

Revision Notes

The Need for Object Oriented Methodology

Hardware technology has been revolutionized several times. Software technology has lagged behind in matching these advances. As a result, software is expensive, of insufficient quality, hard to manage, etc. Object-oriented methods are viewed as a possible step to overcome this software crisis.

The task of software is to bridge the gap between concepts in an application and computer concepts through a combination of software design principles (abstraction, information hiding, modularization, etc.) and using the expressiveness of programming languages (basic control structures, procedures and functions, data structures, etc.)

Encapsulation: separation of inside and outside of an object – forbids clients from seeing the internal details.

Inheritance: one class inherits structure of data and functionalities of another

Polymorphism: the ability to manipulate objects of distinct classes using only knowledge of their common properties without regard for their exact class.

A good system is one which meets its users' needs. It should be useful and usable, reliable, flexible, affordable and available. Advances in software have revolutionized many areas. But there are still frequently occurring problems:

- Systems which do not meet their users' requirements and/or have technical failings
- Inflexibility
- Maintenance requires highly skilled professionals

A good system is a collection of modules. Modules often depend upon one another (such as a client and a server). A good system has low *coupling* (dependency on other modules), an *interface* (defines some features of the module on which its clients may rely), makes use of *encapsulation* (hides the details of the implementation of an object) and *abstraction* (extracting essential properties of a concept), has a high level of *cohesion* (connectivity to other modules) and has reusable modules.

Software engineering involves defining a clear set of requirements, using a defined process with clear phases, each of which has an end-product, and performing continuous testing, verification and validation. The system should use relevant architectures and components and make sensible use of tools.

Fundamentals of Object Technology

An *Object* is a tangible entity which exhibits some well-defined behaviour. It has clearly defined boundaries, modelling some part of reality. An Object has:

- State: encompasses all of the properties of the object (usually static) and the current values of each of these properties (usually dynamic).
- Behaviour: how an object acts and reacts, in terms of its state changes and message passing.
- Identity: property of an object which distinguishes it from all other objects.

The Object-oriented view of a system is a collection of objects, where each object models an entity or event in an application problem, and where all objects work together to achieve the goal of the system.

A *class* is a set of objects that share a common structure and a common behaviour – in effect, a template for an object. A single object is an *instance* of a class.

Three basic kinds of relationships between classes:

- inheritance (generalisation/specialisation): a subclass inherits the structure and behaviour of its superclass. An abstract class is one that cannot be instantiated but can be inherited from. Such a class can include implementation – an interface cannot.
- aggregation (whole/part): an object is composed of a set of parts
- association: denotes some semantic dependency among otherwise unrelated classes.

Revision Notes

Access Rights

Private - can only be accessed by its own operation
Public - can be accessed by operations of any class
Protected - accessed by direct subclass only

Polymorphism is the ability of objects of different classes to respond to the same message, usually in different ways.

UML

UML is a language and notation for specification, construction, visualisation and documentation of models of software systems. Developed by Grady Booch, Ivar Jacobson and James Rumbaugh, there are four types of diagram: use case diagrams, class diagrams, behavioural diagrams and implementation diagrams.

UML is *not* a programming language, a CASE tool or a method.

Use Case Diagrams

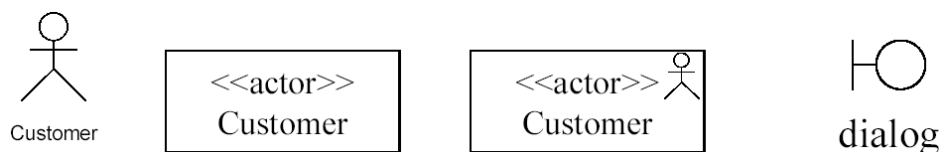
A *Use Case* describes a task that a user can perform using the system. Use Cases describe requirements for the system. A task described by a use case is composed of activities. A use case can have different variations called scenarios. A use case should not be used for functional decomposition.

Use Case notation allows for the following fields to be defined:

Use Case Number, Name of use case (normally verb+noun describing case), Actors, Preconditions, Postconditions, Invariants, Non-functional requirements, Process description, Exceptions, error situations, Variations, Rules, Services, Contact partners, sessions, Notes/open questions, Documents, references, dialog samples, Diagrams (sequence and collaboration, class, activity and state).

An Actor is an external entity which is involved in the interaction with the system described in a use case. It represents a role in the system – not necessarily a user or other system.

Notation



Actors can be generalized/specialized, demonstrating inheritance properties similar to that of Objects, e.g., a Home Insurance clerk, Car insurance clerk and Field service clerk all extend (i.e. have the same functionality, plus a bit more, as) Office Clerk.

Use Case Diagrams show the relationships between a set of use cases and the actors involved in these use cases. They are a tool for requirement determination, describing those activities which are to be supported by the software under development.

Relationships between use cases

Include: If several use cases include a part of functionality that is the same, this can be extracted into a separate use case, and included into the base cases which use it. This reduces data redundancy.

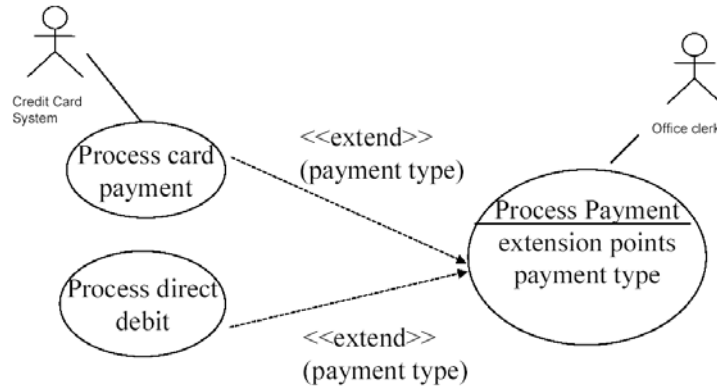
Extend: a use case is optionally extended by functionality of another use case, i.e. if a certain condition is fulfilled a use case, it extends to another use case – used to show variations in functionality.

Generalization: sub use case inherits behaviour and semantics from super use cases.

Revision Notes

How to identify use cases

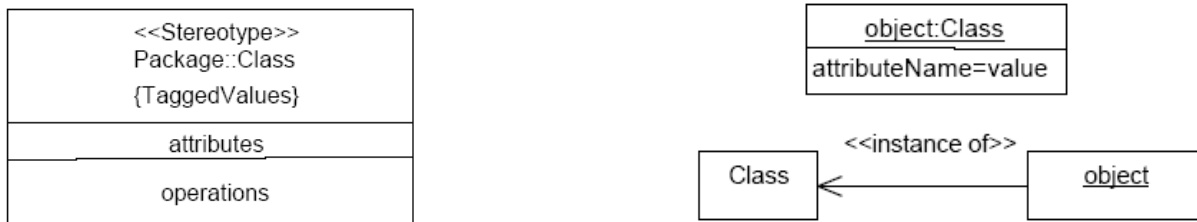
- Define the list of actors and then try to work out the use cases for each actor.
- Identify external events to which system/user reacts.



Class Diagrams

A *class* contains the definition of the attributes, the operations, and the semantics of a set of objects. An *object* is an actual existing and acting unit which encapsulates this state and behaviour.

Notation



An *attribute* is a data element contained in each object of a class.

- name : data type
- constraints
- tagged values (example: {readonly})
- optional attribute: optionalAttr[0..1]: Class
- mandatory attribute (default): mandatoryAttr[1]: Class
- sets are denoted by [*]

Derived attributes are those whose value is calculated automatically. Class attributes/variables do not belong to an individual object, but are attributes of a class.

Access restriction:

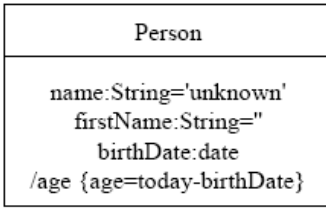
- *public* - visible and usable by all
- *protected* access is allowed to the class itself and its subclasses
- *private* - access is allowed to the class itself

Notation:

- visibility attribute : Class = InitialValue {TaggedValue}{Constraint}
- /derivedAttribute
- classAttribute
- +publicAttribute
- #protectedAttribute
- privateAttribute



Examples



colour : {red, blue, green}
 radius : Integer = 25 {readonly}{radius > 0}
 -counter : Integer
 time : DateTime::Time
 dynamArray[*]
 name[1] : String
 firstName[0..1] : String
 firstNames[1..5] : String

Operations, Methods

Operations are services which may be required from an object. Method are the implementation of operation. Information on the activity the object is expected to carry out is passed by messages.

Notation:

name(parameter:ParameterType = DefaultValue, ...) : ReturnType {TaggedValues}{Constraints}
 I.e., setPosition(x:Integer = 1, y : Integer = 1) : Boolean{abstract} {(x>0) and (y>0)}

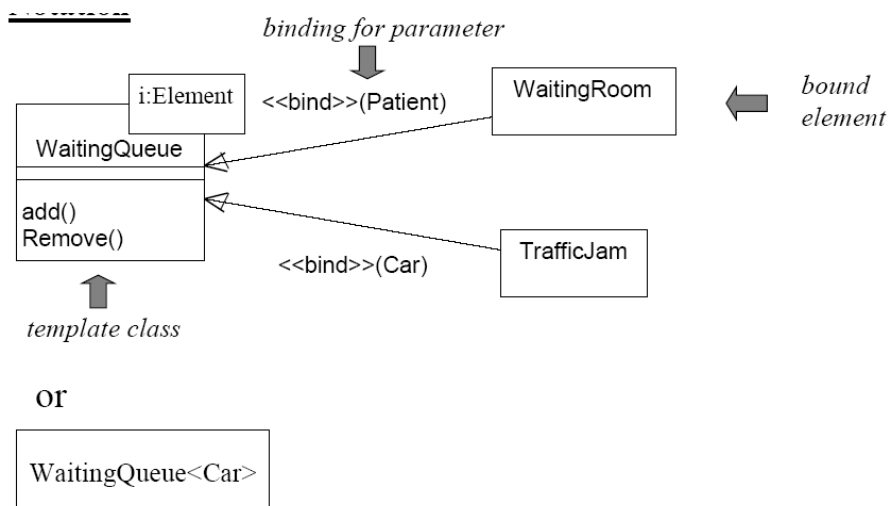
+publicOperation()
 #protectedOperation()
 -privateOperation()

Types of operations with respect to system state:

- query - operation that gets a value from a class without changing the system state
- modifier - operation that changes the system state

Parameterised Classes

Parameterised class is a template equipped with generic formal parameters, which can be used to generate common classes. A class generated with the aid of a parameterised class is called a bound element. Typical application are collection classes (STL).

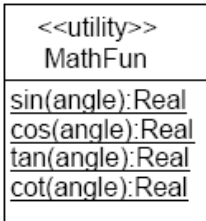


Revision Notes

Abstract Classes are given the tagged value {abstract}

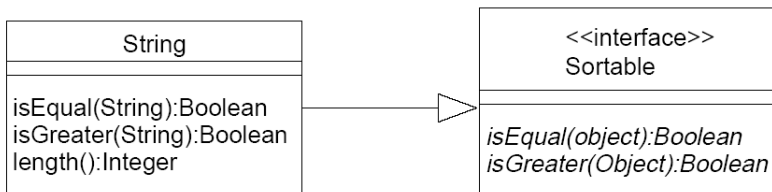
Utility Classes are collections of global variables and functions which are combined into a class. They are not true classes.

Notation:

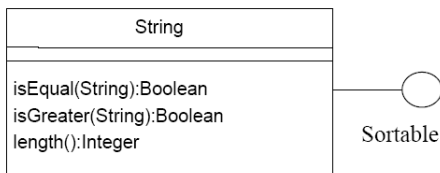


Interfaces describe a selected part of the externally visible behaviour of model elements. Interface classes are abstract classes which define abstract operations exclusively. An interface contains a set of signatures for operations that classes wishing to provide this interface need to implement. Inheritance relationships are possible.

Notation:

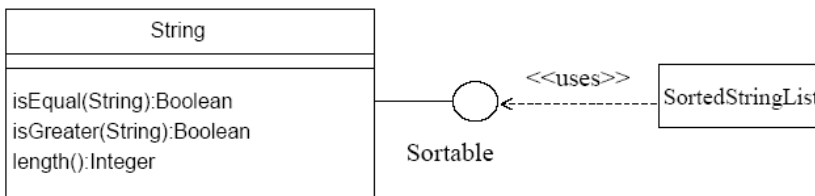


or



18

Example:



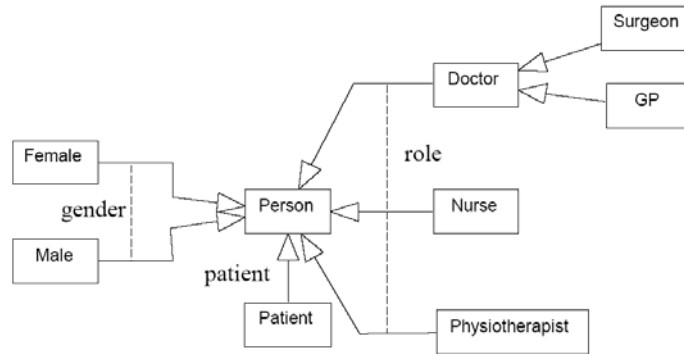
Abstract classes versus interfaces:

- both allow for definition of interface and defer the implementation until later
- Abstract class:
 - o may have attributes
 - o may have implementation of some of the methods
 - o no objects
- Interface:
 - o no attributes
 - o no implementation of the methods
 - o no objects

Revision Notes

Inheritance is a relation between superclasses and subclasses which enables attributes and operations of a superclass to become accessible to its subclasses.

A *Discriminator* denotes the aspect relevant for hierarchical structuring of the properties. A *Partition* is an entirety of subclasses based on the same discriminator. For example:

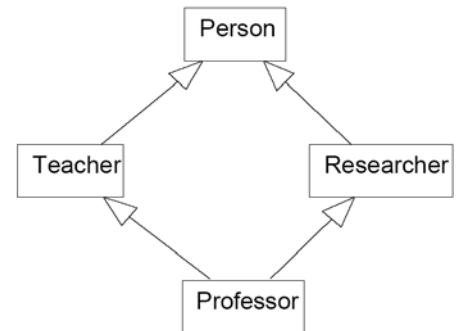


Multiple classification allows multiple types for an object (not supported in Java and in C++).
 Legal combinations of subtypes: (Female, Patient, Nurse), (Male, Physiotherapist), (Female, Patient), (Female, Doctor, Surgeon)
 Illegal combinations: (Male, Doctor, Nurse)

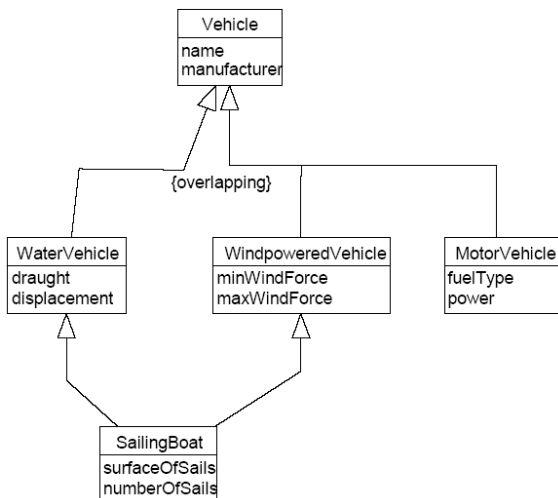
Multiple inheritance enables a class to specialise from more than one superclass.

Problems:

- Name collision: two or more different superclasses use the same name for some of their properties
 - o Solution: address their property in a fully qualified way
- Repeated inheritance: two or more superclasses share a common superclass
 - o Solution: superclass is not replicated



Overlapping versus Disjoint

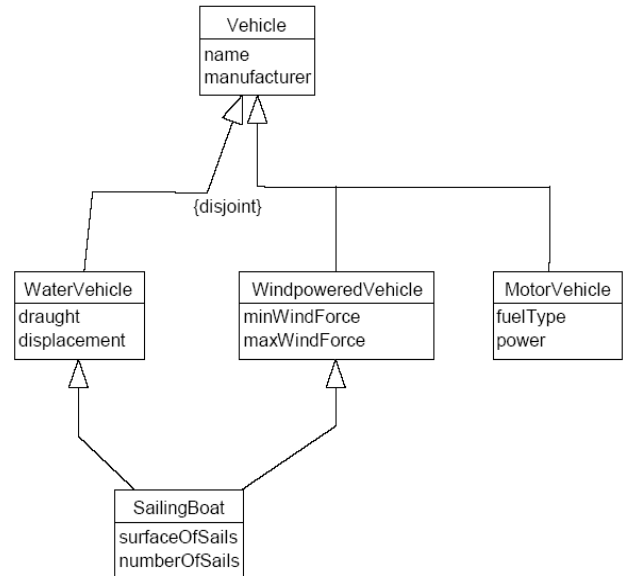
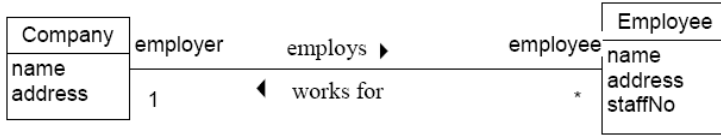


Overlapping: It is acceptable for one class to specialise both the overlapping subclasses of Vehicle.

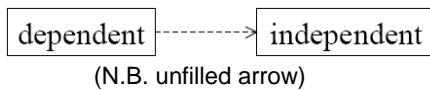
Revision Notes

Disjoint: This specialisation is not correct - an object cannot be an instance of both WaterVehicle and WindpoweredVehicle

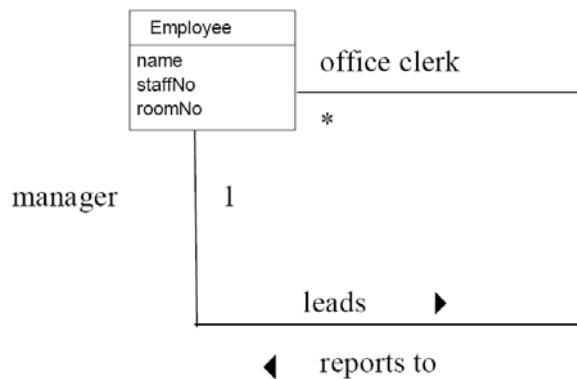
Association describes a connection between classes. The multiplicity of an association specifies the number of objects of the opposite class with which an object can be associated.



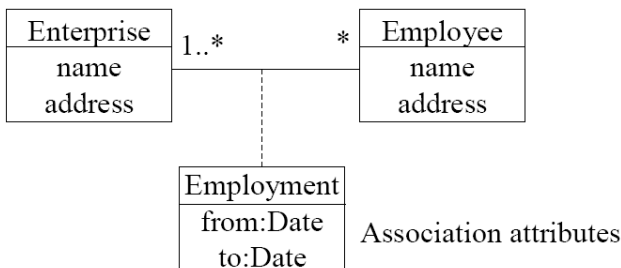
Dependency is a relation between two model elements which shows that a change in one element requires a change in the other. This influences the order of compilation.



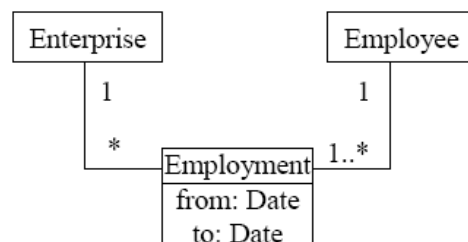
Recursive associations are those in which only one class is involved, e.g.,



An *association class* (attributed association) is a model element which has both the properties of a class and an association. It has attributes that can be associated neither to one nor to the other class which are associated to each other, e.g.,

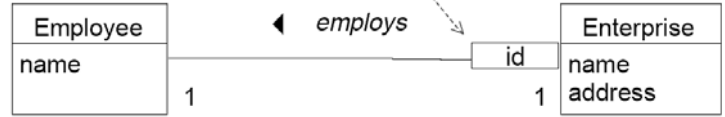


Promoting an association class to a full class enables an Employee to have more than one Employment with the same Enterprise:

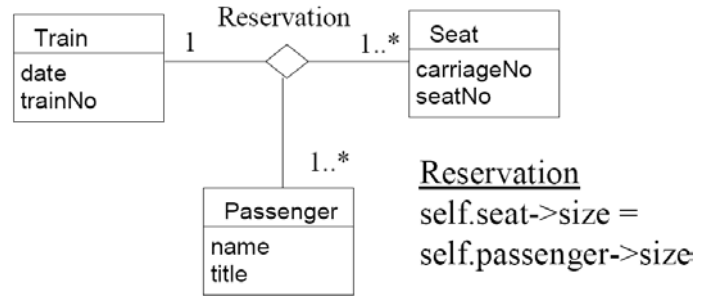
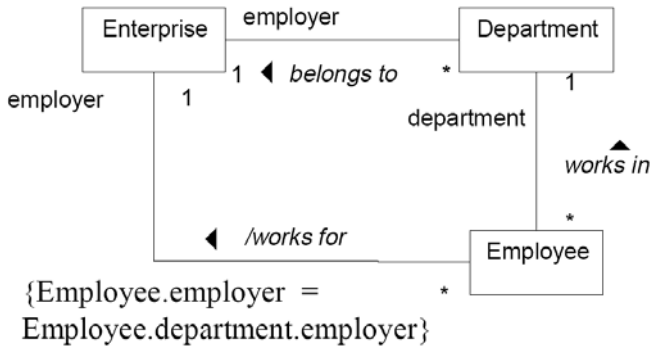


Revision Notes

A *qualified association* is an association in which qualifying attributes are used to subdivide the referenced set of objects into partitions, where each partition may occur only once. Multiplicity always refers to one partition.

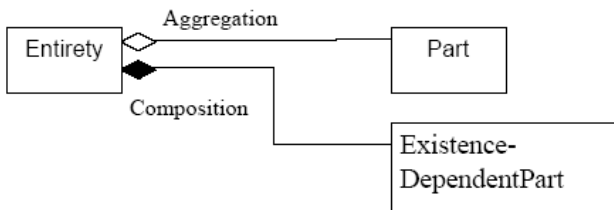
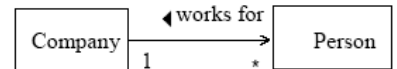


A *derived association* is an association which is calculated when required (below left)



An *N-ary association* is one that involves more than two association roles (above right). Such an association could be expressed using binary expressions, but this has the disadvantage that it is possible to navigate to an association not covered by the original association occurrences.

A *Directed association* is an association in which you can directly navigate from one of the involved association roles to the other, but not vice versa.



Aggregation is an association in which the involved classes represent a whole-part hierarchy. *Composition* is a strict form of aggregation, in which the parts are existence-dependent on the aggregate. Cardinality on the side of the aggregate can only be 1. Each part is part of exactly one composition object.

A *Constraint* is an expression which restricts the possible contents, states or the semantics of a model element such as

- legal set of values of an attribute
- pre- or postconditions for messages or operations
- special context for messages or relationships
- specific order
- chronological condition

Examples:

- {subset}
- {xor}
- a {a>0}
- /age {age=today-birth}
- colour : {red, blue, green}

Revision Notes

Tagged values are keyword/value pairs which extend the semantics of individual model elements. They can influence code generation.

Examples:

```
{private}
{Author=Tara King}
{transient}
{persistent}
GeomFigure {abstract Version=1.3}
visible:Boolean {readonly}
```

A *stereotype* is used as a means of specifying that a modelling element conforms to the well understood pattern of behaviour. They can specify possible usage contexts of a class, a relationship or a package. A model element can be classified with an arbitrary number of stereotypes.

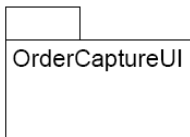
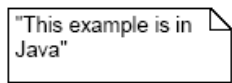
Types of stereotypes:

- decorative: make model elements visually appealing
- descriptive: describe context of usage
- restrictive: describe formal restrictions to existing model elements
- redefining: define new stereotypes

Examples:

```
<<domain class>>, <<model>>, <<view>>, <<controller>>, <<enumeration>>, <<overlapping>>, <<disjoint>>, <<implements>>, <<include>>, <<extend>>, <<actor>>
```

Notes are comments to a diagram or an arbitrary element in a diagram, without any semantic effect such as information on development state, version of a class, developer responsible for the class or its last modification, project management, etc.



Packages are collections of model elements which are used to structure the entire model into smaller units. They are built on the basis of logical relationships. The name of an element in a package must be unique. Each model element can be referenced in other packages as `PackageName::ClassName`

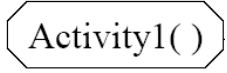
A package must conform to the interface of the general package, <<global>>. All packages in the system have a dependency to this package.

Refinement relations are relations between similar elements of different degrees of detail. *Realisation relations* are relations between interface and its implementation.

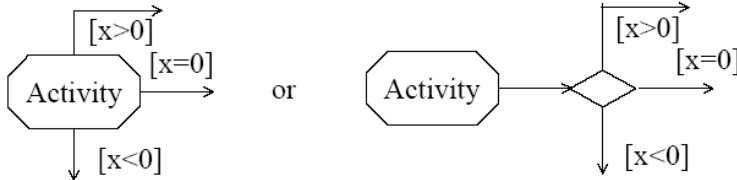
Revision Notes

Activity Diagrams

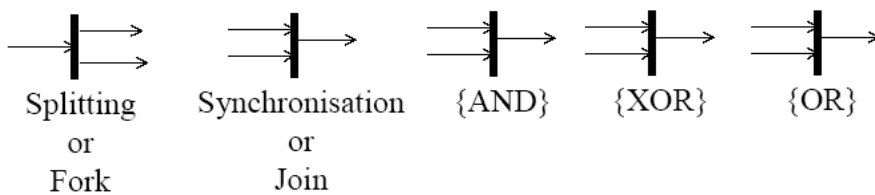
Activity diagrams describe the procedural possibilities of a system with the aid of activities. An activity is a single step in a procedure.



Conditions:

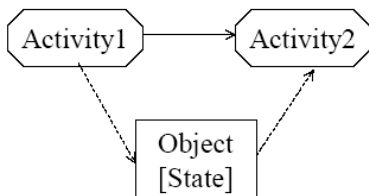


Synchronization:

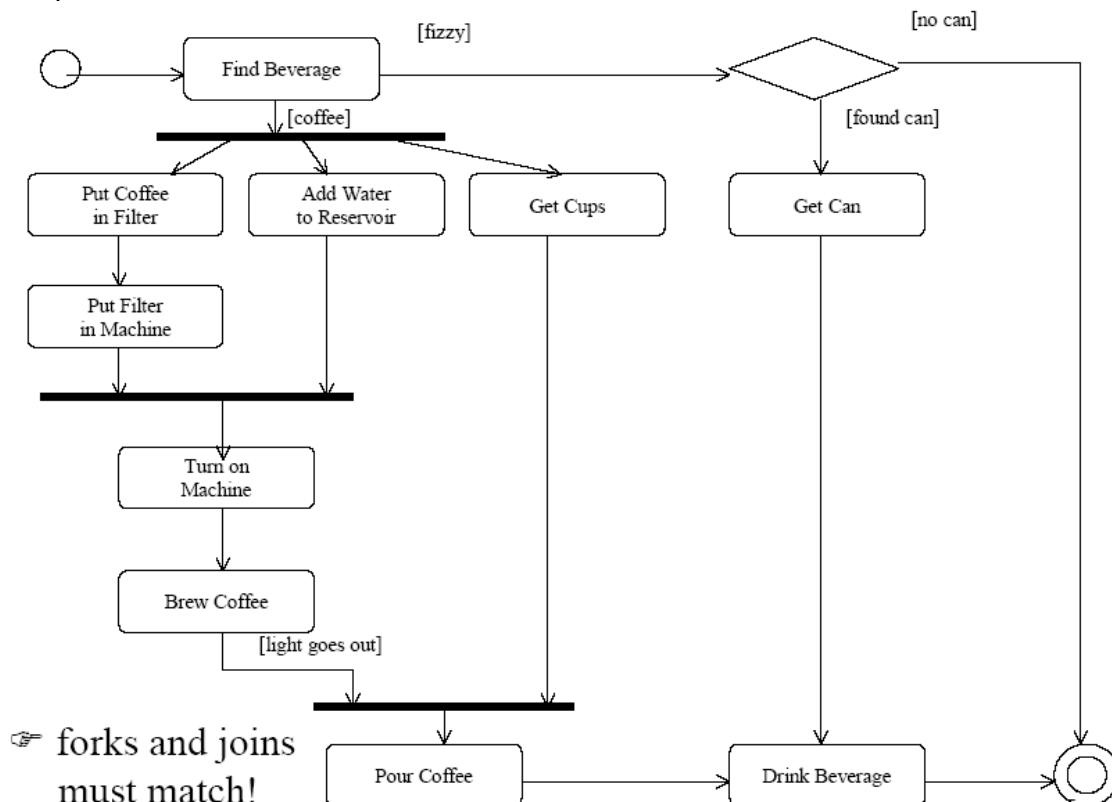


If a condition is false, the thread is considered to be complete as far as the join is concerned. Swim lanes denote responsibility.

State:



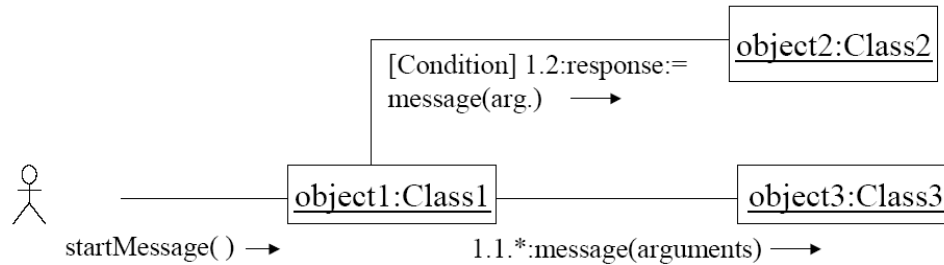
Full Example:



Revision Notes

Collaboration Diagrams

Shows a set of interactions between selected objects in a specific situation, focusing on the relations



between the objects and their topography. They show the chronological course of communication between the objects by numbering the messages. An association relation must exist between the sender and receiver of the message.

PredecessorCondition SequenceExpression : ReturnValue := MessageName (ParameterList)

Sequence Expression

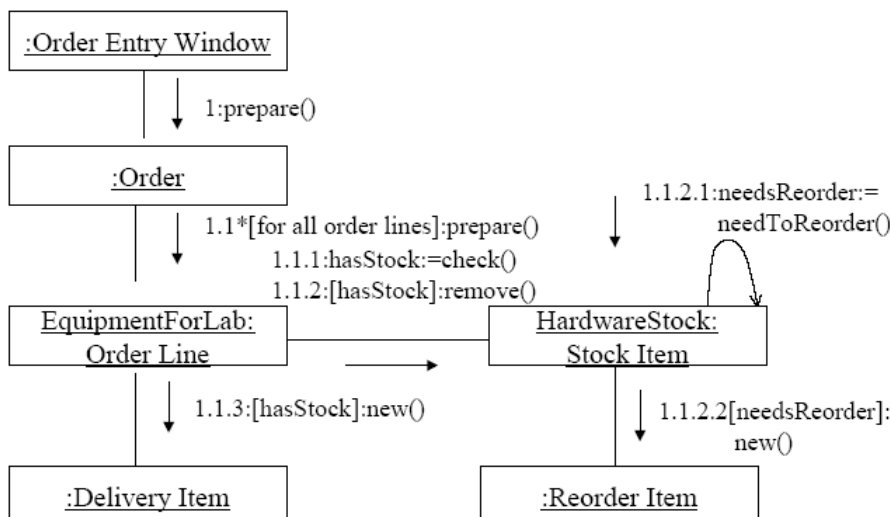
- depth of nesting inside other messages: 1.1
- iteration: 1.2.* [i:= 1..n]
- parallel execution: 1.2.* || [i:= 1..n]
- condition: [x>5]1.2.*

MessageName(ParameterList) corresponds to an operation in the receiving class.

Synchronisation features:

- simple, sequential →
- synchronous →x→
- restricted →↺
- time-dependent →⌚→
- asynchronous →➤

Example:

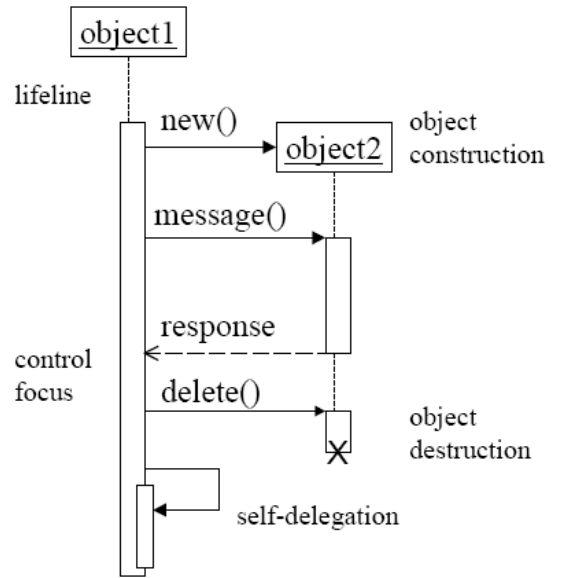
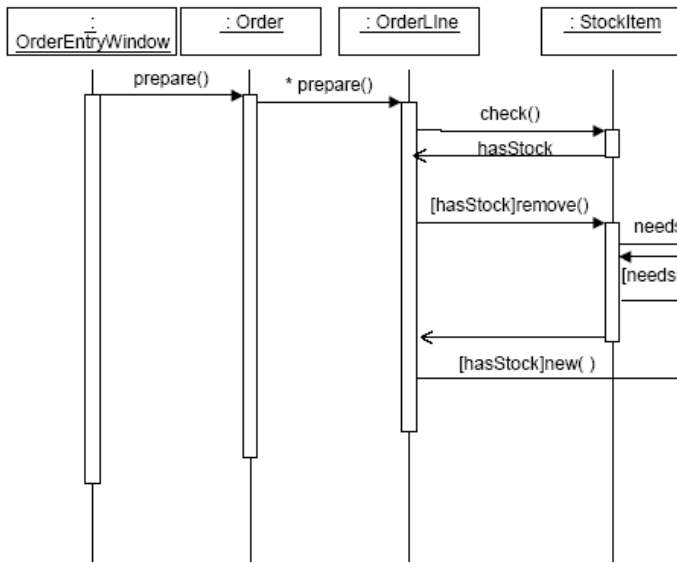


Revision Notes

Sequence Diagrams

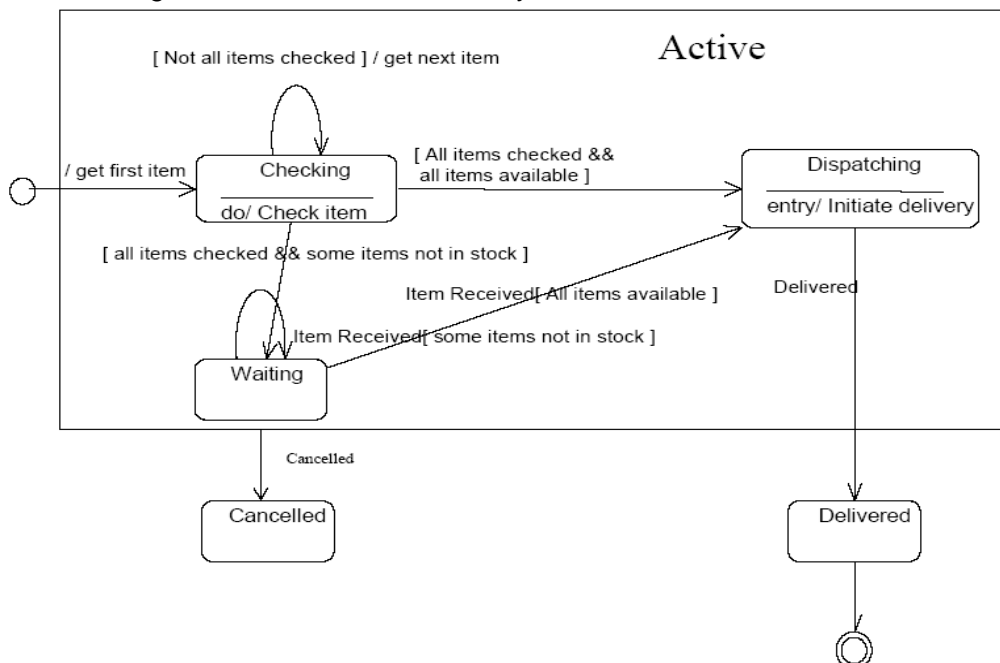
Sequence diagrams show a series of messages exchanged by a selected set of objects with an emphasis on the chronological course of events.

- lifeline : represents the object's life during the interaction
- self-call : a message that an object sends to itself
- [condition] : a message is sent only if the condition is true
- iteration marker *[] : a message is sent many times to multiple receiver objects
- return : indicates a return from a message



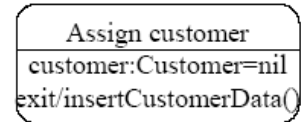
State Diagrams

A state diagram shows a sequence of states an object can get into during its lifetime, together with the events that cause changes of states. It is essentially a finite automaton.



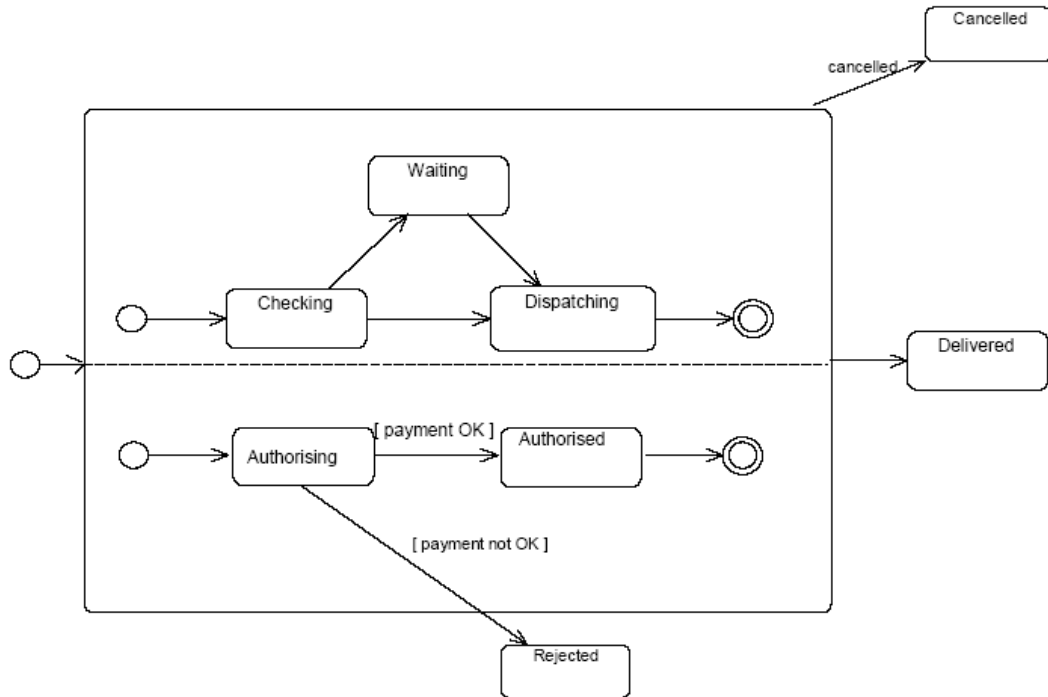
Revision Notes

States belongs to a single class and represents a combination of possible attribute values that the objects of this class may assume.



Description of the state variable: variable:Class = InitialValue {Tagged Value} {Constraint}
Description of the activity: triggers / Activity

A number of states can be represented sequentially as well as being combined:



Events and Transitions

An event is an occurrence which triggers a state transition. An event may be triggered when a condition defined for a transition is satisfied or when the object receives a message:

Behavioural Diagram Summary

When to use activity diagrams?

- Describe the sequence of actions for several objects and use cases.
- Support parallel behaviour.
- It cannot show clear links among actions and objects.

When to use collaboration and sequence diagrams?

- Capture the behaviour of a single use case.
- Good at showing collaborations among objects.

When to use state diagrams?

- Describe the behaviour of an object across several use cases.
- Use state diagrams only for those classes that exhibit interesting behaviour.

Revision Notes

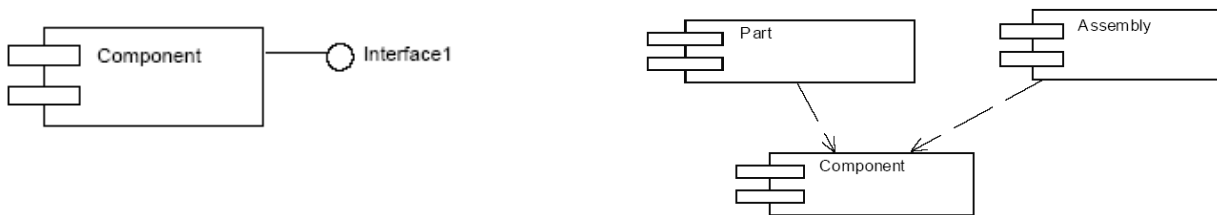
Component Diagrams

A component represents a physical piece of program code. Component diagrams show the interrelations between components through a logical (packages) and physical (components) perspective.

UML has 5 types of components:

- Executable
- Library
- Table
- File
- Document

The properties a component exports are defined by its interface.

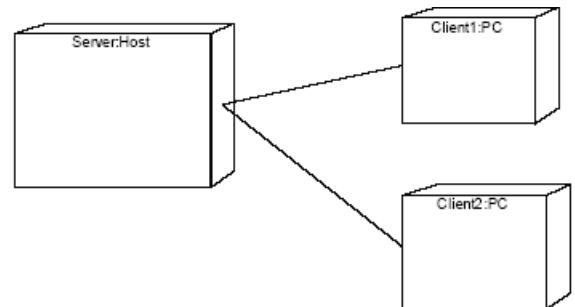


Dependencies among the components show how changes to one component may cause other components to change (communication and compilation dependencies).

Use an implementation diagram to show physical information, different from associated logical information.

Deployment Diagram

A Deployment diagram shows the physical relationships among software and hardware components in the system. It shows which components run on which node. Connections among nodes show the communication paths over which the systems will interact. A node is an object which is physically present at run time and has computing power or memory.



Rational Rose

Advantages and Disadvantages of using CASE tools, such as Rational Rose

Advantages of using CASE tools

- Good for large, complex applications
 - o team of developers
- Decreases the development time
 - o an automatic way to move from one diagram to another and to code
- Names and data are maintained in a consistent manner
- Provides synchronization for different developers

Disadvantages

- Fixed development approach
- Limitation in the flexibility of the documentation
- Costs (software, manuals, training)

Object-Oriented Analysis & Design

Analysis

Precise abstraction of *what* the desired system must do, not how it will be done. Object-Oriented Analysis is method which examines requirements from the perspective of the classes and objects.

Design

How the system will meet the requirements. Object-Oriented Design refers to any method that leads to an object-oriented decomposition.

Object-Oriented Programming

Method of implementation in which programs are organised as cooperative collections of objects, each of which represents instance of some class, and whose classes are related to one another via inheritance relationships.

Case Example: Car Rental System

See Lecture Notes

Reuse

Architecture, code, designs, documentation and tests can all be reused. Reuse is not reuse of language or tools that are not part of the system. Cut and paste and component reuse are both valid methods. Reuse can save time, increase reliability and reduce market cycle time.

Reuse Difficulties

- searching problem
- does it do what you want?
- do you trust it?
- unnecessary things may be imported
- which components are genuinely reusable?

Components must be sufficiently general, properly documented and thoroughly tested. Using components is more beneficial than building them. Object orientation enables higher level of reuse than traditional software development methods as it encourages the high cohesion / low coupling style, it concentrates on problem domain objects and it brings the benefits of inheritance.

Design patterns

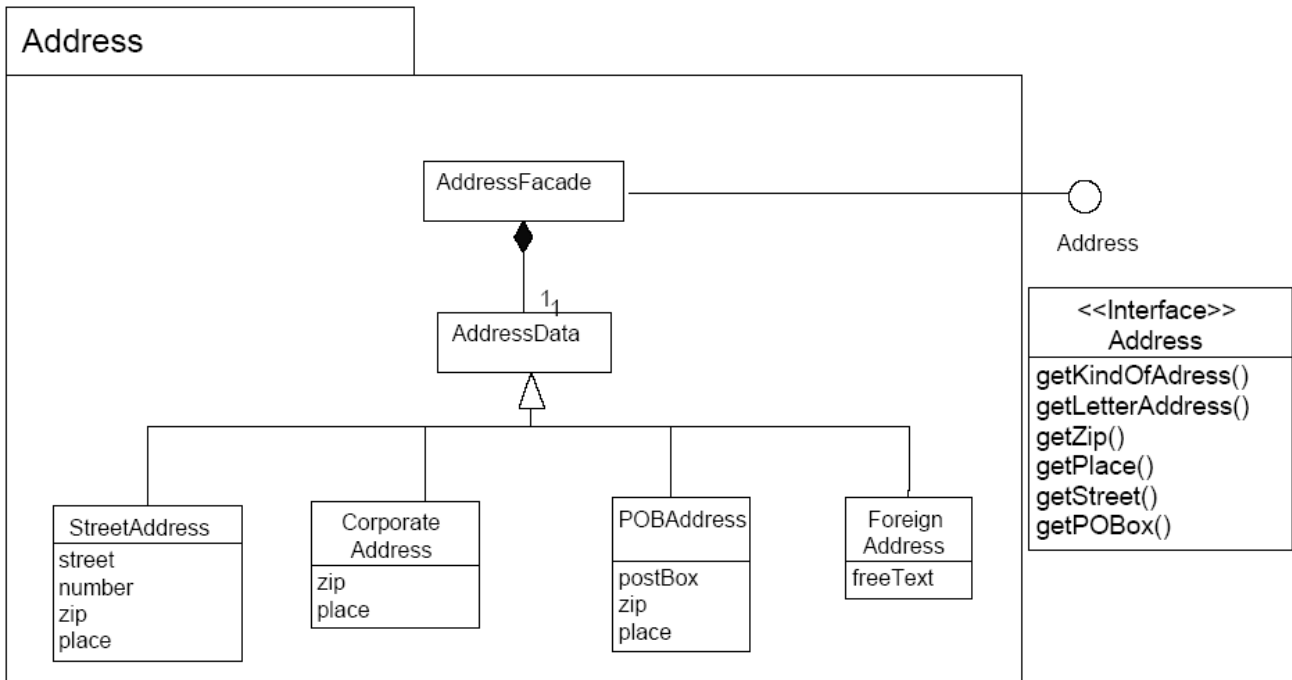
A pattern is a named, well understood good solution to a common problem.

Documenting Patterns:

- Name: the name of the pattern that conveys the essence of the pattern.
- Abstract: a very brief overview of the pattern
- Context: architectural or business context and the critical factors for the pattern that will make it work in a particular situation.
- Problem: the general description of the problem that pattern helps to solve
- Solution: the components of the pattern, their relationships, responsibilities and collaborations
- Consequences: the results and trade-offs of applying the pattern

Revision Notes

Example. Facade hides a complexity of a subsystem behind a single class:



- Name: Façade
- Abstract: defines a high-level interface to subsystem hiding its structure.
- Context: building easy-to-use subsystem.
- Problem: subsystem consists of many related classes, each providing part of the subsystem's functionality. Clients have to understand the structure of the subsystem. It increases the coupling of the system. Changes to the structure of the subsystem may require changes to the clients.
- Solution: add a new class Façade which provides a single unified interface to the subsystem. An object of this class forwards the messages to the appropriate object inside the subsystem.
- Consequences: clients do not depend on the structure of the subsystem.

A *Framework* is a reusable chunk of architecture. Usually a framework provides classes which have to be specialised when the framework is applied and operations which have to be implemented. Solutions are not optimal, even obsolete.

Verification, Validation, Testing

Verification is the process of checking whether the system is correctly implemented to its requirements.
Validation is the process of checking if the system does the job it is supposed to do.
Testing ensures verification and validation.

Verification

1. Verify that the use cases described in the UML model satisfy the requirements specification.
2. Verify that the classes are capable of providing the use cases.
3. Verify that the code corresponds to the classes in the design.

Check correspondence between UML model and the program. Ranges from a developers check to a formal proof. Could use a UML modelling tool, a person who did not develop a system or a checklist.

Revision Notes

Validation

Is uncertain and requires customers as it concerns usability. Usability testing can be performed by colleagues, experts in usability and real users.

Testing

A successful test is one that finds a bug.

Kinds of testing:

- usability testing
- module testing
- integration testing
- system testing
- acceptance testing
- performance testing
- stress testing
- regression testing

Tests have to be:

- repeatable
- documented (both tests and the results)
- precise
- done on configuration-controlled software

Consider automation of the testing.

When to write the test specification? - as early as possible

Scenarios in use cases describe the requirements and serve as source of test cases.

Usability testing cannot be automated!

Problems of Object Orientation:

What is a unit test?

Class: problem with the states of objects each transition should be tested

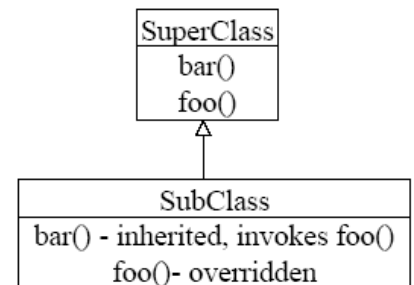
Encapsulation: state-reports method

Inheritance:

Problems of testing

- boring
- expensive in time and hence money
- planned mostly for the end of the project
- customers pressure developers into delivering on time

Reviews and inspections: The meetings do not discuss how to remedy the defects!



Software Development

Traditional Waterfall model

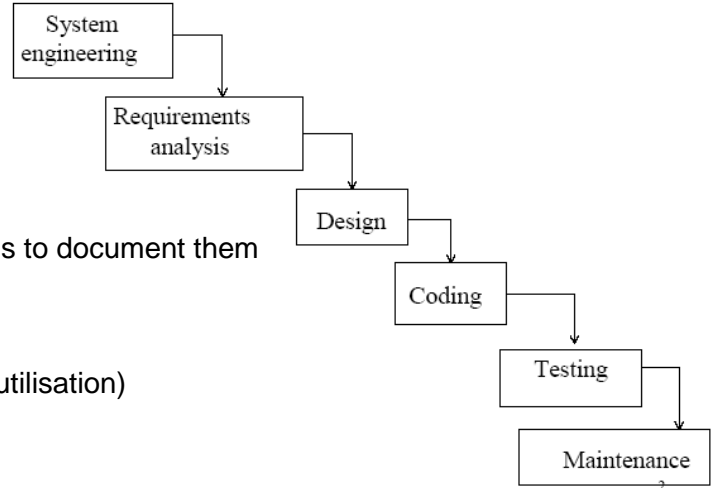
One phase is completed before the next is entered

System engineering

- High level specification
- Defines major elements: human, software, hardware and how they interact

Requirements analysis

- Defines users requirements
- Use of fact finding techniques and techniques to document them (Interviewing, Data Flow Diagrams ...)
- Requirements specification contains:
 - o detailed description of all the tasks
 - o constraints (response time, memory utilisation)
 - o design / implementation directives
- +
- o maintenance support for the system
- o training of the customer' staff



Design

- Detersmines how to construct a system that delivers the requirements
- Specification of a software architecture

Coding

- Translation of design into program code

Testing

- Ensures that the system meets requirements
- Several levels of testing

Maintenance

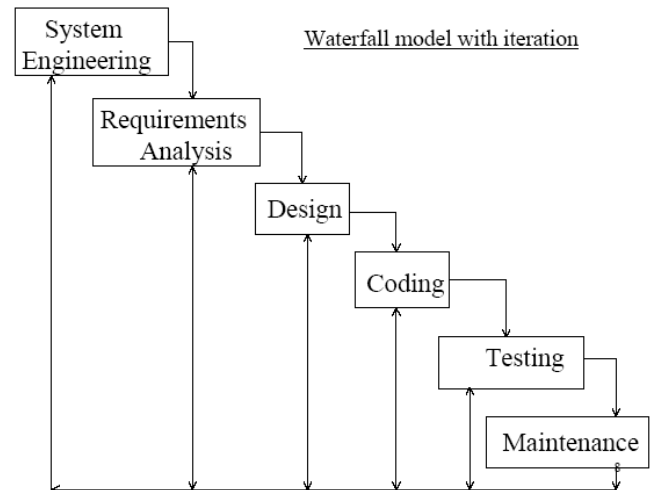
- The system is subject to change
- Corrective maintenance (corrections of the errors)
- Perfective maintenance (implementation of certain aspects of system behaviour)
- Adaptive maintenance (accomodation of changing requirements).

Advantages

- Enables allocation of tasks within a phase.
- The progress can be evaluated at the end of each phase.

Disadvantages

- Projects rarely flow in a sequential process.
- Difficult to define all requirements at the beginning of a project.
- Unresponsive to changes.
- A working version of the system is not seen until late in the project's life.
- Repairing problems further along the lifecycle becomes progressively more expensive.
- Maintenance costs can be as much as 70% of systems costs



Revision Notes

Prototyping

A prototype is a system or partially complete system that is built quickly to explore some aspects of the system requirements. Constructed with various objectives.

Advantages

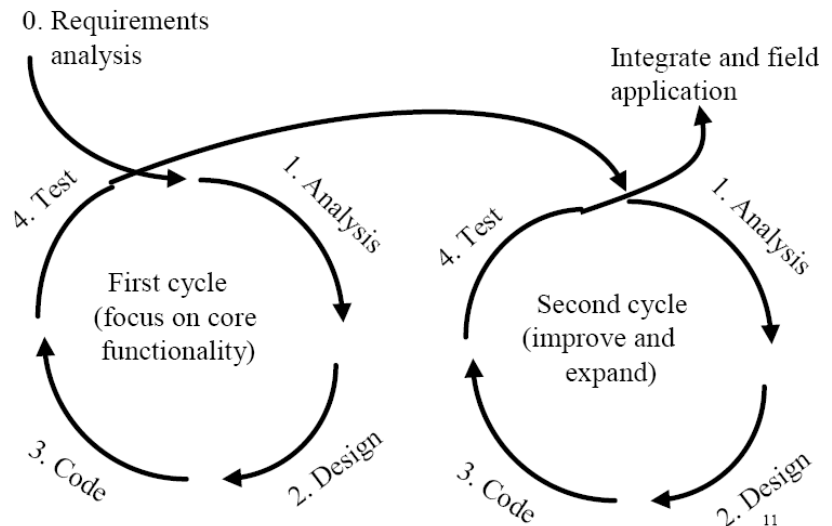
- Early identification of misunderstandings between developers and users.
- Identification of difficulties in the interface.
- Feasibility and usefulness of the system can be tested.

Disadvantages

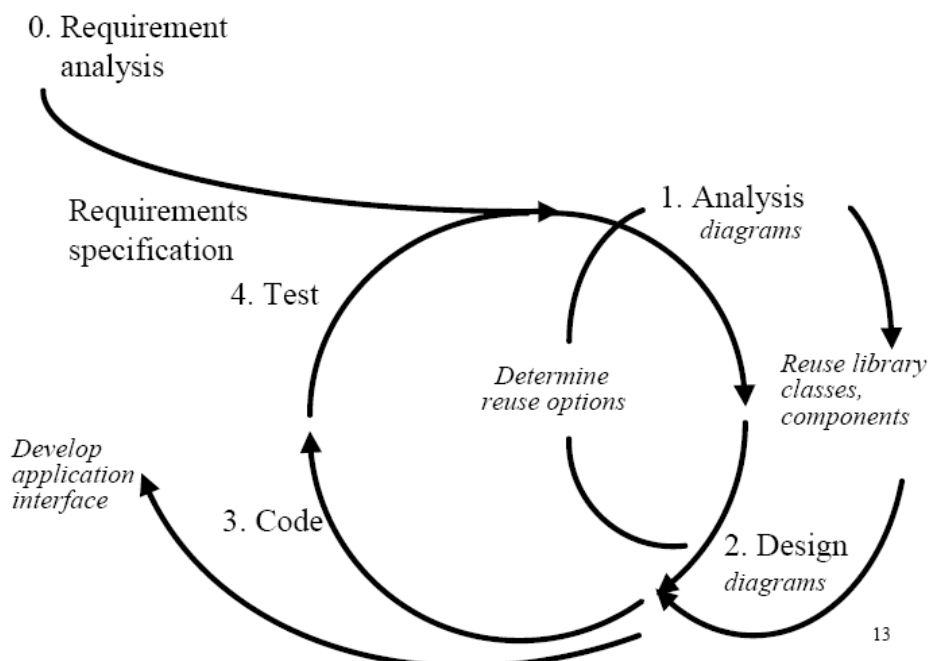
- The user may perceive the prototype as part of the final system.
- May divert attention from functional to interface issues .
- Requires significant user involvement.
- Difficult management of the system life cycle.

An Iterative Approach to Object-Oriented Development

1. Define the scope and requirements for the entire project and create a high-level description of the entire application.
2. Select a portion of the entire application to develop first
 - the hardest part of the application or
 - a portion of the system that supports some of the functionality that the end users are most interested in.
3. Plan what part of application will be developed during the first prototyping cycle, the second cycle, and so on.



One Cycle of the Object-Oriented Development



Revision Notes

Phases of a development cycle

1. Requirements analysis
 - specification of the scope and requirements
 - accompanied by scenarios of the system's behaviours
 - a good requirements statement enables easier modularization of the application.
2. Analysis Phase
 - diagrams
 - no rigid distinction between analysis (what the application should do) and design (how to do it)
 - classes from the domain of the application refer to real things in the application
 - infrastructure classes handle the details of programming
3. Design Phase
 - extended diagrams + infrastructure objects
 - shift from logical relationship between objects to the physical layout
 - reuse
 - o class libraries
 - o development of classes to be reused
4. Coding Phase
 - no sharp break between design and coding
 - object-oriented modelling tool
 - development of the code for the application interface
5. Testing Phase
 - testing the code to see if it functions properly.
 - testing leads back to analysis and design

Benefits of object-oriented methodology

- Productivity
 - o object-oriented methodology can increase the productivity of the project team and can reduce the project schedule as much as an order of magnitude
 - o intrinsic power of the object-oriented programming languages
 - o reuse of classes and objects
- Rapid development
 - o Reusability
 - o prototyping
- Quality
 - o absence of defects
 - o better in object-oriented methodology (reuse, encapsulation)
- Maintainability
 - o greater in object-oriented methodology (encapsulation) than in conventional architectures based on hierarchy of functions