
TDLOG séance 5 - Architecture logicielle

Xavier Clerc – xavier.clerc@enpc.fr

3 novembre 2025

Table des matières

1	Introduction	2
2	Généralités	2
2.1	Principes généraux	2
2.2	Éléments d'appréciation	3
3	Architectures en couches	4
3.1	Définition	4
3.2	Exemple	5
4	Pipelines	5
4.1	Définition	5
4.2	Exemples	5
5	MVC	6
5.1	Définition	6
5.2	Exemple	6
6	Architectures client-serveur	7
6.1	Définition	7
6.2	Exemple	8
7	Architectures <i>cloud computing</i>	8
7.1	Présentation générale	8
7.2	Différentes formes	8
7.3	Questions soulevées	9
8	Composition de motifs architecturaux	9
9	Interfaces graphiques : généralités	11
10	Interfaces graphiques : librairie Qt en Python	12
10.1	Présentation	12

10.2 Notions de signal et de slot	13
10.3 QtDesigner	13
10.4 Qt et Python	16

1 Introduction

Lors des séances précédentes, nous avons étudié les bases de la programmation orientée objet Python. Au cours de cette séance, nous allons étudier la manière dont un logiciel peut être construit sur ces briques de base.

Après une brève introduction, nous présenterons (sans chercher l'exhaustivité) un certain nombre d'architectures logicielles *classiques* (sections 3 à 6), et les plus récentes architectures *cloud* (section 7). Ces architectures forment d'une certaine manière le vocabulaire de base utilisé entre développeurs pour discuter d'une nouvelle architecture, qui est souvent la combinaison de plusieurs motifs de base (section 8). Ensuite, nous présenterons les éléments de base de la programmation d'interfaces graphiques (section 9) et les illustrerons à l'aide de la librairie Qt (section 10).

2 Généralités

Définir ce qu'est l'architecture en informatique est une gageure. Si le terme est relativement clair en ce qui concerne les aspects matériels, il reste relativement flou en ce qui concerne les aspects logiciels. Dans le cas du matériel, l'architecture désigne les composants de base (mémoire, microprocesseur, *etc*) ainsi que leur agencement -- y compris par des descriptions *géométriques* dans une forme d'analogie avec l'architecture au sens des arts. Dans le cas du logiciel, on est naturellement tenté de procéder de même par analogie.

Cependant, on bute très rapidement sur de nombreux problèmes. Premièrement, on ne possède pas de définition consensuelle (et encore moins indépendante du langage de programmation) de ce qu'est un composant. Deuxièmement, le logiciel est une création immatérielle que l'on ne *voit* pas. Troisièmement, le logiciel est sans cesse en mouvement : modifié, corrigé, amélioré -- dans ces opérations quelle est la part relevant de l'architecture et quelle est la part relevant du simple ornement ?

Devant cette difficulté à définir précisément ce qu'est l'architecture d'un logiciel¹, nous avons décidé de procéder de manière semblable au cheminement proposé dans l'ouvrage *Beautiful Architecture*. Nous allons procéder en donnant quelques principes généraux et éléments d'appréciation avant de proposer quelques exemples de motifs architecturaux communs.

2.1 Principes généraux

Nous présentons ici quelques-uns des principes qui guident les développeurs et architectes lorsqu'ils conçoivent du logiciel. Il est cependant important de garder à l'esprit que l'architecture

1. Difficulté largement partagée dans la littérature.

est essentiellement un jeu de compromis (*trade-offs*) ; il faut non seulement déterminer les principes auxquels on adhère, mais également savoir équilibrer des principes potentiellement contradictoires.

Diviser pour régner (*divide and conquer*) il faut subdiviser les problèmes trop gros pour atteindre des tailles rendant les sous-problèmes plus faciles à appréhender.

Pratique une architecture ne doit pas simplement être belle, elle doit être facile à maintenir pour les développeurs.

Usage le logiciel est avant tout à destination des utilisateurs, pour lesquels l'architecture peut être invisible.

Minimisation des concepts utilisés utiliser la meilleure solution en chaque point peut conduire à du code plus difficile à maintenir.

Une chose à un endroit la duplication de l'information mène inévitablement à des difficultés de maintenance.

Versatilité idéalement une architecture doit pouvoir s'adapter à divers usages.

Couplage faible plus les composants sont faiblement couplés (c'est-à-dire utilisables de manière indépendante), plus il sera facile d'en remplacer un sans remettre en cause l'architecture dans son ensemble.

YAGNI (*You Ain't Gonna Need It*) il n'est pas nécessaire d'architecturer ou de programmer des composants inutiles (*p. ex.* en visant une trop grande généralité).

KISS (*Keep It Simple, Stupid!*) la recherche de la simplicité doit guider les choix de conception.

2.2 Éléments d'appréciation

S'il est, on l'a dit, difficile de définir ce qu'est l'architecture et donc plus encore ce qu'est une bonne architecture, il est en général facile de reconnaître une *belle*² ou une *mauvaise* architecture.

Le premier élément d'appréciation tient aux principes mis en place, à leur uniformité, et facilité d'appréhension. Une bonne architecture est en ce sens une architecture homogène, aisément compréhensible et fondée sur un petit nombre de concepts.

Le second élément d'appréciation est le *test of time* (ou *épreuve du temps*). Une architecture est bonne si elle a su sans trop de modifications rester d'actualité en dépit de l'évolution de l'écosystème technologique. Deux exemples classiques sont *Unix* et le *web* ; tous deux sont bâtis sur un petit nombre de concepts qui ont résisté aux apports technologiques. D'autres architectures peuvent être uniformes à un instant donné, mais être submergées par l'entropie au fil du temps.

2. Une des acceptations du mot est relative à l'adéquation d'un objet à sa fonction.

3 Architectures en couches

3.1 Définition

Un type d'architecture parmi les plus communs est l'architecture en couches. Le plus souvent, il s'agit de couches horizontales empilées les unes sur les autres. Ce type d'architecture, illustré par la figure [Figure 3.1 \(a\)](#) exprime en général l'intuition que les couches les plus hautes sont aussi les plus abstraites.

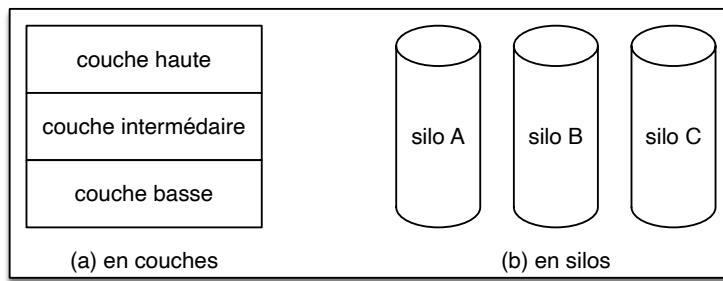


FIGURE 3.1 – Architectures en couches et en silos.

Ainsi, dans une architecture en couches, on trouvera dans les couches basses les librairies de base (*p. ex.* celles interagissant directement avec le matériel). Les couches intermédiaires proposeront des abstractions permettant idéalement de s'affranchir des notions des couches de bases. Enfin, les couches supérieures seront celles dans lesquelles on trouvera le code réellement spécifique à l'application.

Du point de vue du code, on estime être face à une architecture en couches si les classes d'une couche ne font référence directement qu'à des classes de la même couche et occasionnellement d'une couche voisine. Comme il est fréquent que cette contrainte stricte ne soit pas respectée, on voit souvent des architectures principalement en couches, mais possédant un élément qui traverse les couches, comme le montre la figure [Figure 3.2](#).

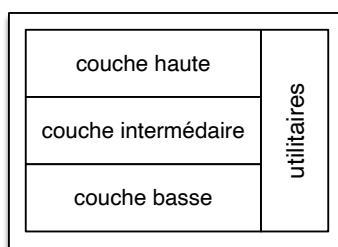


FIGURE 3.2 – Architecture en couches, avec élément transverse.

Enfin, si l'on veut éviter d'exprimer l'idée d'abstraction croissante (et seulement l'idée de séparation), on peut avoir recours à une architecture dite en silos (*cf.* figure [Figure 3.1 \(b\)](#)). Cela permet de faire ressortir le fait que les différents éléments sont de niveau d'abstraction comparable. De même que pour une architecture en couches, on souhaite souvent qu'un silo ne soit en relation directe qu'avec les silos voisins.

3.2 Exemple

Un exemple classique d'architecture en couche est celui des *piles réseaux* :

- les couches basses concernent le matériel et les *drivers* associés ;
- les couches intermédiaires s'intéressent au transport de données entre deux machines ;
- les couches supérieures définissent des protocoles d'échanges entre applications.

On *monte en abstraction* du matériel à l'application, en passant par le transport.

4 Pipelines

4.1 Définition

L'architecture de type *pipeline* est également très commune. Il s'agit d'une des notions centrales du système d'exploitation *Unix* et son *shell*. Les notions de redirections (*p. ex.* <, >) et de *pipe* (*i. e.* |) permettent de combiner facilement des programmes en ligne de commande.

L'idée du pipeline, illustrée à la figure Figure 4.1, est d'enchaîner les traitements les uns à la suite des autres. D'une certaine manière, il ne s'agit que de la composition au sens mathématique de plusieurs fonctions ; au lieu d'écrire $y = f(g(h(x)))$ on écrit en *shell* `h < x | g | f > y` qui présente du reste les calculs de gauche à droite.



FIGURE 4.1 – Architecture en *pipeline*.

Bien sûr, une telle architecture prend toute sa puissance lorsque les éléments composés peuvent être plus riches que des traitements séquentiels sur des listes d'éléments, par exemple des filtres qui éliminent certains éléments. Cette architecture est également bien adaptée au cas où les données ne sont disponibles que petit à petit ; chaque élément passe à son tour dans le *pipeline*. Elle se prête donc à la parallélisation : les traitements successifs peuvent s'effectuer en parallèle, à condition qu'ils soient indépendants (*p. ex.* pas de variable globale partagée par les traitements).

Enfin, de telles architectures s'expriment très proprement dans des langages disposant du mécanisme d'exception. Si les types d'exceptions qui peuvent être levés par les différentes phases sont disjoints, on peut écrire sans difficulté la composition d'un côté et les traitements d'erreurs d'un autre côté. Il est également en général possible de tester chaque sous-partie indépendamment.

4.2 Exemples

Le premier exemple classique d'utilisation d'une telle architecture est l'écriture de compilateurs, typiquement :

1. phases d'analyses lexico-syntactiques (identification des mots et des expressions) ;
2. phases de transformation de l'arbre syntaxique ;
3. phases de production du code binaire.

Un autre exemple commun d'utilisation d'une architecture en pipeline est le traitement à la chaîne d'images ou de vidéos. Il est ainsi fréquent qu'une suite de traitements doive être effectuée sur un

grand nombre d'images (qu'elles soient ou non constitutives d'une animation) ou d'échantillons sonores.

5 MVC

5.1 Définition

Le modèle **MVC** pour modèle-vue-contrôleur (ou **model-view-controller**) est une architecture pour les interfaces graphiques. Elle s'applique aussi bien pour les applications classiques que dans le cadre d'applications web. La figure [Figure 5.1](#) illustre cette architecture, les flèches les communications et les pointes des flèches étant doublées lorsque les communications entre éléments sont plus fréquentes.

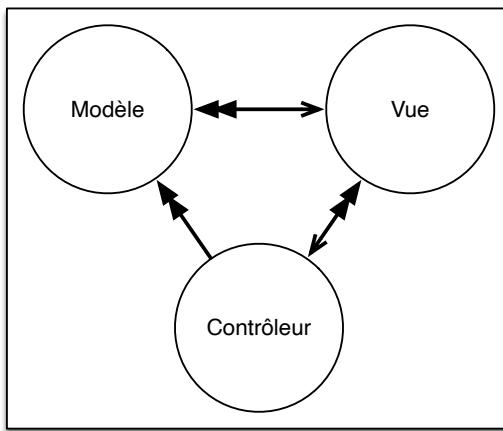


FIGURE 5.1 – Architecture **MVC**.

Le modèle contient l'ensemble des données, ainsi que les traitements associés. La vue est l'artefact graphique par lequel l'utilisateur peut lire ou modifier ces données (il est possible d'avoir plusieurs vues partageant un même modèle). Enfin, le contrôleur orchestre les changements nécessaires : si les données sont modifiées, il faut en informer la vue ; si une action telle qu'un clic de souris est effectuée, il faut la prendre en compte depuis la vue jusqu'au modèle.

5.2 Exemple

La figure [Figure 5.2](#) représente l'architecture de l'application Emacs (éditeur de texte), elle est reprise du chapitre 11 de l'ouvrage *Beautiful Architecture*¹. Comme on le voit, bien qu'articulée autour des concepts **MVC**, elle en diffère légèrement quant à la communication entre les divers éléments.

Le modèle contient l'ensemble des données manipulées, et la vue se charge de l'ensemble de l'affichage. Le contrôleur permet à la fois de modifier les informations textuelles et visuelles.

1. Références complètes en fin de document.

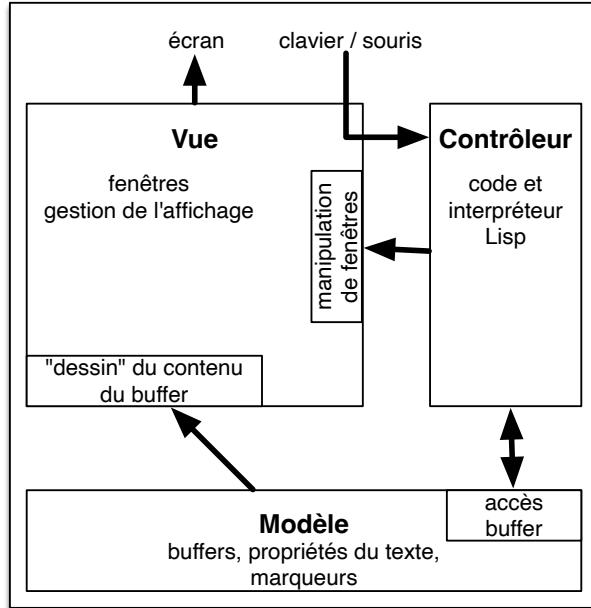


FIGURE 5.2 – Architecture MVC de l'application GNU Emacs.

6 Architectures client-serveur

6.1 Définition

Une architecture client-serveur est composée de deux programmes interagissant : le serveur qui propose des services et le client qui les consomme. Le terme "service" est pris dans son acception la plus large : il peut s'agir simplement de données mises à disposition, ou bien de traitements sur des données. En général, dans ce modèle, il y a un serveur qui contient les données faisant foi et des clients qui peuvent lire et/ou modifier ces données. Il s'agit donc d'une architecture centralisée (autour du serveur).

Il arrive fréquemment que le serveur ne soit pas d'un seul bloc, mais composé de plusieurs entités, on parle dans ce cas d'architecture multi tiers. Le terme "tiers" est à entendre au sens de "tierce partie" et non $\frac{1}{3}$. Ainsi parler d'architecture 3-tiers n'est pas un pléonasme, et parler d'architecture 4-tiers n'est pas une erreur.

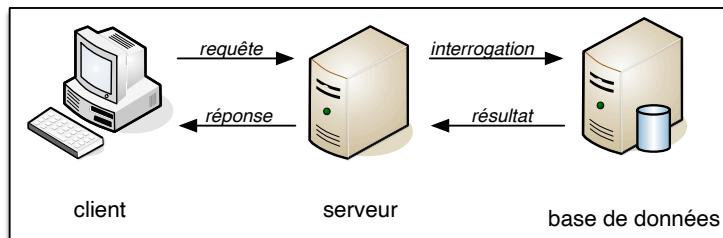


FIGURE 6.1 – Architecture 3-tiers.

6.2 Exemple

L'exemple canonique d'architecture client-serveur 3-tiers est le cas d'un site web adossé à une base de données. Comme le montre la figure [Figure 6.1](#), les 3 tiers sont :

- le *client* qui utilise son navigateur Internet pour émettre des requêtes et afficher les réponses associées ;
- le *serveur* qui reçoit des requêtes d'un nombre quelconque de clients et fournit des réponses en interrogeant la base de données ;
- la *base de données* qui contient l'ensemble des données manipulées par le serveur.

7 Architectures *cloud computing*

7.1 Présentation générale

Superficiellement, le *cloud computing* ressemble à un système client-serveur : des clients, de natures différentes (*p. ex.* ordinateurs, téléphones, tablettes) se connectent à un (ensemble de) serveur(s) centralisé(s). La grande promesse qui différencie le *cloud computing* du client-serveur traditionnel est la dimension *elastic computing*, c'est-à-dire l'idée selon laquelle les ressources mises à disposition vont s'adapter à la demande.

Cette dimension d'adaptation dynamique est simple à appréhender dans le cas de l'hébergement d'un site web. Dans le cadre classique, lorsque l'on souscrit un abonnement, on choisit un ensemble de caractéristiques pour l'hébergement, typiquement espace disque, puissance machine, et bande passante réseau. Même s'il est souvent possible de changer d'abonnement, on voit que le principe est d'estimer *a priori* les nécessités de service pour choisir l'offre la mieux calibrée, à savoir : (i) au-dessus des nécessités pour fournir le service attendu et (ii) *pas trop* au-dessus pour éviter des coûts trop importants.

Dans le cadre d'un hébergement de type *cloud computing*, l'hébergé dispose typiquement d'un tableau de bord lui permettant à chaque instant de changer les paramètres de son hébergement pour les ajuster aux besoins du moment. Il n'y a donc pas de nécessité de calibrer sa demande *a priori*, et l'on ne paie à chaque instant que ce qui est strictement nécessaire pour offrir un service de qualité.

7.2 Différentes formes

Il est d'usage de distinguer trois types d'hébergement *cloud*, correspondant à des niveaux de service plus ou moins riches.

IaaS (Infrastructure As A Service) Ce type de *cloud* correspond essentiellement à l'externalisation d'un parc informatique. Plutôt que d'administrer elle-même des machines, une entreprise se connecte depuis des postes *légers* (*i. e.* essentiellement chargés de l'affichage) à des serveurs dans le *cloud*. L'entreprise a alors la possibilité de demander plus ou moins de puissance de calcul et de stockage.

PaaS (Platform As A Service) Ce type de *cloud* est une extension de *IaaS*. Dans ce cadre, outre l'externalisation de machines, l'entreprise souhaite également externaliser l'installation du logiciel de base (système d'exploitation, serveur web, *etc*). De ce fait, elle se libère du travail de gestion de la configuration et des mises à jour.

SaaS (Software As A Service) Ce type de *cloud* est une extension de *PaaS*. Dans ce cadre, l'externalisation est complète et fournit une solution clefs en main à l'entreprise. Outre le logiciel de base, tous les applicatifs nécessaires (*p. ex.* tableur, traitement de texte) sont également installés par l'opérateur de *cloud*. L'entreprise peut se contenter de disposer dans ses murs de machines uniquement pourvues d'un navigateur web.

La figure Figure 7.1¹ résume les différents types de cloud, en mettant en exergue les parties externalisées selon le type.

7.3 Questions soulevées

Bien entendu, si l'externalisation de tout ou partie du parc informatique d'une entreprise peut être intéressante (notamment dans le cas de petites entreprises qui ne peuvent se permettre le recrutement d'une personne dédiée à ces questions), l'utilisation du cloud pose un certain nombre de problèmes.

Le plus évident est que si l'ensemble des données voire applicatifs ne sont disponibles que par le biais d'une connexion Internet, il devient plus difficile de travailler en déplacement ou dans les transports où une telle connexion peut être intermittente voire impossible.

Cependant, la plus grande question reste celle de la sécurité et confidentialité des données. En effet, dans ce modèle, une entreprise partage une très grande partie de ses données, probablement stratégiques, avec un tiers. Elle perd donc la maîtrise de leurs sécurité et confidentialité, encore que la cryptographie soit une réponse possible. Dans certains cas particuliers, cela peut l'empêcher d'avoir accès à certains *clouds*. Par exemple, certaines législations nationales imposent que les données médicales soient stockées dans le pays d'utilisation du service, rendant factuellement impossible l'usage du *cloud* d'un opérateur situé à l'étranger.

8 Composition de motifs architecturaux

Il est rare, sinon exceptionnel, qu'une application complexe puisse être caractérisée par un seul des modèles d'architecture précédents. Le cas le plus commun est qu'une application est la composition de plusieurs des motifs ci-dessus, et d'autres qui n'ont pas été présentés. Pour s'en tenir à des éléments architecturaux présentés ici, prenons le cas d'une application collaborative permettant de gérer les mails et calendriers d'un certain nombre d'utilisateurs.

Nous allons avoir un serveur qui va contenir l'ensemble des informations, à savoir : utilisateurs, messages échangés et rendez-vous. Sur ce serveur, on peut imaginer une structuration en couches dans laquelle :

- les couches basses se chargent du stockage des différents éléments dans la base de données, de leur transmission sur le réseau ;
- les couches intermédiaires permettent la manipulation des différents éléments ;
- les couches hautes proposent des fonctionnalités avancées telles que la recherche ou la prise de rendez-vous automatique en fonction des agendas.

Pour être accessibles, ces données doivent pouvoir être manipulées par les utilisateurs depuis leur poste de travail. L'application dans son ensemble est ainsi de type client-serveur.

1. Reproduction de http://commons.wikimedia.org/wiki/File:Cloud_Computing_-_les_diff%C3%A9rents_mod%C3%A8les_de_service.svg

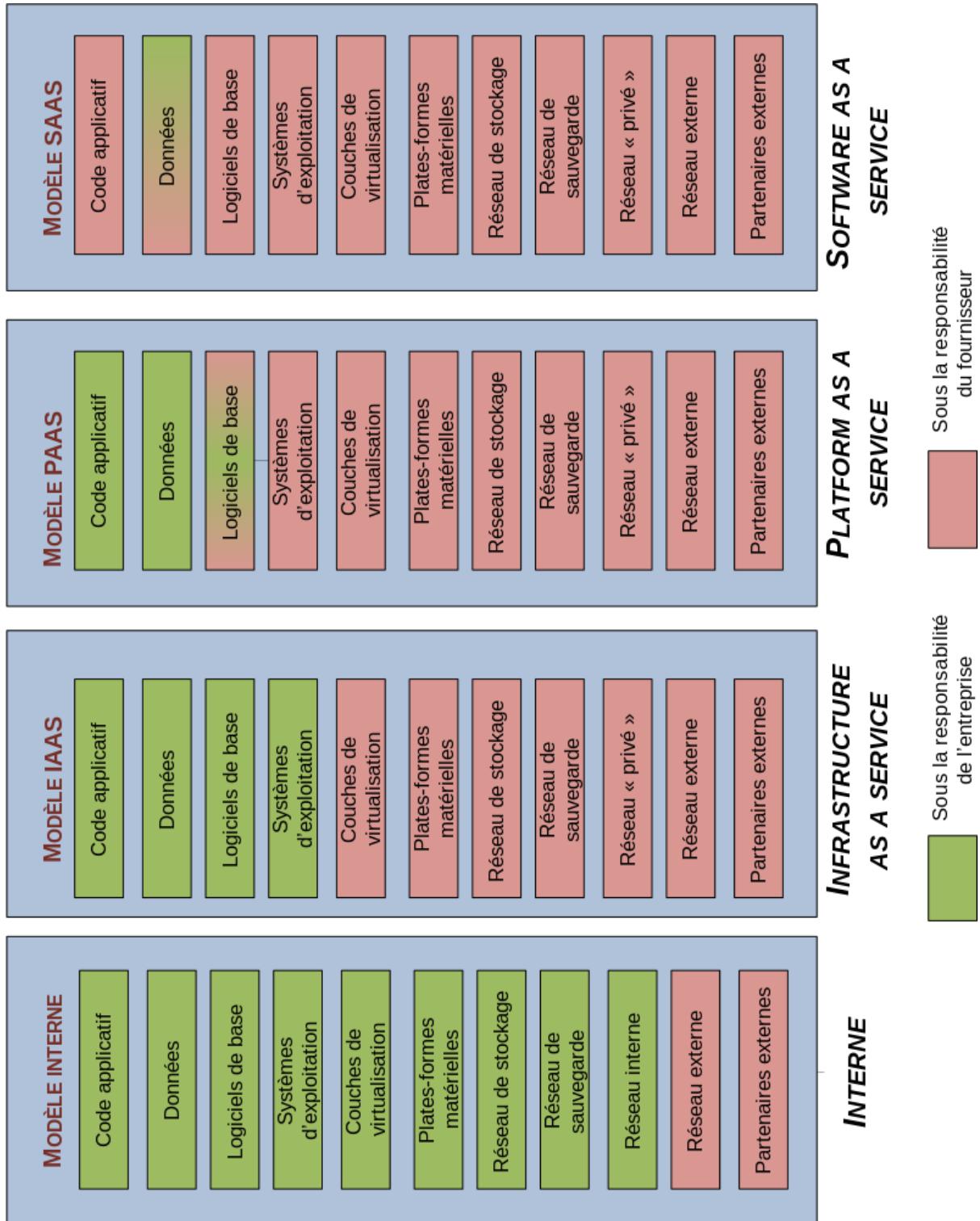


FIGURE 7.1 – Différents types de *cloud*.

L'interface graphique sera vraisemblablement programmée en utilisant les principes de l'architecture MVC.

Si l'on passe en revue l'ensemble de cette section, on se rend compte que l'on a décrit de manière succincte, mais réaliste, un système qui combine les motifs architecturaux suivants :

- architecture en couches ;
- architecture client-serveur ;
- architecture MVC.

9 Interfaces graphiques : généralités

Les interfaces graphiques sont un moyen commode d'interaction avec un système informatique. Elles existent depuis le milieu des années 70 et, si elles ont beaucoup changé superficiellement depuis, la plupart des concepts sous-jacents n'ont en réalité que peu évolué. La grande majorité des langages de programmation propose l'écriture d'interfaces graphiques par le biais de bibliothèques (on trouvera également les termes *framework* ou *toolkit*) reposant sur le paradigme de programmation *événementiel*.

La plupart des interfaces programmées aujourd'hui reposent sur le paradigme WIMP (pour *Windows, Icons, Menus, Pointer*). Il s'agit d'interfaces 2D dont la plupart des composants sont rectangulaires, et sont manipulés à l'aide d'un dispositif de pointage (*p. ex.* souris, *trackpad*). Cependant, on observe une tendance lourde à la multiplication d'interfaces non basées sur ce paradigme (on parle parfois d'interfaces post-WIMP). Par exemple, il y a de nombreuses expérimentations dans le domaine des interfaces en 3D, tout particulièrement pour les tâches de modélisation et de visualisation. Les modes d'interaction sont également plus riches : systèmes *multi-touch*, sans contact¹, etc

Néanmoins, toutes ces interfaces reposent sur la notion de composant (*component*, *control*, ou *widget*) et sont le plus souvent basées sur un modèle objet. De fait, on trouve à la racine de la hiérarchie une classe représentant un composant en toute généralité (celui-ci est par exemple défini par ses dimensions et une méthode d'affichage), et des classes dérivées pour chacun des composants. La relation d'héritage permet d'organiser les composants : typiquement, ceux manipulant du texte, ceux correspondant à des boîtes de dialogue, etc

Lorsque l'on construit une interface graphique, les différents composants qui la constituent s'organisent sous la forme d'un arbre (on parle parfois de *scène*). Ainsi, la fenêtre d'un éditeur de texte aura par exemple deux composants-fils : un pour la barre d'outils et un pour la zone de texte. Le composant de la barre d'outils contient lui-même plusieurs fils, un pour chaque bouton constitutif de la barre.

Cette organisation arborescente n'est pas seulement utile conceptuellement, elle permet de définir comment les différents composants s'affichent les uns par rapport aux autres. Le *layout* (agencement ou mise en page) définit la stratégie d'affichage. Dans notre exemple, la zone de texte occupe la partie centrale de la fenêtre et la barre se trouve sur la partie supérieure. Le *layout* permet entre autres de savoir comment faire évoluer l'organisation relative des composants lorsque la fenêtre est redimensionnée.

Les notions de composant et de *layout* permettent de définir comment une interface graphique s'affiche, mais pas comment elle se comporte. Le modèle usuel de définition du comportement

1. Par exemple <https://www.ultraleap.com>

d'une interface graphique est le modèle événementiel. À chaque interaction possible avec un composant (par exemple un clic) est associé un événement, et le développeur peut indiquer comment l'interface doit réagir lorsqu'un événement donné se produit sur un composant donné.

Lorsqu'une interface graphique est utilisée, il n'y a la plupart du temps pas de calcul en cours. Le système se contente d'afficher les composants, de gérer les interactions de l'utilisateur avec ceux-ci, et le cas échéant de gérer les événements correspondants. Cela se fait par le biais d'une *boucle d'événements* qui se charge d'appeler le code associé à un couple ⟨ composant, événement ⟩ lorsque nécessaire. Selon la librairie utilisée, cette boucle d'événements est soit implicite, soit à appeler explicitement une fois la construction de l'interface terminée.

Pour enregistrer auprès de l'interface le code à exécuter lorsqu'un événement donné se produit sur un composant donné, on a recours à un *callback*. Un *callback* désigne, selon la librairie et le langage de programmation, une fonction, une méthode ou une classe chargée du traitement d'un événement. Son enregistrement se fait en passant cette fonction/méthode/classe en paramètre à une méthode appelée sur le composant. Cette fonction/méthode/classe sera en retour appelée (d'où le terme *callback*) lorsque l'événement se produira sur le composant.

10 Interfaces graphiques : librairie Qt en Python

10.1 Présentation

Qt¹, prononcé comme les mots anglais "cute" ou "cutie", est un *framework* permettant la programmation d'applications graphiques. Il propose en outre un certain nombre de classes pour des tâches *annexes* comme les communications réseau, les accès aux bases de données, *etc*. Qt a été développé successivement par différentes entreprises : Trolltech, Nokia, Digia et aujourd'hui The Qt Company.

Une particularité de Qt est qu'il est *cross-platform*, ce qui signifie qu'il peut être utilisé sur différents systèmes d'exploitation et différentes architectures sans que le code ait à être modifié. Sur les différentes plateformes, Qt présente un aspect *natif*, ce qui signifie que l'application est visuellement semblable à une application développée spécifiquement pour la plateforme. Les plateformes supportées incluent notamment :

- Android ;
- iOS ;
- Linux (en particulier KDE) ;
- macOS ;
- Windows.

Qt est originellement développé pour et en C++, mais il existe de nombreux *bindings* pour d'autres langages. Outre Python, qui nous intéresse ici, Qt est ainsi accessible depuis Java, C#, OCaml, et de nombreux autres.

Les versions actuellement supportées (5. x et 6. x) sont publiées sous les licences GPL et LGPL, ce qui signifie qu'en pratique il est possible de les utiliser dans n'importe quel développement (*open source* ou non). Il existe en outre une version commerciale, qui ouvre droit à un support par The Qt Company (*p. ex. correction de bug, avis de consultants, etc*).

1. Disponible à l'adresse <https://www.qt.io>

10.2 Notions de signal et de slot

Les signaux et slots forment la base du système mis en place par Qt pour la communication entre les objets d'une application. Il s'agit d'une abstraction permettant d'appliquer le *pattern* d'observation. Ce *pattern* est utilisé lorsque des objets (*observateurs*) souhaitent être notifiés des modifications apportées à un autre objet (*observé* ou *observable*). Un exemple classique de ce *pattern* est un tableau : la valeur d'une case, s'il s'agit d'une formule, dépend des valeurs d'autres cases. Il est alors naturel de modéliser cela en indiquant que la case-formule *observe* les valeurs des autres cases. Ainsi, lorsque l'une des autres cases est modifiée, la case-formule est mise à jour.

Un signal Qt représente abstrairement un événement se produisant dans le système, et les composants de Qt prédéfinissent un certain nombre de signaux. Par exemple, il existe un signal pour un clic de souris, ou pour la modification d'une zone de texte.

Un slot Qt est une fonction qui est appelée en réponse à un signal. Il s'agit simplement de fonctions ou de méthodes que l'on enregistre pour qu'elles soient appelées à l'émission de signaux donnés. Un slot Qt est donc essentiellement une forme de *callback*.

Un des traits caractéristiques de l'implémentation originale de Qt est que le système de signaux et de slots est *type safe*, ce qui signifie que le compilateur vérifie que l'on enregistre un slot auprès d'un signal compatible (au sens des types de données manipulés). Sans surprise, cette propriété est perdue en Python, dont le typage est dynamique.

10.3 QtDesigner

L'écriture manuelle d'interfaces graphiques est une tâche souvent fastidieuse, et des outils graphiques ont été développés afin de permettre d'agencer les éléments composant une interface graphiquement plutôt qu'en écrivant des lignes de code.

Pour Qt, le programme QtDesigner permet de créer une interface graphique par glisser-déposer (*drag and drop*). La figure Figure 10.1 présente une capture d'écran de l'outil. On y voit au centre l'interface graphique en cours de construction : barre de menu, cases à cocher, champs de texte et de date, bouton, etc. La palette sur la gauche présente les différents composants disponibles, qu'il suffit de faire glisser sur la partie centrale. La palette sur la droite présente les différentes propriétés du composant sélectionné ; il est ainsi possible de modifier son nom, son apparence, sa géométrie, etc

Les interfaces graphiques créées avec le programme QtDesigner peuvent être sauvegardées sous forme de fichiers .ui (dont le contenu est en pratique un document au format XML). Ces fichiers peuvent ensuite être transformés en code pour un langage donné. Pour Python, il suffit d'exécuter le programme en ligne de commande pyuic6 avec en paramètre un fichier .ui pour générer le code de classes Python implémentant l'interface. Le code suivant donne un exemple de code généré, correspondant à l'interface de la figure Figure 10.1.

```
1 class Ui_MainWindow(object):
2     def setupUi(self, MainWindow):
3         MainWindow.setObjectName("MainWindow")
4         MainWindow.resize(800, 600)
5         self.centralwidget = QtWidgets.QWidget(MainWindow)
6         self.centralwidget.setObjectName("centralwidget")
    (suite sur la page suivante)
```

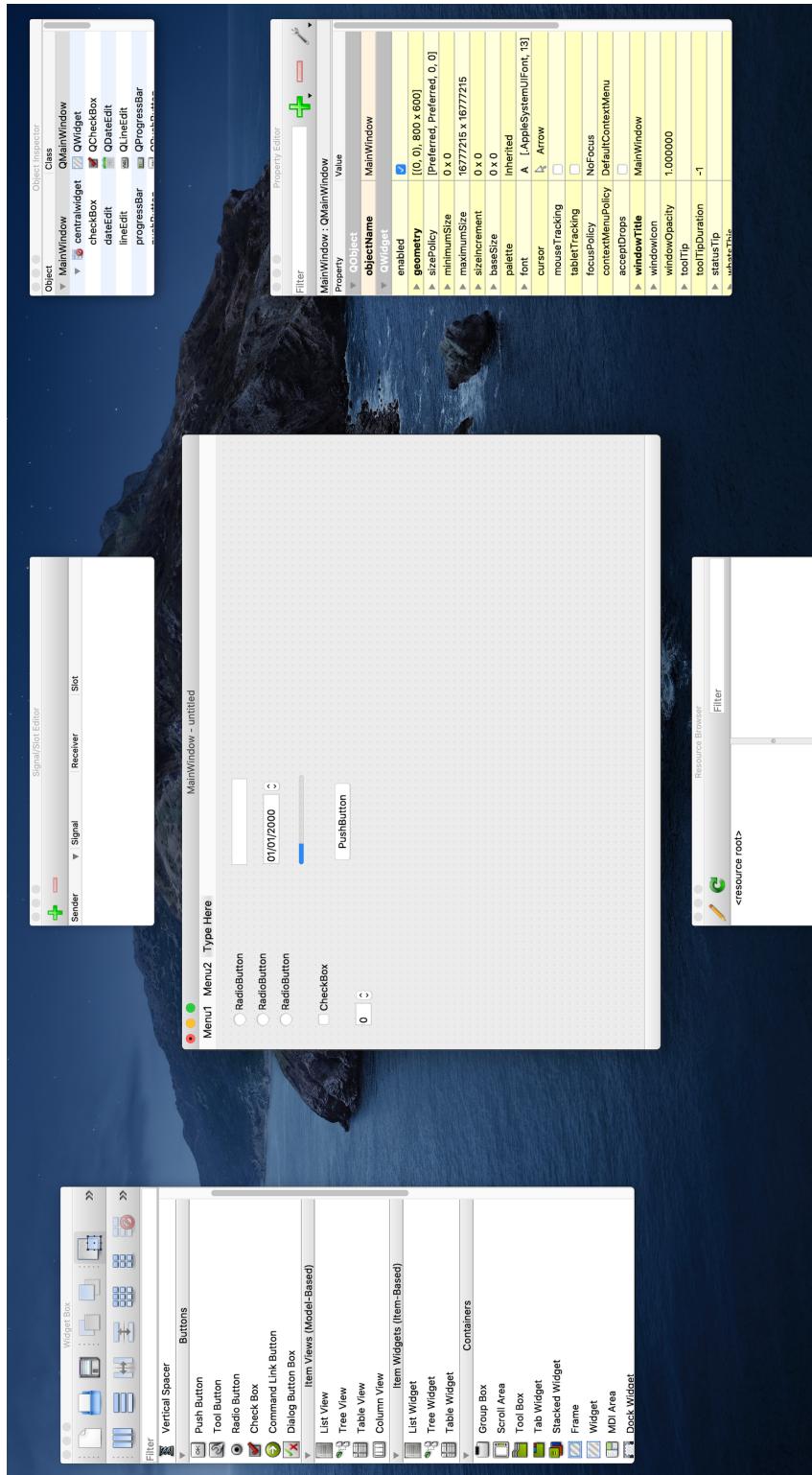


FIGURE 10.1 – Capture d'écran de QtDesigner.

(suite de la page précédente)

```
7     self.radioButton = QtWidgets.QRadioButton(self.centralwidget)
8     self.radioButton.setGeometry(QtCore.QRect(30, 20, 100, 20))
9     self.radioButton.setObjectName("radioButton")
10    self.radioButton_2 = QtWidgets.QRadioButton(self.centralwidget)
11    self.radioButton_2.setGeometry(QtCore.QRect(30, 50, 100, 20))
12    self.radioButton_2.setObjectName("radioButton_2")
13    self.radioButton_3 = QtWidgets.QRadioButton(self.centralwidget)
14    self.radioButton_3.setGeometry(QtCore.QRect(30, 80, 100, 20))
15    self.radioButton_3.setObjectName("radioButton_3")
16    self.checkBox = QtWidgets.QCheckBox(self.centralwidget)
17    self.checkBox.setGeometry(QtCore.QRect(30, 130, 87, 20))
18    self.checkBox.setObjectName("checkBox")
19    self.spinBox = QtWidgets.QSpinBox(self.centralwidget)
20    self.spinBox.setGeometry(QtCore.QRect(30, 180, 48, 24))
21    self.spinBox.setObjectName("spinBox")
22    self.lineEdit = QtWidgets.QLineEdit(self.centralwidget)
23    self.lineEdit.setGeometry(QtCore.QRect(240, 20, 113, 21))
24    self.lineEdit.setObjectName("lineEdit")
25    self.dateEdit = QtWidgets.QDateEdit(self.centralwidget)
26    self.dateEdit.setGeometry(QtCore.QRect(240, 60, 110, 24))
27    self.dateEdit.setObjectName("dateEdit")
28    self.progressBar = QtWidgets.QProgressBar(self.centralwidget)
29    self.progressBar.setGeometry(QtCore.QRect(240, 100, 118, 23))
30    self.progressBar.setProperty("value", 24)
31    self.progressBar.setObjectName("progressBar")
32    self.pushButton = QtWidgets.QPushButton(self.centralwidget)
33    self.pushButton.setGeometry(QtCore.QRect(240, 150, 113, 32))
34    self.pushButton.setObjectName("pushButton")
35    MainWindow.setCentralWidget(self.centralwidget)
36    self.menuubar = QtWidgets.QMenuBar(MainWindow)
37    self.menuubar.setGeometry(QtCore.QRect(0, 0, 800, 22))
38    self.menuubar.setObjectName("menuubar")
39    self.menuMenu1 = QtWidgets.QMenu(self.menuubar)
40    self.menuMenu1.setObjectName("menuMenu1")
41    self.menuMenu2 = QtWidgets.QMenu(self.menuubar)
42    self.menuMenu2.setObjectName("menuMenu2")
43    MainWindow.setMenuBar(self.menuubar)
44    self.statusbar = QtWidgets.QStatusBar(MainWindow)
45    self.statusbar.setObjectName("statusbar")
46    MainWindow.setStatusBar(self.statusbar)
47    self.menuubar.addAction(self.menuMenu1.menuAction())
48    self.menuubar.addAction(self.menuMenu2.menuAction())
49    . . .
```

10.4 Qt et Python

Il existe plusieurs *bindings* Python pour Qt, les deux plus utilisés étant PyQt² et PySide³. Nous présentons ici PyQt qui est plus activement maintenu et propose un bon support de QtDesigner. Une documentation complète de PyQt6 est disponible aux adresses suivantes :

- <https://wiki.python.org/moin/PyQt> pour une collection de liens ;
- <https://www.riverbankcomputing.com/static/Docs/PyQt6/> pour le point d'entrée de la documentation ;
- <https://www.riverbankcomputing.com/static/Docs/PyQt6/sip-classes.html> pour la liste des classes Qt accessibles depuis Python.

Structure d'un programme

Les composants graphiques de Qt sont accessibles par le module `PyQt6.QtWidgets`, en particulier la classe `QApplication` qui se charge de la gestion de la boucle d'événements et permet de configurer les paramètres valables pour l'application dans son ensemble (*p. ex.* thème, configuration de la souris, *etc*). Le déclenchement de la boucle d'événements est explicite, et s'effectue par un appel à la méthode `exec()`. Celle-ci retourne un entier qui correspond au code de sortie de l'application. La structure habituelle d'un programme est donc la suivante :

```
1 import sys
2 import PyQt6
3 import PyQt6.QtWidgets as widgets
4
5 application = widgets.QApplication(sys.argv)
6     ...
7 sys.exit(application.exec())
```

où l'ellipse correspond le plus souvent à la création et à l'affichage de la fenêtre principale. Le fait de passer `sys.argv` au constructeur de l'application lui permet de lire sur la ligne de commande un certain nombre de paramètres contrôlant le comportement de Qt (*p. ex.* `-im` permet de choisir la méthode de saisie).

Widgets de base

Le premier composant (*widget*) à utiliser est probablement `QMainWindow` qui représente, comme son nom l'indique, la fenêtre principale d'une interface. Le programme se termine lorsque cette fenêtre est fermée. Une manière typique de procéder est d'écrire une classe qui hérite de `QMainWindow` (ou `QWidget`) et qui crée dans le constructeur (ou une méthode appelée depuis le constructeur) les différents éléments de l'interface. Le programme suivant montre le canevas de base d'un programme PyQt.

```
1 import sys
2 import PyQt6
3 import PyQt6.QtWidgets as widgets
4
```

(suite sur la page suivante)

2. <https://www.riverbankcomputing.com/software/pyqt/intro>
3. <https://wiki.qt.io/PySide2>

```

5  class MyWindow(widgets.QMainWindow):
6      def __init__(self):
7          super().__init__(None)
8          ... # instructions to build the interface
9          self.setGeometry(200, 100, 600, 400) # size and position of
10         ↴ the window
11
12 if __name__ == "__main__":
13     application = widgets.QApplication(sys.argv)
14     main_window = MyWindow()
15     main_window.show()
16     sys.exit(application.exec())

```

L'ajout de *widgets* peut alors se faire simplement en appelant le constructeur d'un élément et en lui passant en paramètre le composant *parent* (*i. e.* celui dans lequel il doit s'afficher). Ainsi, pour ajouter un bouton à notre classe fenêtre, il suffit d'ajouter la ligne suivante dans le constructeur :

```
self.a_button = widgets.QPushButton("click me", self)
```

Nous ne détaillons évidemment pas tous les *widgets* disponibles, et renvoyons à la documentation de PyQt. Voici néanmoins les plus communs :

- bouton : classe `PyQt6.QtWidgets.QPushButton`;
- case à cocher : classe `PyQt6.QtWidgets.QCheckBox`;
- case à cocher parmi plusieurs : classe `PyQt6.QtWidgets.QRadioButton`;
- liste : classe `PyQt6.QtWidgets.QListWidget`;
- liste déroulante : classe `PyQt6.QtWidgets.QComboBox`;
- champ de texte (une ligne) : classe `PyQt6.QtWidgets.QLineEdit`;
- champ de texte (plusieurs lignes) : classe `PyQt6.QtWidgets.QTextEdit`;
- dialogue d'ouverture/sauvegarde de fichier : classe `PyQt6.QtWidgets.QFileDialog`;
- classe parente des dialogues : classe `PyQt6.QtWidgets.Dialog`.

Signaux de base

Les signaux disponibles pour un *widget* donné sont listés sur la page de la documentation correspondant à la classe de ce *widget*. Ensuite, pour lier une méthode à un signal, il suffit d'écrire une ligne semblable à la ligne suivante :

```
a_component.signal_name.connect(self.a_method)
```

Ainsi, pour lier la méthode `a_button_was_clicked` au bouton défini ci-dessus, il suffit d'écrire :

```
self.a_button.clicked.connect(self.a_button_was_clicked)
```

et de définir la méthode comme suit :

```
def a_button_was_clicked(self):
    ...
```

Le signal `clicked` est le plus commun. Toujours pour un bouton, il est possible d'enregistrer un *callback* appelé lorsque celui-ci est pressé (`pressed`) ou relâché (`released`). Les *widgets* permettant la saisie de valeurs proposent des signaux qui sont déclenchés lorsque la valeur est modifiée, ainsi :

- `PyQt6.QtWidgets.QLineEdit` et `PyQt6.QtWidgets.QTextEdit` proposent un signal `textChanged`;
- `PyQt6.QtWidgets.QCheckBox` propose un signal `stateChanged`.

Certains signaux, comme `stateChanged`, acceptent que la méthode utilisée comme *callback* prenne un paramètre additionnel qui contiendra alors des informations sur l'événement ayant déclenché le signal. Dans le cas de `stateChanged`, il s'agira du nouvel état du *widget*.

Mise en page

Les éléments présentés jusqu'à maintenant permettent de créer une interface graphique par l'ajout de composants et l'enregistrement de *callbacks* exécutés lorsque des événements donnés se produisent. Si cela est suffisant pour obtenir des interfaces graphiques fonctionnelles, cela n'est pas suffisant pour obtenir des interfaces graphiques *utilisables*.

Il est en effet nécessaire de pouvoir placer les composants librement afin de pouvoir les agencer de manière ergonomique (et esthétique). Une première manière de positionner les composants est de le faire de manière *absolue*, en utilisant par exemple la méthode `setGeometry(x, y, largeur, hauteur)`.

Cependant, cette manière de faire n'est pas seulement fastidieuse, elle implique en outre de gérer manuellement un certain nombre de contraintes implicites comme par exemple l'alignement des différents composants. Pour ces raisons, on a le plus souvent recours à des *layouts* qui se chargent automatiquement de l'agencement, selon des règles prédéfinies.

Par exemple, les *layouts* `PyQt6.QtWidgets.QHBoxLayout` et `PyQt6.QtWidgets.QVBoxLayout` permettent respectivement de placer les composants horizontalement et verticalement, en utilisant toute la place disponible et respectant les règles d'alignement. Le *layout* `PyQt6.QtWidgets.QGridLayout` permet de placer les *widgets* dans une grille, là encore en respectant les règles d'alignement.

Lorsque l'on souhaite utiliser un *layout* dans une fenêtre, il faut instancier un *widget* qui contiendra les composants de ce *layout* puis indiquer que le *widget* est l'élément central de l'interface. Cela se fait par exemple de la manière suivante :

```
central = widgets.QWidget()
horizontal = widgets.QHBoxLayout()
horizontal.addWidget(...)
...
horizontal.addWidget(...)
central.setLayout(horizontal)
self.setCentralWidget(central)
```

Annexes

À retenir

Parmi les motifs architecturaux présentés, les plus souvent rencontrés (et donc les plus importants à connaître et reconnaître) sont : les architectures pipeline, MVC, et client/serveur.

Concernant les interfaces graphiques, l'essentiel est d'en comprendre les concepts de base : composant, *layout*, événement, boucle d'événements, et *callback*.

Références

- "Beautiful Architecture" (Diomidis Spinellis et Giorgios Gousios, O'Reilly) propose une quinzaine de chapitres indépendants, chacun présentant l'architecture d'un projet (logiciel et/ou matériel).
- <https://www.riverbankcomputing.com/static/Docs/PyQt6/> est le point d'entrée de la documentation de PyQt6, qui propose notamment une documentation complète de chaque composant/classe.
- <http://zetcode.com/gui/pyqt5/> propose des tutoriels pour l'utilisation de PyQt5.