Improving Design of Ground Control Station for Unmanned Aerial Vehicle: Borrowing from Design Patterns

Mlađan Jovanović¹ Dušan Starčević² and Zoran Jovanović¹
¹Faculty of Electrical Engineering, ²Faculty of Organizational Sciences, University of Belgrade
mladjan@rcub.bg.ac.rs, starcev@fon.rs, zoran@rcub.bg.ac.rs

Abstract

Reusable software architectures and supporting components are the focus of an increasing number of software organizations attempting to reduce software costs. One essential attribute of reusable software architecture is that it effectively isolates the logical, or static, aspects of the application from any product-specific variations in the physical architecture, or execution environment. A primary element of this isolation is hardware and low-level software (e.g., operating system) independence. A design pattern documents a reference design for the solution to a recurring problem encountered in object-oriented software development. The fundamental theme of design patterns is to encapsulate the concepts that vary. This paper describes our experiences on developing reusable object-oriented software architecture for GCS (Ground Control Station) for UAV (Unmanned Aerial Vehicle) using design patterns. In addition, various ways that software architecture attributes can be designed for flexibility without introducing volatility into the physical architecture are described.

1. Introduction

Due to technological advances and increasing investment, interest in Unmanned Aerial Vehicles (UAVs) as a practical, deployable technological component in many civil applications is rapidly increasing and becoming a reality, as are their capabilities and availability [1]. UAV is a complex and challenging system to develop. It operates autonomously in unknown and dynamically changing environment. This requires different types of subsystems to cooperate. In order to realize all functionalities of the UAV, the software part becomes very complex real-time system expected to execute real-time tasks concurrently [2].

Present research describes an initiative to assess the potential for reuse of GCS UAV software across multiple aircraft platforms, and to define and demonstrate supporting software architecture based upon open commercial hardware, software, standards and practices. This initiative produced a set of reusable object-oriented software. Proposed software architecture employs design patterns as reusable artifacts for software development. Essential aspects of the resulting logical architecture developed therein are described.

This paper is organized as follows: In Section 2 we restate results and principles from related developments. Section 3 defines the design goals of software architecture and describes solution developed toward meeting these goals. Finally, Section 4 closes this paper with a conclusion and a brief discussion.

2. Related Work

2.1. UAV software support

Uninhabited vehicles can be used in many applications and domains, particularly in environments that humans cannot enter (e.g. deep sea) or prefer not to enter (e.g. war zones) [3]. The promise of relatively low cost, highly reliable and effective assets that are not subject to the physical, psychological or training constraints of human pilots has led to much research effort across the world [4].

UAV platforms offer a unique experimental environment for developing, integrating and experimenting with many Artificial Intelligence technologies such as automated planners, knowledge representation systems, chronicle recognition systems, etc. [5].

The integration of ground wireless ad hoc networks and airborne UAVs is a promising solution for real-time data collection in wireless sensor networks [6], [7].

Timely information about highway traffic conditions is very important, especially when traffic incidents or accidents occur. Unmanned aircraft equipped with video cameras and/or other sensors may be able to...
deliver the necessary information through video images with relatively low operational costs and risks to human life [8].

Theunissen et al. [9] introduced a concept to facilitate the integration of UAVs into controlled airspace using a datalink dialog with civil ATC (Air Traffic Control) system. To realize the desired integration, three development activities were performed: 1) the integration of the UAV simulation environment, an ATC simulator and an airspace command and control system into a common simulation environment; 2) the implementation of datalink interfaces to allow the exchange of the messages between the different simulators and 3) the development of the associated functions and user interfaces that will improve completion of the associated tasks.

The company CAE has developed the synthetic environment as a practical representation of the real world for the UAV simulation [10]. A ground control station is being used to operate a simulated UAV in a synthetic environment. The synthetic environment denotes practical representation of the real world comprising the UAV air vehicle simulation, payload simulation, and comprehensive tactical environment.

The Georgia Institute of Technology (GIT) designed and developed an UAV multi-sensor system that provides terrain navigation information [11].

Rasmussen et al. [12] developed MultiUAV, a simulation that is capable of simulating multiple unmanned aircraft vehicles which cooperate to accomplish a predefined mission. The simulation was constructed using the Mathwork’s Simulink simulation software.

Hong et al. [13] proposed a hierarchical architecture for Unmanned Autonomous Helicopter System that guarantees the real-time performance of hard real-time tasks and the re-configurability of soft real-time or non-real-time tasks under the RT-Linux. This software architecture has four layers: hardware, execution, service agent and remote user interface layer according to the reactiveness level for external events. In addition, the layered separation of concurrent tasks makes different kinds of mission reconfiguration possible in the system.

Modeling is becoming increasingly important since models are used as “filters” or “placeholders” for not-yet-developed systems elements allowing system elements to come “on-line” when they are ready, rather than waiting for the entire system(s) maturation [9],[14]. However, only limited research efforts are concerned with the environments, methods, and tools for the systematic and formalized development of software for UAV GCS [2], [7]. A number of researches concentrated on embedded software development for supporting the particular area of UAV applicability while considering employing software maintenance, adaptability and reusability paradigms as completely separated from the actual software development process [2], [4]-[14].

2.2. The Role of Design Patterns

A design pattern is normally described as a solution to a recurring design problem in a particular context [15]. To this definition, we add a crucial point: a design pattern is a fluid, adaptable solution. It is key to recognize that a pattern does not represent a single, static solution to problems within its context. A design pattern is an outline of a solution that must be adapted for its eventual use [16]. A pattern is better thought of as a family of design solutions. A pattern description describes the basic structure of the family members and outlines common ways this structure can be specialized for different applications.

Design patterns have increasingly gained acceptance not only as reusable constructs for software development but also the documentation and comprehension of the architectural design of a software system [17], [18].

The basic problem with design patterns is that they are abstract, reusable designs that exist only as documents. A pattern describes the basic structural elements and their interactions that solve a design problem within its context. However, this is an outline of the solution; the structure must be adapted to the specific characteristics of the application. There is no concrete representation that can be reused—a pattern must be implemented each time it is used in a design. At least part of the reason for this is that most patterns take on application specific characteristics. An obvious solution is to provide an object-oriented framework that implements the design pattern [19]. In this sense, a framework is a basic outline of a particular application, such as a graphical user interface [20]. The framework implements the flow of control through the classes and objects that make up the application, and application logic is inserted in hook methods. The framework could capture the commonalities in the different pattern variations as well as flexibility and adaptability that are essential to capture the full utility of design patterns.
3. Proposed Framework

3.1. Architecture Goals

Given the goals for UAV software architecture, two key goals of the software architecture itself were enumerated:

1) Contain change.
2) Maximize reusability.

Naturally, there are many types of change that appear in a complex UAV mission computing software:

1) Differences in avionics subsystems (e.g. radar, cockpit controls and displays, INS)
2) Differences in mission computing hardware – both in terms of number of processors and in terms of the exact microprocessor chosen.
3) Differences in system requirements.

The established adage of “encapsulate change” can be somewhat broadened to fundamental theme of design patterns, “encapsulating the concept that varies” [15]. This encapsulation promotes design reuse across the releases of the software application [21]. It is commonly envisaged that by such organization of code base and design, software reuse could lead to substantial gains in productivity [22], [23].

A goal to maximize reusability equates to minimizing cost in a reuse-driven affordability process. One corollary to this goal is that the developed software must be granular enough that specific aircraft products can select the exact software elements applicable to their system. This requires elements with the ability to be plugged together to create a deliverable system. This also requires elements with few underlying assumptions about execution environment in which they are employed.

For the purposes of this discussion, physical architecture is defined as the hardware and run-time software structures that control the use of processing and communication resources, i.e., the dynamic runtime view of the system. In contrast, the logical architecture is defined as the design elements, the element development model, and inter-element structural relationships, i.e., the static compile-time view of the system.

3.2. Software Architecture

The operator may observe the system through four separate views [24]. First presents an instrument table for monitoring basic flight data and state of the vehicle. Second displays terrain map together with terrain profile and serves for mission tracking. Third view presents three-dimensional representation of the environment and visualizes vehicle’s exact position and orientation. Forth view renders real-time video stream from camera mounted on the aircraft. In addition, UAV receives remote-control commands from separated manual console in order to perform such actions as taking off, landing, changing its direction of flight or directing a camera at a target. Once the UAV is airborne, the operator is responsible for manning flying platform using the console.

Our approach is inspired by the model-driven development, where software development’s primary focus and products are models rather than computer programs. In this way, it is possible to use concepts that are much less bound to underlying technology and are much closer to the problem domain [25].

The overview of the proposed GCS software architecture is shown in Figure 1.

![Fig. 1. Overview of the proposed software architecture.](image)

Environment state data and flight platform data are produced as messages complying with the specific formats and are processed by corresponding modules to make them suitable for visual presentation. Terrain and navigational modules visualize terrain environment and waypoints tracking. User also performs monitoring task and identifies failure when it occurs. This is achieved with aircraft instrument presentation module. Through the network interface system receives data from the aircraft. These data includes environment state and flight platform data, as well as real-time video stream incoming from the aircraft. Planning and plan execution tasks base its decisions on information derived from the camera vision subsystem. In addition, terrain view visualizes aircraft’s environment generated from digital terrain model as well as vehicle’s exact position and orientation.

Figure 2 presents a Unified Modelling Language (UML) [26] deployment diagram which describes possible system landscape. Part of the communication infrastructure for receiving data from flying platform includes video receiver which converts wireless analog...
video signal into PAL/NTSC format; video encoder which digitally encodes the video and flight data receiver. Central computer is connected with these devices and contains corresponding software modules (Datasource, Instruments, Map, Camera, 3DNavigation). Produced messages and video streams can be further passed to interested clients through computer network.

Fig. 2. Possible physical deployment of the proposed software architecture.

3.3. Design Details

Following subsections give a closer look at some important issues concerned with realization of the proposed framework. Framework is introduced by describing software modules whose development was driven by principles established in overall software architecture goals mentioned above.

3.3.1. Aircraft Instruments

UAV operators can incur a high workload due to the fact that today's modern aircrafts produce vast amount of data which has to be presented in real-time. In situations where operator must react in a limited period of time and avoid hazardous situations, it is very important to present flight data in a form that can be easily interpreted and processed having in mind throughput of human sensory and perceptual apparatus. Requirements for aircraft instruments' design are as follows:

- Cognitive goal. In order to decrease cognitive fatigue, controls should operate in a way that represents operator’s intuitive understanding.
- Response goal. This concerns minimizing UAV response time and is achieved by underlying implementation technology.

In order to streamline and optimize operator-vehicle interface we have decided to classify instrument types and to model the structure of an instrument as is shown in Figure 3.

Fig. 3. Simplified UML description of instrument types hierarchy and structure of an instrument.

AbstractInstrument presents an abstraction of common properties for all types of instruments. VisualInstrument and ComplexInstrument are derived from base class. VisualInstrument is further derived according to specific kinds of instruments which can be found in aircraft cockpits. Introduction of ComplexInstrument type enables modeling of composite instrument containing other instrument types. This kind of realization of instruments hierarchy presents an example of Composite design pattern [15]. By using Composite, one can recursively create with composites - i.e. containers - and leaves either complex or hierarchical structures like trees. From the user's point of view, this pattern provides a unique interface. So, the user can address in the same way leaves and containers. In our case, presented approach to modeling aircraft instruments enables construction of instrument tables of random complexity and layout of instruments.

The process of creation of instruments in runtime (Fig. 4) is realized using Factory Method design pattern [15].

Fig. 4. Simplified UML illustration of structure for creating instruments.
The factory method pattern is a creational design pattern. Like other creational patterns [15], it deals with the problem of creating objects (products) without specifying the exact class of object that will be created. The factory method design pattern handles this problem by defining a separate method for creating the objects of different types. More generally, the term factory method is often used to refer to any method whose main purpose is creation of objects. Factory methods are common in toolkits and frameworks where library code needs to create objects of types which may be subclassed by applications using the framework [19], [27]. Our approach employs the pattern in order to centralize and automate the process of creating specific types of instruments found in aircraft cockpits.

An example of concrete instrument table implemented from underlying model is shown in Figure 5. Presented view operates in a way that represents an operator’s intuitive understanding. Controls that have different functions are distinguishable from one another in order to clearly assess flight status data. Instruments and controls with related functions are grouped together in a logical arrangement which helps reduce instrument scan time and lowers operator’s workload. An approach to design of user interfaces is presented in [28].

![Fig. 5. Possible realization of a software instrument table.](image)

3.3.2. Mission Navigation

UAV missions are monitored and controlled by two-dimensional navigational view and three-dimensional terrain view. Figure 6 describes a simplified UML description of module for controlling and navigating UAV missions. Module’s main frame (MapMainFrame) includes instances of three views: route navigation view (MapVerticalViewPanel), terrain profile view (MapProfilePanel) and mission data control view (MapDataPanel). Map connection thread (MapConnectionThread) is responsible for communicating with Datasource module and keeping the corresponding views up to date. Since planning of missions is also allowed, communication between views must be established. In our case, it is realized using Mediator design pattern [15]. With the mediator pattern communication between objects is encapsulated with a mediator object. Objects no longer communicate directly with each other, but instead communicate through the mediator. This reduces the dependencies between communicating objects, thereby lowering the coupling and improving maintenance and/or refactoring [29], [30].

![Fig. 6. UML description of UAV mission navigation.](image)

Process of rendering of route navigation view and terrain profile view is realized relying on Observer design pattern [15]. Classes and interfaces which realize rendering of corresponding views are shown in Figure 7.

![Fig. 7. The UML structure of mission navigation rendering mechanism.](image)
The intent of Observer is to keep a set of objects, i.e. the observers, up to date when the state of an object they depend on, i.e. the subject, has changed. In other words, this pattern implements a one-to-many dependency between the subject and the observers. Concretely, it allows you to attach anonymously a set of observers to a subject. Then, when the state of the subject changes it automatically invokes the callback update method of each observer. In our approach, each view is comprised of number of layers. Each layer is presented and rendered by corresponding observer object. In this way it is possible to detach/attach layers to corresponding views without affecting the rest of the scene. In addition, introduction of layers for visualizing specific kind of data (i.e. weather forecast, firefighting, landslide, traffic signalization, etc.) is facilitated by adding observer object that will conform to the specific interface in corresponding branch of hierarchy.

Figure 8 shows an example of mission control navigation display realized according to previously described models. Navigational view comprises terrain profile view (bottom), route navigation view (top) and data control view (left). Each mission consists of number of routes containing waypoints that need to be visited and reported. Operator manually enters waypoints positions. Based on their positions planned routes are calculated. During the flight aircraft can deviate from planned route and operator is allowed to correct aircraft’s position and orientation using manual control console through three-dimensional terrain view.

Rendering of three-dimensional terrain view employs terrain description which is built from elevation data on a regularly spaced grid. These data are tiled with triangle strip arrays. Figure 9 gives an example of rendered three-dimensional terrain environment. Each elevation point is depicted with nine parameters describing color, normal and coordinates. Parameters are contained by reference and shared between user and core graphics routines which allow more efficient memory usage. In addition, three-dimensional terrain view model realizes standard requirements imposed to similar systems, such as rendering LOD (level of detail) mechanism and 6-DOF (six degree-of-freedom) model.

3.3.3. Network Interface

One of the fundamental capabilities of the proposed architecture is to receive messages provided by the UAV’s sensors as well as real-time video stream from the camera mounted on the aircraft.

Network communication software is encapsulated in Datasource module and has two parts. One part is a set of Network daemon threads that is able to read messages incoming from flight data receiver and distribute them to interested modules which can be deployed in a local or remote environment. Specialized classes are designed to be called by an application that maintains the state of the environment and the vehicle. This description is updated as messages from the aircraft are being received. Communication between Datasource and other modules is realized using Remote Proxy design pattern [15] as is shown in Figure 10. ServerConnectionThread maintains collection of ClientConnectionThread instances which present local representative of the remote client. In this instance, the remote client is Instruments module, i.e. InstrumentsConnectionThread which keeps instrument table up to date. Similar realization has already been introduced by Sharp [31] on the example of Boeing avionics software.
The intent of the Proxy is to provide a surrogate object to a real object. Actually, the surrogate receives client method calls and invokes the same method on the real object. The surrogate object and the real object share the same interface or super class. Hence, the client is not aware that it is calling Proxy’s methods rather than the methods of real object. We have applied this pattern for remote invocation.

### 3.3.4. Camera Vision

Additional part presents subsystem for processing video stream incoming from the vehicle. On the side of the UAV the onboard computer system is integrated with camera gimbal mounted beneath the aircraft fuselage. To support UAV operations, a video camera on the UAV produces a video stream that must be displayed with minimal real-time delay on display console. There are several steps to this process:

- Video feed from off-board source (UAV).
- Sending video to hosts on aircraft’s network.
- Users’ hosts receive video and display it.
- Users analyze received data and send commands to UAV to control it.

Our architecture supports the first three of these steps. The fourth step, in which operator interacts with the UAV in real time, is manifested in a QoS requirement: at least on certain display consoles, images must appear a very short time after the camera records them; we cannot buffer or interrupt the stream in order to accommodate variable latency and periods of network congestion.

![Fig. 11. Camera view realization.](image)

**3.4. Implementation Details**

Developed models may be used for automation of some phases of the design of UAV GCS. We used the UML to describe these models. In order to validate the framework, we have been developing tools for generation of software components. These tools take developed models as an input, and produce modules (packages) containing Java code files which make a skeleton of a designed software GCS.

Upon developed framework we have built solution that uses RTSJ (Real Time Specification for Java) implementation as a middleware layer of software architecture. Figure 12 gives a view of software architecture in terms of software modules and their layered organization. Application layer – comprises software modules which make the design of our system; Middleware – comprised of corresponding implementation platform libraries; System layer – includes system level software and libraries.

![Fig. 12. Layered organization of software architecture.](image)

RTSJ [32] was designed with the ability to combine plain Java components with the real-time ones in a type-safe and higher performance manner. The Ovm virtual machine [33] presents an open source implementation of the RTSJ virtual machine and has shown a promising role in development of lightweight middleware for embedded systems [34]. It implements some core VM features such as thread scheduling and memory management and is used as a core execution environment in the developed system. Given the same constraints placed on large scale real-time embedded C++ applications, the Ovm running RTSJ classes provide comparable performance [34]. In general, the Java language itself offered better portability and productivity over a traditional language such as C++. The main concern expressed was about the level of maturity of tools and vendor support.

This paper restates principle results from previously published work [35] where details concerning experimental evaluation of the software architecture are given.
More recent research on extending standard software development techniques for modeling embedded real-time systems both in terms of expressing models [36]-[39] and defining straightforward action semantics for executing models by compiling them into a programming language to target specific implementation [40]-[42] has shown promising role for model-driven development of real time and embedded systems. In this sense, we see the ability of the mechanisms described in this paper to be extended in ways important in fulfilling emerging operational requirements for more highly dynamic and adaptive system resource management.

4. Conclusion and Discussion

Present work describes an approach to improve design of software for UAV GCS with regard to software reusability, adaptability, maintainability and productivity issues. The framework anticipates four separate views. First presents an instrument table for monitoring basic flight data and state of the vehicle. Second displays terrain map together with terrain profile and serves for mission tracking. Third view presents three-dimensional representation of the environment and visualizes vehicle’s exact position and orientation. Forth view renders real-time video stream from camera mounted on the aircraft.

Design patterns organize the design concepts against various fragments of the application code base systematically. They have been regarded as successful artifacts for reusable object-oriented software development. The success of an UAV GCS application rests in large part on the developer’s ability to remove specific software dependencies and intercomponent dependencies as both of these factors induce variability. Fostering the adaptation of design patterns in typical software application development can lead to extensible software and substantial gains in productivity.

Our experience has been that the software architecture of the subject application must be and can be made capable of absorbing these dependencies, and that a careful combination of standard patterns satisfies many of the design forces. Separating architecture concerns in this way allows reuse of software architecture mechanisms, provides centralized modifiability of these aspects and facilitates the primary goal of more highly reusable application components. Framework built upon proposed architecture has completed development and has been tested. These efforts have reinforced both the feasibility and the challenges associated with applying these architectural techniques to large-scale real-time systems.

With the proposed solution we can apply this to the actual unmanned aircraft and can confirm its exact operation. Also, the proposed solution can be applied to various unmanned control systems. Our current and future work involves the improvement of the proposed framework with UML support for model-driven development of real time and embedded systems.

5. References