Architecture Logicielle

2018

Partie 1 : Introduction à l'Architecture Logicielle
Partie 2 : Les patrons de conception

Les objectifs généraux de ce cours sont :

Permettre à l'étudiant d'acquérir des compétences en matière de conception architecturale des applications logicielles réparties.

Les objectifs spécifiques de ce cours sont :

- Comprendre les architectures logicielles ainsi que leurs composants.
- Comprendre les principaux patrons de conception (Design Patterns) et être capable de les mettre en œuvre.

Partie 1 : Introduction à l'Architecture Logicielle

Sommaire

1-Définition

L'**architecture logicielle** décrit d'une manière symbolique et schématique les différents éléments d'un ou de plusieurs systèmes informatiques, leurs interrelations et leurs interactions. Contrairement aux spécifications produites par l'analyse fonctionnelle, le modèle d'architecture, produit lors de la phase de conception, ne décrit pas ce que doit réaliser un système informatique mais plutôt comment il doit être conçu de manière à répondre aux spécifications. L'analyse décrit le « **quoi faire** » alors que l'architecture décrit le « **comment le faire** ».

2-Contexte et motivation

La phase de conception logicielle est l'équivalent, en informatique, à la phase de conception en ingénierie traditionnelle (mécanique, civile ou électrique); cette phase consiste à réaliser entièrement le produit sous une forme abstraite avant la production effective. Par contre, la nature immatérielle du logiciel (modelé dans l'information et non dans la matière), rend la frontière entre l'architecture et le produit beaucoup plus floue que dans l'ingénierie traditionnelle. L'utilisation d'outils CASE (Computer-aided software engineering) ou bien la production de

l'architecture à partir du code lui-même et de la documentation système permettent de mettre en évidence le lien étroit entre l'architecture et le produit.

L'architecture logicielle constitue le plus gros livrable d'un processus logiciel après le produit (le logiciel lui-même). En effet, la phase de conception devrait consommer autour de 40 %[1] de l'effort total de développement et devrait être supérieure ou égale, en effort, à la phase de codage mais il peut être moindre. L'effort dépend grandement du type de logiciel développé, de l'expertise de l'équipe de développement, du taux de réutilisation et du processus logiciel.

Les deux objectifs principaux de toute architecture logicielle sont la réduction des coûts et l'augmentation de la qualité du logiciel; la réduction des coûts est principalement réalisée par la réutilisation de composants logiciels et par la diminution du temps de maintenance (correction d'erreurs et adaptation du logiciel). La qualité, par contre, se trouve distribuée à travers plusieurs critères; la norme ISO 9126 est un exemple d'un tel ensemble de critères.

2.1 Critères de qualité logicielle.

L'interopérabilité extrinsèque exprime la capacité du logiciel à communiquer et à utiliser les ressources d'autres logiciels comme les documents créés par une certaine application.

L'interopérabilité intrinsèque exprime le degré de cohérence entre le fonctionnement des commandes et des modules à l'intérieur d'un système ou d'un logiciel.

La portabilité exprime la possibilité de compiler le code source et/ou d'exécuter le logiciel sur des plates-formes (machines, systèmes d'exploitation, environnements) différentes.

La compatibilité exprime la possibilité, pour un logiciel, de fonctionner correctement dans un environnement ancien (compatibilité descendante) ou plus récent (compatibilité ascendante).

La validité exprime la conformité des fonctionnalités du logiciel avec celles décrites dans le cahier des charges.

La vérifiabilité exprime la simplicité de vérification de la validité.

L'intégrité exprime la faculté du logiciel à protéger ses fonctions et ses données d'accès non autorisés.

La fiabilité exprime la faculté du logiciel à gérer correctement ses propres erreurs de fonctionnement en cours d'exécution.

La maintenabilité exprime la simplicité de correction et de modification du logiciel, et même, parfois, la possibilité de modification du logiciel en cours d'exécution.

La réutilisabilité exprime la capacité de concevoir le logiciel avec des composants déjà conçus tout en permettant la réutilisation simple de ses propres composants pour le développement d'autres logiciels.

L'extensibilité exprime la possibilité d'étendre simplement les fonctionnalités d'un logiciel sans compromettre son intégrité et sa fiabilité.

L'efficacité exprime la capacité du logiciel à exploiter au mieux les ressources offertes par la ou les machines où le logiciel sera implanté.

L'autonomie exprime la capacité de contrôle de son exécution, de ses données et de ses communications.

La transparence exprime la capacité pour un logiciel de masquer à l'utilisateur (humain ou machine) les détails inutiles à l'utilisation de ses fonctionnalités.

La composabilité exprime la capacité pour un logiciel de combiner des informations provenant de sources différentes.

La convivialité décrit la facilité d'apprentissage et d'utilisation du logiciel par les usagers.

2.2 Diminution de la dégradation du logiciel

Une architecture faible ou absente peut entraîner de graves problèmes lors de la maintenance du logiciel. En effet, toute modification d'un logiciel mal architecturé peut déstabiliser la structure de celui-ci et entraîner, à la longue, une dégradation (principe d'entropie du logiciel). L'architecte informatique devrait donc concevoir, systématiquement, une architecture maintenable et extensible.

Dans les processus itératifs comme UP (Unified Process), la gestion des changements est primordiale. En effet, il est implicitement considéré que les besoins des utilisateurs du système peuvent changer et que l'environnement du système peut changer. L'architecte informatique a donc la responsabilité de prévoir le pire et de concevoir l'architecture en conséquence ; la plus maintenable possible et la plus extensible possible.

Bien des logiciels ont été créés sans architecture par plusieurs générations de développeurs, ayant chacun usé d'une imagination débordante pour réussir à maintenir l'intégrité du système. Une telle absence d'architecture peut être qualifiée *d'architecture organique*. En effet, un développeur confronté à une telle architecture a plus l'impression de travailler avec un organisme vivant qu'avec un produit industriel. Il en résulte que la complexité du logiciel fait en sorte que celui-ci est extrêmement difficile à comprendre et à modifier. À la limite, modifier une partie du système est plus proche, en complexité, de la transplantation cardiaque que du changement de carburateur

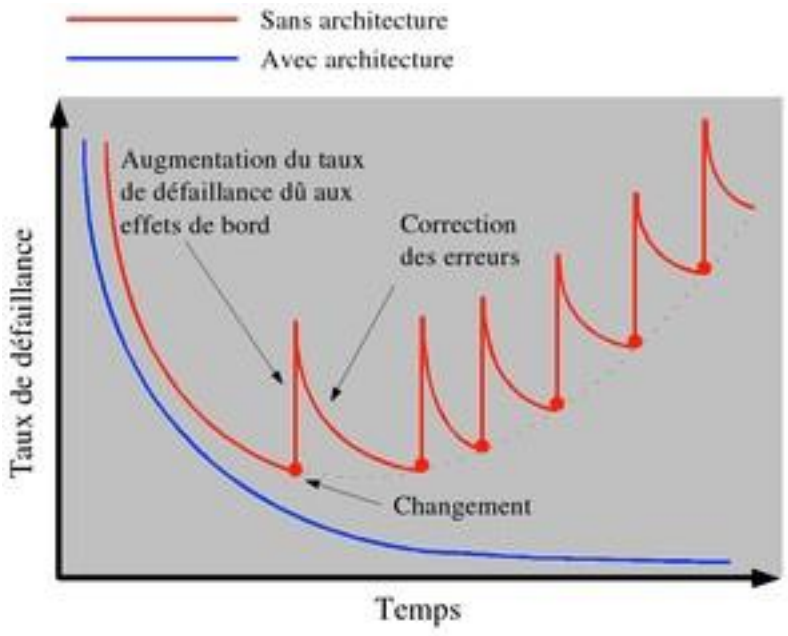

Figure 1 : Par œuvre personnelle, inspirée de : Pressman R. S., Software Engineering: A Practitioner's Approach, Fifth Edition. McGraw-Hill. Chapitre 1, p. 8, 2001., CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1801195

2.3 Développement pour et par la réutilisation

La réutilisation de composants logiciels est l'activité permettant de réaliser les économies les plus substantielles[2], encore faut-il posséder des composants à réutiliser. De plus, la réutilisation de composants nécessite de créer une architecture logicielle permettant une intégration harmonieuse de ces composants[3]. Le développement par la réutilisation logicielle impose donc un cycle de production-réutilisation perpétuel et une architecture logicielle normalisée.

Une réutilisation bien orchestrée nécessite la création et le maintien d'une bibliothèque logicielle et un changement de focus; créer une application revient à créer les composants de bibliothèque nécessaires puis à construire l'application à l'aide de ces composants. Une telle bibliothèque, facilitant le développement d'application est un framework (cadriciel) d'entreprise et son architecture, ainsi que sa documentation sont les pierres angulaires de la réutilisation logicielle en entreprise.

Le rôle de l'architecte informatique se déplace donc vers celui de bibliothécaire. L'architecte informatique doit explorer la bibliothèque pour trouver les composants logiciels appropriés puis créer les composants manquants, les documenter et les intégrer à la bibliothèque. Dans une grande entreprise, ce rôle de bibliothécaire est rempli par l'architecte informatique en chef qui est responsable du développement harmonieux de la bibliothèque et de la conservation de l'intégrité de son architecture.

3-Les modèles d’architecture

3.1 Introduction

La description d'un système complexe comme un logiciel informatique peut être faite selon plusieurs points de vue différents mais chacun obéit à la formule de Perry et Wolf:

architecture = elements + formes + motivations

Selon le niveau de granularité, les éléments peuvent varier en tailles (lignes de code, procédures ou fonctions, modules ou classes, paquetages ou couches, applications ou systèmes informatiques), ils peuvent varier en raffinement (ébauche, solution à améliorer ou solution finale) et en abstraction (idées ou concepts, classes ou objets, composants logiciels). Les éléments peuvent également posséder une *temporalité* (une existence limitée dans le temps) et une *localisation* (une existence limitée dans l'espace). ***3.2 Le modèle conventionnel***

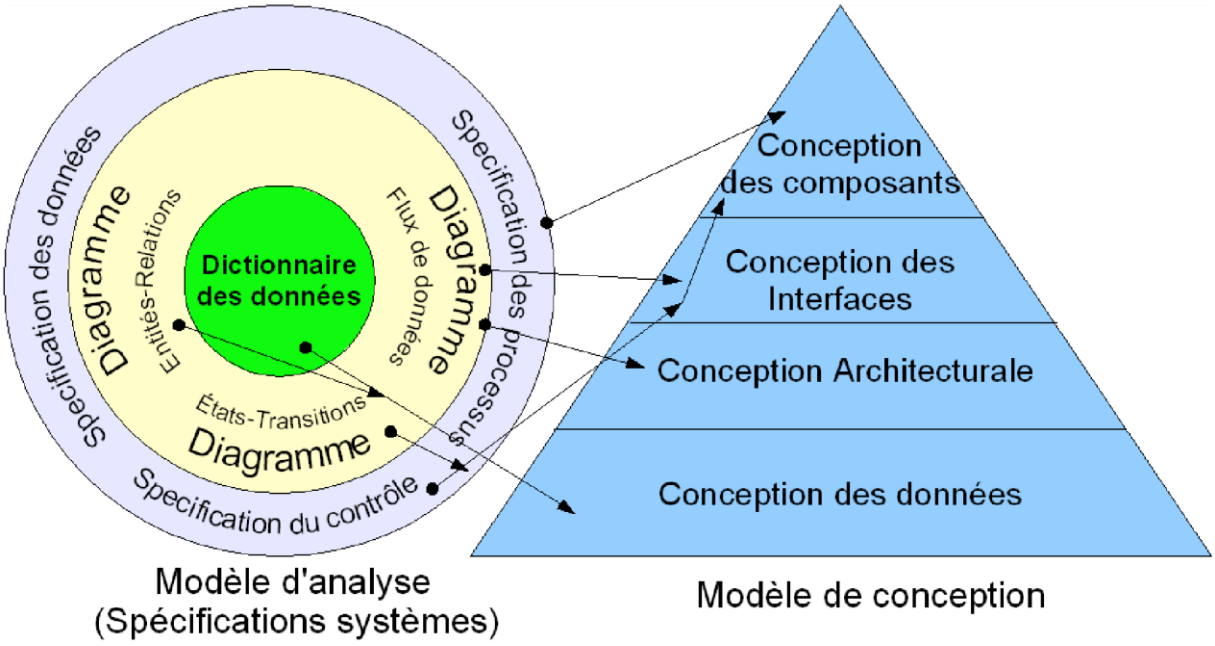

Figure 2 : Par œuvre personnelle, inspirée de Pressman R. S., Software Engineering: A Practitioner's Approach, Fifth Edition. McGraw-Hill. Chapitre 13, p. 337, 2001, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1798648

Ce diagramme décrit, à gauche, les spécifications systèmes qui sont également représentées par des diagrammes (Entités-Relations, Flux de données, États-Transitions). Et à droite, nous avons les différentes activités de conception prenant comme intrants les livrables de la phase d'analyse. Nous voyons que l'architecture logicielle traditionnelle nécessiterait de produire au moins quatre vues distinctes : une architecture des données (conception des données), une architecture fonctionnelle et/ou modulaire (conception architecturale), une autre architecture fonctionnelle et/ou modulaire pour les interfaces utilisateurs (conception des interfaces) et une architecture détaillée (ordinogrammes, états-transitions) des différents modules (conception des composants).

La pyramide exprime que chaque couche est bâtie sur la précédente. En effet, les composants réalisant les fonctionnalités du logiciel doivent manipuler des éléments de données qui doivent donc être préalablement décrits. De même, les composants réalisant les interfaces utilisateurs doivent utiliser les fonctionnalités du logiciel préalablement décrites. Et finalement, la création de l'architecture détaillée de chacun des composants du logiciel nécessite, évidemment, que ceux-ci soient préalablement inventés.

Ce modèle d'architecture impose une séparation claire entre les données, les traitements et la présentation.

3.2.1 Modèle d'analyse ou modèle d'architecture ?

Puisque l'analyse produit également des diagrammes, il est naturel de se questionner, en effet, quand se termine l'analyse et quand commence l'architecture ? La réponse à cette question est fort simple : les éléments des diagrammes d'analyse correspondent à des éléments visibles et compréhensibles par les utilisateurs du système, alors que les éléments des diagrammes d'architectures ne correspondent à aucune réalité tangible pour ceux-ci.

3.3 Le modèle des 4 + 1 vues

Le modèle de Kruchten[5] dit modèle des 4 + 1 vues est celui adopté dans l'Unified Process ou Processus Unifié. Ici encore, le modèle d'analyse, baptisé vue des cas d'utilisation, constitue le lien et motive la création de tous les diagrammes d'architecture.

Figure 3 : Par œuvre personnelle, d'après : Muller P-A, Gaertner, N., Modélisation objet avec UML, 2em édition. Eyrolles, p. 202, 2003, CC BY-SA 3.0, https://commons.wikimedia.org/w/index.php?curid=1801284

3.3.1 La vue des cas d'utilisation

La vue des cas d'utilisation est un modèle d'analyse formalisé par Ivar Jacobson. Un cas d'utilisation est défini comme un ensemble de scénarios d'utilisation, chaque scénario représentant une séquence d'interaction des utilisateurs (acteurs) avec le système.

L'intérêt des cas d'utilisation est de piloter l'analyse par les exigences des utilisateurs. Ceux-ci se sentent concernés car ils peuvent facilement comprendre les cas d'utilisation qui les concernent. Cette méthode permet donc d'aider à formaliser les véritables besoins et attentes des utilisateurs; leurs critiques et commentaires étant les briques de la spécification du système.

L'ensemble des cas d'utilisation du logiciel en cours de spécification est représenté par un diagramme de cas d'utilisation, chacun des scénarios de celui-ci étant décrit par un ou plusieurs diagrammes dynamiques : diagrammes d'activités, de séquence, diagrammes de communication ou d'états-transitions.

3.3.2 La vue logique

La vue logique constitue la principale description architecturale d'un système informatique et beaucoup de petits projets se contentent de cette seule vue. Cette vue décrit, de façon statique et dynamique, le système en termes d'objets et de classes. La vue logique permet d'identifier les différents éléments et mécanismes du système à réaliser. Elle permet de décomposer le système en abstractions et constitue le cœur de la réutilisation. En effet, l'architecte informatique récupérera un maximum de composants des différentes bibliothèques et cadriciels (framework) à sa disposition. Une recherche active de composants libres et/ou commerciaux pourra également être envisagée.

La vue logique est représentée, principalement, par des diagrammes statiques de classes et d'objets enrichis de descriptions dynamiques : diagrammes d'activités, de séquence, diagrammes de communication ou d'états-transitions.

3.3.3 La vue des processus

La vue des processus décrit les interactions entre les différents processus, threads (fils d'exécution) ou tâches, elle permet également d'exprimer la synchronisation et l'allocation des

objets. Cette vue permet avant tout de vérifier le respect des contraintes de fiabilité, d'efficacité et de performances des systèmes multitâches.

Les diagrammes utilisés dans la vue des processus sont exclusivement dynamiques : diagrammes d'activités, de séquence, diagrammes de communication ou d'étatstransitions.

3.3.4 La vue de réalisation

La vue de réalisation permet de visualiser l'organisation des composants (bibliothèque dynamique et statique, code source...) dans l'environnement de développement. Elle permet aux développeurs de se retrouver dans le capharnaüm que peut être un projet de développement informatique. Cette vue permet également de gérer la configuration (auteurs, versions...).

Les seuls diagrammes de cette vue sont les diagrammes de composants.

3.3.5 La vue de déploiement

La vue de déploiement représente le système dans son environnement d'exécution. Elle traite des contraintes géographiques (distribution des processeurs dans l'espace), des contraintes de bandes passantes, du temps de réponse et des performances du système ainsi que de la tolérance aux fautes et aux pannes. Cette vue est fort utile pour l'installation et la maintenance régulière du système.

Les diagrammes de cette vue sont les diagrammes de composants et les diagrammes de déploiement.

4-Les styles architecturaux

L'architecture logicielle, tout comme l'architecture traditionnelle, peut se catégoriser en styles[7]. En effet, malgré les millions de systèmes informatiques construits de par le monde au cours des cinquante dernières années, tous se classent parmi un nombre extrêmement restreint de styles architecturaux[8]. De plus, un système informatique peut utiliser plusieurs styles selon le niveau de granularité ou l'aspect du système décrit. Nous ferons remarquer que, comme en architecture traditionnelle, c'est souvent par le mélange d'anciens styles que les nouveaux apparaissent.

4.1 Architecture en appels et retours

L'architecture en appels et retours est basée sur le raffinement graduel proposé[9] par Niklaus Wirth. Cette approche, également appelée décomposition fonctionnelle, consiste à découper une fonctionnalité en sous-fonctionnalités qui sont également divisées en sous-sous-fonctionnalités et ainsi de suite; la devise diviser pour régner est souvent utilisée pour décrire cette démarche.

Si à l'origine cette architecture était fondée sur l'utilisation de fonctions, le passage à une méthode modulaire ou objet est toute naturelle; la fonctionnalité d'un module ou d'un objet est réalisée par des sous-modules ou des sous-objets baptisés travailleurs (worker). Le terme hiérarchie de contrôle est alors utilisé pour décrire l'extension de cette architecture au paradigme modulaire ou objet. Une forme dérivée de cette architecture est l'architecture distribuée où les fonctions, modules ou classes se retrouvent répartis sur un réseau.

4.2 Architecture en couches

La conception de logiciels nécessite de recourir à des bibliothèques. Une bibliothèque très spécialisée utilise des bibliothèques moins spécialisées qui elles-mêmes utilisent des bibliothèques génériques. De plus, comme nous l'avons déjà mentionné, le développement efficace de composants réutilisables nécessite de créer une bibliothèque logicielle ; l'architecture en couches est la conséquence inéluctable d'une telle approche. En effet, les nouveaux composants utilisent les anciens et ainsi de suite, la bibliothèque tend donc à devenir une sorte d'empilement de composants. La division en couches consiste alors à regrouper les composants possédant une grande cohésion (sémantiques semblables) de manière à créer un empilement de paquetages de composants ; tous les composants des couches supérieures dépendants fonctionnellement des composants des couches inférieures.

4.3 Architecture centrée sur les données

Dans l'architecture centrée sur les données, un composant central (SGBD, Datawarehouse, Blackboard) est responsable de la gestion des données (conservation, ajout, retrait, mise à jour, synchronisation, ...) . Les composants périphériques, baptisés clients, utilisent le composant central, baptisé serveur de données, qui se comporte, en général, de façon passive (SGBD, Datawarehouse). Un serveur passif ne fait qu'obéir aveuglément aux ordres alors qu'un serveur actif (Blackboard) peut notifier un client si un changement aux données qui le concerne se produit.

Cette architecture sépare clairement les données (serveurs) des traitements et de la présentation (clients) et permet ainsi une très grande intégrabilité, en effet, des clients peuvent être ajoutés sans affecter les autres clients. Par contre, tous les clients sont dépendants de l'architecture des données qui doit rester stable et qui est donc peu extensible. Ce style nécessite donc un investissement très important dans l'architecture des données. Les datawarehouses et les bases de données fédérées sont des extensions de cette architecture.

4.4 Architecture en flot de données

L'architecture en flot de données est composée de plusieurs composants logiciels reliés entre eux par des flux de données. L'information circule dans le réseau et est transformée par les différents composants qu'elle traverse. Lorsque les composants se distribuent sur une seule ligne et qu'ils ne font que passer l'information transformée à leur voisin, on parle alors d'architecture par lot (batch). Si les composants sont répartis sur un réseau informatique et qu'ils réalisent des transformations et des synthèses intelligentes de l'information, on parle alors d'architecture de médiation. Les architectures orientées évènements font également partie de cette catégorie.

4.5 Architecture orientée objets

Les composants du système (objets) intègrent des données et les opérations de traitement de ces données. La communication et la coordination entre les objets sont réalisées par un mécanisme de passage de messages. L'architecture orientée objets est souvent décrite par les trois piliers : encapsulation, héritage et polymorphisme. L'encapsulation concerne l'architecture détaillée de chaque objet, les données étant protégées d'accès direct par une couche d'interface. De plus, les sous-fonctions, inutiles pour utiliser l'objet, sont masquées à l'utilisateur de l'objet. L'héritage permet d'éviter la redondance de code et facilite l'extensibilité du logiciel, les fonctionnalités communes à plusieurs classes d'objets étant regroupées dans un ancêtre commun. Le polymorphisme permet d'utiliser des objets différents (possédant des

comportements distincts) de manière identique, cette possibilité est réalisée par la définition d'interfaces à implémenter (classes abstraites).

4.6 Architecture orientée agents

L'architecture orientée agents correspond à un paradigme où l'objet, de composant passif, devient un composant projectif :

En effet, dans la conception objet, l'objet est essentiellement un composant passif, offrant des services, et utilisant d'autres objets pour réaliser ses fonctionnalités; l'architecture objet n'est donc qu'une extension de l'architecture en appels et retours, le programme peut être écrit de manière à demeurer déterministe et prédictible.

L'agent logiciel, par contre, utilise de manière relativement autonome, avec une capacité d'exécution propre, les autres agents pour réaliser ses objectifs : il établit des dialogues avec les autres agents, il négocie et échange de l'information, décide à chaque instant avec quels agents communiquer en fonction de ses besoins immédiats et des disponibilités des autres agents.

Quelques références bibliographiques de la partie 1 :

1. ↑ Pressman R. S., Software Engineering: A Practitioner's Approach, Third Édition. McGraw-Hill. Chapitre 4, p. 107, 1992.
2. ↑ Yourdon E., Software Reuse. Application Development Strategies. vol. 1, n0. 6, p. 28-33, juin 1994.
3. ↑ David Garlan, Robert Allen, John Ockerbloom, *Architectural Mismatch: Why Reuse Is So Hard*, IEEE Software, Nov./Dec. 1995
4. ↑ Perry D.E, Wolf A.L., Foundation for the study of Software Architecture. ACM Software Eng. Notes, p. 40-50, octobre 1992
5. ↑ Philippe B. Kruchten, *The 4+1 View Model of Architecture* [archive], IEEE Software, novembre 1995.
6. ↑ Jacobson I., Booch G., Rumbaugh J., The Unified Software Development Process, (ISBN 0-201-57169-2)
7. ↑ Bass L., Clement P., Kazman R., Software Architecture in Practice, Addison-Wesley, 1998
8. ↑ David Garlan et Mary Shaw, *An Introduction to Software Architecture* [archive], CMU-CS-94-166, School of Computer Science, Carnegie Mellon University, janvier 1994
9. ↑ Wirth N., Program Development by Stepwise Refinement, CACM, vol. 14, no. 4, 1971, pp. 221-227

Partie 2 : Les patrons de conception

Sommaire

1- Introduction

Un patron de conception (plus connu sous le terme anglais « *Design pattern* ») est une solution générique permettant de résoudre un problème spécifique.

En général, un patron de conception décrit une structure de classes utilisant des interfaces, et s'applique donc à des développements logiciels utilisant la programmation orientée objet.

Cependant, les patrons de conception sont généralement utiles pour les applications ayant une taille importante et/ou dans les projets où plusieurs applications différentes interagissent entre elles (via un moyen de communication).

2- Classification des patrons de conception

Il y a différentes façons de classer les différents patrons de conception. Cette partie présente les plus connus, classés en fonction de leurs auteurs, puis, si possible, classés par catégorie fonctionnelle.

2.1 Ensemble de patrons de conception

Il y a différents ensembles de patrons de conception, créés par différents auteurs.

- Les plus connus sont ceux du « **Gang of Four** » (ou **GoF** : Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) décrits dans leur livre « Design Patterns -- Elements of Reusable Object-Oriented Software » https://sophia.javeriana.edu.co/~cbustaca/docencia/DSBP-2016-03/recursos/Erich%20Gamma,%20Richard%20Helm,%20Ralph%20Johnson,%20John%20M.%20Vlissides-Design%20Patterns_%20Elements%20of%20Reusable%20ObjectOriented%20Software%20%20-Addison-Wesley%20Professional%20(1994).pdf en 1995. Les patrons de conception tirent leur origine des travaux de l'architecte Christopher Alexander dans les années 70.
- **Les patrons GRASP** sont des patrons créés par Craig Larman qui décrivent des règles pour affecter les responsabilités aux classes d'un programme orienté objets pendant la conception, en liaison avec la méthode de conception BCE (pour « Boundary Control Entity » - en français MVC « Modèle Vue Contrôleur »).
- **Les patrons d'entreprise** (*Enterprise Design Pattern*) créés par Martin Fowler, décrivent des solutions à des problèmes courants dans les applications professionnelles. Par exemple, des patrons de couplage entre un modèle objet et une base de donnée relationnelle.
- **D'autres patrons** créés par divers auteurs existent et décrivent des solutions à des problèmes différents de ceux vus précédemment.

2.2 Catégorie fonctionnelle

Les patrons de conception peuvent être classés en fonction du type de problème qu'ils permettent de résoudre. Par exemple, les patrons de conception de création résolvent les problèmes liés à la création d'objets.

3- Les patrons de GoF

3.1 Introduction:

Les patrons de conception créés par le « Gang of Four » (GoF en abrégé) sont décrits dans leur livre « Design Patterns -- Elements of Reusable Object-Oriented Software ». Les 4 auteurs du livre (Erich Gamma, Richard Helm, Ralph Johnson et John Vlissides) sont surnommés la bande des quatre (« Gang of Four » en anglais).

Ces patrons de conception sont classés en trois catégories :

- **Les patrons de création** décrivent comment régler les problèmes d'instanciation de classes, c'est à dire de création et de configuration d'objets (objet en unique exemplaire par exemple).
- **Les patrons de structure** décrivent comment structurer les classes afin d'avoir le minimum de dépendance entre l'implémentation et l'utilisation dans différents cas.
- **Les patrons de comportement** décrivent une structure de classes pour le comportement de l'application (répondre à un évènement par exemple).

3.2 Les patrons de création :

Un patron de création permet de résoudre les problèmes liés à la création et la configuration d'objets.

Par exemple, une classe nommée `RessourcesApplication` gérant toutes les ressources de l'application ne doit être instanciée qu'une seule et unique fois. Il faut donc empêcher la création intentionnelle ou accidentelle d'une autre instance de la classe. Ce type de problème est résolu par le patron de conception "Singleton".

Les différents patrons de création sont les suivants :

Singleton
: Il est utilisé quand une classe ne peut être instanciée qu'une seule fois.

Prototype
: Plutôt que de créer un objet de A à Z c'est à dire en appelant un constructeur, puis en configurant la valeur de ses attributs, ce patron permet de créer un nouvel objet par recopie d'un objet existant.

Fabrique
: Ce patron permet la création d'un objet dont la classe dépend des paramètres de construction (un nom de classe par exemple).

Fabrique abstraite
: Ce patron permet de gérer différentes fabriques concrètes à travers l'interface d'une fabrique abstraite.

Monteur
: Ce patron permet la construction d'objets complexes en construisant chacune de ses parties sans dépendre de la représentation concrète de celles-ci.

3.3 Les patrons de structure:

Un patron de structure permet de résoudre les problèmes liés à la structuration des classes et leur interface en particulier.

Les différents patrons de structure sont les suivants :

Pont
: Utilisation d'interface à la place d'implémentation spécifique pour permettre l'indépendance entre l'utilisation et l'implémentation.

Façade
: Ce patron de conception permet de simplifier l'utilisation d'une interface complexe.

Adaptateur
: Ce patron permet d'adapter une interface existante à une autre interface.

Objet composite
: Ce patron permet de manipuler des objets composites à travers la même interface que les éléments dont ils sont constitués.

Proxy
: Ce patron permet de substituer une classe à une autre en utilisant la même interface afin de contrôler l'accès à la classe (contrôle de sécurité ou appel de méthodes à distance).

Poids-mouche
: Ce patron permet de diminuer le nombre de classes créées en regroupant les classes similaires en une seule et en passant les paramètres supplémentaires aux méthodes appelées.

Décorateur
: Ce patron permet d'attacher dynamiquement de nouvelles responsabilités à un objet.

3.4 Les patrons de comportement :

Un patron de comportement permet de résoudre les problèmes liés aux comportements, à l'interaction entre les classes.

Les différents patrons de comportement sont les suivants :

Chaîne de responsabilité
: Permet de construire une chaîne de traitement d'une même requête.

Commande
: Encapsule l'invocation d'une commande.

Interpréteur
: Interpréter un langage spécialisé.

Itérateur
: Parcourir un ensemble d'objets à l'aide d'un objet de contexte (curseur).

Médiateur
: Réduire les dépendances entre un groupe de classes en utilisant une classe Médiateur comme intermédiaire de communication.

Mémento
: Mémoriser l'état d'un objet pour pouvoir le restaurer ensuite.

Observateur
: Intercepter un évènement pour le traiter.

État
: Gérer différents états à l'aide de différentes classes.

Stratégie
: Changer dynamiquement de stratégie (algorithme) selon le contexte.

Patron de méthode
Définir un modèle de méthode en utilisant des méthodes abstraites.

Visiteur
Découpler classes et traitements, afin de pouvoir ajouter de nouveaux traitements sans ajouter de nouvelles méthodes aux classes existantes.

4- Les patrons GRASP

GRASP signifie ***G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns/**P**rinciples*.

Ces patrons de conception donnent des conseils généraux sur l'assignation de responsabilité aux classes et objets dans une application. Ils sont issus du bon sens de conception, intuitifs et s'appliquent de manière plus générale.

Une responsabilité est vue au sens conception (exemples : création, détention de l'information, etc.) :

- elle est relative aux méthodes et données des classes, • elle est assurée à l'aide d'une ou plusieurs méthodes,
- elle peut s'étendre sur plusieurs classes.

En UML, l'assignation de responsabilité peut être appliquée à la conception des diagrammes de collaboration.

Les patrons de conception GRASP sont les suivants :

Expert en information
Affecter les responsabilités aux classes détenant les informations nécessaires.

Créateur
Déterminer quelle classe a la responsabilité de créer des instances d'une autre classe.

Faible couplage
Diminuer le couplage des classes afin de réduire leurs interdépendances dans le but de faciliter la maintenance du code.

Forte cohésion
Avoir des sous-classes terminales très spécialisées.

Contrôleur
Affecter la responsabilité de réception et traitement de messages systèmes.

Polymorphisme
Affecter un nouveau comportement à l'endroit de la hiérarchie de classes où il change.

Fabrication pure
Créer des classes séparées pour des fonctionnalités génériques qui n'ont aucun rapport avec les classes du domaine applicatif.

Indirection
Découpler des classes en utilisant une classe intermédiaire.

Protection Ne pas communiquer avec des classes inconnues.

5- Les patrons d’entreprise

Les patrons de conception d'entreprise répondent aux problèmes d'architecture des applications d'entreprise (base de données, service web, ...) et sont décrits dans le livre « Patterns of Enterprise Application Architecture » écrit par Martin Fowler.

Les patrons d'entreprises sont nombreux et sont catégorisés de la façon suivante :

- Logique du domaine (*Domain Logic Patterns*)
- Architecture de source de données (*Data Source Architectural Patterns*)
- Comportement objet-relationnel (*Object-Relational Behavioral Patterns*)
- Structure objet-relationnel (*Object-Relational Structural Patterns*)
- Association méta-données objet-relationnel (*Object-Relational Metadata Mapping Patterns*)
- Présentation web (*Web Presentation Patterns*)
- Distribution (*Distribution Patterns*)
- Concurrence locale (hors-ligne) (*Offline Concurrency Patterns*)
- État de session (*Session State Patterns*)
- Patrons de base (*Base Patterns*)

6- Autres patrons de conception

D'autres patrons de conception que ceux vus précédemment existent. En voici quelques uns :

Type fantôme
: Utiliser un type pour ajouter une contrainte à la compilation.

Double-dispatch
: Permettre l'appel à une méthode surchargée en recourant au type dynamique d'un argument.

Post-Redirect-Get
: Éviter la soumission multiple d'un formulaire web lors d'un rafraichissement.

Map-Reduce
: Parallélisation d'un traitement sur des données volumineuses.

Évaluation retardée
: Retarder l'évaluation d'une fonction ou expression jusqu'à utilisation concrète du résultat.

Copie sur modification
: Retarder la création d'une copie privée d'une structure tant qu'elle n'est pas modifiée.

Injection de dépendance
: Ce patron de conception est utilisé pour le couplage dynamique.

Inversion de contrôle
: Ce patron de conception est utilisé pour réduire la dépendance à une séquence d'exécution particulière.

Modèle-Vue-Présentateur
: Ce patron de conception est dérivé du patron Modèle-Vue-Contrôleur.

Écart de génération
: Ce patron de conception est utilisé pour séparer une classe générée automatiquement et la partie personnalisation du code.

Objet nul
: Utiliser un objet nul dont les méthodes ne font rien au lieu d'utiliser une référence nulle.

Quelques références bibliographiques sur les patrons de conception:

- français *Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides* (traduction : *JeanMarie Lasvergères*) - **Design Patterns** : *Catalogue de modèles de conceptions réutilisables* Éditions Vuibert - 1999 - *490 pages* - (ISBN 2711786447)
- français *Craig Larman* - **UML 2 et les Design Patterns** *(3ème édition)* - (ISBN 2744070904)
- français *Eric Freeman, Elisabeth Freeman, Kathy Sierra, Bert Bates* - **Design patterns** : *Tête la première (1ère édition)* - 2005 - (ISBN 2841773507)

- anglais *Christopher Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, S. Angel* - **A Pattern Language : Towns, Buildings, Construction** - 1977 - (ISBN 0195019199)
- anglais *Martin Fowler* - **Patterns of Enterprise Application Architecture** *(6ème édition)* Éditions Addison-Wesley - 2004 - (ISBN 0321127420)
- anglais *James O. Coplien, Douglas C. Schmidt* - **Pattern Languages of Program Design** - 1995 - (ISBN 0201607344)
- anglais *Craig Larman* - **Applying UML and Patterns** : *An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd ed.)* - Éditions Prentice Hall PTR - 2005 - (ISBN 0-13-148906-2)