eSolutions

ICT Volume 3 : Application Standards

ICT 3.1.1-2014 Object Oriented Design Standards

Abstract

This document defines the standards related to Object Oriented software design.

Document Control

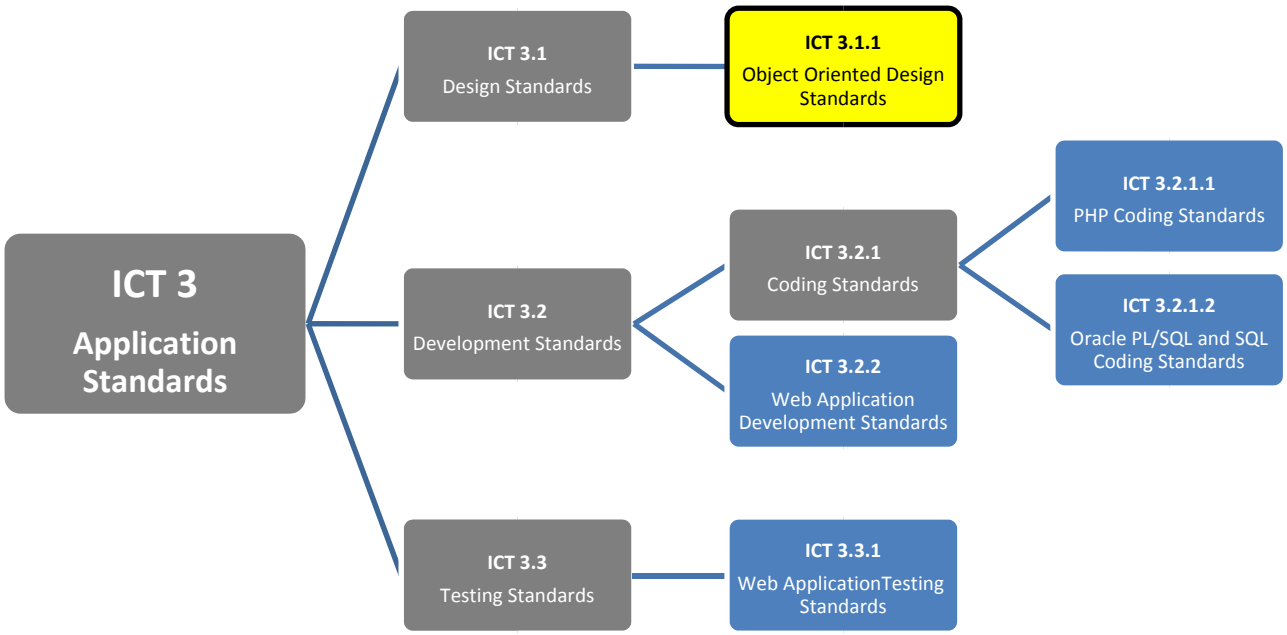
Document Title	ICT 3.1.1-2014 Object Oriented Design Standards
Version	2014
Controlled Copy (Electronic Reference)	

Document History

Ver.	Primary Author(s)	Description of Version	Date Completed
0.01	Steven George	Initial draft	26-11-2008
0.02	Steven George	Standards compliance	25-02-2009
0.03	Steven George	Standards compliance	18-03-2009
1.0	Steven George	Annual review	05-07-2010
2.0	Steven George	2011 Update	02-11-2010
2.1	Michael Heley	Standards Compliance Review	11-09-2014

Table of Contents

ICT Volume 3 : Application Standards



Standards Brief

This document serves to outline standards that shall apply within Deakin University.

Standard Document Access

All Deakin University staff and authorised/approved contracted personnel are provided access to this document.

Policy

These standards must be used in conjunction with all other referenced standards, and when considered in isolation from the referenced standards may not constitute adequate conformance.

Conflict of Information or Clarification

Whenever a conflict of information occurs or clarification of instruction is required, all queries shall be made to Deakin University eSolutions (DeS).

1 Naming

1.1 Namespaces

1.1.1 Namespaces shall be used to separate identity

Each class shall reside within a defined namespace. All namespaces shall derive from a base namespace.

1.2 Casing

Casing for files, namespaces, classes and class members shall conform to the specifications set out in the respective Coding Standards document.

2 Documentation

2.1 The design of any software must be documented

Documentation is required for the design review and approval processes.

2.2 The Unified Modelling Language (UML) shall be used

Object Oriented software design shall be prepared using UML notation.

2.2.1 Mandatory models

The following models shall be required as part of any Object Oriented software design. Model must consist of get/set/is/has/can methods, and refrain from any business logic.

2.2.1.1 Class diagram

The class diagram must include (where applicable):

1. Class Name
2. Inheritance
3. Attributes, including encapsulation and data type
4. Methods, including signature, encapsulation and data type
5. Exceptions

2.2.2 Supplementary Models

Supplementary models are optional and may be used to convey or illustrate various functions or aspects of a component, class or application. Examples include:

- State diagram

- Use case diagram
- Sequence diagram

3 Architectural Conventions

3.1 Abstraction

3.1.1 Abstract data types shall be defined by classes.

Where an Abstract Data Type (ADT) is required (either implicitly or explicitly), a class shall define that ADT.

Arrays are not an acceptable implementation of an ADT.

3.1.2 Classes should be an abstraction of a single entity in the problem domain

A class should represent one entity, and this entity should be easily defined. A class should not represent more than one entity in the problem domain. A class can bundle various functionality, however the functionality must relate to a single entity.

For example, a "Database" class encompassing several various database functions is satisfactory; however a "Common" class encompassing many un-related functions used by the application is not satisfactory.

3.1.3 The role of a class should be able to be defined in one sentence, without qualifying statements.

It should be possible to state "This class represents X". Or, it should be possible to state "This class provides functionality for entity X". It should not be possible to state "This class represents an X only when Y is true", or "This class provides functionality for entity X and Y".

3.1.4 Classes shall exhibit low coupling and high cohesion

Classes should be an abstraction of a single entity (see 3.1.2) and should not rely on the functionality of other entities, except for the interfaces exposed by the other entities.

3.1.5 Classes shall be re-usable

Classes shall be designed in a way that promotes re-usability across multiple applications.

3.2 Encapsulation

3.2.1 A class shall encapsulate its private members

Private members of a class, which are involved with the detail of the class implementation, must be encapsulated by the class.

3.3 Interfaces

3.3.1 Classes must exhibit a well defined public interface

Each member of the public interface shall relate to the class abstraction.

3.3.2 Classes must not display implementation leakage

Classes shall not allow details of the implementation become part of the class interface

3.3.3 The order in which member methods are defined is irrelevant

The order in which methods are defined within the class should have no impact on the functionality of the class.

3.3.4 The order in which member methods are called is irrelevant

The order in which methods are called from a class should have no impact on the results generated by the class.

3.3.5 Classes shall be self-initialising

Construction shall evade any requirement for further initialisation. Therefore, any required initialisation must be performed within the constructor.

Any required parameters for a newly instantiated object shall be required parameters of the class constructor.

Any optional parameters for a newly instantiated object should be provided after the object is instantiated via modifier methods.

3.3.6 Where classes share a common interface, Interfaces should be implemented.

In the instance where a common interface is being enforced, Interfaces should be implemented by each of the classes that expose that interface.

3.4 Class members

3.4.1 All functions shall be in the context of some class.

There shall be no functions within the global scope. Each method shall be contained within the context of a class, and therefore also a namespace.

4 Error handling

4.1.1 Custom exceptions shall be documented in the design

Any custom exceptions shall be depicted within the design of the software.

5 Appendix A

Definitions

Term/Abbreviation.	Definition
Abstraction	The mechanism and practice of abstraction reduces and factors out details so that one can focus on a few concepts at a time.
Abstract Data Type (ADT)	A mathematical model for a certain class of data structures that have similar behaviour.
Coupling	Coupling (or dependency) is the degree to which each program module relies on each one of the other modules.
Cohesion	A measure of how strongly-related and focused the various responsibilities of a software module are. If the methods that serve the given class tend to be similar in many aspects the class is said to have high cohesion. In a highly-cohesive system, code readability and the likelihood of reuse is increased, while complexity is kept manageable.
Class	The basic building block of software in the *object-oriented programming* paradigm.
Namespace	An abstract container providing context for the items (names, or technical terms, or words) it holds and allowing disambiguation of homonym items residing in different namespaces
UML	A standardized general-purpose modelling language in the field of software engineering