Modelling Software Architecture

Emmanuel Letier

Motivation and Orientation

- Modelling helps communicating and analyzing software architecture
- Many architecture modelling methods and notations
  - General agreement: multiple views are needed
  - But no agreement what the views should be
    - Many variants
    - Lack of clear definitions
- A *pragmatic* approach
  - The course covers the main views and core concepts present in all methods

Architecture Models

Context Diagram
describes the software system's interactions with its environment (people, hardware devices and other software systems)

Functional Viewpoint
a ***run-time*** view of the system's components and connectors

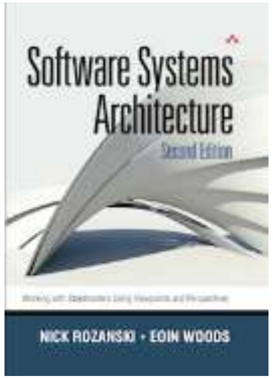
Deployment Viewpoint
a ***run-time*** view of the system's infrastructure (nodes and networks), and where each component runs

Development Viewpoint
a ***design-time*** view of the organisation of the system's code into modules and of the module dependencies

The Functional Viewpoint

Chapter 17 of

N. Rozanski and E. Woods, *Software Systems Architecture : Working With Stakeholders Using Viewpoints and Perspectives*, 2nd *Ed.*, Addison-Wesley, 2012.



The Functional Viewpoint (Component Diagram)

What

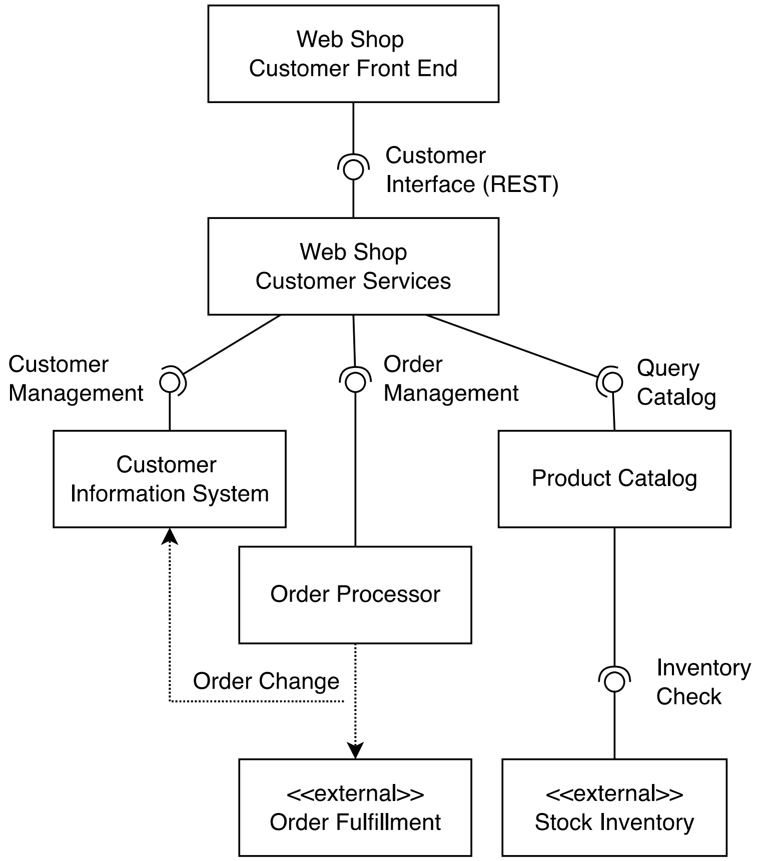
Describes the system’s functional elements (aka. components), their responsibilities, interfaces, and primary interactions.

Why

- To facilitate development and testing through ***separation of concerns***.
- To facilitate reasoning about functional and quality properties at a higher-level than code.

Who

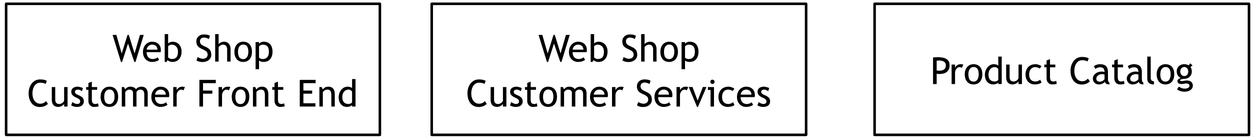
Software architects, development team, operation team, client and other stakeholders.


Web Shop Customer Front End
Customer Interface (REST)
Web Shop Customer Services
Customer Management
Order Management
Query Catalog
Customer Information System
Product Catalog
Order Processor
Order Change
Inventory Check
<<external>> Order Fulfillment
<<external>> Stock Inventory

Core Concepts

Functional Element (Component)

= a run-time element that perform specific functions in the system.



Interaction Element (Connector)

= an element that enables run-time interactions between functional elements, such as procedure call, RPC, REST interactions, interactions through a RESTful API, middleware for transmitting messages or events between functional elements.

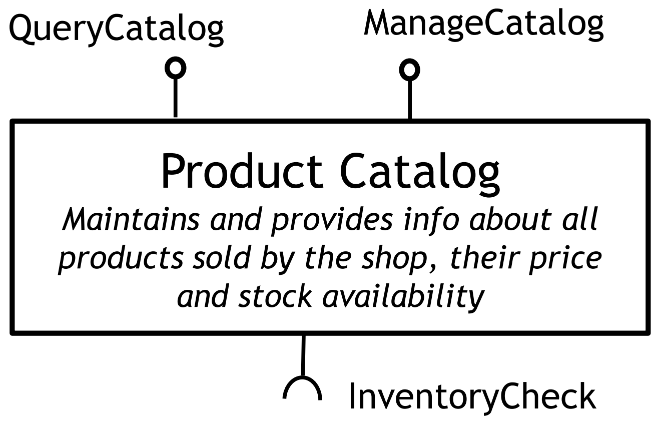
Functional Element (a.k.a. component)

- Definition: architectural element that
  - encapsulates a subset of the system's functionality or data
  - made accessible through well-defined interfaces
- Attributes
  - Responsibility: what information the element manages, what services it offers, what activities it initiates
  - Provided interfaces: specification of the services provided by the element (how it is seen from the outside)
  - Required interfaces: interfaces required by the element to fulfill its responsibilities

- Core Principle: Information hiding
  - Functional elements encapsulate secrets
  - The element's implementation is hidden from the outside

UML Notation: Components

<<component>> Product Catalog
responsibility *Maintains and provide info about all products sold by shop, their price and stock availability*
provided interfaces QueryCatalog ManageCatalog
required interfaces InventoryCheck


QueryCatalog
ManageCatalog
Product Catalog
Maintains and provides info about all products sold by the shop, their price and stock availability
InventoryCheck

Component’s secret: how the provided interfaces are implemented.

Interaction Element (a.k.a. connector)

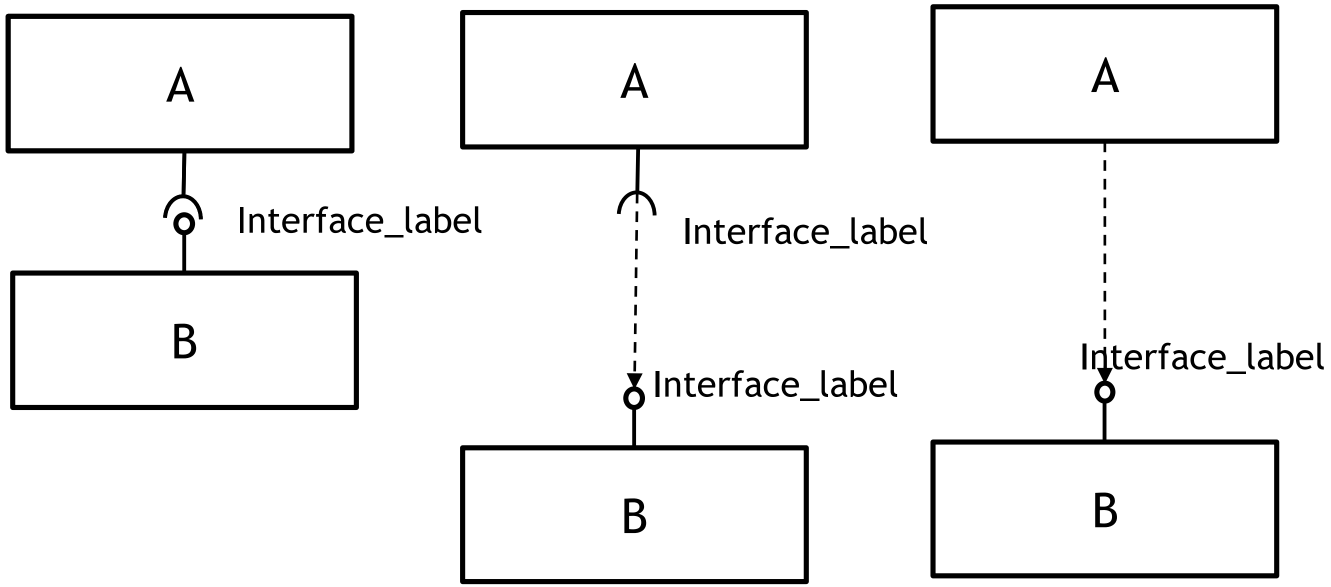
- Definition: architectural element that enables run-time communication and interactions between functional elements

  E.g.

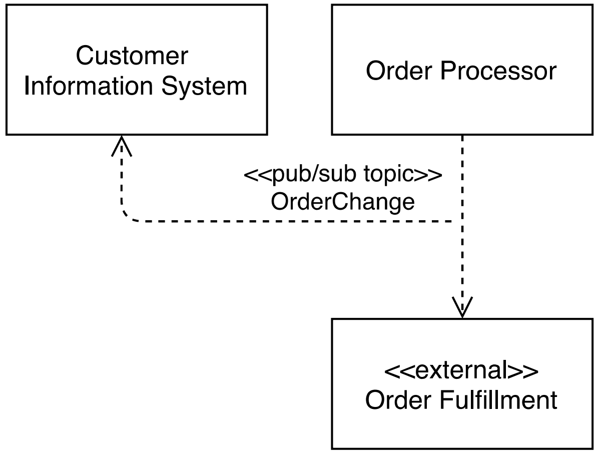
  - Procedure call (local or remote)
  - Pipes
  - Client-server middleware
    - RESTful APIs,
  - Event bus/Message broker

UML Notation: Procedure calls, API calls

Three notations, same meaning



UML Notation: Message-Oriented Interactions


Customer Information System
Order Processor
<<pub/sub topic>> OrderChange
<<external>> Order Fulfillment

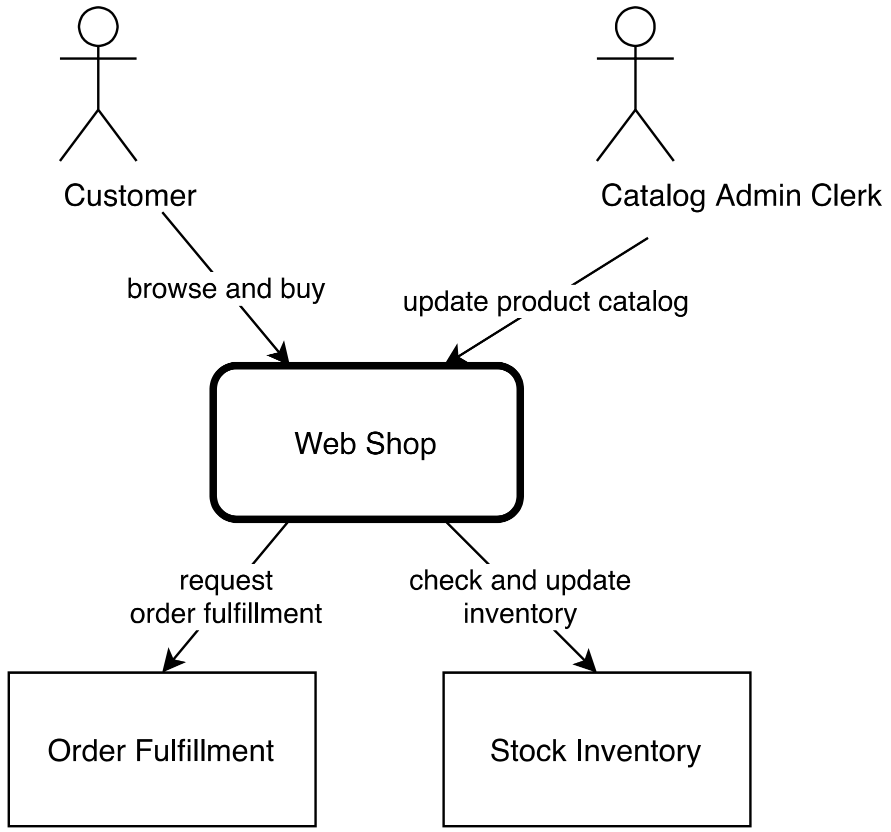
The order processor sends "order change" notifications to a message bus (not shown in the model) that forwards the notifications to all components that have subscribed to the "order change" topic.

Relation to Context Diagram

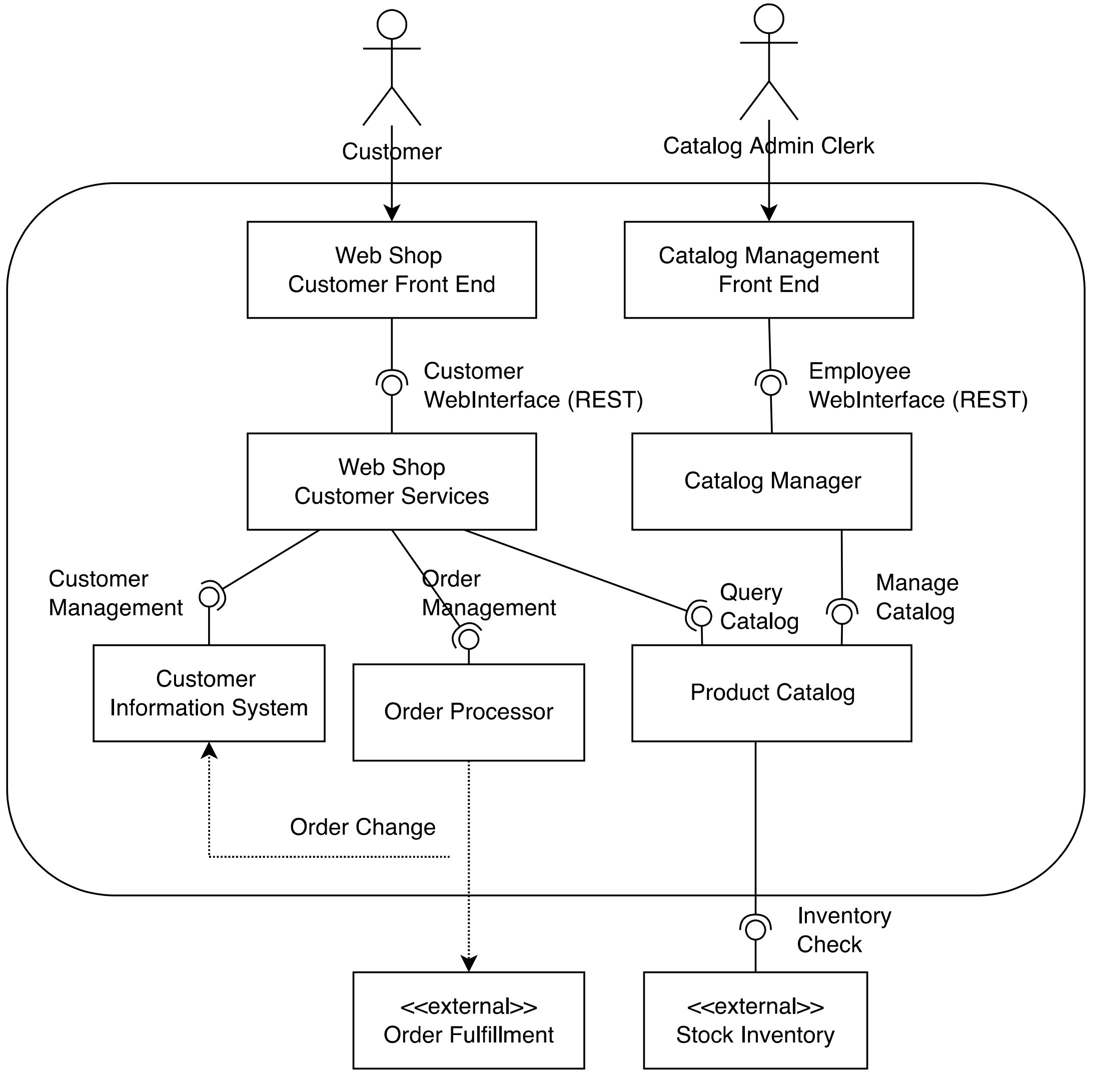
The functional view zooms inside Machine shown the context diagram.

Interactions with external systems must be the same.

Example: Context diagram for the web shop application



Web Shop: Functional View


Customer
Catalog Admin Clerk
Web Shop Customer Front End
Catalog Management Front End
Customer WebInterface (REST)
Employee WebInterface (REST)
Web Shop Customer Services
Catalog Manager
Customer Management
Order Management
Query Catalog
Manage Catalog
Customer Information System
Order Processor
Product Catalog
Order Change
Inventory Check
<<external>> Order Fulfillment
<<external>> Stock Inventory

Key Concerns for the Functional View

- Fit for purpose
  - The elements work together to provide the required functionality and quality (performance, extensibility, etc.)
- Well-defined responsibilities
  - Each functional element has clearly defined responsibilities
- Strong Cohesion (“single responsibility”)
  - Each functional element has a few, strongly related responsibilities.
- Weak Coupling (“few interactions”)
  - Functional elements have few interdependencies (few links to other functional elements).

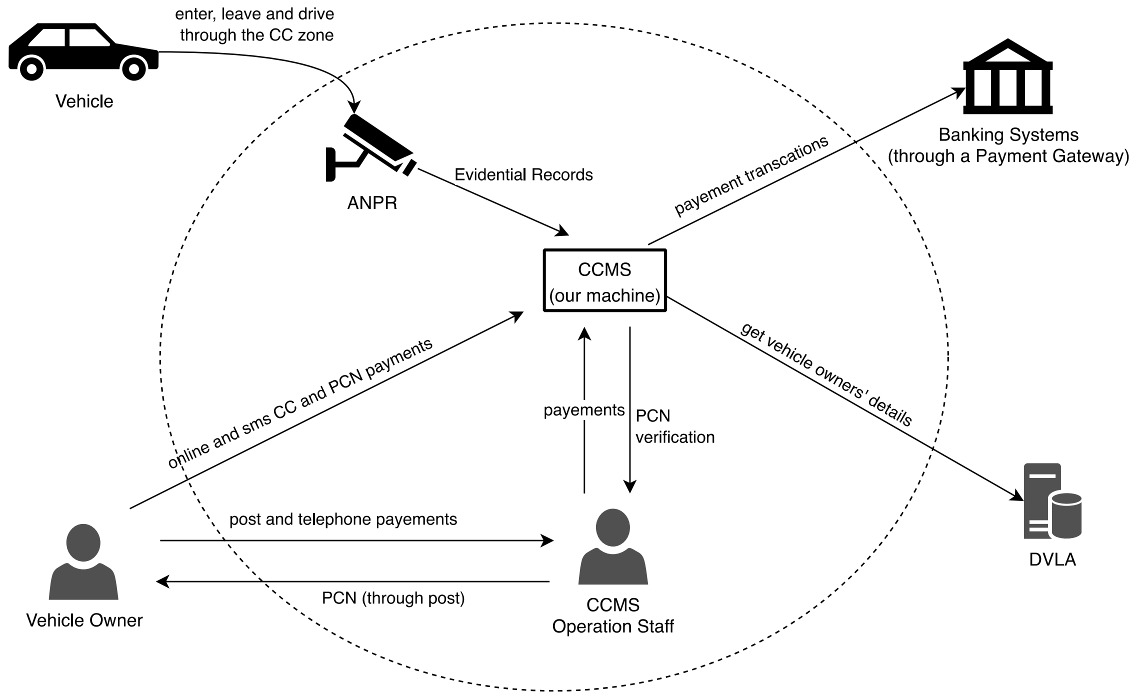
Guidelines for creating a Functional View

1. Start with the context diagram and a list of main functionalities
2. Draw a functional view for one functionality at a time
    1. Identify sub-tasks and responsibilities to achieve the functionality
    2. Create one functional element for each task/responsibility
    3. Define the provided and required interfaces for each functional element

    Hints:

    - Focus on tasks and responsibilities, not on technologies. Imagine you're setting up a 1950s corporation (components = departments; connectors = internal phone and mail systems).
    - Focus on **modularity**. Pay less or no attention to other quality requirements like security, availability, scalability. They will be considered later.
3. Group the functional views into a single diagram
    1. Consider simplifying by merging functional elements with similar responsibilities.
    2. Keep multiple functional views if a single view is too complex.

Exercise: Congestion Charge System



Vehicle
enter, leave and drive through the CC zone
ANPR
Evidential Records
payement transcations
Banking Systems (through a Payment Gateway)
CCMS (our machine)
get vehicle owners' details
DVLA
online and sms CC and PCN payments
payements
PCN verification
post and telephone payements
PCN (through post)
Vehicle Owner
CCMS Operation Staff

Main functionalities

1. Receive CC payments (online, sms, post and phone)
2. Receive ERs from ANPR cameras
3. Create, validate and send PCN when CC not paid

Try for yourself before looking at possible solution on Moodle.

Common Mistake #1: Poorly Defined Responsibilities

Avoid component names that are vague, non domain-specific, or refer to implementation technology only.

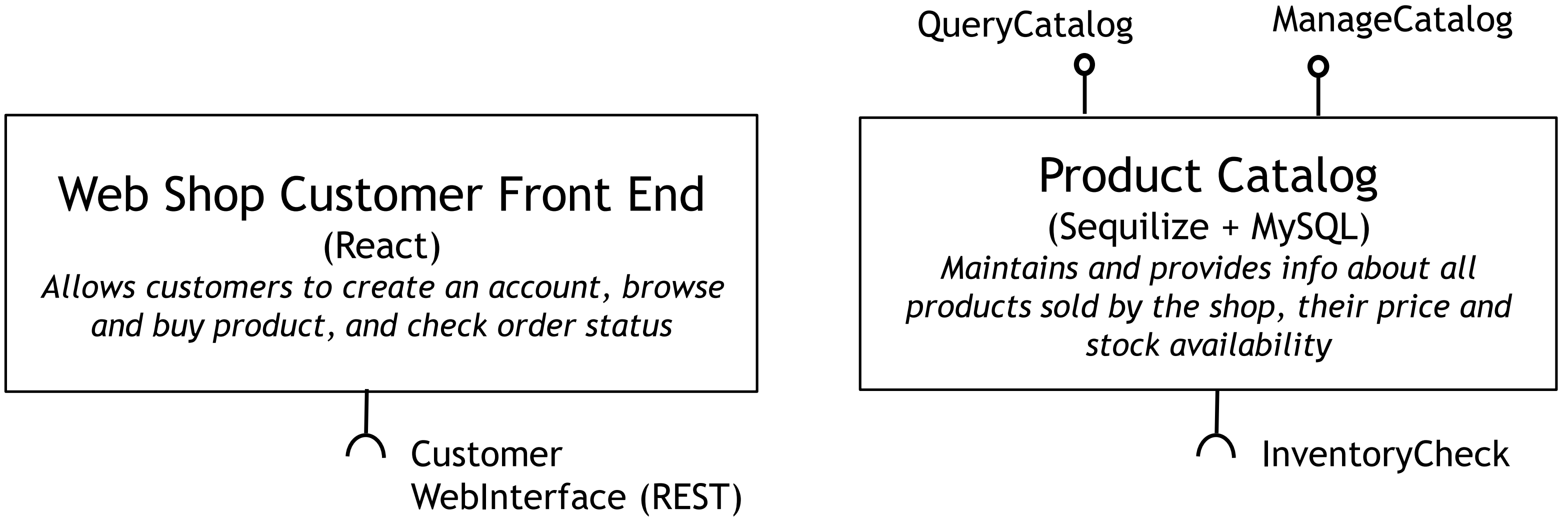
- React Front End
- Application Manager
- Web Shop Database

Use informative application-specific names

- Web Shop Customer Front End
- Web Shop Customer Services
- Product Catalog

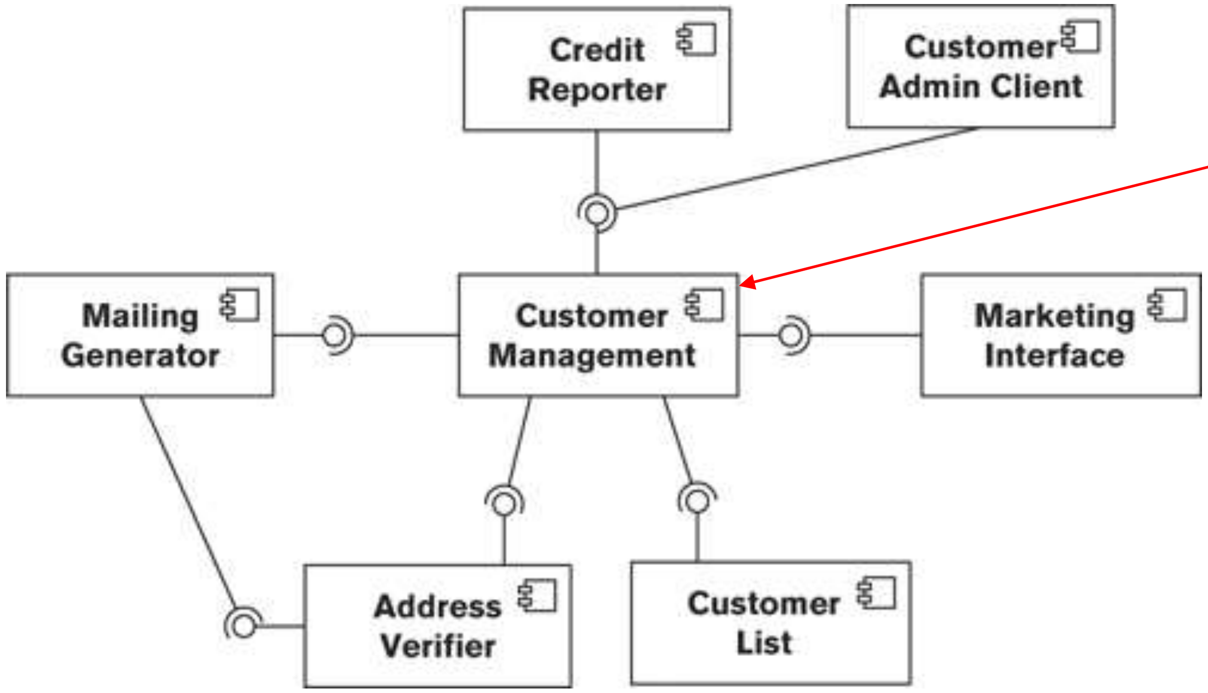
Extend the component name with a description of responsibility and implementation technology (examples next slide)

Tip: Annotate Components with Responsibility and Technology


QueryCatalog
ManageCatalog
Web Shop Customer Front End
(React)
Allows customers to create an account, browse and buy product, and check order status
Product Catalog
(Sequilize + MySQL)
Maintains and provides info about all products sold by the shop, their price and stock availability
Customer WebInterface (REST)
InventoryCheck

Common Mistake #2: Weak Cohesion

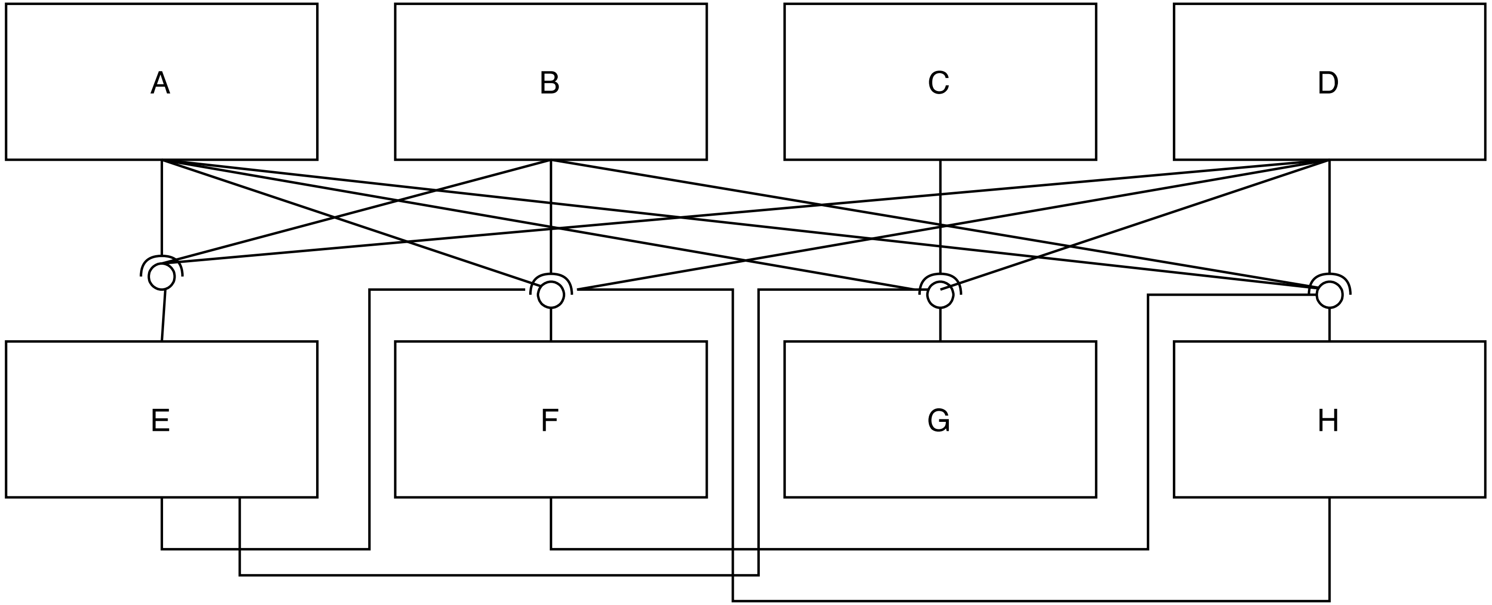
How to recognize it: a functional element with too many responsibilities and interactions (a.k.a a “God element”)



Too many responsibilities and interactions

Common Mistake #3: Strong Coupling

How to recognize it: Spaghetti architecture, too many dependencies



Common Mistake # 4: Wrong Level of Details

Too detailed (over-specification):

- Sign: include elements that are not architecturally significant (lower-level implementation classes)
- Consequences: hard to understand and analyze

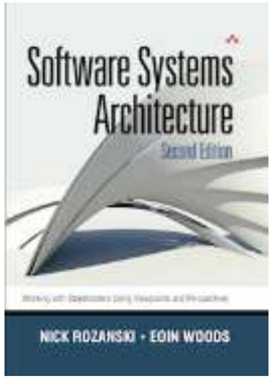
Not detailed enough (under-specification):

- Sign: Components are too coarse-grained, their responsibilities too broad
- Consequences: architecturally significant decisions remain vague or unaddressed

The Deployment Viewpoint

Chapter 21 of

N. Rozanski and E. Woods, *Software Systems Architecture : Working With Stakeholders Using Viewpoints and Perspectives, 2nd Ed.*, Addison-Wesley, 2012.



The Deployment Viewpoint

Defines the physical environment in which the system is intended to run, including

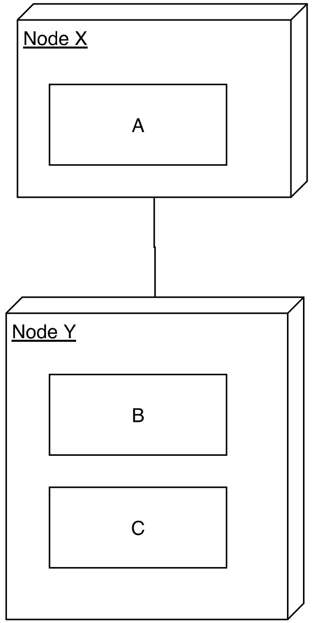
- the hardware or hosting environment (e.g., processing nodes, network interconnections, and disk storage facilities)
- the technical environment requirements for each type of processing node in the system, and
- the mapping of software elements to the runtime environment that will execute them

Readers: System administrators, developers, testers, and assessors.

Basic Concepts

- Node: a hardware platform on which software can be installed and executed and/or on which data may be placed.
- Link: a network connection between nodes.
- Deployment: the mapping from functional elements to nodes.

UML Notation: Deployment Diagrams



The model shows two nodes, Node X and Node Y, connected by a network link.

Functional element A runs on Node X

Functional elements B and C run on Node Y

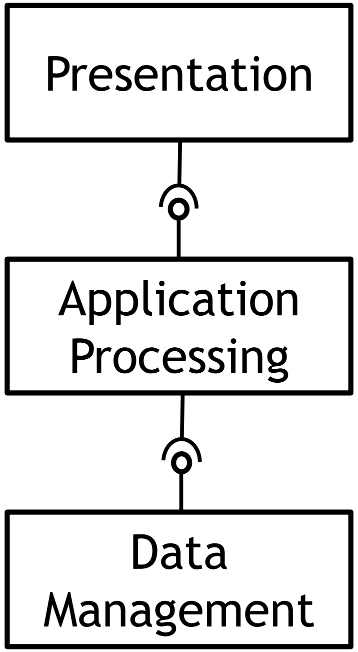
This is the most basic notation.

- You can also have nested nodes to specify execution environments (e.g. the OS)
- You can annotate nodes and links with their characteristics (e.g. cpu and ram, bandwidth)

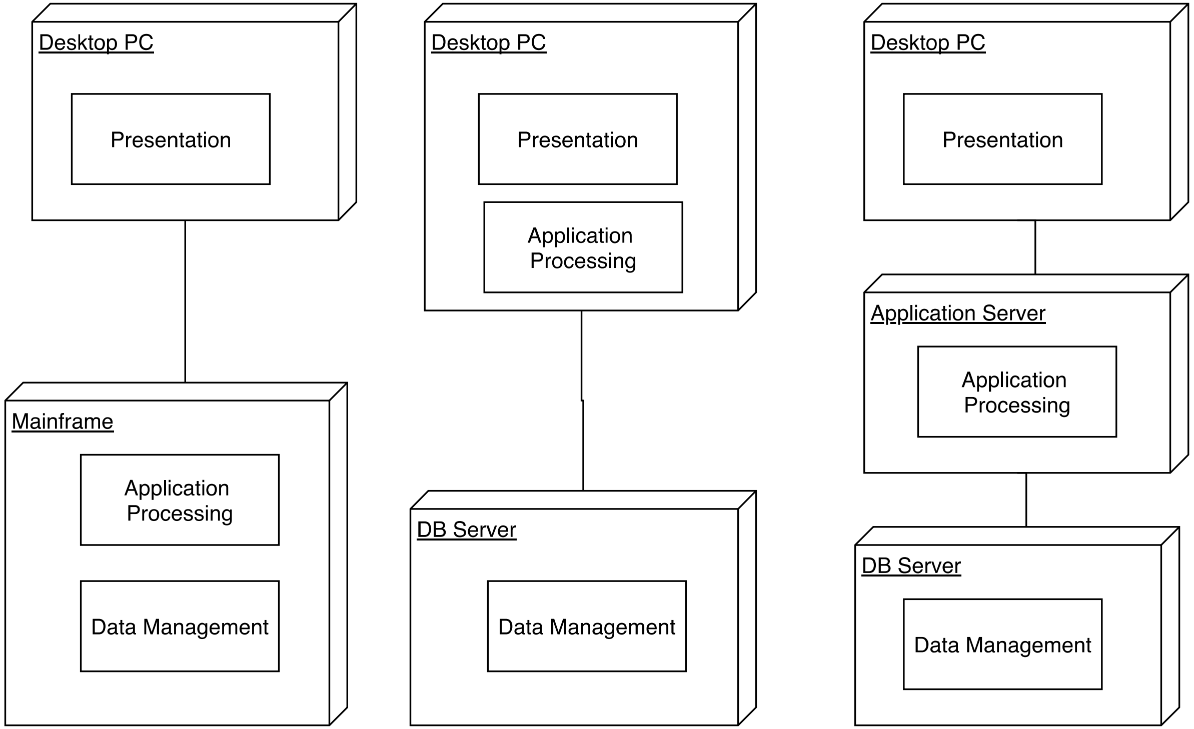
See https://www.uml-diagrams.org/deployment-diagrams.html

Example: Functional View of a Generic 3 Tier Architecture

- Presentation tier
  - responsible for capturing inputs and presenting outputs to users
- Application processing tier
  - responsible for the application main functionalities
  - Can itself be composed on several tiers
- Data management tier
  - responsible for storing and managing access to data

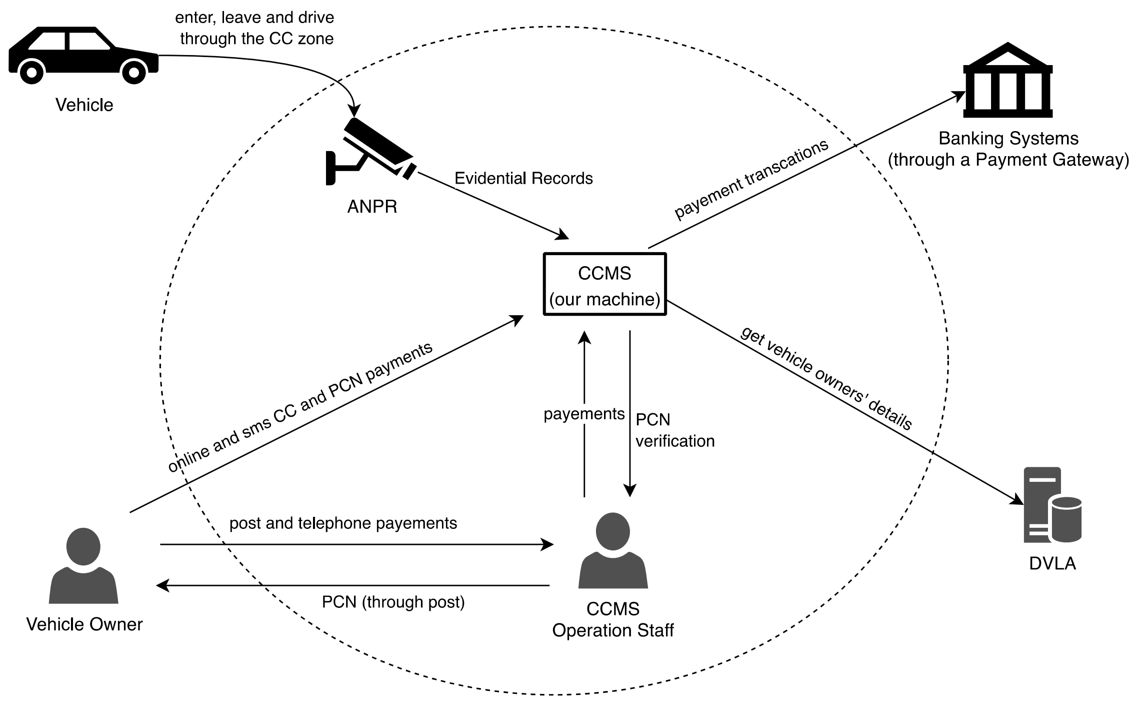


Three Possible Deployments



Real systems often many more functional elements and nodes.

Exercise: Congestion Charge System



enter, leave and drive through the CC zone
Vehicle
ANPR
Evidential Records
payement transcations
Banking Systems (through a Payment Gateway)
CCMS (our machine)
get vehicle owners' details
online and sms CC and PCN payments
payements
PCN verification
DVLA
post and telephone payements
PCN (through post)
Vehicle Owner
CCMS Operation Staff

Draw one plausible deployment view for the functional view we elaborated earlier

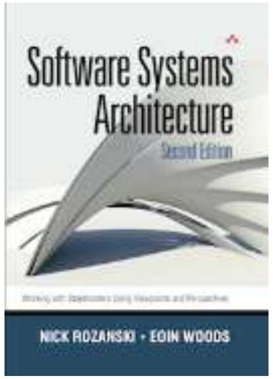
Note: deployment decisions affected by quality requirements such as performance, availability and cost.

Try for yourself before looking at possible solution on Moodle.

The Development Viewpoint

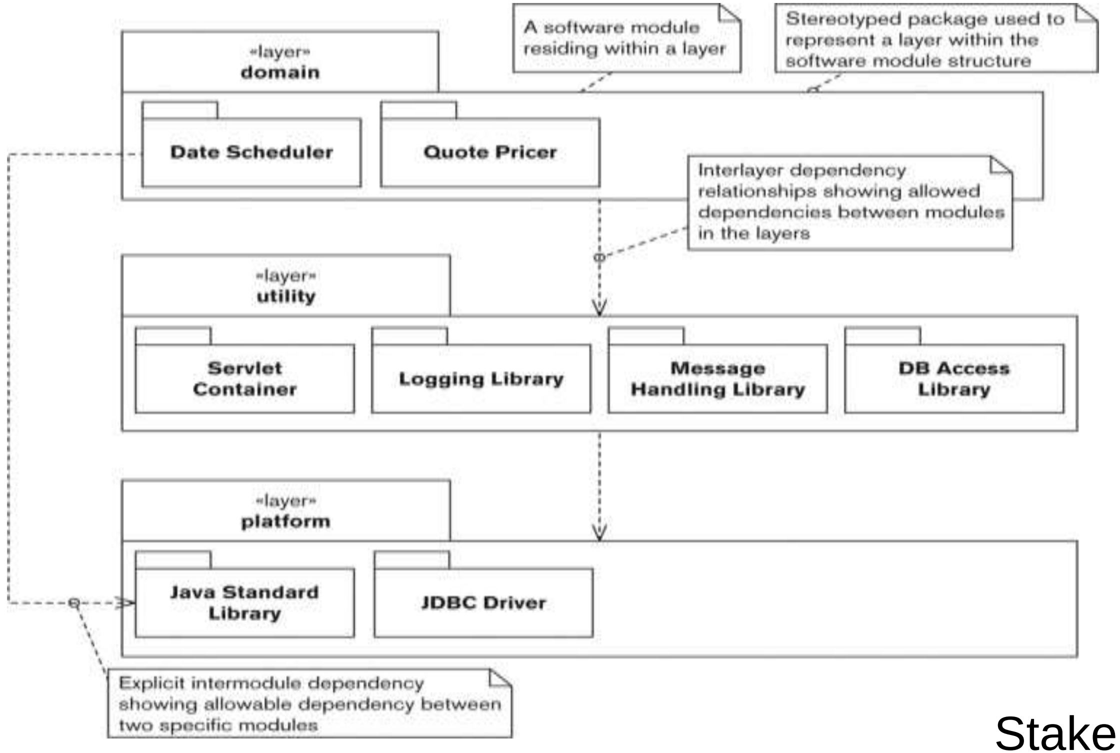
Chapter 20 of

N. Rozanski and E. Woods, *Software Systems Architecture : Working With Stakeholders Using Viewpoints and Perspectives, 2nd Ed.*, Addison-Wesley, 2012.



The Development View (UML Package Diagrams)

Describes the system organization into modules and their ***design-time*** relationships (Use, Import/Export)



A software module residing within a layer
Stereotyped package used to represent a layer within the software module structure
«layer» domain
Date Scheduler
Quote Pricer
Interlayer dependency relationships showing allowed dependencies between modules in the layers
«layer» utility
Servlet Container
Logging Library
Message Handling Library
DB Access Library
«layer» platform
Java Standard Library
JDBC Driver
Explicit intermodule dependency showing allowable dependency between two specific modules

Stakeholders: developers, testers

Module

Definition(s): a grouping of source files providing some independent, reusable functionality; a work unit for the software developers and testers.

Attributes

- Specification of provided interfaces
- Implementation code

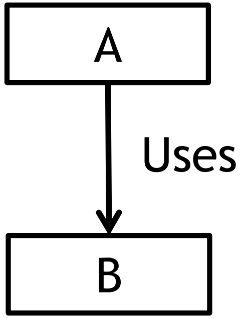
Relation to functional elements

- Functional elements execute code defined in modules
- A single module can include
  - part of the code executed by a functional element,
  - all of the code executed by a functional element, or
  - code that is executed by multiple functional elements

The "Use" Relation

D. Parnas, Designing software for ease of extension and contraction, ICSE 1978 and IEEE TSE 1979

Definition. A uses B means that the correct implementation of A (the fact that A satisfies its specification) depends on the presence of a correct implementation of B (the fact that B satisfies its specification)



A uses B means that:

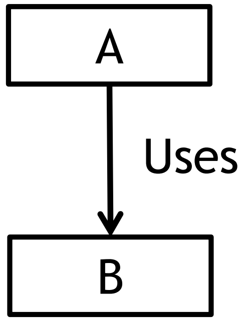
- Developers of A can assume B is present and satisfying its specification.

  They don't need to know anything about B's implementation, nor about any other module they don't use.

The “Use” Relation

D. Parnas, Designing software for ease of extension and contraction, ICSE 1978 and IEEE TSE 1979

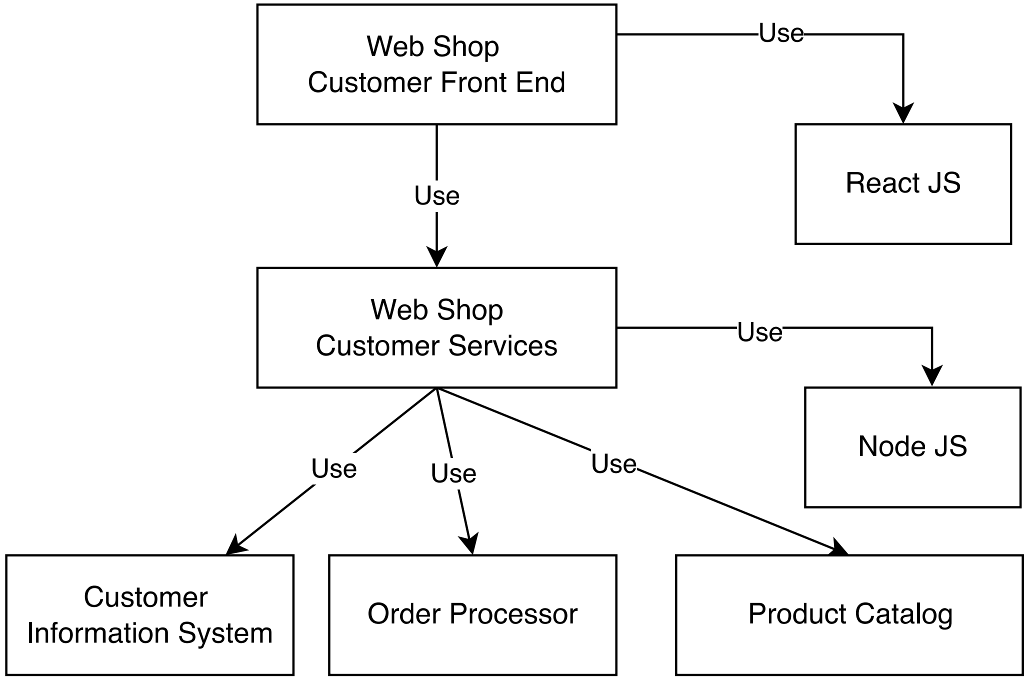
Definition. A uses B means that the correct implementation of A (the fact that A satisfies its specification) depends on the presence of a correct implementation of B (the fact that B satisfies its specification)



A uses B means that:

- Developers of A rely on B. They can assume B is present and correct.
- They need to know B’s specification, but not its implementation.
- Developers of B need to know B’s specification, but they don’t need to know anything about A.

Examples



A module in your code base might use

- another module in your code base
- a module in another code base
- a generic third-party library
- a framework

Use(A,B) is not Invokes(A, B)

Use(A, B) often coincides with an interaction from A to B in the functional view (component diagram) but not always.

Examples:

1. Inversion of Control in application frameworks

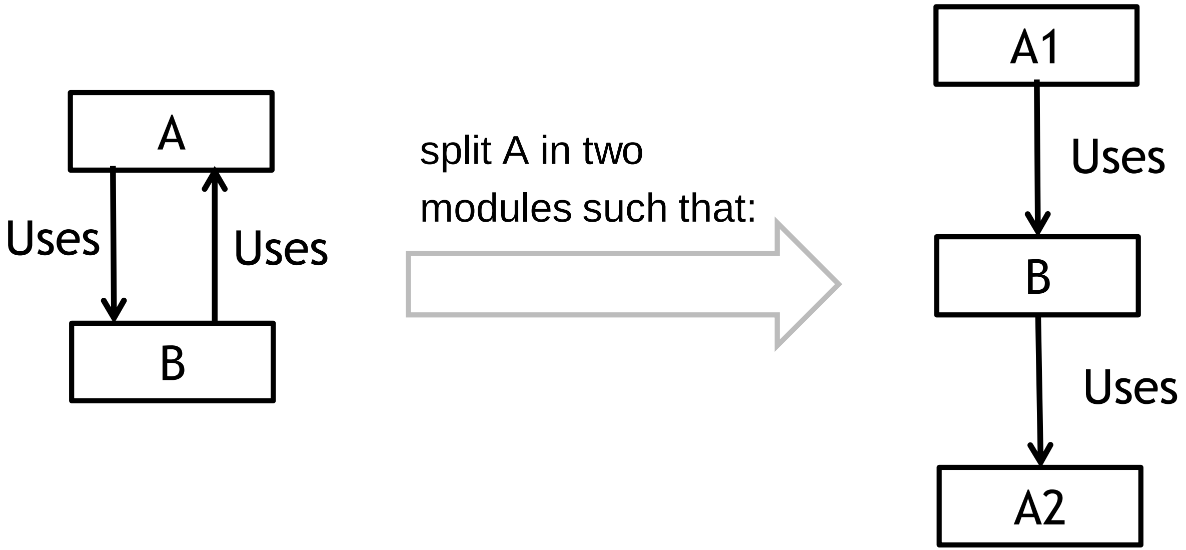
   An application specific module A uses the framework, but it is the framework that invokes methods in A.

2. B = Garbage Collector, Interrupt Handler

   Many modules depend on presence and correct functioning of the garbage collector and interrupt handler even though they never directly invoke them.

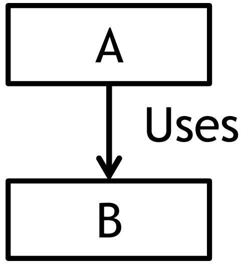
The USE relation should be acyclic

- Dependency cycles make the system hard to test and evolve
- Resolve cycles by “sandwiching” (Parnas, 1979)



Criteria for deciding that A uses B

1. A should be simpler to implement because it uses B.
2. B should not substantially more complex to implement because it cannot use A.
3. You can imagine system variants having B and not A.
4. You cannot imagine system variants having A and not some implementation of B.



Key Concerns for the Development View

- Modularity: Allow modules to be developed, tested and revised independently.

- Incrementality: Allow software to grow in small increments by
  - Identifying the *minimal* subsystem that might perform a useful service (minimum viable product)
  - Identifying a set of *minimal* extensions that add additional value to the system

- Product Families: Support the development of product variants

Architecture Models: Summary

Context Diagram
describes the software system's inputs and outputs with its environment (people, hardware devices and other software systems)

Functional Viewpoints
a run-time view of the system's components and connectors

Deployment Viewpoint
describes the system's networking and hardware infrastructure and where components execute

Development Viewpoint
a design-time view of the organisation of the system's code into modules with controlled module dependencies