of model and views

Organizer Database
Keep “Model, View and Control” (MVC) separate.

Advantages

- Improves reusability
- Adding views possible without changing current code
- Response can be dynamically chosen
- Easier to guarantee consistency of model and views
Keep “Model, View and Control” (MVC) separate.

Advantages

- Improves reusability
- Adding views possible without changing current code
- Response can be dynamically chosen
- Easier to guarantee consistence of model and views
Keep “Model, View and Control” (MVC) separate.

Advantages

- improves reusability
- adding views possible without changing current code
- response can be dynamically chosen
- easier to guarantee consistence of model and views
Keep “Model, View and Control” (MVC) separate.

Advantages

- improves reusability
- adding views possible without changing current code
- response can be dynamically chosen
- easier to guarantee consistence of model and views
Keep “Model, View and Control” (MVC) separate.

Organizer Database

Advantages
- improves reusability
- adding views possible without changing current code
- response can be dynamically chosen
- easier to guarantee consistence of model and views
Keep “Model, View and Control” (MVC) separate.

Advantages
- improves reusability
- adding views possible without changing current code
- response can be dynamically chosen
- easier to guarantee consistence of model and views
The Observer Pattern

**Purpose**
Defines a 1-to-n dependance relationship between objects, so that the state change of one object causes a notification of the dependant objects.

**Applicability**
- If the state change of one object, requires to update several other objects.
- If an object has to inform several other unknown objects.

**Consequence**
- Independant reuse of subject and observer.
- Observers can be added without changing the observer or subject.
The Observer Pattern

Purpose
Defines a 1-to-n dependance relationship between objects, so that the state change of one object causes a notification of the dependant objects.

Applicability
- If the state change of one object, requires to update several other objects.
- If an object has to inform several other unknown objects.

Consequence
- Independant reuse of subject and observer.
- Observers can be added without changing the observer or subject.
The Observer Pattern

Purpose
Defines a 1-to-n dependance relationship between objects, so that the state change of one object causes a notification of the dependant objects.

Applicability
- If the state change of one object, requires to update several other objects.
- If an object has to inform several other unknown objects.

Consequence
- Independant reuse of subject and observer.
- Observers can be added without changing the observer or subject.
Class structure of the observer pattern

Subject
- register(Observer)
- remove(Observer)
- notify()

ConcreteSubject

Observer
- update()

ConcreteObserver

ConcreteObservern

pull the observer has to ask for the new subject state (getState)
push change information is handed over to the observer when notifying
Class structure of the observer pattern

Subject
- register(Observer)
- remove(Observer)
- notify()

ConcreteSubject

Observer
- update()

ConcreteObserver1

ConcreteObservern

pull  the observer has to ask for the new subject state (getState)
push  change information is handed over to the observer when notifying
Class structure of the observer pattern

Subject
- `register(Observer)`
- `remove(Observer)`
- `notify()`

Observerv
- `update()`

ConcreteSubject

ConcreteObserver1

ConcreteObserver_\text{n}

pull  the observer has to ask for the new subject state (`getState`)

push  change information is handed over to the observer when notifying
Dynamic behaviour in a typical scenario

anActor \quad aConcreteSubject \quad aConcreteObserver1 \quad aConcreteObserver2

change() \quad \rightarrow \quad notify() \quad \rightarrow \quad update() \quad \rightarrow \quad getState()

update() \quad \rightarrow \quad getState()
Dynamic behaviour in a typical scenario

anActor → aConcreteSubject

change() → notify() → update() → getState()

aConcreteObserver1

aConcreteObserver2

update() → getState()
Dynamic behaviour in a typical scenario

- anActor
- aConcreteSubject
- aConcreteObserver1
- aConcreteObserver2

- change()
- notify()
- update()
- getState()
Dynamic behaviour in a typical scenario

anActor       aConcreteSubject       aConcreteObserver1       aConcreteObserver2

change()       | notify()       | update()       | update()

getstate()     | getstate()     |                |
Dynamic behaviour in a typical scenario

anActor

aConcreteSubject

aConcreteObserver1

aConcreteObserver2

change() → notify() → update() → getState() → update() → getState()
Dynamic behaviour in a typical scenario

anActor  aConcreteSubject  aConcreteObserver1  aConcreteObserver2

change()  

notify()  update()  

update()  

getState()  getState()
Dynamic behaviour in a typical scenario

anActor -> aConcreteSubject
change() -> notify()
notify() -> update()
update() -> getState()

aConcreteObserver1

aConcreteObserver2
update() -> getState()
Class structure of the observer pattern

**Subject**
- `register(Observer)`
- `remove(Observer)`
- `notify()`

**Organizer Database**

**Observer**
- `update()`

**OrganizerDB**
- `addEntry(Date date, String descr)`
- `getEntries(Date from, Date to)`

**DayView**
- `update()`

**MonthView**
- `update()`

### Observer Methods
- `observer`
Class structure of the observer pattern

Subject
- `register(Observer)`
- `remove(Observer)`
- `notify()`

Observer
- `update()`

Organizer Database
- `addEntry(Date date, String descr)`
- `getEntries(Date from, Date to)`

DayView
- `update()`

MonthView
- `update()`

November
<table>
<thead>
<tr>
<th>Mo</th>
<th>Tu</th>
<th>Wed</th>
<th>Th</th>
<th>Fr</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mo</td>
<td>Tu</td>
<td>Wed</td>
<td>Th</td>
<td>Fr</td>
</tr>
<tr>
<td>Mo</td>
<td>Tu</td>
<td>Wed</td>
<td>Th</td>
<td>Fr</td>
</tr>
<tr>
<td>Mo</td>
<td>Tu</td>
<td>Wed</td>
<td>Th</td>
<td>Fr</td>
</tr>
</tbody>
</table>

Friday, 19th November
8
9
...
...
...
17
Class structure of the observer pattern

Subject
- register(Observer)
- remove(Observer)
- notify()

Observer
- update()

Organizer Database
- addEntry(Date date, String descr)
- getEntries(Date from, Date to)

DayView
- update()

MonthView
- update()
Class structure of the observer pattern

Subject
- register(Observer)
- remove(Observer)
- notify()

* Observer
- update()

Organizer Database
- addEntry(Date date, String descr)
- getEntries(Date from, Date to)

OrganizerDB

DayView
- update()

MonthView
- update()
Updating views after adding an entry

```java
Karl
officeCal:OrganizerDB
day1911:DayView
november:MonthView
```

```java
addEntry(newDate, "lab course")
notify()
update()
getEntries(day1911, day1911)
```

```java
getEntries(day0111, day3011)
update()
```
Updating views after adding an entry

- Karl
- officeCal:OrganizerDB
- day1911:DayView
- november:MonthView

```java
addEntry(newDate, "lab course")
notify()
update()
getEntries(day1911, day1911)
update()
getEntries(day0111, day3011)
```
Updating views after adding an entry

Karl officeCal:OrganizerDB day1911:DayView november:MonthView

addEntry(newDate, "lab course")

notify()

update()

getEntries(day1911, day1911)

update()

getEntries(day0111, day3011)
Updating views after adding an entry

Karl | officeCal:OrganizerDB | day1911:DayView | november:MonthView

addEntry(newDate, "lab course")

notify()

getEntries(day1911, day1911)

update()

gEntries(day0111, day3011)

update()
Updating views after adding an entry

Karl

officeCal:OrganizerDB

day1911:DayView

november:MonthView

addEntry(newDate, "lab course")

notify()

update()

getEntries(day1911, day1911)

update()

getEntries(day0111, day3011)
Updating views after adding an entry

```
addEntry(newDate, "lab course")
```

```
notify()
```

```
update() ->
getEntries(day1911, day1911)
```

```
update() ->
getEntries(day0111, day3011)
```
Updating views after adding an entry

- **Karl**
- **officeCal:OrganizerDB**
- **day1911:DayView**
- **november:MonthView**

```java
addEntry(newDate, "lab course")
notify()
update()
getEntries(day1911, day1911)
```

```java
update()
getEntries(day0111, day3011)
```

Design Patterns: Software Design Libraries
November 19, 2004 10 / 17
The Commando Pattern

**Purpose** Encapsulate a command as an object. Allows to keep track of actions (transactions).

**Applicability** The pattern is applicable e.g. for
- parametrising clients,
- logging commands,
- realising undo functionality or
- caching commands in a queue in order to execute them to a certain time in future.
The Commando Pattern

**Purpose**
Encapsulate a command as an object. Allows to keep track of actions (transactions).

**Applicability**
The pattern is applicable e.g. for
- parametrising clients,
- logging commands,
- realising undo functionality or
- caching commands in a queue in order to execute them to a certain time in future.
The Commando Pattern

**Purpose** Encapsulate a command as an object. Allows to keep track of actions (transactions).

**Applicability** The pattern is applicable e.g. for

- parametrising clients,
- logging commands,
- realising undo functionality or
- caching commands in a queue in order to execute them to a certain time in future
The Commando Pattern

**Purpose** Encapsulate a command as an object. Allows to keep track of actions (transactions).

**Applicability** The pattern is applicable e.g. for
- parametrising clients,
- logging commands,
- realising undo functionality or
- caching commands in a queue in order to execute them to a certain time in future.
The Commando Pattern

**Purpose** Encapsulate a command as an object. Allows to keep track of actions (transactions).

**Applicability** The pattern is applicable e.g. for
- parametrising clients,
- logging commands,
- realising undo functionality or
- caching commands in a queue in order to execute them to a certain time in future.
The Commando Pattern

**Purpose**  Encapsulate a command as an object. Allows to keep track of actions (transactions).

**Applicability**  The pattern is applicable e.g. for

- parametrising clients,
- logging commands,
- realising undo functionality or
- caching commands in a queue in order to execute them to a certain time in future
Call of `execute()` on `Command` invokes the receiver's `action()` operation.

Not necessary each `Command` subclass invokes same `action()` operation.

How could the pattern be altered to avoid encoding the receiver association as attribute?
The Commando Pattern

- Call of `execute()` on Command invokes the receiver's `action()` operation.
- Not necessary each Command subclass invokes same `action()` operation.
- How could the pattern be altered to avoid encoding the receiver association as attribute?
The Commando Pattern

- Call of `execute()` on Command invokes the receiver's `action()` operation.
- Not necessary each `Command` subclass invokes same `action()` operation.
- How could the pattern be altered to avoid encoding the receiver association as attribute?
Typical Commando Scenario

:Caller

ConcreteCommand

:Receiver

execute()

action()
Typical Commando Scenario

:Caller \rightarrow :ConcreteCommand \rightarrow :Receiver

execute() \rightarrow action()
Typical Commando Scenario

:Caller

:ConcreteCommand

:Receiver

execute()

action()
The organizer performs timer triggered actions

One feature of the organizer in the former examples is to execute actions, when a certain date (time) has been reached.

For example:

- beep 5 minutes before a meeting
- send to all participants of a meeting an invitation e-mail a week before it is scheduled
- record a TV movie
The organizer performs timer triggered actions

One feature of the organizer in the former examples is to execute actions, when a certain date (time) has been reached.

For example:

- beep 5 minutes before a meeting
- send to all participants of a meeting an invitation e-mail a week before it is scheduled
- record a TV movie
The organizer performs timer triggered actions

One feature of the organizer in the former examples is to execute actions, when a certain date (time) has been reached.

For example:
- beep 5 minutes before a meeting
- send to all participants of a meeting an invitation e-mail a week before it is scheduled
- record a TV movie
The organizers commando framework

Organizer → Timer

Command
execute()

Concrete Commands
- Beep
  execute()
- SendMail
  execute()
- RecordTV
  execute()

Receivers
- Speaker
  beep()
- MailServer
  sendMail(...)
- VCR
  play()
  record()
“Design Patterns: Elements of Reusable Object-Oriented Software”, by E. Gamma, R. Helm, R. Johnson and J. Vlissides, Addison-Wesley (1995)


Design Patterns ...

- document design solutions on an abstract level
- prevent from reinventing the wheel
- support communication between developers
- make expert knowledge usable for beginners

but:

They don’t make your design decisions.
Libraries For Designing System

- document design solutions on an abstract level
- prevent from reinventing the wheel
- support communication between developers
- make expert knowledge usable for beginners

but:

They don’t make your design decisions.
Libraries For Designing System

- document design solutions on an abstract level
- prevent from reinventing the wheel
- support communication between developers
- make expert knowledge usable for beginners

but:

They don’t make your design decisions.
Libraries For Designing System

- document design solutions on an abstract level
- prevent from reinventing the wheel
- support communication between developers
- make expert knowledge usable for beginners

but:

They don’t make your design decisions.
Libraries For Designing System

- document design solutions on an abstract level
- prevent from reinventing the wheel
- support communication between developers
- make expert knowledge usable for beginners

but:

They *don’t* make your design decisions.