University of South Australia

School of Information Technology and Mathematical Sciences

COMP 3023 Software Development with C++

Group Project

Networked Asset Manager

Introduction

This document specifies the requirements for the group project of the Software Development with C++ course, SP5 2018. This assignment will develop your skills by developing an application in a team environment while allowing you to put into practice what has been taught in class during this course.

The assignment will require you to design, implement, test, and debug a GUI-based networked database[1] application using object-oriented methods and the application of Design Patterns. The application will store information about Assets, their types and properties, and their maintenance as well as support the sharing of this information with other running instances of the application over the network. To realise this, several Design patterns will need to be incorporated into the design and implementation of the application. The project will require you to use collaboration tools and version control to manage the project and collaborate with your team members as you progress through the assignment as well as document your classes appropriately through UML class diagrams and Doxygen comments. To support concurrent development by each team member, Unit Testing will be very important to ensure that things are working before a GUI is able to test each component of the application. The work in this assignment is to be performed **as a group** and will be **submitted via LearnOnline** at the end of week 13: **due by 2 November 2018 11:59 PM**. Refer to section *Submission Details* for specific instructions.

If any parts of this specification appear unclear, please ensure you seek clarification.

Learning Outcomes

After completing this assignment, you will have learnt to:

- Collaboratively design a class hierarchy using UML
- Apply Object-Oriented principles (encapsulation, reuse, etc.) and Design Patterns to the software design
- Implement the software design in C++ within a team environment
- Develop a cross-platform application using a cross-platform framework (Qt)
- Use professional tools for collaboration and software development, such as: version control, project management, issue tracking, and continuous integration
- Write code adhering to coding standards, guidelines and good programming practices

Design Patterns

In developing the class design for the Networked Asset Manager, the following the Design Patterns **will** be incorporated:

- Singleton (extensible version)
- Abstract Factory
- Prototype

[1] You will not actually be using a database, just manipulating objects in memory.

- Type-Object

Unlike the previous assignment you do not have to identify where to apply the Design Patterns (there is complexity elsewhere in this project). However, you should read the description of the patterns.

For more information on Abstract Factory and Prototype[2], refer to the book:
Gamma, E, Helm, R, Johnson, R and Vlissides, J 1995, *Design patterns: elements of reusable object-oriented software*, Addison-Wesley, ISBN: 978-0-201-63361-0.[3]
Be aware that the C++ examples from the book are for an older version of C++. If using them for guidance, they must be updated to C++ 14.

For the Type-Object pattern, refer to the article (posted to the course website for convenience):

Woolf, B., Johnson, R.: The type object pattern. Pattern Lang. Progr. Des. 3, 132 (1996)

Adequate descriptions of the above patterns along with some code examples can also be found on Wikipedia and elsewhere:

- Singleton: https://en.wikipedia.org/wiki/Singleton_pattern (Note: the Wikipedia entry does not consider extensibility of the Singleton.)
- Abstract Factory: https://en.wikipedia.org/wiki/Abstract_factory_pattern
- Prototype: https://en.wikipedia.org/wiki/Prototype_pattern
- Type-Object: http://gameprogrammingpatterns.com/type-object.html

Task Description

In this assignment, you will design and implement a GUI-based networked database application that fulfils the following specification. The application will store details about assets that a company owns, including records of maintenance for auditing purposes. However, the company operates in a highly dynamic environment with new types of assets becoming available all the time; therefore, the application will also keep track of the *types* of asset and their properties, which may differ between asset types.

The aim of the assignment is to build a software application in a team environment using a cross-platform framework (Qt), GUI and networking libraries. The style of application (a database application with a web service-style API) is very common in the real-world and something you are likely to come across when you graduate. The design and implementation of the application will require features of C++ taught in the latter half of the course, such as templates, in addition to those features covered previously, such as inheritance and polymorphism. The focus of this assignment is the collaborative development of the application. Due to the added difficulty and complexity of managing a team, this specification will be more detailed when it comes to the application of the Design Patterns and the major interfaces that are required. You will be required to develop a UML Class Diagram that incorporates the described interfaces plus additional classes/functions your group considers necessary. This will be a key piece of documentation that supports communication between team members. Moreover, the separation of tasks between team members and the use of Unit Testing will be important to allow concurrent development of different aspects of the application by different team members. The specification consists of some larger and more complex aspects, as well as some smaller components: the idea being that group members of different capability levels should each be able to contribute to the group effort.

First, you will need to ***develop a class design using UML class diagrams*** based on the requirements in the remainder of this specification. A simple and free UML editor that can be used for this assignment is UMLet

[2] The Type-Object pattern is **not** in the GoF *Design Patterns* book.
[3] Scanned PDFs of the relevant sections are available from the course website as the library has limited copies of the book.

(http://umlet.com/). The UML class diagram must include all classes, public data members, public member functions, protected data members and member functions (if any), and associations between classes. The diagram must identify the key roles/participants of the Design Patterns occurring in the design. The diagram may optionally display private member functions, but private member variables should not be included.

Workplan and Group Collaboration

To complete this assignment, it is important that you take a planned, methodical approach in collaboration with your group members. You and your group will need to think about how to break down the assignment into smaller chunks of functionality that can be allocated among the team and implemented and tested independently. Working in an incremental manner, and testing as you progress, helps to limit the scope of errors that are introduced as you perform the implementation and will assist when integrating your work with others. Moreover, an incremental approach allows you to focus on a single problem at a time as well as refactor and redesign the application in a controlled manner as you (re)discover requirements or issues that were not obvious at first.

To enable adequate collaboration with your group members, the first order of business will be to organise a platform for collaboration. At the centre of this collaboration framework will be a version control system (Git) through an online repository. Possible online repositories include BitBucket [https://bitbucket.org/] and GitLab [https://about.gitlab.com/] (which many of you may already be using). Note: whichever you choose you **must** ensure that your **repository is private** and **cannot be seen by anyone outside your group**. Online version control interfaces like BitBucket, GitLab, and GitHub[4] provide some basic project management features such as issue tracking and wikis. However, for improved collaboration and project management, I suggest you integrate Trello (or similar). Trello [https://trello.com/] is a project management and collaboration tool popular among small, agile teams. It is quite simple and will help you keep track of tasks, assign them to group members, and indicate to your team when they are complete.

This assignment should be completed in the following stages:

1. Read this assignment specification in full.
2. Check your group allocations and contact your group members.
3. Read this assignment specification again.
4. Organise with your team an online repository and collaboration environment—someone will need to create the repository and invite the others to access it. This step will be essential in the in ensuring ongoing communication with team members.
5. You and your team create an initial class design of the application—this will serve as a primary communication tool between team members on what is expected and should be maintained throughout the project.
6. Implement class and function stubs—this will allow group members to work somewhat independently without issues due to missing classes, functions, etc. It will also allow tests to be created based on the expected interfaces.
7. Identify team members' strengths and weakness and assign tasks to individual team members.
8. Perform your task(s)
9. Use the online collaboration tools and in person meetings (if possible) to manage the project and monitor progress. (Back to 8.)
10. Successfully complete the project.

[4] GitHub is highly popular for open source projects; however, it requires a paid subscription to obtain private repositories. While you get private repositories for free as a student, it is unclear what happens if GitHub decides you no longer fulfil the student requirements, that is, they may make your private repositories public. Therefore, **do not** use GitHub for your project.

General Functionality

The Networked Asset Manager application will need to support the creation, management, and deletion of assets and asset related information in a networked environment. Logged on users (in the real-world security and permissions, as well as knowing who modified the data, are serious concerns) will be able to view and modify the details of assets, etc., through a GUI with changes synchronised between other instances of the application. In addition, users will be able to search for assets using some of their common attributes (e.g., serial number).

Core to the application is an Asset Register that stores the assets and related details, such as their type, their properties, maintenance records, and custodians (i.e., the person responsible for managing the asset). Each type of information is stored in and retrieved from the Asset Register using its unique ID. Each stored entity has some additional meta-data to help manage it: last edit time, and the last edit user. In a real-world application it the storage class would be backed by a database; however, for simplicity we will just store the objects in memory using the map data structure.

You will need to make use of Qt's Signals and Slots mechanism (an implementation of the Observer Design Pattern) to communicate between components and ensure that the state of the application is correctly synchronised.

The user must be able to select at runtime (i.e., when they "login") whether they want to run the application in "online" or "offline" mode. This will be handled using the extensible singleton pattern with a networked and non-networked version of the asset storage class. Below are some details of achieving this.

AbstractAssetRegister «singleton» (derived from: QObject)

The `AbstractAssetRegister` class will provide the interface for all types of asset register (offline, networked, possibly a future database backed implementation, etc.), including access to the singleton instance. `AbstractAssetRegister` is an abstract class and must not be instantiable itself; moreover, it must be a subclass of `QObject` to attain the Signals and Slots capability.

Constructors:

- `explicit AbstractAssetRegister(QObject*)`—since the class is derived from `QObject`, it should have the common `QObject` constructor for specifying the parent object (you could make the parent the singleton instance of `QApplication`, for example). For extensibility of the Singleton pattern, the constructor must be `protected`.

Instance-level interface (i.e., non-`static` functions) includes (but is not limited to):

- `username()`/`setUsername(…)`[5]—accessor and mutator for managing the "logged-in" user. This will just be a simple string (`QString` since we are in the Qt framework) specified by the user through the GUI. There will not be any control over what users are allowed, permissions, etc. it will only be used to record who created and modified an asset (or related entity) as part of the entity's meta-data.
- `generateId()`—since we are operating in a distributed environment it is important that we avoid clashes between entity identifiers created by different instances of the application running on different (or the same) computer. A relatively simple way to achieve this is to use randomly generated IDs using an algorithm designed for such systems such as UUIDs (Universally Unique Identifiers). This function will create and return a new UUID (as a `QString`) to be used as an identifier. This can be done using Qt with the `QUuid` class and its `createUuid()` function.

[5] Where an ellipsis (i.e. '…') appear in the parameter section of a function signature, it means it requires 1 or more parameters but they have not been explicitly provided as part of this specification.

- `storeEntity(…)`—stores an entity (`Asset`, `AssetType`, `Custodian`, or `MaintenanceRecord`) in the `AssetRegister`: must return `true` if the entity was added, `false` if the it was not (e.g., due to the ID not being unique). May be a single function using inheritance and polymorphism to a common base class, or multiple overloaded functions. Document your design decision.
- `retrieveEntity(id)`—retrieves an entity (`Asset`, `AssetType`, `Custodian`, or `MaintenanceRecord`) from the `AssetRegister` using the entity's identifier. Must return a `nullptr` if an entity with that identifier cannot be found. As above, may be one function or a group of overloaded functions, document your decision.
- `deleteEntity(id)`—remove the entity with the given identifier from the register: must return `true` if the entity was removed, `false` if it was not (e.g., it may have already been removed). Calling this function multiple times with the same identifier will not have any negative consequences. Make sure you clear any dynamically allocated memory when deleting an entity from the register (refer to `QObject::deleteLater()` when cleaning up `QObject`s).
- `allEntities([type])`[6]—returns a collection of all entities stored in the register. Modifying the returned collection must not modify the register. You must be able to retrieve all entities regardless of concrete class, as well as all entities of a particular class (i.e., `Asset`, `AssetType`, `Custodian`, `MaintenanceRecord`).

The class level interface (`static` members) includes:

- `template<T> instance()`—`static` member function to return the single instance of this class (or any of its subclasses). Use lazy initialisation to create the single instance if one does not currently exist. To support extensibility, this function will be a template function: the type parameter will specify the concrete subclass that is to be created (if the register instance does not already exist). Once an instance of a type has been created, that instance will always be returned, regardless if the function is called with a different type: i.e., there cannot be multiple register instances of different types instantiated at the same time.
  For example, to select the simple `AssetRegister` class as the singleton, the first call to this function should be: `AbstractAssetRegister::instance<AssetRegister>()`
  Note: you can create a non-template version of the function for convenience. This convenience function should log an error if it is called before an instance of any concrete register types is created.
- `register`—the `static` data member of type `AbstractAssetRegister`. Must be able to support polymorphism. To allow subclasses to manipulate the `static` member (to support the extensibility requirement) this member must be `protected`. This will allow the creation of a `MockAssetRegister` for testing. Such a special subclass should have `static` member function that can clear the register class variable, allowing a new register object to be created the on the subsequent call to `instance()`.

Signals:

- addedEntity(id, [type])—emitted when a new entity is added to the register.
- deletedEntity(id, [type])—emitted when an entity is deleted from the register.

AssetRegister (derived from: AbstractAssetRegister)

The first concrete subclass of `AbstractAssetRegister` is the `AssetRegister`. This class is relatively straightforward: it simply implements each of the functions of the interface defined by `AbstractAssetRegister`.

Important: to support the extensible singleton, this class must declare `AbstractAssetRegister` to be a `friend class`. This will allow the `instance<T>() static` function (which is defined in the scope of `AbstractAssetRegister` and not its subclasses) to access the protected constructor of `AssetRegister`.

[6] Parameters in square brackets '[' & ']' are optional parameters.

NetworkedAssetRegister (derived from: AssetRegister)

The `NetworkedAssetRegister` will behave in the same way as the standard `AssetRegister` (its core behaviour is inherited) but adds support for synchronising itself over the network. It will be instantiated when the user chooses to "log-in" using "online mode". It must have a data member of type `NetworkManager` (refer to the Networking section below), which it uses to exchange entities with other instances of the application running in online mode. You may use a combination of function overrides and signals/slots to ensure the state of the distributed application is synchronised across all participants. When updating the stored entities, you must check the `lastEditedTime` and only update the local entity if the `lastEditedTime` is prior to that of the entity received from the network. Refer to section RegisteredEntity for the attributes of entities.

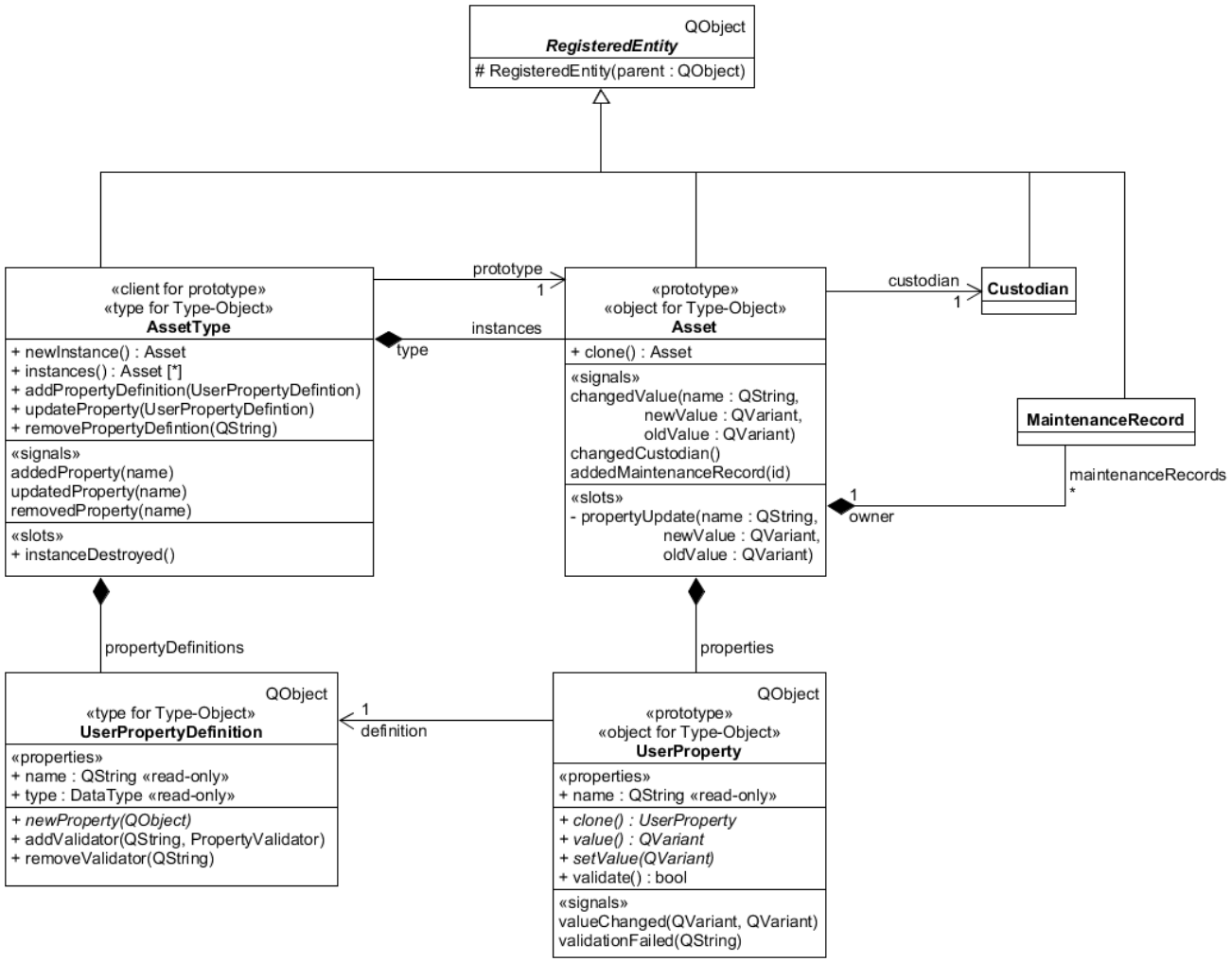
Assets, Asset Types, and their Properties

First and foremost, the Asset Register application needs to be able to store information about the company's assets. This includes information such as the serial number, model, purchase price, who manages it, and what maintenance has been performed on it. However, it also includes information about properties[7] that differ between different types of asset: for example, Pump assets have properties such as flow rate and alignment (horizontal/vertical), while Computers may indicate their operating system, MAC address, IP address, etc. Therefore, we need to be able represent the *types* of asset in the system alongside the assets themselves. In addition, we must represent the *user-defined properties* (or user properties) specific to the types defined at runtime, which will allow functionality such as validation of property *values* (e.g., the operating system property for Computer may only allow the values: "OSX", "Windows", "Linux").

There are a couple of Design Patterns that can help to dynamically define the types of asset at runtime. The first is Prototype, which allows special instances (called prototypes) to be created that represent the type itself. These prototypes can be *cloned* (i.e., copied) to create the actual instances of the type. Defining the `Asset` class as prototype would allow the description of an asset type through the specification of the typical values for that type; however, the copies often become disconnected from the prototype and the prototype does distinguish between information related to the type itself and that of the real instances.

[7] Qt also provides a concept of 'dynamic properties', do not confuse the user-defined properties that we are creating here with the dynamic properties of Qt (which are still compile time constructs).

The Type-Object Design Pattern also allows the dynamic definition of types/classes at runtime. Rather than representing the type as prototype, it defines a "type" class (such as `AssetType`) to represent types themselves and an "object" class (`Asset`) to represent the instances. The type class is then responsible for creating, initialising, and possibly tracking, instances of the type. While this approach allows the separation of type information vs. object information, it makes the type class responsible for creating instances (which may be



better handled by the object class) and can lead to duplication if, for example, you want to manage default values for attributes and user-properties (which is better handled by the Prototype Pattern).

To adequately support the representation of assets, their types, and properties, we will combine the Prototype and Type-Object patterns. The following is an extract of a UML diagram illustrating how this will be done. The diagram is minimal, illustrating only the core interface required to support the representation of `Assets`, `AssetTypes`, `UserProperties`. It also illustrates the associations to `Custodian` and `MaintenanceRecord` for clarity. You will need to expand on what is illustrated to support your own design and implementation.

The following includes a brief description of the classes and their data members.

RegisteredEntity

DERIVES FROM: QOBJECT

Is an abstract class that is the parent of all classes that are directly stored in the `AssetRegister` (i.e., Asset, `AssetType`, `Custodian`, and `MaintenanceRecord`). It inherits from `QObject` so that each of its derived classes inherit the Signals & Slots mechanism from Qt, among other things. Its main purpose is to provide metadata that each derived class requires to be properly managed by the `AssetRegister`:

- id—the unique identifier of the entity, used to store and retrieve entities in the register, as well as synchronise entities across the network.
- lastEditTime—a timestamp (QDateTime) the indicates when the last modification was made to the entity; it is used for synchronisation across the network.
- lastEditedBy—the name of the logged in user (from AssetRegister) who was the last change the entity.

These data members must not be modifiable by the user from the GUI.

Each derived class of `RegisteredEntity` should specify appropriate signals that allow others (such as the UI) to be notified of interesting changes, for example, the addition of a property or change of an attribute or user property value.

Custodian

DERIVES FROM: REGISTEREDENTITY

A `Custodian` is someone who has responsibility to manage an `Asset`. One `Custodian` can manage many `Assets` and the details for `Custodians` can be managed independently of any `Asset`. It is suggested that this class should be implemented before the main `Asset`, `AssetType`, and `MaintenanceRecord` classes; it is largely independent and relatively simple, allowing you to learn aspects of the design and the Qt framework (GUI, and Signals/Slots, networking, etc.) before moving on to the more complex and interrelated classes. It is dependent on `UserProperty`, but will allow you to test the `UserProperty` and `UserPropertyDefinition` classes before embedding them in the more complex `Asset`/`AsssetType` structure.

Data Members:

- name—everyone needs a name
- department—a `UserProperty` of type `QString` that restricts the allowable values to the following: “Maintenance”, “Operations”, “Logistics”
- phoneNumber—a `UserProperty` of type `QString` that restricts the allowable values to (mostly) valued phone numbers. The constraints are as follows:
    - 8-11 digits (inclusive)
    - Optional ‘+’ at the beginning (and only the beginning)
    - Spaces are allowed
    - No other characters

  This allows values such as: 83026611, 8302 6611, 08 8302 6611, +61 8 8302 6611
  But not: 8302, 8302 6611 12345, 61+ 8 8302 6611, ext. 25000

Asset Type

DERIVES FROM: REGISTEREDENTITY

The `AssetType` class allows the definition of types at runtime. Its primary purpose is to describe the types of properties that its instances can have (along with validation requirements), the default values of those properties, and manage the `Asset` instances of itself (e.g., adding a new property definition must update each instance with a matching property). The `AssetType` class requires the following data properties:

- name—the name of the `AssetType`, e.g., ‘Centrifugal Pump’
- prototype—this is a special instance of `Asset` that the `AssetType` uses to manage the default values of properties as well as make new instances (by cloning the prototype).
- instances—this is the collection of `Assets` that are the *instances* of this `AssetType`. The prototype must not be included in this collection. Each Asset of the collection must refer back to the `AssetType` of the instances collection it is in (this is the more important association, as the collection of instances can

always be inferred from the back-references). This is a composition relation; therefore, destroying the `AssetType` should destroy all of its instances.

- propertyDefinitions—a collection of `UserPropertyDefinitons` that describe the properties specific to the type. This is what allows individual assets to have more specific properties as well as validate the values of those properties (the validators are attached to the `UserPropertyDefinitions`, not the `UserProperties` themselves). The properties are managed by name, i.e., there can only be one `UserPropertyDefinition` with a given name at any one time.

When the `propertyDefintions` of an `AssetType` are updated, you need to ensure that all instances of the `AssetType` have their properties updated to match.

The interface for `AssetType` should provide functions to manipulate the property definitions (add/remove, access, etc.), create a new instance of itself (i.e., clone its prototype), and otherwise keep track of its instances (for example, you want to track the destruction of its instances via the `QObject::destroyed` signal to ensure the instances collection is kept up-to-date when instances are deleted from the `AssetRegister`.

Asset

DERIVES FROM: REGISTEREDENTITY

The Asset class maintains the information related to the company's assets. This includes details of the asset itself (e.g., its type, model, serial number, and values for user-defined properties specified on its `AssetType`) as well as who manages it (its Custodian) and details of its maintenance (i.e., its `MaintenanceRecords`).

Asset is forms part of the prototype pattern, and therefore needs to be "clonable" (through a `clone()` function), this is not the same things as 'copyable'. The `clone()` function must produce a deep-copy, that is a copy that is completely independent of the original to prevent changes in one from affecting the other. Copyable, on the other hand, often only performs a shallow-copy where data members are simply copied value-by-value (as is the way of the default copy-constructor in C++) which means pointers, for example, remain pointing to the same object. When implementing the `clone()` function be careful not to clone the type of the `Asset`, a cloned `Asset` should refer to the same `AssetType` as the `Asset` from which it was cloned.

There are some properties that all Assets have, these are to be implemented as ordinary data members of the `Asset` class, including:

- serialNo—an identifier typically written on the object, its serial number. This is not the same as its 'id' (inherited from RegisteredEntity), which is an internal system identifier for keeping track of the object in the system. It is entirely possible for two assets of different types to have the same serialNo.
- brand—the maker of the asset, e.g., "ACME Pumps Co."
- model—the specific model of the Asset. While many assets may be of the same type (e.g., Pump, Computer, etc.), they may be of different models (e.g., 'ACME Co. Pump X3000')
- purchasePrice—the cost of the asset when it was bought (as a `UserProperty` of type `double` with a precision of 2, refer to `UserPropertyDefinition` below)
- purchaseDate—the date (`QDateTime`) at which the asset was bought by the company
- disposalDate—the date (`QDateTime`) at which the asset was disposed of (sold, thrown away, etc.) by the company. If an asset is not yet disposed, it will have no value for `disposalDate`; in contrast to `purchaseDate` which will always have a value as `Assets` in the `AssetRegister` are those that have been purchased by the company.
- custodian—a reference (not necessarily C++ reference) to a `Custodian` object. Note the cardinality of exactly 1 is from the user's perspective. That is, user interface must not allow an `Asset` to be stored or

saved if it has no `Custodian`; however, enforcing such a constraint at the lowest-level in the code is not always possible nor useful. For example, the `protoype` of an `AssetType` does not require a `Custodian`.

- maintenanceRecords—a collection of MaintenanceRecord objects that track the maintenance an Asset has had. The records should be kept in order based on their `timestamp` (i.e., the time the maintenance was performed, see Maintenance Record below) and there cannot be more than one record per `timestamp` (for the same `Asset`). This is a composition relation such that deleting an `Asset` deletes **all** its `MaintenanceRecords`.

When any of these values change, the `Asset` should emit a signal to indicate that an attribute has changed its value. In the case of `maintanenceRecords`, this may be when a record is added or updated, not just assigned.

The Asset class also contains a collection of `UserProperties`, which match the `UserProperyDefinitions` of the Asset's type. That is, the size of the properties collection must be the same as the size of the `propertyDefinitions` collection of the `AssetType`, and each `UserProperty` must refer to a `UserPropertyDefintion` in the `propertyDefinitions` collection. When the `UserPropertyDefinitions` of an `AssetType` are changed, the Assets that are instances must be kept synchronised.

Each `UserProperty` holds a value for a property described by its associated `UserPropertyDefinition`. When values change, the `Asset` should be notified, and a signal emitted by the `Asset` to let interested parties know that one of its properties has changed (e.g., a GUI might want to update the display if the Asset indicates a value has changed).

You may make the `AssetType` class a friend class of Asset to allow `AssetType` access to the private members of Asset. This is justifiable as the two class are very closely related, particularly due the prototype instances associated with each `AssetType`.

Note that `Asset` and `AssetType` can initially be developed without referencing `UserProperties`. If you take an incremental approach to development, they can be added after the initial classes are implemented and the basic instantiation functionality is working.

Maintenance Record

DERIVES FROM: REGISTEREDENTITY

The `MaintenanceRecord` class is relatively simple, it just holds data members describing the maintenance activity, who performed it, and when it was performed, for an Asset. The data members are as follows:

- owner—a reference to the `Asset` this `MaintenanceRecord` is for
- timestamp—the date/time (`QDateTime`) that the maintenance was performed
- performedBy—the name of the person who performed the maintenance (can be anything): `QString`
- maintenanceType—a `UserProperty` of type `QString` that restricts the allowable values to the following: "Inspection", "Minor Repair", "Major Repair", "Replace Broken".

In addition, the `MaintenanceRecord` class could implement a `clone()` function to simplify the cloning of their parent `Asset`.

UserPropertyDefinition

DERIVES FROM: QOBJECT

This class represents the description of a property and its validation constraints (using the `UserPropertyValidator` class). Paired with the `UserProperty` class, it is another instance of the Type-Object pattern: `UserPropertyDefintion` serves as the *type* of a `UserProperty` in an `Asset`. It has several keys aspects:

- name—the name of the property, each `UserProperty` of the `UserPropertyDefinition` will have the same name. Only one property with the same name can occur in an `AssetType`/`Asset` at one time, i.e., adding a `UserPropertyDefinition` to an `AssetType` using the name of a property already present must do nothing (use `updateProperty` to replace a property with an existing name).
- type—the type of the property, specified using an enumeration. You must support at least the following data types: `int`, string (using `QString` for compatibility with the rest of the Qt application), and `double`.
- validators—a collection of `UserPropertyValidators` that will be used to ensure that property values are correct.

The interface for `UserPropertyDefinition` must provide methods to create a new instance (i.e., a `UserProperty` object of the correct type that refers to the creating `UserPropertyDefinition`) and add validators. You **do not** need to support changing the type of an existing `UserPropertyDefinition`.

Unlike `AssetType`, `UserPropertyDefinition` does not keep track of its 'instances'. A `UserPropertyDefinition` and a `UserProperty` for an `AssetType`/`Asset` can always be reconciled by name. However, the two classes should be largely independent of `AssetType` and `Asset` so that they can be used as standard attributes of other classes (refer to 'phoneNumber' of the `Custodian` class).

You must support the ability to specify a precision for properties of type `double` (i.e., the number of digits after the decimal place). This could be done through a derived class, or at the level of `UserPropertyDefinition` with a function that is only applicable to definitions of that type.

UserProperty

DERIVES FROM: QOBJECT

The `UserProperty` class stores the value of a user-defined property for Assets (plus the occasional data member for other classes, such as 'phoneNumber' on Custodian). It fills the role of 'object' in the Type-Object Pattern alongside `UserPropertyDefinition`. To implement user-defined properties, we will combine inheritance/polymorphism, templates, and template specialisation.

This class makes use of the Qt class QVariant to provide a general interface to properties that may be of different types. The `QVariant` class can internally store values of many different types and provides functions to convert between them. By leveraging this, it greatly simplifies our handling of user-defined properties.

The general interface of `UserProperty` requires the following:

- name—the name of the property, must match that of its associated `UserPropertyDefinition`
- value/setValue(…)—retrieve/set the value of the property using the `QVariant` class
- validate()—validate whether the property's value is correct with respect to the validators stored on the associated `UserPropertyDefinition`
- clone—`UserProperty` is another instance of the prototype pattern; the clone function returns a deep copy of the `UserProperty`, allowing `Asset's` to be more easily cloned (you just need to clone each of its `UserProperties`). Note: the cloned `UserProperty` should refer to the same `UserPropertyDefinition` as the `UserProperty` from which it was cloned.

`UserProperty` should also include the following signals:

- valueChanged(new, old)—emitted when the value of the property changes. Note: when a slot is executed from receiving a signal, the function `sender()` can be called to return a pointer to the sending object (the `UserProperty` in this case).

- validationFailes(`QString` error)—emitted when the validation of the value fails. It should provide a message to indicate what the error is (this will be generated from the `PropertyValidator`)

To implement the specific types of `UserProperty` for `int`, string (`QString`), and `double`, you will need to use templates and template specialisation. You will need to:

1. Create a template class called `TypedUserProperty`, with a single template parameter 'T', and a data member of type 'T'. Note: Templates are declared and defined in the header file, not source file.
2. Ensure the template class inherits from `UserProperty`.
3. Implement each of the functions of the generic interface for `TypedUserProperty` (if there is not already an appropriate implementation on `UserProperty` or it cannot be inherited): e.g., `value()`, `validate()`, `clone()`
4. Use template specialisation to implement `setValue(…)` functions appropriate for the types `int`, `QString`, and `double` as required. Note: since these are fully specialised functions you will need to put them in the source file (e.g., 'typeduserproperty.cpp') or use the `inline` qualifier on their definitions in the header.

If the validation fails when a setting a property's value, the value should be unchanged from its original value when `setValue(…)` returns.

The template specialisation for properties of type `double` will need to take into account the `precision` of its `UserPropertyDefinition`, i.e., the number of digits to specify after the decimal point. This can be achieved by multiplying the value by a multiple of ten based on the precision, rounding the value (i.e., make it a whole number), and dividing by the same multiple of ten (you could specialise the `value()` function to do the division when the value is retrieved if you like). For example, for a precision of 2:

```
value = std::round(newValue * 100) / 100;
```

The multiple of 10 can be determine by: 10^precision
where 'precision' is the precision value defined on the `UserPropertyDefinition` of the `UserProperty`.

Note: when implementing the template specialisations, you may need to implement the specialisation for the entire `TypedUserProperty` class, depending on how you have implemented the rest of the functions and the inheritance.

UserPropertyValidator

DERIVES FROM: NOTHING

The `UserPropertyValidator` class is a quite simple. You may note that it is another instance of the Strategy pattern where the *context* is the `UserPropertyDefinition`/`UserProperty` classes and the *concrete strategies* are the specific types of validation that can be performed such as: minimum values, maximum value, enumeration, etc.

The `UserPropertyValidator` class will be an abstract class with the following interface:

- operator() (…)—`UserPropertyValidator` is a functor, that is it overloads the function call operator '`()`'. The overloaded operator takes a `UserProperty` (reference) and returns true if the value of the property conforms to the validation criteria.
- failureMessage(…)—returns a failure message (as a `QString`) for the given `UserProperty`. It provides the whole property so that the message may be constructed from as much information as required. For example, a minimum value constraint violation may report:
  "The property '<name>' must be greater than <min>. Value was <value>."

where <name> is the `name` of the property, <min> is the minimum value allowed by the validator, and <value> is the `value` retrieved from the `UserProperty`.

You **will need** to create concrete subclasses of `UserPropertyValidator` for the following validator types. They must support as many data types as possible (e.g., minimum value validator should work for both `int` and `double` properties), which can be achieved using templates.

- Minimum Value—given a specified minimum value, property values must be equal to or greater than the minimum. Must work with `int` and `double`.
- Maximum Value—given a specified maximum value, property values must be equal to or less than the maximum. Must work with `int` and `double`.
- Enumeration—given a list of values, property values must be in the list to be considered valid. Must work with int, double, and string (`QString`).
    - There are two special Enumeration validators: Department (refer to section Custodian) and Maintenance Type (refer to section Maintenance Record)
    - These special cases may have classes defined for them or they may simply be instances of the Enumeration validator.
- Phone Number—used by the 'phoneNumber' property of the `Custodian` class. Validates strings (`QString`) to ensure they conform to the requirements of a phone number. Refer to the Custodian section for details of the constraints.

User Interface

You are required to build a user interface using Qt's Widgets module. The project includes an empty main window class. You will need to implement it along with any other GUI classes you require to fulfil the requirements of the user interface. The requirements are quite loose in terms of how it the views/screens are actually defined. You may use any combination of windows, widgets, dialog boxes, etc. to fulfil the requirements. Moreover, you may make the GUI as fancy or simple as you want or are capable of given the timeframe. The following sections describe the interfaces that are required for the application.

User Login

When the application first starts, the user must see the User Login. The User Login must:

- Allow the user the enter their username
- Allow the user to specify online/offline mode
- If online mode is chosen, the user must be able to specify an 'endpoint'—the URL of another instance of the application on the network. The endpoint URL will be used to query for the complete list of application instances that the local instance must communicate with. This field may be empty (e.g., it is the first running instance) but the server should still start up to allow other instances to connect.
- Once the relevant details have been entered, the user must be able to 'login'.

When the user logs in, the singleton instance of `AbstractAsssetRegister` must be initialised and the username set: offline mode must initialise a standard `AssetRegister`, online mode must initialise a `NetworkedAssetRegister`.

Once the user has successfully logged in, they must be taken to the *Main Menu/Dashboard* and they must never be able to return to the *Login Screen* until they restart the application. (If needing to close the application to switch online/offline mode is good enough for Valve's Steam, it is good enough for us.)

Main Menu/Dashboard

The Main Menu/Dashboard provides a location to access the main functionality of the Asset Manager application. At a minimum it must allow the user to:

- View the stored Asset Types
- View the stored Assets
- View the stored Custodians
- Search for assets by serial number, model, and/or brand.

This may be achieved in-place or by navigating to another screen.

Optionally, the user may be able to:

- Add a new Asset Type to the register
- Add a new Asset to the register
- Add a new Custodian to the register

This may be achieved in-place, by navigating to another screen, displaying a dialog box, etc.

View Asset Types

When the user selects to View stored Asset Types, they must be presented with a view that lists (by some means such as `QListWidget`) all the Asset Types stored in the register. The user must be able to:

- Select an existing Asset Type and:
    - View the details of the Asset Type
    - (optionally) Edit the details of the Asset Type
    - Delete the Asset Type
- Add a new Asset Type to the register
- Return to the Main Menu

View Assets

When the user selects to View stored Assets, they must be presented with a view that lists (by some means such as `QTableWidget`/`QTableView`) all the Assets stored in the register. A table widget is suggested as it would be best for the user to several of the data members common to all Assets, such as serial number, brand, model, etc. The user must be able to:

- Select an existing Asset and:
    - View the details of the Asset
    - (optionally) Edit the details of the Asset
    - Delete the Asset
- Add a new Asset to the register
- Search for assets by serial number, brand, and or model.
- Return to the Main Menu

View Custodians

When the user selects to View stored Assets, they must be presented with a view that lists (by some means such as `QListWidget`/`QTableWidget`/`QTableView`). The user must be able to:

- Select an existing Custodian and:
    - View the details of the Custodian

    - (optionally) Edit the details of the Custodian
    - Delete the Custodian
- Add a new Custodian to the register
- Return to the Main Menu

View/Edit Asset Type

When the user selects to View/Edit an Asset Type, they must be presented with a view that shows the attributes and properties for the Asset Type and allow the user to activate "Edit Mode". The attributes to be shown include:

- `id` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `lastEditedBy` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `lastEditTime` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `name`
- `propertyDefinitions`; user must be able to view the `name`, `type`, and *default value* of each (Remember: the default value is stored on the `prototype` of the `AssetType`)

Optionally, the user can view/edit/add/remove the currently set validators on each `UserPropertyDefinition` of the `AssetType`.

When in "Edit Mode", the user must be able to modify the attributes, add/remove `UserPropertyDefinition`s, and ***save the changes*** (storing them in the Asset Register, synchronising the changes over the network if in "online" mode, and returning to "View Mode") or ***cancel the edit*** (restoring the original values and returning to "View Mode").

When editing existing `UserPropertyDefinitions` of an `AssetType`, only the *default value* must be editable. Editing the other values can be achieved by deleting and adding a new `UserPropertyDefinition` with the same name.

View/Edit Asset

When the user selects to View/Edit an Asset, they must be presented with a view that shows the attributes and properties for the Asset and allow the user to activate "Edit Mode". The attributes to be shown include:

- `id` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `lastEditedBy` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `lastEditTime` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `serialNo`
- `brand`
- `model`
- `purchasePrice`
- `purchaseDate`
- `disposalDate`
- `properties`; each of the `UserProperties` must be viewable including their `name`, `type`, and `value`. Note: the type could be retrieved by checking the `UserProperty` itself, or by accessing the `UserProperty`'s definition.
- `maintenanceRecords`; may be displayed inline using a table widget (e.g., `QTableWidget` or `QTableView`) or displayed in another screen, dialog box, etc.

`MaintenanceRecords` are owned by an `Asset` and can only be viewed from the `Asset` itself. The user must be able to view/edit the properties of the `MaintenanceRecords` including: `owner`, `timestamp` (see QDateTimeEdit), `performedBy`, and `maintenanceType`. The `maintenanceType` property may be specified using a drop-down list

(see QComboBox) to enforce the requirement, or the validation can be used to warn the user of an invalid selection.

When in "Edit Mode", the user must be able to modify the above attributes (including those of `UserProperties` and `MaintenanceRecord`s), add `MaintenanceRecords`, and ***save the changes*** (storing them in the Asset Register, synchronising the changes over the network if in "online" mode, and returning to "View Mode") or ***cancel the edit*** (restoring the original values and returning to "View Mode").

For auditing purposes, the user **must not** be able to ***delete*** `MaintenanceRecord`s.

View/Edit Custodian

When the user selects to View/Edit a Custodian, they must be presented with a view that shows the attributes and properties for the Custodian and allow the user to activate "Edit Mode". The attributes to be shown include:

- `id` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `lastEditedBy` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `lastEditTime` (from `RegisteredEntity`); must be read-only in "Edit Mode"
- `name`
- `department`; when editing, the allowable departments may be selectable from a drop-down list (see QComboBox) to enforce the requirement, or the validation can be used to warn the user of an invalid selection.
- `phoneNumber`

When in "Edit Mode", the user must be able to modify the above attributes (including those of `MaintenanceRecord`s), add `MaintenanceRecords`, and save the changes (storing them in the Asset Register, synchronising the changes over the network if in "online" mode, and returning to "View Mode") or cancel the edit (restoring the original values and returning to "View Mode").

Search for Assets

When the user selects "Search for Assets" they must be able to enter a serial number, brand, and/or model for which to search. When performing the search, each non-blank value must match, i.e., if all three are non-blank an Asset will only match if it has a matching *serial number* **and** a matching *brand* **and** a matching *model*. *Matching* is determined by an **exact string match**.

Optionally, regular expressions or simple globbing can be used to perform the search.

After entering one or more values, the user must be shown a list (in some form) of Assets that meet the search criteria. This functionality **should** reuse the View Assets interface and allow all the same functionality, such as selecting, viewing/editing, deleting an Asset.

General Notes and Requirements

The *View/Edit Entity* views can be configured using a QFormLayout, which provides a simple two column layout designed to present labels and input fields side-by-side. If necessary, a widget can be expanded to fill both columns of the `QFormLayout`.

Before an entity can be saved/stored in the Asset Register, the attributes and properties must be validated. If validation of any property/attribute fails, the user must be warned with an appropriate message and the save **must not** be allowed to proceed until the validation errors are rectified.

Modifying a value in the user interface **must not** change the underlying object or store the value in the Asset Register until the user confirms the save. This could be achieved by making use of the `clone()` function to make a copy in which the current values are stored without overriding the old object until the user confirms the save.

Alternatively, you could use data members on the form/widget classes that you create for the different fields and copy the values into the object once the user confirms the save. You may find the `UserProperty`/`UserPropertyDefinition` classes useful to handle this and fulfil the validation requirements even for attributes that are not represented using `UserProperties` on the underlying object.

Saving/storing an entity (`AssetType`, `Asset`, `MaintenanceRecord`, `Custodian`) must update the `lastEditedBy` attribute to the logged in user (i.e., the username from the register singleton instance) and the `lastEditTime` to the current time (refer to QDateTime).

Each main screen, view, or window (not including dialogs or other temporary windows) must be able to return to the *Main Menu/Dashboard*. This could be achieved using the Menu or Toolbar. If not using the Menu or Toolbar, they should be removed from the main window (and any others) entirely.

The *View/Edit* screens must be able to return to their original list screen: e.g., when viewing an `AssetType` the user must be able to return to the *View Asset Types* screen.

"Edit Mode" may be achieved through navigating to a separate screen or by enabling the widgets to allow editing.

It should be possible to reuse the View/Edit screens for each type when adding a new entity (for `AssetType`, `Asset`, and `Custodian`).

Networking

When a user selects "online" mode, the Asset Manager application will be able to communicate over the network with other instances of the application to share Asset information. The instances of the application will operate in a decentralised manner, sharing updates made locally to each instance of the application of which it is aware. The message exchanges will be performed using a simple protocol over HTTP, a bit like a RESTful web-service. REST (Representationl State Transfer) is a style of client-server architecture designed for (and developed out of) use on the internet and which has become highly popular over the last 5-10 years (although its origins go back much further). RESTful web-services are commonly implemented over HTTP and make use of URLs to locate *resources* on other systems and perform operations on them using standard HTTP methods (such as GET, PUT, POST, and DELETE). Due to the popularity of such web-services and the fact that the Asset Manager application is about storing, editing, querying resources (i.e., `AssetType`s, `Asset`s, `Custodian`s, etc.) it makes sense to have a REST API.

To implement and make use of a REST API we need a HTTP server and a HTTP client: the Asset Manager application will act in both roles to synchronise its content with other instances of the application in a decentralised manner. Due to the popularity of REST and the prevalence of HTTP in general there are plenty of libraries around that can handle HTTP communication. The Qt framework itself provides client-side capabilities, but it is lacking server support; therefore, for the server-side capabilities we will be using the open-source library QHttp [https://github.com/azadkuh/qhttp]. QHttp is a library built on Qt to provide HTTP server and client capabilities. As it is built on Qt, it uses the Signals and Slots mechanism to notify other objects when messages are sent and received. Although QHttp provides both client- and server- code, we will only be using its server-side capabilities to give you some experience integrating disparate libraries.

Important: the provided Qt Creator project contains an example of using both QHttp's server capabilities and Qt's client-side code to send and receive messages.

To integrate the two libraries and provide a simple to use, common interface (or façade[8]) for them, you will need to create a class called `NetworkManager`.

NetworkManager

DERIVES FROM: QOBJECT

The `NetworkManger` class will encapsulate an instance of `QHttpServer` and QNetworkAccessManager for receiving and sending requests, respectively.

Apart from this the underlying networking objects, the `NetworkManager` manages a set of `Connection`s. Each `Connection` represents another instance of the Asset Manager application with which it is communicating and is simply a pair of data members (and, hence, can be a `struct`): the `address` (see QHostAddress) of the instance, and its `port` number. A `NetworkManager` **must never** send a request to itself, but it may send requests to another instance on same computer (identifiable by the different port number).

Since we are operating in a dynamic/decentralised environment, servers may come up or go down without notice. Therefore, any request to a `Connection` may fail or *timeout*. If this happens, the `NetworkManager` must remove the `Connection` from the set to avoid contact it again. If the server comes back up it will resync the connections and the instance will be able to communicate with it again.

The NetworkManager interface must be at the level of application Messages: for example, the `NetworkManager` will process the QNetworkReply internally, create an appropriate `ResponseMessage` object (note: the provided code has the beginnings of the `Message` class hierarchy, refer to section Messages below) and emit a signal with that message object for the `NetworkedAssetManager` to process. The core idea is that the `NetworkedAssetManager` should not have to know about the underlying HTTP implementations.

The `NetworkManager` class has the following interface:

- start()—configures the server object (`qhttp::server::QHttpServer`) and begins listening for request on a specified port (Note: use ports 8101, 8102, or 8103). It must also synchronise its set of `Connections`, as described below in under the Protocol section. Once this function returns, it must be ready to send and receive application `Message`s.
- stop()—stops the server from listening to requests, should be called when the application is being shutdown.
- sendRequest(RequestMessage, […])—sends a `RequestMessage` of some type. You may include other parameters as necessary depending on how much you manage internally.
- sendResponse(ReponseMessage, […])—sends a `ResponseMessage` of some type in response to a request. You may need to provide some kind of identification so that the `NetworkManager` knows what underlying response (i.e., `qhttp::server::QHttpResponse`) it must send the `ResponseMessage` data through. Alternatively it could be managed internally by mapping one to the other.

The `NetworkManager` class requires the following signals:

- requestReceived(RequestMessage)—emitted after the HTTP server component receives a request and it is processed into an application-level `RequestMessage`. The `NetworkedAssetManager` should connect to this signal so that it can perform the required operation and respond to the request by calling sendResponse(…)

[8] The Façade pattern is Design Pattern about putting a simplified interface over complex subsystems to (among other things) help reduce complexity and dependencies.

- responseReceived(ReponseMessage)—emitted after the HTTP client component receives the response to a request and it is processed into an application-level `ResponseMessage`. The `NetworkedAssetManager` should connect to this signal so that it can process the response to a request.

REST-like API

The following table describes the API in terms of the path in a URL (that is the part after, and including, the first slash) that is being requested and the HTTP method used for the request. We only deal with a subset of HTTP methods for this application: GET, PUT, POST, and DELETE.

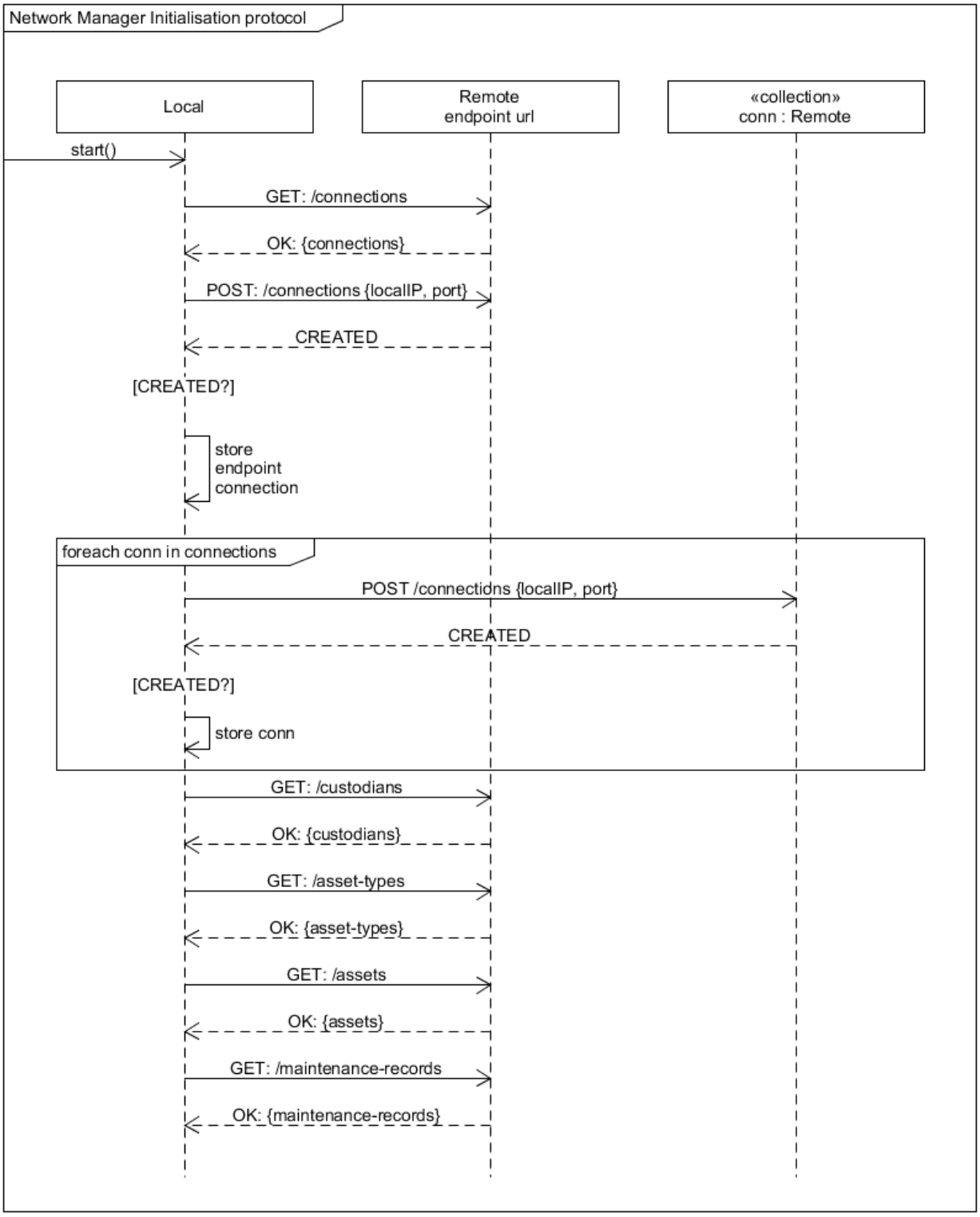
URL Path	GET	PUT	POST	DELETE
/connections	Retrieve the set of connections managed by the NetworkManager. Used when initially finding other instances to talk to.	N/A	Register a connection (yourself) with another NetworkManager. The connection data is in the request payload and must match the source, otherwise it will not be accepted. Responds with CREATED if successful.	N/A
/asset-types	Retrieve the collection of AssetTypes managed by the Asset Manager instance	N/A	Create a new AssetType using the data provided in the request payload. Responds with CREATED if successful.	N/A
/asset-types/{id}	Retrieve the single AssetType with the specified **id**. Responds 404 NOT FOUND if the AssetType does not exist in the Asset Manager.	Create or update the AssetType with the specified **id** using the data in the request payload.	N/A	Delete the AssetType with the specified **id**.
/assets	Retrieve the collection of Assets managed by the Asset Manager.	N/A	Create a new Asset using the data provided in the request payload. Responds with CREATED if successful	N/A
/assets/{id}	Retrieve the single Asset with the specified **id**. Responds 404 NOT FOUND if the Asset does not exist in the Asset Manager.	Create or update the Asset with the specified **id** using the data in the request payload.	N/A	Delete the Asset with the specified **id**.
/custodians	Retrieve the collection of Custodians managed by the Asset Manager.	N/A	Create a new Custodian using the data provided in the request payload. Responds with CREATED if successful	N/A
/custodians/{name}	Retrieve the single Custodian with the	Create or update the	N/A	Delete the Custodian

	specified **id**. Responds 404 NOT FOUND if the Custodian does not exist in the Asset Manager.	Custodian with the specified **id** using the data in the request payload.		with the specified **id**.
/maintenance-records	Retrieve the collection of MaintenanceRecords managed by the Asset Manager. Mostly used during initial synchronisation.	N/A	Create a new MaintenanceRecord using the data provided in the request payload. Responds with CREATED if successful	N/A
/maintenance-records?asset={asset-id}	As above, but limits the results to the MaintenanceRecords owned by the Asset with the specified **id**	N/A	N/A	N/A
/maintenance-records/{id}	Retrieve the single MaintenanceRecord with the specified **id**. Responds 404 NOT FOUND if the MaintenanceRecord does not exist in the Asset Manager.	Create or update the MaintenanceRecord with the specified **id** using the data in the request payload.	N/A	Delete the MaintenanceRecord with the specified **id**.

The second to last URL uses the query string aspect of URLs to parameterise the request while maintaining the flat URL structure—an alternative would be to have maintenance-records related paths follow those for asset to indicate the ownership but we have defined a top-level path for each type of `RegisteredEntity` stored in the AssetRegister.

Protocol

To manage the synchronisation amongst Asset Manager instance across the network, we need a protocol to first discover the other Asset Manager instances and then synchronise the content of the (Networked)AssetRegister. The below diagram illustrates the initialisation protocol. Note: the protocol is not perfect and does not allow for participants to drop out and reconnect without shutting down and restarting the application. It is intended, first and fore most, to be simple.



The protocol begins by retrieving all of the connection registered at the original endpoint provided at the Login Screen. The local Asset Manager instance then registers itself at each of the remote instances, storing the details of the connections if they respond with CREATED. After registering itself with each remote instance, the `NetworkManager` proceeds to retrieve each collection of `RegisteredEntity` type and store them locally. If a particular remote fails while trying to synchronise a collection, the `NetworkManager` should try another (the data is duplicated across all instances, so there should be no difference). If all connections are exhausted before all collections have been synchronised, the `NetworkManager` should shut down, the user should be notified, and the application should quit.

Note: the order of synchronisation is important, it is defined such that there are no issues with dependencies. Custodians have no dependencies (but are required by Assets) so they are synchronised first. `AssetTypes` are next as they are required by Assets (each Asset refers to its type); while `AssetTypes` track their instances, they are not necessary to describe the type and are not included in the data serialisation of an `AssetType`. Assets are synchronised next, finally the `MaintenanceRecords` are synchronised. `MaintenanceRecords` are owned by an Asset, but Assets can happily exist without `MaintenanceRecords` (similar to `AssetTypes` and Assets above).

Once this synchronisation is complete, the Asset Manager is ready to have users update the content of the Register.

Testing the networking component

Note: testing multiple instances of the application on the same computer will work regardless. The following information is for testing across multiple computers.

Note: Since the application is HTTP-based and the data is sent using a plain-text format, you can test if your server is working by opening a Web-browser and typing the URL into the address bar. This may be useful if you are not sure if it is a the server or client that is having trouble.

When testing your application from the computer pools on campus, you must ensure you have the correct configuration, otherwise your application will be blocked by the local firewall. You need to ensure the following for testing between different computers when on campus:

- Ports: 8101, 8102, 8103
- Computer Pool used by the course: F1-13, F1-15, F1-17, P1-13, P1-15
- Your project folder is on the D: drive
- You must not change the name of the project.

The restrictions are in place the minimise the scope of the changes to the firewall. The last two are due to the firewall rules being tied to the path of the application. For your information, the executable must be in one of the following locations to work correctly:

D:\build-assignment2-networked_asset_manager-Desktop_Qt_5_9_1_MinGW_32bit-Debug\xbin\asset_manager.exe

D:\build-assignment2-networked_asset_manager-Desktop_Qt_5_9_1_MinGW_32bit-Release\xbin\asset_manager.exe

When testing your application from your home computer, there are several ways of ensuring the communication between your own computer and instances running on the computers of other group members. You can configure Port Forwarding on your router to allow connections from outside to the computer that is running the Asset Manager application. For this to work you will also need to your external IP address (that is the one your router has with the rest of the Internet). There are many websites that will echo back your external IP Address for your information.

The second option is to use a VPN (Virtual Private Network). There are several free applications around that will let you set up a VPN with a small number of users. One of those application is Hamachi [https://www.vpn.net/]. You will need to download an install the application. When you run it, you will be given options to configure and setup a VPN. Communicate with your team to determine appropriate settings. Once configured, you should all appear to be on a local network and, therefore, will not need to change your router configuration. Just be sure to specify the endpoint URL with the IP address on the VPN, not your normal local IP address.

Messages

To handle the exchange of entities across the network we send `Messages`, which abstract away from the underlying libraries for sending/receiving messages and help convert between the application data (i.e., entities)

and the data payload required for a format/exchange mechanism. There are two main types of `Message`: `RequestMessage` and `ResponseMessage`, which are translated into HTTP requests and responses by the `NetworkManager`. The provided code has the beginning of `Message` class hierarchy, which can be extended as required. Moreover, the existing classes are incomplete and can be extended and modified as necessary. The concrete `Message` classes for this application will serialise objects into a simple text-based format. Some requests and responses do not require data and, hence, will not require specific concrete classes but can instantiate `RequestMessage` and `ResponseMessage` directly.

The creation of `Messages` is supported by the Abstract Factory Design Pattern. This pattern defines an interface for creating families of related classes, while hiding the specific classes that are created. This allows, for example, to support many different data types (simple plain-text format, XML, etc.) where each concrete factory class would create messages that serialise objects into the desired format, although we will only deal with the simple text-format defined below.

You will need to create an abstract class called `MessageFactory` as well as a concrete factory class called `PlainTextMessageFactory`. The `PlainTextMessageFactory` will create instances of `PlainText…Message` as required, keeping in mind some messages can be handled using `RequestMessage` and `ResponseMessage` directly—for these cases the implementation can be specified on `MessageFactory`, allowing `PlainTextMessageFactory` to simply inherit the default implementations.

Each message creation function of `MessageFactory` returns either a `RequestMessage` (or a derived class) or `ResponseMessage` (or a derived class) and can be used when creating the message from the sender or when receiving the data. The interface defined by `MessageFactory` basically includes a function to create each of the URL Path/HTTP Method pairs of the REST API described above (refer to section REST-like API). Many of the functions are small and will reuse the same classes with only changes in parameters. You may also be able to parameterise the functions themselves to reduce the number of functions required. For example, messages for

- GET /custodians
- GET /asset-types
- GET /assets
- GET /maintenance-records

could be defined in a single function—e.g., `createGetCollection(collectionType)`—that translates the parameter into the appropriate collection URL path.

The following describes the data format of the message content. It requires that the objects be serialised into a plain-text format and be deserialised at the other end. You must determine an appropriate method of serialisation and deserialization: for example, you may use QTextStream and overloaded input/output operators (`>>` and `<<`, respectively) to read and write entities from/to text and subsequently converted into a QByteArray, which is required for the actual payload (see QString::toUtf8()). Note: if you take this approach you will need to overload the operators `QTextStream` not `std::istream`/`std::ostream`; there are some differences between them and the manipulators they supports, so you will need to read the documentation. Note: messages that contain more than one entity will serialise one entity per line.

The basic format of the data structures the output into fields separated by pipe '|' symbols. To protect against additional pipe symbols breaking the format, fields that contain user-entered text must be encoded using *URL percent encoding*, which is an encoding style used by URLs that encoded unfriendly characters using a percent '%' sign followed by a pair of hex digits. Refer to QByteArray::toPercentEncoding() and QByteArray::fromPercentEncoding().

Custodian

The data format for serialising `Custodian` objects is as follows (ignore linebreak, serialisation is one entity per line):

```
Custodian|<id>|<lastEditedBy:%>|<lastEditTime:iso8601>|<name:%>|<department>|<phoneNumber>
```

Where fields in angle brackets denote attributes of `Custodian`, with special instructions included after a colon ':'. For example, <name:%> indicates that the `name` attribute should be percent encoded and written in that position. The instruction, 'iso8601' instruction for date/time data fields indicates it should be serialised in ISO 8601 format (see QDateTime::toString and QDateTime::fromString).

For example, an instance of Custodian may be serialised as:

```
Custodian|cust001|matt|2018-09-14T00:00:00+09:30|Johnny|Maintenance|0880001111
```

AssetType

The data format for serialising `AssetType` is more complex as it needs to serialise the `UserPropertyDefinitions`. The format is as follows (ignore line breaks in format definition, serialisation is one entity per line):

```
AssetType|<id>|<lastEditedBy:%>|<lastEditTime:iso8601>|<name:%>|<propertyDefinitions.size>|<
propertyDefinitions:separated by '|'>
```

Where each `UserPropertyDefinition` in `propertyDefinitions` is serialised as:

```
<name:%>|<type: 'int', 'double', 'string'>|<default value:%>|min=<min validator>,max=<max
validator>,enum=<enum validator>
```

Where <min validator> is the value of the minimum value validator if present (otherwise empty), <max validator> is the value of the maximum value validator, and <enum validator> is a colon separated list of the enumeration values (percent encoded). For example,

```
AssetType|type001|matt|2018-09-14T00:00:00+09:30|Pump|0
AssetType|type002|matt|2018-09-14T00:00:00+09:30|Fan|1|flow|double|0|min=,max=,enum=1.0:2.5
```

Note: when deserialising the data, if the number of property definitions exceeds the number specified in field 6, an error response should be provided: BAD_REQUEST

Asset

The data format for `Assets` is as follows:

```
Asset|<id>|<lastEditedBy:%>|<lastEditTime:iso8601>|<type.id>|<serialNo:%>|<brand:%>|
<model:%>|<purchasePrice>|<purchaseDate:iso8601>|[disposalDate:iso8601]|<custodian.id>|
<properties.size>|<properties:separated by '|'>
```

Each `UserProperty` is serialised as:

```
<name:%>|<value:%>
```

Each `UserProperty` can be reconciled against their definition through name-based look-up on the `AssetType`.

For example,

```
Asset|asset001|matt|2018-09-14T00:00:00+09:30|type002|x23-zz0|
ACME Co.|x23series|100.22|2018-09-14T00:00:00+09:30||cust001|1|flow|2.5
```

```
Asset|asset002|matt|2018-09-14T00:00:00+09:30|type001|P-101|Pumps%27R%27Us||50.00|2018-09-14T00:00:00+09:30|2018-09-15T00:00:00+09:30|cust001
```

Note: %27 is an apostrophe ‘ character

MaintenanceRecord

MaintenanceRecords are serialised as follows:

```
MaintenancRecord|<id>|<lastEditedBy:%>|<lastEditTime:iso8601>|<owner.id>|<timestamp:iso8601>|<performedBy:%>|<maintenanceType>
```

For example:

```
MaintenanceRecord|rec001|matt|2018-09-14T00:00:00+09:30|asset001|2018-09-20T00:00:00+09:30|maintenance guy 01|Inspection
```

Unit Testing

As this is a group project, you will need to work independently for a time before the application starts coming together into an integrated whole. Until then Unit Testing will be essential in ensuring that the code you are producing is correct before you begin integrating your code into a complete application. To that end, the Qt Unit Testing module, Qt Test, has been configured in the provided code for your convenience. Qt Test is quite a simple Unit Testing framework that integrates nicely into Qt-based applications, allowing testing of the classes with Signals & Slots and GUIs, and can be used directly from Qt Creator. A Test Case in Qt Test is simply a class that has private slots (each slot is considered a test) and is registered as a Test Case using a macro. Inside the tests functions themselves, you use macros such as `QVERIFY(…)` and `QCOMPARE(…, …)` to check conditions and log any failures of those conditions to hold. You will need to read up on using Qt Test [http://doc.qt.io/qt-5/qttest-index.html] to make use of framework: one particularly useful feature is “data-driven” testing.

To simplify the test setup a little, there is an extra file ‘autotest.h’ that makes it easier to execute multiple Test Cases at once. Moreover, there is an example test case already setup which demonstrates how to use the framework as well as how to use the networking APIs required for the project (`QHttpServer` and `QNetwordAccessManager`).

When implementing your Unit Tests keep the following guidelines in mind:

- One Test Case per class (or per small number of related classes): e.g., for the `AssetRegister` class you would have a Test Case called `TestAssetRegister`; for `UserProperties` you could have a single Test Case that includes the concrete types for `int`, `double`, and `QString` since they are small and strongly related.
- Keep your tests small: small tests make it easier to determine what a test is doing as well as minimises (accidental) dependencies between parts/variables of the test.
- Test only one thing: related to the above but with a different emphasis, testing only one thing improves the identification of errors by separating assertions (or in the case QVERIFY and QCOMPARE) into different tests, thereby helping to identify more than one symptom of a bug. The more information you have about a bug, the quicker you will be able to find and fix it.
- Write tests for failure: your tests should initially fail, which helps ensure that your tests are not accidentally passing.
- Give your tests good names: this goes without saying as all function, variable, and class names should be useful and descriptive.

Note: Although the source files from the main application project area incorporated into the test project automatically, Qt Creator does not always refresh the test project correctly after adding a file to the application project, which may result in compilation errors (usually 'Undefined reference …'). To resolve this, simply run 'qmake' on the test project: right-click on the project and select it from the drop-down.

Documentation

Your code must be documented appropriately using Doxygen comments. Comment blocks ***must be used in your header*** file to document your ***classes and class members***. Private members should also be documented when it is not obvious for what they are used—but first check if the code can be improved as clean code is better than comments. Use ***comments sparingly in source files***, document blocks of code (switch statements, if else groups, loops, etc.) rather than individual statements: do not comment individual statements, unless the outcome of the statement is not obvious. Optionally, you can add comments to 'main.cpp' to document the 'main page' of the generated HTML documentation.

Code Style

You must write you code conforming to the code style set for this course. Remember, consistency is the primary goal of any style guide: the easier it is to understand your code, the easier it is to maintain (real-world), and allocate marks (hint-hint).
See http://codetips.dpwlabs.com/style-guide

Version Control

You must use version control to keep track of your progress during implementation, preferably using git. You should perform regular commits as you implement features and fix errors. Your commit comments should reflect the ***context*** of the changes that were made in each commit—a fellow developer can always diff the contents to see *exactly* what changed but that does not provide the context. You should try to ensure each commit comment contains a short subject line.

Your submission will comprise your entire version control repository, so it should be specific to the assignment. For example, if you use git, you must create the git repository in the root directory for your assignment. Do not commit your assignment to a repository created in a parent folder. Your repository must contain all source files required to compile and run your program (including a Doxygen config file) but must not include any generated or extraneous files.

You ***must*** use an online version control repository, such as BitBucket, to collaborate with your group. However, you must ensure that the assignment is ***private***, i.e., ***cannot be seen by people other than your group***.

Implementation Rules and Hints

You have been provided with an initial project that contains everything you need to get started. It contains 3 subprojects: asset_manager, asset_manager_tests, and qhttp-embed. The last is the source code for a simple HTTP server that is being used for the project. The first is the folder for the main application, it contains: 'main.cpp' (in a subfolder called 'app'), which starts the application and displays the main windows; 'mainwindow.h/cpp', which is an empty main window (the same as what you get when you create a new GUI project), and some files related to the initial Message class hierarchy (in the 'network' folder). You are free to change any of the files in the asset_manager project except the 'main.cpp' (you should not have to touch that at all). The asset_manager_tests subproject has set up customised QTest environment to simplify the creation of Unit Tests for the project. Moreover, it includes an example test case (`TestExample`, 'tst_testexample.h/cpp') which does two things: 1) provides an example of a Test Case with some test functions, 2) demonstrates how to use the Qhttp library and the standard Qt Network Access Manager for performing the network functions, and 3)

illustrates how to use the `Message` classes to support the networking component—'tst_testexample.h' contains example concrete request and response message types.

You will need to use the Qt data types for this project. You are free to use any of the Qt library classes as you require. Some suggestions have been provided throughout this specification, but you will need to read the Qt documentation yourself as we cannot cover everything during class. You can use C++ STL if you wish; however, Qt classes are generally a better fit for Qt-based GUI applications—there will be less translation to do at the interface between GUI and your code. You may not use any additional libraries (other than those provided).

Your initial UML class design may be incomplete or incorrect. Your first design should attempt to cover the major requirements, in particular the identification of the design patterns. Do not worry if you need to revise and refactor the design as you progress through the assignment. However, the UML design is the primary tool for communicating the structure of the code with your team members; therefore, it is important that it be kept up-to-date as you progress and that your team members are notified of any changes to the structure—ideally you will go through a process of discussion and approval before changes are made.

Once you have your initial diagram, it is a good idea to create (non-functioning) class and function stubs. This will allow your individual team members to begin implementing their portion of the assignment without having compilation issues due to another portion being incomplete. In particular, this will allow Unit Tests to be created. **Remember:** in a test-driven development approach, tests should be written to fail before the implementation is performed and the tests passed. This is to prevent having tests that *accidentally* pass.

It is a good idea to commit the source file (but not the PNG or PDF) of your UML class design to version control. This will allow you to track the evolution of your design as you progress through the assignment.

It is recommended that you test individual components as you go, rather than trying to test the whole application through the interface. Writing code to test specific elements will speed up development as you will not need to constantly enter data through the menu interface. Depending on how you allocate the work, one team member may be responsible for writing the tests; in this case, it is important to focus on implementing tests for high-priority components first.

Be sure to check for null pointers before attempting to use an object through a pointer and reset pointers to null where appropriate.

Consider when and where it is appropriate to use smart pointers vs. bare pointers and the effect this choice may have on constructors, destructors, etc. Do not forget that Qt provides its own set of smart pointers: refer to the documentation on QPointer, QSharedPointer, etc. They operate much like the smart pointers of the C++ STL, but provide additional features, such as thread-safety, that are important within a GUI environment.

Casts in your code should only need to be performed between a base class pointer and a derived class. Be aware that Qt has its own casting functions that work with the Qt smart pointer classes. Refer to the Qt documentation for the particular smart pointers to see what functions are available for casting.

Submission Details

You must submit your complete Qt Creator project folder inside a ***single zip*** file. The zip file must include ***all source files*** required to compile and run your program (including a ***Doxygen config file***) as well as your ***version control directory***: e.g., if using git you must ensure the '.git' folder is present in the zip file. It must not include any generated or extraneous files such as the '*.pro.user' file—the UML image file is the one exception to the no generated files rule. Place your UML class diagram in your zip file. Your diagram ***must*** be submitted in ***PNG*** or ***PDF*** format.

Once you have created the zip file according to the specifications, you are required to upload it to the ***Assessment Item 3: Project*** submission via ***LearnOnline***. Only ***one submission*** is required ***per group***. The deadline for submission is **2 November 2018 11:59 PM**. After which time the submission will be considered late and attract a penalty as detailed in the course outline. Please organise ahead of time who will be responsible for submitting the project to ensure there are no accidental late submissions.

Marking Criteria

Your assignment will be marked both automatically and manually; automatically using unit testing, shell scripts and static code analysis tools. Your code will be inspected for style – remember consistency is the primary goal of all style guides, the easier it is to understand your code, the easier it is to allocate marks.
Note: the criteria with negative marks are penalties that will be applied if the submission does not meet the expected criteria. They are structured such that an otherwise perfect submission will be unable to receive an HD if and only if the *full* penalties are applied.

To Be Released

Academic Misconduct

This is a group assignment: your submitted files will be checked against that of other groups, and other sources, for instances of plagiarism.

Students are reminded that they should be aware of the academic misconduct guidelines available from the University of South Australia website.

Deliberate academic misconduct such as plagiarism is subject to penalties. Information about Academic integrity can be found in Section 9 of the Assessment policies and procedures manual at:

http://www.unisa.edu.au/policies/manual/