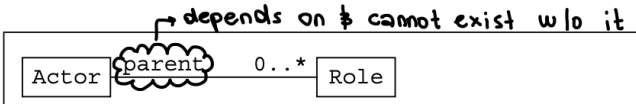
Técnicas de Diseño (75.10)

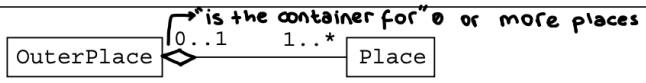
Domain Model

Collaboration Patterns



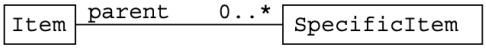
Use to model the participation of a person, organization, place, or thing in a context.

- An `actor` knows about zero to many `roles`, but typically takes on only one of each kind.
- A `role` represents a unique view of its `actor` within a context. The `role` depends on its `actor` and cannot exist without it.



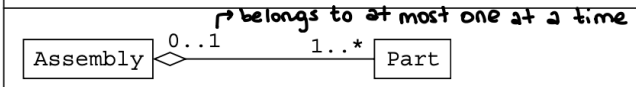
Use to model a hierarchy of locations where events happen.

- An `outer place` is the container for zero or more `places`.
- A `place` knows at most one `outer place`. The `place's` location depends on the location of its `outer place`.



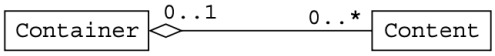
Use to model a thing that exists in several distinct variations.

- An `item` is the common description for zero to many `specific items`.
- A `specific item` knows and depends on one `item`. The `specific item's` property values distinguish it from other `specific items` described by the same `item`.



Use to model an ensemble of things.

- An `assembly` has one or more `parts`. Its `parts` determine its properties, and the `assembly` cannot exist without them.
- A `part` belongs to at most one `assembly` at a time. The `part` can exist on its own.

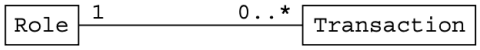


Use to model a receptacle for things.

- A `container` holds zero or more `content` objects. Unlike an `assembly`, it can be empty.
- A `content` object can be in at most one `container` at a time. The `content` object can exist on its own.

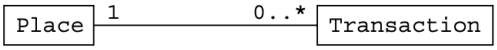
Use to model a classification of things.

- A `group` contains zero or more `members`. `Groups` are used to classify objects.
- A `member`, unlike a `part` or `content` objects, can belong to more than one `group`.



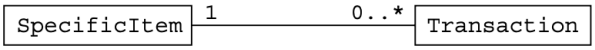
Use to record participants in events.

- A `transaction` knows one `role`, the doer of its interaction.
- A `role` knows about zero or more `transactions`. The `role` provides a contextual description of the person, organization, thing, or place involved in the `transaction`.



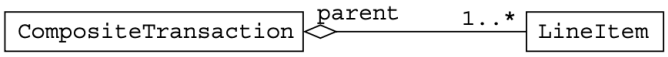
Use to record where an event happens.

- A `transaction` occurs at one `place`.
- A `place` knows about zero to many `transactions`. The `transactions` record the history of interactions at the `place`.



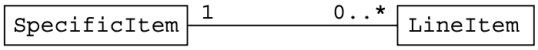
Use to record an event involving a single thing.

- A `transaction` knows about one `specific item`.
- A `specific item` can be involved in zero to many `transactions`. The `transactions` record the `specific item's` history of interactions.



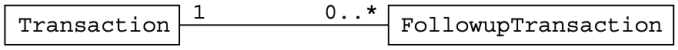
Use to record an event involving more than one thing.

- A composite transaction must contain at least one line item.
- A line item knows only one composite transaction. The line item depends on the composite transaction and cannot exist without it.



Use to record the particular involvement of a thing in an event involving multiple things.

- A specific item can be involved in zero to many line items.
- A line item knows exactly one specific item. The line item captures details about the specific item's interaction with a composite transaction.



Use to record an event that occurs only after a previous event.

- A transaction knows about some number of follow-up transactions.
- A follow-up transaction follows and depends on exactly one previous transaction.

Object Definition

- **D**efinition
  - name the class, indicate superclasses & specify interfaces exhibited
  - define variables & collaborations
- **I**nitialise
  - create construction method (parameters for property values & necessary collaborations)
  - sets properties to initial values & create collections for collective collaborations
- **A**ccess
  - property & collaborations accessors
  - *test* & *do* methods
- **P**rint
  - values of properties & collaborators
  - generic collaborators are asked by specific ones to be described

    - interacting entity collaborators are asked by events to be described
- **E**quals
    - if the receiving object is equal to another by comparing property values & collaborators
    - generic collaborators are asked by specific ones to be described
    - interacting entity collaborators are asked by events to be described
- **R**un
    - sample objects with property values & for collaborators

Solid Principles

Open-Closed Principle

The Principle

> 📜 Software entities (classes, modules, functions, etc.) should be **open for extension**, but **closed for modification**.

Open for extension → the behaviour of the module can be extended.

Closed for modification → the source code is inviolate.

Abstraction

Abstract base classes in which the unbounded group of possible behaviours ir represented by all their possible derivative classes.

Programs that conform to the open-closed principle are changed by adding new code, rather tan by changing the existing one, avoiding a cascade of changes.

Strategic Closure

The designer must choose the kinds of changes against which to close the design in order to considerate that not matter how *closed* a module could be, there'll always be some kind of change.

Using Abstraction to gain Explicit Closure

Ordering abstraction → policy which implies that given two objects it's possible to discover which ought to be drawn first.

Using a *Data Driven* Approach to Achieve Closure

Doesn't force changes in every derived class.

Heuristics & Conventions

Make all Member Variables Private

- they should be known only to the methods of the class that defines them
- when they change, every function that depends upon them must be changed
  - no function that depends upon a variable can be closed with respect to that variable
- methods of a class are not closed to changes in the member variables of that class
  - subclasses are closed against changes to those variables → **encapsulation**

No Global Variables

- no module that depends upon them can be closed against any other module that might write to them
- in cases where they have very few dependents or can't be used ir an inconsistent way they do little harm

RTTI is Dangerous

- run time type identification should be avoided
- the open-closed principle is violated whenever a new type of the class is derived

Liskov Substitution Principle

The Principle

> 📜 **Functions** that use pointers or references to **base classes** must be able to use **objects of derived classes** without knowing it.

If there is a function which doesn't conform to the LSP, then that function uses a pointer or reference to a base class, but most know about all the derivatives of that

base class. Such a function violates de *open-closed* principle because it must be modified whenever a new derivative of the base class is created.

Validity is not Intrinsic: when considering whether a particular design is appropriate or not, one must not simply view the solution in isolation but in terms of the reasonable assumptions that will be made by the users of that design.

What Went Wrong? (W^3): all derivatives must conform to the behaviour that clients expect of the base classes that they use.

Design by Contract:

- **preconditions** must be true in order for the method to execute
- the method guarantees that de **postcondition** will be true
- when using an object through its base class interface, the user knows only the pre & post conditions of the base class

Dependency Inversion Principle

Bad Design

A piece of software that fulfills its requirements and yet exhibits any of the traits:

- **Rigidity** → hard to change because it affects too many other part of the system
  - the impact of the change cannot be estimated when the extent of that cascade of change cannot be predicted → the cost of the change impossible to predict
- **Fragility** → when it's changed, unexpected parts of the system break
  - often the new problems are in areas unrelated with the changed one
  - decreases the credibility of the design and maintenance organisation → product's quality is unable to be predicted
- **Immobility** → it's not reusable in other application because it cannot be disentangled from the current one
  - the desirable parts of the design are highly dependent upon details that are not desired
  - the cost of the separation is higher than the cost of redevelopment of the design

The Principle

> 📜 **High level** modules should not depend upon **low level** modules. **Both** should depend upon **abstractions**.
>
> **Abstractions** should not depend upon **details**. **Details** should depend upon **abstractions**.

High level modules

- contains the identity of the application; yet when they depend upon the lower level modules, the changes made to the last ones can have direct effects upon them
- are wanted to be able to reuse
- *High level policy*
  - *the abstractions that underlie the application are the truths that don't vary when the details are changed*
  - *the abstraction must be isolated from the details of the problem, then the dependencies of the design must be directed so the details depend upon the abstractions*

Dependency Inversion can be applied wherever one class sends a message to another.

Layering

> "… all well structured object-oriented architectures have clearly-defined layers, with each layer providing some coherent set of services though a well-defined and controlled interface."

Use it for

- create reusable frameworks
- construct code resilient to change

Interface Segregation Principle

Deals with the disadvantages of *fat* interfaces (not cohesive) → they can be broken up into groups of member functions.

Separate Clients mean Separate Interfaces: clients exert forces upon their server interfaces.

The Principle

> 📜 **Clients should not** be forced to **depend** upon **interfaces** that they **don't use**.

When a client depends upon a class that contains interfaces that the client doesn't use -but that other clients do use- then that client will be affected by the changes that those other clients force upon the class.

Class Interfaces vs Object Interfaces

Clients of an object don't need to access it through the interface of the object; rather, they can access it through:

- delegation
- base class of the object (multiple inheritance)

Single Responsibility Principle

The Principle

> 📜 There should **never** be more than one reason for a class to **change**.

If a class assumes more than one responsibility there will be more than one reason for it to change.

The responsibilities become coupled which leads to fragile designs that break in unexpected ways when changed.

Responsibility: *a reason for change*.

Clean Code

Meaningful Names

- **Intention-Revealing Names**
  - the name should answer all the big questions
    - *why it exists*
    - *what it does*
    - *how it is used*
  - the problem isn't the simplicity of the code but the *implicitly* of it (the degree to which the context is not explicit in the code itself)
- **Avoid Disinformation**
  - avoid leaving false clues that obscure the meaning of code
    - do not refer to a group as a *List* unless it's actually a list
  - beware of using names which vary in small ways
  - using inconsistent spellings
- **Make Meaningful Distinctions**
  - it's not sufficient to add number series or noise words eve though the compiler is satisfied
  - noise words are meaningless distraction & redundant
- **Use Pronounceable Names**
  - if you can't pronounce it, you can't discuss it without sounding like an idiot
- **Use Searchable Names**
  - *the length of a name should correspond to the size of its scope*
    - if a variable or constant might be seen or used in multiple places in a body of code, it's imperative to give it a search-friendly name
- **Avoid Encodings**
  - encoded names are seldom pronounceable & easy to mis-type
- **Member Prefixes**
  - don't need to do it
  - classes & functions should be small enough to don't need them
  - people quickly learn to ignore prefix or suffix to see the meaningful part of the name

- **Interfaces & Implementations**
  - the preceding *I* is a distraction at best and too much information at worst
- **Avoid Mental Mapping**
  - readers shouldn't have to mentally translate names into other
- **Class Names**
  - should have noun or noun phrase names
  - shouldn't be a verb
- **Method Names**
  - should have verb or verb phrase names
  - accessors, mutators & predicates should be names for their value
  - when constructors are overloaded, use static factory methods with names that describe the arguments
- **Don't Be Cute**
  - if names are too clever they will be memorable only to people who share the author's sense of humor & only as long as these people remember the joke
  - choose clarity over entertainment
  - say what you mean, mean what you say
- **Pick One Word Per Concept**
  - pick one word for one abstract concept & stick with it
  - a consistent lexicon is a great boon to the programmers who must use the code
- **Don't Pun**
  - avoid using the same word for two purposes
- **Use Solution Domain Names**
  - use computer science (CS) terms
    - algorithm names
    - pattern names
    - math terms

        - etc
- **Use Problem Domain Names**
    - when there is no “programmer-eese” for the job, use the name from the problem domain
- **Add Meaningful Context**
    - place names in context for the reader by enclosing them in well-named classes, functions or namespaces
- **Don’t Add Gratuitous Context**
    - shorter names are generally better than longer ones, so long as they are clear
    - add no more context to a name than is necessary

Functions

- **Small**
    - 2 to 4 lines long
    - each one must led to the next in a compelling order
- **Blocks & Indenting**
    - 1 or 2 lines long
    - also implies that functions shouldn’t be large enough to hold nested structures
- **Do One Thing**
    - *Functions should do one thing. They should do it well. They should do it only.*
    - if a function does only those steps that are one level below the states of it’s name, then the function is doing one thing
- **Sections within Functions**
    - functions that do one thing can’t be reasonably divided into sections

One Level of Abstraction per Function

- **Reading Code from Top to Bottom → *The Stepdown Rule***

    - we want to be able to read the program as though it were a set of TO paragraphs, each of which is describing the current level of abstraction & referencing subsequent TO paragraphs at the next level down

- **Switch Statements**
    - make sure that each switch statement is buried in a low-level class & is never repeated
    - they can be tolerated if they
        - appear only once
        - are used to create polymorphic objects
        - are hidden behind an inheritance relationship
- **Use Descriptive Names**
    - the smaller & more focused a function is, the easier it is to choose a descriptive name
    - use the same phrases, nouns & verbs in the function names you choose for your modules

Function Arguments

- **niladic (0)**
    - ideal
- **monadic (1)**
    - asking a question about the argument
    - operating on the argument
    - transform the argument into something else and returning it
    - **event**
        - there is an input argument but no output argument
- **dyadic (2)**
    - harder to understand than a monadic
    - they come at a cost & should take advantage of what mechanisms may be available to convert them into monads
- **triadic (3)**

    - should be avoided
    - significantly harder to understand than dyads
    - the issues of ordering, pausing & ignoring are more than doubled
- **polyadic (>3)**
    - requires very special justification & shouldn't be used anyway

Arguments…

- are hard
- take a lot of conceptual power
- are a different level of abstraction than the function name & forces to know a detail that isn't important at that point
- even harder from a testing POV

Flag Arguments

- are ugly
- complicates the signature of the method proclaiming that this function does more than one thing (it does different thing depending on the flag's value)

Argument Objects → when a function seems to need more than 2 or 3 arguments, it is likely that some of those arguments ought to be wrapped into a class of their own.

Argument Lists → if the variable arguments are all treated identically, then they are equivalent to a single argument type of *List*.

Verbs & Keywords → explain the intent of the function & the order & intent of the arguments.

- monad: the function & argument should form a nice verb/noun pair

Had No Side Effects

- avoid promising doing one thing but do other hidden things
- make unexpected changes to
    - the variables of its own class
    - the parameters passed into the function or to system globals

- devious & damaging mistruths that often result in strange temporal couplings & order dependencies

Output Arguments

- are harder to understand than *input* → we are not used to expect information to be going out through the arguments
- should be avoided
- if the function must change the state of something, the state of its owning object must be changed instead

Command Query Separation

- functions should either do something or answer something but not both

Prefer Exceptions to Returning Error Codes

- returning error codes violates the command query separation
- promotes commands being used as expressions in the predicates of *if* statements
- lead to deeply nested structures

using exceptions instead make the error processing code can be separated from the happy path code and can be simplified (*try-catch* statement)

Extract try-catch Blocks → out into functions of their own

Error Handling Is One Thing → a function that handles errors should do nothing else.

Don’t Repeat Yourself

- it bloats the code & will requiere four-fold modifications should the algorithm ever have to change
- it’s a four-fold opportunity for an error of omission
- may be the root of all evil in software

Comments

The proper use of comments is to compensate for our failure to express ourself in code. We must have them because we cannot always figure out how to express ourselves without them.

Comments Do Not Make Up for Bad Code: rather than spend your time writing comments that explain the mess you've made, spend if cleaning that mess.

Good Comments

- **Legal Comments**
  - refer to a standard license or other external document rather tan putting all the terms & conditions into the comment
- **Informative Comments**
  - basic information (e.g.: return value of an abstract method, time & date format, etc.)
- **Explanation of Intent**
  - provide the intent behind a decision
- **Clarification**
  - translate the meaning of some obscure argument or return value into something that's readable
  - take care that there is not better way & then take even more care that they are accurate
- **Warning of Consequences**
  - warn other programmers about certain consequences
- **TODO Comments**
  - *TODO* → jobs that the programmer thinks should be done but for some reason can't do at the moment
- **Amplification**
  - amplify the importance of something that may otherwise seem inconsequential

Bad Comments

- **Mumbling**
  - plopping in a comment just because it feels like it should be written or because the process requires it

    - any comments that force to look in another module for the meaning of that comment has failed to communicate and is not worth the bits it consumes
- **Redundant Comments**
    - not more informative than the code
    - not justify the code or prove intent or rationale
    - not easier to read than the code
    - less precise than the code & entices the reader to accept that lack of precision in lieu of true understanding
- **Misleading Comments**
    - a statement in a comment that isn't precise enough to be accurate
- **Mandated Comments**
    - it's silly to have a rule that says that every function must have a doc related or every variable must have a comment
    - clutter up the code
    - propagate lies
    - lend to general confusion & disorganisation
- **Journal Comments**
- **Noise Comments**
    - restate the obvious & provide no new information
- **Scary Noise**
    - redundant noisy comments written out of some misplaced desire to provide documentation
- **In Replacement of a Function or a Variable**
- **Position Markers**
    - use them sparingly & only when the benefit is significant, otherwise they'll fall into the background noise & be ignored
- **Closing Brace Comments**
- **Commented-Out Code**

    - others who see that commented-out code won't have the courage to delete it, they'll think it is there for a reason & is too important to delete
    - delete code, good source code control system will remember it for us
- **Nonlocal Information**
    - make sure it describes the code it appears near
    - don't offer systemwide information in the context of a local comment
- **Too Much Information** (e.g.: historical information)
- **Unobvious Connection**
    - the connection between a comment & the code it describes should be obvious
    - the purpose of a comment is to explain code that does not explain itself, but not needs its own explanation
- **Function Headers**
    - a well-chosen name for a small function that does one thing is better than a comment header

Refactoring

Any fool can write code that a computer can understand. Good programmers write code that humans can understand.

Starting Program

> 💡 *When you find you have to add a feature to a program, & it's code is not structured in a convenient way to add the feature, first refactor the program to make it easy to add the feature, then add the feature.*

First Step

> 💡 *Before you start refactoring, check that you have a solid suite of tests. These must be self-checking.*

Decomposing & Redistributing the Statement Method

- decompose methods into smaller pieces
- smaller pieces of code tend to make things more manageable

> 💡 *Refactoring changes the programs in small steps. If you make a mistake, it's easy to find the bug.*

Architect Software Patterns

Micro Kernel

- add additional applications features as plug-ins to the core application
- provide extensibility as well as feature separation

Components

Core System

- main application logic

Plug-in Modules/Servers

- additional functionality

Servicies

Internal: extensibility for the core functionality of the core system

External: implement abstractions over the core functionality by consuming its services, the core doesn't know about the existence of these servers

Usage

Clients make use of the core by using the external servers, generally through an adapter.

Adapter → acts as a middleman between the external server and the client in order to avoid coupling between the client and the external server.

Is generally used for **systems that have a stable core functionality, but it needs to interact with different systems that can change over time**.

MVC

Is commonly used for **developing user interfaces**

- divides the responsibilities of the logic of the application from the views that the user sees
- separates the domain logic from the views, making them more decoupled (a model doesn't know of the existence of any view in particular)

Components

Model

- central component of the pattern
- manages the data and the domain logic for the application

Views

- any representation of the information that the model holds

Controller

- accepts inputs & converts it to command for the model or the view
- controls the operations & the interactions between the user and the data

Layers

- components are organised into horizontal layers
- one layer only talks with the layers below if any
- leads components to be are modularised & decoupled from each other
- communication between layers must be in a standardised way
- generally they are
  - **Presentation Layer**
  - **Services Layer**
    - may or not exists
    - multiple levels can also exist
  - **Domain Layer**
    - one or more levels

- **Persistence Layer**
- **Database Layer**
  - sometimes represented at the same levels as persistency

Pipe & Filter

- data architecture pattern
- useful when there exists the need of **transforming and analysing data in multiple levels**
- levels of data processing can be added or removed relatively easily

Broker

- middleman between two **different pieces of software that need to communicate between each other, but without the exact knowledge of their mutual existence**
- makes it easier to change the modules without affecting the components that talk with it
- adds the overhead of the middleman

The *4+1* View Model of Software Architecture

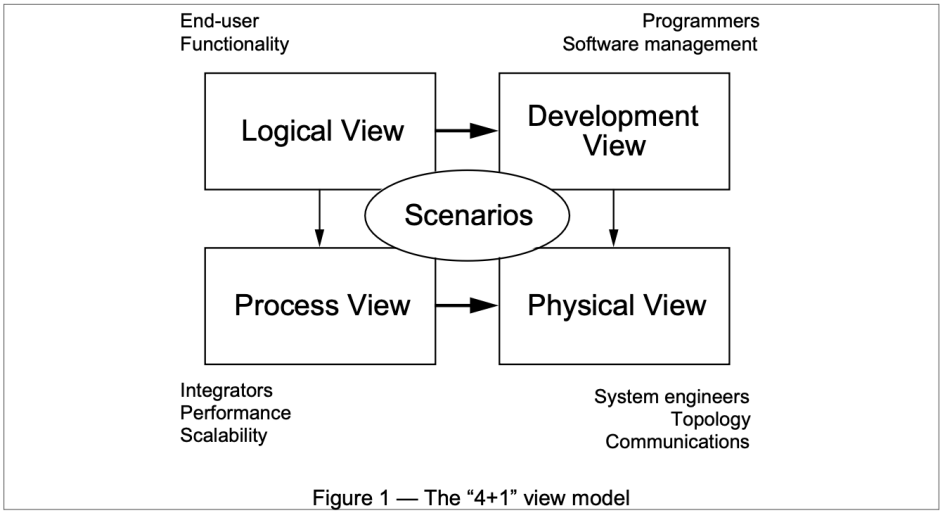


Figure 1 — The “4+1” view model

Logical View

Object-Orientated Decomposition

- supports the functional requirements
  - what the system should provide in terms of services to its users
  - identify common mechanisms & design elements across the various parts of the system
- the system is decomposed into a set of key abstractions in the form of objects or object classes

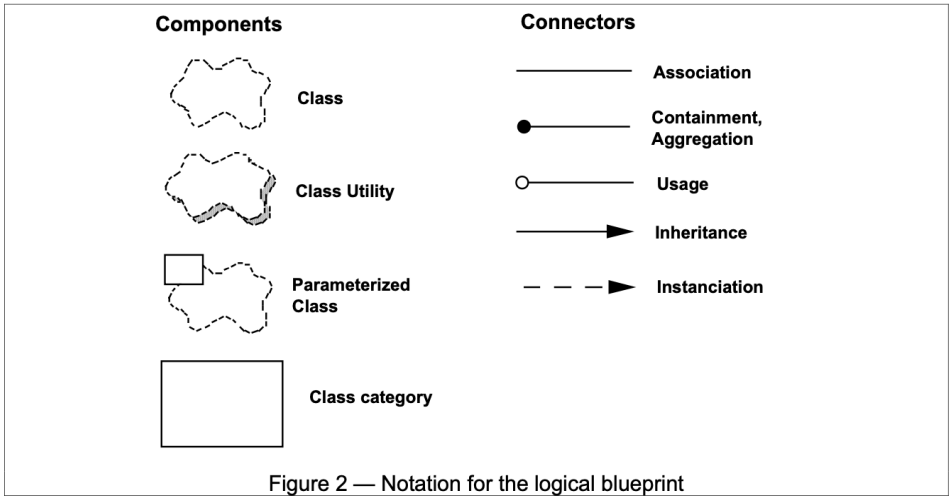


Figure 2 — Notation for the logical blueprint

Process View

Process Decomposition

- non-functional requirements
  - performance
  - availability
- address issues of
  - concurrency & distribution
  - system’s integrity
  - fault-tolerance
  - how the main abstractions from the logical view fit within the process architecture

- can be described at several levels of abstraction, each one addressing different concerns
  - at the highest, the process architecture can be viewed as a set of independently executing logical networks of communicating programs (called *processes*)
  - **process**
    - grouping of tasks that form an executable unit
    - represent the leve at which the process architecture can be tactically controlled
- software is partitioned into a set of independent *tasks*
  - **task**: separate thread of control that can be scheduled individually on one processing node
  - **major tasks** → architectural elements that can be uniquely addressed
  - **minor tasks** → additional tasks introduced locally form implementation reasons

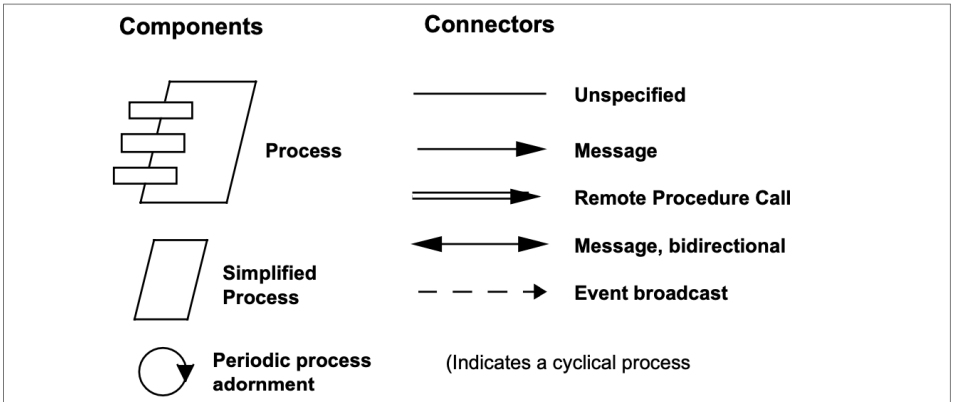


Figure 4 — Notation for the Process blueprint

Development Architecture

Subsystem Decomposition

- focuses on the actual software module organisation on the development environment
- software is packaged in small chunks that can be developed by one or a small number of developers

- subsystems are developed in a hierarchy of layers, each one providing a narrow & well-defined interface to the layers above
- serves as the basis for establishing a line-of-product
  - requirement allocation
  - allocation of work teams
  - cost evaluating & planning
  - monitoring the progress of the project
  - reasoning about software reuse
  - portability & security

Rules

- **Partitioning**
- **Grouping**
- **Visibility**

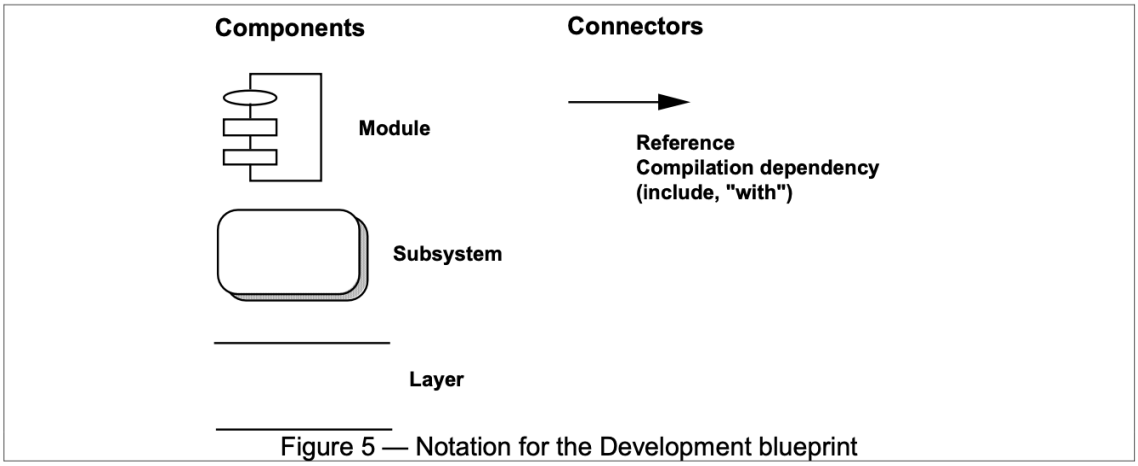


Figure 5 — Notation for the Development blueprint

The Physical Architecture

Mapping the Software to the Hardware

- takes into account primarily the non-functional requirements of the system
  - availability
  - reliability (fault-tolerance)
  - performance (throughput)

    - scalability
- the software executes on a network of computers or processing nodes
- the elements identified (networks, processes, tasks & objects) need to be mapped onto the nodes
- the mapping of the software to the nodes needs
    - to be flexible
    - have a minimal impact on the source code

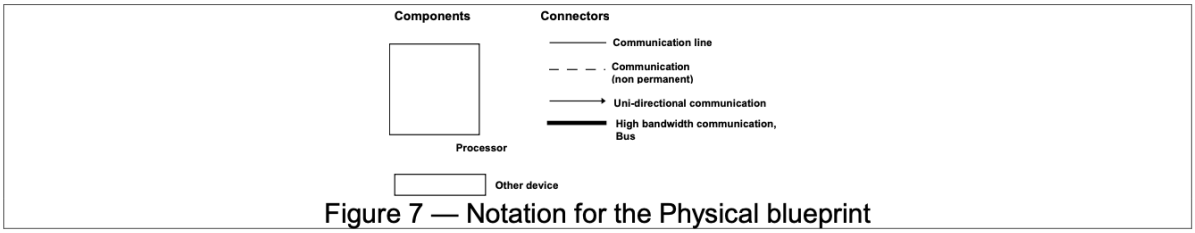


Figure 7 — Notation for the Physical blueprint

Scenarios

Putting it all Together

- instance of more general use cases
- abstraction of the most important requirements

Correspondence Between the Views

- views are not orthogonal or independent of each other (elements are connected)
- **Logical view** motivates design decisions in…
    - **Development view**
    - **Process view**
    - design in these two should support the different operations between the objects present in the system
- **Development** and **Process** views affect the **Physical view**
    - decisions at the time of how the process should communicate, and the implementation of the requirements will affect the physical design of the system

Quality Attributes

- *measurable or testable property of a system that is used to indicate how well the system satisfies the needs of its stakeholders*
- *do not indicate if the system answers the requirements of the client (that should be a given) but they measure how well the system answers the requirements*

Parts

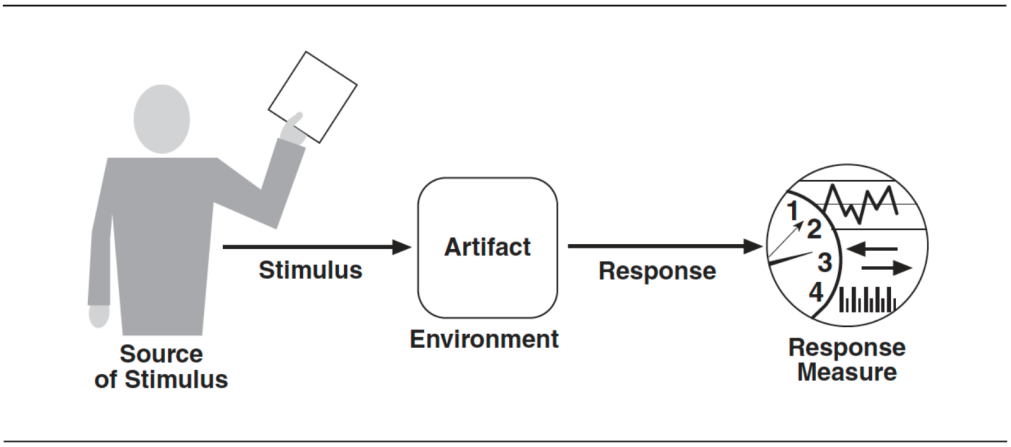


FIGURE 4.1 The parts of a quality attribute scenario

- **Source of Stimulus**
  - entity that interacts with the system being measured
- **Stimulus**
  - action that produces an appropriate response in the system
- **Environment**
  - state of the system at the moment of the stimulus
- **Artifact**
  - the system (or part of it)
- **Response**
  - measurable activity that is triggered by the stimulus
- **Response Measure**
  - measurement taken for the response
  - it varies depending of the Quality Attribute being tested

Availability

How much of the time the system is *up*

up meaning may be that if a system is running but the responses are taking too long or not being generated at all, it may not be **up**.

Having great availability requires redundancy in the infrastructure → the cost of the whole system goes up.

Reliability

Ability to perform its intended functions without failure, errors, or disruptions & deliver accurate and consistent results under expected and unexpected conditions

It’s not the same as availability, but a reliable system will be generally more available.

Interoperability

How easy is it for a system to interact with others

Modifiability

How easy is it to modify a system given a change in the requirements

Performance

Ability to achieve the expected time-response

Usability

How easy is for a user to use the software or application

Security

Faculty of protecting data against no authorised processes & guarantee access to whom who have authorisation