Object-Oriented Design
- Progress from Requirements Analysis to Design

Zhiming Liu
zhimingliu88@swu.edu.cn
http://computer.swu.edu.cn
Centre for Research and Innovation in Software Engineering (RISE)
School of Computer and Information Science
Southwest University
Chongqing, China

The Design Phase

Study

- The nature of the design phase
- The distinct features of OO design
- UML interaction diagrams for modelling design
- Design patterns for assigning responsibility to objects
- Design class diagrams

Artefacts: Interaction diagrams and Design Class Diagrams

Recall Object-Oriented Analysis

- **Artefacts of requirements analysis**

Recall Object-Oriented Analysis

- **Artefacts of requirements analysis**
  1. **Use cases**: extended descriptions and use case diagrams, where from and what for?

Recall Object-Oriented Analysis

- **Artefacts of requirements analysis**
  1. **Use cases**: extended descriptions and use case diagrams, where from and what for?
  2. **Conceptual class diagrams:** what are they and where from and what for?

Recall Object-Oriented Analysis

- **Artefacts of requirements analysis**
  1. **Use cases**: extended descriptions and use case diagrams, where from and what for?
  2. **Conceptual class diagrams:** what are they and where from and what for?
  3. **Use case sequence diagrams:** what are they and where from and what for?

Recall Object-Oriented Analysis

- **Artefacts of requirements analysis**
  1. **Use cases**: extended descriptions and use case diagrams, where from and what for?
  2. **Conceptual class diagrams:** what are they and where from and what for?
  3. **Use case sequence diagrams:** what are they and where from and what for?
  4. **Use case operations and their contracts**: what are they and what for?

Recall Object-Oriented Analysis

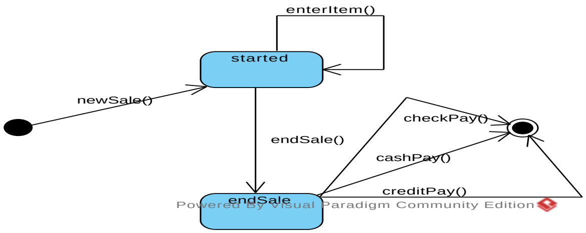
- **Artefacts of requirements analysis**
  1. **Use cases**: extended descriptions and use case diagrams, where from and what for?
  2. **Conceptual class diagrams:** what are they and where from and what for?
  3. **Use case sequence diagrams:** what are they and where from and what for?
  4. **Use case operations and their contracts**: what are they and what for?
- Class diagrams, use case sequence diagrams and contracts of use case operations together model the behaviour of the systems

Use Case Behaviour

- Each execution of a use case operation changes the system from one state to another

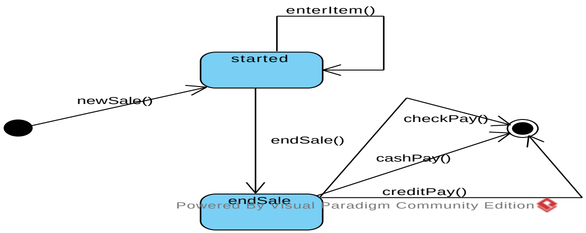
Use Case Behaviour

- Each execution of a use case operation changes the system from one state to another
- A use case sequence diagram has a state diagram representing its dynamic behaviour



Use Case Behaviour

- Each execution of a use case operation changes the system from one state to another
- A use case sequence diagram has a state diagram representing its dynamic behaviour



- Functionality of an operation in terms of pre- and post-conditions can be inserted.

Object-Oriented Design in RUP

- The behaviours of the use cases are external and global
- They correspond to the method invocations in the “Main Method”, or GUI methods

Object-Oriented Design in RUP

- The behaviours of the use cases are external and global
- They correspond to the method invocations in the "Main Method", or GUI methods
- E.G. *startSale*(), *enterItem*(), *endSale*(), *cashPay*(), *creditPay*() and *checkPay*(), in use case *Process Sale*

Object-Oriented Design in RUP

- The behaviours of the use cases are external and global
- They correspond to the method invocations in the “Main Method”, or GUI methods
- E.G. *startSale*(), *enterItem*(), *endSale*(), *cashPay*(), *creditPay*() and *checkPay*(), in use case *Process Sale*
- OO Design is to decide how to realise the contracts each use case operation through interactions among objects of the classes in the conceptual class diagram(s)

Object-Oriented Design in RUP

- The behaviours of the use cases are external and global
- They correspond to the method invocations in the “Main Method”, or GUI methods
- E.G. *startSale*(), *enterItem*(), *endSale*(), *cashPay*(), *creditPay*() and *checkPay*(), in use case *Process Sale*
- OO Design is to decide how to realise the contracts each use case operation through interactions among objects of the classes in the conceptual class diagram(s)
- The interactions between objects realising a use case operation is represented by a UML interaction diagram

Object-Oriented Design in RUP

- The behaviours of the use cases are external and global
- They correspond to the method invocations in the "Main Method", or GUI methods
- E.G. *startSale*(), *enterItem*(), *endSale*(), *cashPay*(), *creditPay*() and *checkPay*(), in use case *Process Sale*
- OO Design is to decide how to realise the contracts each use case operation through interactions among objects of the classes in the conceptual class diagram(s)
- The interactions between objects realising a use case operation is represented by a UML interaction diagram
- Code can then be constructed from these interactions diagrams and the class diagram

Model Object Interactions in UML

- the UML mode for design mainly includes a set of interaction diagrams and a design class diagram
  1. **collaboration diagrams**
  2. **object sequence diagrams**

  Either can be used to express similar or identical messages interactions.
- The object interaction diagrams are the most important for design, and require the greatest degree creative effort

Object Sequence Diagrams vs Use Case Sequence Diagrams

- A use case sequence diagram models the interaction between the system/component and its environment (actors), including other use case sequence diagrams

Object Sequence Diagrams vs Use Case Sequence Diagrams

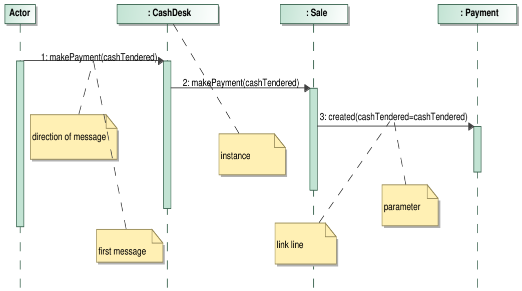
- A use case sequence diagram models the interaction between the system/component and its environment (actors), including other use case sequence diagrams
- An object sequence diagram models the interactions between objects inside the system (component)
- Use case sequence diagrams identify provided methods

Object Sequence Diagrams vs Use Case Sequence Diagrams

- A use case sequence diagram models the interaction between the system/component and its environment (actors), including other use case sequence diagrams
- An object sequence diagram models the interactions between objects inside the system (component)
- Use case sequence diagrams identify provided methods
- An object sequence diagram decomposes a use case operation into interactions among objects, and defines methods of classes

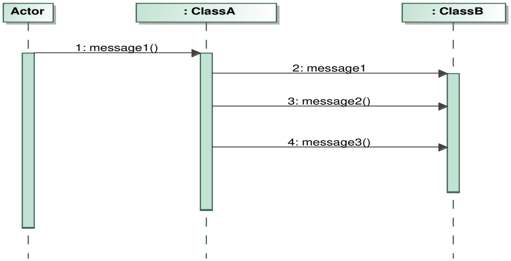
Example: *cashPay*()

1. Cashier sends request *cashPay(cashTendered*) to *CashDesk*
2. *CashDesk* carries it out by sending *cashPay*(*cashTendered*) to *Sale*.
3. *Sale* carries out the task by creating a *Payment*.


Actor
: CashDesk
: Sale
: Payment
1: makePayment(cashTendered)
2: makePayment(cashTendered)
3: created(cashTendered=cashTendered)
direction of message
instance
parameter
first message
link line

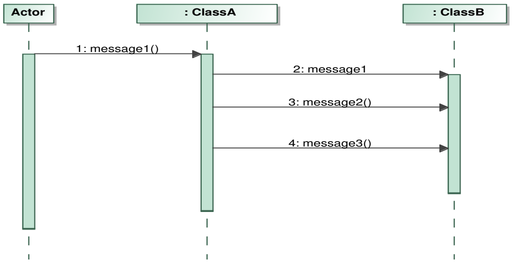
Notation of Object Sequence Diagrams in UML

- Object sequence diagram



Notation of Object Sequence Diagrams in UML

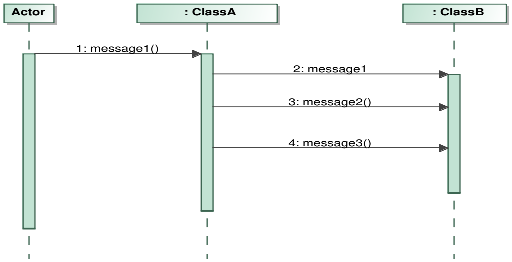
- Object sequence diagram



- instances of classes in boxes

Notation of Object Sequence Diagrams in UML

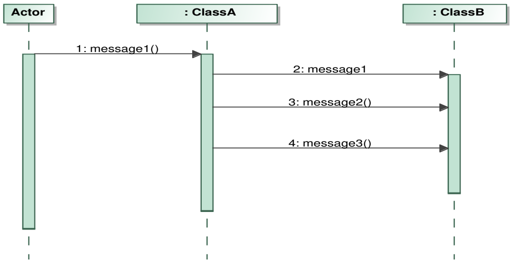
- Object sequence diagram



- instances of classes in boxes
- a directed link between two objects represents a instance of an association and visibility

Notation of Object Sequence Diagrams in UML

- Object sequence diagram



- instances of classes in boxes
- a directed link between two objects represents a instance of an association and visibility
- a message represents an invocation of a method of the target object (server) from the source object (client)

Messages, Links and Associations

1. a message can be passed between two objects only when the two objects are linked

Messages, Links and Associations

1. a message can be passed between two objects only when the two objects are linked
2. two objects can only be linked by an association of the classes of the objects

Messages, Links and Associations

1. a message can be passed between two objects only when the two objects are linked
2. two objects can only be linked by an association of the classes of the objects
3. any method of the server can be invoked by the client when there is link between them

Messages with Return Value

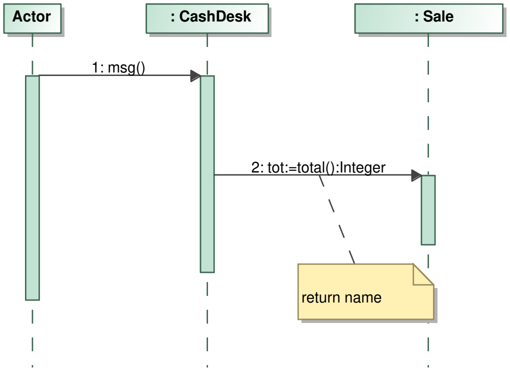
Standard syntax for messages with return value:

return := message(p : pType) : returnType

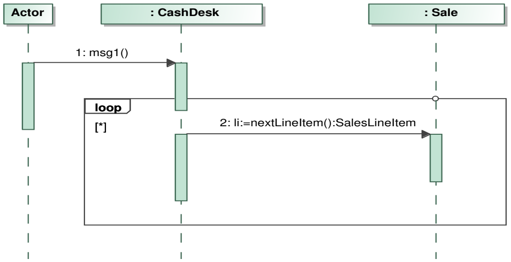
Messages with Return Value

Standard syntax for messages with return value:

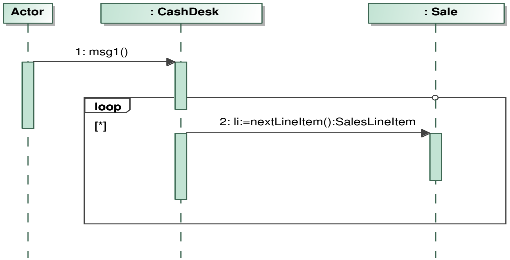
return := message(p : pType) : returnType

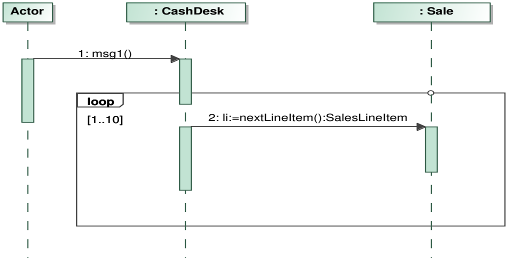


Loops in Object Sequence Diagrams



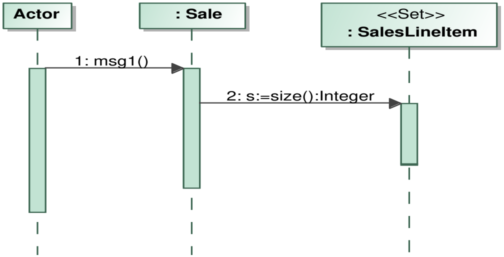
Loops in Object Sequence Diagrams





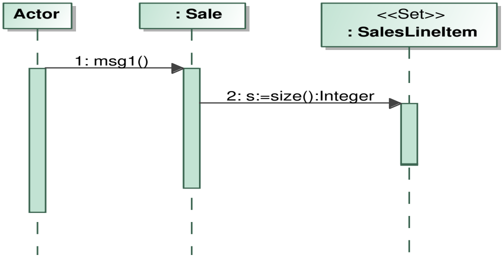
Messages to Multiobject

- In UML diagrams the stereo type $<<$ *set* $>>$ is used.



Messages to Multiobject

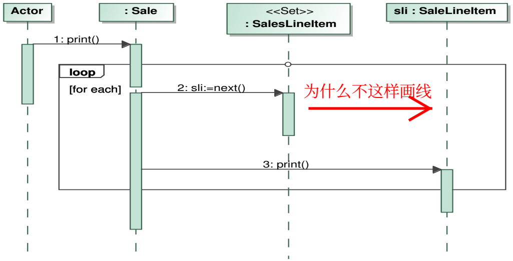
- In UML diagrams the stereo type $<<$ *set* $>>$ is used.



- A message sent to a multiobject is sent to the single object, not broadcast to each element in the multiobject

Method on Each Object in Multiobject

1. an iteration to the multiobject to extract links to the individual objects
2. then a message sent to each individual object using the (temporary) link

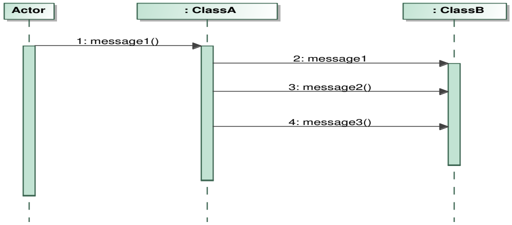


Model Decomposition of Method

Interaction diagrams illustrate the decomposition in terms of interactions between objects

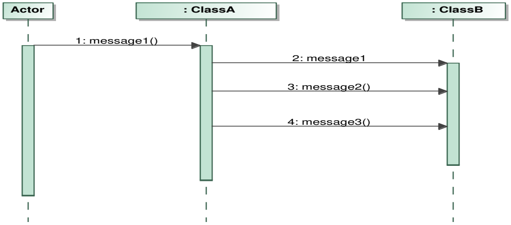
Model Decomposition of Method

Interaction diagrams illustrate the decomposition in terms of interactions between objects



Model Decomposition of Method

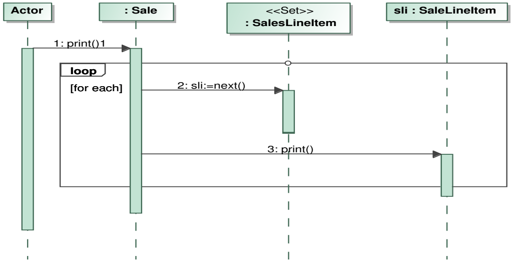
Interaction diagrams illustrate the decomposition in terms of interactions between objects



- *msg*1() is decomposed into three methods of ClassB:

ClassA:: *msg*1(){
ClassB.*message*1();
ClassB.*message*2();
ClassB.*message*3()}

Another Example



print() is decomposed into a loop of printing each SaleLineItem of the Sale:

```
Sale:: print(){
for each SaleLineItem sli do sli.print()
}
```

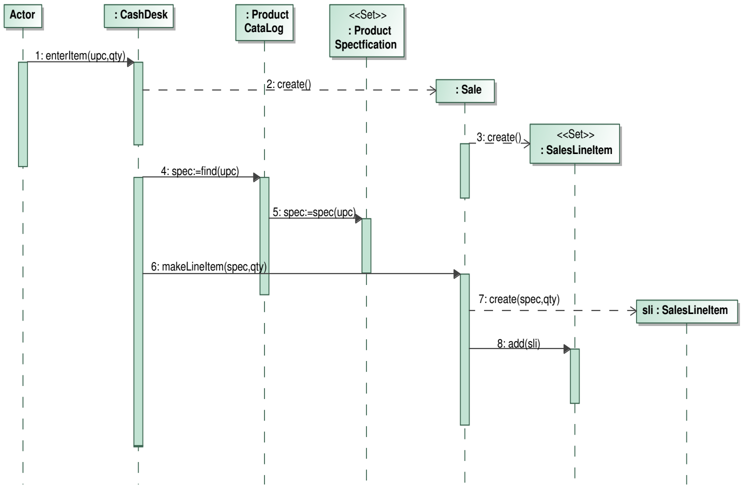
In Java

A multiobject can implemented as a vector (i.e a container object)

```
class Sale {
private SaleLineItem sli;
private Vector lineItems = new Vector();
public void print() {
Enumeration e = lineItems.elements();
while ( e.hasMoreElements() )
sli = ((SaleLineItem) e.nextElement());
sli.print() }
}
```

课堂练习：如果要直接从lineitems画线过去，该怎么写程序？（5分钟）

More Complicated Decomposition



Programming Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
```

Programming Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
if (isNewSale() ) {
```

Programming Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
if (isNewSale() ) {
sale = new Sale()
};
```

Programming Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
if (isNewSale() ) {
sale = new Sale()
};
ProductSpecification spec =
productCatalog.specification (upc);
```

Programming Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
if (isNewSale() ) {
sale = new Sale()
};
ProductSpecification spec =
productCatalog.specification (upc);
sale.PmakeLineItem(spect,qty);
}
```

Programming Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
if (isNewSale() ) {
sale = new Sale()
};
ProductSpecification spec =
productCatalog.specification (upc);
sale.PmakeLineItem(spect,qty);
}
```

- *makeLineItem(spect,qty)* method of Sale is further decomposed

Programming Textual Form

```
CashDesk:: enterItem(int upc, int qty) {
if (isNewSale() ) {
sale = new Sale()
};
ProductSpecification spec =
productCatalog.specification (upc);
sale.PmakeLineItem(spect,qty);
}
```

- *makeLineItem(spect,qty)* method of Sale is further decomposed

```
Class Sale {...
private Vector lineItems = new Vector();
public void makeLineItem
(ProductSpecification spec, int qty)
{ lineItems.addElement(new SaleLineItem(spect,qty))
}
......
}
```

Models Needed for Design

Models Needed for Design

- class model
  - data/objects needed for a task are held by different objects are represented there
  - associations between classes represents knowledge of one object about another
  - objects of the classes participate in interactions illustrated in the interaction diagrams.
- use case sequence diagrams for identification of the use case operations

Models Needed for Design

- class model
  - data/objects needed for a task are held by different objects are represented there
  - associations between classes represents knowledge of one object about another
  - objects of the classes participate in interactions illustrated in the interaction diagrams.
- use case sequence diagrams for identification of the use case operations
- contracts of interface methods: identifies subtasks in post-conditions

Patterns for Assigning Responsibilities (GRASP)

A responsibility (or functionality) is a contract or obligation of an object

1. Doing responsibilities: actions an object can perform:
    - doing something (action) itself
    - initiating an (action) in other objects
    - controlling and coordinating activities in other objects
2. Knowing responsibilities: knowledge an object maintains:
    - know about private encapsulated data
    - know about linked objects
    - know about things it can derive or calculate

relation between doing and knowing?

Examples

1. We may assign the responsibility for print a *Sale* to the *Sale* instance – “a *Sale* is responsible for printing itself” (doing)
2. We may assign the responsibility of knowing the date of a *Sale* to the *Sale* instance itself–“a *Sale* is responsible for knowing its date”(knowing).

Examples

1. We may assign the responsibility for print a *Sale* to the *Sale* instance – “a *Sale* is responsible for printing itself” (doing)
2. We may assign the responsibility of knowing the date of a *Sale* to the *Sale* instance itself–“a *Sale* is responsible for knowing its date”(knowing).

Knowing responsibilities are inferable from the conceptual model

- An object “knows” its own attributes, e.g. a student knows his name, ages, address, etc; a sale knows its date, time, etc.
- An object ”knows” the objects it has links with, e.g. a student knows the modules he takes; a sale knows its lineItems,

Methods of Objects

- Responsibilities of an object are implemented as *methods*
- A method may “act” itself or collaborate with other methods and objects

Methods of Objects

- Responsibilities of an object are implemented as *methods*
- A method may “act” itself or collaborate with other methods and objects
- E.g. *Sale* may define a method *print*() that performing printing the *Sale* instance

Methods of Objects

- Responsibilities of an object are implemented as *methods*
- A method may "act" itself or collaborate with other methods and objects
- E.g. *Sale* may define a method *print*() that performing printing the *Sale* instance
- Printing a sale is to print all the lineItems of the sale

Methods of Objects

- Responsibilities of an object are implemented as *methods*
- A method may “act” itself or collaborate with other methods and objects
- E.g. *Sale* may define a method *print*() that performing printing the *Sale* instance
- Printing a sale is to print all the lineItems of the ale
- To do so, *Sale* may have to send a message to *SalesLineItem* objects asking them to print themselves.

Methods of Objects

- Responsibilities of an object are implemented as *methods*
- A method may "act" itself or collaborate with other methods and objects
- E.g. *Sale* may define a method *print*() that performing printing the *Sale* instance
- Printing a sale is to print all the lineItems of the ale
- To do so, *Sale* may have to send a message to *SalesLineItem* objects asking them to print themselves.

How to assign a responsibility to an object?

Responsibilities Assignment

1. Start with the responsibilities identified from the use cases and contracts

Responsibilities Assignment

1. Start with the responsibilities identified from the use cases and contracts
2. Assign these responsibilities to objects

Responsibilities Assignment

1. Start with the responsibilities identified from the use cases and contracts
2. Assign these responsibilities to objects
3. Decide what the objects needs to do to fulfil these responsibilities in order to identify further responsibilities

Responsibilities Assignment

1. Start with the responsibilities identified from the use cases and contracts
2. Assign these responsibilities to objects
3. Decide what the objects needs to do to fulfil these responsibilities in order to identify further responsibilities
4. Assign further identified responsibilities to objects

Responsibilities Assignment

1. Start with the responsibilities identified from the use cases and contracts
2. Assign these responsibilities to objects
3. Decide what the objects needs to do to fulfil these responsibilities in order to identify further responsibilities
4. Assign further identified responsibilities to objects
5. Repeat these steps until the identified responsibilities are fulfilled and a collaboration diagram is completed

Example

- The contract of *cashPay*() in use case Process Sale, identify the responsibility to print a *Sale*

Example

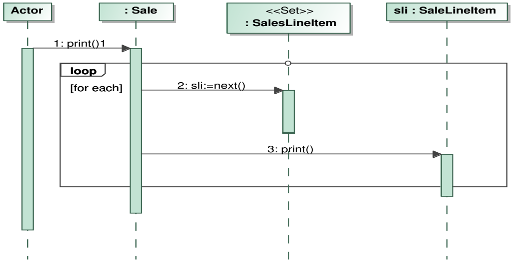
- The contract of *cashPay*() in use case Process Sale, identify the responsibility to print a *Sale*
- Assign the responsibility for print a *Sale* to the *Sale* instance – the *Sale* is responsible for printing itself (doing).

Example

- The contract of *cashPay*() in use case Process Sale, identify the responsibility to print a *Sale*
- Assign the responsibility for print a *Sale* to the *Sale* instance – the *Sale* is responsible for printing itself (doing).
- This responsibility is invoked with a *print* message to the *Sale*

Example

- The contract of *cashPay*() in use case Process Sale, identify the responsibility to print a *Sale*
- Assign the responsibility for print a *Sale* to the *Sale* instance – the *Sale* is responsible for printing itself (doing).
- This responsibility is invoked with a *print* message to the *Sale*
- Fulfilment of this responsibility requires collaboration with *SalesLineItem* asking them to print



Patterns

- Patterns: general principles and idiomatic solutions that guide the creation of software
- Each of these principles or solutions describes a problem(s) to be solved and a solution(s) to the problem(s)

Patterns

- Patterns: general principles and idiomatic solutions that guide the creation of software
- Each of these principles or solutions describes a problem(s) to be solved and a solution(s) to the problem(s)
- A pattern is a problem/solution pair that can be applied to new context, with advice on how to apply it in novel situations:

Patterns

- Patterns: general principles and idiomatic solutions that guide the creation of software
- Each of these principles or solutions describes a problem(s) to be solved and a solution(s) to the problem(s)
- A pattern is a problem/solution pair that can be applied to new context, with advice on how to apply it in novel situations:

  Pattern Name: name given to the pattern
  Solution: description of the solution of the problem
  Problem: description of the problem that the pattern solves

Patterns

- Patterns: general principles and idiomatic solutions that guide the creation of software
- Each of these principles or solutions describes a problem(s) to be solved and a solution(s) to the problem(s)
- A pattern is a problem/solution pair that can be applied to new context, with advice on how to apply it in novel situations:

Pattern Name:	name given to the pattern
Solution:	description of the solution of the problem
Problem:	description of the problem that the pattern solves

We study Expert, Creator, Low Coupling, High Cohesion, and Controller