Patterns and Testing

Lecture # 7

Department of Computer Science and Technology
University of Bedfordshire

Written by David Goodwin,
based on the lectures of Marc Conrad and Dayou Li
and on the book *Applying UML and Patterns (3rd ed.)*
by *C. Larman* (2005).

Modelling and Simulation, 2012

Outline

Patterns

Introduction

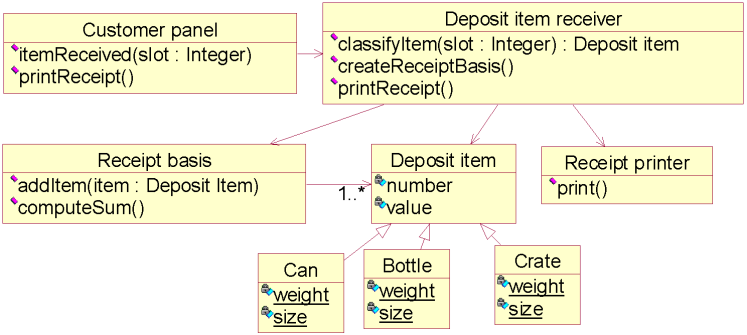
- An object-oriented system is composed of objects sending messages to other objects.
- The quality of the overall design depends on which object is doing what.
- That is, the quality depends on how we assign responsibilities to the objects.
- Problem: Define “good quality”.

There are two types of Responsibilities.

- **Knowing**
  - about private encapsulated data
  - about related objects
  - about things it can derive or calculate

- **Doing**
  - doing something itself
  - initiating action in other objects
  - controlling and coordinating activities in other objects

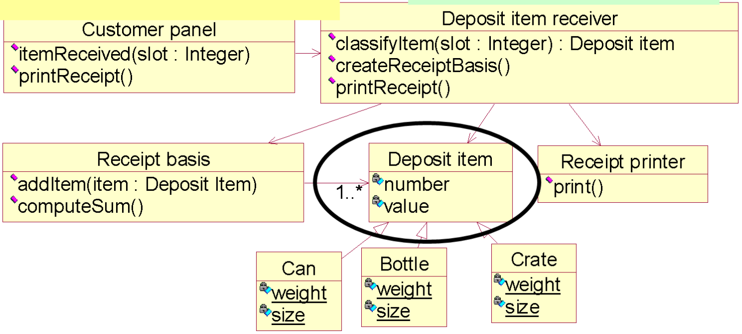
Example: the Recycling machine - Knowing and Doing



Example: the Recycling machine - Knowing and Doing

- Knowing
  - about private encapsulated data
  - about related objects
  - about things it can derive or calculate

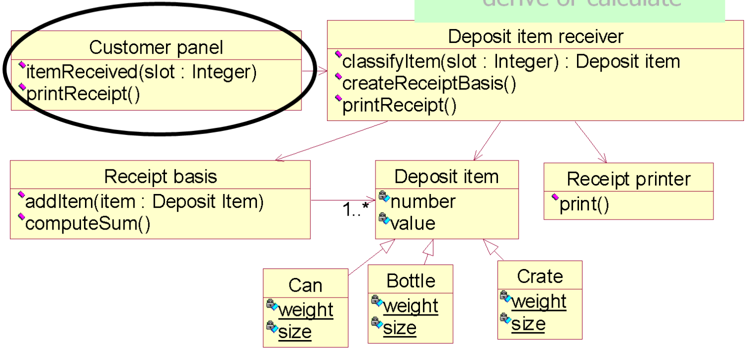
- *Deposit item* knows about private data as number and value



Example: the Recycling machine - Knowing and Doing

- *Customer panel* knows about the Deposit item receiver where it sends it messages to.

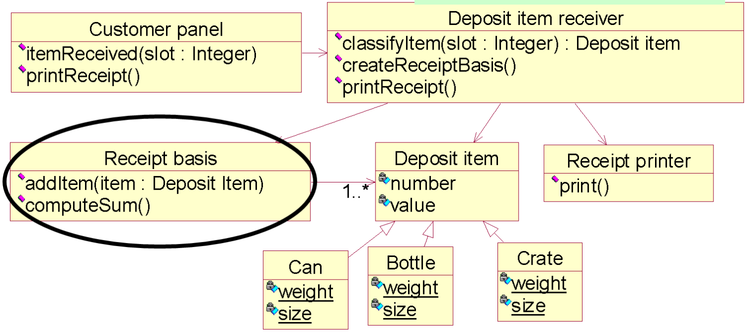
- Knowing
  - about private encapsulated data
  - about related objects
  - about things it can derive or calculate



Example: the Recycling machine - Knowing and Doing

- *Receipt basis* knows all the items which have been inserted into the recycling machine and is therefore able to compute the sum of their values.

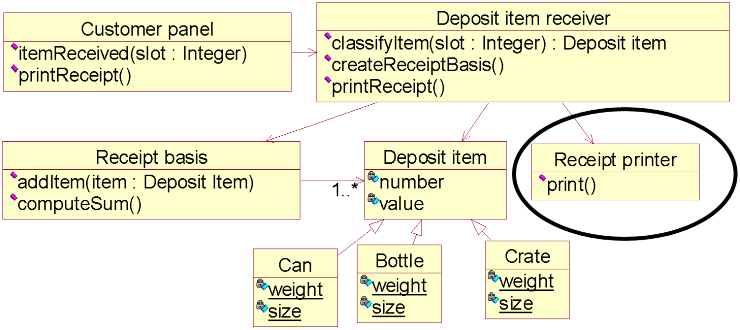
- Knowing
  - about private encapsulated data
  - about related objects
  - about things it can derive or calculate



Example: the Recycling machine - Knowing and Doing

- Doing
  - doing something itself
  - initiating action in other objects
  - controlling and coordinating activities in other objects

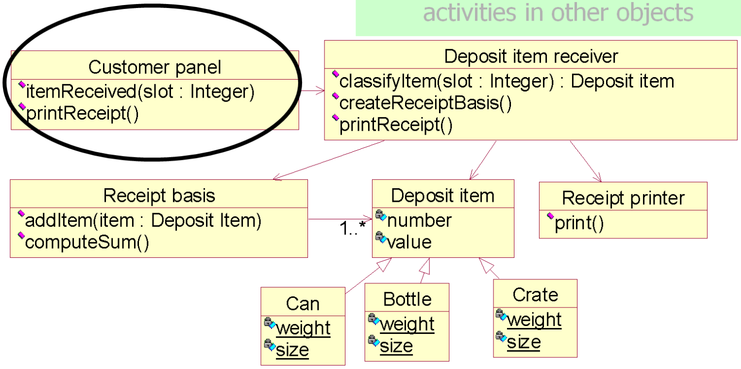
- The Receipt printer does print receipts.



Example: the Recycling machine - Knowing and Doing

- The Customer panel initiates the classification and receipt printing action in the Deposit item receiver.

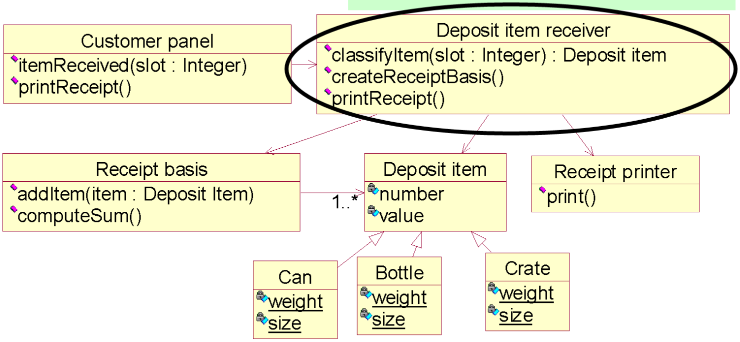
- Doing
  - doing something itself
  - initiating action in other objects
  - controlling and coordinating activities in other objects



Example: the Recycling machine - Knowing and Doing

- The Deposit item receiver controls the overall system behavior by assigning tasks to other objects (Receipt basis, Receipt printer).

- Doing
  - doing something itself
  - initiating action in other objects
  - controlling and coordinating activities in other objects



Good design/Bad design

- Consider the following alternative design of the recycling machine.
  - A class responsible for printing and holding the data of bottle and crate.
  - The can class is also responsible for customer input and computing the sum.
  - One more class doing all the rest of the tasks.
- **Is this a good design?**

Good design/Bad design

- Our feeling says that the previous example is not a good design.
- Is it possible to give this "feeling" a more solid, more objective, more traceable, and more comprehensible foundation?
- Answer: Yes, by using **patterns**.

GRASP - PATTERNS

- GRASP stands for *General Responsibility Assignment Software Patterns*.
- GRASP can be used when designing interaction (sequence) diagrams and class diagrams.
- GRASP try to formalize "common sense" in object oriented design.
- They do not usually contain "new" ideas. They try to codify existing knowledge and principles.

The GRASP patterns

- Creator
- Expert
- Low Coupling
- Controller
- High Cohesion

- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

GRASP - patterns for responsibilities CREATOR: Problem

- Who should be responsible for creating a new instance of some classes?
- The creation of objects is one of the most common activities in OO systems.
- It is useful to have a general principle for the assignment of responsibilities.
- Assigned well, the design can support low coupling, increased clarity, encapsulation, and resusability.

GRASP - patterns for responsibilities CREATOR: Solution

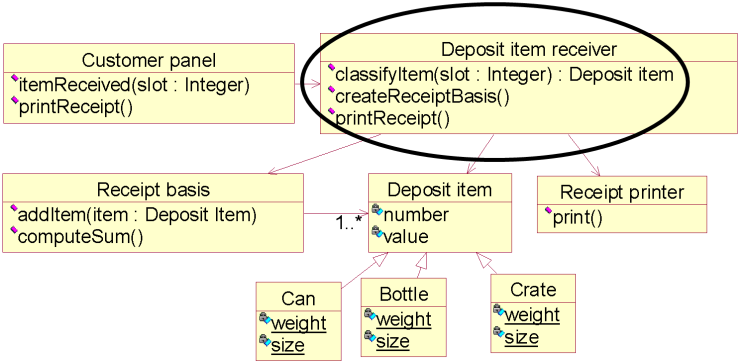
- Assign class B the responisbility to creat an instance of class A if one of these is true:
  - B aggregates A.
  - B contains A.
  - B records instances of A objects.
  - B closely uses A objects.
  - B has the initializing data that will be passed to A when it is created.
- B is a creator of A objects
- if more than one option applied, usually chose "aggregates or contains"

GRASP - patterns for responsibilities CREATOR: Discussion

- The creation of objects is one of the most common activities in an object-oriented system.
- This pattern is useful to find out who should be responsible for creating objects.
- The last point (B has initializing data of A) is actually an example of the Expert pattern (B is an expert with respect to creating A).
- In an Aggregation the lifetime of the part is usually the same as the lifetime of the whole. So the idea that the whole creates the part is straightforward.

GRASP - patterns for responsibilities CREATOR: Example

- The Deposit item receiver has all the necessary data for creating a Deposit item object.



GRASP - patterns for responsibilities EXPERT: Problem

- What is a general principle of assigning responsibilities to objects?
- When interactions between objects are defined, we chose assignment of responsibilities to software classes.
- Chosen well, systems tend to be easier to understand, maintain and extend.

GRASP - patterns for responsibilities EXPERT: Solution

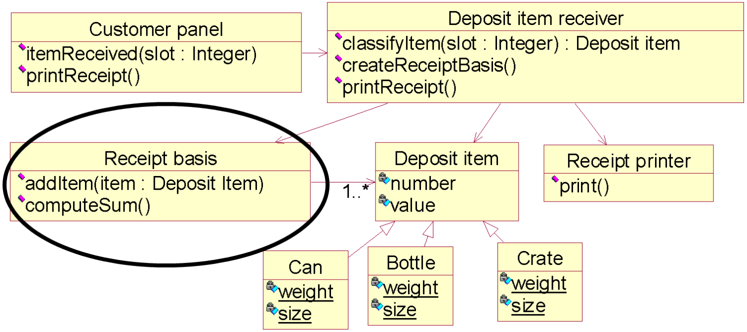
- Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility.

GRASP - patterns for responsibilities
EXPERT: Discussion

- Expert is the basic guiding principle in object-oriented design.
- Expert leads to designs where a software object does those operations which are normally done to the real-world thing it represents ("Do it Myself")
- Real-world example:
  - When going for medical treatment - which person would you ask for an appointment? The cleaner, the receptionist, or the doctor?

GRASP - patterns for responsibilities Expert: Example

- The Receipt basis aggregates all Deposit item objects which have been inserted in the machine. So it is an *Expert* for computing the total of the values of these items.



GRASP - patterns for responsibilities LOW COUPLING: Problem

- How to support low dependency, low change impact, and increased reuse?
- Coupling:
  - measurement of how strongly one element is connected to, has knowledge of, or relies on another.
  - A class with high coupling relies on many other classes, and may suffer from the following:
    - forced local changes because of changes in related classes
    - harder to understand in isolation
    - harder to reuse because its use requires the additional presence of the classes on which it is dependent.

GRASP - patterns for responsibilities
LOW COUPLING: Solution

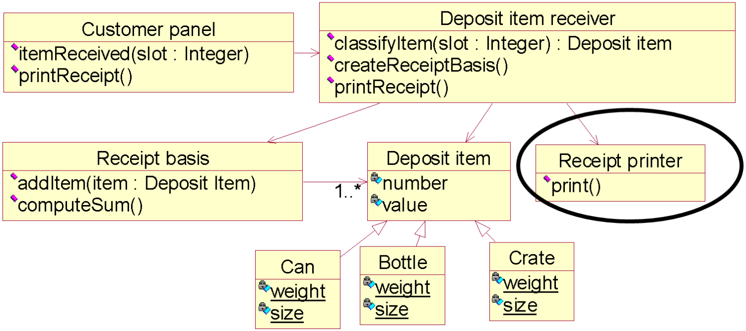
- Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

GRASP - patterns for responsibilities LOW COUPLING: Discussion

- Low Coupling is an evaluative pattern which a designer applies while evaluating all design decisions.
- Coupling happens in the same forms as visibility: local, global, as a parameter, as an attribute.
- A subclass is strongly coupled to its superclass, so subclassing needs to be considered with care!
- Low Coupling supports reuseability, so classes which are inherently very generic in nature should have especially low coupling.

GRASP - patterns for responsibilities
Low Coupling: Example

- The Receipt printer is not dependent on other objects in this design.
- Similarly the Deposit item, but it is structurally dependent on the overall system.



GRASP - patterns for responsibilities CONTROLLER: Problem

- What first object beyond the UI layer receives and coordinates ("controls") a system operation?
- System operations are the major input events upon our system.
- A controller is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

GRASP - patterns for responsibilities CONTROLLER: Solution

- Assign the reponsibility to a class representing one of the following:
  - Represents the overall "system", a "root object", a device that the software is running within, or a major subsystem
  - Represents a Use Case scenario within which the system event occurs.
    - Use the same controller class for all system events in the same Use Case scenario
    - Informally, a session is an instance of a conversation with and Actor. Sessions can be of any length but are often organised in terms of Use Cases.

GRASP - patterns for responsibilities HIGH COHESION: Problem

- How to keep objects focused, understandable, and manageable, and as a side effect, support low coupling?
    - Cohesion is a measure of how strongly related and focused the reponsibilities are.
    - An element with highly related responsibilities that does not do much work is of high cohesion.

GRASP - patterns for responsibilities HIGH COHESION: Solution

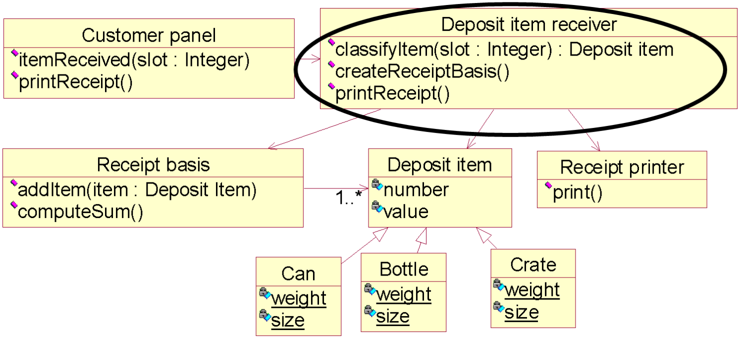
- Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

GRASP - PATTERNS FOR RESPONSIBILITIES HIGH COHESION: DISCUSSION

- Benefits:
  - Clarity and ease of comprehension of the design is increased.
  - Maintenance and enhancements are simplified.
  - Low coupling is often supported.
- Rule of thumb:
  - A class with high cohesion has a relatively small number of methods, with highly related functionality, and does not too much work.

GRASP - patterns for responsibilities
HIGH COHESION: Example

- The Deposit item receiver has two unrelated tasks, namely classifying the items and printing the receipt.
- Solutions:
  - Split the class in two, *or*
  - Assign the "printReceipt" responsibility to someone else (e.g. the Receipt basis).

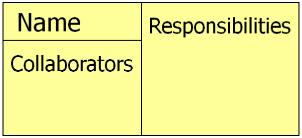


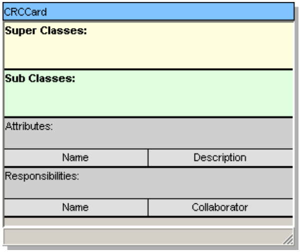
GRASP - patterns for responsibilities Polymorphism, Pure Fabrication, Indirection, Protected Variations

- Polymorphism
  - How to handle alternatives based on type?
- Pure Fabrication
  - Who, when you are desperate?
- Indirection
  - How to de-couple objects?
- Protected Variations
  - To whom should messages be sent?

CRC cards & role playing

- Not part of the UML design process but useful in detecting responsibilities of objects are CRC cards (developed by Kent Beck and Ward Cunningham).
- CRC stands for Class-Responsibility-Collaborator. They look like:





CRC cards & role playing

- CRC cards are index cards, one for each class, upon which the responsibilities and collaborators of a class are written.
- They are developed in a small group session where people role play being the various classes.
- Each person holds onto the CRC cards for the classes that they are playing the role of.

Testing

Purpose of Testing

- Purpose of testing
  - Finding differences between the expected behaviour specified by models and the observed one of the implemented system
  - The differences reflect failures of a piece of software
  - Verification: Are you build the product right? (Does it work properly?)
  - Validation: Are we build the right product (Does it satisfy user's requirement?)

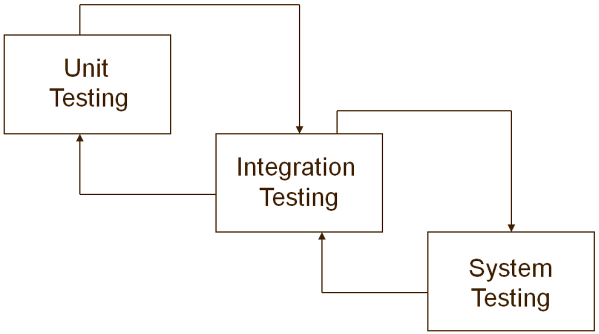
Purpose of Testing

- Causes of a failure
  - Failures are caused by faults, also known as bugs
  - An error is a human action that results in a program containing faults
  - Errors can take place at any stage of a software life cycle
  - Finding an error is a diagnostic progress contain mapping from differences detected in testing to errors

Test Stratagies

- White Box (structural) test
  - Every independent execution path through the code is tested and all conditional statements are tested for true and false statements
- Black Box (specification) test
  - The ‘behaviour’ of object/class is tested and test case design should be based upon domain knowledge.

OBJECT ORIENTED TESTING



Unit Testing

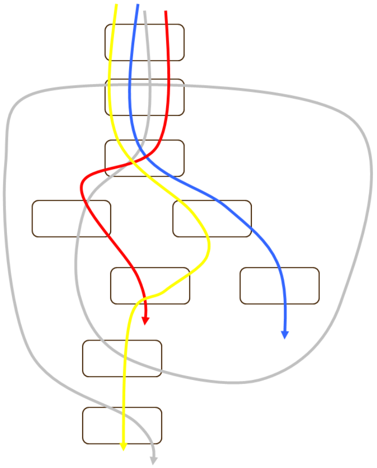
- Aim of Unit testing
    - to test objects/classes, blocks and service packages
    - more complicated than unit testing in traditional program testing, as an object contains both attributes and operation and because of inheritance and polymorphism.

Unit Testing

- Specification testing
  - black box testing
  - equivalence partitioning: partitioning possible inputs into several categories and set one test case for each category
- State based testing
  - tests are performed based on the encapsulated state and the interaction of the operations of an object

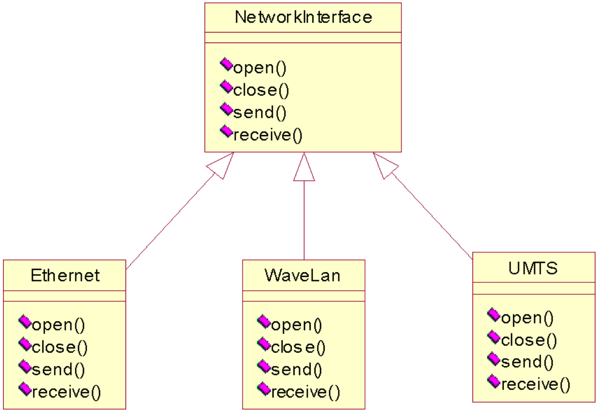
Unit Testing

- Structural testing
  - white box test, also known as path testing



Unit Testing

- Polymorphism testing
  - all possible bindings should be identified and tested



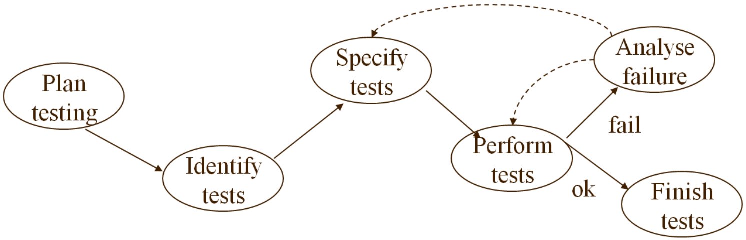
Integration Testing

- Integration testing
  - earlier than traditional cases because objects and classes communicate with one another.
  - Integration testing is any type of software testing that seeks to verify the interfaces between components against a software design. Normally integrated in an iterative way, allows interface issues to be localised more quickly and fixed.
  - Integration testing works to expose defects in the interfaces and interaction between integrated components. Progressively larger groups of tested software components corresponding to elements of the architectural design are integrated and tested until the software works as a system.

System Testing

- System testing of software or hardware is testing conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.
- System testing falls within the scope of black box testing, and as such, should require no knowledge of the inner design of the code or logic
- Each use case is initially tested separately based on requirement model.
- The entire system is tested as a whole after all use case are tested.
- Testing several use case in parallel.
- Testing several use case at the same time.

Testing Procedure



Large Software Management

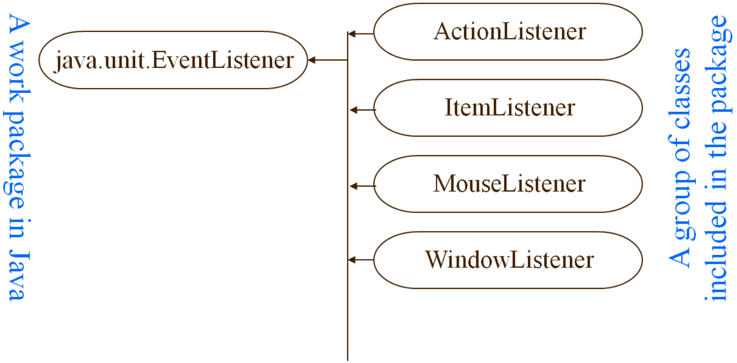
- Functional decomposition - in function-oriented programming, a function is broken down into sub-functions and further into sub-sub-functions and small pieces of programs are developed to implement these sub-sub-functions. (Note: operations and data are separate.)
- It looks like that we can use this idea to break down a use case into sub- or even sub-sub-cases. However, the separation of operation and data does not satisfy OOP's theme.

Large Software Management

- Package - grouping classes together into higher-level units called wok package, assignment or task. (Note: operations and data are not separate in a work package as it is a group of classes and, therefore, package is widely used in OOP.)
- Self-contained - a work package is self-contained, that is, the development of a work package follows the entire procedure of waterfall model.
- Smaller work package is more manageable.
- Work packages are assigned to individuals or teams for completion.

Large Software Management

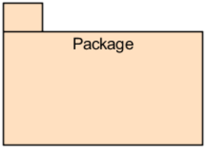
- Work package example



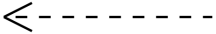
Package Diagrams

Package Diagrams

- A package diagram show packages and the dependency between packages.
  - Package:



  - Dependency:

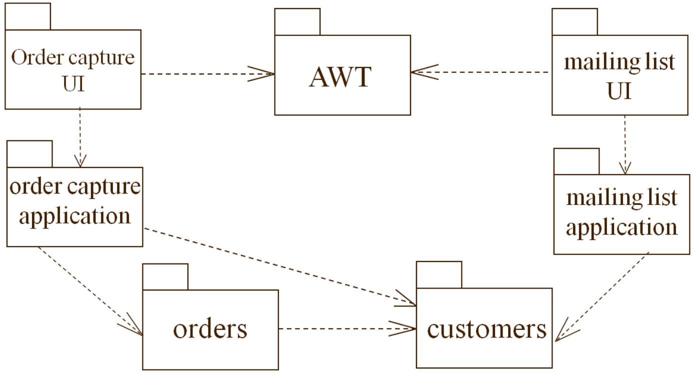


Package Diagrams

- Dependency - if changes to definition of a class in a package A causes the changes in classes in another package B, we say that B has dependencies with A.
    - On class sends message to another (return value from a method).
    - One class mentions another as a parameter (parameter of a method).
    - One class has another as a part of its data (defining reference variable)
- Dependency is not transitive.

Package Diagrams

- Package diagram example


Order capture UI
AWT
mailing list UI
order capture application
mailing list application
orders
customers