Design Patterns for Addition of Adaptive Behavior in Graphical User Interfaces

Samuel Longchamps, Ruben Gonzalez-Rubio

Sherbrooke University,
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 and used in a GUI application case study. This case study shows concrete usage of the patterns and is 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.**

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

I. Introduction

As applications become increasingly complex and distributed, adaptive software has become a research subject of great interest to tackle related challenges. 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.

Many researchers have proposed models and frameworks to implement adaptive behavior in a generic manner for components-based software [1]–[4]. These solutions typically require significant effort to modify an existing software architecture and make many 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 which 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 [5]–[8], 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 which would include the design patterns' artifacts, the Qt cross-platform toolkit along with the Qt Designer graphical editor was 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.

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. The prototype application with adaptive GUI is presented in Section IV and an analysis of the gains procured by the use of the proposed design patterns are presented in Section V. The paper concludes with Section VI 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

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 [9]) and external (for "self-situated" [9] or "context-aware" applications). 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 [10]. 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 [7], [11]–[15]. Multiple simple sensors can be composed to form a complex sensor, which provide higher-level contextual data (Sensor Factory pattern [15]). 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

 ISBN: 978-1-61208-532-6

[15]). When a variety of adaptation data is monitored, it provides a modeled view of the software context, sometimes 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 [15]). Quantization can be done on multiple abstraction levels and thresholds can be used to trigger adaptation events (Adaptation Detector [15]).

B. Adaptation Schemes in Components

Many researchers aimed at defining a design pattern for an adaptive component that would allow for various schemes of adaptation in a generic way. Two main approaches can be extracted from previous work: 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.* [5]. It is similar to the adaptive component proposed by Chen, Hiltunen, and Schlichting [16], 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.* [17], 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 [18], [19] and buffering requests [20]. State transfer between components can be used when possible, otherwise the computing job must be restarted [16], [19].

b) ***Parametric adaptation****:* Rather than substituting a whole component by a more appropriate one, parametric adaptation is when 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* [1] 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 [10]). 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 [21].

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 [1], [13]. 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 [7] 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 which 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.

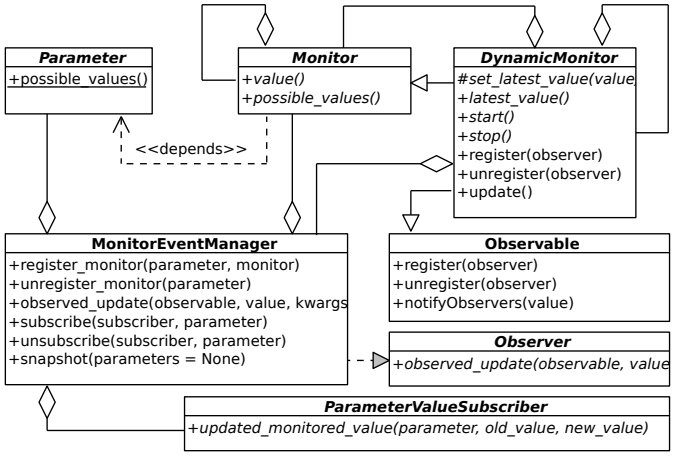

Figure 1. Monitor pattern UML diagram

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 which 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 [22]. 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 [12] to be done through hierarchical construction of monitors: a parameter can define high-level domain values which 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 [7], information sharing, observer [22].

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 (substitution by which component) and the execution of substitution (replacing a component or forwarding calls to it) are difficult to implement in an extensible way. The proxy pattern [22] 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 which 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.

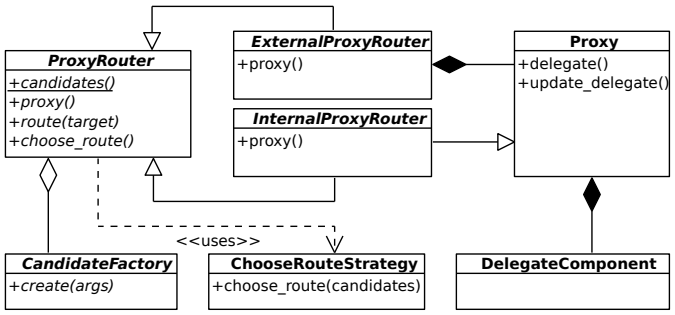

Figure 2. Proxy router pattern UML diagram

Participants:

- **Proxy:** Gang of Four [22] 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 which 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 [22] factory pattern for a candidate. Used as part of candidates definition.

 ISBN: 978-1-61208-532-6

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 [22] 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 which is not portable. Separating specific from generally applicable strategies and composing them should help with this constraint.
Related patterns: Adaptive component [16], virtual component [5], master-slave [23], component insertion/removal, server reconfiguration [7], proxy [22].

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 any impact on the rest of the system.
Participants:

- **Adaptive:** An adaptive component which defines means for acquiring the adaptation space. It can be used as a subscriber to a parameter value provider.
- **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.

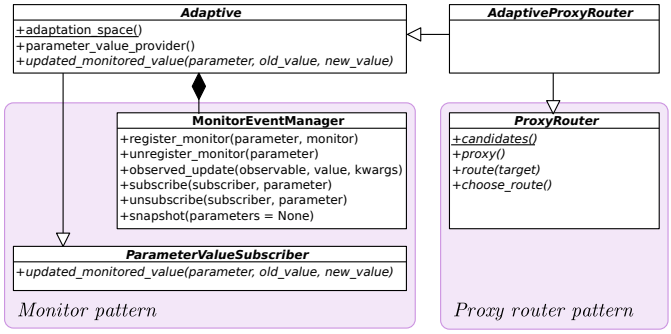

Figure 3. Adaptive component pattern UML diagram

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. 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 which adaptation space does not overlap with a snapshot of the current state).
Consequences: Because of the explicit declaration of adaptation space, strategies can easily reason 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. Any component can be made adaptive and does not require modifications to other components. 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 monitor event manager 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. 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 [16], virtual component [5].

IV. PROTOTYPE

This section presents AdaptivePy, a reference library implementing the three design patterns presented in this paper,

along with a prototype as a case study for analyzing the gains they procure compared to an *ad hoc* implementation.

A. AdaptivePy

AdaptivePy implements artifacts from all three design patterns described in this paper. 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, such as enum-based discrete-value parameters, push/pull dynamic monitor decorators, operations over adaptation spaces (extend, union, filter) and an adaptation strategy based on substitution candidates' adaptation space restrictiveness. The library is freely available from the PyPi repository under the name "adaptivepy" and is distributed under LiLiQ-P v1.1 license.

B. Case Study 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 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 type used is a form of *alternative elements* [24]. The GUI is made plastic by replacing control widgets displaying the available options at runtime, conserving the option selection feature in any resulting interface. Because there is a varied number of options, some widgets are more appropriate than others to display them, while some cannot display certain amounts of options. A checkbox can handle two options, radio buttons could be used for ranges of two to four options and a combo box 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 are subjectively considered most appropriate for. These 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's graphical editor "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.

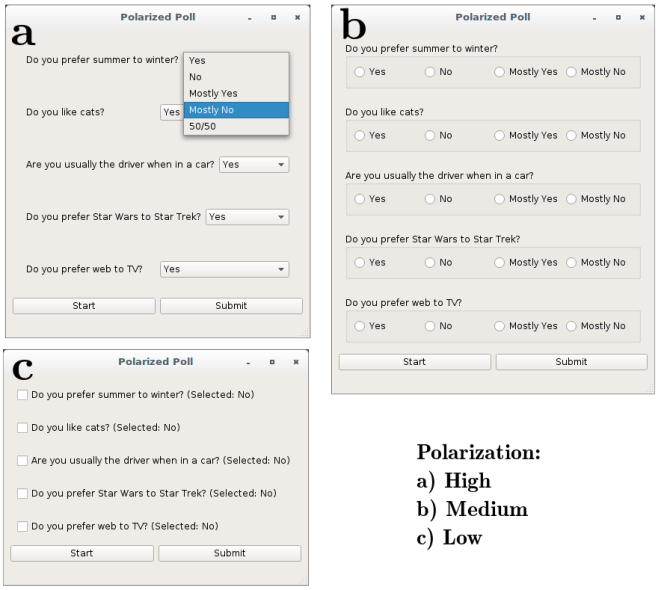

Figure 4. Adaptive case study application "Polarized Poll"

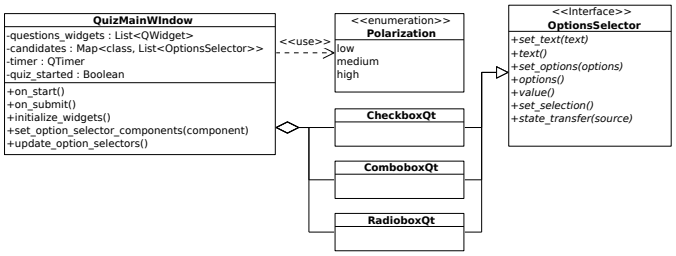

Figure 5. Simplified UML diagram of *ad hoc* implementation of case study application

V. Results

The windows shown on Fig. 4 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 which updates its value by means of a polling dynamic monitor every second, allowing to easily observe adaptation.

A. Ad hoc *Application*

A simplified UML diagram of the *ad hoc* implementation is shown on Fig. 5. 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.

 ISBN: 978-1-61208-532-6

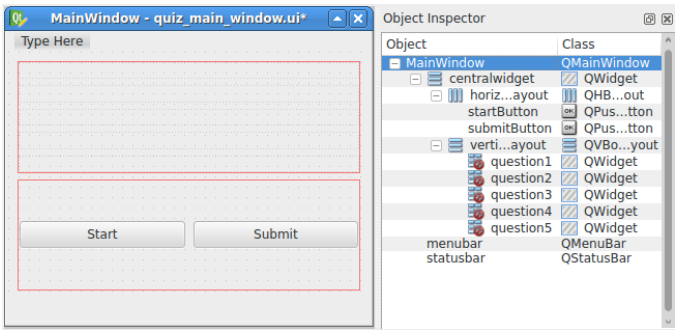

Figure 6. Qt Designer using plain widgets as placeholder for *ad hoc* implementation

Fig. 6 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 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. 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. 7. 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

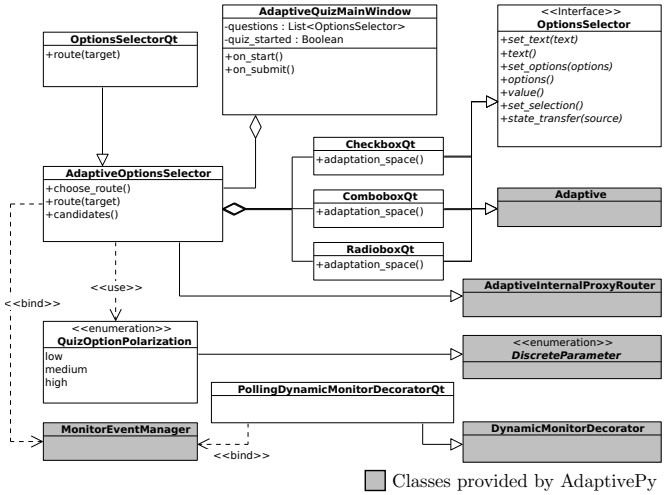

Figure 7. Simplified UML diagram of case study application implementation using AdaptivePy

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`.

Fig. 8 shows Qt Designer as the main window is created with the AdaptivePy-based implementation. When compared to Fig. 6, 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

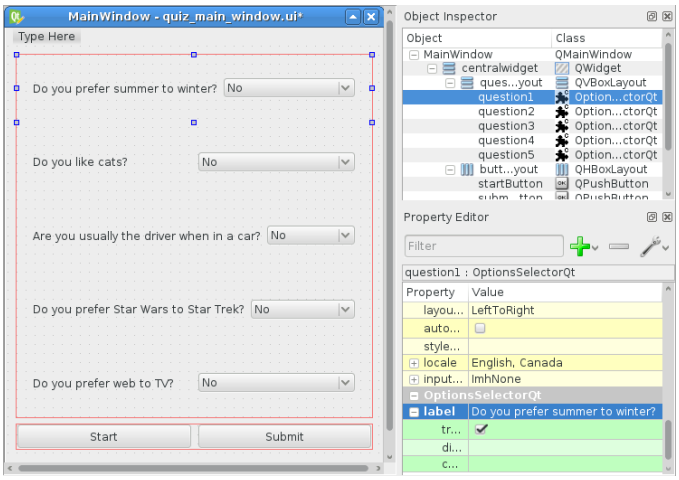

Figure 8. Qt Designer using adaptive components developed with AdaptivePy

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 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. Self-healing action such as replacing a failing component can be easily 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.

VI. 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. Even though a simple application was 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.

Future work will focus on exploring parameters types with more complex value domains and try to formalize a structure to express them. Also, the lack of adaptation quality metrics for verification and validation methods limits the evaluation of gains. To alleviate this limitation, new metrics using concepts of the design patterns presented in this paper will be explored. The goal is to better quantify the quality level of prototypes with regard to adaptation.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] 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.

 ISBN: 978-1-61208-532-6

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

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

[21] 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.

[22] 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.

[23] 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.

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

 ISBN: 978-1-61208-532-6