

Les composants logiciels

Noël Plouzeau

Mots-clés : composants, réutilisation, contrats.

Résumé : la conception par composants logiciels est une technique efficace de construction de logiciels, ayant pour but d'améliorer la réutilisation des architectures, d'assurer une bonne qualité par la réutilisation intensive, la réduction des coûts par la création d'un marché de composants éprouvés et bien maintenus. Ce chapitre présente les caractéristiques des composants logiciels, les techniques élémentaires permettant de les construire et leur place dans les processus de construction de logiciel.

Fabriquer facilement des logiciels efficaces et fiables est l'un des grands rêves de l'industrie du logiciel. Il s'agit d'atteindre enfin l'étape de la production en série, comme d'autres industries l'ont fait depuis des décennies. Le concept de composant logiciel est pressenti depuis longtemps comme un élément important de cette « révolution industrielle ». Cependant, il n'existe à ce jour aucun consensus établi sur les notions essentielles des composants logiciels, et sur la manière de les intégrer au processus de construction de logiciel.

Ce chapitre présentera certaines idées importantes de la notion de composant logiciel, en choisissant un point de vue plutôt général et détaché des aspects purement technologiques. Les concepts présentés gravitent autour de la définition donnée par Clemens Szyperski (Szyperski, 2002) :

« Un composant logiciel est une unité de composition dotée exclusivement d'interfaces spécifiées par des contrats et de dépendances contextuelles explicites. Un composant logiciel peut être déployé indépendamment et peut être composé par des constructeurs tiers. »

1. Vers un modèle économique du logiciel

Pour pouvoir analyser les difficultés et les solutions de l'industrialisation du logiciel, il faut préciser les modèles de production existants ou à atteindre. L'acquéreur d'un logiciel recherche un produit capable de fournir des résultats bien définis, dans un environnement d'emploi lui aussi bien défini et avec

des contraintes de qualité et de coût d'usage. Le producteur de logiciel cherche à fabriquer des produits adaptés au marché, avec le meilleur rapport qualité/coût de production possible. Un acteur intermédiaire (par exemple une société de services en informatique) peut réaliser l'adaptation d'un produit aux besoins d'un acquéreur. La tâche de cet acteur intermédiaire est difficile, car il a à sa charge la réalisation effective de l'adaptation aux besoins pour chacun de ses clients, pour toute la durée de vie du produit final. Plusieurs procédés de réalisation de cette adaptation sont envisageables.

1.1. Conception à grain fin

L'acteur intermédiaire crée le logiciel de toutes pièces, en n'employant que des services élémentaires (par exemple des requêtes de base de données) et en codant le reste de l'architecture. Cette façon de faire a un certain nombre d'avantages :

- le produit peut être adapté de façon très fine aux besoins du client ;
- les techniques de réalisation sont éprouvées.

Le procédé a aussi des inconvénients majeurs :

- le coût est élevé, comme pour tout artefact fait sur mesure, car il faut des développeurs spécialisés ;
- à l'intérieur même du logiciel, des problèmes analogues risquent d'être résolus de façons différentes, en entraînant une redondance coûteuse ;
- l'hyperspécialisation de la solution la rend peu apte à supporter des évolutions technologiques

(imposées par le marché) ou de fonction (imposées par le client).

1.2. Conception à gros grain

L'adaptateur a recours dans ce cas à un produit existant et configurable. L'objectif est de le spécialiser aux besoins du client. Bien évidemment, le produit doit être conçu dès l'origine pour offrir des possibilités de configuration. Plusieurs mécanismes existent pour mettre en œuvre cette spécialisation :

- des langages spécialisés (langages de script) permettent de coder des fonctions supplémentaires qui sont exécutées automatiquement dans certaines situations (programmation à événements) ;
- le logiciel configurable se présente comme un *framework*, c'est-à-dire une architecture « à trous » où certains fragments doivent être complétés par du code écrits dans un langage de programmation conventionnel (en général à objets).

Ce procédé offre les avantages suivants :

- masquage de bon nombre de problèmes technologiques, qui sont à la charge du fabricant du logiciel configurable ;
- amélioration de la fiabilité du produit : le logiciel configurable étant utilisé par de nombreux clients, les bogues sont rapidement découverts et corrigés.

Citons les inconvénients suivants :

- un seul logiciel même très configurable ne suffit pas à couvrir les besoins de l'utilisateur, il faudra donc recourir à plusieurs logiciels ;
- relative rigidité de la solution obtenue, en termes de ressources nécessaires (souvent plus importantes qu'une solution à grain fin) ;
- contraintes de langages : les développeurs doivent maîtriser les langages imposés par le fabricant du logiciel.

1.3. Conception à grain moyen

La conception à grain moyen repose sur l'emploi de gros fragments de logiciels, spécialisés, hautement configurables et interopérables, disponibles sur catalogue, rigoureusement testés et n'imposant aucun langage de développement à leurs utilisateurs. Il s'agit donc de réunir le meilleur des deux approches précédentes, la conception à grain fin et la conception à gros grain. Les fragments évoqués ici ne sont ni plus ni moins que des composants logiciels.

1.4. Des systèmes de plus en plus hétérogènes et évolutifs

Dire que le logiciel est devenu omniprésent en termes de lieux, formes et services est devenu une banalité. Pour illustrer le propos de la complexité et de l'hétérogénéité, citons deux domaines bien connus : les services en ligne et les systèmes enfouis. L'utilisateur de services en ligne veut pouvoir accéder dans une même page interactive à un ensemble de services intégrés. Par exemple, l'accès à un site de banque doit rassembler de façon homogène des services d'accès aux comptes, des informations boursières, des moyens de suivi personnalisé du client. Or ces services sont en général réalisés par des applications logicielles distinctes, dans des organismes distincts qui emploient chacun des méthodes propres pour le stockage et l'accès aux données. Il est aisé de comprendre que la construction de l'application bancaire intégrant tous ces services n'est pas chose facile, la question des standards d'interaction entre systèmes différents constituant un problème complexe. Le problème est rendu encore plus délicat par l'évolution continue des techniques et des standards. Dans un tout autre domaine d'application, les constructeurs d'automobiles sont également confrontés à de délicats problèmes d'hétérogénéité. Une voiture actuelle comporte de nombreux éléments dotés de calculateurs (freins, contrôle moteur, contrôle de trajectoire, transmission, etc). Pour réduire les coûts, les différents éléments sont conçus non pour une marque ou un modèle spécifique mais pour une gamme de services. Les concepteurs d'un modèle de voiture ont pour tâche de choisir, adapter, configurer de nombreux sous-systèmes. Les contraintes de performances (temps de réponse, par exemple), de fiabilité (résistance aux défaillances) et d'évolutivité (adaptation aux contraintes fluctuantes du marché) sont importantes.

1.5. Les composants comme concept structurant

La construction d'un logiciel par composants cherche à instaurer un juste milieu entre les deux cas extrêmes de *sur mesure* et de *prêt-à-porter*. Un logiciel conçu par composants contient principalement un ensemble de modules interconnectés (appelés composants). Les caractéristiques d'un tel assemblage seraient les suivantes, dans le cas idéal :

- chaque composant rend un service bien spécifique, dont tous les aspects sont totalement décrits, y compris les dépendances envers des services fournis par son environnement ;

- il n'est pas nécessaire de disposer du code source d'un composant pour pouvoir l'utiliser totalement et correctement ;
- la mise en œuvre de chaque composant est robuste, configurable et adaptée à l'évolution des services, des pratiques industrielles et des technologies qui les réalisent ;
- le coût d'un composant est relativement faible car vendu à de nombreux exemplaires ;
- les composants sont capables de coopérer d'une façon bien organisée et il est facile de les assembler ;
- les propriétés d'un assemblage de composants peuvent être déduites des propriétés de chacun des composants et de leur mode d'interaction.

2. Propriétés générales des composants logiciels

2.1. Séparation forte entre utilisation et construction

Un composant logiciel est une unité modulaire de production de services, avec une séparation complète entre la définition des services rendus et la façon dont ils sont mis en œuvre. Les utilisateurs d'un composant (personnes chargées de construire une application à composants) et constructeurs de composants (personnes chargées de réaliser un composant) mènent des activités très différentes. Un composant est une « boîte noire » très bien étiquetée mais dont les détails internes de réalisation doivent rester inconnus de l'utilisateur. Le fait de masquer des détails de mise en œuvre n'est pas en soi la marque exclusive de la conception par composants logiciels. Par exemple, la conception par objets s'appuie fortement sur ce principe pour la réalisation des patrons de conception (*cf.* le chapitre *Patrons de conception*). Le concept de composant logiciel va plus loin, en insistant sur l'indépendance entre la réalisation interne et les conditions d'exécution, c'est-à-dire sur l'environnement du composant. Autrement dit, un composant logiciel est fabriqué dans l'ignorance totale des architectures qui l'utiliseront. En effet, un composant logiciel qui fait trop d'hypothèses sur son environnement devient trop spécialisé et donc probablement non rentable. Pour résumer, un composant logiciel digne de ce nom :

- peut être utilisé efficacement en ignorant tout de sa structure interne ;

- peut être conçu et codé en ignorant tout de l'architecture environnante à l'exécution.
- Grâce à cette séparation, les concepteurs d'applications à composants et les concepteurs de composants ont des métiers bien distincts.

2.2. Relations client/fournisseur

La spécification des services offerts et requis par un composant est l'un des points vitaux de la conception par composants logiciels. En effet, un concepteur d'application à composants ne peut s'appuyer que sur la description des services pour « connaître » un composant. Cette connaissance est une condition indispensable pour pouvoir configurer le composant, le connecter et demander l'exécution de service.

La capacité pour un composant logiciel de décrire toutes ses caractéristiques constitue le service primordial (au sens propre) que les composants doivent offrir. Ces caractéristiques de fonctionnement se répartissent en plusieurs catégories :

- les services fournis par le composant logiciel, et les moyens d'y accéder ;
- les services requis par le composant logiciel (qui doivent être fournis par son environnement, c'est-à-dire les autres composants et l'environnement d'exécution) ;
- les paramètres de configuration.

2.2.1. Notion de contrat de service

Il est fréquent que dans la vie courante un client et un fournisseur se mettent d'accord sur les termes de la réalisation d'un service. Cet accord s'obtient par la rédaction et l'acceptation commune d'un contrat. Ce contrat définit les contributions de chacun en vue de la réalisation d'un service. Le client s'engage à fournir certaines ressources (financières, matérielles). Le fournisseur s'engage à fabriquer certains produits (documents, produits manufacturés, etc), à condition que le client remplisse sa part du contrat.

Ce modèle de contrat est de plus en plus souvent employé dans la conception de logiciel en général (Meyer, 1992), et de logiciel à composants en particulier. Le contrat est établi entre les éléments de logiciel « client » d'un composant et le composant agissant comme « fournisseur ». Comme pour les contrats de la vie courante, un contrat de service comporte une partie *obligations du client* et une partie *obligations du fournisseur*. La réalisation d'un service n'est correcte que si ces deux parties sont satisfaites lors de l'exécution du service.

Les obligations du client portent en général sur la validité des données transmises au composant four-

naisseur lors de la demande d'exécution de service. Par exemple, le fournisseur peut exiger que les données transmises soient comprises entre certaines valeurs limites (bornes).

Les obligations du client peuvent aussi porter sur l'exécution préalable d'autres requêtes auprès du même composant fournisseur ou bien d'autres parties et composants du système. Ce concept existe aussi dans la vie de tous les jours, par exemple lorsque certains documents (par exemple des déclarations ou des autorisations) doivent être obtenus avant de pouvoir demander un certain service.

Les obligations du fournisseur porte sur l'effet de l'exécution du service : production de données, validité de ces données, actions faites sur les composants logiciels environnants, délais de réalisation du service, débit minimum à atteindre (par exemple dans le cas de flux multimédia), etc.

Le non-respect d'un contrat durant l'exécution d'un service est le signe d'un dysfonctionnement. Il est fréquent que l'architecture de composant comporte un mécanisme de surveillance, qui permet de détecter ce problème de non-respect des obligations lors de l'exécution. Si une des obligations du client n'a pas été respectée alors le problème vient de la mise en œuvre de la partie client. Si une des obligations du fournisseur n'a pas été respectée, l'erreur provient de la mise en œuvre du composant fournisseur, à la condition que toutes les obligations du client aient été respectées.

2.2.2. *Catégories de contrats*

Il existe toutes sortes de contrats de service. Dans le cas général, un contrat comportera des clauses sur des propriétés de natures différentes, telles que :

- le type des paramètres et données échangés ;
- les conditions booléennes (prédicats) qui doivent être vraies pour que le service soit considéré comme valable (en d'autres termes les préconditions et postconditions) ;
- les conditions de coordination de l'exécution du service par rapport à d'autres exécutions du service (antériorité, exclusion mutuelle de demandes, etc);
- les conditions quantitatives de l'exécution du service : délai maximum pour exécuter le service, débit minimum du réseau, etc. (Beugnard, 1999).

2.2.3. *Connexions et ports*

La partie cliente communique avec la partie fournisseur au moyen de points d'accès au service. Un point d'accès propose un ou plusieurs moyens d'interactions, par exemple des opérations (au sens

logiciel) que le client peut appeler, des ports permettant l'accès à des flux de donnée entrants ou sortants, des mécanismes d'abonnement à des signaux (comme par exemple des alarmes, des notifications de modification de l'état du système). Bien que la notion de contrat de service soit un concept important dans les architectures à composants, elle n'est pas spécifique de celle-ci. Cette notion de contrat figure en bonne place dans la conception par objets (voir par exemple la conception par contrats (Meyer, 1992)). La relation entre l'objet client (qui appelle une opération) et l'objet fournisseur (qui exécute l'opération appelée) est asymétrique : l'objet fournisseur ne connaît pas l'objet client. Le contrat qui lie les deux objets n'a de signification que durant l'exécution de l'opération. Cette volatilité du contrat est insuffisante pour garantir un fonctionnement correct du modèle de conception par composants, c'est pourquoi bon nombre d'architectures à composants proposent des mécanismes d'établissement d'une relation durable et symétrique entre client et fournisseur.

Certaines architectures logicielles s'appuient sur les concepts de ports et de connexions pour gérer ces relations entre composants. Dans ces architectures, un composant expose des ports de communication (un peu à la manière des ports de communication physique d'un réseau). Un port peut être soit un port d'accès à des services du composant (services fournis) soit un port permettant au composant d'accéder à des services externes nécessaires à son bon fonctionnement (services requis). Pour qu'un composant A puisse fonctionner, il faut connecter chaque port de service requis à des ports de services fournis d'autres composants B, C, etc. Lorsque le composant A réalisera un service, il pourra à son tour faire appel à des services, au moyen des ports de service requis. Les requêtes de service que fera le composant A seront alors transmises via la connexion au port de service fourni d'un autre composant.

L'emploi d'un mécanisme de ports permet de déterminer si un assemblage est valide : les besoins d'un composant (qui sont explicites) doivent être satisfaits par d'autres composants. Ceci prévient la construction de mauvaises configurations, dans lesquelles certains composants ne peuvent fonctionner pour cause de besoins non résolus.

De plus, l'emploi d'un mécanisme de port permet la coopération de composants tout en empêchant que les composants « se connaissent » directement (ce qui serait contraire aux principes de la conception par composants). Les ports jouent le rôle de relais dans la relation intercomposant.

Il faut remarquer que les termes de *client* et *fournisseur* utilisés plus haut désignent des rôles joués par deux composants dans le contexte de l'utilisation d'un service. Il est parfaitement possible (et même très fréquent) qu'un composant joue le rôle de client pour un certain service et le rôle de fournisseur pour un second service. La réalisation de ce second service fera par exemple appel au premier service. La figure 1 donne un exemple d'assemblage pour un système d'antiblocage de frein :

- un composant *Capteur roue* englobe tout le système mécanique et électronique qui observe la rotation de la roue ;
- un composant *Détecteur blocage* reçoit un flux de valeurs de vitesse et prend des décisions de commande du frein ;
- le composant *Frein* englobe tout le système mécanique, avec un système d'interface logicielle.

Ces trois composants peuvent parfaitement être développés et fournis par trois sociétés différentes. Leurs contrats de service sont définis par les interfaces *Configuration*, *Flux de vitesse* et *Commande frein*.

Figure 1. Exemple d'assemblage de composants

2.3. Description des dépendances

Un composant logiciel s'appuie sur un certain nombre de ressources pour réaliser ses services. Certaines de ces ressources sont les services fournis par d'autres composants logiciels, accessibles *via* le mécanisme de ports et de connexions décrit plus haut. D'autres types de ressources sont requises pour l'exécution correcte du composant : disponibilité et puissance d'une unité de traitement (processeur), espace mémoire, mécanismes de protection mémoire, etc.

Bien qu'un composant logiciel soit une boîte noire du point de vue de sa structure interne, il est vital que l'ensemble de ces dépendances soit explicitement documenté pour qu'il puisse être utilisé correctement. Ces dépendances peuvent même être décrites dans une notation analysable par les outils de développement de logiciel. Un certain nombre de calculs de validité peuvent alors être effectués. L'outil d'aide au développement de logiciel à composants peut vérifier :

- que chaque composant utilisé est correctement connecté à des composants qui lui fournissent les services qu'il requiert ;

- que les paramètres d'environnement sont compatibles avec la configuration choisie pour chaque composant.

L'outil peut également aider les concepteurs lors du choix de composants fournisseurs de service, à partir d'une bibliothèque de tels composants.

Si la notation de dépendance des contrats de service (cf. §2.2.1) inclut la définition de paramètres quantitatifs (tels que par exemple le temps moyen mis par le composant pour rendre un certain service), l'outil d'aide au développement est capable de calculer les paramètres quantitatifs pour un assemblage de composants, que ce soit pour les services fournis par l'assemblage ou pour les services requis par l'assemblage (quantification des besoins).

2.4. Composition

Les paragraphes précédents ont mentionné les différents attributs propres aux architectures à composants : modules développés isolément, opacité totale de la mise en œuvre, spécification explicite des services fournis, des services requis et de la relation entre eux. Ces différents attributs de composants ne prennent tout leur sens que lorsque les composants sont assemblés, ou plutôt *composés* les uns avec les autres.

Les travaux sur les modèles de composant logiciel cherchent souvent à obtenir le résultat suivant : les propriétés de la composition d'un ensemble de composants peuvent être obtenues par un calcul composant les propriétés de chaque composant de l'ensemble. Autrement dit, si chaque composant d'un assemblage déclare un certain nombre de propriétés (par exemple les services offerts, les services requis, les contrats de qualité de service) alors on doit pouvoir déduire automatiquement les propriétés de l'ensemble (toujours en terme de services offerts, requis et contrats). Dans le cas où l'on tente de connecter deux composants qui sont incompatibles, le résultat du calcul des propriétés de l'assemblage doit montrer qu'il ne fonctionne pas.

Pouvoir calculer les propriétés d'un assemblage est un atout important pour l'évaluation de la qualité d'un logiciel bâti sur cet assemblage, mais à l'heure actuelle de nombreuses difficultés théoriques et pratiques non encore résolues empêchent l'application réelle de cette technique.

2.4.1. *Compatibilité élémentaire entre service offert et service requis*

La compatibilité élémentaire concerne la connexion de deux composants, l'un offrant un service et l'autre le requérant. Cette interconnexion sera valide si et

seulement si le contrat de service fourni recouvre au moins les exigences du contrat du service requis. En pratique, il faut effectuer un calcul de compatibilité entre les descriptions de contrats, et cela suppose donc l'existence de techniques de calcul de comparaison. Dans beaucoup de mises en œuvre pratiques de composants logiciels, les techniques de compatibilité se limitent à la compatibilité de types, selon tel ou tel système de types (en utilisant par exemple les notions d'héritage et de sous-typage). Par exemple, un composant C1 peut requérir l'accès à une interface de type A (comportant une liste d'opérations). Un composant C2 fournit une interface de type B. Si B est un sous-type de A alors C2 est capable de satisfaire les besoins de C1, et les services offerts et requis sont compatibles : l'assemblage de C2 et C1 est valide.

Il est à noter que le système de type de l'architecture à composants n'est normalement pas celui du langage de programmation utilisé pour le codage (en vertu du principe d'indépendance des composants vis-à-vis du langage de programmation).

Plus généralement, le contrôle de la compatibilité entre contrat requis et contrat offert concerne les catégories de propriétés définies au §2.2.2. Il n'est alors pas possible de prouver dans le cas général qu'un contrat de service offert implique la satisfaction d'un contrat de service requis.

2.4.2. *Assemblage dynamique et substituabilité*

Un assemblage de composants peut être défini de façon statique, c'est-à-dire préétabli avant l'exécution de l'application, ou comporter des parties variables à l'exécution. Un cas bien connu est le chargement dynamique de modules d'extension dans une application de navigation Web. La compatibilité du nouveau composant au sein de l'architecture devra alors être contrôlée dynamiquement, au moment de la découverte de ce composant. Plus généralement, la compatibilité d'un composant pourra être évaluée à chacune des phases du cycle de vie du composant (*cf.* §3.5).

3. Réalisation des composants

Nous avons vu jusqu'ici les composants logiciels comme des unités de fourniture de service, connues et contrôlées grâce à la spécification des services, sans aucune connaissance des détails de réalisation. Nous avons adopté en quelque sorte le point de vue du concepteur *par* composants. Nous traitons ci-après des techniques utiles au concepteur de composants.

Du strict point de vue de la conception par composants, aucune technique spécifique n'est imposée. Bien que souvent employée, la conception par objets n'est pas indispensable. Nombre de composants ont été dans le passé conçus en employant le langage de programmation C (par exemple des composants COM). Cependant, la conception par objets apporte de nombreuses techniques bien utiles pour la réalisation d'une mise en œuvre de composant, en laissant cependant nombre de problèmes à résoudre, comme par exemple la résolution des différences entre le modèle de type et d'interaction des composants et celui du langage de programmation.

3.1. Polymorphisme et composants

La conception par objets place la notion de type au centre de ses préoccupations. Les langages à objets permettent de séparer la définition des opérations (l'interface) de la réalisation. Chaque opération peut posséder une ou plusieurs mises en œuvre. Comme pour les composants, l'emploi d'une opération ne requiert pas la connaissance de la mise en œuvre, grâce au principe de substituabilité de type, dit aussi polymorphisme. Ainsi, tout type B déclaré comme mettant en œuvre une interface A peut apporter une mise en œuvre aux opérations de A. Les propriétés du modèle de type des composants ne permettent pas pourtant d'appliquer aveuglément ces propriétés de langage à objets.

En effet, supposons qu'un composant A expose les interfaces I1 et I2, comportant chacune une opération *imprimer* (dont l'exécution produit une copie papier d'un document, par exemple). On a donc deux opérations différentes car venant de deux interfaces différentes. Une manière simple de mettre en œuvre cette opération *imprimer* consiste à déclarer une classe A qui hérite des interfaces I1 et I2 et donc produit une mise en œuvre unique pour l'opération *imprimer*. Cependant, la signification et les contrats associés aux deux opérations imprimées déclarées par le composant n'ont aucune raison d'être identiques ou simplement compatibles. En effet, l'utilisateur du composant A fera explicitement mention de l'interface qui l'intéresse lorsqu'il appellera l'opération *imprimer*. En conséquence, il ne faut en général pas fusionner les deux opérations *imprimer* mais conserver deux mises en œuvre distinctes : l'une pour l'opération venant de I1, l'autre pour celle venant de I2.

3.2. Sous-composants

La mise en œuvre d'un composant peut ne comporter aucun composant interne (on a dans ce cas un com-

posant « primordial ») ou au contraire un ou plusieurs sous-composants. Ses sous-composants coopèrent pour réaliser globalement les services du composant. Bien que l'idée de construire un composant par un tel assemblage soit séduisante sur le papier, la mise en œuvre concrète de cette idée n'est pas simple.

La difficulté principale est qu'il faut disposer d'un moyen de connexion et de configuration, qui spécifie comment l'invocation d'un service au niveau du composant englobant est réalisée concrètement. Ce langage peut être *déclaratif* (c'est-à-dire qu'il définit les règles d'invocation sans donner la façon dont elles sont mises en œuvre concrètement) ou bien *impératif* (c'est-à-dire donnant les instructions qui à l'exécution produiront l'effet escompté). Dans bien des modèles de composants, ce langage est en fait un langage de programmation classique. Lors de son initialisation, un fragment de code source crée les sous-composants, les configure et met en place les mécanismes de gestion des invocations de service. On dénomme parfois ce fragment de code source de la « colle logicielle » (*glueware*).

La demande d'un service sur un port du composant englobant sera réalisée par le déroulement d'une collaboration entre sous-composants. Plusieurs techniques existent :

- la demande est interceptée par un fragment de code source écrit par le développeur ; ce fragment exécute alors un algorithme contenant et coordonnant des appels de services aux sous-composants ;
- la demande est automatiquement transmise à un sous-composant, sans l'intervention d'un fragment de code source de coordination dans le composant englobant.

3.3. Extensibilité

L'extensibilité est la capacité de construire un nouveau composant logiciel à partir d'un composant existant. Deux sortes d'extensibilités existent :

- l'extensibilité interne requiert la connaissance de la structure interne du composant, et l'accès au code source ; elle est donc réservée au concepteur de composant, par exemple pour faciliter la production de nouvelles variantes ou intégrer de nouveaux services ;
- l'extensibilité externe ne requiert aucune connaissance de la structure interne ; elle est accessible à l'utilisateur de composant logiciel, pour ajouter des services propres ou mieux adapter les services du composant.

L'extensibilité interne est fondée sur les mécanismes d'extension du langage de mise en œuvre du composant. Dans le cas d'un langage à objets classique, il s'agira du sous-typage ou de mécanismes d'extension construits sur des patrons de conception (par exemple *Stratégie*, *Adaptateur*). Les avantages et inconvénients de l'extension par sous-typage sont bien connus, citons entre autres :

- les problèmes de covariance et contravariance associés à l'extension des contrats définis par des types ainsi que les contrats définis par des préconditions et des postconditions ;
- les problèmes liés au contrôle de la concurrence d'accès aux services.

L'extensibilité externe s'appuie sur les propriétés de la notation et du modèle de composant logiciel dans lesquels les composants sont spécifiés. Ces modèles incluent souvent des mécanismes d'adaptation qui permettent d'ajouter des traitements supplémentaires aux traitements du composant étendu (qui est bien évidemment vu comme une boîte noire).

3.4. Assurance de qualité

3.4.1. Validation

On entend par validation toute activité qui permet d'obtenir une confiance suffisante dans la conformité de fonctionnement d'un composant logiciel par rapport à sa spécification. La validation s'appuie sur des techniques de test et aussi sur des mécanismes de certification et d'authentification.

Les techniques de test ont pour but d'éprouver le composant, c'est-à-dire de montrer que la réalisation du composant est capable d'offrir correctement les services que le composant propose à son environnement, en essayant de mettre en évidence des cas où le service rendu par le composant est différent du service attendu. Le domaine du test de logiciel distingue traditionnellement le test unitaire et le test d'intégration. Le test unitaire repose sur l'essai une à une des différentes opérations offertes par le composant. Un environnement de test (banc de test) active les opérations en suivant un plan préétabli. Le test d'intégration a pour but d'essayer le composant dans un environnement plus réel, en le faisant interagir avec d'autres composants.

L'emploi des techniques de spécification de service au moyen de contrats a été présenté à la section 2.2.1. Les contrats offrent un avantage certain lorsqu'il faut réaliser les tests unitaires des composants. En effet, la spécification des préconditions permet de délimiter les valeurs des paramètres des opérations ; la spécification des postconditions contraint le résul-

tat et aide à construire des « oracles » (processus capable de décider si le résultat est valide ou non). À ce niveau, les composants logiciels se testent comme les logiciels plus « classiques », d'une façon analogue au test de classe pour les logiciels à objets.

La spécification des contrats de synchronisation permet de construire des tests incluant des séquences complexes d'appel d'opérations. Ce type de test revêt une importance particulière pour les composants logiciels, car ils s'appuient sur des collaborations avec d'autres composants pour exécuter correctement leurs services. Dans un contexte sans composants logiciels, le test des propriétés de collaboration est réalisé lors de l'étape traditionnelle du test d'intégration. Un composant logiciel étant destiné à s'exécuter dans un environnement inconnu à sa création, le test des propriétés de synchronisation remplace en théorie le test d'intégration.

Le test des propriétés extra-fonctionnelles des composants logiciels présente de nombreuses difficultés. Si le test s'effectue sur un banc de test, il est difficile de reproduire les conditions nominales d'utilisation du composant. En effet, supposons que le contrat de qualité de service (propriété extra-fonctionnelle du composant) spécifie un débit en termes de transactions effectuées par seconde. Ce débit dépend des performances de l'environnement d'exécution du composant, par exemple la puissance de calcul ou le débit de périphériques de stockage. Selon les algorithmes utilisés dans le composant, le débit peut être une fonction linéaire des performances de l'environnement, mais il peut ne plus respecter cette linéarité en dehors d'un certain intervalle de performances. Pour cette raison, le test de performances de type boîte noire est peu fiable. De façon analogue, le test des contrats de qualité de service d'un assemblage de composants présente aussi de sérieuses difficultés. Les assemblages peuvent se comporter de façon non déterministe dans certaines situations, et donner des performances chaotiques qui ne sont pas calculables par des opérations simples.

Le test *in situ* d'un composant logiciel ou d'un assemblage de composants a l'avantage de faciliter la mise en condition réelle de l'architecture. Il est cependant difficile de bien réaliser de tels tests. En premier lieu, il faut que l'infrastructure d'exécution permette la collecte fiable d'informations « de bas niveau » sur les performances de l'architecture en cours de test. De tels mécanismes de collecte sont rarement disponibles dans des environnements d'exploitation car ceux-ci sont généralement conçus pour maximiser l'efficacité et la collecte

d'information est coûteuse en ressources et donc pénalisante en terme d'efficacité. Comme pour l'observation de phénomènes physiques, l'observation peut perturber le comportement du système observé. D'autre part, il est difficile de contrôler les conditions réelles d'exécution et donc d'explorer différentes situations d'exécution de façon systématique.

Un composant logiciel ayant satisfait aux tests unitaires ne requiert pas en principe le passage de tests d'intégration puisque le but ultime de la construction par composants est de garantir qu'un assemblage de composants corrects est correct. Le test unitaire étant à la charge du fournisseur de composant, le concepteur d'applications à composants logiciels n'a à sa charge que le test global de l'application. Dans la pratique, les mécanismes de description des services ne permettent pas encore de garantir qu'un assemblage de composant est correct. Le concepteur d'application par composants aura donc intérêt à tester les assemblages au moyen des techniques de test d'intégration.

3.4.2. Certification de validité

Les techniques de certification ont pour but de montrer à l'utilisateur d'un composant que la validité d'un composant par rapport à une spécification est approuvée par une autorité garante et que le composant n'a pas été altéré après passage de ces tests. Ces tests peuvent avoir été effectués par l'autorité de certification, d'une façon analogue aux tests par un laboratoire d'essai des composants. Le laboratoire d'essai n'ayant pas accès au code source, il ne réalisera que des tests unitaires de conformance du type *boîte noire*. La certification externe est une pratique courante pour certains types de logiciels, par exemple les pilotes de périphérique (qui sont en fait une forme primitive de composants logiciels). Bien évidemment un laboratoire d'essai ne testera que certains aspects d'un composant logiciel, par exemple selon des scénarios d'utilisation nominaux standard. Ces scénarios incluront les paramètres extra-fonctionnels tels que les performances et la consommation de ressources partagées (espace mémoire, processeur) employées dans telle ou telle configuration.

3.4.3. Robustesse

La robustesse est la capacité à résister à des défaillances internes ou externes. Les tests subis par un composant logiciel ne garantissent pas l'absence de faute interne. De même, l'environnement du composant peut agir sur lui de manière imprévue, par exemple en ne respectant pas les contrats de service.

Dans tous les cas de figure, le comportement du composant logiciel doit être sûr. Une propriété recherchée est le confinement d'erreur : une erreur interne à un composant doit être détectée au plus tôt et ne doit pas déclencher de comportement visible erroné. Ceci peut impliquer que le composant logiciel passe dans un mode de fonctionnement dégradé, voire un mode d'arrêt (selon une approche nommée « arrêt sur panne », en anglais *fail-stop*). La détection d'erreur peut s'appuyer sur des mécanismes internes de surveillance des contrats : par exemple, les préconditions peuvent être vérifiées au début de l'appel d'un service, et les postconditions à la fin de l'exécution du service. Le confinement d'erreur repose en général sur des mécanismes d'isolation des ressources, notamment les espaces mémoire : le matériel gérant les accès à la mémoire empêche tout accès hors des zones allouées au composant. D'une façon générale, les mécanismes de protection offerts par les systèmes d'exploitation sont à la base des techniques de confinement d'erreur.

3.5. Cycle de vie

Le concept de composant logiciel n'a d'intérêt que s'il trouve une place appropriée dans l'ensemble d'un processus de construction de logiciel. La construction par composants suivra généralement les grandes étapes exposées ci-après, qui sont le fruit de l'adaptation des phases classiques du cycle de vie d'un logiciel aux particularités d'emploi des composants.

3.5.1. Analyse par composants

L'étape d'analyse a pour fonction l'identification des concepts et des problèmes à résoudre au moyen du système logiciel. Certaines techniques d'analyse des besoins introduisent le concept de composant dès le début de l'activité de modélisation. L'activité d'analyse permet alors de repérer un composant d'analyse déjà identifié et étudié dans un travail d'analyse antérieur, ou de définir un composant pour une réutilisation ultérieure dans un autre travail d'analyse. Ces techniques considèrent des domaines de service (expression d'une catégorie de travaux à réaliser) comme des composants. L'emploi de la notion de composant à ce niveau permet de réutiliser le travail d'analyse dans d'autres contextes similaires. Il est à noter cependant que l'identification de composants au moment de la phase d'analyse n'est pas indispensable pour effectuer ensuite la conception au moyen de composants.

3.5.2. Conception par modèle

La phase de conception a pour but la construction d'une architecture capable de répondre aux besoins décrits dans la phase d'analyse. De nombreuses méthodes s'appuient sur la construction de modèles à l'aide d'une notation graphique ou textuelle (par exemple UML). Les méthodes de conception par composants préconisent de s'appuyer sur des composants existants pour concevoir l'architecture, et d'organiser sous formes de composant certaines parties de l'architecture obtenue, afin de faciliter l'évolution et le réemploi.

Certains procédés de conception préconisent de construire des modèles d'architecture indépendants des détails de plate-forme matérielle (type de processeur, taille mémoire, etc) et logicielle (type de système d'exploitation, technologies de communication, langage de développement, etc). Les composants ainsi obtenus peuvent être réutilisés dans d'autres architectures, même si celles-ci sont destinées à fonctionner sur des plates-formes matérielles différentes. La conception de l'architecture dépendante de la plate-forme est réalisée par la transformation de l'ensemble du modèle indépendant de la plate-forme en modèle dépendant de cette plate-forme. Cette transformation peut en principe être automatisée, au moins en partie.

Il est à noter que lors de l'étape de conception, le concepteur ne choisit pas en réalité telle ou telle mise en œuvre de composant mais tel ou tel service, défini par ses interfaces, ports et contrats. La sélection d'une mise en œuvre se fera au cours de la phase de déploiement.

3.5.3. Déploiement

Cette phase a pour but la sélection des mises en œuvre de composants, des lieux et modalités d'exécution des différentes instances de composants lors de l'exécution de l'application, la fabrication de la distribution, autrement dit du fichier qui contiendra tout ce qu'il faut pour installer l'application, puis finalement l'installation et la configuration de l'application. Selon le domaine d'application, les techniques de déploiement peuvent varier énormément. Par exemple, une application de gestion fondée sur des services Web et une architecture de type trois tiers seront conçues pour installer sur le poste client les composants chargés de l'interface homme-machine et sur le serveur d'exécution la partie dite « logique métier » (*business logic*). Une application de contrôle de freinage dans un véhicule aura des contraintes bien différentes, en terme d'occupation mémoire et d'allocation de processeur.

3.5.4. *Gestion des variantes et versions successives*

La construction de logiciel construit par composants facilite le traitement du problème de l'évolution des technologies. Supposons que le logiciel comporte un composant sophistiqué faisant appel à l'état de l'art technologique à un moment donné. Ce composant constituant une source de revenus pour l'entreprise qui le commercialise devra impérativement suivre les évolutions technologiques pour rester compétitif. Les acheteurs de ce composant pourront facilement éviter le vieillissement de leurs applications à composants. Il leur suffira d'obtenir les nouvelles versions des composants, pour maintenir leur application à jour. A contrario, une application conçue avec des parties entièrement internes à la société ne pourra aisément suivre les évolutions des technologies, des standards et des pratiques, car l'économie d'échelle ne jouera pas.

L'évolution des composants peut être considérée comme une dimension supplémentaire. Le long de cette dimension, les différentes propriétés des composants seront déclinées en variantes. L'industrie du logiciel a depuis longtemps pris en compte la notion de version dans le contexte général d'un logiciel fait sans composants. Par exemple, les fournisseurs de bibliothèques de logiciel adoptent souvent un système de numéro de version majeure et version mineure pour étiqueter les évolutions de la mise en œuvre d'une bibliothèque. Deux versions de bibliothèque ayant le même numéro majeur de version sont compatibles du point de vue des interfaces des procédures de la bibliothèque. Autrement dit, l'emploi d'une nouvelle version sans changement de numéro n'impose pas de modification du code source de l'application mais seulement une nouvelle édition de liens.

La conception par composants bouleverse quelque peu ce schéma classique car les spécifications de service et les mises en œuvre peuvent subir des évolutions totalement séparées. En effet, des fournisseurs différents peuvent exister pour une même version de spécification de service, chaque fournisseur proposant différentes versions de mises en œuvre, tandis que la spécification du service peut elle-même évoluer.

D'autre part, une architecture à composants n'est pas en général une structure en couches, où chaque couche ne dépend que des services des couches inférieures. Un changement de version d'un composant peut remettre en cause ces dépendances.

De plus, certains fournisseurs construiront leurs produits par assemblage et par extension d'autres composants (de fournisseurs tiers).

4. **Aspects pratiques et technologies existantes**

Les idées et les concepts de composant logiciel exposés plus haut n'ont de valeur que si ils sont mis en pratique par les fournisseurs de composant et leurs clients. Pour qu'un marché du composant logiciel puisse se mettre en place, un certain nombre de conditions doivent être réunies :

- un standard de description de composants doit mettre en place un langage commun entre fournisseurs et clients ;
- un ou plusieurs standards d'exécution doivent exister ;
- les plates-formes d'exécution doivent proposer des services d'exécution de composants qui obéissent aux standards d'exécution ;
- l'ensemble de ces standards technologiques doit être supporté par des outils de conception adaptés.

4.1. **Standard de description de composants**

Toutes les notions présentées dans les sections précédentes ne sont que des concepts théoriques. Une entreprise qui décide de mettre un composant sur le marché doit pouvoir exprimer dans un certain langage spécialisé toutes les propriétés de son composant. Une société qui s'intéresse à ce composant doit pouvoir comprendre de façon complète et non ambiguë les propriétés du dit composant. Cette compréhension mutuelle entre fournisseur et client ne fonctionne que s'il existe une langue commune, suffisamment riche et précise pour ne laisser aucune place à des erreurs d'interprétation. Pour prendre un exemple concret et non informatique, les sociétés du domaine de la construction mécanique ont élaboré depuis longtemps un langage commun, graphique (le dessin industriel de mécanique). Grâce à cette notation, un constructeur de machine peut examiner et choisir des sous-ensembles sur des catalogues de fournisseurs.

À l'heure actuelle, l'industrie du composant logiciel ne dispose pas encore d'un langage unifié pour la description des composants. Il est d'ailleurs illusoire de penser qu'un seul langage puisse traiter correctement les besoins de domaines d'application très variés. Pour aborder ce problème de la diversité, les recherches actuelles s'appuient sur le concept de métalangage : un tel langage permet de décrire d'autres

langages. Partant de ce principe, il est possible de définir un langage de description de composant qui couvre les besoins élémentaires communs à tous les domaines. Des extensions permettent de décrire les propriétés de composants spécifiques d'un certain domaine d'application. De même, les détails propres à telle ou telle technologie seront décrits par des extensions spécifiques de ce domaine.

Pour que ces langages de description puissent être réellement utilisables, ils doivent gagner un statut de standard et donc être proposés et gérés par un organisme de standardisation reconnu, tel l'ISO (International Standards Organisation) ou l'OMG (Object Management Group). À l'heure actuelle, ces deux organismes ont adopté la notation UML comme langage commun et générique pour la description d'architectures logicielles.

Enfin, les langages de description de composant doivent s'appuyer sur des modèles bien construits sur un plan formel (au sens mathématique du terme), pour qu'il n'y ait pas d'ambiguïté ou d'erreur d'interprétation des descriptions d'architectures et de leurs propriétés. La sélection de modèles formels sous-jacents est une tâche difficile, surtout lorsqu'il s'agit d'établir le consensus général qu'un standard doit offrir.

4.2. Standards et plates-formes d'exécution

Disposer de la description des propriétés d'un composant logiciel n'est pas suffisant pour l'exécuter concrètement sur une infrastructure informatique. En effet, les différents concepts propres à la conception de composants (ports, connexions, interfaces, contrats, etc.) doivent être réalisés concrètement sur une architecture matérielle. Cette réalisation passe par l'emploi de compilateurs, de bibliothèques, de noyaux d'exécution (*runtime*), de services fournis par les systèmes d'exploitation, etc. On désigne tout cet ensemble par le nom d'infrastructure d'exécution. De même que les vendeurs d'application proposent souvent une version d'un produit pour système d'exploitation, les vendeurs de composants doivent fournir à leur catalogue une version d'un composant pour chaque infrastructure d'exécution. Le nombre de telles infrastructures est élevé, en particulier en raison de la multiplicité des plates-formes matérielles. Outre la multiplication des efforts, la construction de toutes ces variantes exige la collaboration de spécialistes de chaque plate-forme, ce qui est difficile à réaliser. Plusieurs techniques peuvent réduire cette multiplication des variantes :

- les techniques fondées sur la transformation de modèles (passage d'un modèle indépendant de la plate-forme d'exécution à un modèle dépendant de cette plate-forme) ;
- les techniques fondées sur la synthèse de mises en œuvre spécifiques à partir de fragments spécifiques réutilisables ; l'une des techniques est le tissage d'aspect (Kiczales, 1997), qui permet de « greffer » des fragments spécialisés à un cœur indépendant d'une plateforme ;
- les techniques fondées sur l'emploi de machines virtuelles conçues pour être indépendante de la plate-forme (approches Java et .Net).

4.3. Outils d'aide à la conception

Ainsi qu'il a été dit plus haut dans ce chapitre, la construction d'applications à composants recouvre des activités bien spécifiques et différentes. En conséquence, des outils différents devront fournir une aide différente selon l'activité.

Les concepteurs d'applications à composants doivent pouvoir :

- rechercher des spécifications de service standard, en fonction de divers critères (domaine de service, détails divers, etc.) ;
- rechercher des mises en œuvre, selon les services assurés, la disponibilité sur telle ou telle plate-forme, les coûts de licence et de maintenance, etc ;
- assembler les composants dans une application, vérifier la compatibilité des services, tester les mises en œuvre et déployer leur application.

Les concepteurs de composants doivent disposer d'un environnement facilitant la création d'un composant à partir de code source dans tel ou tel langage de programmation, ou par extension et assemblage de composants existants.

5. Conclusion

Un chemin considérable a été parcouru depuis les prévisions de McIlroy en 1969. De nombreux travaux de recherche et de développement ont permis de construire des architectures commerciales de composants, utilisables dans des applications de taille réelle. Un marché du composant commence à s'établir. Cependant, ces architectures commerciales ne permettent pas de mettre en œuvre toutes les idées issues de la recherche.

Bibliographie

- G. Beneken, U. Hammerschall, M. Broy, M. V. Cengarle, J. Jürjens, B. Rumpe, and M. Schoenmakers, Componentware: State of the Art 2003, in *Understanding Components Workshop*, 2003.
- A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, Making Components Contract Aware, *IEEE Computer Special Issue on Components*, vol. 13, 1999.
- E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, Addison-Wesley, 1995.
- J.-M. Jézéquel, M. Train, and C. Mingins, *Design Patterns and Contracts*: Addison-Wesley, 1999.
- G. Kiczales, Aspect Oriented Programming, *ACM Sigplan Notices*, vol. 32, pp. 162, 1997.
- B. Meyer, Applying Design by Contract, *IEEE Computer Special Issue on Inheritance and Classification*, vol. 25, pp. 40-52, 1992.
- M. McIlroy, J. Buxton, P. Naur, and B. Randell, Mass Produced Software Components, *Software Engineering*, 1969.
- C. Szyperski, *Component software, beyond object-oriented programming*: Addison-Wesley, 2002.