Design Patterns for Gradual Composition of Adaptive Graphical User Interfaces

Samuel Longchamps, Ruben Gonzalez-Rubio

Université de Sherbrooke
Sherbrooke, Québec, Canada
Email: {samuel.longchamps, ruben.gonzalez-rubio}@usherbrooke.ca

***Abstract*—Graphical user interfaces (GUI) in modern software are increasingly required to adapt themselves to various situations and users, rendering their development more complex. To handle complexity, we present in this paper three design patterns, *Monitor*, *Proxy router* and *Adaptive component*, as solutions to the gradual implementation of adaptive behavior in GUI and general component-based software. Rather than proposing new adaptation mechanisms, we aim at formalizing a basic structure for progressive addition of different mechanisms throughout the development cycle. To do so, previous work on the subject of design patterns oriented toward adaptation is explored and concepts related to similar concerns are extracted and generalized in the new patterns. These patterns are implemented in a reference Python library called AdaptivePy, which is used to provide practical examples of their applications. Also, a GUI application case study is presented and compared to a functionally equivalent *ad hoc* implementation. We observe that separation of concerns is promoted by the patterns and testability potential is improved. Moreover, adaptation of widgets can be previewed within a graphical editor. This approach is closer to the standard workflow for GUI development, which is not possible with the *ad hoc* solution. Because the patterns suit any components-based software, they can be applied together or individually in different applications to solve specific adaptation challenges.**

Keywords–adaptive; design pattern; graphical user interface; context; library.

I. Introduction

Modern software application developers face many challenges, but one recurring challenge is to build their software in such a way that it can be used on different platforms, by different types of persons and in a variety of contexts. While an application has a some specific purpose, there are many ways to provide the service it offers such that all users are satisfied.

An example of this principle in our daily lives is a bank. While its core business is to keep their customers' money safe and make it grow, they offer a variety of packages to suit different types of customers. A bank also interacts with its customers differently depending on their knowledge regarding the financial market.

Implementing such adaptive behavior in software applications remains a challenge. As applications become increasingly complex and distributed, many implement adaptation in an *ad hoc* manner and recurrent solutions have rarely been formalized. One area of modern applications where adaptation requirements have flourished is graphical user interfaces (GUI). Because they are generally engineered using a descriptive language and oriented toward specific platforms, it is hard to produce a single GUI, which automatically adapts itself to its multiple usage contexts [1].

Many researchers have proposed models and frameworks to implement adaptive behavior in a generic manner for components-based software [2]–[5]. These solutions typically require significant effort to modify an existing software architecture and make specific technological choices and assumptions. They are limited both in terms of gradual integration to the software and in portability, for a framework usually targets a certain application domain (e.g., distributed client-server systems). As a more portable approach, we propose to use design patterns for formalizing structures of components that can be easily composed to produce specialized adaptive mechanisms. While some work has been done to propose design patterns for the implementation of common adaptive mechanisms [6]–[9], the present work aims at generalizing widespread concepts used in these patterns. In doing so, their integration in existing software is expected to be easier and more predictable.

As a proof-of-concept, a reference implementation of the design patterns has been done as a Python library called AdaptivePy. An application was built as a case study using the library to validate the gains provided by the patterns compared to an *ad hoc* solution. Special attention was paid to the compatibility to modern GUI design workflow. In fact, rather than create a specialized toolkit or create a custom designer tool that would include the design patterns' artifacts, the Qt cross-platform toolkit along with the Qt Designer graphical editor were used. The application workflow is presented and compared to original methods and advantages are highlighted. We expect that through the case study, the patterns' usage and advantages will be clearer and offer hints on how to structure an adaptive GUI.

This paper is an extended version of a conference paper published in the proceedings of Adaptive 2017 held in Athens, Greece [1]. We extend on the previous paper by providing a more in-depth description of the patterns and by providing additional examples as practical demonstrations of how to apply the patterns using AdaptivePy. Also, more specialized challenges related to the design of adaptive applications are identified and solved with minimal code examples

The remainder of this paper is organized as follows. Fundamental concepts of software adaptation extracted from previous work are described in Section II. The design patterns inspired from the concepts are presented in Section III. AdaptivePy is presented and followed by practical example usages in Section IV. The prototype application with adaptive GUI is presented in Section V and an analysis of the gains procured by the use of the proposed design patterns are presented in Section VI. The paper concludes with Section VII and some future work is discussed.

II. Concepts of Software Adaptation

This section presents major concepts of adaptation from related work classified in three concerns: data monitoring, adaptation schemes and adaptation strategies.

A. Adaptation Data Monitoring

The context of a software refers to the environment in which it is executed. Example contextual data are the geographical position, light intensity, temperature, but also the computing platform on which an application is executed. A computing platform can be composed of many parts such as hardware components, operating system, computing capability and, in the case of GUI, user-interface toolkit [10].

Contextual data on which customization control rely, referred to as *adaptation data* in this paper, can come from various sources, both internal (for "self-aware" applications) and external (for "self-situated" or "context-aware" applications) [11]. Given these two types of adaptation data, we consider a system fully *adaptable* if it can both be customized based on internal and external adaptation data. If the system is autonomous in the control of both matter, then it is considered fully *adaptive*. The level of adaptivity and adaptability can provide more or less control over customization and flexibility of adaptation. Each application use case can benefit from a certain level of both [12]. A challenge is therefore to make it easy to implement and change the level of adaptivity and adaptability wanted for any application feature throughout the development process.

The acquisition of contextual data to be used as adaptation data is part of a primitive level, which is necessary for other more complex adaptation capabilities to be implemented [13]. Contextual data is usually acquired by a monitoring entity (sensors/probes/monitors) responsible for quantizing properties of the physical world or internal state of an application [8], [14]–[18]. Multiple simple sensors can be composed to form a complex sensor, which provides higher-level contextual data (Sensor Factory pattern [18]). Internal contextual data can be acquired simply by using a component's interface, but when the interface does not provide the necessary methods, introspection can be used (Reflective Monitoring [18]). When a variety of adaptation data is monitored, it provides a modeled view of the software context, which may be shared within a group of components. Some event-based mechanism with registry entities can be used to propagate adaptation data to interested components (Content-based Routing [18]). Quantization can be done on multiple abstraction levels and thresholds can be used to trigger adaptation events (Adaptation Detector [18]). This is then used to proactively alert some external adaptation mechanism to perform a selection of the most appropriate components and check if no system constraints are violated.

A system using external data for adaptation would be considered self-situated while one using internal data would be considered self-aware [11]. Self-situated systems are also referred to as context-aware, where context is the operational environment [19]. Context-aware will be used in this paper rather than self-situated to emphasize the distinction between self (internal) and context (external) as categories of adaptation data.

Self-awareness is a basic requirement for self-adaptivity since it is through a representation of itself that a system can deduce how it satisfies given constraints and modify itself to improve their satisfaction. Self-awareness can be achieved through self-monitoring of a system's components by software entities as it is the case for autonomic managers in the MAPE-K model [11].

A recurrent problem shared by any monitoring system is the need for agreement between components, which perceive different contexts, e.g., when there is no centralized coordination controller. To be tackled, this problem needs a form of structure for synchronization and sharing of data. Another major challenge is the inherent complexity of managing, requesting and using adaptation data. Testability of components requiring certain adaptation data is finally undermined since each different value potentially lead to a different behavior of the component and every other depending on it. There is a need to explicitly evaluate expected ranges of monitored adaptation data and prevent contextual view mismatch between interacting components.

B. Adaptation Schemes in Components

Four main types of adaptation concerns or objectives have been proposed by Hinchey and Sterritt [11]: self-configuration, self-healing, self-optimization and self-protection. Different qualities a system must have to enable these objectives are to be self-aware, self-situated, self-monitored and self-adjusted. While some concrete solutions have been proposed for self-optimization and self-healing [20], our main concern for the design of GUI is self-configuration. We synthesize two prominent types of adaptation schemes for self-configuration used in components-based software engineering: *component substitution* and *parametric adaptation*.

a) ***Component substitution****:* The underlying principle of component substitution is to replace a component by a functionally equivalent one with regard to a certain set of features. This can also be done by adding an indirection level to the dispatching of requests and forwarding them to the appropriate component. The first pattern applying this concept is probably the Virtual Component pattern by Corsaro, Schmidt, Klefstad, *et al.* [6]. It is similar to the adaptive component proposed by Chen, Hiltunen, and Schlichting [21], but adds the principle of dynamic (un)loading of substitution candidates. In both cases, an abstract proxy is used to dispatch requests to a concrete component, which is kept hidden from the client. This approach is also used by Menasce, Sousa, Malek, *et al.* [22], who proposed architectural patterns to improve quality of service on a by-request dispatch to one or many components. To maintain the software in a valid state before, during and after the substitution, many techniques have been proposed, such as transiting a component to a quiescent state [23], [24] and buffering requests [25]. State transfer between components can be used when possible, otherwise the computing job must be restarted [21], [24]. An application of this principle in GUI could be to replace a checkbox by a switch. This is seen in touch-enabled GUI where a mouse or a touch panel can be used as a pointing device.

b) ***Parametric adaptation****:* Rather than substituting a whole component by a more appropriate one, parametric adaptation relates to how a component can adapt itself to be more appropriate to a situation. This is usually done by tuning *knobs*, configurable units in a component (e.g., variables used in a computation). Knobs can be exposed in a *tunability*

interface [2] for use by external control components, either included by design or automatically generated at the meta-programming level (e.g., with special language constructs, such as annotations [13]). An example of this adaptation pattern is how different implementations of an algorithm are chosen based on their respective tradeoffs between quality metrics (performance, precision, resources usage, etc.). The tunability domain of each knob is explicit and may vary over time. For example, if a new algorithm is discovered in the middle of a large computing job, an adaptation mechanism that is kept aware of the knob's possible values is able to switch to it if it judges that it will perform better overall [26]. The difference between parametrization and customization through an application's business logic is subtle and can be subjective to a certain point. Many applications apply customization on the basis of information we consider as adaptation data. One major difference that can be identified is that when a component is customized, the knowledge of what can be customized is not shared explicitly to other components as an adaptation space. This adaptation space is the domain of customization that can *safely* be applied.

A current problem is that these two types of adaptation, component substitution and parametrization, are rarely if at all used cooperatively. A software might be adaptive in that it reconfigures its architecture by swapping components, but the concrete components remain unchanged and the adaptation knowledge is centralized, often in the form of rule-based constraints, which are limited in reusability and reside at a higher abstraction level than the individual components. On the other hand, when a component uses data to adapt its behavior, this knowledge is hidden as implementation detail and the limits of its adaptation space are unknown to other parts of the system. The difficulty to acquire and interpret this knowledge is a limitation of both approaches that can be tackled by including it in a basic structure of adaptive component. Furthermore, the ability to reason about how adaptivity constrains and impacts components is a key information which can be used by adaptation mechanisms. Making this knowledge both explicit and accessible is therefore desirable.

C. Adaptation Strategies

No single adaptation strategy is universal for all software. Most past work has been done on applying component substitution using various strategies. For example, many researchers have explored rule-based constraints along with an optimization engine to devise architectural reconfiguration plans [2], [16]. This popular approach has tainted proposed frameworks that tend to be limited to this strategy only. An important principle is that strategies are separate from the component's implementation and can be easily changed. In fact, it is desirable to externalize adaptation strategies in order to be able to easily develop, modify and test them separately. Ramirez [8] calls this class of design patterns "decision-making", since they relate to when and how adaptation is performed. Because these design patterns are concrete adaptation strategies, their artifacts are mainly related to specific strategies (e.g., inference engines, rules, satisfaction evaluation functions). The approach of this class of patterns is typically related to rule-based constraints solving, but a more general goal is to select which plan or components from a set to reconfigure the system with.

III. Design Patterns

This section presents design patterns that realize the concepts presented in Section II with some improvements. When used together, we believe they provide the sought structure for adaptive software. Unified modeling language (UML) diagrams are used to show the structure of the patterns in a standardized way.

A. Monitor Pattern

Classification: Monitor and analyze.
Intent: A monitor provides a value for one type of adaptation data to interested entities.
Motivation: There is a need to quantize raw contextual data as parameters of adaptation data with explicitly defined domain and in specialized modules decoupled from the rest of the application. Adaptation data needs to be reasoned about in arbitrarily high abstraction level and be proactive in the adaptation detection process. Agreement for monitored data should be implied by design in order to allow for safe information sharing among interacting components.
Structure: Fig. 1 shows the structure of the monitor pattern as a UML diagram.

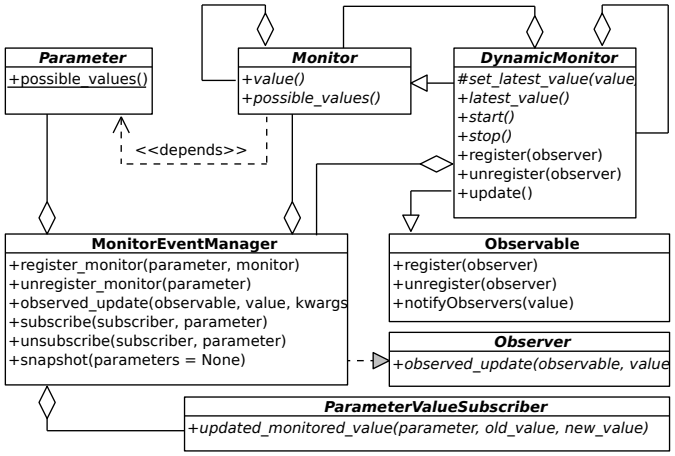

Figure 1. Monitor pattern UML diagram

Participants:

- **Parameter:** A parameter is one type of adaptation data as defined in Section II-A. Its possible values domain is explicitly defined and forms a state space. Many range types can be used to model a parameter's domain.
- **(Static) Monitor:** Provides a stateless (further referred to as "static") means of acquiring a value within a subset of a certain parameter's domain. Formally, $\Omega_M \subseteq \Omega_P$ for possible values Ω of monitor M and parameter P. A monitor can be an aggregation of other static monitors, but not of dynamic monitors.
- **Dynamic monitor:** Additionally to providing a value for a parameter, schedules the acquisition of the value and alerts an observer that a new value has been acquired. Some form of polling or interrupt-based thread awakening needs to be employed along with a previous value to know if the value has changed compared to the latest value, in which case an event

notification is triggered to interested entities. This makes it inherently stateful. Like a static monitor, it can be an aggregation of other monitors. The particularity is that it can aggregate both static and dynamic monitors.

- **Monitor event manager:** Registry entity that allows for a client component to subscribe to a parameter and be alerted when a new value is acquired. Similarly, a dynamic monitor can be registered within the manager and provide a value to any subscriber of the corresponding parameter. In such manager, monitors and parameters are related by a one-to-one relationship; a given parameter can only be monitored by a single monitor.
- **Observable/Observer:** See Gang of Four observer pattern [27]. Used for monitor registering mechanism.
- **Parameter value subscriber:** Provides a means to be notified when a new value of a parameter it has subscribed to has been acquired.

Behavior: An adaptation data type can be formalized as a parameter in terms of the quantized values the system expects to use. A static monitor provides a means to concretely quantize raw contextual data from a sensor or introspection to a value within a defined domain expected by the system. The quantization can be done using fixed or variable thresholds. A dynamic monitor adds scheduling behavior, which allows to provide a value based on accumulated data over time and apply filtering. The monitor event manager is alerted by monitors and dispatches the new value to related subscribers. The dependency regarding subscribers is with the parameters for which they requested to be notified, but actual monitoring is done separately.
Consequences: As monitors are hierarchically built, higher-level abstraction information can be provided. This pattern allows the analysis step of a MAPE-K loop [15] to be done through hierarchical construction of monitors: a parameter can define high-level domain values that are provided by a monitor composed from lower-level ones and components can use this to simplify their adaptation strategies. High-level adaptivity logic is reusable in that parameters are abstract and can easily be shared among projects. Monitors can be chained such that only the concrete data acquisition has to be redone between projects, keeping scheduling and filtering as reusable entities.
Constraints: To assure agreement between interacting components, it is necessary for adaptive components which depend on a common parameter to also subscribe to the same monitor event manager. These components are therefore part of the same *monitoring group*. This can be checked statically or be assumed by contract. The need for a one-to-one relationship between a monitor and a parameter within a monitoring group is based on this agreement requirement. A monitoring group can be thought of as a single entity that cannot have duplicate or contradicting attributes, e.g., it cannot be at two positions at once. In this example, an attribute is a parameter and a monitor is the entity providing the value for this attribute.
Related patterns: Sensor factory, reflective monitoring, content-based routing, adaptation detector [8], information sharing, observer [27].

B. Proxy Router Pattern

Classification: Plan and execute.
Intent: A proxy router allows to route calls of a proxy to a component chosen among a set of candidates using a designated strategy.
Motivation: When implementing component substitution, a way to clearly separate concerns relating to the adaptation logic (choice of substitution candidate) and the execution of substitution (replacing a component or forwarding calls to it) are difficult to implement in an extensible way. The proxy pattern [27] allows to forward calls to a designated instance, but does not specify how control of the routing process should be implemented. Candidate components need to be specified in a way that does not necessitate immediate loading or instantiation and that is mutable (to allow runtime discovery). To maximize reusability, strategies should be devised externally.
Structure: Fig. 2 shows the structure of the proxy router pattern as a UML diagram.

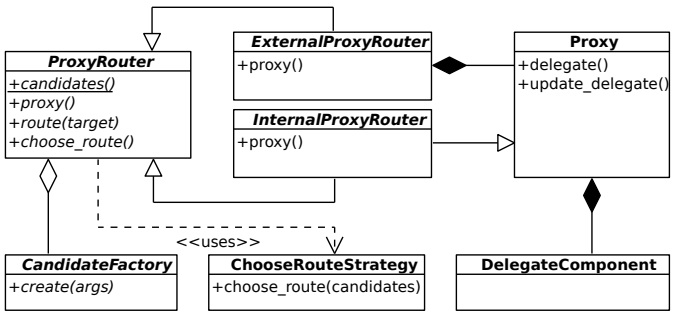

Figure 2. Proxy router pattern UML diagram

Participants:

- **Proxy:** Gang of Four [27] proxy pattern, with the exception that the interface is not necessarily specified (e.g., forwarding to introspected methods). It is responsible for making sure no calls are lost when a new delegate is set.
- **Delegate component:** Concrete component that is proxied. It must be specified as part of the proxy router's candidates set.
- **Proxy router:** Keeps a set of component candidates and allows to control the routing of the calls a proxy receives to the appropriate candidate chosen by some strategy. The proxy router is responsible for ensuring any state transfer and initialization of candidate instances.
- **Candidate factory:** Gang of Four [27] factory pattern for a candidate. Used as part of candidates definition. Can do local loading/unloading for external candidates.
- **Choose route strategy:** Concrete strategy to choose which candidate among a set to use, based on Gang of Four [27] strategy pattern. It uses accessible information from the application, candidates (e.g., adaptation space, descriptor, static methods) or any inference engine available to make a choice.
- **External/Internal proxy router:** Depending on the use, a proxy router can *use* an external proxy (as

a member) or internally *be* a proxy (through inheritance). To allow for both schemes, a means to acquire the proxy is provided and returns either the member object (external) or a reference to the proxy router itself (internal).

Behavior: A set of candidates is either statically specified or discovered at runtime (e.g., looking for libraries providing candidates). The proxy router is then initialized by choosing a candidate using the strategy and controls the proxy to set an instance of the chosen candidate as active delegate. At any time, a new candidate can be chosen and set as active delegate of the proxy.
Consequences: The proxy router pattern allows for flexible and extensible specification of component substitution. The strategies to choose a candidate to route to can be reused in any project with consistent information acquisition infrastructure, such as the one provided by the monitor pattern. Candidates need not be specified statically and control related to routing can be done both internally and externally.
Constraints: Strategies might rely on certain project specific information that is not portable. Separating specific from generally applicable strategies and composing them should help with this constraint.
Related patterns: Adaptive component [21], virtual component [6], master-slave [28], component insertion/removal, server reconfiguration [8], proxy [27].

C. Adaptive Component Pattern

Classification: Analyze and plan.
Intent: Use monitored adaptation data to control parametric adaptation and component substitution by making adaptation spaces explicit.
Motivation: A basic structure is needed to easily add adaptive behavior in the form of parametrization or substitution. Components need a way to explicitly provide means for adaptation strategies to reason about their adaptation space in order to formulate plans. This information should be external to a base component if the adaptation is to be added gradually. Most importantly, an adaptive component must behave like any non-adaptive component and be used among them without side effects on the rest of the system. Complementarily to monitors, which provide values within a domain explicitly defined by a parameter, components require a certain domain of values they support and are expected, by contract, to adapt themselves to (parametrically or by substitution). This domain is an *adaptation space* that can be reasoned about to devise efficient adaptation strategies.
Participants:

- **Adaptive:** An adaptive component that defines means for acquiring the adaptation space. It can be used as a subscriber to a parameter value provider. The adaptation space is a dictionary of parameters with a set of values it supports. To acquire monitored values, it has a reference to one and only parameter value provider. It can therefore subscribe to a parameter and receive updates when a new value is detected, triggering parametric adaptation when needed. If an unexpected value (outside its adaptation space) is received, an exception can be raised and some higher-level adaptation mechanism can be fired (e.g., substitute the component for another one).
- **Monitor event manager:** Parameter value provider realized with the monitor pattern (see Section III-A).
- **Parameter value subscriber:** Provides a means to be notified when a new value of a parameter it has subscribed to has been acquired (see Section III-A).
- **Proxy router:** Proxy router pattern (see Section III-B)
- **Adaptive proxy router:** Adaptive version of a proxy router allowing to drive the routing process (substitution) using monitored data.

Structure: Fig. 3 shows the structure of the adaptive component pattern as a UML diagram.

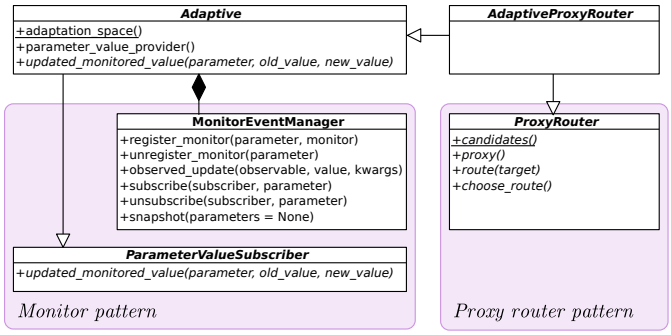

Figure 3. Adaptive component pattern UML diagram

Behavior: A component to be made adaptive can inherit the adaptive interface or a specific decorator can be created if the component's code should remain unchanged. The adaptive implementation defines what base adaptation space it will support. For example, in GUI implementations, this could be the availability of a toolkit or the type of input medium used (e.g., touch screen, mouse/keyboard, pen). Then, knobs can be defined within the component and used as variables to compute, for example, its size or lay outing specifications. Tuning can be done when an updated parameter value is received. For substitution, the process is the same, but uses the AdaptiveProxyRouter interface. Specific strategies can be created, using as many generic filters as possible (e.g., filter out candidates with adaptation space not overlapping with a snapshot of the current state).
Consequences: Because of the explicit declaration of adaptation space, strategies can be reasoned about how a component can behave in a situation. For example, a strategy can use the fact that a component's space is too specific or too wide. A significant advantage is that this can be previewed and tested by mocking the corresponding monitors (assuming that the designer's device has the adequate dependencies). Any component can be made adaptive and does not require modifications to other components. Even a parameter value provider can become gradually more complex. It could initially be based on a configuration file, which is essentially static during the application's execution, and be replaced by a more elaborate one when needed. Because of the support for both parametric adaptation and component substitution, the basic structure proposed in this pattern is suitable for virtually any adaptive mechanism based on monitored data and components adaptation spaces.
Constraints: Like stated in Section III-A, interacting adaptive components must subscribe to the same parameter value

provider to assure consistency in decision-making processes. While arbitrarily large hierarchies of adaptive components can be composed, there is an inherent overhead induced in the adaptation and routing process. Because a component subscribing to some parameter value provider such as the monitor event manager has no guarantee that this parameter is being actively monitored, adaptive components need to define a default behavior or immediately request a snapshot of the current state. If exceptions are used for non-monitored parameters (no value in the snapshot), their handling should be carefully done based on how monitors are registered (e.g., if monitors are concurrently registered as components are created). To minimize this effect, it is preferable to register monitors prior to creating any adaptive component.

Related patterns: Monitor (III-A), proxy router (III-B), adaptive component [21], virtual component [6].

IV. Patterns Realized

This section aims at providing a more practical foundation for the usage of the patterns presented in Section III. An overview of AdaptivePy, a library implementing the patterns, is first presented. Then, minimal use cases for the patterns are provided and implemented using AdaptivePy. These should demonstrate how the common problems identified are solved by the patterns and practically put in place. Finally, an introduction to the use of the patterns for GUI implementations with AdaptivePy and Qt is presented.

A. AdaptivePy

AdaptivePy implements artifacts from all three design patterns described in this paper. The library is freely available from the PyPi repository (`https://pypi.python.org`) under the name "adaptivepy" and is distributed under LiLiQ-P v1.1 license. The Python language was chosen because it is reflective, dynamically typed and many toolkit bindings are freely available. Beyond the patterns, AdaptivePy provides some useful implementations:

- Enumerated and discrete-value parameters
- Monitor event manager with a global instance as default provider for adaptive components
- Polling (pushed values) and pull dynamic monitor as decorators over static monitors
- Fixed (always provides the same value) and random static monitors
- Methods for operations on adaptation space (extension, union, filter)
- Strategy for choosing the most restricted component with narrowest adaptation space for a set of parameters
- Automatic computation of aggregated adaptation space for substitution candidates of a proxy router

While AdaptivePy is a fully working implementation of the patterns presented in this paper, it is possible to make different choices to realize the artifacts. For example, the MonitorEventManager artifact presented in the *Monitor* pattern could be realized as multiple managers, which coordinate the view on the environment and the propagation of the monitored values. Because the main objective in this paper is to demonstrate the technique for a single-host GUI, a centralized MonitorEventManager was deemed more appropriate.

One area in which special care must be taken when implementing the patterns is to minimize the work necessary to expand the amount of monitors and components. Since a complex system is expected to be composed from hundreds, if not thousands of components, the work necessary to add a new parameter, monitor or strategy must be kept minimal. This challenge is greatly mitigated by the use of a dynamic language like Python, where it is possible to compose classes at runtime.

It is necessary to mention that the aim of AdaptivePy is primarily demonstrative in the sense that it illustrates the applicability of the patterns presented in this paper. While crucial to the success of adaptation in any given application, the end-user's perception of increased usability due to adaptation is not the purpose of AdaptivePy. In fact, the effect of adaptation is expected to greatly vary from one application to the other, depending on what is adapted and when it is triggered. AdaptivePy, and consequently the patterns presented in this paper, aim at counteracting the complexity of implementing different adaptation strategies and structures in a gradual manner. Having each concern changeable and testable as separately as possible is then expected to provide the best end-user perceived usability with maximal predictability and minimal development effort through prototyping and A/B testing.

For each of the three patterns presented in this paper, a small example using AdaptivePy is given and explained.

B. Monitor Pattern

As presented in Section III-A, the *Monitor* pattern is concerned with acquisition and analysis of adaptation data. To express the environment in terms of contextual data, there is a need to model the environment into data of known range. This quantization is done on some raw data, which could be coming from an hardware sensor, network data provider or any process executing on the host machine (including the monitoring application itself).

An important aspect of the modeling of the environment is that raw data can be further refined into higher-level data through the cascading of monitors. For example, a hardware sensor could provide temperature data ranging from $-50°$ to $50°$C at $0.5°$ intervals. A higher-level modeling of this data could be to classify the values into an enumeration of three temperature levels: { Cold, Normal, Hot }. Table I shows a possible classification of the temperature levels as provided by the hardware sensor.

The artifact from the *Monitor* pattern that allows to provide a range of possible values is the parameter. The state space of the hardware sensor provided data is discrete, expressed as a range with step $[-50, 50[: 0.5°$C. The state space of the temperature level is an enumeration with three unique values. These parameters can be monitored by static monitors since they are stateless, assuming the hardware sensor can be queried

TABLE I. Suggested Classification of Temperature Levels

Level	Range
Cold	$[-50, 18[$
Normal	$[-18, 30[$
Hot	$[30, 50[$

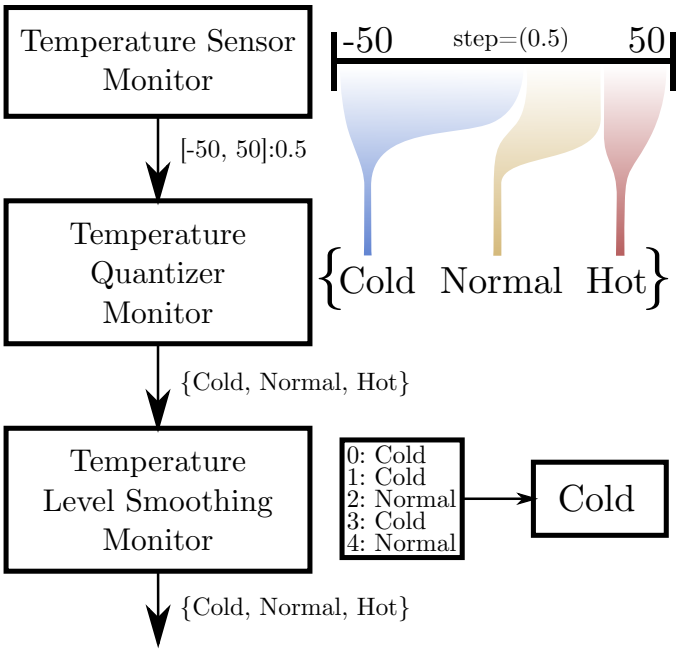

Figure 4. Realization of a temperature monitoring architecture

at any given time. By using this temperature sensor monitor, a second monitor could realize the classification algorithm based on the predefined thresholds presented in Table I. This monitor is considered *complex* because it relies on other monitors.

A refinement to this monitoring structure is to filter the monitored data. In fact, the hardware sensor's raw data might provide subsequent values oscillating between two quantized levels. A smoothing monitor could then be realize to address this issue. A simple moving average filter could be implemented and provide an average of the past M values. Another approach would be to provide the temperature level, which represents most of the past M samples. In all cases, because some state is necessary (previous values), the monitor is considered dynamic rather than static. An implication of being a dynamic monitor is that time affects its output value. At any time, a *latest value* can be queried and used in a snapshot of the system's current contextual state.

Fig. 4 provides a summarized view of the monitoring architecture as described previously. AdaptivePy allows to implement this architecture with minimal effort. Listing 1 shows how to declare the identified parameters using AdaptivePy. Similarly, Listing 2 shows how to declare the monitors using AdaptivePy and the threshold values from Table I.

We see from Listing 1 that parameters can be defined to represent adaptation data as state spaces in a trivial way. These parameters are then used in Listing 2 to define the possible values that can be provided by the monitors. We see that the monitors provide all the possible values of their corresponding parameter. This means that, at runtime, the application can encounter every state of the modeled environment. In this example, we see the cascading feature of the monitors to further refine contextual data into high level adaptation data. The *TempSensorMonitor* acquires raw sensor data, quantized into a discrete range. Then, the *TempQuantizerMonitor* turns this discrete state space into a higher level enumerated state space. Finally, the *TempLvlSmoothingMonitor* applies the `most_common` function to the last five values to smooth variations. A feature of this last monitor is that it is not stateless, which prevents it from being used by other static monitors. It is therefore turned into a DynamicMonitor using a default implementation that is pull-based. This default implementation is realized using the instance decorator *PullDynamicMonitorDecorator*. A pull-based dynamic monitor uses an external scheduling mechanism to perform updates during the lifetime of the application. It is omitted for simplicity in this example. However, it could trivially be implemented as a button the user has to push or as a polling mechanism using a timer which triggers an update at regular intervals.

```
RawTemperature = DiscreteParameter(-50.0, 50.0, 0.5)

class TemperatureLevel(EnumeratedParameter):
    low = 0
    medium = 1
    high = 2
```

Listing 1. Parameter declaration using AdaptivePy

```
class TempSensorMonitor(Monitor):
    def value(self):
        # Query the hardware sensor
    def possible_values(self):
        return RawTemperature.possible_values()

class TempQuantizerMonitor(Monitor):
    def value(self):
        value = TemperatureSensorMonitor().value()
        if -50 <= value < 18:
            return TemperatureLevel.Cold
        elif 18 <= value < 30:
            return TemperatureLevel.Normal
        else # 30 <= value < 50
            return TemperatureLevel.Hot
    def possible_values(self):
        return TemperatureLevel.possible_values()

class TempLvlSmoothingMonitor(Monitor):
    def __init__(self):
        self._last_five_values =
            [TemperatureLevel.Cold] * 5
        self._current_index = 0
    def value(self):
        value = TempQuantizerMonitor().value()
        self._last_five_values[self._current_index]
            = value
        self._current_index += 1
        self._set_latest_value(
            most_common(self._last_five_values))
    def possible_values(self):
        return TemperatureLevel.possible_values()

TempLvlSmoothingDynamicMonitor = \
    PullDynamicMonitorDecorator(
        TempLvlSmoothingMonitor())
```

Listing 2. Monitor definition using AdaptivePy

C. Proxy Router Pattern

A proxy router, as presented in Section III-B, is a component that acts as a proxy and can be controller to route to different delegates. The group corresponding to the possible delegates is called the *substitution candidates* of the proxy

router. An important element to the maintainability of an applications is the possibility to change parts related to a concern without affecting others. The *Proxy Router* pattern favorises decoupling of the code related to choosing the appropriate substitution delegate at an appropriate time and how to realize the substitution itself. Because strategies are highly dependent on specific application domains, they are expected to vary from one application to the other. However, the way components substitution can be implemented is mostly independent from the application domain.

A major challenge which transcends application domains is how to determine when adaptation should take place. In GUI, frequent changes in the layout of controls might destabilize users who have learned the position of certain controls. However, a GUI can improve responsiveness and better accommodate various types of users by adapting to their needs. For each scenario, specific strategies for these choices might be implemented and tested. One example of such strategy is to prevent applying adaptation when the user is using the application or a specific feature. By modeling a "busy" state for the user, it is possible to create a strategy that is aware of this state and provides the same substitution candidate as before until the user is not busy anymore. Similarly to monitors, one could cascade various routing strategies to create a more complex strategy.

A benefit of cascading strategies is that it allows to create generic strategies that rely on domain agnostic adaptation data such as the busy state discussed previously. Also, it is possible to control the timing of adaptation and computational load through filtering. Using the *Proxy Router* pattern, it is possible to realize multiple strategies and apply them to a common structure which is reusable across applications.

Assuming a parameter "Busy" with an enumerated state space of {yes, no} and a monitor "BusyMonitor" that provides all of the parameter's possible values, one can implement the cascading of strategies with AdaptivePy as shown in Listing 3. The proxy router *MyProxyRouter* is a trivial proxy router with two substitution candidates: *Candidate1* and *Candidate2*. It uses the *InternalProxyRouter* scheme (see Section III-B), which implements the router through inheritance. In AdaptivePy, the implementation of the proxy redirects calls to the `__getattr__` method to the delegate object's `__getattr__`, which makes the object behave as if it truly is the delegate. *MyProxyRouter* uses an externally defined strategy for routing that is represented as *MyStrategy* and instantiated in *MyProxyRouter*'s constructor.

An interesting feature of *BusyFilterStrategy* is that it is an adaptive strategy. Being adaptive, it has access to the "Busy" state which is being monitored by "BusyMonitor", in this case registered to the global monitor event manager. It acquires the "Busy" state through a local snapshot, that is a structure regrouping all the states the component is aware of. It does so in the `choose` method and then decides whether the value of the chosen candidate should be updated by querying the cascaded strategy or not.

By breaking down strategies into reusable sub-strategies that can be cascaded, the maintenance and extensibility of an application can be improved. Because it is possible to modify strategies individually and to cascade them as high level blocks in the proxy router, the lack of reusability in strategies is mitigated. Also, the challenge of determining when to adapt can be solved in steps rather than all at once. By gradually adding strategic elements to the proxy router as a common structure, components of an application that were not adaptive can acquire adaptive behavior as substitution candidates and strategies are developed rather than by refactoring the structure each time.

```
class MyProxyRouter(AdaptiveInternalProxyRouter):
    @classmethod
    def candidates(cls, arguments_provider=None):
        return { Candidate1: lambda: Candidate1(),
                 Candidat2: lambda: Candidat2() }
    def __init__(self):
        super().__init__()
        self._busy_filter = BusyFilterStrategy()
        self._specific_strategy = MyStrategy()
    def choose_route(self):
       return self._busy_filter.choose(lambda:
           self._specific_strategy.choose_route(
               self.candidates()))

@AdaptationSpace({Busy: Busy.possible_values()})
class BusyFilterStrategy(Adaptive):
    def __init__(self):
        super().__init__()
        self._candidate = None
    def choose(self, cascade_strategy):
        busy_state = self.local_snapshot().get(Busy)
        adapt = self._value is None or \
           busy_state == Busy.no
        if adapt:
            self._candidate = cascade_strategy()
        return self._candidate
```

Listing 3. Proxy router with filter strategy definition using AdaptivePy

D. Adaptive Component Pattern

Section IV-C contained an example of an adaptive component in the form of an adaptive strategy. In fact, the requirements to become adaptive are minimal: define an adaptation space and join and monitoring group by subscribing to a single parameter value provider. From that point on, a component is alerted when state changes within its adaptation space are detected. Also, it can request a local snapshot of the states corresponding to the parameters in its adaptation space. Using these values, it can apply parametric adaptation and, if it is a proxy router, component substitution.

Using the *Adaptive Component* pattern, it is possible to transform a previously non-adaptive component into an adaptive one. AdaptivePy utilizes the Python class decorators semantic to inject an adaptation space to any class. Then, by inheriting from the *Adaptive* class, a component can join a specific a parameter value provider by specifying it in the *Adaptive* constructor. This parameter value provider is realized using the MonitorEventManager from the *Monitor* pattern. It can then subscribe to any of the parameters in its adaptation space.

The declaration of a simple adaptive component is presented in Listing 4. Reusing the monitors from Section IV-B, the adaptive component *TempAdaptiveComponent* uses the temperature to adapt parametrically in the `updated_monitored_value` method. We see that, contrarily to the other examples, the adaptation space does

```
from adaptivepy.state_space.enumerated_state_space
    import EnumeratedStateSpace as Ess

@AdaptationSpace({TemperatureLevel:
    Ess({TemperatureLevel.Low,
         TemperatureLevel.Normal})})
class TempAdaptiveComponent(Adaptive):
    def __init__(self,
        parameter_value_provider=None):
        super().__init__(parameter_value_provider)
        self._subscribe_to_all_parameters()
    def updated_monitored_value(self, parameter,
        old_value, new_value):
        # Implement parametric adaptation
        if new_value is TemperatureLevel.Low:
            ... # Do something
        else: # new_value is Temperature.Normal
            ... # Do something else
```

Listing 4. Adaptive component definition using AdaptivePy

not fully cover the parameter's possible values. Because it is not supported, the implications of reaching the `TemperatureLevel.Hot` state are undefined for this component. If an application uses this component and can reach this state, component substitution should be implemented to swap this component with another one which supports the missing states. An advantage of this design is that a developer can focus on an explicitly defined region of an application's adaptation space and ignore other states in their implementation. This is seen in the implementation of the `updated_monitored_value`, where only the states defined in the adaptation space are handled. In this way, if the `new_value` is not `TemperatureLevel.Low`, it can only be `TemperatureLevel.Normal`. This is the case because the component has no knowledge of `TemperatureLevel.Hot`.

By specializing adaptive components, the service they offer is expected to be better suited at the region of adaptation space they define. By adding specialized component and adaptive behavior to non-adaptive components, an application can be ported to an adaptive form gradually. Also, because the patterns presented in this paper serve specific concerns, the adaptive components are not expected to be affected by changes in the monitoring or their structural arrangement when used for an application. The latter is possible because of the basic structure provided by the proxy routers and by the self-contained nature of components-based software.

E. Patterns applied to GUI

The patterns presented in this paper can be applied to GUI to create adaptive components as custom widgets and layouts. The general-use toolkit Qt was chosen for the case study, therefore this section will focus on Qt implementations. Qt provides a graphical editor, Qt Designer, for designing the GUI in a language independent descriptive language. Since this is the *de facto* approach, it is also the favored workflow. Note that this is true for many other toolkits (e.g., Gtk with Glade, JavaFX with SceneBuilder).

An *ad hoc* solution would be to add a placeholder widget in the GUI and replace them at runtime with the adequate component. Setting the appropriate control needs to be done entirely programmatically, along with any customization necessary, in the window's class that owns the control. This leads to a lack of extensibility, a tangling of concerns between the adaptation concern and the components' own concern. Moreover, the approach is not compatible with normal GUI design workflow, which involves previewing the application in the graphical editor before adding logic.

By controlling monitors from the *Monitor* pattern, one can visualize any adaptation done by components for the given toolkit. If a different toolkit is to be used (e.g., when porting an application), the necessary work is to create a candidate component for a proxy router using the new toolkit and adding a toolkit parameter value as an adaptive space definition. A conversion from one description language to the other would also be needed. As for the structure provided by the *Proxy Router*, only the binding to the toolkit's widget replacing logic is to be ported. As for the *Adaptive component* pattern, the components simply need to support the adequate portion of the adaptation space, which includes a toolkit parameter if any adaptation logic is dependent on different toolkit.

In this paper, because Python is used rather than C++ (Qt's native language), an external plugin for Qt Designer is necessary to load custom widgets. This is provided by PyQt as "libpyqt5" for GNU/Linux. Custom widgets are created using the `QPyDesignerCustomWidgetPlugin` base class. Fields can also be added using `pyqtProperty` and use the underlying adaptive component's interface to customize the component. This is especially useful with proxy router components since any customization is automatically applied to any candidate and state transfer can be more easily handled. It is also possible to use properties to control adaptive behavior by means of exposed knobs.

V. Prototype

Adaptivity can help in improving usability in different ways. One usability principle of graphical user interfaces is to take into account the user's cognitive limitations into consideration for the presentation of controls. For example, the number of elements in a group one can remember from short-term memory is used to limit the number of grouped controls displayed to the user. This number is not confidently known, but some suggested that chunks of 4 ± 1 elements can be accurately remembered using short-term memory, while it was originally estimated to be averaging around 7 ± 2 [29]. We draw inspiration from this usability principle in our case-study prototype application.

The case study application is a special poll designed to favor polarization. Five yes/no questions are asked to a user and answered by selecting the most appropriate response among a list of options. The possible options provided include yes, no, mostly yes, mostly no and 50/50. To favor polarization, statistics from the previous answers are used to restrict the range of options provided to the user. If the polarization is judged insufficient because of mixed responses (low polarization), fewer options are provided. On the contrary, if virtually all users have answered yes (high polarization), more options in between will be given. The workflow of the application is to start the "quiz" using a Start button, choose appropriate options and send the form using a Submit button. If some options remain unselected, a prompt alerting the user is shown and the form can be submitted again once all options are selected.

The adaptation used is a form of *alternative elements* [30]. This provides a form of "plastic" GUI in that it adapts itself, but retains its usability [31]. The GUI is made plastic by replacing control widgets displaying the available options at runtime, conserving the option selection feature in any resulting interface. To minimize the visual overload, some widgets are more appropriate than others to display them, while some cannot display certain amounts of options. A checkbox can handle two options with a single control. Radio buttons could handle many options, but to follow the usability guideline of congnitive limitation, we could use it to display up to four options at a time. Finally, a combo box can handle many options, but it does not display all the options on the window unless it is clicked. For our usage, it is a better choice for five and more options. Of course, radio buttons can hold more options and the combo box less, but the amounts suggested represent the ranges they better suit the usability principle. Because many other variables need to be taken into account and affect the usability, the ranges can be chosen by a designer and further refined through user testing, which means they must be easy to edit.

Polarization levels act as adaptation data to drive adaptation. An appropriate solution would allow to design the GUI within Qt Designer and to preview of the adaptation directly, rather than having to add the business logic beforehand. It would also allow for gradual addition and modification of control widget types without necessitating changes in unaffected modules.

The toolkit used for this application is Qt 5 through the PyQt5 wrapper library. It is a cross-platform toolkit library, which provides implementations of widgets like checkboxes, combo boxes are radio buttons groups. The concrete work is therefore limited to implementing how these components can replace each other at the appropriate time and how they are included in a main user interface. We are therefore more interested in the underlying structure of adaptation within the application than specific adaptation strategies and their user-perceived effectiveness. Once an appropriate structure is in place, we expect these can be more easily devised, tested and improved.

VI. Comparing Ad hoc and AdaptivePy

The windows shown on Fig. 5 are the resulted GUI for the application in all three polarization states. Because this case study's focus is on GUI, the monitoring of past responses was simulated and a random monitor is used instead. This monitor updates its value by means of a polling dynamic monitor every second, allowing to easily observe adaptation.

To emphasize the differences between the *ad hoc* solution and the one using patterns, adapters for each three control widgets (checkbox, radiobox and combobox) were created and are used in both applications. They all implement a common interface `OptionsSelector`, which defines common operations on the controls such as `set_text` for the question labelling and `set_options` that takes pairs of text and corresponding value for averaging past answers. The goal is to compare the implementation of adaptation rather than adding new type of adaptation, thus the same control abstraction approach was used for controls in both cases.

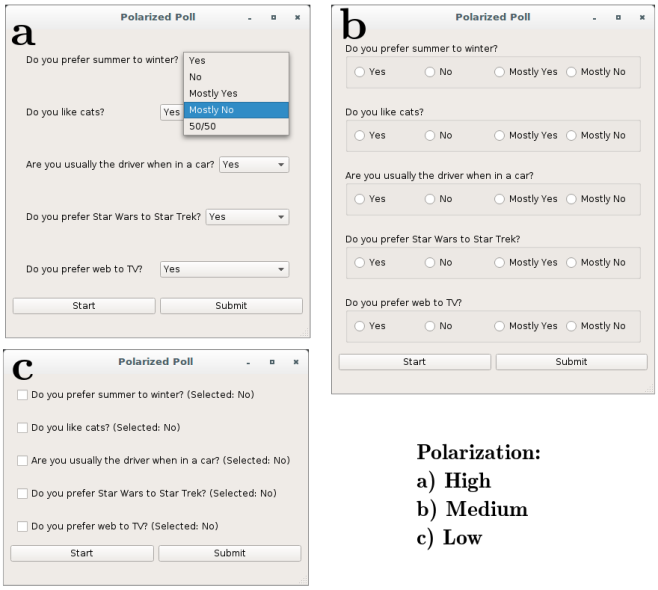

Figure 5. Adaptive case study application "Polarized Poll"

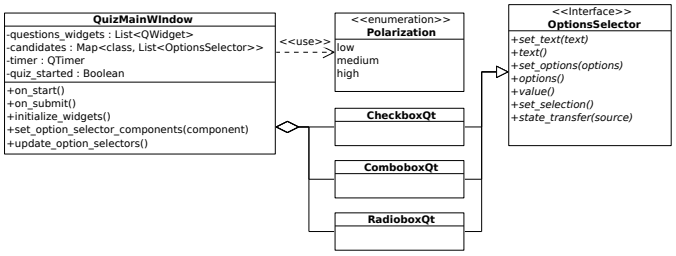

Figure 6. Simplified UML diagram of *ad hoc* implementation of case study application

A. Ad hoc *Application*

A simplified UML diagram of the *ad hoc* implementation is shown on Fig. 6. The chosen approach is to add placeholder widgets in QuizMainWindow which will be substituted by an appropriate component instance at runtime: CheckboxQt, ComboboxQt or RadioboxQt. A polarization level defined in the enum Polarization is bound to each of these types. A timer within QuizMainWindow polls the polarization value and calls `set_options_selector_components` with the appropriate type. Adaptation control, along with any customization necessary, is entirely done in QuizMainWindow.

Fig. 7 shows Qt Designer as the main window is created for the *ad hoc* implementation. Notice that because placeholder components are blank, no feedback is given to the designer. It is therefore not possible to test the controls or set the question label. This makes the approach incompatible with the usual GUI design workflow, which involves previewing the application in the graphical editor before adding business logic.

When analyzing the *ad hoc* code, it is obvious that separation of concerns is not respected since the option selection logic is tangled to its owner element, the main window. Concerns such as scheduling for recomputing polarization and component substitution are mixed with GUI setup and handling

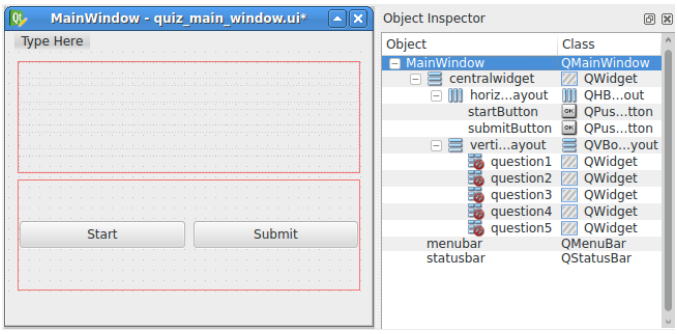

Figure 7. Qt Designer using plain widgets as placeholder for *ad hoc* implementation

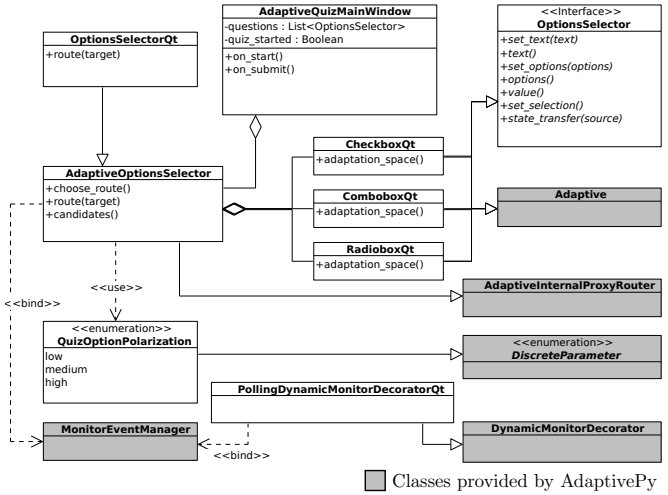

Figure 8. Simplified UML diagram of case study application implementation using AdaptivePy

of the business flow. This leads to a lack of extensibility, a tangling of concerns and limits unit testing of components. A method is used to select which control component to use based on the polarization, but this solution remains inflexible. The knowledge of adaptation is hidden and cannot be used to devise portable strategies.

One of our goals is to gradually add adaptation mechanisms to GUI implementations, but this is difficult since modification of important classes will add risk of introducing defects. Also, there is no easy way to work on adaptation mechanisms separately from the application. In fact, we cannot separately test the adaptation logic and integrate it after. Another limitation, in this case specific to GUI, is that all settings specific to the widgets (e.g., question labels) cannot be set from the graphical editor. This is a strong deviation from the usual GUI workflow. Generally, the lack of cohesion induced by the inadequate separation of concerns is a sign of low code quality. Because no adaptation mechanism can easily be introduced, modified and reused in other projects, the *ad hoc* implementation works for its specific application case, but is subject to major efforts in refactoring when requirements and features will be added throughout its development cycle.

B. Application Using AdaptivePy

A simplified UML diagram of the application is shown on Fig. 8. From it, we see that the polarization is a discrete parameter and is used by AdaptiveOptionsSelector, specifically to define its adaptation space based on the ones provided by its substitution candidates: CheckboxQt, ComboboxQt and RadioboxQt. Additionally to adaptation by substitution, RadioboxQt can parametrically adapt to changes of polarization levels {low, medium}, since they respectively correspond to 2 and 4 options. Its behavior is that the appropriate number of options is shown depending on the polarization level. AdaptiveQuizMainWindow is free of adaptation implementation details and simply uses the AdaptiveOptionsSelector instances as a normal OptionsSelector. OptionsSelectorQt is a subclass to AdaptiveOptionsSelector, which is used as a graphical proxy to candidate widgets. It also defines properties used in Qt's graphical editor Qt Designer, in this case the question label.

Every AdaptiveOptionsSelector instance is made a subscriber to the QuizOptionPolarization parameter at initialization. They are updated when a change in the monitored value is detected, i.e., when a monitor detects a value is different from the previous one. This is because identical subsequent parameter values are expected by default to lead to the same state, so they are filtered out. In the case of AdaptiveOptionsSelector, because it is a proxy router, `choose_route` is called to determine which substitution candidate to route to. Prior to using an adaptation strategy to select the most appropriate candidate, inappropriate ones can be filtered out using `filter_by_adaptation_space`. This function, provided by AdaptivePy, takes a list of candidates along with a snapshot of the current monitoring state and only returns those with adaptation space supporting the current context. Then, a strategy like `choose_most_restricted` is used to choose among valid components. If no component is valid, an exception is raised. With a candidate chosen, all that remains is configuring the proxy router by calling the `route` method with the chosen candidate. This method must also take care of state transfer between the previous and new proxied components. This feature is already defined in the common interface OptionsSelector as `state_transfer`. The `route` method takes care of the state transfer and updates the proxy (done by the library). Subscription to the polarization parameter is done at initialization.

Fig. 9 shows Qt Designer as the main window is created with the AdaptivePy-based implementation. When compared to Fig. 7, we notice that the designer has a full view of how the application will look. Moreover, the currently displayed adaptation can be controlled through the setup of the monitors. For example, it is possible to replace the random value by one acquired from a configuration file and trigger adaptation manually. Also, each question is simply a OptionsSelectorQt component rather than a placeholder component and the question is entered directly from the graphical editor using the label property (bottom-right). A major advantage is that adaptive components can be reused in other interfaces because they are provided as standalone components. The need for easy edition of adaptation spaces is also addressed by modifying or overriding the `adaptation_space` method of adaptive components.

The main difference compared to the *ad hoc* implementation is that no adaptation concern can be found in the owner

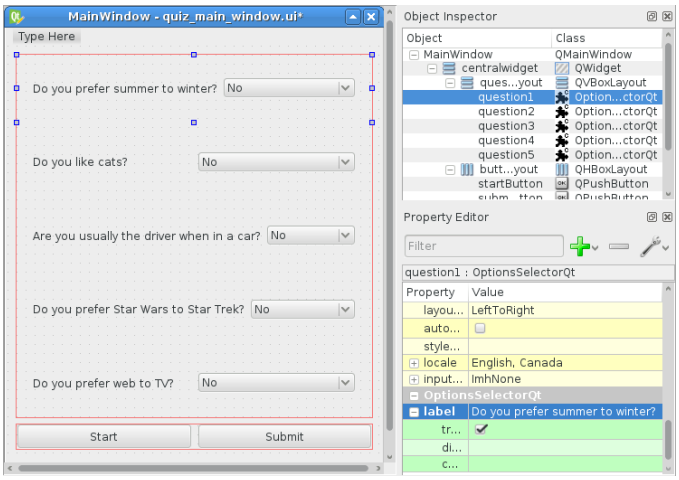

Figure 9. Qt Designer using adaptive components developed with AdaptivePy

class, the main window. An enumerated discrete parameter with three polarization values (low, medium and high) is monitored by a polling dynamic monitor decorator over a random static monitor. To create the adaptive component, an adaptive proxy router was used as a base with candidates being the three same control classes as the *ad hoc* implementation, but in adaptive form (each having an adaptation space related to different polarization levels).

To create the custom widget, two classes are necessary: a dedicated class, which inherits QWidget and its related factory with metadata used by Qt Designer such as its name, group and description. A template class was created to ease this step. The only logic specific to the options selector is related to additional properties. In this case, a string property for the question label along with binding to the options selector's interface method (getter and setter) is used. The option values are also set by this class to centralize the customization logic (this could also have been done using a property). Finally, the logic regarding the lay outing was implemented as a layout with a single element: the proxy delegate. When routing, the QWidget adapter removes the proxy delegate from the layout and adds the new one provided by the base option selector class (described previously). No additional modification other than removing the logic related to the old options selector implementation was necessary.

The adaptation logic is essentially located in the adaptive proxy router class: AdaptiveOptionsSelector. Because adaptation is separated from the rest of the business logic, the main window class can use the adaptive components without the knowledge of adaptation. The only logic remaining is with regard to buttons handling (Start and Submit buttons). It is clear in this implementation that the knowledge of adaptation space, which was hidden in the *ad hoc* implementation, is used to efficiently choose a substitution candidate. More so, the radiobox is suitable for two to four options and therefore covers low and medium polarization through parametric adaptation. It could then be used instead of the checkbox if a strategy for choosing the less restricted candidate had been used or if a malfunction is detected in the checkbox implementation rendering it inadequate as a candidate. This parametric adaptation behavior cannot easily be included in the *ad hoc* implementation since the knowledge of polarization is kept at the owner component level. The component would need to provide a mean through its interface to customize a component based on polarization, but this would affect all other components as well.

Self-healing action such as replacing a failing component can be realized by monitoring the components and including this logic as a strategy. This is not easily realizable in the *ad hoc* implementation. In the prototype, a radio box could safely replace a checkbox since it parametrically covers its full adaptation space, overlapping on {low} polarization. Also, from this case study, we can see that arbitrarily large hierarchies of adaptive and non-adaptive components can be built without tangling code or affecting other components when adding new adaptive behavior.

VII. Conclusion and Future Work

Design patterns presented in this paper can be used as a basic structure to accomplish various levels of adaptation in GUI. Adaptive components can be used with other modules such as recommendation engines to provide more or less automation and proactive adaptation. Monitors can also be extended and even implemented as adaptive components themselves, relying on other more primitive monitors. Proxy routers allow to simplify hierarchical development of arbitrarily large sequences of component substitutions. The patterns form together an effective approach for the integration of various adaptation mechanisms and, in the case of GUI, can be used to provide a more usual workflow than the *ad hoc* implementation.

AdaptivePy, as a reference library, is an example of the viability of the patterns when used in a concrete implementation. Although simple examples and a prototype application were used to observe gains, the solution is applicable to more complex scenarios where multiple parameters, monitoring groups and large hierarchies of adaptive components. The patterns are general enough that they can be used for adding adaptive behavior based on user, environment and platform variations.

Although an analysis of the prototype has been done using concepts of separation of concerns and quality principles in Section VI, there is a lack of quantitative metrics directly aimed at adaptive software. Example of metrics that would be interesting to automatically acquire are the quality in term of adaptation space coverage, adaptation complexity for a set of components sharing a common context and a measure of overhead in adaptation realization in a large hierarchy. Acquiring these metrics would allow to easily compare strategies used for component adaptation and provide guidelines to developers on which strategy is most appropriate in certain circumstances.

Future work will focus on exploring adaptation quality metrics such that verification and validation methods can be used as an objective evaluation of gains. New metrics using concepts of the design patterns presented in this paper will therefore be explored. The goal is to better quantify the quality level of prototypes with regards to adaptation.

References

[1] S. Longchamps and R. Gonzalez-Rubio, "Design patterns for addition of adaptive behavior in graphical user interfaces," in *Proceedings of the Ninth International Conference on Adaptive and Self-Adaptive Systems and Applications*, 2017, pp. 8–15.

[2] F. Chang and V. Karamcheti, "A framework for automatic adaptation of tunable distributed applications," *Cluster Computing*, vol. 4, no. 1, pp. 49–62, 2001, ISSN: 1573-7543.

[3] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Software: Practice and Experience*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[4] Y. Maurel, A. Diaconescu, and P. Lalanda, "Ceylon: A service-oriented framework for building autonomic managers," in *2010 Seventh IEEE International Conference and Workshops on Engineering of Autonomic and Autonomous Systems*, Mar. 2010, pp. 3–11.

[5] M. Peissner, A. Schuller, and D. Spath, "A design patterns approach to adaptive user interfaces for users with special needs," in *Proceedings of the 14th International Conference on Human-computer Interaction: Design and Development Approaches - Volume Part I*, ser. HCII'11, Orlando, FL: Springer-Verlag, 2011, pp. 268–277.

[6] A. Corsaro, D. C. Schmidt, R. Klefstad, and C. O'Ryan, "Virtual component - a design pattern for memory-constrained embedded applications," in *In Proceedings of the Ninth Conference on Pattern Language of Programs (PLoP*, 2002.

[7] G. Rossi, S. Gordillo, and F. Lyardet, "Design patterns for context-aware adaptation," in *2005 Symposium on Applications and the Internet Workshops (SAINT 2005 Workshops)*, Jan. 2005, pp. 170–173.

[8] A. J. Ramirez, "Design patterns for developing dynamically adaptive systems," Master's thesis, Michigan State University, 2008.

[9] T. Holvoet, D. Weyns, and P. Valckenaers, "Patterns of delegate mas," in *2009 Third IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, Sep. 2009, pp. 1–9.

[10] K. Majrashi, M. Hamilton, and A. Uitdenbogerd, "Multiple user interfaces and cross-platform user experience: Theoretical foundations," in *CCSEA 2015*, AIRCC Publishing Corporation, 2015, pp. 43–57.

[11] M. G. Hinchey and R. Sterritt, "Self-managing software," *Computer*, vol. 39, no. 2, pp. 107–109, 2006.

[12] M. Peissner, D. Häbe, D. Janssen, and T. Sellner, "Myui: Generating accessible user interfaces from multimodal design patterns," in *Proceedings of the 4th ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '12, Copenhagen, Denmark: ACM, 2012, pp. 81–90.

[13] M. Salehie and L. Tahvildari, "Self-adaptive software: Landscape and research challenges," *ACM Transactions on Autonomous and Adaptive Systems (TAAS)*, vol. 4, no. 2, p. 14, 2009.

[14] M. L. Berkane, L. Seinturier, and M. Boufaida, "Using variability modelling and design patterns for self-adaptive system engineering: Application to smart-home," *Int. J. Web Eng. Technol.*, vol. 10, no. 1, pp. 65–93, May 2015, ISSN: 1476-1289.

[15] IBM, "An architectural blueprint for autonomic computing," IBM Corporation, Tech. Rep., 2005.

[16] S. Malek, N. Beckman, M. Mikic-Rakic, and N. Medvidovic, "A framework for ensuring and improving dependability in highly distributed systems," in *Architecting Dependable Systems III*, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 173–193.

[17] V. Mannava and T. Ramesh, "Multimodal pattern-oriented software architecture for self-optimization and self-configuration in autonomic computing system using multi objective evolutionary algorithms," in *Proceedings of the International Conference on Advances in Computing, Communications and Informatics*, ser. ICACCI '12, Chennai, India: ACM, 2012, pp. 1236–1243.

[18] A. J. Ramirez and B. H. Cheng, "Design patterns for developing dynamically adaptive systems," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ACM, 2010, pp. 49–58.

[19] M. Parashar and S. Hariri, "Autonomic computing: An overview," in *Proceedings of the 2004 International Conference on Unconventional Programming Paradigms*, ser. UPP'04, Le Mont Saint Michel, France: Springer-Verlag, 2005, pp. 257–269.

[20] D. Weyns, S. Malek, and J. Andersson, "On decentralized self-adaptation: Lessons from the trenches and challenges for the future," in *Proceedings of the 2010 ICSE Workshop on Software Engineering for Adaptive and Self-Managing Systems*, ser. SEAMS '10, Cape Town, South Africa: ACM, 2010, pp. 84–93.

[21] W.-K. Chen, M. A. Hiltunen, and R. D. Schlichting, "Constructing adaptive software in distributed systems," in *Distributed Computing Systems, 2001. 21st International Conference on.*, Apr. 2001, pp. 635–643.

[22] D. A. Menasce, J. P. Sousa, S. Malek, and H. Gomaa, "Qos architectural patterns for self-architecting software systems," in *Proceedings of the 7th International Conference on Autonomic Computing*, ser. ICAC '10, Washington, DC, USA: ACM, 2010, pp. 195–204.

[23] H. Liu and M. Parashar, "Accord: A programming framework for autonomic applications," *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, vol. 36, no. 3, pp. 341–352, May 2006, ISSN: 1094-6977.

[24] J. Zhang and B. H. C. Cheng, "Model-based development of dynamically adaptive software," in *Proceedings of the 28th International Conference on Software Engineering*, ser. ICSE '06, Shanghai, China: ACM, 2006, pp. 371–380.

[25] H. Gomaa, K. Hashimoto, M. Kim, S. Malek, and D. A. Menascé, "Software adaptation patterns for service-oriented architectures," in *Proceedings of the 2010 ACM Symposium on Applied Computing*, ser. SAC '10, Sierre, Switzerland: ACM, 2010, pp. 462–469.

[26] P. Kang, M. Heffner, J. Mukherjee, N. Ramakrishnan, S. Varadarajan, C. Ribbens, and D. K. Tafti, "The adaptive code kitchen: Flexible tools for dynamic application composition," in *2007 IEEE International Parallel and Distributed Processing Symposium*, Mar. 2007, pp. 1–8.

[27] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1995.

[28] H. Gomaa and M. Hussein, "Software reconfiguration patterns for dynamic evolution of software architectures," in *Software Architecture, 2004. WICSA 2004.*

Proceedings. Fourth Working IEEE/IFIP Conference on, Jun. 2004, pp. 79–88.

[29] N. Cowan, "The magical number 4 in short-term memory: A reconsideration of mental storage capacity," *Behavioral and Brain Sciences*, vol. 24, no. 1, pp. 87–114, 2001.

[30] M. Bezold and W. Minker, *Adaptive multimodal interactive systems*. Springer Science & Business Media, 2011.

[31] J. Coutaz, "User interface plasticity: Model driven engineering to the limit!" In *Proceedings of the 2Nd ACM SIGCHI Symposium on Engineering Interactive Computing Systems*, ser. EICS '10, Berlin, Germany: ACM, 2010, pp. 1–8.