Template for specific development (Contract Id)

DSA Assignment Design Documentation

Table of Contents

[1 Introduction 4](#_Toc476588813)

[1.1 Purpose 4](#_Toc476588814)

[1.2 Scope 4](#_Toc476588815)

[2 SYSTEM OVERVIEW 5](#_Toc476588816)

[2.1 SYSTEM CHARACTERISTICS 5](#_Toc476588817)

[4 SYSTEM DESIGN 5](#_Toc476588818)

[4.3 Documentation Standards 5](#_Toc476588819)

[4.1 Naming conventions 6](#_Toc476588820)

[4.2 Programming Standards 6](#_Toc476588821)

[4.3 Software development tools 6](#_Toc476588822)

[4.4 Outstanding Issues 7](#_Toc476588823)

[4.5 Decomposition Description 7](#_Toc476588824)

[5 .Component Description 8](#_Toc476588825)

[Component Identifier 9](#_Toc476588826)

[Type 9](#_Toc476588827)

[Purpose 9](#_Toc476588828)

[Function 9](#_Toc476588829)

[Subordinates 9](#_Toc476588830)

[Dependencies 9](#_Toc476588831)

[Document Control 11](#_Toc476588832)

**Document the code and the system**

**Technical documentation (code, functions, configuration, database, administration etc.) is the priority here. To achieve maximum marks you may also want to analyse/explore tools such as PHPDocumentor or some of the other open-source debugging/documentation tools.**

# Introduction

1. This section should provide an overview of the entire document and a description of the scope of the system and its intended usage. The scope should also describe external interfaces to the system, external dependencies and provide a brief overview of the ‘characteristics’ of the system, commenting on aspects such as real-time use, security considerations, concurrency of users etc.
2. The operating system, development language to be used for the system development and any COTS packages that will be used, such as databases etc. should also be referred to in the introduction. This should be sufficient for the casual reader to gain a good appreciation of the key building blocks of the system. The reader should also be introduced to the security measures that need to be included within the system. Where strong authentication models are required this may need to show how aspects such as authentication of users may need to be implemented using PKI for example.

## Purpose

1. This section should:

describe the purpose of this document;

specify the intended readership of this document.

## Scope

1. This section should:

identify the products to be produced;

explain what the proposed system will do (and will not do, if necessary);

define relevant benefits, objectives and goals as precisely as possible;

define any security risks associated with the system;

Be consistent with similar statements in higher-level specifications, if they exist.

## 2 SYSTEM OVERVIEW

#1 This section should briefly introduce the system context and design, and discuss the background to the project. This section may also summarise the costs and benefits of the selected architecture, and may refer to feasibility studies and prototyping exercises. This section should also describe how the design proposed aligns with the IDA Architecture Guidelines and makes use of the outputs of IDA Horizontal Actions and Measures (HAMs).

## 2.1 SYSTEM CHARACTERISTICS

#2 The description of the system should be given in terms of the Architecture of the solution that is being implemented with high level data flows described to set the context of the system, i.e. to look at its external interfaces. This section should also set out to ‘characterise’ the system describing aspects of its operation that indicate if the system has, inter alia:

 to operate in real-time or in bursts, linked to month-end reporting, for example

 the nature of the interface to the users of the system

 many concurrent users

 to be highly resilient or fault tolerant

 to provide security features to protect data

 to be scaleable and easily maintainable in the future

 to have any special back-up facilities to protect important data.

code, functions, configuration, database, administration

# SYSTEM DESIGN

## 4.3 Documentation Standards

1. For a software implementation, this section should contain the standard module header (if necessary) and contain instructions for its completion. In addition this section should define or reference guidelines on the ration of lines of code to comment statements. It may be that these rules highlight specific areas of code where the commentary should be literally line-by-line, as this is a particularly difficult area. Other areas, which may be less difficult, could be commented on a ratio of five lines of code to one line of commentary.

## Naming conventions

1. This section should explain all naming conventions used, and draw attention to any points a maintenance programmer would not expect. A table of the filetypes and the permitted names or extensions for each is recommended for quick reference.
2. Conventions for naming files**,** programs, modules, and possibly other structures such as variables and messages, should all be documented here.

## Programming Standards

1. This section should define the project programming standards. Whatever languages or standards are chosen, the aim should be to create a convenient and easily usable method for writing good-quality software. If an application development tool is used there may be other conventions that need to be defined, e.g. colour schemes.
2. When programming in any new language, a standard for its use should be written to provide guidance for programmers. This standard may be referenced or included here.
3. Where there are external interfaces, the programming standards for the interfaces required should be referenced.
4. In general, the programming standard should define a consistent and uniform programming style. Specific points to cover are:

modularity and structuring;

headers and commenting;

indenting and layout;

library routines to be used;

language constructs to use;

language constructs to avoid.

## Software development tools

1. This section should list the tools chosen to assist software development, including testing. The actual software chosen will be heavily dependent upon the language in which the system will be implemented.
2. The list may include:

an application development too;

a configuration manager / builder;

HTML authoring tools;

a word processor for documentation;

a tool for drawing diagrams;

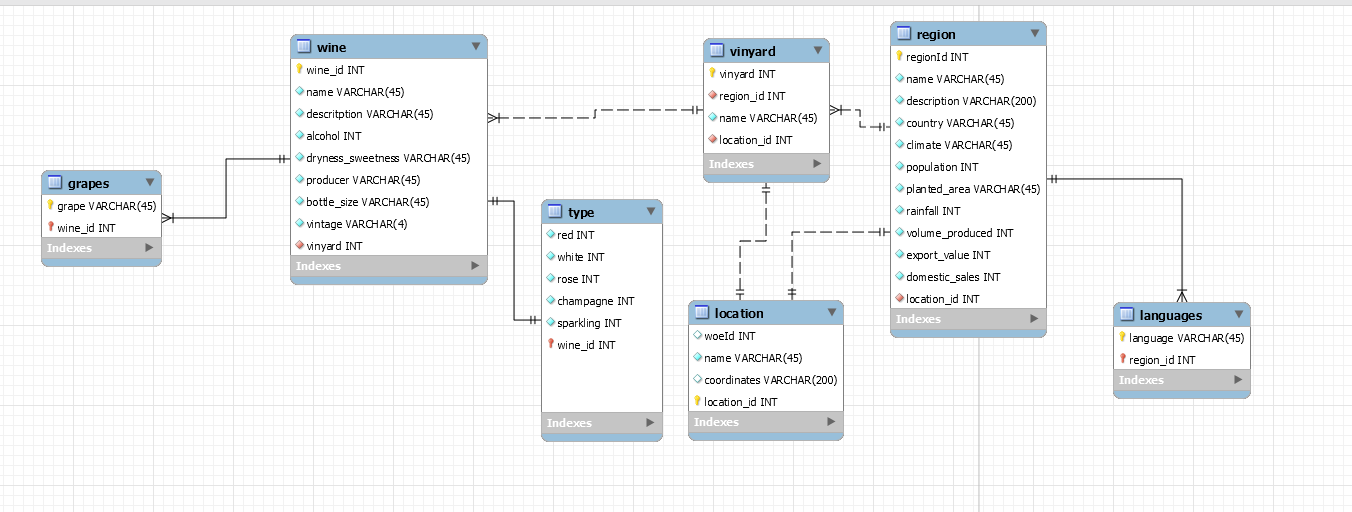
automated testing tools.

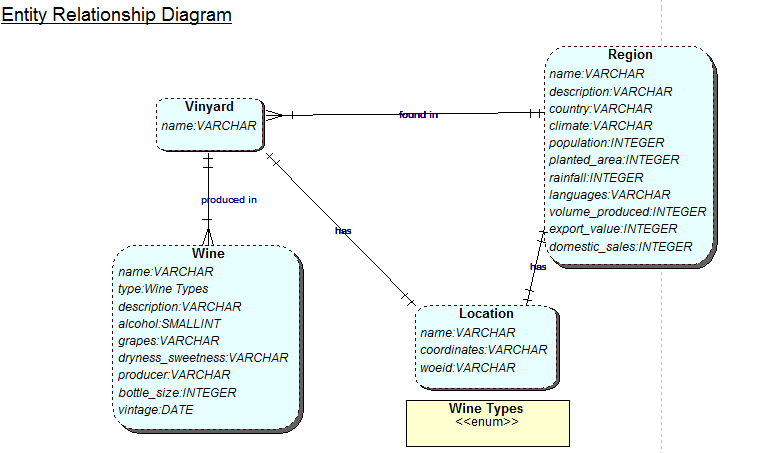
1. Prototyping projects might make use of an interpretative tool, such as an incremental compiler/interpreter/debugger.
2. External interfaces may require some of the modules to be pre-compiled.

## Outstanding Issues

1. Provide details of any design issues that remain unresolved at the date of issue of this document. Explain options, pros and cons, and give an estimate of which option is most likely. Outline impact of each option on the rest of the design.

## Decomposition Description

1. The software components should be summarised. This should be presented as structure charts or object diagrams showing the hierarchy, control flow and data flow between the components.
2. If the UML paradigm is used, then the decomposition description should make extensive use of the nine ‘UML Diagrams’ that in effect define the operation of the system.



# 5 .Component Description

1. For a software implementation, this and the previous section should provide sufficient information for a programmer to produce the software, and for a maintainer, who may not be the developer, to make subsequent changes. The detailed content will depend upon the software tool to be used. The software may be produced using coding statements written by an application programmer. In contrast, it may be automatically generated by an application development tool, or indeed a mixture of both.
2. It is worth reflecting for a moment on the term ‘component’ and giving some definition as to what a ‘component’ might entail. The following definitions are offered to the reader:

* Client-Based Components – User-centric graphical interface classes and widgets (e.g. Java Advanced Windowing Toolkit, Motif, Swing, Java Beans) implemented with automated tools like GUI builders and testers.
* Implementation Components – General-purpose language libraries or bindings that aid in the implementation of a design in a particular language (e.g. JDK, container classes, middleware wrappers, data portability streams), implemented through tools like UML code generators and repositories.
* Infrastructure Components – General-purpose processes built for a particular middleware architecture (e.g. CORBA, EJB, and TUXEDO) that can be customised for a specific task (e.g. logging, transactions, queuing), implemented through service and servant modelling tools (e.g. Rational Rose or BEA Webgain Studio).
* Architecture Components – Reusable architectural and configuration concepts that are documented and ready for reapplication (e.g. publish/subscribe, Store & Forward, Push) implemented through automated tools like UML modelling tools.

1. Component-based design offers a great economy of effort by encapsulating functionality at the right level. Application components offer re-use and can easily be enhanced. Re-useable components capture the repeated functionality of common system behaviour (such as infrastructure services).
2. The descriptions of the components should be laid out hierarchically. There should be subsections dealing with the following aspects of each component:

* 5.n Component identifier
* 5.n.1 Type
* 5.n.2 Purpose
* 5.n.3 Function
* 5.n.4 Subordinates
* 5.n.5 Dependencies
* 5.n.6 Interfaces
* 5.n.7 Resources
* 5.n.8 References
* 5.n.9 Processing
* 5.n.10 Data
* Stuff hetdjei djyeojd fheidj

1. The number 'n' should relate to the place of the component in the hierarchy.

## Component Identifier

1. Each component should have a unique identifier. The identifiers to be used for components should be defined by the project and described elsewhere.

### Type

1. This section should describe the type of component, e.g. task, subroutine, subprogram, package, file.
2. The contents of some component description sections depend on the component type. For the purpose of this template the categories: executable, i.e. contains computer instructions, or non-executable, i.e. contains only data, are used.

### Purpose

1. The purpose of a component should be defined by tracing it to the software requirements that it implements.
2. Backwards traceability depends upon each component description explicitly referencing the requirements that justify its existence.

### Function

1. The function of a component must be defined in this document. This should a short description of what the component does and will depend upon the component type e.g. it may be a description of the process or of the data to be stored or transmitted.
2. More detail will be provided in Processing (see below).

### Subordinates

1. This section should list the modules that are ‘called by’ this component. The subordinates of a database could be the files that ‘compose’ it. The subordinates of an object are the objects that are ‘used by’ it.

### Dependencies

1. The dependencies of a component should be defined by listing the constraints placed upon its use by other components. For example:

* what operations have to have taken place before this component is called?
* what operations are excluded when this operation is taking place?

Document Control

|  |  |
| --- | --- |
| **Title:** |  |
| **Issue:** |  |
| **Date:** |  |
| **Author:** |  |
| **Distribution:** | EC DG Enterprise – Gavino Murgia Project Team |
| **Reference:** |  |
| **Filename:** |  |
| **Control:** | Reissue as complete document only |

Document Signoff

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Nature of Signoff** | **Person** | **Signature** | **Date** | **Role** |
| Authors |  |  |  | Project Member |
| Reviewers | Mark Pillatt |  |  | Consultant |