ECOLE NATIONALE SUPERIEURE DE MECANIQUE ET D'AEROTECHNIQUE

THESE

Pour l'obtention du grade de

DOCTEUR DE L'UNIVERSITE DE POITIERS

(Faculté des Sciences Fondamentales et Appliquées)
(Diplôme National – Arrêté du 30 Mars 1992)

Ecole Doctorale des Sciences pour l'Ingénieur

Secteur de recherche: INFORMATIQUE

Présentée par

Guillaume TEXIER

Contribution à l'ingénierie des systèmes interactifs : Un environnement de conception graphique d'applications spécialisées de conception

Directeurs de Thèse: Guy PIERRA, Laurent GUITTET

Soutenue le 13 Novembre 2000
Devant la commission d'examen

JURY

Y. Bertrand	Président	Professeur, Université de Poitiers
A. Bouras	Rapporteur	Professeur, Université Lyon 2
B. David	Rapporteur	Professeur, Ecole Centrale de Lyon
J.D. Fekete	Examinateur	Maître de conférences, Ecole des Mines de Nantes
P. Girard	Examinateur	Professeur, Université de Poitiers
L. Guittet		Maître de conférences, ENSMA, Poitiers
H. Legrand	Examinateur	Directeur de Recherche, Matra Datavision
G. Pierra		Professeur, ENSMA, Poitiers

LABORATOIRE D'INFORMATIQUE SCIENTIFIQUE ET INDUSTRIELLE

Ecole Nationale Supérieure de Mécanique et d'Aérotechnique

Téléport 2 - 1, avenue Clément Ader - BP 40109 - 86961 Futuroscope

Tél: 05.49.49.80.63 - Fax: 05.49.49.80.64

Je remercie

Guy Pierra pour m’avoir accueilli au sein de son la laboratoire, pour la confiance qu’il m’a accordée, pour le temps qu’il m’a consacré afin que je puisse réaliser mes travaux.

Bertrand David et Abdelaziz Bouras pour les remarques critiques et constructives qu’ils ont observées et pour le temps qu’ils ont consacré à la relecture de cette thèse.

Jean-Daniel Fekete et Hervé Legrand pour avoir accepté d’assister à ma soutenance et d’être membre du jury.

Laurent Guittet qui a consacré énormément de temps à ce travail, tant au point de vue de la rédaction que de la recherche proprement dite et de la réalisation de la maquette. Je le remercie pour tous ces conseils et sa bonne volonté pour m’aider à résoudre toutes sortes de problèmes.

Yamine Ait-Ameur, Patrick Girard et Jean-Claude Potier pour l’aide qu’ils m’ont apporté au cours de ces années passées au LISI.

Tous les membres du laboratoire, et plus particulièrement Dago, Dave, Laurent, Eric, Fabrice, Francis, Michael, Manu G et Manu G, Mourad, Pascal pour leur amitié précieuse.

Mon père, ma mère, Florence et Domi ainsi que tous mes amis de Poitiers et d’ailleurs pour leur soutient.

A Virginie et à Margaux,

Table des matières

Introduction

La productivité résultant d'un système de CAO (Conception Assistée par Ordinateur) dépend très largement de la possibilité de l'adapter au processus de conception spécifique de l'entreprise où il est implanté.

Cette adaptation consiste à intégrer, dans le système, des objets spécifiques (par exemple l'entreprise conçoit des sièges d'avions ou des roulements) ayant un comportement spécifique (le nombre de renforts d'un siège dépend du nombre de sièges contigus) dont la conception fait l'objet d'une procédure et donc d'un dialogue spécifique.

Une application spécialisée de conception technique est une application de conception technique qui est adaptée à un domaine ou à une activité particulière. Il existe sur le marché un grand nombre d'applications spécialisées de conception. Par exemple, les paysagistes utilisent des logiciels qui manipulent des éléments spécifiques à leurs corps de métiers (les arbres, les fleurs, etc…). On peut aussi citer les logiciels « grand public », tel que «intérieur 3D », qui permettent à un néophyte de dessiner une maison ou un appartement et d'y placer des meubles. Dans ce cas les éléments manipulés par le système sont les murs, les portes, les meubles etc.

La création d'une application spécialisée demande un gros effort de développement. En effet, chaque élément manipulé par le système doit être programmé de façon traditionnelle en écrivant le code. Ainsi, d'une part, les spécialistes du domaine visé par l'application spécialisée ne sont en général pas informaticiens et ne peuvent donc pas écrire eux même le code nécessaire à la description de leur activité. D'autre part, les spécialistes de l'informatique n'ont pas, en général, une connaissance assez approfondie du domaine d'activité pour répondre à tous les besoins des utilisateurs de l'application spécialisée.

Le but de ce travail est de montrer qu'il est possible de créer des applications de conception technique généralistes permettant à un spécialiste d'un domaine d'activité particulier, de créer interactivement des applications spécialisées de conception manipulant les éléments de son domaine sans recourir à la programmation traditionnelle.

Pour créer une application, on distingue traditionnellement deux composants à concevoir. Il y a d'une part le composant regroupant les primitives spécifiques à l'application, appelé noyau fonctionnel, et d'autre part l'interface qui regroupe la présentation (ce avec quoi l'utilisateur interagit) et le dialogue qui décrit comment l'utilisateur interagit avec les éléments du noyau fonctionnel. En conséquence, la spécialisation d'une application, nécessite :

1. la description des nouveaux éléments spécifiques au domaine d'application visée; c'est-à-dire la description de la façon de les construire, ainsi que les fonctions et les primitives permettant de les interroger et de les modifier,
2. la modification de l'interface de manière à ce que l'utilisateur puisse interagir avec les éléments spécifiques du domaine; c'est-à-dire la modification de l'interface générique pour offrir à l'utilisateur la possibilité de construire, de détruire, de modifier ou d'interroger des éléments spécifiques d'un domaine et ce de la même manière que pour les éléments natifs de l'application.

Ces deux points nous permettent d'identifier clairement les objectifs de notre travail, à savoir :

1. Etudier la faisabilité, puis définir une méthode permettant la définition interactive, sans programmation, des éléments spécifiques que l'on souhaite ajouter aux objets manipulés par un système pour spécialiser celui-ci en direction d'un domaine,
2. Développer une approche, un modèle et des outils permettant la modification, sans programmation, de l'interface d'un système interactif pour accéder à de nouvelles fonctions ou de nouvelles classes d'objets manipulés, eux-mêmes définis de façon interactive.

La première partie de ce mémoire présente le contexte de l'étude. Nous commençons par décrire le modèle d'architecture sur lequel nous basons notre travail. Ceci permet de clarifier les deux objectifs à atteindre : la modification dynamique de l'interface et la définition interactive de classes d'objets spécifiques d'un domaine d'activité.

Il existe de nombreux outils permettant de définir l'interface des applications graphiques interactives. Ces outils se basent soit sur une description explicite de la couche de présentation (approche ascendante), soit sur la définition du dialogue (approche descendante). En ce qui concerne notre problème, on note que l'approche ascendante n'offre aucune solution efficace pour la définition du contrôleur de dialogue d'une application de CAO. L'approche descendante, quant à elle, n'autorise pas, à notre connaissance, la modification dynamique de l'interface. Ceci nous amènera dans le chapitre suivant à proposer une nouvelle approche pour la représentation modulaire du contrôleur de dialogue [Bass, et al. 1991] d'une application graphique interactive.

En ce qui concerne la définition interactive de nouvelles classes d'objets, les progrès réalisés depuis le début des années 90, tant dans le domaine de la programmation par démonstration (PBD) [Cypher 1993] que de la géométrie paramétrée [Shah et Mäntylä 1995] permettent à un utilisateur non informaticien de créer des programmes interactivement. On note cependant

qu'aucune de ces deux approches, en fait très voisines, ne permet, à l'heure actuelle, de définir complètement de nouvelles classes d'objets. Elles se contentent en général de ne définir que très partiellement les objets de la classe, soit à travers leurs méthodes de construction soit à travers certaines modifications possibles. Une extension de ces techniques apparaît donc nécessaire si l'on souhaite permettre à un utilisateur final de définir complètement les objets spécifiques de son domaine d'activité (e.g. meubles de cuisines, poutres et plaques,…) à partir d'un système manipulant seulement des objets généraux (lignes, contours, primitives solides,…).

Le chapitre II présente notre contribution dans le domaine de la modélisation et du développement d'Interface Homme-Machine. Nous y proposons notre technique de conception du contrôleur de dialogue basée sur la réification d'éléments du dialogue. La boîte à outils que nous proposons pour spécifier et développer le dialogue rassemble un ensemble de classes qui étendent les classiques widgets, dont les flots de sortie ne sont que des événements vers des objets orientés langage qui permettent de spécifier une syntaxe pour le flot de dialogue. Ces classes permettent de décrire l'interface d'une application interactive tant statiquement que dynamiquement. Dans le cas particulier du problème que ce travail vise à résoudre, elles permettent en particulier de modifier dynamiquement l'interface d'une application afin que l'utilisateur puisse accéder à de nouvelles primitives qu'il aurait décrites.

Dans le chapitre III, nous étudions comment permettre à un utilisateur d'un système interactif de créer interactivement (c'est-à-dire sans programmation textuelle) de nouvelles classes d'objet relatives à son domaine d'intérêt. Si des méthodes issues de la géométrie paramétrique et de la programmation par démonstration permettent de définir interactivement les constructeurs associés à de telles classes, elles ne permettent ni de définir des méthodes d'interrogations (sélecteur de classe) ni d'établir des relations entre classes autres que des relations d'agrégations. En règle générale, une unique méthode peut être associée à une classe d'objet définie interactivement. Nous proposons alors d'introduire la notion d'attribut dérivé qui est une propriété dont la valeur peut se calculer algorithmiquement à partir d'un objet défini interactivement et nous montrons comment de tels algorithmes peuvent eux-mêmes être spécifiés interactivement. Notre approche permet donc non seulement de décrire les méthodes de construction de nouvelles classes d'objets, mais également les méthodes de modification et celles d'interrogation des objets. L'approche proposée se base sur un modèle de géométrie paramètrée auquel nous avons ajouté des concepts issus de la programmation par démonstration.

Le quatrième chapitre décrit l'outil que nous avons réalisé pour valider nos propositions. Nous montrons notamment l'intégration des modèles et des méthodes décrites aux chapitres II et III. Nous expliquons sur des exemples simples comment il permet, à un utilisateur final, de décrire de nouvelles classes d'objets. Ensuite, nous montrons comment l'utilisation de la boîte à outils du dialogue permet l'intégration dynamique de nouvelles classes d'objets. Nous présentons ensuite les solutions qui ont été retenues pour l'intégration statique. Enfin, nous donnons un exemple complet d'utilisation de cette maquette.

Chapitre I
Problématique et contexte de l'étude

Il existe sur le marché un grand nombre d'applications de conception technique spécialisées qui offrent à leur utilisateur des primitives de haut niveau d'abstraction afin de modéliser le domaine particulier qu'elle visent à couvrir. Parmi les exemples fréquents, on peut citer les systèmes de conception de cuisines intégrées, la conception de charpentes métalliques, les éditeurs de schémas particuliers. Or, les primitives de ces applications sont souvent difficiles à programmer pour les spécialistes du domaine visé. Inversement, une bonne connaissance des primitives pertinentes est souvent difficile à transférer du spécialiste du domaine vers une équipe de développement non spécialisée. De cette double difficulté est né l'objectif de cette étude, que nous formulons par une question :

Est-il possible de générer des applications interactives spécialisées à partir d'une application généraliste par des techniques qui ne demandent que peu ou pas de programmation textuelle de façon à ce que de tels développements puissent être fait d'une part plus rapidement et d'autre part par des acteurs plus proche du domaine d'application ?

C'est à cette question que notre étude vise à répondre en montrant qu'il est possible de générer de telles applications en se basant sur la génération interactive de nouvelles classes à partir des classes de base existantes au sein d'un système de conception technique généraliste.

En fait, la conception d'une Application Graphique Interactive (AGI) et en particulier d'un système de CAO, nécessite la prise en compte de trois aspects différents : ce que fait l'application, la présentation qu'elle offre sur l'écran et enfin le dialogue qu'elle supporte avec un utilisateur. Afin de structurer le travail des concepteurs de telles applications, des modèles d'architectures comme Seeheim [Pfaff 1985], Arch [Bass, et al. 1991] [UIMS 1992], PAC [Coutaz 1987] ou H^4 [Guittet 1995] ont été proposés. Ils préconisent habituellement de séparer la partie noyau fonctionnel (actions de construction et de représentation des objets de l'application), du contrôle du dialogue, et de la présentation de l'interface. On peut donc d'ors et déjà dire qu'un système de création d'applications spécialisées devra permettre de décrire les trois parties identifiées dans les modèles d'architecture. Pour réduire la difficulté de conception d'une application spécialisée, nous proposons d'utiliser la programmation par démonstration afin de décrire les éléments du noyau fonctionnel et nous utiliserons des outils de type générateur d'interface pour définir son interface (le contrôleur de dialogue et la présentation).

Ce chapitre présente un état de l'art des méthodes et des outils utilisés pour la création des différents composants d'une application graphique interactive.

La première partie de ce chapitre est consacrée aux modèles d'architectures utilisés pour définir une application graphique. Nous décrirons plus particulièrement le modèle d'architecture H^{4} mis en œuvre dans la suite de l'étude.

La deuxième partie est consacrée à l'étude des méthodes et des outils de conception d'application graphique interactive. Nous discuterons notamment les différents types de générateurs d'interface.

Enfin, dans la dernière partie, nous présenterons les différentes méthodes utilisées pour permettre à un utilisateur non informaticien de créer des programmes et notamment, les primitives du noyau fonctionnel d'une application graphique interactive.

1 Les modèles d'architecture

Les modèles d'architecture sont utilisés pour structurer les applications interactives en plusieurs modules afin de faciliter la création d'applications importantes, la maintenance du code, et de permettre la réutilisation de certains modules.

Il existe plusieurs grandes familles de modèles d'architecture. Ils ont tous pour caractéristique de décomposer une application en trois grandes parties : la présentation gère les dispositifs d'entrée-sortie (souris, clavier, écran) ainsi que la visualisation du système à l'écran ; le noyau fonctionnel contient les données et les fonctions spécifiques de l'application ; et enfin, le contrôleur de dialogue gère les langages de dialogue, c'est-à-dire les entrées de l'utilisateur et les réactions du système à ces entrées.

- Les modèles macro-agents ou modèles généraux [Fekete 1996b] tel que Seeheim et Arch décomposent l'application en différents modules en précisant leur nombre, leur nature et leur organisation mais sans préciser leur structure interne.
- L'approche « orienté-objet » est à l'origine des modèles basés sur des micro-agents tels que PAC ou MVC [Goldberg 1984]. Ils décrivent l'application comme une composition de micro éléments qui communiquent entre eux. Ainsi, le modèle PAC est constitué d'agents PAC, eux même découpés en trois sous modules : une partie abstraction rassemblant les

données propre de l'agent, une partie présentation gérant la visualisation sur l'écran et une partie contrôle. Les agents PAC communiquent entre eux par leur partie contrôle.

- Une approche mixte a vu le jour au début des années 90. Il s'agit des modèles d'architectures mixtes tel que H^4 ou PAC-AMADEUS [Nigay 1994]. Ces modèles sont structurés en macro-agents (en général ceux définis dans Arch) qui sont constitués de micro-agents. Ainsi dans PAC-AMADEUS, les macro-agents sont constitués d'agents PAC.

Du point de vue architecture, notre travail est basé sur le modèle H^4, qui répond bien au besoin des systèmes de type CAO. Nous décrivons d'abord ci-dessous les modèles de Seeheim et Arch dont les grands principes ont été appliqués au modèle d'architecture H^4 . Cette partie n'a pas pour but d'être exhaustive, mais permet de définir le cadre nécessaire pour notre travail.

1.1 Le modèle de Seeheim

Le modèle de Seeheim (Figure I.1) isole la gestion du dialogue et la sémantique de l'application. Il définit trois composants logiques servant d'interface entre l'utilisateur et le noyau de l'application. Ces trois composants sont :

La présentation : ce composant est responsable de l'aspect externe de l'application. Il gère les dispositifs d'entrées-sorties tels que l'écran et la souris.

L'interface avec l'application : cette couche regroupe tout ce que l'interface utilisateur à besoin de connaître de l'application pour fonctionner. Elle permet aussi de transformer les données venant de la couche de présentation pour qu'elles soient utilisables par l'application.

Le contrôleur de dialogue : ce composant est situé entre la présentation et l'interface de l'application. Il contrôle le flot des données fournies par l'utilisateur via la couche de présentation. Il appelle les actions de l'application grâce à l'interface de cette dernière.

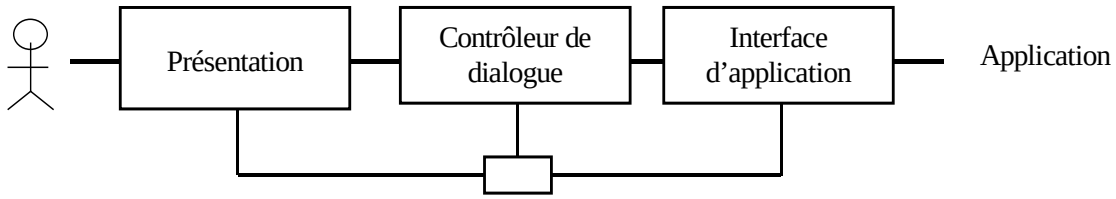


Figure I.1: Seeheim

Le modèle de Seeheim est le premier à avoir défini les grands composants d'une application graphique interactive. La séparation du contrôle du dialogue et de la présentation se retrouve

dans tous les modèles d'architecture ultérieurs. Cependant, l'utilisation de ce modèle pour implémenter une application graphique interactive reste difficile car le rôle de chacun de ces composants n'est pas suffisamment explicite [Tarby 1993]. De plus le lien direct entre le contrôleur de dialogue et la présentation permet difficilement de porter une application d'un support sur un autre. Ce problème sera réglé par le modèle d'architecture Arch.

1.2 Le modèle Arch

Le modèle Arch (Figure I.2) est issu du modèle de Seeheim. Il divise l'application en cinq éléments, et précise beaucoup plus le rôle de chacun d'eux par rapport au modèle de Seeheim.

Le composant de **domaine** contrôle, manipule et conserve les données spécifiques du domaine.

L'adaptateur de domaine implémente les tâches utilisateur relatives au domaine. Il organise les données du domaine pour qu'elles soient manipulables et affichables.

Le contrôleur de dialogue est responsable du séquencement des tâches et de la conservation de la cohérence entre les objets du domaine et leurs vues.

La boîte à outils implémente l'interaction physique avec l'utilisateur (tant au niveau matériel que logiciel).

La présentation permet de séparer la boîte à outils du reste de l'application. Elle est composée de représentations abstraites d'objets d'interactions.

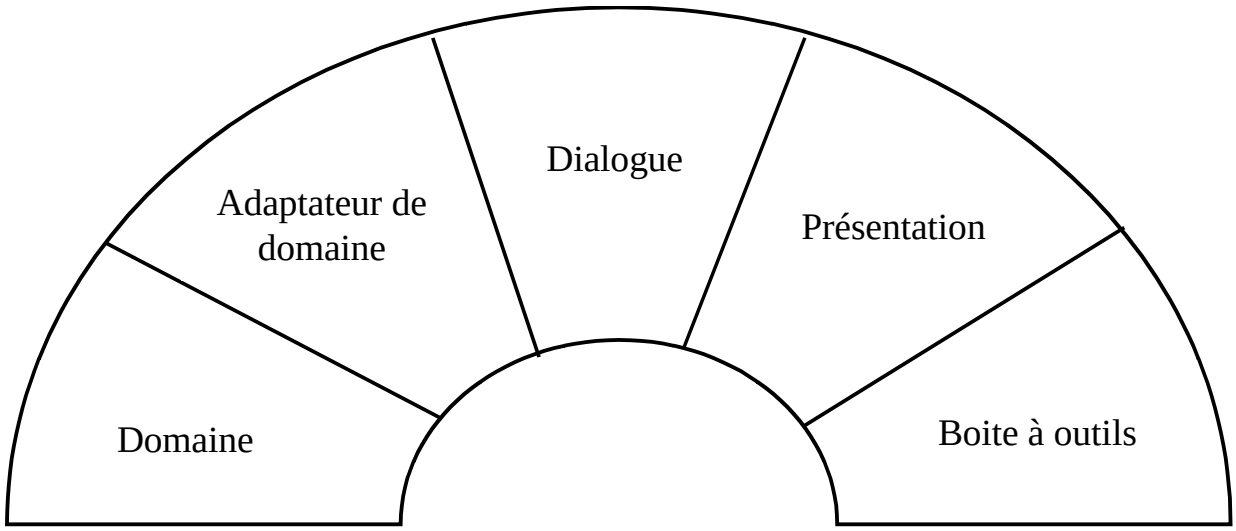


Figure I.2: Le modèle Arch

Le modèle Arch est plus à même de supporter les changements que le modèle de Seeheim, que ce soit au niveau de la boîte à outils ou du domaine grâce aux deux adaptateurs que sont l'adaptateur de domaine et de la présentation.

Les concepteurs du modèle Arch ont pris en compte le fait que toutes les applications ne demandent pas la même quantité de travail pour réaliser certains modules et que certaines fonctions de l'application peuvent être prises en compte par différents modules selon le cas. Il s'agit du caractère « slinky » (traduire par souple) attribué à ce modèle. Par exemple, une fonction d'affichage ou de contrôle qui nécessite la connaissance d'une partie de la sémantique du modèle peut être déléguée à la présentation. On parle alors de délégation sémantique. C'est la cas lorsque pour des raisons de rapidité d'exécution, une fonction de déplacement d'objet est implantée au niveau de la présentation afin d'utiliser la manipulation directe. Lors de l'interaction, seul l'objet de présentation représentant l'objet du modèle est déplacé avec un écho fantôme. L'objet du modèle n'est effectivement déplacé qu'à la fin de l'interaction.

Comme le modèle de Seeheim, Arch ne précise pas la structure interne de ces composants. Ceci est le but des architectures mixtes telles que PAC-AMADEUS et H^4 qui décrivent le corps de chaque composant et notamment celui du contrôleur de dialogue à l'aide de micro-agents.

1.3 Le modèle d'architecture H^4

Le modèle d'architecture H^4 a été créé pour répondre aux besoins des concepteurs de système CAO en terme d'architecture. Ce modèle découle directement de Arch mais précise en plus la structure interne des composants qu'il possède.

Dans cette partie, nous expliquons les difficultés de conception des systèmes CAO. Puis nous montrons comment le modèle H^4 a permis de les prendre en compte.

1.3.1 Spécificités des systèmes CAO

Selon Y. Gardan [Gardan 1991] « la CAO se préoccupe de la création des données qui décrivent l'objet à concevoir, de la manipulation de ces données afin d'aboutir à une forme achevée de conception, et de la génération des informations nécessaires à la fabrication de cet objet à partir de ces données ». Cette définition introduit implicitement deux notions importantes : le modèle et l'interface homme-machine. Le modèle contient les données et les fonctions permettant de construire des objets. L'interface homme machine permet à l'utilisateur d'accéder à ces données et à ces fonctions.

Pour bien comprendre les difficultés liées à la conception d'un système CAO, nous proposons d'étudier ces systèmes en nous basant sur la typologie des systèmes interactifs proposée dans [Pierra 1995]. Nous étudierons les caractéristiques d'arité des tâches, de structure des tâches, d'autonomie des objets et de structure des objets.

La notion de tâche discutée est une conceptualisation en terme de but utilisateur des fonctions remplies par un système. Une tâche correspond a un but de l'utilisateur qui est associé à une fonction du système susceptible de l'accomplir.

1.3.1.1 Arité des tâches

Une application supporte des tâches mono-objets si chaque tâche n'utilise qu'un objet du domaine (au sens du modèle Arch) pour s'exécuter. Elle supporte des tâches multi-objets si une tâche comporte plusieurs objets du domaine.

Les systèmes CAO supportent les tâches multi-objets. Par exemple, ils permettent de construire des droites tangentes à deux cercles. Donc, contrairement aux systèmes mono-objets où les tâches peuvent être intégrées dans les objets conformément au paradigme objet, les tâches des systèmes CAO doivent être représentées indépendamment des objets. Par exemple, la tâche qui consiste à créer un cercle tangent à trois droites est une tâche multi-objets puisque elle a trois droites comme paramètres. Le fait que les tâches des systèmes de CAO portent sur plusieurs objets, amène ces systèmes à utiliser de préférence le mode préfixé pour leur dialogue. C'est-à-dire que pour effectuer une tâche, l'utilisateur désigne d'abord cette tâche, à l'aide d'une case de menu par exemple, puis indique au système les objets sur lesquels elle porte. Cette caractéristique des tâches d'un système CAO limite, de fait, la possibilité d'utiliser la technique de manipulation directe fréquemment utilisée dans les systèmes de dessin [Patry 1999].

1.3.1.2 Structure des tâches

Une application supporte des tâches atomiques si l'utilisateur peut spécifier chaque tâche indépendamment. Le résultat d'une tâche est enregistré immédiatement par le composant spécifique du domaine. L'application supporte des tâches structurées si pour certaines tâches au moins, l'utilisateur doit fournir, (dans un ordre quelconque, mais en général en pré-ordre), l'arbre représentant la hiérarchie des tâches et des sous-tâches. Cette hiérarchie correspond à l'analyse en buts et sous-buts décrite par Norman [Norman 1986]. Le résultat d'une telle tâche n'est connu qu'une fois que l'utilisateur a donné complètement la hiérarchie de tâches/sous-tâches qui la composent.

Le propre des systèmes de conception technique est de supporter les tâches structurées, par exemple la création d'un cercle dont le rayon est la distance entre deux points. On voit ici que la tâche de création du cercle à besoin du résultat de la tâche de calcul de la distance pour s'exécuter. Dans de telles séquences de dialogue, le contexte du dialogue ne peut être conservé ni dans le composant de domaine (tant que l'arbre des tâches/sous-tâches n'est pas achevé, ni au niveau de la boîte à outils (car le calcul de la distance met en jeu le composant du domaine). Contrairement aux systèmes supportant uniquement des tâches atomiques, le contexte du dialogue doit être modélisé explicitement. La création d'un cercle dont le rayon est égal à la distance entre deux points est un bon exemple de dialogue associé à une tâche structurée. L'utilisateur émet, pour commencer, la volonté de créer un cercle. Puis il donne la valeur des paramètres de création. Le paramètre, représentant le rayon, a pour valeur le calcul de la distance entre deux points ce qui correspond à une tâche de niveau d'abstraction plus bas que la création du cercle puisque le résultat de cette tâche est destiné à être utilisé par une autre tâche. Il s'agit donc d'une véritable phrase, associée à une structure de type syntaxique.

1.3.1.3 Autonomie des objets du domaine

Les objets du modèle sont dits relationnels si la représentation de chaque objet du domaine dépend d'autres objets du domaine, sinon ils sont dits autonomes. Si un objet du domaine est autonome, il peut être en bijection avec un unique objet d'interaction qui supporte la fonction de représentation [Duke et Harrison 1993]. Si les objets du domaine sont reliés les uns aux autres (fonctionnellement ou structurellement) alors la modification d'un objet du modèle peut influer sur la présentation des autres.

Les applications de CAO utilisent toujours des objets ayant une structure relationnelle. Ceci entraîne que la représentation des objets du modèle ne peut être entièrement gérée par la couche de présentation. Par exemple, dans un système, si un cercle « paramétrique » du modèle est défini comme étant tangent à trois droites, la modification de l'une d'entre elles entraîne la modification du cercle. Leurs représentations ne sont pas indépendantes. De même, dans un modeleur de type topologique, chaque arête est liée aux sommets qui la limitent et chaque face est liée aux arêtes qui en constituent la frontière.

1.3.1.4 Structure des objets du domaine

Les objets du modèle sont structurés lorsque plusieurs niveaux d'un objet, structuré par agrégation, peuvent être accessibles à l'utilisateur. Ils sont simples lorsqu'un objet du modèle n'est pas une partie d'un autre objet du domaine. Lorsque les objets sont simples, leur

désignation ainsi que l'écho de cette désignation peuvent être réalisés directement dans la couche de présentation. Lorsque les objets sont structurés, la désignation ne peut être interprétée que par le composant de domaine. Les informations concernant les données de sélection venant de la boîte à outils (par exemple : position du pointé ou objet graphique) doivent donc être transmises au composant de domaine.

Les systèmes CAO utilisent souvent deux modèles, un modèle d'objets simples (non structurés) appelé CSG (Constructive Solid Geometry) et un modèle d'objets structurés appelé B-Rep (Boundary Representation). Le modèle CSG est notamment utilisé lors de toutes les opérations booléennes comme la fusion de deux objets ou la coupure d'un objet par un autre. Le modèle B-Rep permet d'accéder à toute la structure de l'objet.

Le modèle B-Rep illustre bien la difficulté de sélection des objets structurés puisque le même pointé de sélection sert à désigner un objet, une face (A), une arête (B) ou un sommet (C) (voir Figure I.3).

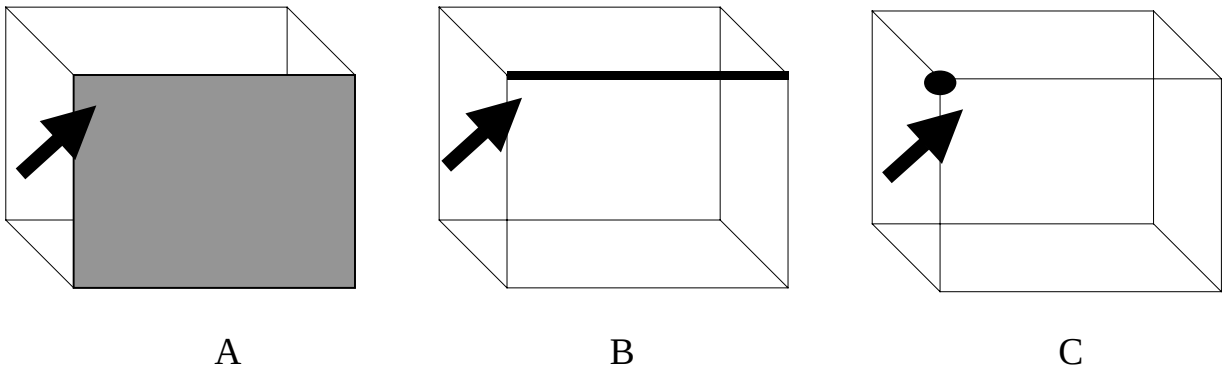


Figure I.3: Sélection des objets structurés

Le problème de la désignation des éléments B-Rep d'un objet vient du fait que les boîtes à outils ne permettent pas de représenter directement ce type d'objet. En général, elles ne prennent en compte que des éléments géométriques simples comme les segments, les cercles, les arcs etc… et n'ont pas connaissance de la structure topologique de l'objet affiché. Cependant, depuis quelques années, les éditeurs de logiciels de CAO proposent des bibliothèques géométriques qui permettent d'afficher des objets structurés. Ainsi, la bibliothèque CAS.CADE de Matra-DataVision permet d'afficher n'importe quelle forme complexe. Ces objets de présentation connaissent l'ensemble de la structure de l'objet qu'ils

représentent et permettent ainsi la désignation des élément topologiques (faces, arrêtes, sommets) au niveau de la boîte à outils.

1.3.1.5 Synthèse sur les caractéristiques des systèmes CAO

Nous pouvons maintenant établir les besoins en terme d'architecture des systèmes CAO.

- Les systèmes CAO traitent des tâches multi-objets. Donc, pour la majorité des systèmes CAO le dialogue devra être préfixé (l'utilisateur indique la commande suivie de ses opérandes).
- Les tâches sont structurées et doivent être explicitement modélisées et regroupées en niveaux d'abstraction.
- Les systèmes CAO travaillent sur des objets dont les représentations sont reliées entre elles, et les objets qu'ils manipulent sont eux-mêmes structurés. Ces deux points indiquent que seul le noyau fonctionnel connaît la sémantique des objets du domaine et est donc capable de gérer la représentation des objets tant en affichage qu'en désignation.

Le modèle d'architecture H^4 a été créé pour permettre de concevoir plus facilement le dialogue des systèmes CAO tant dans l'organisation des macro-agents que dans celle des micro-agents qui les composent.

1.3.2 Le modèle H^4

Le modèle d'architecture H^4 (Figure I.4) est un modèle mixte. Il décrit non seulement les macro composants de l'architecture ainsi que leurs relations mais aussi leur structure. Il précise explicitement la structure du contrôleur de dialogue afin que celui-ci supporte les tâches structurées. Nous allons dans un premier lieu décrire les macro composants de H^4. Puis nous décrirons les éléments nécessaire à la gestion des tâches structurées. Enfin, nous expliquerons la structure interne du contrôleur de dialogue dans H^4.

1.3.2.1 Les macro composants de H^4

Le modèle d'architecture H^4 se compose de cinq modules dont quatre possèdent une sous-structure hiérarchique.

1. **Le noyau fonctionnel** : ce composant décrit le modèle de données relatif à l'application ainsi que les primitives agissant sur ce modèle.

2. **L'adaptateur de noyau fonctionnel** : ce module est chargé de l'appel des actions du noyau fonctionnel selon les demandes de l'utilisateur qui lui sont fournies via le contrôleur de dialogue. Il transforme les données venant du contrôleur de dialogue en données utilisables par le noyau fonctionnel. Il gère l'affichage des données du noyau fonctionnel en utilisant des primitives fournies par l'adaptateur de présentation.
3. **Le contrôleur de dialogue** gère le flot des entrées de l'utilisateur venant de l'adaptateur de présentation. Il appelle l'adaptateur de noyau fonctionnel afin de réaliser des actions sur le noyau fonctionnel. C'est à ce niveau que le dialogue est structuré en tâches/sous-tâches sous forme d'agents de contrôle appelés interacteurs de contrôles.
4. **L'adaptateur de présentation** est une boîte à outils virtuelle. Il permet l'indépendance entre l'application et le système de fenêtrage avec les outils d'interactions (les « widgets ») qui lui sont associés. Il est chargé de transformer les données venant de l'utilisateur via la boîte à outils, en données utilisables par le contrôleur de dialogue. Il doit fournir à l'application les outils permettant de réaliser l'affichage des données du noyau fonctionnel.
5. **La présentation** gère les entrées et les sorties de l'application (tant physiques que logicielles) à l'aide d'outils d'interactions (widgets). Ce composant correspond à la boîte à outils de Arch.

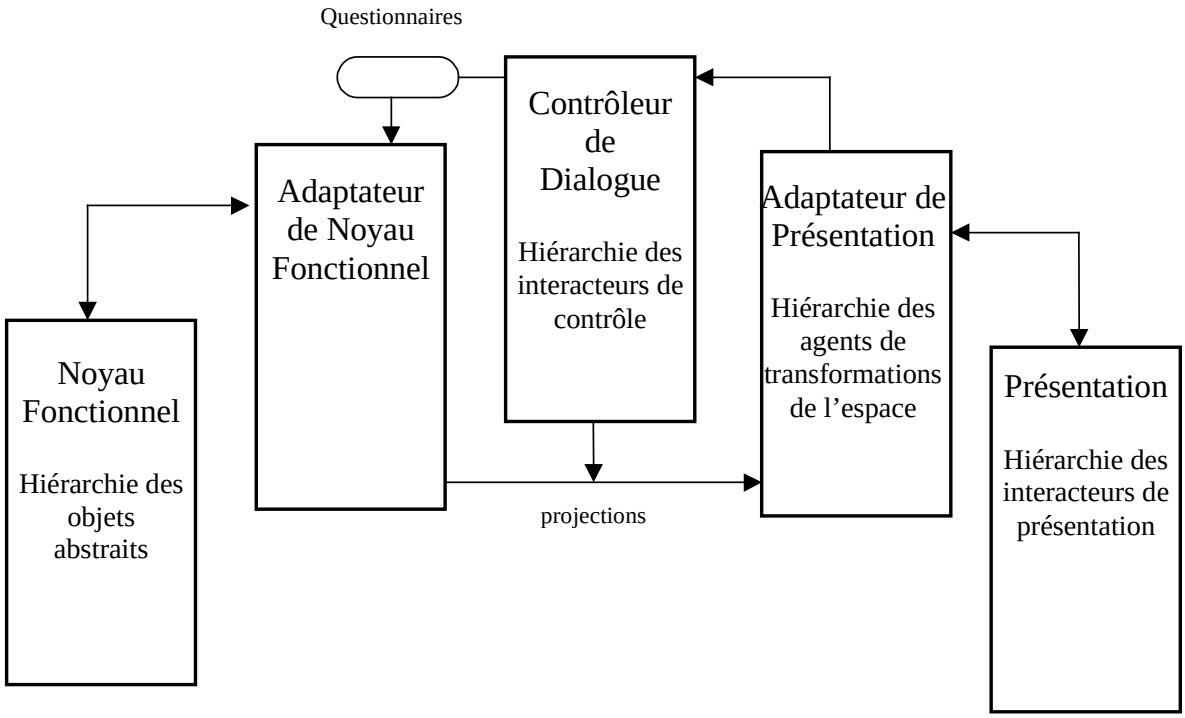


Figure I.4: le modèle H[4]

1.3.2.2 Le dialogue structuré

Pour atteindre un but de haut niveau, l'utilisateur décompose celui-ci en sous-buts afin d'atteindre des buts suffisamment modestes pour pouvoir être réalisés directement par des actions disponibles dans le système. Cette décomposition en buts/sous-buts a été analysée par Norman [Norman 1986] et correspond à la notion de tâches structurées. Une tâche structurée est donc caractérisée par le fait qu'elle utilise le résultat calculé par une autre tâche.

L'énoncé d'une tâche structurée peut s'effectuer de deux façons.

L'énoncé en post-ordre consiste :

- à effectuer d'abord les tâches de bas niveau,
- à mémoriser leur résultat comme objet du domaine,
- à effectuer ensuite les tâches de plus haut niveau en désignant les résultats intermédiaires précédents.

Cette approche impose une importante surcharge cognitive puisque, ainsi que le remarque Norman, la conception de l'arbre des buts/sous-buts est effectuée en pré-ordre.

Définir en pré-ordre une tâche structurée nécessite la possibilité d'exprimer la hiérarchie des tâches et sous-tâches qui constituent cette tâche structurée. Cette expression constitue un **dialogue structuré** dans lequel chaque tâche (sauf celles appelées terminales) s'effectue dans le contexte d'une tâche de plus haut niveau pour laquelle elle produit un résultat. L'exemple de la Figure *I.5* (appelé « création de segment » dans la suite) montre la hiérarchie des tâches dans le cadre d'une construction géométrique.

Dans cet exemple, l'utilisateur crée un segment dont l'une des extrémités est le centre d'un cercle et l'autre l'intersection de deux segments. Nous voyons ici le cas où plusieurs tâches de même niveau coexistent (Intersection et Centre) et où une tâche doit mémoriser des éléments intermédiaires. La tâche création d'un segment mémorise le résultat du calcul de l'intersection en attendant de recevoir la deuxième extrémité du segment.

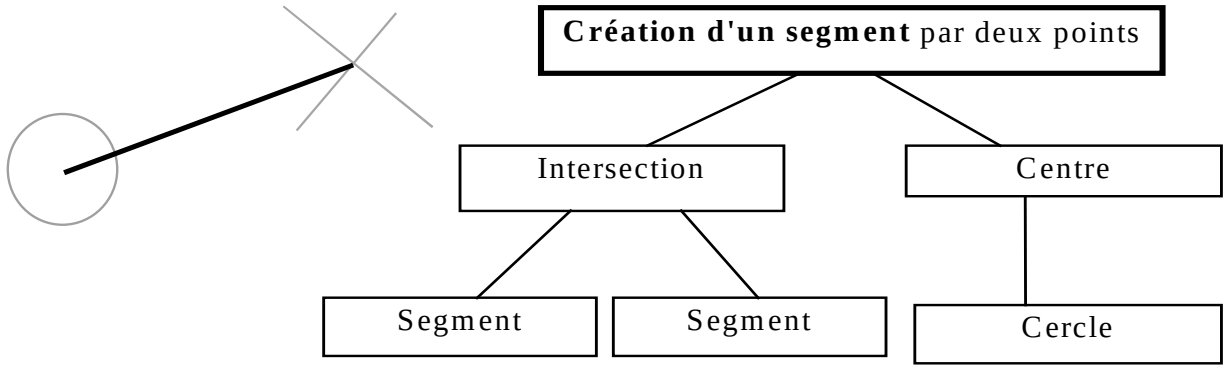


Figure *I.5*: Création d'un segment par contraintes

Ainsi que le montre l'exemple, un dialogue structuré est un dialogue possédant les caractéristiques suivantes :

Le mécanisme de production/consommation

La notion de consommation décrit le fait qu'une tâche utilise le résultat d'une autre tâche pour s'exécuter. Si nous reprenons l'exemple « création de segment », la tâche de « création d'un segment » consomme des points. La notion de production décrit le fait qu'une tâche génère une information qui sera utilisable pour une autre tâche. La tâche Intersection produit un point.

La hiérarchie des tâches

Une hiérarchie naturelle est induite par la notion de consommation/production. Nous appellerons tâches terminales[1] les tâches identifiées comme pouvant être associées à un des objectifs de plus haut niveau (au sens de Norman) de l'utilisateur (créer un segment dans l'exemple « création de segment »). Les tâches terminales modifient l'état du système mais ne produisent pas de résultats directement exploités par une autre tâche. On appelle sous-tâches de production celles dont le rôle est de produire les opérandes attendues par d'autres tâches (par exemple « centre de cercle »). Les tâches terminales doivent donc être placées au-dessus des tâches de production.

[1] Attention, *terminal* ne signifie pas feuille de l'arbre, car c'en est au contraire la racine.

Les niveaux d'abstraction du dialogue

Les tâches d'un système interactif supportant un dialogue structuré sont regroupées en différents niveaux d'abstraction. Les tâches constituant un même niveau d'abstraction rendent le même type de service à l'application. L'exemple «création de segment» est découpé en trois niveaux d'abstraction :

1. Le niveau de la création regroupe toutes les tâches terminales de création d'objets géométriques (segment, cercle ...).
2. Le niveau des tâches de calcul regroupe les tâches permettant d'extraire ou de calculer une caractéristique d'entités géométriques (calcul de l'intersection de deux droites).
3. Le niveau de la désignation regroupe les tâches permettant de sélectionner une entité géométrique en fonction de la position de la souris.

Une convention fréquente, pour permettre à l'utilisateur de changer facilement d'objectif en cours de définition d'une tâche structurée, est de considérer que l'activation d'une tâche interrompt la tâche du même niveau d'abstraction éventuellement en cours. Par exemple, une demande de création de segment « annulerait » une création de cercle en cours. Ce n'est pas le cas lorsqu'une tâche utilise les services d'une sous-tâche de niveau d'abstraction inférieur. Dans ce cas, l'état de la tâche supérieure ne change pas et reste au contraire en attente tant que la sous-tâche n'est pas terminée.

L'unité d'information produite et consommée

L'information qui transite de tâche productrice en tâche consommatrice, est caractérisée par son type. Elle est indépendante de son mode de production. Effectivement, dans l'exemple « création de segment », l'intersection de deux segments fournit le même type d'information que le centre d'un cercle. L'expression d'un dialogue structuré nécessite :

- l'identification des unités d'information indépendamment de leur mécanisme de production, de telles unités d'information sont fréquemment appelés des « jetons » (ou « token »),
- la description des tâches en termes de production/consommation et leur affectation à un niveau d'abstraction, ce qui définit la syntaxe choisie pour le langage de dialogue.

Représentation d'un dialogue structuré

Nous avons décrit dans cette partie les différents éléments qui font qu'un dialogue est structuré, c'est-à-dire qui permettent à un système de gérer des tâches structurées décrites en termes d'un arbre de buts/sous-buts au sens de Norman. L'expression d'un tel dialogue nécessite :

- l'identification des unités d'information indépendamment de leur de production,
- la description des tâches en termes de production/consommation,
- l'organisation des tâches en niveaux d'abstraction,
- la hiérarchisation des tâches et donc des différents niveaux d'abstraction.

Nous allons voir dans la section suivante comment ces différents éléments sont représentés dans le modèle H^4.

1.3.2.3 Le contrôleur de dialogue

La structure du contrôleur de dialogue de H^4 a été définie de façon à ce qu'il puisse supporter un dialogue structuré. Nous allons décrire maintenant les éléments de cette structure.

Les unités d'information utilisables par les tâches sont modélisées par des **jetons**. Les jetons sont classés par nature et contiennent une valeur dépendant de leur nature. On distingue deux grandes familles de jetons :

- les **jetons paramètres** contiennent les valeurs utilisées par les actions de l'application (par exemple, un jeton dont la nature est « position », représentant une position à l'écran, contient les deux réels représentant les coordonnées de cette position),
- les **jetons commandes** servent à exprimer les intentions de l'utilisateur. Leur nature correspond au nom générique de la tâche à réaliser. En général ils ne contiennent aucune valeur. Par exemple, le jeton dont la nature est créer_cercle peut être utilisé pour appeler n'importe quelle tâche de création de cercle.

Les tâches sont modélisées par des questionnaires qui correspondent à des signatures de fonctions. Un questionnaire est une fonction de l'adaptateur de présentation qui est associée à une liste de types de jetons paramètres qui constituent ses paramètres d'entrée et à un type de jeton paramètre en sortie s'il modélise une sous-tâche de production. Ces fonctions sont chargées de faire le lien entre le contrôleur de dialogue et les fonctions de l'application. Elles sont implantées dans l'adaptateur de noyau fonctionnel du modèle H^4 (voir Figure I.4). La tâche terminale *créer_segment* de l'exemple « création de segment » a, en entrée, deux jetons

positions. La sous-tâche *centre_de_cercle* a en entrée un jeton cercle qui véhicule un cercle de l'application et en sortie un jeton position dont les cordonnées sont celles du centre du cercle.

Pour regrouper des tâches en niveaux d'abstraction, le modèle d'architecture H^4 propose la notion d'**interacteur** de dialogue ou plus simplement interacteur [Guittet et Pierra 1993a, Guittet et Pierra 1993b]. Les interacteurs de dialogue sont différents des interacteurs d'entrée-sortie tels que les définissent Duke et Harrison [Harrison et Duce 1994], Paternò [Paternò 1994, Paternò et Faconti 1994] ou même ceux de Myers [Myers 1990] définis dans Garnet. En effet, les interacteurs de contrôle sont chargés de gérer la structure et l'appel des primitives du système, alors que les interacteurs d'entrée-sortie sont chargés de la gestion des échanges entre le système et l'utilisateur.

Les interacteurs servent à contrôler l'appel des questionnaires. Pour cela, chaque interacteur contient un automate de type réseau de transitions augmenté (RTA, ou ATN en anglais) tel que les définis Woods [Woods 1970]. Les RTA sont des automates à états finis qui contiennent la notion de registre, sorte de variable d'état de l'automate, visible uniquement de celui-ci. Le franchissement d'une transition peut être subordonné (on dit également gardé) à la valeur d'un registre. Des actions systématiques ou non, concernant ces registres, peuvent être associées au franchissement des transitions. Les automates servent à contrôler l'appel des questionnaires de l'application. Le registre conserve l'ensemble des jetons paramètres qui arrivent dans l'automate entre deux appels de questionnaire. Ces jetons sont fournis au questionnaire lors de son appel afin qu'il les utilise comme paramètres, ensuite le registre est vidé.

Dans les implantations déjà réalisées du modèle H^4 [Guittet 1995], chaque automate possède un état initial. Comme le dialogue des applications CAO est préfixé à cause de l'arité des tâches (voir § 1.3.1.1), les transitions à partir de l'état initial se font toutes sur des jetons commande. Lorsqu'un automate appelle un questionnaire, il revient ensuite dans l'état initial. La Figure I.6 montre un automate permettant d'appeler un questionnaire de création de cercle avec une position (le centre du cercle) et un nombre (le rayon du cercle) comme paramètres.

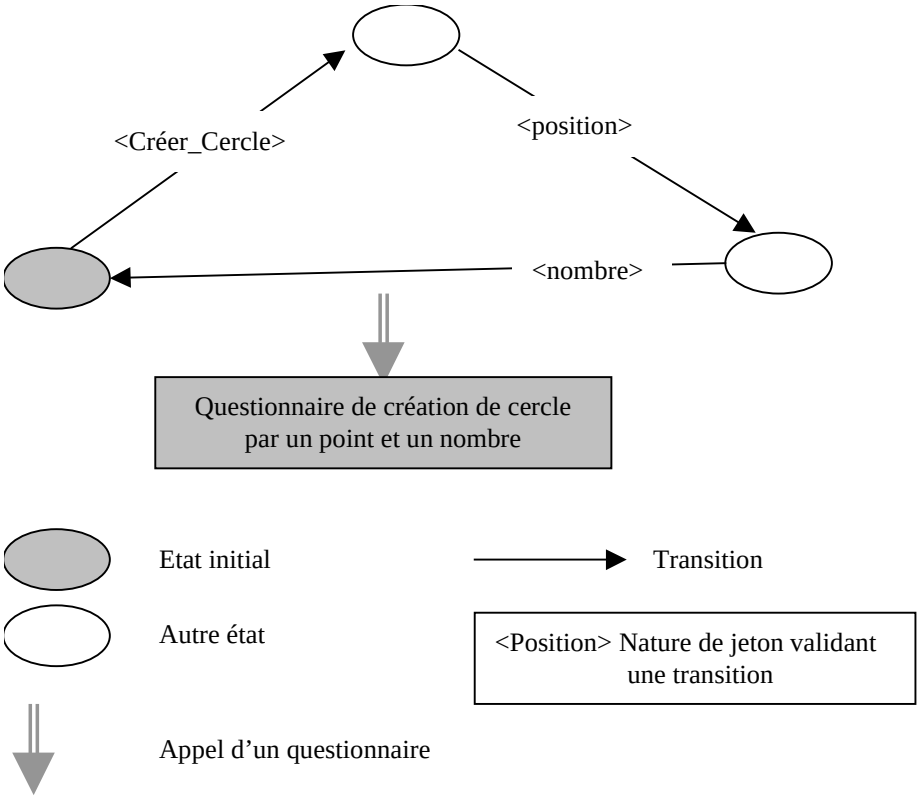


Figure I.6: RTA d'un interacteur

Les interacteurs sont utilisés pour regrouper les tâches en niveaux d'abstraction en fonction du service qu'elles rendent : création, calcul, interrogation etc… Dans l'exemple « création de segment » on peut choisir de distinguer trois niveaux d'abstraction :

1. l'interacteur de création contient le questionnaire *créer_segment*,
2. l'interacteur d'expression contient les questionnaires *Centre_de_cercle* et *Intersection_de_droites*,
3. l'interacteur de désignation contient les questionnaires *sélection_de_segment* ou *sélection_de_cercles*.

L'organisation hiérarchique des tâches est à la charge du **moniteur**. Cette hiérarchie se présente sous la forme d'une hiérarchie d'interacteurs. Ces derniers sont organisés de bas en haut en suivant l'ordre croissant de leur niveau d'abstraction. Le moniteur est chargé de récupérer les jetons venant de la présentation et de les transmettre aux interacteurs en suivant la hiérarchie. L'indépendance entre les tâches vient du fait qu'une tâche ou un questionnaire ne sait pas d'où vient le jeton que lui fournit le moniteur par l'intermédiaire de l'interacteur. Elle ignore aussi comment sera utilisé le jeton qu'elle produit. Le moniteur permet l'organisation hiérarchique des interacteurs, donc des questionnaires.

1.3.2.4 Exemple de fonctionnement

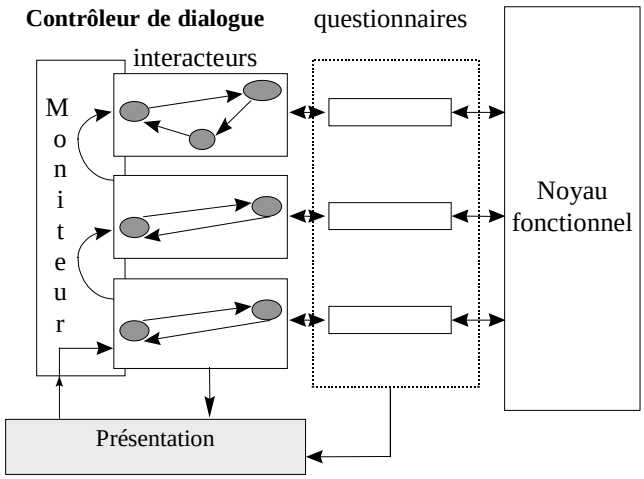


Figure I.7: fonctionnement du contrôleur de dialogue

Le mode fonctionnement (Figure I.7) consiste à véhiculer un jeton à travers la hiérarchie d'interacteurs. Le moniteur récupère un jeton venant de la présentation (ce jeton correspond à une entrée de l'utilisateur) et le transmet au premier interacteur de la hiérarchie (l'interacteur représentant les sous-buts du plus bas niveau de l'arbre des buts/sous-buts.). Si l'interacteur n'attend pas ce jeton, alors il le rend au moniteur. S'il est en attente de ce jeton, il le stocke et change d'état (on dit qu'il y a **consommation** du jeton). De plus, si ce jeton lui permet d'appeler un questionnaire, alors il rend au moniteur l'éventuel jeton produit par ce questionnaire (on dit qu'il y a **production** d'un jeton). Le moniteur transmet ainsi les jetons d'interacteur en interacteur jusqu'à ce qu'il n'y ait plus de production.

1.4 Synthèse sur les architectures

Les modèles d'architectures, qu'ils soient composés de macro-composants ou de micro-composants, préconisent tous de séparer le noyau fonctionnel, la présentation et le contrôle. Donc, si nous suivons ces recommandations, nous pouvons d'ors et déjà dire que pour définir ou pour spécialiser de manière interactive une application de conception technique, l'utilisateur devra, d'une part, définir de nouvelles primitives pour le noyau fonctionnel, et d'autre part, définir les moyens d'accéder à ces nouvelles primitives. Il faudra que le système permettant de définir de nouvelles primitives permette également que son dialogue ainsi que sa présentation soient modifiés.

Notre travail se base sur le modèle d'architecture H***4***. Nous l'avons choisi pour deux raisons :

- la première est qu'il répond tout à fait aux besoins qui résultent des caractéristiques des applications de conception technique, telles que les applications de CAO,
- la seconde est qu'il appartient à la catégorie des modèles hybrides, et définit donc non seulement le découpage de l'application en macro-composants, mais en plus, il définit parfaitement leur rôle et la façon de les concevoir.

Nous disposons donc d'un modèle d'architecture permettant de définir quelle devra être la structure des éléments à générer pour modifier ou spécialiser un système de CAO. En particulier, en ce qui concerne le dialogue, la définition de nouveaux objets, associés à de nouvelles primitives, nécessitera la modification des jetons et des interacteurs de contrôle.

2 Conception d'interface

La conception de l'interface d'une application représente une grande part du travail de développement. Une étude a montré qu'en moyenne 48 % du code d'une application est dédié à l'interface utilisateur et que 50 % du temps d'implémentation est pris par la création de l'interface [Myers 1995]. Pour que la charge de travail des concepteurs d'application graphique interactive soit moins importante, un grand nombre d'outils d'aide au développement de l'interface ont été créés depuis une quinzaine d'années. Ces outils suivent l'une des deux grandes approches utilisées pour la conception d'interfaces [Coutaz 1990] :

- d'une part, l'approche ascendante qui part de la description de la couche de présentation pour aller vers le contrôle,
- d'autre part, l'approche ascendante qui part de la description du contrôleur de dialogue pour générer tout ou partie de la présentation.

Cette section décrit et compare les deux approches. Puis nous discuterons une nouvelle approche, appelée approche mixte, qui constitue un mélange des deux approches précédentes.

2.1 Approche ascendante

L'approche ascendante consiste à créer la couche de présentation de l'application à l'aide d'éléments de présentation tels que des fenêtres, des boutons, des menus, etc. Ensuite, le développeur relie ces éléments de présentation avec les actions du noyau fonctionnel par un mécanisme de fonction de rappel (le terme anglais « call-back » est très fréquemment utilisé pour désigner les fonctions de rappel). Les fonctions de rappel sont des fonctions qui sont invoquées lorsque l'utilisateur interagit avec un élément de présentation. Par exemple, une fonction de rappel est appelée lorsque l'utilisateur clique sur un bouton. Les éléments de

présentation appelés communément « widget », pour « window + gadget », sont regroupés dans des boîtes à outils. Il en existe un nombre important. Certaines d'entre elles sont présentées dans la section suivante.

En plus des boîtes à outils, un grand nombre d'outils de conception d'application interactive qui se basent sur l'approche ascendante sont proposés. Les squelettes d'applications fournissent aux concepteurs un cadre général pour définir leur interface, ils doivent y ajouter les éléments spécifiques à l'application finale. Les générateurs d'interfaces permettent d'instancier la présentation de manière graphique et permettent, dans la plupart des cas, de gérer l'ensemble d'un projet.

2.1.1 Les boîtes à outils

Cette section décrit les boîtes à outils au sens de Arch. Nous commençons par définir les boîtes à outils de manière générale. Puis, nous expliquons les mécanismes de contrôle inclus dans les boîtes à outils afin de voir comment elles permettent de contrôler tout ou partie d'une application. Ensuite, nous exprimons les besoins des systèmes de conception technique en terme d'affichage. Enfin, nous décrivons les services d'affichage d'objets géométriques offerts par deux boîtes à outils différentes : Tcl/Tk [Ousterhout 1994] et les MFC [Procise 1999].

2.1.1.1 définition

Une boîte à outils (toolkit en anglais) est une bibliothèque d'objets d'interaction : les widgets. La présentation de l'application est créée en instanciant des widgets comme les fenêtres, les menus, les boutons etc… Les widgets ont deux rôles importants dans l'application :

- ils sont chargés de transformer les informations venant des périphériques d'entrées (clavier, souris, micro…) en données utilisables par l'application,
- ils offrent des services pour gérer les périphériques de sortie (écran, enceintes, périphérique à retour d'efforts …), permettant ainsi au système de rendre compte de son état.

Les boîtes à outils présentent plusieurs avantages. D'abord, elles permettent la réutilisation d'outils d'interaction de plus haut niveau d'abstraction que les procédures fournies par les pilotes des périphériques d'entrées-sorties. Ceci facilite l'écriture de la couche de présentation de l'interface de l'application. Ensuite, elles standardisent l'aspect (look) de toutes les applications construites avec la même boîte à outils ainsi que le comportement des objets de présentation (feel).

On trouve deux grandes méthodes pour décrire la couche de présentation en agençant des widgets :

- La première méthode est purement statique et consiste à décrire la présentation de l'interface à l'aide d'un langage de description dédié. Les fichiers ainsi créés sont éventuellement compilés, puis, dans tous les cas, le résultat est interprété au lancement de l'application pour créer la couche de présentation. Les langages de description, contrairement aux langages de programmation structurés, ne supportent pas les structures de contrôles comme les itérations et les conditionnelles. Par contre, ils permettent de décrire facilement et très clairement des compositions de widgets. Cette simplicité facilite la génération de la description textuelle de l'interface à partir d'une description graphique, c'est ce que font les générateurs d'interface. Par contre, les widgets créés par cette méthode sont difficilement modifiables dynamiquement par l'application car, au moment de leur création, le programme de l'application ne peut pas y accéder facilement. Par exemple, le langage UIL permet de décrire une interface avec les widgets de X-Motif [Heller et Ferguson 94]. Avec ce type de langage, la description des widgets et des fonctions de rappel est séparée car les fonctions de rappel sont toujours décrites avec un langage de programmation. Il est tout de même intéressant de noter que des environnements de programmation comme Visual C++ [Microsoft 1999], qui associe un générateur d'interface et un système de gestion de projet, permettent d'accéder facilement à posteriori aux éléments de présentation créés statiquement.
- La seconde méthode consiste à décrire les widgets et les fonctions de rappel dans le même programme dans le langage de programmation. Cette méthode est la seule permettant d'instancier dynamiquement les widgets de l'application.

Dans la suite de cette section, nous nous intéresserons à deux aspects des boîtes à outils qui concernent la création d'application de conception technique : le contrôle et l'affichage.

2.1.1.2 Le contrôle de l'application

Comme nous l'avons vu dans la section précédente, les boîtes à outils sont chargées de transformer les interactions de l'utilisateur en données utilisables par l'application. Pour cela l'interface instancie un certain nombre de widgets qui permettent à l'utilisateur d'interagir avec l'application.

Les interactions de l'utilisateur sont transformées par le programme de scrutation associé à une boîte à outils en événements représentant l'action de l'utilisateur sur un widget. Lorsqu'un

widget reçoit un événement, il a deux types de comportement : un comportement « interne » et un comportement « externe ».

Le comportement interne représente une réaction propre d'un widget à un événement. Par exemple, lorsque que l'utilisateur appuie sur le bouton de la souris et que le pointeur se trouve sur un bouton, ce dernier « s'enfonce ». En général, le comportement interne des widgets d'une boîte à outils est peu ou pas modifiable. On remarque que certains widgets n'ont souvent qu'un comportement interne (les bouton-menus ne servent qu'à faire apparaître un menu). Ces widgets ne servent que localement dans la présentation et n'ont aucun lien ni avec le contrôle du dialogue ni avec la sémantique de l'application. Il n'ont qu'un rôle esthétique et/ou opératoire.

Le comportement externe représente la réaction de l'application à une action de l'utilisateur. Ce comportement doit être programmé par le concepteur. Pour cela, il active les fonctions de rappels (ou Call-Backs en anglais). Les Call-Backs sont des fonctions qui sont appelées lorsqu'un événement arrive sur un widget. Ils sont différents pour chaque widget et pour chaque événement. Grâce à ce mécanisme, les boîtes à outils permettent de contrôler l'application. En effet, le programmeur peut appeler une fonction du noyau fonctionnel de l'application par l'intermédiaire d'une fonction de rappel. Ceci montre le double rôle des boîtes à outils qui, outre leur rôle de présentation, permettent d'assurer le contrôle de l'application.

De gros problèmes peuvent survenir lorsque la même fonction de rappel doit faire un travail différent selon l'état de l'application. Ainsi, un clic souris sur la zone graphique peut avoir plusieurs significations. Par exemple, il peut servir à désigner un objet géométrique ou bien, une position servant de paramètre pour la construction d'un cercle. Il apparaît alors qu'un très grand nombre d'informations concernant le contrôle de l'application doivent être conservées au niveau de la présentation. De ce fait, le code à écrire grossit de manière importante lorsqu'une nouvelle fonctionnalité est ajoutée à l'application. De plus, l'expérience montre que les interfaces des applications réelles possèdent souvent des centaines de fonction de rappel, ce qui rend le code difficile à modifier et à maintenir [Myers et Rosson 1992]. C'est la raison pour laquelle, très tôt, les modèles d'architecture de systèmes interactifs ont préconisé de séparer la présentation et le contrôle. Nous avons vu, en particulier, comment le modèle H^4 utilisait des automates au niveau du contrôleur de dialogue pour mémoriser l'état du dialogue, et donc l'interprétation qu'il convenait de faire des événements émis par la présentation.

2.1.1.3 Affichage géométrique

La conception d'une application CAO implique l'utilisation de primitives graphiques capables de représenter les éléments géométriques du noyau fonctionnel. J.D. Fekete [Fekete 1996b] identifie trois modèles graphiques dans les boîtes à outils : Pixellaire, vectoriel 2D et 3D.

Le modèle pixellaire représente les objets graphiques comme un ensemble de points indépendants les uns des autres. Une fois créés, les objets graphiques n'ont pas d'existence propre dans l'application, ils n'existent que dans le « buffer » vidéo de l'ordinateur. Ils ne peuvent en aucun cas être modifiés. Ainsi, pour modifier l'affichage d'un objet du noyau fonctionnel, tous les objets graphiques doivent être effacés puis réaffichés en tenant compte des modifications sur les objets du modèle. Cela entraîne des problèmes de rapidité pour la manipulation directe puisqu'à chaque mouvement de souris, tous les objets graphiques sont effacés puis réaffichés. De plus, ce modèle ne permet pas de désigner les objets graphiques directement dans la boîte à outils. La désignation d'un objet doit être confiée au noyau fonctionnel. Les modèles pixellaires sont présents dans toutes les boîtes à outils.

Dans **le modèle vectoriel**, les objets graphiques sont considérés comme des widgets. Ils peuvent être modifiés puisqu'ils existent réellement sous forme d'objets instanciés dans le module de présentation. Leur modification est gérée directement par le module de présentation sans que l'application n'ait à les retracer. De plus, le programmeur peut leur associer un comportement particulier grâce aux fonctions de rappel. Les modèles vectoriels facilitent la conception des systèmes CAO car le programmeur peut faire une association entre les objets du modèle et les objets graphiques qui les représentent. Cela permet de simplifier l'écriture du code de la désignation et celui du réaffichage lors de la modification d'un objet du modèle si la présentation de l'objet n'est pas dépendante des autres objets. Ce modèle vectoriel permet également d'utiliser la manipulation directe sans que la vitesse de réponse ne soit trop lente puisque certains comportements peuvent être instanciés directement dans la couche de présentation.

Toutes les boîtes à outils ne se basent pas sur un modèle vectoriel. On peut cependant citer GKS [GKS 1985], Phigs [PHIGS 1989] (qui ne sont quasiment plus utilisées maintenant), Tcl/Tk que nous étudierons un peu plus tard et Amulet [Myers, et al. 1997].

Les **modèles 3D** constituent le prolongement 3D des modèles vectoriels. Ils permettent d'afficher des objets 3D sans que le programmeur n'ait à décrire toute la topologie de l'objet à afficher. Ces modèles sont bien sûr de plus en plus utilisés pour les applications de CAO mais

aussi pour les logiciels permettant de faire de l'animation comme 3D Studio de Autodesk [Autodesk 1992], ou pour les jeux. Les plus connus de ces modèles sont OpenGL [Neider, et al. 1993], Glide et DirectX [Bargen et Donnelly 1998]. Malgré leur confort d'utilisation comparé au codage de la visualisation de formes 3D à partir d'objets 2D, ces modèles ne permettent pas, en général, au programmeur d'accéder à la topologie de l'objet. Le problème de la désignation des objets structurés (voir § 1.3.1.4) reste alors entier, la désignation devant être réalisée au niveau du noyau fonctionnel puisque c'est là que la topologie est connue. Ce problème est cependant résolu dans certains systèmes comme CAS.CADE [CAS.CADE 2000] dont les objets graphiques connaissent leur topologie ce qui permet de déléguer la désignation des faces, des arêtes ou des sommets au niveau de la couche de présentation.

Pour illustrer notre propos sur les boîtes à outils, nous allons en décrire deux qui nous semblent particulièrement intéressantes de par les concepts qu'elles mettent en œuvre et qui, toutes deux mais à des titres divers, facilitent la création de la présentation d'un système CAO.

2.1.1.4 TCL/TK

Deux points semblent devoir être notés concernant cette boite à outils largement utilisée, d'une part, la facilité d'utilisation et d'apprentissage de Tcl/Tk, et d'autre part, les possibilités graphiques importantes offertes par cette boite à outils.

Tcl/Tk regroupe la boîte à outils Tk pour (Tool Kit) et le langage Tcl (Tool Command Language) qui permet d'instancier les éléments de Tk. Tk est une boîte à outils standard qui possède un grand nombre de widgets et qui gère les entrées de l'utilisateur avec des événements et des fonctions de rappel. Tcl est un langage de script interprété au même titre que le shell Unix ou que Perl [Wall, et al. 1996]. Il existe des interpréteurs Tcl pour de nombreuses plate-formes (Windows, Mac, Unix). Une application Tcl peut donc être exécutée sur ces environnements sans que le code soit modifié.

Tcl/Tk est très répandu dans le monde universitaire car il permet de créer rapidement une interface pour une application et son apprentissage est assez aisé. Cette facilité d'utilisation et d'apprentissage vient du fait que Tcl est un langage de script. Les langages de script, contrairement aux langages de programmation, sont faiblement typés. Tcl ne manipule que des chaînes de caractères, il n'y a pas de restrictions sur la manière d'utiliser les variables et tous les composants ainsi que les valeurs sont représentés de manière uniforme. Pour donner un exemple de cette facilité d'utilisation comparons la création d'un bouton avec Tcl/Tk et C++/MFC :

Button .b – text Hello –height 20 –width 50

Cette commande Tcl crée un nouveau bouton avec le texte Hello. Ce bouton a une largeur et une longueur déterminées par les attributs *height* et *width.*

Pour créer le même objet avec C++/MFC, le code est plus conséquent car le programmeur doit absolument donner une valeur à tous les paramètres du constructeur du bouton.

*CButton *B1= new CButton();*

CRect r (0,0,20,50);

B1->Create (Name,BS_PUSHBUTTON|WS_VISIBLE,r,this,250);

D'après notre expérience de l'utilisation de Tcl/Tk, les trois principaux avantages de cette boite à outils sont:

1. un code compact et assez court,
2. un code facile à comprendre car seuls les éléments nécessaires sont décrits,
3. un apprentissage très rapide : en effet, un développeur n'a besoin de comprendre que peu d'options pour commencer à créer une application. Au fur et à mesure que ses connaissances augmentent, il peut améliorer son interface en décrivant de plus en plus finement les différents widgets.

La Figure I.8 montre l'interface de Ebp [Potier 1995] que nous avons réalisé à l'aide de Tcl/Tk.

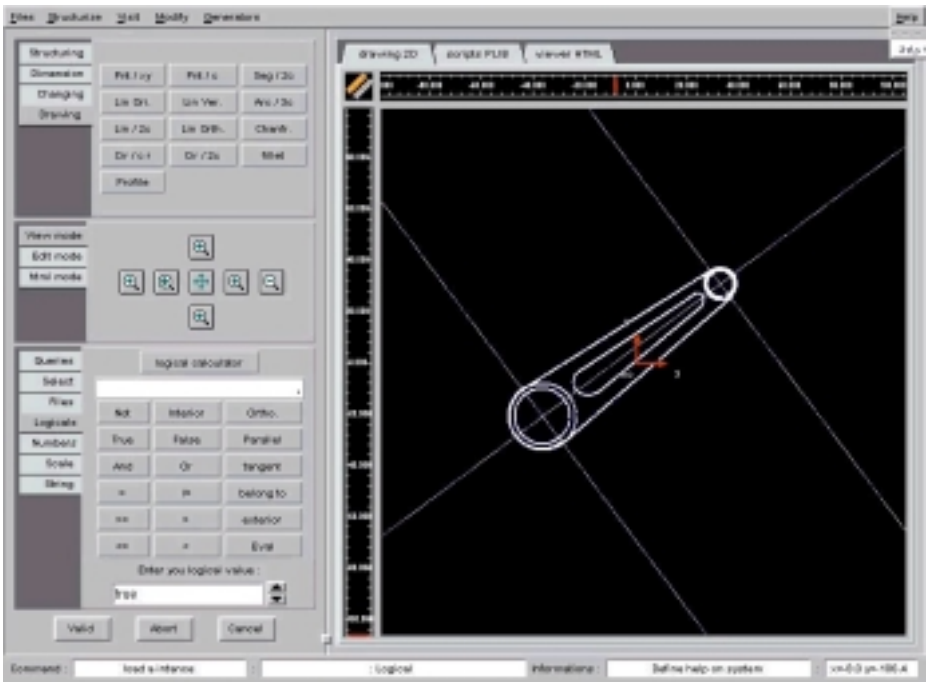


Figure I.8: Exemple d'interface avec Tcl/Tk

Un deuxième point important de la boîte à outils Tcl/Tk pour le domaine de la CAO est qu'elle offre de grandes possibilités pour la création de dessins en deux dimensions. Tcl/Tk se base sur un modèle vectoriel 2D. Chaque objet graphique a une existence réelle, il peut être sélectionné, modifié, ou détruit indépendamment des autres. Cela entraîne une grande facilité d'écriture de procédures de manipulation directe pour les objets du noyau fonctionnel lorsque ceux-ci ne sont ni relationnels, ni structurés. Leurs présentations graphiques peuvent être considérés comme des widgets de la boîte à outils. Tcl/Tk permet de leur associer des fonctions de rappel. Par exemple, ils peuvent réagir à un clic souris facilitant ainsi leur désignation. Enfin, dernier point, mais non le moindre, les objets graphiques peuvent être marqués. Le marquage se fait par l'attribut « tag » suivi d'une liste de chaînes de caractères représentant par exemple le type, ou l'identité de l'objet de l'application représenté par l'objet de présentation. Prenons l'exemple d'une application qui permet de créer des roues et des volants. Ces deux classes d'objets sont représentées dans l'interface par des cercles. Ces cercles peuvent être marqués avec les tags « roue » et « volant » selon ce qu'ils représentent. Une partie de la sémantique des objets de l'application est ainsi représentée au niveau la couche de présentation et non plus uniquement au niveau du noyau fonctionnel. Cela permet de différencier, au niveau de l'interface, la sélection des roues et des volants sans faire appel au noyau fonctionnel. Ceci permet également de différencier les comportements en testant la valeur de cet attribut. Il s'agit donc d'un mécanisme facilitant la désignation sémantique.

2.1.1.5 MFC

Les MFC (Microsoft Foundation Classes) constituent un deuxième exemple et représentent l'une des boîtes à outils les plus utilisées aujourd'hui. Elle se base essentiellement sur le langage de programmation C++. Comme la plupart des boîtes à outils, elle est constituée d'un ensemble de widgets de présentation qui répondent aux interactions de l'utilisateur par des fonctions de rappel.

Contrairement à Tcl/Tk, les MFC n'offrent au programmeur d'interface que peu de possibilités graphiques. En effet, les MFC n'offrent qu'un modèle graphique 2D pixellaire, ce qui est un inconvénient lorsque l'on veut réaliser une application de CAO où les besoins graphiques sont importants. Par contre, un avantage non négligeable de cette boîte à outils est qu'elle offre la possibilité de s'interfacer assez simplement avec de nombreux modeleurs graphiques. Ceci n'est pas le cas de la plupart des autres boîtes à outils, qui même lorsqu'elles peuvent intégrer des widgets externes, exigent l'interfaçage avec des visualiseurs spécialisés ce qui augmente considérablement le travail de conception.

2.1.1.6 MFC+CAS.CADE

Le couplage entre les MFC et la partie graphique de la bibliothèque CAS.CADE offre un dispositif répondant parfaitement aux besoins des concepteurs de systèmes CAO en termes de boîte à outils. Le programmeur dispose dans ce cas, des nombreuses possibilités offertes par les MFC en termes de widgets, et d'un environnement graphique complexe permettant d'afficher des formes en 2 ou 3 dimensions. Cette partie graphique est une sur-couche d'OpenGL.

La partie graphique de CAS.CADE, et notamment l'AIS (Application Interactive Services), permet de visualiser un modèle B-Rep en instanciant des AIS_Shape. Les AIS_Shape sont des objets graphiques qui connaissent leur structure interne (faces, arêtes et sommets). La désignation d'un élément de cette structure peut donc être effectuée au niveau de la couche de présentation sans faire appel au noyau fonctionnel. Cependant, contrairement à Tcl/Tk, CAS.CADE n'offre pas la possibilité d'enrichir simplement le modèle des objets graphiques. Ainsi, les objets graphiques de CAS.CADE ne permettent pas de faire la différence entre deux cercles qui représentent des éléments différents de l'application.

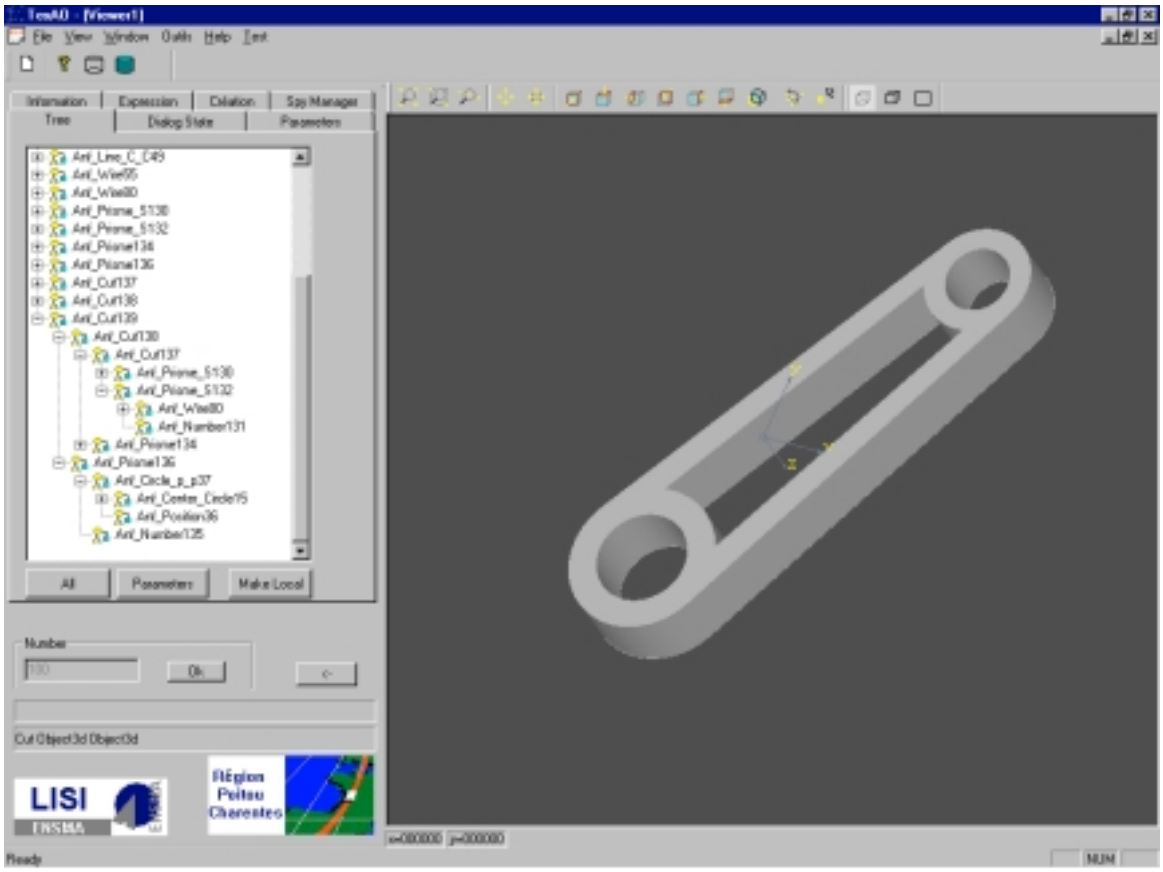


Figure I.9: Utilisation des MFC et de CAS.CADE

La Figure I.9 montre un exemple d'application dont l'interface a été réalisée avec les widgets des MFC et la bibliothèque graphique de CAS.CADE.

2.1.2 Les squelettes d'application

Les boîtes à outils fournissent aux développeurs des objets de présentation qui ont un comportement interne et un comportement externe. La programmation du comportement externe des widgets est laissé à l'entière charge du développeur. Cependant, on s'aperçoit que dans de nombreux cas, une partie du comportement externe des widgets doit toujours exister (par exemple, dans les systèmes CAO, faire un zoom ou déplacer une vue). Avec une simple boîte à outils, pour chaque nouvelle application que le concepteur créera, il devra réécrire presque à l'identique cette partie du code. De plus, la plupart des applications d'un domaine se basent sur la même architecture. Or, pour chaque nouvelle application, cette architecture doit être redéfinie.

Il paraît alors intéressant de définir, une fois pour toutes, les éléments que l'on retrouve systématiquement dans les applications d'un certain domaine ; c'est la notion de squelette

d’application. Par exemple, un squelette d’application pour les systèmes CAO proposera des fenêtres de visualisation d’éléments 3D. Il associera à ces fenêtres des boutons permettant de modifier la visualisation (rotation, translation, changement de la couleur du fond etc…). Le développeur n’aura alors plus qu’à ajouter le code vraiment spécifique à l’application qu’il veut créer.

L’utilisation des squelettes d’application facilite le développement d’une plus grande partie de l’interface que les boîtes à outils puisqu’ils décrivent une partie du comportement externe des widgets et donnent un cadre de développement au concepteur. Ce dernier peut alors se concentrer sur la conception des parties spécifiques de l’application. De plus, cette méthode garantit une homogénéité élevée au niveau du « look and feel » des applications créées à partir d’un même squelette. Au total, on estime qu’un squelette d’application, lorsqu’il est bien adapté au domaine visé, diminue le temps de développement de l’interface d’un facteur quatre à cinq par rapport aux seules boîtes à outils [Shumerck 1986].

Bien sûr, l’utilisation d’un squelette d’application nécessite cependant que le concepteur connaisse la boîte à outils sous jacente (il sera certainement amené à ajouter des widgets) et l’architecture sur laquelle le squelette est basé. De plus, les squelettes d’applications sont par nature stéréotypés, ce qui limite leur domaine d’application. Plus le squelette d’application est proche de l’application finale, moins le développeur a de travail, mais moins ce squelette est adaptable à un grand nombre d’applications (Figure I.10).

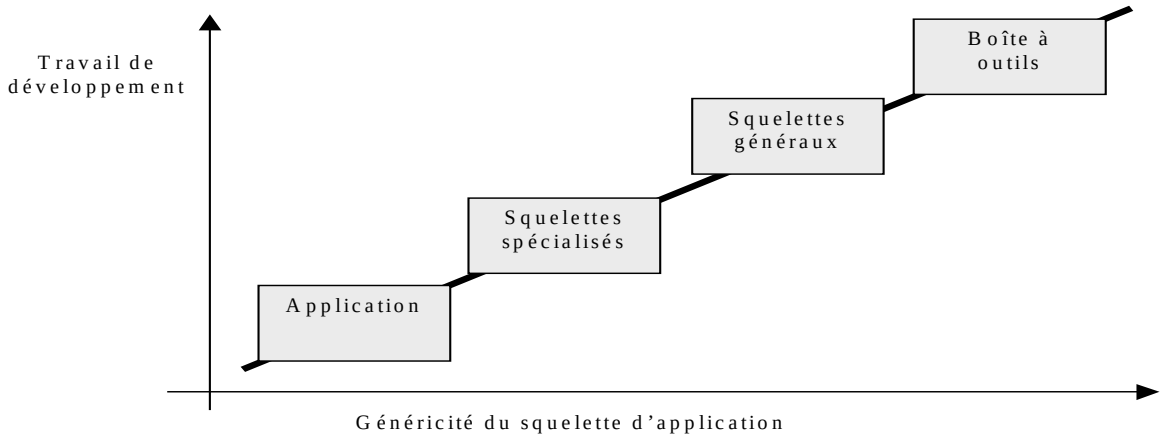


Figure I.10: rapport entre le travail de développement et les squelettes d'application

La Figure I.10 montre également le rapport entre le temps de travail et la généricité des squelettes d’application. Les squelettes d’application généraux désignent ceux pour qui le type d’application visé n’est pas défini. Ils instancient, en général, la gestion du système de

fenêtrage sans spécifier le contenu de ces fenêtres. Les squelettes spécialisés connaissent le contenu des fenêtres qu'ils gèrent. Ils sont difficilement adaptables à l'affichage d'une autre forme de données (par exemple un squelette d'application permettant d'afficher du texte ne peut pas afficher d'éléments géométriques en trois dimensions).

2.1.3 Les générateurs ascendants

Les générateurs ascendants, plus connus sous le nom de générateurs d'interfaces ou GUI-Builders « Graphic User Interface Builders », sont des logiciels qui permettent aux concepteurs de créer facilement la couche de présentation de façon graphique. Pour cela, le concepteur dessine l'interface qu'il désire réaliser pour son application à partir de primitives graphiques représentant les widgets de la boîte à outils sous-jacente ou à partir d'un dessin « main levée » représentant un brouillon de l'interface comme dans SILK [Landay et Myers 1995]. A partir de cette description le GUI-Builder génère le code pour instancier les widgets composant l'interface dessinée. La plupart des générateurs ascendants décrivent la présentation dans un langage qui leur est spécifique. Le fichier qui contient les ressources est alors interprété au moment de l'exécution. Par exemple, le générateur de X-Motif (VUIT) crée un fichier UIL qui, après avoir été compilé, sera interprété par l'application. Ce type d'application est bien au point et existe pour quasiment toutes les boîtes à outils : Visual C++ et Visual Basic de Microsoft pour les MFC, VUIT pour X-Motif, Tk-builder pour Tcl/Tk … . On remarque cependant que certaines applications n'utilisent pas forcément de langage de description comme XXL [Lecolinet 1999] qui gère directement la description des widgets en C ou C++, et les générateurs de Tk qui utilisent directement Tcl.

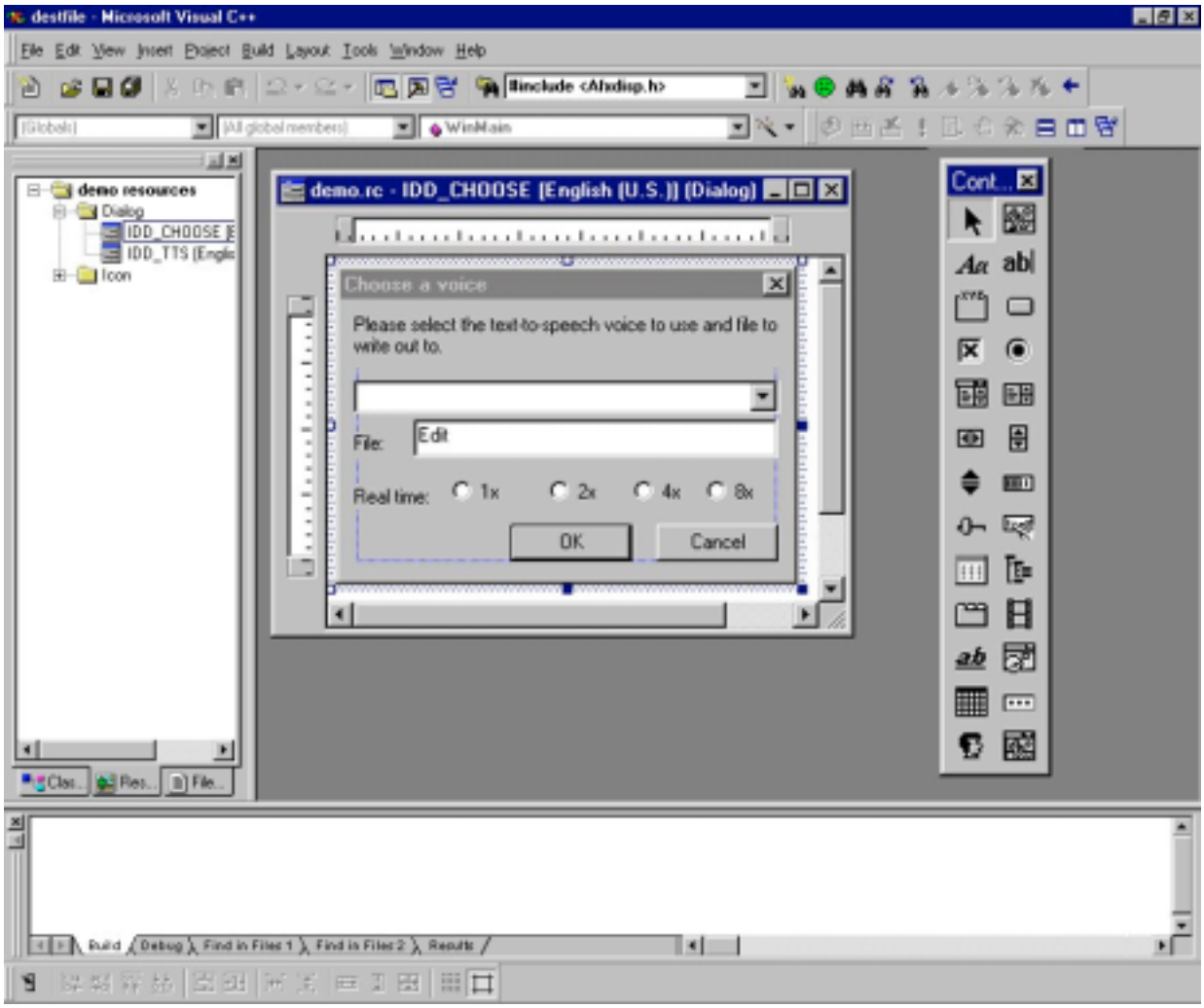


Figure I.11: Exemple de GUI-Builder (Visual C++)

La plupart des GUI-Builders sont intégrés à un environnement de programmation qui permet de définir les fonctions de rappel et d'en générer l'entête. Le concepteur n'a alors plus qu'à lier la présentation au reste de l'application dans le corps des fonctions de rappel.

Il est intéressant de noter que ces environnements de création d'interfaces intègrent de plus en plus souvent des squelettes d'application prédéfinis. Leurs utilisations allègent la charge de travail du développeur car des fonctions très générales comme le rafraîchissement des vues ou l'organisation des fenêtres, sont déjà entièrement codées. Par exemple, dans Visual C++, il existe différents squelettes d'applications, appelés « Wizzard » (magicien), permettant de créer de nouveaux widgets par agrégation ou des applications mono ou multi-fenêtres.

2.1.4 Conclusion

Les outils que nous avons présentés ici permettent de créer facilement la couche de présentation d'une application. Ils déchargent le concepteur de l'écriture d'une grande partie du programme

nécessaire puisqu'ils prennent en charge la gestion des outils d'entrées/sorties. Par exemple, la création directe d'un pilote gérant une souris posséderait en général des centaines de lignes de programme. Cette programmation est déjà effectuée dans la boîte à outils. L'utilisation d'outils comme les GUI-Builders facilite encore plus le travail puisque l'aspect externe de l'interface n'est pas créé de façon textuelle avec un langage de programmation ou de description, mais graphiquement et par manipulation directe, en respectant le principe « What You See Is What you Get » WYSIWYG (ce que vous voyez est ce que vous obtenez).

La méthode de conception ascendante, introduite par le type d'outils présenté précédemment, consiste à créer d'abord la couche de présentation de l'application, en instanciant les widgets d'une boîte à outils qui composeront les éléments de cette présentation. La présentation est ensuite liée aux autres composants de l'application par l'intermédiaire des fonctions de rappel. Cette méthode de conception est très utilisée car, comme nous l'avons vu précédemment, elle dispose de nombreux outils et elle est générique et peut être utilisée pour tous les types d'applications.

Le principal inconvénient de cette approche est que les outils ne permettent pas de créer un contrôleur de dialogue explicite. En général, les développeurs lient les actions du noyau fonctionnel à la présentation directement par l'intermédiaire des fonctions de rappel. Or, cette méthode de programmation devient très difficile à utiliser dans le cas d'un dialogue structuré comme celui des systèmes CAO. En effet, dans un dialogue structuré, les actions à réaliser lors de l'appel de chaque fonction de rappel dépendent largement de l'état du système. Celui-ci doit donc être modélisé au niveau de la présentation. Par exemple, la fonction de rappel appelée lors d'un clic sur une zone de dessin peut servir à désigner un objet, à créer un point utilisé lors d'une construction ou bien encore à calculer une distance qui sera elle même utilisée pour un autre calcul. Le widget doit donc décider quelle interprétation est pertinente. Ainsi, plus le dialogue de l'application est important, plus le code écrit dans les fonctions de rappel est important. Cela rend l'application de moins en moins fiable et de plus en plus difficile à mettre au point, à maintenir et/ou à modifier.

Au total, les boîtes à outils et les générateurs ascendants sont de bons outils pour créer la présentation d'une application graphique interactive, mais il ne permettent pas de décrire facilement la logique du dialogue et en particulier celles des dialogues structurés des systèmes CAO. Nous proposerons dans le chapitre II d'ajouter aux boîtes à outils existantes de nouveaux outils permettant de contrôler le dialogue. Cette extension permettra alors à un concepteur de définir le dialogue de son application de la même manière qu'il définit la présentation.

2.2 Approche descendante

L'approche descendante part au contraire du contrôleur de dialogue. Elle consiste à décrire le dialogue de l'application à l'aide d'un langage spécifique. Cette description permet alors de générer la couche de présentation de l'application. Dans cette section, nous définissons plus précisément ce que sont les générateurs descendants et en particulier, les systèmes « basés sur modèle ». Puis nous en étudierons deux plus particulièrement : Gipse [Patry et Girard 1999] et LSI [Guittet 1995] qui semblent les mieux adaptés à la création d'applications CAO.

2.2.1 Principe de fonctionnement

La notion de générateur descendant a vu le jour au début des années 80. Elle propose aux développeurs de décrire l'interface de l'application dans un langage spécialisé de haut niveau d'abstraction. Cette spécification est ensuite traduite en programme exécutable ou interprétée pour générer l'interface (contrôle et présentation). Les outils utilisant cette approche sont connus sous le nom de Système de Gestion d'Interface Utilisateur (SGUI ou UIMS : User Interface Management Systems en anglais [Olsen 1992]). Les premiers d'entre eux ne s'occupaient principalement que de la spécification du dialogue de l'application [Green 1986]. Ils utilisaient des réseaux de transitions [Jacob 1986], des grammaires [Olsen 1983, Olsen 1986] ou des représentations des évènements [Singh et Green 1991] pour spécifier la réponse du système aux évènements de la couche de présentation. Au début des années 90, les langages de spécification devinrent plus sophistiqués, prenant en compte de plus en plus d'éléments, tant de dialogue que de présentation, et permettant de générer des interfaces de plus en plus évoluées. Les systèmes actuels utilisent de multiples descriptions : celle des tâches utilisateur, celle de la structure et des relations des informations manipulées par l'application, celle de la couche de présentation, etc.

Les outils sont désormais connus sous le nom de systèmes *basés sur modèles* [Szekely 1996] (Model Based Systems (MBS) en anglais). Angel Puerta [Puerta 1996], les définit ainsi : *« Les systèmes basés sur modèles sont basés sur la thèse selon laquelle il est possible de définir de façon déclarative toutes les caractéristiques importantes d'une interface utilisateur. Un environnement de génération d'interface, automatique et compréhensif, peut alors être fabriqué autour de ce modèle. »*

Les MBS permettent donc de construire une description de l'application qui sera ensuite utilisée pour créer l'interface de cette application. On désigne, en général, cette description sous

le terme de modèle d'où le nom de systèmes basés sur modèle. Le modèle peut être constitué d'un ou plusieurs modèles particuliers comme le modèle des tâches, le modèle de la présentation, etc. Les MBS possèdent, en général, plusieurs types d'outils différents comme des générateurs d'interfaces, des générateurs d'aides pour l'utilisateur final, des outils d'évaluation ergonomique de l'interface, des outils d'aide à la conception de l'application, etc.

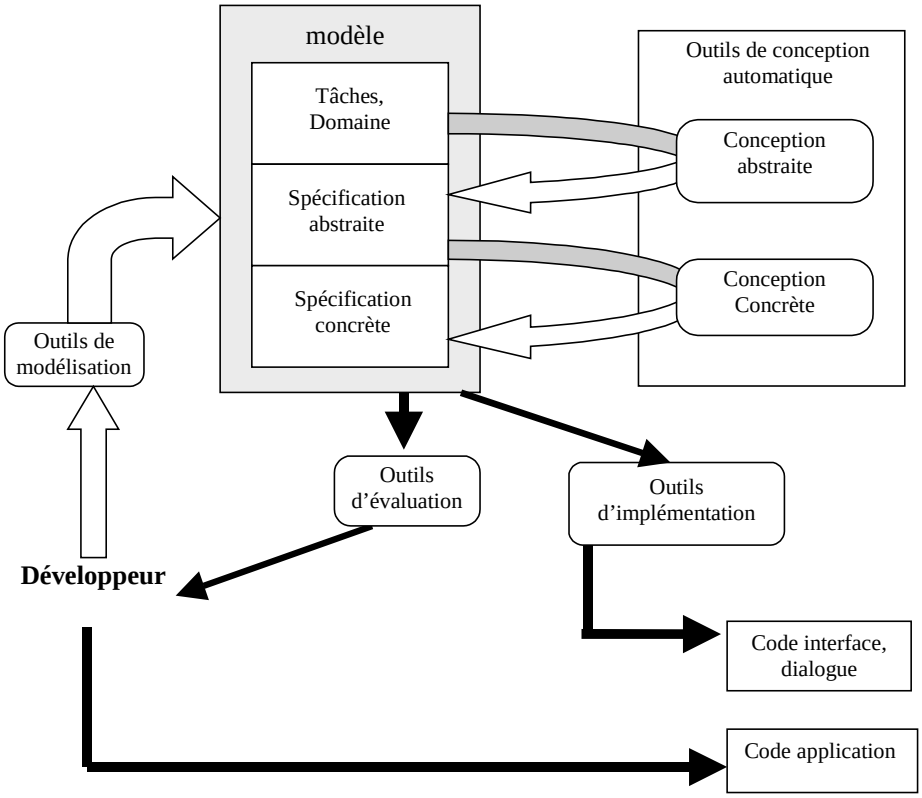


Figure I.12: Processus de développement avec un MBS

La Figure I.12 montre, d'après [Szekely 1996], le processus de développement d'une application à l'aide d'un générateur descendant. Les rectangles arrondis représentent les différents outils, les autres rectangles représentent les informations consommées ou produites par les outils. Le développeur utilise les outils de modélisation pour créer le modèle. Les outils de conception automatique sont utilisés pour effectuer certaines transformations sur le modèle. Leur utilisation est, soit obligatoire, soit laissée à la discrétion du développeur. Les outils d'implémentation génèrent le code de l'interface, soit sous forme d'un langage interprété par un interpréteur spécifique, soit sous la forme de code qui devra être compilé. Dans la suite, nous donnons le détail des composants les plus importants d'un système basé sur modèle. Il est cependant intéressant de noter que si tous les générateurs descendants se basent sur une représentation du dialogue pour générer l'interface, tous ne possèdent pas la totalité des outils présentés sur la Figure I.12, et détaillés ci-dessous.

2.2.1.1 Le modèle

Le terme modèle désigne une description des caractéristiques de l'interface d'une application interactive. Le modèle est donc le composant principal des MBS. Il peut contenir des descriptions du dialogue, du domaine et de la présentation. Il peut être décomposé en trois niveaux d'abstractions [Szekely 1996].

Le **modèle du domaine** et le **modèle des tâches** se trouvent au niveau d'abstraction le plus élevé. Le modèle du domaine représente les données et les fonctions supportées par l'application. Le modèle des tâches représente les tâches que l'utilisateur doit accomplir à l'aide de l'application. Le modèle des tâches est décomposé en une hiérarchie de tâches/sous-tâches, les feuilles de cet arbre correspondent aux opérations devant être disponibles sous forme de primitives de l'application H^4.

Le second niveau d'abstraction, appelé **spécifications abstraites** (voir Figure I.12), représente la structure et le contenu de l'interface de l'application. Cette structure est décrite en termes d'objets d'interaction, d'éléments d'information et d'unités de présentation. Les objets d'interaction sont des tâches de bas niveau de l'interface telles que sélectionner un élément dans une liste ou montrer une unité de présentation. Les éléments d'information représentent les données à afficher, que ce soit des constantes de l'application ou des données fournies par le modèle du domaine. Les éléments de présentation sont une abstraction des fenêtres. La spécification abstraite décrit la manière d'afficher les informations dans chaque fenêtre et la façon d'interagir avec ces informations.

Le troisième niveau d'abstraction est la **spécification concrète** de l'interface. Elle précise la façon de représenter les éléments du deuxième niveau d'abstraction (unité de présentation, objet d'interaction et élément d'information) avec les éléments d'une boîte à outils.

2.2.1.2 Les outils de modélisation

Les outils de modélisation aident le concepteur à écrire le modèle de l'application au sens des MBS. Leur but est de masquer tout ou une partie de la complexité de la syntaxe des langages de modélisation, et de fournir aux développeurs une interface conviviale pour faciliter la spécification de la grande quantité d'informations stockées dans le modèle. Ces outils peuvent être des éditeurs de textes pour fabriquer une spécification textuelle du modèle (ITS [Wiecha, et al. 1989, Wiecha, et al. 1990], Mastermind [Szekely, et al. 1995]), des formulaires de création d'éléments du modèle (Mecano [Puerta 1996], Mobi-D [Puerta et Eisenstein 1998,

Puerta 1997]), ou des éditeurs graphiques spécialisés (Humanoid [Szekely, et al. 1993], Fuse [Lonczewski et Schreiber 1996]).

2.2.1.3 Les outils de conception automatique

Les outils de conception automatique permettent de générer certains aspects du modèle à partir de la spécification d'autres aspects. Ainsi, le concepteur n'est pas obligé de spécifier tous les aspects du modèle de l'application, il n'en spécifie que certains et ceux qui manquent sont alors générés par les outils de conception automatique. Par exemple, avec Janus [Balzert, et al. 1996], le développeur ne spécifie que le modèle du domaine, les autres niveaux d'abstraction du modèle sont générés automatiquement par les outils de conception automatiques.

Si l'utilisation de tels outils diminue le travail des développeurs, du fait qu'ils ont moins d'éléments à décrire, elle réduit l'éventail des domaines d'application qu'il est possible de créer (de la même manière que les squelettes d'application voir § 2.1.2). Par exemple, dans Janus, l'outil qui permet de passer du modèle de données à l'application possède implicitement un certain modèle des tâches de l'utilisateur. Dans ce cas, il s'agit typiquement de tâches de gestion de bases de données… Donc, Janus s'adapte parfaitement à la création de systèmes de gestion de bases de données mais ne peut être utilisé pour créer d'autres types d'applications.

Tous les MBS ne possèdent pas d'outils de conception automatique, par exemple Mastermind ou ITS. Le développeur doit alors décrire tous les niveaux d'abstraction du modèle.

2.2.1.4 Les outils de validation

Les outils de validation sont des outils utilisés pour fournir une évaluation de l'application avant que celle-ci ne soit créée. L'approche descendante en général et l'approche basée sur modèle en particulier, contiennent une riche représentation de l'interface que ces outils peuvent analyser. La plupart de ces outils travaillent sur les spécifications concrètes de l'interface et fournissent des critiques et des évaluations, par exemple en vérifiant que toutes les fonctionnalités sont atteignables.

2.2.1.5 Les outils d'implémentation

Les outils d'implémentation traduisent la spécification concrète de l'interface en une représentation qui peut être directement utilisée par une boîte à outils ou un générateur ascendant. Il y a essentiellement trois types d'outils d'implémentation :

- les générateurs de code source (en général C++) comme dans Mastermind ou Janus,

- les générateurs de langages de description d'interface, comme FUSE, génèrent un fichier décrivant la présentation pouvant être lu par un générateur ascendant ou directement interprété par l'application,
- les interpréteurs, comme ITS et HUMANOID, interprètent directement le modèle au moment de l'exécution.

Nous avons décrit ici la structure générale et le mode de fonctionnement des générateurs descendants. Nous abordons, dans la suite, deux exemples concrets de ces outils qui s'appliquent plus particulièrement à la création d'application de conception technique de type CAO. Ils se basent tous les deux sur le modèle d'architecture H^4. Leur différence vient des modèles qu'ils utilisent pour décrire l'interface de l'application.

2.2.2 LSI

LSI signifie Langage de Spécification d'Interface. Son but est de décrire le contrôleur de dialogue d'une application tel qu'il est défini dans le modèle d'architecture H^4. Dans ce modèle, le contrôleur de dialogue est constitué d'une liste de réseaux de transitions augmentés, les interacteurs, dont chaque état est associé à un prompt, et chaque transition, commandée par un jeton d'entrée ou par un sous-automate, appelle un questionnaire, et le cas échéant, produit le jeton retourné par celui-ci. Le langage LSI permet de générer, par compilation, à la fois le code ADA des interacteurs, et la description des données de présentation nécessaires pour les menus. LSI ne permet pas de décrire toute la présentation, mais seulement les cases de menu permettant de produire des jetons de commande ainsi que l'organisation de certaines données du modèle sous la forme de listes de choix (sélection d'un type de trait, d'une couleur etc…).

L'exemple suivant montre la spécification d'un interacteur permettant de créer un cercle par son centre et son rayon. Le texte précédé de – représente des commentaires.

```
DIALOGUE exemple_cercle – initialisation d'un nouvel automate
        PARAMETRES Position, Réel – déclaration des jetons  paramètres
        COMMANDES cercle – déclaration des jetons commandes
AUTOMATE
ETAT Initial –définition d'un état
Cercle RIEN cercle – définition d'une transition avec :
 --la nature du jeton, l'action à appeler, état final

ETAT cercle PROMPT « donnez le centre du cercle ou son rayon »
-- définition d'un prompt à afficher lorsque le dialogue est dans cet état
Position RIEN cercle_position
Réel RIEN       cercle_réel

ETAT cercle_position PROMPT « donnez le rayon »
Réel créer_cercle (Position,Réel,CR) –appel du questionnaire créer_cercle avec les
--jetons conservés dans la pile de l'automate, le questionnaire rend un
--compte rendu CR qui est (toujours) un booléen.
--traitement de la valeur du compte rendu, l'état final dépend de la valeur du
---compte rendu, une boîte de dialogue contenant un message est afficher si
 ---une chaîne de caractère contenant le message est spécifié après la valeur
 ---du compte rendu.
        CR_VRAI
                Initial
        CR_FAUX       « rayon nul »
                cercle_position – si le rayon est nul le système en demande un
--nouveau à l'utilisateur

ETAT cercle_réel PROMPT « donnez le centre »
Postion créer_cercle (Position,Réel,CR)
```

CR_VRAI

Initial

CR_FAUX *« impossible »*

cercle_réel

MENU création cercle – création d'un menu avec la case de menu « cercle »

FIN --Fin de la description de l'interacteur

Figure I.13: Exemple de dialogue en LSI

La Figure I.13 représente un exemple de dialogue spécifié en LSI. Bien que cette description soit faite d'un bloc, on peut distinguer quatre niveaux de représentation de l'interface dans LSI. Le modèle des objets du domaine est représenté par la définition des jetons paramètres. Le modèle des tâches du système est défini par l'interface des questionnaires, par exemple *créer_cercle (Position,Réel,CR).* On reconnaît le nom de la tâche à réaliser et ses paramètres. Le modèle du dialogue est défini par les automates contenus dans les interacteurs, ils définissent les différents états du dialogue, et la séquence des interactions. Enfin LSI permet de décrire une partie de la présentation.

Le but de LSI est de générer automatiquement toute la partie présentation et contrôle d'une application de conception technique (voir Figure I.14). Cette approche laisse au concepteur le soin de lier le dialogue aux actions du noyau fonctionnel dans le corps des questionnaires. Pour générer la présentation, LSI utilise un squelette d'application spécialisé pour les applications de conception technique. Ce squelette contient l'ensemble des fenêtres nécessaires à l'affichage de données géométriques ainsi que des primitives permettant l'affichage d'objets graphiques 2D. LSI a été utilisé pour la réalisation de l'interface du système EBP [Potier 1995].

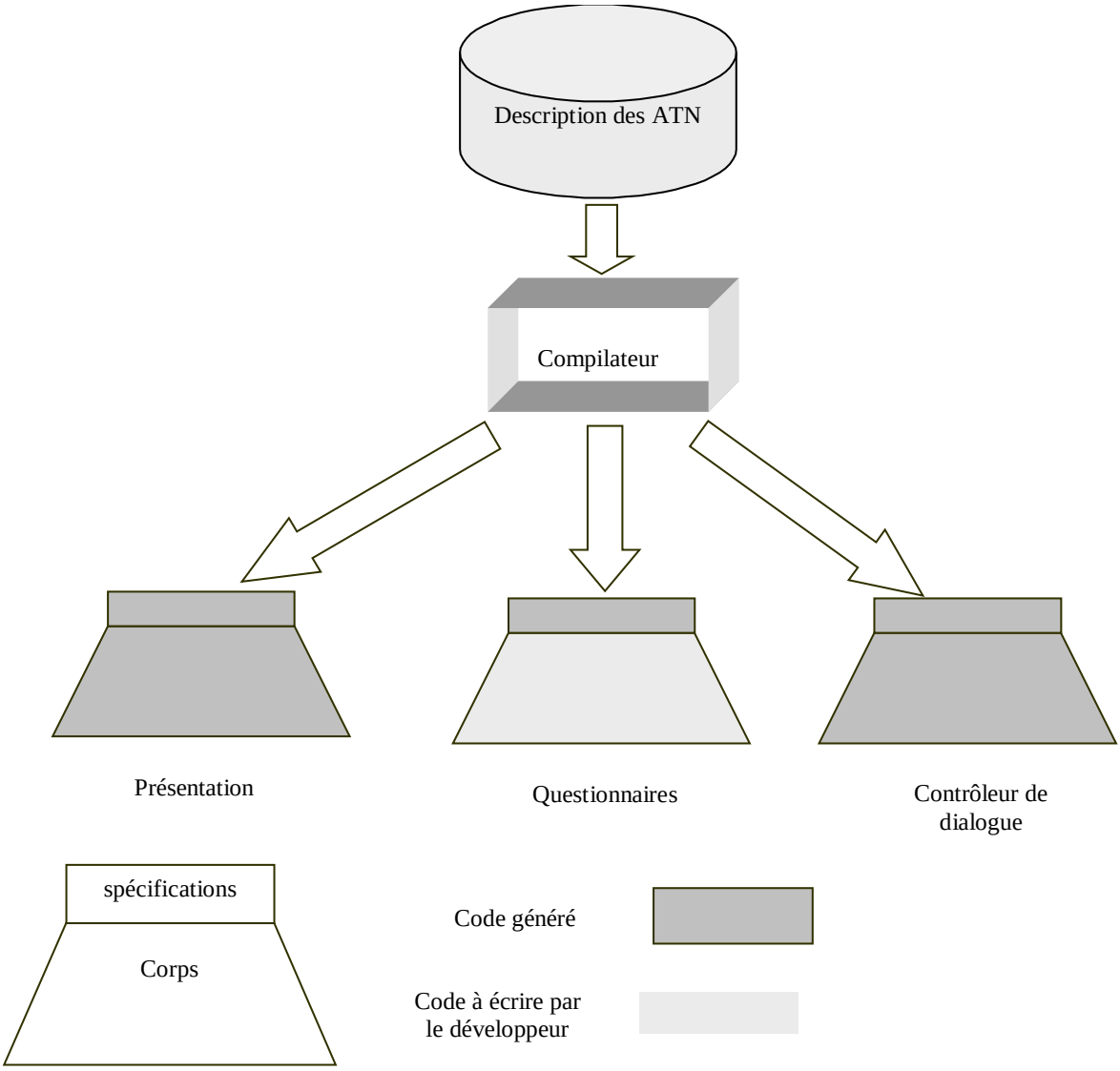


Figure I.14: Compilation du dialogue

Le langage LSI permet de décrire l'aspect présentation et contrôle d'une application graphique interactive. Bien que sa grammaire soit clairement définie, il reste difficile à utiliser car il mélange les descriptions de quatre composantes de l'application en une seule (i.e. absence de structuration). De plus, la description explicite des automates du dialogue est un travail rébarbatif car un grand nombre de transitions possibles nécessites d'être décrites individuellement. C'est pour cette raison que le système Gipse propose de décomposer ces quatre composantes en quatre modèles distincts et décrit le dialogue non pas sous la forme d'automate mais en se basant sur une structuration des tâches du système.

2.2.3 Gipse

Le système Gipse [Patry 1999, Patry et Girard 1999] est un MBS adapté à la conception des applications de conception technique. Comme LSI, il se base sur le modèle d'architecture H^4. Mais contrairement à ce dernier qui mélange dans la même description le modèle des objets, le modèle des tâches, le modèle du dialogue et le modèle de la présentation, Gipse les décompose en quatre modèles distincts.

2.2.3.1 Le modèle des objets

Le modèle des objets de Gipse contient la liste des types de jetons représentant des paramètres de l'application, et fournit un mécanisme de structuration de ces jetons. Ce mécanisme de structuration permet une description plus concise des tâches.

-- JETON

Segment

Arc

Droite

Cercle

Position

Réel

--GROUPES

Objet_Support {Droite, Cercle}

Objet_ Contour{Segment, Arc}

Objet_Graphique { Objet_ Contour, Objet_ Support}

Figure I.15: exemple de modèle des objets

2.2.3.2 Le modèle des tâches

Le modèle des tâches permet de décrire la spécification de l'ensemble des questionnaires de l'application appelés tâches dans Gipse. L'architecture H^4 permet d'accueillir des tâches définies individuellement, sans aucune connexion entre-elles. Un nom, une liste de paramètres d'entrée et un éventuel paramètre de sortie définissent une tâche. En plus de cette description, le modèle des tâches contient des lignes d'aide qui seront utilisées par le système pour fournir une aide dynamique à l'utilisateur.

```
Task {
        Créer_circle (Position, Réel)
        Help {création d'un cercle par centre et rayon} –aide globale
-- de la tâche
        Parameter {Position, « centre ») –nom des paramètres
        Parameter {Réel, « rayon »}
}
Task {
        Centre_circle (Cercle) Position
        Help { calcule le centre d'un cercle}
        Parameter {Cercle, « désignez un cercle »}
}
```

Figure I.16: exemple de tâches

2.2.3.3 Le modèle du dialogue

Le modèle du dialogue définit les interacteurs du contrôleur de dialogue ainsi que leur organisation. Chaque interacteur définit le dialogue à son propre niveau, alors que la hiérarchie des interacteurs gère implicitement par le mécanisme de consomation/production le fait qu'une tâche utilise le résultat d'autres tâches pour s'exécuter.

Les interacteurs organisent les tâches de la même manière que l'approche MAD [Scapin et Pierret-Golbreich 1990]. Les tâches sont structurées en alternative (ALT) et en séquence (SEQ), avec la possibilité de les répéter (ALT* et SEQ*).

```
Interacteur Création {
        SEQ    {
                ALT {
                        Task {New}
                        Task {Open}
                }
                ALT {
                        ALT* {
                                Task {create_circle (Position,Real) }
                                Task {create_Line (Position,Position)}
                        }
                        Task {Close}
                        Task {Quit}
                }
        }
}
```

Figure I.17: Exemple du modèle de dialogue

La Figure I.17 donne un exemple de la définition d'un interacteur de création (celui de plus haut niveau) d'un système de dessin. L'utilisateur doit d'abord créer un document ou en ouvrir un (SEQ). Puis il peut créer un nombre infini d'objets (ALT*), ou fermer le document ou fermer l'application. Les autres interacteurs sont construits de la même manière. La plupart d'entre eux ne contiennent qu'une simple alternative entre les tâches. Le fait que les communications entre les différentes tâches soient implicites dans l'architecture H^4 grâce à l'utilisation du moniteur, simplifie énormément la description du dialogue. Contrairement à LSI, où les automates sont explicitement décrits par le concepteur, Gipse les génère à partir de la description des tâches. Cela diminue d'autant la difficulté de description du dialogue. En contrepartie, Gipse ne permet pas au concepteur de créer des automates différents en fonction des tâches qu'ils permettent d'activer, et il n'autorise pas l'ajout d'appels à des fonctions sur n'importe quelle transition.

2.2.3.4 Le modèle de la présentation

Le modèle de la présentation de Gipse ne permet pas de décrire explicitement toute la couche de présentation de l'application ; il se concentre sur les moyens d'initialiser les tâches de l'application. Il décrit des menus et des sous-menus qui contiennent des étiquettes et des images initiant les tâches correspondantes.

Les différents modèles de Gipse sont interprétés aux moment du lancement de l'application qu'ils décrivent. Alors, les différents menus sont créés dans un squelette d'application qui contient l'ensemble des fenêtres graphiques de l'application, une barre d'état et une barre de menu vide. Le lien entre les tâches et les actions du noyau fonctionnel se fait dynamiquement en scrutant une base de données des questionnaires qui ont le même nom que les tâches.

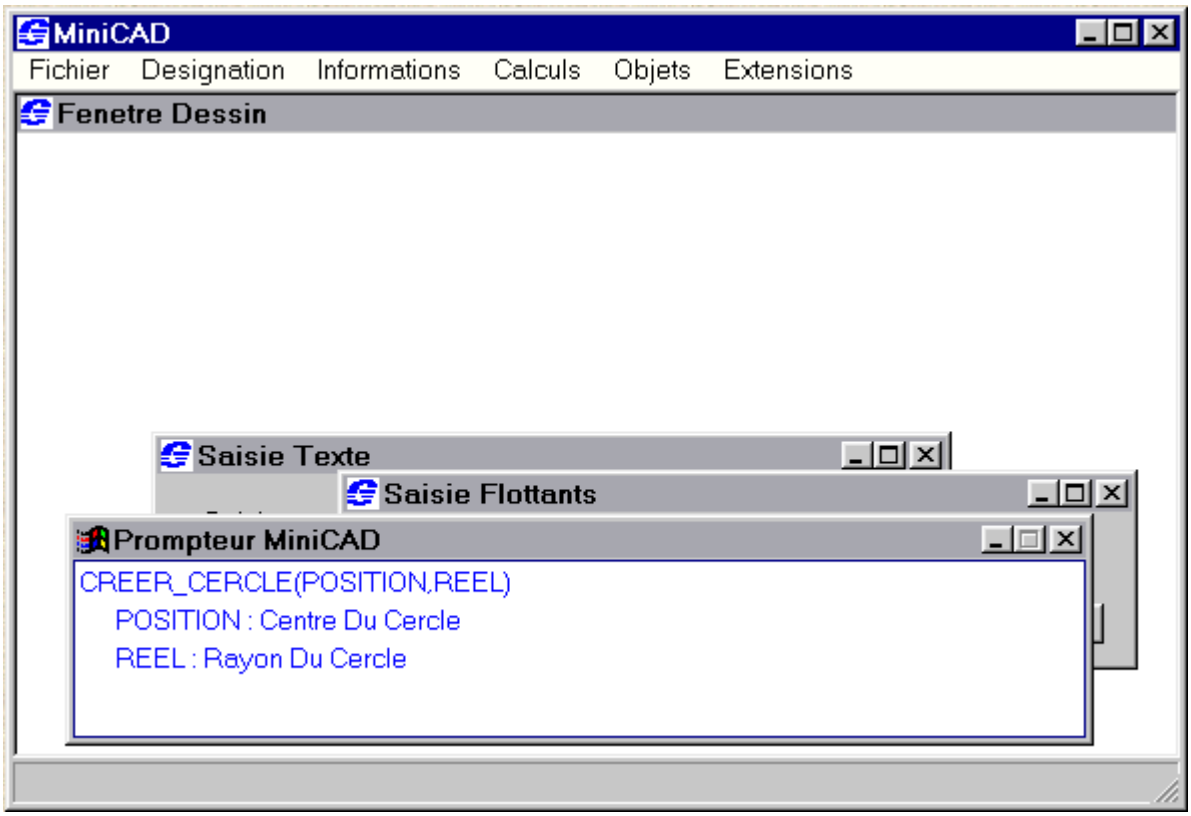


Figure I.18: Interface de Mini CAD

La Figure I.18 montre l'interface de MiniCAD [Patry 1999], qui a été créée avec le système Gipse.

2.2.4 Conclusion

Les outils que nous avons présentés sous le terme de générateurs descendants permettent de créer l'interface (dialogue et présentation) d'une application graphique interactive à partir de la

description du contrôleur de dialogue avec un langage spécifique. Il peuvent posséder toutes sortes d'outils pour la création de la description (des éditeurs de texte aux éditeurs graphiques manipulant des abstractions graphiques d'éléments du langage), pour la vérification de propriétés, pour la génération de code ou pour l'interprétation de la description. Leur rôle est de créer l'interface de l'application en se basant essentiellement sur les éléments importants de contrôle du dialogue. Donc, contrairement aux générateurs ascendants qui n'offrent que peu ou pas d'aide pour la description du dialogue, ces outils facilitent l'écriture du composant contrôleur de dialogue de l'application.

Cependant, les générateurs descendants sont confrontés à plusieurs problèmes :

Les générateurs descendants sont souvent spécialisés dans un type d'application donnée et ne peuvent que très difficilement être utilisés pour créer une application d'un autre type. Par exemple, Janus ne permet pas de créer des applications de gestions de bases de données d'un certain type et ne pourrait être utilisé pour la création d'une application de dessin ou d'un éditeur de texte. Gipse, quant à lui, est très bien adapté à la création d'application de conception technique utilisant le dialogue structuré, mais est incapable de générer une application de gestion de base de données.

Les outils de génération de la couche de présentation, sont, en général, limités à un petit ensemble de widgets et ne permettent pas la création de présentations très originales. Les couches de présentation générées sont souvent simplistes et monotones, et dans la plupart des cas ne peuvent pas être modifiées à posteriori.

De plus, les générateurs descendants sont inadaptés à la modification dynamique de l'interface de l'application. En effet, toute modification de l'interface se fait par une modification de sa description ; le système doit alors régénérer l'application (soit par compilation soit par interprétation).

Dans la section suivante, nous présentons l'approche mixte qui consiste à utiliser une description faite par un générateur descendant dans un générateur ascendant afin que le concepteur puisse décrire graphiquement la couche de présentation.

2.3 Approche mixte

L'approche mixte consiste à décrire l'interface de l'application à l'aide d'un ou plusieurs langages spécifiques d'une façon similaire à celle utilisée par l'approche descendante. Cette description est ensuite utilisée dans un générateur ascendant afin de créer, de manière

interactive, la couche de présentation. Le générateur ascendant utilisé, contraint le concepteur à respecter la description de l'interface qui a été réalisée de manière descendante. Un système suivant l'approche mixte est apparu dernièrement : il s'agit de Mobile [Puerta, et al. 1999].

2.3.1 Mobile

Mobile (Model Based Interface Layout Editor) est un système interactif de développement d'interface basé sur l'approche mixte. Ce système permet au concepteur de décrire le modèle des tâches de son application et d'utiliser une représentation de ce modèle pour créer la couche de présentation à l'aide d'un éditeur graphique.

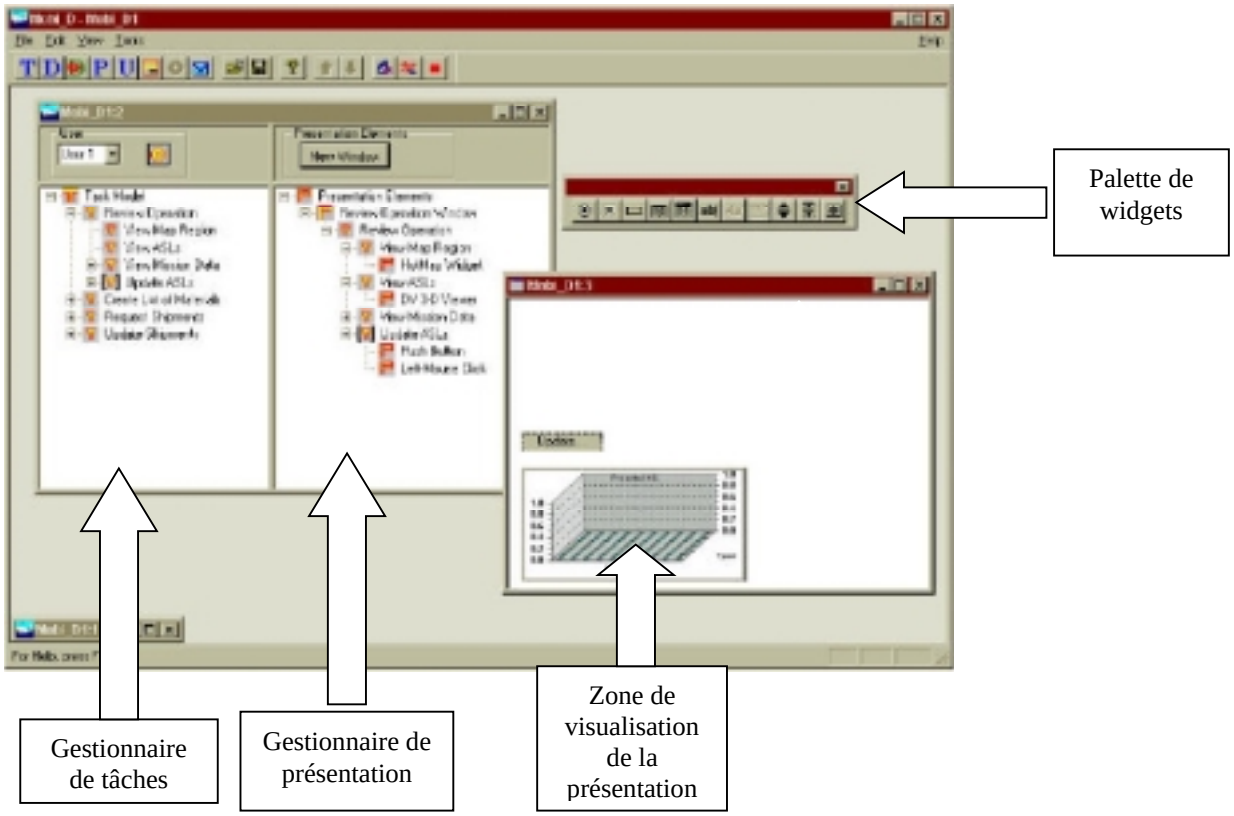


Figure I.19: Mobile

La Figure I.19 montre l'écran principal de Mobile pendant la conception de la présentation. Le gestionnaire de tâches permet d'afficher l'arbre des tâches et sous-tâches de l'application. Les tâches et les sous-tâches sont créées et modifiées à l'aide des outils de Mobi-D. Mobile offre un accès direct à cet outil si l'utilisateur désire créer ou modifier une tâche. Le gestionnaire de présentation montre les éléments composant la présentation ainsi que leurs relations avec les tâches sous la forme d'un arbre. Les éléments du plus haut niveau de l'arbre représentent les fenêtres de l'application. Sous chaque fenêtre, il y a l'arbre des tâches/sous-tâches associé à

cette fenêtre. Chaque fenêtre peut posséder plusieurs arbres de tâches. Les feuilles des arbres de présentation se trouvent juste sous les sous-tâches de plus bas niveau, ce sont des représentations des widgets associés à une technique d'interaction (un événement) permettant d'effectuer ces sous-tâches de bas niveau.

Pour créer la couche de présentation d'une application, dont il a préalablement créé le modèle des tâches, le concepteur crée une ou plusieurs fenêtres qui sont insérées automatiquement dans le gestionnaire de présentation. Puis, il assigne une ou plusieurs tâches à chaque fenêtre en utilisant la technique de « drag and drop ». Si les tâches en question contiennent des sous-tâches, elles sont incluses dans la même fenêtre que leur mère (le concepteur pourra les changer de fenêtre par la suite). Une fois que le concepteur a obtenu l'arrangement entre les tâches et les fenêtres qu'il souhaitait, il désigne chaque feuille sous-tâche (une tâche qui n'a pas de sous-tâche) et leur assigne un widget (à l'aide de la palette de widgets) et un événement pour activer la sous-tâche.

Mobile est l'un des seuls outils de conception d'interface utilisant l'approche mixte. Cette approche consiste à utiliser la description du dialogue d'une application comme la base pour créer de manière ascendante l'interface de l'application à l'aide d'un GUI-Builder. Ce type d'outils a plusieurs avantages :

- il laisse une très grande liberté au concepteur pour la création de la couche de présentation en termes de choix des éléments de présentation et de choix des méthodes d'interaction. L'application créée répond souvent mieux, en termes d'ergonomie et d'esthétique, à la volonté de l'utilisateur que celles générées avec les générateurs descendants,
- Il permet de décrire le dialogue à l'aide d'un langage spécifique de la même manière que les générateurs descendants. Donc, le composant contrôleur de dialogue de l'application est plus facile à écrire qu'avec un générateur ascendant.

Cette approche, si elle règle les problèmes de l'approche ascendante en terme d'écriture du contrôleur de dialogue, ne propose pas de solution pour la modification dynamique du dialogue. En effet, pour modifier le dialogue afin d'accéder à une fonction créée interactivement, il faut impérativement modifier le modèle du dialogue, et donc, soit le recompiler, soit le réinterpréter pour qu'il soit utilisable dans le GUI-Builder pour modifier la présentation.

De plus, comme les générateurs descendants, les outils basés sur l'approche mixte sont spécialisés pour la création d'un certain type d'application et il n'en existe aucun à notre connaissance pouvant être utilisé pour la conception d'application de conception technique.

2.4 Conclusion

Pour concevoir l'interface d'une application interactive, le concepteur a le choix d'un grand nombre d'outils pour l'aider dans son travail. Ces outils se basent sur deux grandes approches : l'approche ascendante ou l'approche descendante (l'approche mixte étant considérée comme une extension de l'approche descendante).

L'approche ascendante consiste à créer l'interface à l'aide d'outils de description de la couche de présentation. Ces outils instancient les widgets composant la couche de présentation et l'entête des fonctions de rappel. Le concepteur lie la couche de présentation au reste de l'application par programmation manuelle réalisée dans le corps des fonctions de rappel.

Cette approche permet de créer très facilement la couche de présentation de l'application et laisse une très grande liberté au concepteur dans le choix de l'aspect externe de l'application. Cette méthode de conception, ainsi que les outils qui l'accompagnent, permettent de décrire n'importe quel type d'applications (aussi bien des systèmes d'édition de textes que des systèmes CAO). Par contre, elle ne fournit aucune aide quant à la description du dialogue. Les seuls éléments de contrôle contenus dans les boîtes à outils sont les fonctions de rappel. Or, dès que le dialogue de l'application possède une structure de type syntaxique (utilisation de tâches structurées par exemple), le code à écrire dans les fonctions de rappel est de plus en plus important et de moins en moins modulaire (accès avec variables globales) de sorte qu'il devient progressivement extrêmement difficile à modifier et à maintenir.

L'approche descendante consiste à créer l'application à partir de la description avec un langage spécifique des éléments pertinents qui la composent. En général, on décrit le langage de dialogue et les objets du domaine manipulés par le dialogue. Le code de l'interface (présentation + contrôle du dialogue) est généré en utilisant cette description.

Cette approche a l'avantage de permettre une description explicite du contrôleur de dialogue de l'application. Cela s'avère très utile lorsque le dialogue supporté par l'application est complexe, notamment dans le cadre des systèmes CAO qui supportent un dialogue structuré.

Les inconvénients de cette approche sont nombreux. D'abord, les outils qu'elle offre ne sont pas génériques : un générateur descendant permettant de créer des applications de gestion d'un

certain type de bases de données ne peut pas créer l'interface d'un système CAO. Ensuite, ces systèmes gèrent très rarement la modification dynamique de l'interface qu'ils ont générée. Cela pose un problème dans le cadre de notre étude puisque notre but est d'ajouter de manière interactive de nouvelles fonctions à un système CAO. Cela implique que l'interface devra être modifiée dynamiquement afin que l'utilisateur puisse accéder aux nouvelles fonctions qu'il aura créées. Enfin la couche de présentation générée ne correspond pas toujours aux souhaits exacts du concepteur, même si la plupart des générateurs descendants permettent aux concepteurs de décrire une partie de leur interface.

L'approche mixte a été développée pour répondre au dernier problème des générateurs descendants décrit ci-dessus. Les outils utilisant l'approche mixte permettent aux concepteurs de définir la couche de présentation de l'application à l'aide d'un générateur ascendant ce qui leur offre une grande liberté d'expression pour la définition de la couche de présentation. Ces outils permettent au concepteur de lier facilement (en général de manière graphique), les éléments de contrôle du dialogue et les widgets de la couche présentation qu'ils ont créés avec le générateur ascendant. Par contre, elle ne permet pas de modifier dynamiquement l'interface, ce qui constitue pour nous un aspect essentiel.

Si nous nous ramenons maintenant à notre problème, il nous parait clair que nous devons utiliser une description explicite du dialogue, afin de pouvoir décrire le dialogue structuré. De plus, il est nécessaire que le dialogue puisse être modifié dynamiquement afin de pouvoir accéder à des fonctions créées interactivement. Des deux approches que nous avons vues, la plus dynamique est l'approche ascendante. Elle se base sur les widgets qui sont des réifications d'éléments de présentation, donc des classes d'objets qui peuvent être instanciées dynamiquement. Par contre, seule l'approche descendante permet une description explicite des éléments de contrôle de l'application. Notre idée est donc d'utiliser conjointement ces deux propriétés et de proposer des réifications des éléments de contrôle du dialogue. Ainsi, grâce à ces éléments, le contrôleur de dialogue de l'application pourra être instancié de la même manière que les widgets instancient la couche de présentation.

Ainsi que nous le livrerons au chapitre II, nous proposerons donc d'utiliser une approche ascendante pour créer l'interface de l'application, mais au lieu de nous arrêter à la description de la couche de présentation, nous proposerons des éléments permettant de décrire également le contrôleur de dialogue. Nous appellerons ces élément de contrôle du dialogue **diagets** pour (dialogue + gadget), par analogie avec les widgets (window + gadget).

3 Programmation interactive

Le but de notre travail est de montrer qu'il est possible d'enrichir les fonctionnalités d'un système de conception technique généraliste afin que l'utilisateur puisse créer, à partir de ce système, un système spécialisé répondant le mieux possible à ses besoins. Comme nous l'avons vu précédemment, pour construire une application interactive, les modèles d'architecture préconisent de séparer le dialogue, la présentation et le noyau fonctionnel.

Nous avons étudié, dans la partie précédente, comment définir le dialogue et l'interface d'une application. Cette section étudie les différents moyens permettant à un utilisateur non informaticien de définir de nouvelles actions au sein du noyau fonctionnel.

Sous le terme de programmation interactive, nous rassemblons les techniques qui permettent à un système de créer des programmes par abstraction de ce que fait un utilisateur avec un système interactif.

La programmation interactive existe actuellement sous deux forme:

- la première est connue sous le nom de programmation par démonstration ou programmation sur exemple. Elle est issue des recherches en interface homme-machine. En première analyse, elle consiste à enregistrer les interactions de l'utilisateur après un processus d'abstraction adapté.
- La seconde est issue des travaux de recherche sur les systèmes CAO. Il s'agit de la géométrie paramétrée. Elle consiste à conserver les relations entre les différentes entités géométriques d'une construction. Ainsi, lorsque l'une d'entre elles est modifiée, la construction entière est réévaluée.

Nous analysons dans la suite les deux méthodes, de programmation interactive : (la programmation par démonstration et la géométrie paramétrée) en étudiant leurs apports et leurs insuffisances par rapport à notre problème qui consiste à permettre à un non informaticien de créer de nouvelles classes d'objets, ou plus précisément de décrire des primitives de construction, de modification et d'interrogation de nouvelles classes d'objets.

3.1 Programmation par démonstration

La programmation par démonstration (PbD) [Cypher 1993] ou programmation par l'exemple est une technique de programmation interactive où un programme est construit à partir d'un exemple de son exécution. Halbert [Halbert 1984] nous donne une définition plus explicite de ce qu'est la programmation par démonstration:

« l'utilisateur écrit un programme qui effectue une tâche particulière en utilisant les mêmes commandes que celles qu'il utiliserait pour effectuer cette tâche de façon interactive. L'utilisateur programme dans l'interface du système. »

Au départ, le but de la programmation par démonstration était de permettre à un utilisateur final d'automatiser l'appel des différentes actions d'un système lors de la réalisation d'une tâche répétitive. Par exemple, le système SmallStar [Halbert 1984], permet de créer un programme qui copie un ensemble de fichiers d'un répertoire vers un autre à partir d'un exemple de copie de fichier effectuée par l'utilisateur. Puis, l'utilisation de la programmation par démonstration s'est attaquée à des problèmes plus généraux. Des systèmes permettant de réaliser des applications complètes ont vu le jour. Par exemple, Gamut [McDaniel et Myers 1998] ou Cocoa [Smith et Cypher 1995] sont des environnements de programmation sur exemple permettant de réaliser des jeux tel que PacMan.

Nous présentons dans la suite un état de l'art des différentes approches de la programmation par démonstration. Nous commençons par évaluer les difficultés inhérentes à la programmation, puis nous décrivons les différentes méthodes proposées par la programmation par démonstration pour résoudre ces difficultés.

3.1.1 Les difficultés de la programmation

Le but d'un programme est de commander un processeur et donc de décrire, d'une certaine manière, l'algorithme qu'il doit exécuter. La forme de description elle-même peut être assez variable [Girard 2000]. D'après [Pierra 1991], un algorithme est défini comme suit :

« Étant donné une action abstraite à réaliser et un processeur défini par l'ensemble des (classes d') objets qu'il sait manipuler et la liste de ses actions primitives, on appelle algorithme l'énoncé de l'ensemble (structuré) d'actions primitives permettant de réaliser cette action, chaque énoncé d'action primitive comportant la désignation des objets, constantes ou variables, sur lesquels elle doit porter. »

Cette définition fait clairement apparaître deux difficultés majeures de la programmation impérative :

- déterminer les *variables* de l'algorithme,
- définir l'*ordre* dans lequel il faut exécuter les actions primitives pour réaliser l'action abstraite.

Un programme est en général constitué d'une suite séquentielle d'actions. La séquence est ainsi la première structure de contrôle, c'est-à-dire la structure d'exécution d'un ensemble d'actions. Ces actions peuvent elles-mêmes être de simples fonctions, ou d'autres structures de contrôle comme les alternatives ou les répétitions.

Décrire une nouvelle classe passe par la description de plusieurs algorithmes. Les objets de la classe doivent pouvoir être construits, ce qui nécessite la définition d'au moins une primitive de construction possédant plusieurs paramètres de construction. Un objet étant destiné à mémoriser des attributs, une classe doit également posséder plusieurs primitives d'interrogations permettant de connaître les valeurs caractéristiques de ces attributs. Plusieurs primitives permettant de modifier l'objet peuvent lui être également associées. La suite de cette section présente les différentes méthodes existantes qui permettent à un non informaticien de décrire des programmes et donc les variables et les primitives permettant de décrire les algorithmes nécessaires pour définir une classe d'objets.

3.1.2 Les macros

Dans un système interactif, l'utilisateur dialogue avec l'application au moyen de commandes qui provoquent des actions immédiates du système qui agissent sur des opérandes [Girard 1992]. Une macro-commande (ou plus simplement macro) rassemble des commandes et leurs opérandes afin d'automatiser un traitement qui nécessite que l'utilisateur actionne plusieurs commandes. Les macros sont enregistrées dans des fichiers, appelés scripts, que le dialogue de l'application peut réinterpréter. Les scripts sont composés d'une succession de commandes, que la macro doit actionner, suivies de la valeur des opérandes. Un script peut donc être assimilé à un programme séquentiel où ont été enregistrées les interactions de l'utilisateur. Lorsque le script est exécuté, il « rejoue » les interactions de l'utilisateur.

La plupart des applications de bureautique possèdent un système permettant d'espionner les commandes que l'utilisateur actionne et d'enregistrer ces commandes et la valeur de leurs opérandes dans un script. Elles permettent donc à l'utilisateur de définir de nouvelles macros interactivement en se basant sur un exemple de l'appel des commandes que la macro devra actionner. Les systèmes d'enregistrement de macros ne permettent de réaliser que des programmes très simples qui n'ont ni paramètres d'entrée ni paramètres de sortie et dont la seule structure de contrôle est une séquence.

```
Sub H4()
'
' H4 Macro
' Macro enregistrée le 30/11/99 par LISI
'
    Selection.TypeText Text:="H4 "
    Selection.MoveLeft Unit:=wdCharacter, Count:=1
    Selection.MoveLeft Unit:=wdCharacter, Count:=2, Extend:=wdExtend
    Selection.MoveRight Unit:=wdCharacter, Count:=1, Extend:=wdExtend
    with Selection.Font
        .Name = "Times New Roman"
        .Size = 12
        .Bold = False
        .Italic = False
       .Superscript = True
    End With
    Selection.MoveRight Unit:=wdWord, Count:=1
    Selection.MoveLeft Unit:=wdWord, Count:=1
    Selection.MoveRight Unit:=wdWord, Count:=1
End Sub
```

Figure I.20: macro enregistrée par MS-Word

La Figure I.20 montre le code généré par l'enregistreur de macro de Word. Il s'agit d'une macro permettant d'écrire «H^4» tel que vous le voyez dans ce document. Le langage utilisé par cette macro est le Microsoft Basic [Microsoft 1996]. Comme le montre cet exemple, les systèmes de macros ne permettent pas à l'utilisateur de définir des paramètres (l'entête de la procédure ne contient pas de définition de paramètres formels « Sub H4() ») . Cependant, en général, ces systèmes permettent une certaine généralisation du programme puisqu'ils peuvent travailler sur la sélection courante (*Selection.Font.Name= "Arial"*), à condition que l'exemple qui a permis de créer le programme manipule la sélection. On peut donc considérer dans ce cas que les macros ont pour seul et unique paramètre l'élément sélectionné.

Les enregistreurs de macros ne permettent de créer que des programmes simples (sans paramètres et s'exécutant en séquence). Pour passer de la macro au programme, deux étapes sont nécessaires [Potier 1995] :

1. **Identifier les objets** manipulés par le programme : dans un programme, deux types d'objets sont manipulés : les constantes, qui, comme leur nom l'indique, gardent la même valeur lors de toute exécution, et les variables qui, à l'inverse, changent de valeur d'une exécution à l'autre. Les valeurs de telles variables peuvent avoir deux origines. Elles peuvent provenir de calculs internes au programme (variables intermédiaires), ou être fournies par l'environnement (les paramètres). Identifier et gérer ces deux types objets constitue la première difficulté à surmonter.
2. **Introduire des structures de contrôle** autres que la séquence : il est bien connu que le pouvoir d'expression des programmes résulte de l'existence (implicite ou explicite) de structures de contrôle [Boehm et Jacopini 1966 , Mills 1975]. Ainsi, on sait que tout langage de programmation impératif doit, au minimum, disposer des trois types de structures de contrôle fondamentales que sont la séquence, l'alternative et la répétition, auxquelles il convient d'ajouter le sous-programme. Or, la nature des scripts issus de l'espionnage de commande est exclusivement séquentielle : les séquences d'interactions sont interprétées dans l'ordre où elles ont été émises. Une exécution alternée entre deux séquences d'interactions ne peut que rarement être définie. De même, l'exécution répétitive d'une séquence ne peut être effectuée qu'en la dupliquant n fois. L'introduction des structures de contrôle non séquentielles (alternative, répétition et sous-programme) constitue donc la seconde difficulté à vaincre.

3.1.3 Identification des objets et contexte dynamique

Généraliser un exemple [Cypher, et al. 1993] revient en premier lieu à identifier parmi les objets introduits par l'utilisateur ceux qui demeureront constants pour chacune des exécutions futures, et ceux qui peuvent/doivent varier. Les travaux d'Halbert [Halbert 1984], Olsen [Olsen et Dance 1988] et Girard [Girard 1992, Girard et Pierra 1990] démontrent toute l'importance de l'identification du statut des objets parmi les interactions espionnées, ainsi que leur classification en variables internes, constantes et paramètres. Généraliser un script et le transformer en véritable programme consiste à permettre au même programme de donner un résultat différent d'une exécution à une autre. Pour cela, il faut remplacer les valeurs contenues dans le script par des noms de variables. C'est en effet la dissociation nom/valeur qui permet à

un programme de pouvoir donner des résultats différents selon les sessions. Le simple remplacement d'une valeur explicite par un nom de variable, à laquelle est associée cette valeur, suffit à généraliser le script et à le transformer en un programme séquentiel.

L'introduction de tels noms ne demande pas forcément une intervention de l'utilisateur. En effet, les systèmes de programmation sur exemple construisent le programme en même temps qu'il s'exécute (ou tout au moins qu'un exemple d'exécution se déroule). En conséquence, les variables du programme possèdent une valeur, y compris dans la phase de création. Il est alors possible de désigner cette valeur dans l'exemple, pour référencer la variable dans le programme. Les « noms » de variables peuvent alors être implicites et générés directement par le système d 'espionnage [Potier 1995].

Cependant, il est nécessaire de reconnaître, parmi les valeurs enregistrées, les constantes, les variables et les paramètres, pour permettre le passage de l'identification des objets par désignation de valeur, tel que cela est fait dans la séquence interactive, à leur nomination, telle qu'elle doit apparaître dans le programme. Une fois cette étape réalisée, le passage du script au programme est réalisé en substituant la valeur de chaque objet par un nom de variable. Le système LIKE [Girard 1992] introduit la notion de contexte dynamique qui permet de dispenser l'utilisateur de toute déclaration explicite. Dans ce système, toute création d'un objet graphique revient implicitement à déclarer une variable ayant pour valeur cet objet. Par la suite, toute désignation de cet objet peut être remplacée par le nom associé. Cette méthode a ensuite été généralisée à tous les types d'objets dans EBP [Potier 1995]. Pour cela, le système considère que les valeurs numériques sont enregistrées comme des constantes, sauf dans le cas où elles sont définies en fonction d'autres valeurs. Dans ce cas, elles sont considérées comme des variables internes dont l'expression du calcul est enregistrée dans le programme.

La définition des paramètres du programme est un point essentiel pour passer du simple script, dont l'exécution donne toujours le même résultat, à un programme où le résultat obtenu dépend des paramètres d'entrée. Selon les auteurs et les systèmes, de nombreuses méthodes ont été proposées pour faciliter l'identification des paramètres du programme construit par l'utilisateur.

- SmallStar d'Halbert [Halbert 1984] impose ainsi à l'utilisateur de remplir un questionnaire,
- Bauer [Bauer 1979] se sert de deux exécutions différentes pour déduire automatiquement la nature des valeurs : constantes, si elles sont identiques entre les deux exemples, ou variables dans le cas contraire,

- Topaz [Myers 1998] est un système permettant de décrire des scripts à partir d'exemples de manipulation d'entités de dessins. Comme les macros, les programmes écrits par Topaz ont pour paramètres les éléments sélectionnés. Cependant, ce système permet d'éditer le script généré et de le modifier pour lui ajouter des paramètres d'entrée,
- les systèmes Like [Girard 1992] et EBP [Potier 1995] quant à eux, demandent à l'utilisateur de définir explicitement les paramètres. Toute valeur donnée en cours de construction, autre que les paramètres, est considérée comme une constante. Tout objet créé en cours de construction est considéré comme une variable (interne) du programme. Il s'agit d'une détermination *a priori* des paramètres du programme,
- Cabri Géomètre [Bellemain 1992 , Laborde et Laborde 1991] détermine les paramètres (géométriques) permettant la construction d'un objet à partir de la succession des interactions de l'utilisateur ayant mené à la construction de cet objet : lorsque l'utilisateur désigne l'objet formant le résultat de sa (nouvelle) commande, le système considère alors comme paramètres les objets qui ont permis la construction et qui ne dépendent pas eux-même d'autres objets. L'utilisateur peut par la suite modifier le script (textuel) pour transformer un paramètre en constante. Il s'agit d'une détermination *a posteriori* des paramètres.

On remarque parmi les différents exemples cités ci-dessus que deux grandes stratégies s'opposent :

- d'une part, il y a la stratégie dite « explicite » où l'utilisateur dit au système quels sont les paramètres de son programme, que ce soit avant la création du programme, comme dans EBP et LIKE, ou après que celui-ci soit construit comme dans SmallStar.
- d'autre part, il y a la stratégie « implicite » où les paramètres du programme sont déduits par le système, comme dans Cabri Géomètre ou comme le fait Bauer.

Pour notre part, nous pensons que dans le cadre de la CAO, les utilisateurs connaissent parfaitement les paramètres de la pièce à créer. Par exemple, ces paramètres peuvent être décrits explicitement dans la norme qui décrit l'objet. Ainsi, le roulement à billes NF E 22-300 a pour paramètres de construction : *D* le diamètre extérieur, *d* le diamètre intérieur, *B* la largeur, *Rs* le rayon de l'alésage (Figure I.21).

Il nous apparaît donc souhaitable que l'utilisateur d'un système CAO qui permet de créer des programmes (et à fortiori des classes) à partir de la construction de pièces géométriques, puisse

définir explicitement les paramètres de la pièce qu'il dessine et qui ont toujours été, au préalable identifiés par lui.

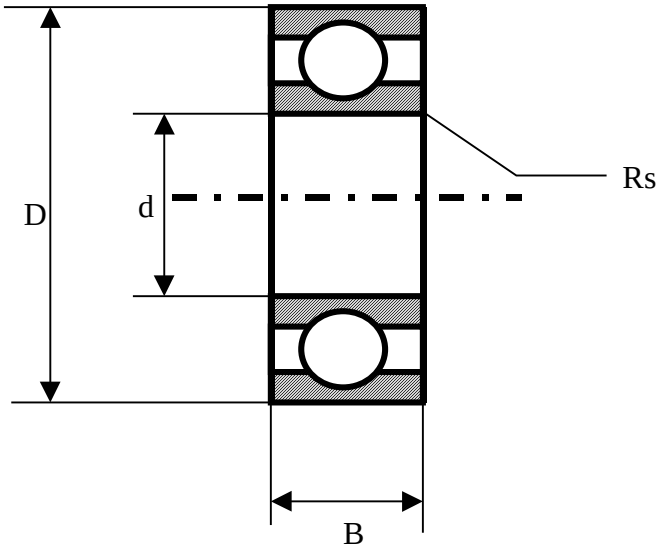


Figure I.21: Exemple de roulement à billes

Concernant les variables internes à un programme, en programmation usuelle, le programmeur doit préalablement déclarer les objets avant de les manipuler. La déclaration d'un objet revient à préciser au processeur le nom par lequel l'objet sera désigné dans le programme et son type qui détermine l'ensemble de ses valeurs admissibles. Lors de l'exécution d'un programme, chaque objet acquiert une valeur et l'ensemble des objets sur lesquels un programme opère durant son déroulement, constitue le contexte d'exécution. La gestion d'un tel contexte est classique : le noyau d'exécution du programme (ou le compilateur) gère une table de symboles. Celle-ci lui permet d'effectuer la correspondance nom/valeur indispensable lorsque l'objet est impliqué dans un calcul.

En programmation par démonstration, cette gestion prend une dimension supplémentaire : elle doit permettre, lors de la phase de création, de déclarer et de nommer automatiquement les nouvelles variables, et de réaliser la substitution inverse (valeur/nom) lors de la phase de ré-exécution. Elle doit, en plus, s'assurer de la cohérence et de la validité des objets manipulés [Potier, et al. 1995]. Ce mécanisme de gestion des associations entre les noms et les valeurs qui existe tout au long du programme à travers la création de nouveaux objets est appelé contexte dynamique.

Le contexte dynamique est constitué de deux parties : le contexte implicite et le contexte explicite.

Le contexte explicite gère les objets que l'utilisateur désigne directement par leur nom (i.e. les paramètres du programme). Sa gestion est analogue à celle rencontrée en programmation classique, où la relation avec les objets de ce contexte est un accès unilatéral de type nom/valeur.

Le contexte implicite gère les objets que l'utilisateur désigne par leur valeur à l'écran. Sa gestion est spécifique à la programmation par démonstration, dans laquelle tout objet graphique créé à une certaine étape du processus constructif, prend le statut de variable interne qui peut être référencée par sa valeur. La représentation de la valeur des objets y est effectuée par référence aux objets contenus dans le modèle du système interactif. Chaque référence (pointeur) étant unique, ceci permet d'établir un accès bilatéral sur les objets de ce contexte : valeur/nom et nom/valeur. La Figure I.22 illustre la gestion des contextes implicites. Les flèches noires montrent le moment où les objets doivent être créés dans le contexte implicite. Les flèches grises représentent comment une variable déclarée implicitement peut être référencée.

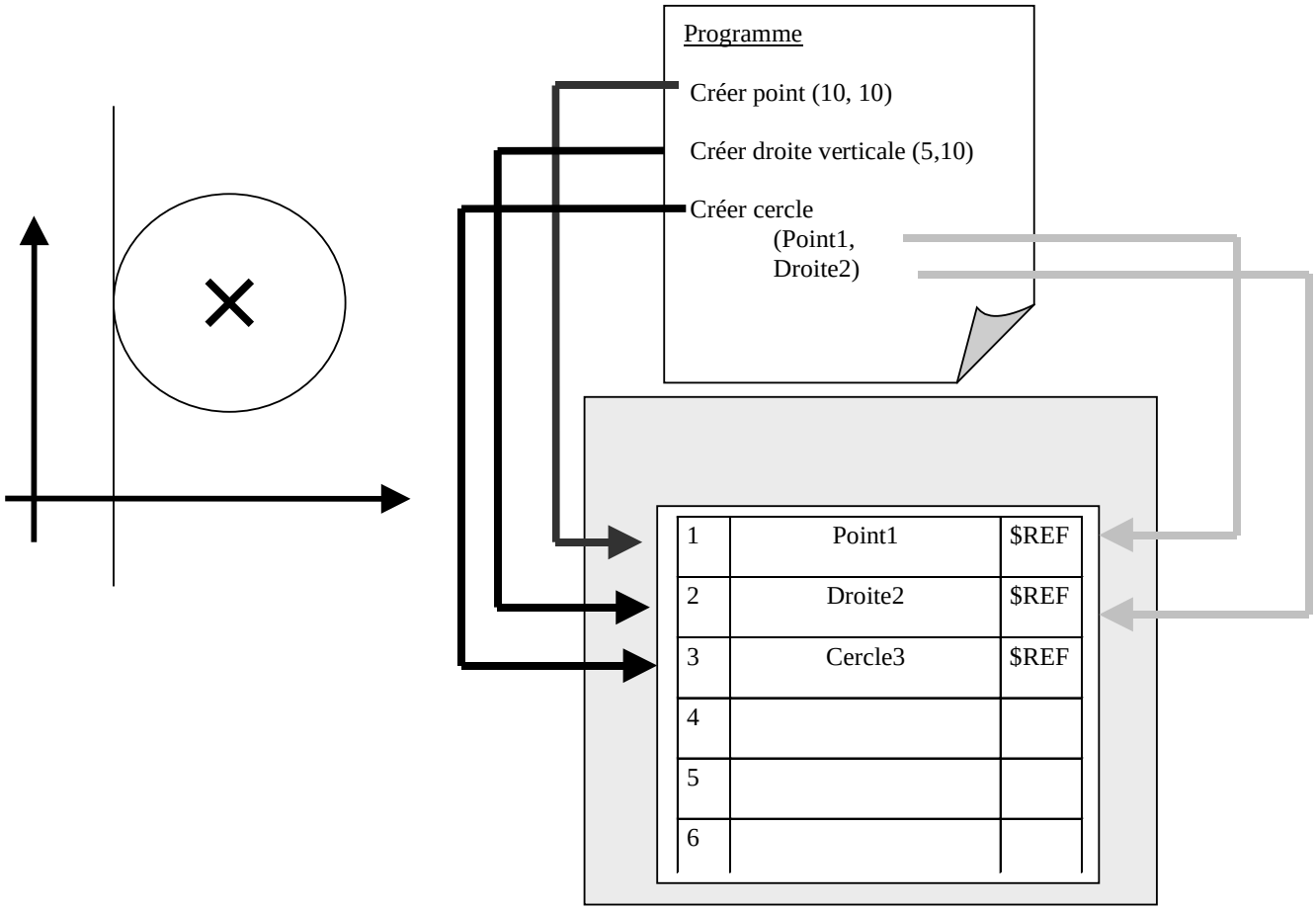


Figure I.22: Exemple de gestion du contexte implicite

Le mécanisme de nomination implicite des objets manipulés ne peut se faire que par la numérotation automatique des différents objets manipulés [Girard 1992, Van Emmerick 1991]. Cette numérotation automatique implique que le même nombre d'objets soit créés d'une exécution à une autre. Cela constitue un frein majeur à l'introduction de structures de contrôle avec l'approche impérative comme nous le verrons dans la suite.

3.1.4 Définition des structures de contrôle

La seconde grande difficulté de conception d'un programme est la mise en place de structures de contrôle autres que la séquence. Comme pour les langages de programmation textuels, il existe deux approches différentes pour l'écriture de structures de contrôle telles que l'alternative et la répétition : l'approche déclarative et l'approche impérative.

3.1.4.1 Approche déclarative

Dans un langage déclaratif, le programmeur ne décrit pas de structure de contrôle, le contrôle du séquencement des actions est géré par un moteur (interpréteur) préexistant. Le concepteur

doit lui fournir des informations sur les objets à traiter ainsi que les relations entre ces objets. Le moteur doit prendre en compte ces relations lors de l'exécution. Ces langages ne décrivent pas explicitement l'état du programme (liaisons entre les variables et les valeurs) ; celui-ci est calculé implicitement par l'interpréteur en fonction du flot des entrées et des relations entre les objets décrits dans le programme. Les relations entre objets peuvent être très diversement définies, par exemple sous forme de règles (Prolog II), sous forme de contraintes (Prolog III), ou encore, sous forme d'expressions arithmétiques (les tableurs tels que Excel ou lotus 1-2-3).

Dans sa taxonomie des systèmes de programmation interactive, P. Girard [Girard 2000] décrit les systèmes déclaratifs comme des systèmes dont la structure de contrôle d'exécution est définie à partir de relations entre objets elles-mêmes définies par le programmeur. Les systèmes de programmation par démonstration permettent de créer des objets et de définir interactivement des contraintes ou des règles de comportement sur ces objets. Lorsque le programme s'exécute, le moteur est chargé de faire en sorte qu'à chaque instant les contraintes ou les règles définies par l'utilisateur soient respectées. Cette technique est très utile pour les jeux de type PacMan où un personnage représentant un objet du programme peut se déplacer en fonction d'événements venant de l'utilisateur (définition de règles de comportement) sans traverser les murs (définition d'une contrainte).

Il existe un grand nombre de systèmes qui se basent sur l'approche déclarative dont, voici quelques exemples:

- Gamut [McDaniel et Myers 1999] permet de créer de petits jeux de type PacMan ou Tic Tac Toe. Pour cela, l'utilisateur définit les différents objets présents dans le jeu (personnage, mur, etc…) puis il définit des contraintes entre ces objets ainsi que le comportement des objets en réponse à une interaction de l'utilisateur,
- PAVLOV [Wolber 1996] permet de créer des animations. Un système de règles correspondant à un ensemble de conditions est généré par « Stimulus-Réponse » : L'utilisateur désigne un objet, un événement, et montre au système ce qui se passe sur l'objet lorsque l'événement est envoyé,
- Gramex [Lieberman, et al. 1998] « Grammars by Example » permet de créer des règles de grammaire en les décrivant interactivement à partir d'un exemple. Les règles sont ensuite introduites dans un parser.

3.1.4.2 Approche impérative

Les langages de programmation impératifs manipulent explicitement l'état du programme (ce que nous avons appelé le contexte) ainsi que l'ensemble des opérations permettant de modifier l'état d'un programme. Le contrôle du séquencement des opérations est, lui aussi, décrit explicitement à l'aide des structures de contrôle que sont la séquence, l'alternative et la répétition.

L'approche impérative de la programmation par démonstration consiste à permettre à l'utilisateur d'introduire des structures de contrôle (autres que la séquence) explicitement lors de la création du programme, c'est-à-dire lors de l'exécution de l'exemple.

On note plusieurs problèmes liés à l'introduction des structures de contrôle en programmation sur exemple :

- le problème de la nomination automatique des objets vient du fait que le contexte implicite du programme (§ 3.1.3) numérote les objets en fonction de leur ordre de création. Ce numéro est utilisé lors de la ré-exécution du programme pour accéder à la nouvelle valeur des variables. Or, si le concepteur utilise une boucle dans laquelle le nombre d'objets créés n'est pas toujours le même, ou si dans les deux branches d'une alternative, le nombre d'objets créés est différent, il en résultera un décalage dans la numérotation des objets créés après l'exécution de la structure de contrôle. Ce décalage entraînera des erreurs de référence dans la suite du programme. Pour résoudre ce problème, les systèmes LIKE et EBP ainsi que Loukipoudis [Loukipoudis 1996] proposent que chaque bloc, à l'intérieur d'une structure de contrôle, possède son propre contexte implicite. Ainsi, le contexte général du programme n'est pas affecté par l'insertion de structures de contrôles,
- la définition des structures alternatives impose à l'utilisateur de définir deux exemples du programme. Le premier correspond à la branche ALORS, le second est utilisé pour créer la branche SINON. Le principe consiste à enregistrer le programme jusqu'à la fin de la branche ALORS. Puis, le programme est ré-exécuté jusqu'à l'évaluation du prédicat, dont la valeur permet à l'utilisateur de décrire la branche SINON par l'exemple.

Ces problèmes font que peu de systèmes permettent à l'utilisateur de définir des structures de contrôle interactivement. Par exemple, Halbert [Halbert 1984] et Van Emmerick [Van Emmerick 1991] proposent d'ajouter les structures de contrôles en éditant le script du programme. A notre connaissance, seuls les systèmes EBP [Potier 1995] et LIKE [Girard 1992]

proposent la création complètement interactive de toutes les structures de contrôle usuelles (séquence, alternative, répétition et sous-programme).

Les interfaces des systèmes autorisant la définition interactive de structures de contrôles, doivent fournir à l'utilisateur non seulement la possibilité de choisir le type de structure de contrôle qu'il désire (alternative, répétition), mais aussi la possibilité d'exprimer des prédicats (valeurs booléennes) qui seront utilisés par les structures de contrôle. Par exemple, le système EBP possède des cases de menu représentant les structures de contrôle et une calculatrice booléenne pour l'expression des prédicats.

Figure I.23: Choix des structures de contrôle dans EBP

La Figure I.23 montre le menu d'EBP permettant à l'utilisateur de gérer la construction interactive des programmes. On remarque que ce menu contient notamment les commandes de gestion des paramètres du programme, et les commandes permettant à l'utilisateur de créer des structures de contrôle autres que la séquence.

3.1.5 PBD et CAO

Pour sélectionner un objet, le système utilise la position de la souris. Or, lorsque le programme est re-exécuté, il y a peu de chance que cette position permette de sélectionner le même objet [Myers 1998]. [Potier 1995] a établi l'existence de cinq relations minimales entre le système d'espionnage et le modèle de l'application pour gérer correctement le contexte dynamique. Certaines sont valables pour tous les types d'application (les relations 2, 3 et 5), les autres sont spécifiques aux applications de dessin (les relations 1 et 4):

1. la relation de désignation, qui permet de désigner les variables du programme par leur valeur dans l'exemple,
2. la relation de création, qui permet la déclaration/nomination implicite des variables par création de valeurs dans l'exemple,
3. la relation d'espionnage qui permet la mémorisation dans le programme des actions utilisées pour construire l'exemple,
4. la relation d'abstraction spatiale, qui permet de remplacer les pointés de positionnement relatif par des informations topologiques indépendantes de toute disposition des entités,
5. la relation d'évaluation qui, permettant l'évaluation des prédicats, autorise l'introduction de structures de contrôle alternatives et répétitives.

Ces cinq relations permettent d'assurer que les substitutions nom/valeur et valeur/nom seront réalisées correctement tant en création de programme que lors de son exécution.

3.1.5.1 Désignation graphique des objets

Dans les programmes à représentation textuelle, le programmeur désigne les variables par leur nom. En programmation par démonstration, les variables sont désignées par leur valeur dans l'exemple. Dans les systèmes CAO, pour désigner un objet, l'utilisateur se sert de la position d'un pointé dans la zone graphique. Or, pour le système de programmation par démonstration, la valeur de cette position n'est pas importante. L'aspect important réside dans l'interprétation de ce pointé en terme d'entité désignée (représentée par sa référence dans le noyau fonctionnel). Cette référence permet d'établir le lien entre l'objet désigné dans l'exemple et son nom dans le programme. L'espionnage des actions doit donc avoir lieu après que les pointés de désignation aient été convertis en références à des objets.

3.1.5.2 Déclaration implicite des objets

Pour pouvoir substituer valeurs et noms, le système de programmation par démonstration doit connaître à tout instant, tant en création qu'en exécution, la valeur de l'ensemble des objets désignables et les noms des variables correspondantes. Chaque création, modification ou destruction d'entité doit être communiquée au système d'espionnage, de sorte qu'il puisse gérer correctement les variables implicites du contexte du programme.

A chaque création dans le modèle, le système d'espionnage doit être prévenu et doit recevoir la référence de l'entité créée (pointeur). Il déclare une nouvelle variable à laquelle il associe la référence de l'entité créée.

3.1.5.3 Espionnage des actions

Le programme enregistré est essentiellement constitué de la trace des opérations constructives effectuées par l'opérateur pour définir l'exemple. Par opération constructive, on entend ici toute opération qui crée une entité dans le modèle, par exemple, la création d'un cercle pour un système de dessin, ou plus simplement la multiplication pour une calculatrice. Les opérations constructives doivent donc être capturées par le système de programmation par démonstration.

Cette capture peut s'effectuer à deux niveaux différents :

- soit au niveau des données arrivant dans le contrôleur de dialogue (capture de bas niveau). Le système enregistre la suite des commandes et des noms des valeurs fournies par l'utilisateur. Cette technique est utilisée dans la majeure partie des enregistreurs de macros, notamment celui de Word, ainsi que dans le système de dessin LIKE [Girard 1992]. Le système d'espionnage ne connaît pas la sémantique des actions appelées par le système, il ne peut donc l'enregistrer dans le programme : il enregistre seulement le nom de la commande telle qu'elle est connue dans le contrôleur de dialogue,
- soit au niveau invocations des fonctions émises par le contrôleur de dialogue, qui correspondent en général à l'appel des procédures du noyau fonctionnel (capture de haut niveau). Ce type d'enregistrements permet de conserver dans le programme, la sémantique des actions appelées ainsi que la structure d'appel de ces actions, ce qui est obligatoire lorsque le dialogue de l'application est structuré. Ce type d'enregistrements est celui utilisé dans EBP [Potier 1995] et dans Cabri-Géomètre [Laborde et Laborde 1991].

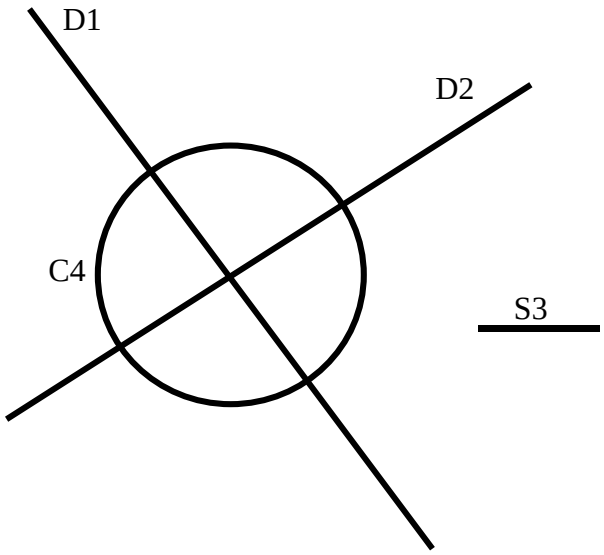


Figure I.24: Création d'un cercle

La Figure I.24 montre la création d'un cercle (*C4*) dont le centre est l'intersection de droite (*D1* et *D2*) et le rayon la longueur d'un segment (*S3*). La Figure I.25 montre le code généré lors de la création de *C4* en utilisant la capture de bas niveau. On voit que la sémantique des actions du noyau fonctionnel n'est pas du tout retranscrite. Par exemple la commande *Créer_cercle* n'a pas de paramètres. Pour que l'enchaînement des commandes conduise au même résultat, le programme doit être interprété par le même contrôleur de dialogue.

```
Créer_Cercle

Intersection D1,D2

Longueur S3

C4 est créé – message provenant du noyau fonctionnel pour dire
--qu'un nouvel objet géométrique a été créé
```

Figure I.25: Code généré avec la capture de bas niveau

La Figure I.26 montre le code généré dans le cas de la capture de haut niveau. Le code généré retranscrit l'appel des actions du noyau fonctionnel. Le programme peut alors être interprété, toute application possédant la même interface (API) pour ces actions. On retrouve ici toutes les variables intermédiaires (*I1* et *I2* dans l'exemple) qui résultent de l'utilisation d'expressions.

```
I1 = intersection (D1,D2)
I2= Longueur (S3)
C4=Créer_Cercle(I1,I2)
```

Figure I.26: Code généré avec la capture de haut niveau

La première méthode de capture des actions est très efficace dans le cas où l'application n'utilise que des tâches atomiques. Dans ce cas, les programmes sont constitués uniquement de noms de commandes suivis de leurs opérandes (nom ou valeur selon le cas). Cela retranscrit parfaitement l'appel des actions du noyau fonctionnel. Par contre, dans le cas de tâches structurées, seule la capture de haut niveau est capable de retranscrire explicitement les appels des actions du noyau fonctionnel qui sont effectuées dans le programme.

3.1.5.4 Abstraction spatiale

La quasi totalité des systèmes de CAO utilisent des constructions par contraintes. Or, certaines d'entre elles ont plusieurs solutions (par exemple, il existe deux droites passant par un point et tangentes à un cercle Figure I.27). Pour lever ce type d'ambiguïté, le système utilise la valeur du pointé de sélection lors de la construction.

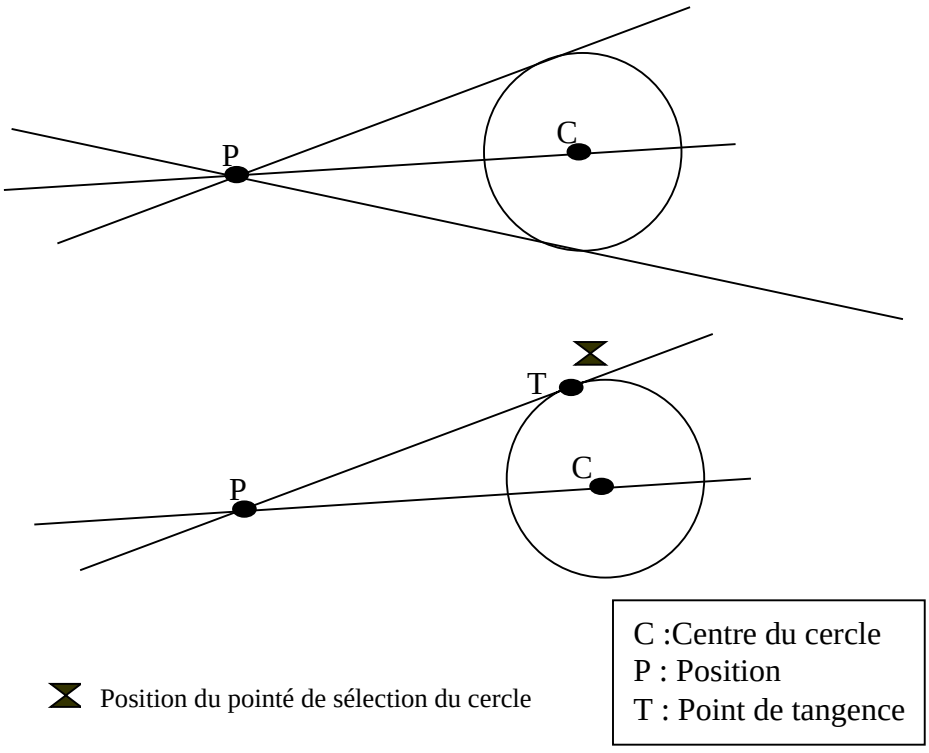


Figure I.27: Construction par contrainte d'une droite

La Figure I.27 montre la construction d'une droite passant par un point et tangente à un cercle. Le système utilise le pointé de sélection du cercle pour enlever l'ambiguïté. La droite construite est alors la plus proche du pointé de sélection. Or, le système de programmation par démonstration ne peut enregistrer le pointé de désignation tel quel pour lever l'ambiguïté lors de la re-exécution du programme. En effet, si le dessin n'est plus au même endroit, le pointé de désignation enregistré n'a plus de sens. Il apparaît donc nécessaire de remplacer ce pointé dans le programme par un élément topologique que seule l'action destinataire est en mesure de fournir. Chaque action doit donc fournir au programme (et inversement, récupérer en mode exécution) une information de type topologique (indépendante de toute position) qui caractérise l'interprétation faite par chaque action d'une information spatiale de positionnement relatif.

Ainsi, lors de la création du programme, l'action détermine à l'aide du pointé de désignation une ou plusieurs valeurs topologiques, indépendantes de la position des objets, susceptibles de lever l'ambiguïté. Ces valeurs, par exemple l'orientation trigonométrique ou non des cercles, sont stockées dans le programme. Lors de l'exécution du programme, l'action ignore la valeur des pointés de désignation associés aux objets, mais elle utilise les valeurs topologiques conservées dans le programme afin de lever les ambiguïtés de construction. Sur l'exemple de la Figure I.27, on conserverait par exemple le fait que *T* est à gauche du vecteur *PC*.

3.1.5.5 Evaluation des prédicats

Comme nous l'avons vu, l'introduction de structures de contrôle autres que la séquence (c'est-à-dire les répétitions et les alternatives), nécessite que le système fournisse à l'utilisateur la possibilité de définir des prédicats. Un prédicat est une proposition logique exprimée le plus souvent à l'aide de relations entre expressions numériques portant sur des entités du modèle. Par exemple, dans le cadre d'un système de CAO, l'expression logique : *distance(Droite1, Point2) > 10.0* est un prédicat. La Figure I.28 montre la calculatrice booléenne d'EBP qui permet à l'utilisateur de définir des prédicats qui seront utilisés notamment pour la création des structures de contrôle alternatives.

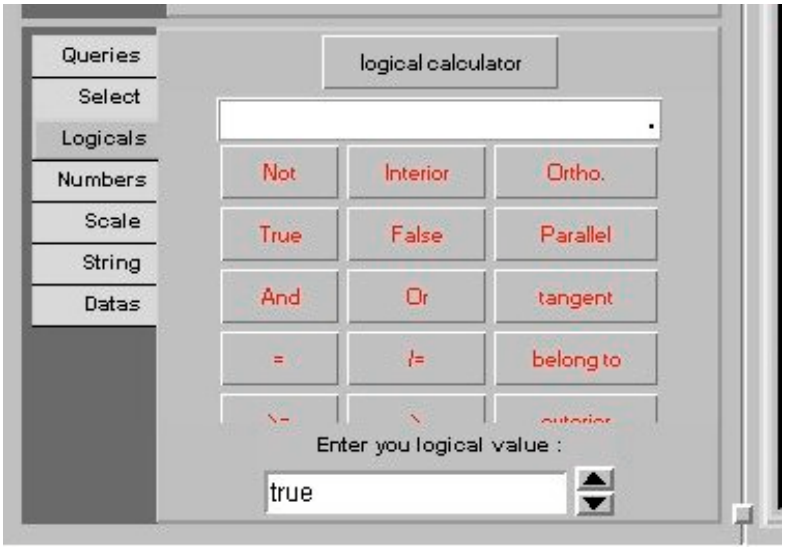


Figure I.28: Calculatrice booléenne d'EBP

3.1.6 Synthèse

La programmation par démonstration a pour but de permettre à des utilisateurs non informaticiens de créer des programmes. Pour cela, les systèmes de programmation par démonstration espionnent les interactions de l'utilisateur et se servent des données ainsi recueillies pour abstraire un programme. Le programme peut enregistrer directement les commandes et les valeurs des données, il s'agit alors d'un script. Il est composé d'une suite de noms de commandes suivies par les valeurs de leurs opérandes. Les scripts ne peuvent être re-interprétés que par le contrôleur de dialogue de l'application qui a permis de les créer. Ils sont utilisés soit pour identifier les anomalies de fonctionnement de l'application, soit pour créer des macro-commandes non paramétrées.

Pour passer d'un script à un véritable programme, deux difficultés ont dû être résolues :

- un programme travaille sur des objets qui sont soit des paramètres, soit des variables internes, soit des constantes. Les différentes valeurs utilisées lors de la construction de l'exemple doivent être classées dans l'une de ces trois catégories,
- dans les langages impératifs, l'enchaînement des actions est régi par les structures de contrôle que sont la séquence, l'alternative et la répétition. Les systèmes de programmation par démonstration doivent permettre à l'utilisateur de définir de telles structures de contrôle dans leur programme.

Pour résoudre ces deux problèmes, la solution adoptée consiste à définir un contexte dynamique capable de substituer le nom des objets dans le programme par leur valeur dans

l'exemple en mode exécution et de remplacer la valeur par le nom en mode création. Une structuration particulière du contexte dynamique permet alors d'introduire interactivement les structures de contrôle.

En mode déclaratif, le problème des structures de contrôle n'existent pas puisque la structure de contrôle de l'exécution du programme est codée dans le système de programmation par démonstration. L'écriture du programme revient alors à définir un certain nombre de règles sur et entre les objets. Lorsque le programme est rejoué, le système de programmation par démonstration est chargé de résoudre un système d'équations afin que chaque règle soit respectée. Cependant, l'approche déclarative ne laisse que peu de liberté au concepteur car il doit déterminer les contraintes de façon à ce que le système soit à même de trouver le résultat souhaité.

La programmation par démonstration nous offre une piste pour déterminer comment définir de nouvelles classes d'objets de manière interactive. En effet, cette méthode est capable de définir des programmes complexes qui ont des paramètres, des constantes, des variables internes et qui supportent d'autres structures de contrôle que la séquence. Cependant, parmi tous les exemples que nous avons cités, aucun d'entre eux ne permet de définir de nouvelles classes d'objets, c'est-à-dire des unités modulaires auxquelles sont associées plusieurs programmes correspondant à des actions différentes telles qu'actions de construction, de modification et d'interrogation.

La suite de ce chapitre présente une deuxième forme de programmation interactive : la géométrie paramétrée.

3.2 La géométrie paramétrée

La géométrie paramétrée est issue des travaux de recherche sur les systèmes CAO. Pour [Gardan 1991], il s'agit de :

> *Pouvoir définir, de manière aussi conviviale que possible, un objet paramétré, aussi bien dans sa forme que dans ses relations avec d'autres objets (assemblages…) ou dans sa fabrication.*

La géométrie paramétrée a été introduite afin de faciliter la modification des objets géométriques en changeant les valeurs de leurs dimensions. Pour cela, la méthode consiste à mémoriser, sous une certaine forme, le processus de construction de l'objet afin de pouvoir le re-exécuter lorsque l'utilisateur change une des valeurs utilisées pour construire l'objet. Cet

enregistrement du processus de construction d'un objet ressemble donc à un programme créé interactivement puisqu'il est créé lors de la première construction de l'objet, et re-exécuté chaque fois qu'une des valeurs qu'il contient est modifiée.

Une des particularités des systèmes de CAO est de permettre de décrire des pièces géométriques en exprimant des contraintes entre entités géométriques lors de leur construction. L'enregistrement de ces contraintes constitue alors une certaine forme de programme de construction de la pièce. Le premier système à permettre l'enregistrement des contraintes s'appellait MEDUSA [Newell, et al. 1983] au début des années 80. Depuis le début des années 90, cette technique s'est beaucoup développée et la plupart des systèmes sont capables d'enregistrer un ensemble de contraintes sur des objets pour définir une forme géométrique et de réévaluer cette forme lorsque l'une des dimensions impliquées dans une contrainte de cette construction est modifiée. Ces systèmes sont appelés « Dimension Driven Systems » (DDS) [Roller 1990] (système piloté par les dimensions). L'une des principales caractéristiques des DDS est que la structure de données des objets du modèle est duale. En effet, les objets du modèle paramétrique conservent non seulement leur valeur courante, mais aussi le processus de construction qui a permis d'obtenir cette valeur.

Nous étudions dans cette partie comment est enregistré le processus de conception d'une pièce paramétrée. Nous commençons notre étude en expliquant les deux grandes approches utilisées pour conserver le processus de conception des pièces paramétrées, sont l'approche équationnelle et l'approche fonctionnelle. Nous étudions ensuite la structure des modèles paramétriques, puis décrivons les différents problèmes rencontrés lors de la création et la réévaluation des programmes enregistrés.

3.2.1 Conservation du processus de conception

Les objets d'un modèle paramétrique ont la particularité de conserver leur processus de conception afin que le système puisse le réévaluer lors d'une modification. Il existe de nombreuses méthodes permettant de conserver le processus de conception d'un objet. Ces méthodes peuvent être classées suivant deux approches : l'approche équationnelle et l'approche fonctionnelle.

3.2.1.1 Approche équationnelle

L'approche équationnelle, aussi connue sous le nom de géométrie variationnelle ou d'approche déclarative, consiste à résoudre un système d'équation *: « Soit un modèle, composé d'une*

description géométrique, topologique ou approximative, et d'un nombre suffisant de contraintes géométriques, nous voulons que le modèle précis soit évalué automatiquement par le système » [Roller 1990]. Le rôle de l'utilisateur consiste à créer les entités géométriques et à énoncer les contraintes. Le système, quant à lui, est chargé de calculer la ou les solutions possibles.

L'approche équationnelle peut se formuler comme suit :

Soit p, l'ensemble des paramètres définis dans un domaine *D*, (dans la plupart des cas $D \subset \mathbf{R}^n$), et s l'ensemble des entités du modèle (points, courbes, numériques,…) qui décrivent une pièce paramétrée. s appartient à l'ensemble *S* qui permet de décrire toutes les pièces possibles. Un modèle déclaratif est représenté par une équation :

$$\boldsymbol{A(p,s) = 0 \; ; \; p \in D \, , \; s \in S}$$

A est un opérateur qui n'est généralement ni linéaire ni convexe.

L'approche équationnelle consiste à décrire une pièce paramétrée à travers une telle équation. Ainsi, le processus de construction de l'objet se résume à l'enregistrement, dans la pièce finale, de ses valeurs et de l'équation qui représente ses contraintes. Lorsqu'une des valeurs est modifiée, le système résout le système d'équations obtenu à partir des contraintes, afin de modifier la pièce pour que toutes les contraintes soient de nouveau satisfaites.

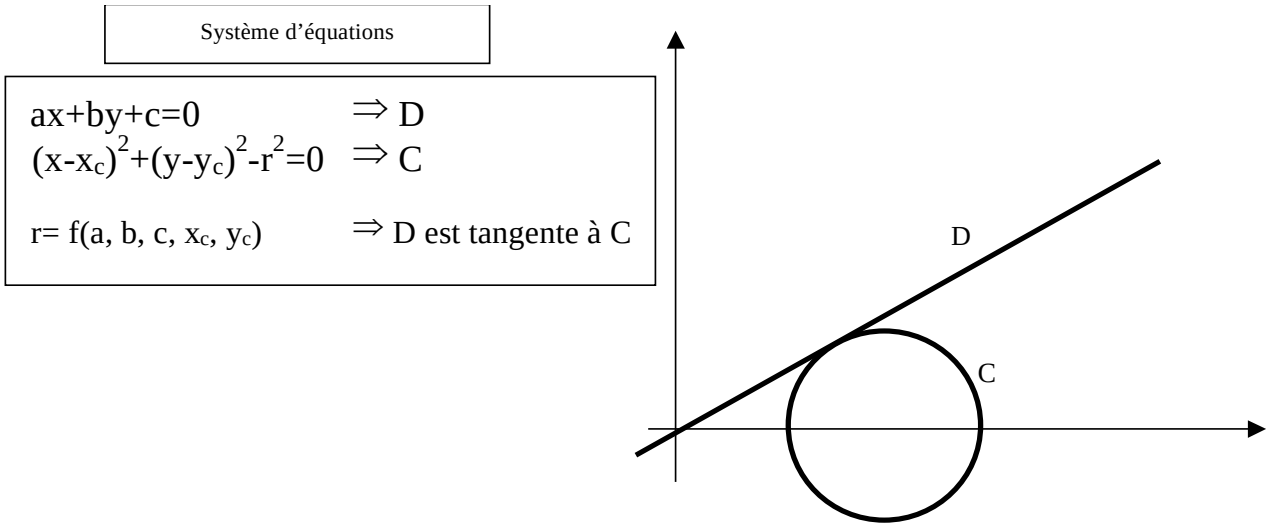


Figure I.29: Construction suivant l'approche équationnelle

La Figure I.29 décrit la construction d'un cercle centré en un point et tangent à une droite. Le système d'équations lié a cette construction permet de conserver la tangence entre les deux

éléments géométriques. Donc, si le rayon du cercle est modifié par l'utilisateur, la droite sera re-créée et si la droite est modifiée, le rayon du cercle sera recalculé en conséquence. L'ordre de définition des équations n'est pas important dans un système équationnel, il s'agit donc de systèmes déclaratifs.

De nombreuses méthodes ont été utilisées pour résoudre les systèmes d'équations découlant de la description déclarative de pièces paramétrées. Les plus efficaces semblent être les méthodes de réduction de graphe [Bouma, et al. 1995, Owen 1991].

Malgré les nombreux progrès réalisés par cette approche depuis son apparition dans les années 60 [Sutherland 1963], trois problèmes persistent [Pierra, et al. 1996a]:

1. le système d'équations a souvent plusieurs solutions comme Bouma [Bouma, et al. 1995] l'a montré sur des exemples simples. Sur la Figure I.29, la droite peut être construite au dessus ou au dessous du cercle,
2. lorsque le degré des équations est supérieur à quatre, il est impossible de calculer exactement les solutions. Seules quelques unes peuvent être approchées numériquement,
3. le fait de supporter des contraintes non-orientées interdit aux systèmes déclaratifs l'utilisation de constructions procédurales (i.e. extrusions, CSG, modélisation à partir de formes caractéristiques) qui conduisent à la définition de contraintes orientées.

Pour résoudre le premier problème, les systèmes basés sur cette approche, comme Pro-Ingenieer, proposent de capturer l'intention de l'utilisateur grâce à des heuristiques. Cependant, le fait que chaque système utilise ses propres heuristiques, et qu'il n'existe pas encore de mécanisme général assurant le déterminisme du processus de résolution du système d'équations, entraîne que deux systèmes différents ne trouverons pas la même solution pour le même système d'équations.

Une conséquence du troisième problème est que, si l'approche déclarative est très pratique pour les utilisateurs dans le cadre du dessin 2D à « main levée » où les entités sont d'abord construites puis contraintes. elle n'est, par contre, pas facilement applicable aux constructions 3D. En 3D, elle ne sert actuellement qu'au positionnement d'entités géométriques 3D [Kramer 1992] ou au positionnement de formes caractéristiques (form features ou features) [Laakko et Mäntylä 1996, Shah, et al. 1994].

Bien que cette approche résolve un nombre de problèmes de plus en plus important au fur et à mesure qu'elle évolue, elle n'est pas encore pratiquement utilisable pour paramètrer un modèle 3D. Une autre approche complémentaire, est donc nécessaire.

3.2.1.2 Approche fonctionnelle

L'approche fonctionnelle, aussi appelée approche constructive [Roller 1990] ou paramétrique, ou encore approche par propagation [Mäntylä 1990, Senella 1993], cherche à résoudre un problème différent. Etant donné une classe de formes dont le processus de conception est connu, et qui peut être supporté par l'interface d'un certain système de conception, nous voulons que chaque instance, caractérisée par les valeurs de ses paramètres, soit générée de manière déterministe.

Si on utilise une notation mathématique, un modèle paramétrique est une fonction.

$\mathbf{F : } \boldsymbol{D \rightarrow S}\mathbf{ ; s=F(p)}$ où F est la fonction qui définit l'instance à partir de la valeur de ses paramètres.

Pour chaque ensemble de paramètres qui appartiennent à D, le modèle paramétrique définit exactement une instance.

Plusieurs prototypes, comparés par Solano [Solano et Brunet 1994], et plusieurs systèmes commerciaux sont basés sur cette approche. Ils utilisent des modèles 2D et 3D. De plus, ils supportent le « feature based design » (conception à partir de formes caractéristiques, par exemple, faire une rainure sur un cube correspond à placer la feature rainure sur l'objet cube) [Hoffmann et Juan 1993, Laakko et Mäntylä 1996, Shah, et al. 1994]. Certains permettent même de construire des objets avec des structures de contrôle simplifiées, de type alternatives ou répétitives.

L'une des caractéristiques principales des modèles fonctionnels est que la fonction F est toujours exprimée comme une composition de fonctions :

$$\mathbf{F = f_n \circ f_{n-1} \circ .. \circ f_1}$$

Si on reprend l'exemple de la Figure I.29, en supposant maintenant que la droite est construite avant le cercle, la construction du cercle correspond à une fonction qui a une droite orientée et un point en entrée, fait correspondre le cercle : C=Cercle(P3,D) en sortie. Cela est illustré par la Figure I.30.

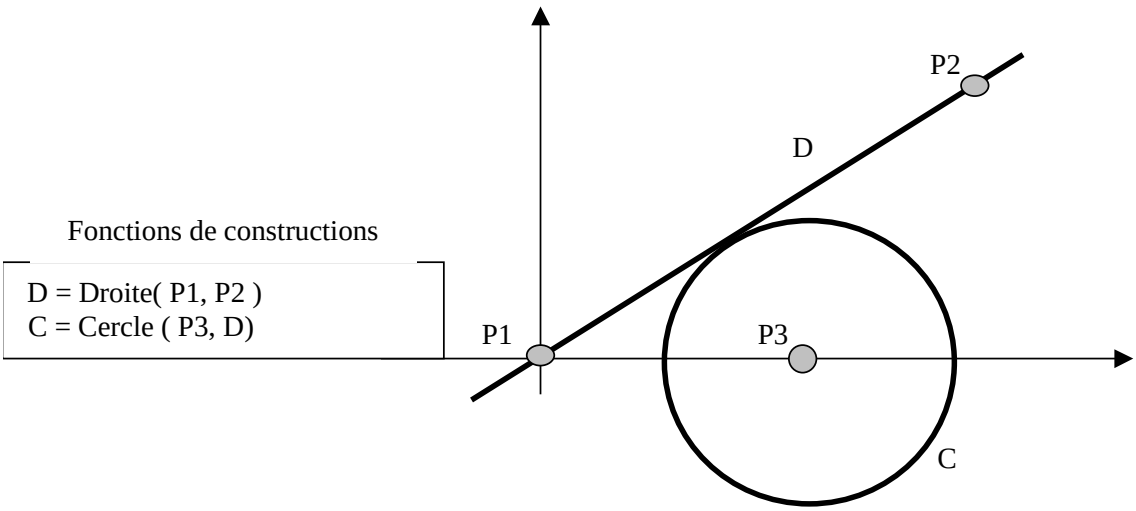


Figure I.30: Construction fonctionnelle

La modification du point *P1* ou du point *P2*, entraîne d'abord une modification de *D*, puis une modification de *C*. Une modification de *P3* n'entraîne qu'une modification de *C*. On voit donc l'une des différences majeures avec les systèmes variationnels. Dans un système fonctionnel, pour garantir que le cercle soit toujours tangent à la droite, le rayon du cercle est calculé. Ce rayon ne peut donc pas être modifié par l'utilisateur. Ceci montre la différence avec les systèmes variationnels où le rayon du cercle serait modifiable et où sa modification entraînerait une réévaluation du système d'équations. En fait, les systèmes variationnels résolvent le système d'équations globalement, alors que les systèmes fonctionnels résolvent les équations une par une dans l'ordre de leur construction.

La limite majeure de cette approche est que les fonctions paramétriques fournies par le système constituent les seules contraintes qui peuvent être enregistrées. Par exemple, il est impossible de contraindre un cercle à être tangent à trois droites si cette contrainte (fonction) n'existe pas explicitement dans le système. De plus, les systèmes fonctionnels ne permettent de décrire que des contraintes acycliques.

Par contre, et bien que dans certains cas 2D, il soit plus facile et/ou rapide de déterminer les contraintes en utilisant la méthode déclarative, l'utilisation de l'approche fonctionnelle possède de nombreux avantages :

- le processus de résolution des contraintes est déterministe et indépendant du système de résolution puisque chaque fonction paramétrique donne un unique résultat,
- cette approche peut être utilisée pour des constructions 2D et 3D qui se basent, soit sur le modèle CSG, soit sur le modèle B-Rep,

- les constructions de haut niveau comme les opérations booléennes et le « feature based design » sont supportées par cette approche.

3.2.1.3 Synthèse

L'approche équationnelle est déclarative. Elle se base sur la création d'objets géométriques entre lesquels l'utilisateur définit des contraintes, souvent à posteriori. Le système résoud alors le système d'équations résultant de ces contraintes et modifie le dessin pour qu'elles soient toutes respectées.

L'approche fonctionnelle est impérative. Elle consiste à créer successivement les différentes entités qui composent la pièce à l'aide de fonctions de construction par contraintes (par exemple, créer un cercle tangent à trois droites). La pièce est alors construite élément par élément, chaque élément étant défini par une fonction dont les paramètres sont des entités construites antérieurement.

Les systèmes déclaratifs sont en général plus faciles à utiliser que les systèmes fonctionnels, mais leur domaine d'application est moins grand. L'approche déclarative n'est pas toujours déterministe. Les systèmes peuvent être sous-contraints ce qui entraîne une multitude de solutions, ou sur-contraint, le problème à résoudre par le système est alors NP complet [Bouma, et al. 1995]. L'approche fonctionnelle, au contraire, est déterministe puisque chaque fonction possède une et une seule image, et qu'une pièce est définie par une composition de fonctions.

Les avantages et les inconvénients de chaque approche ont conduit certains auteurs à définir des modèles mixtes [Laakko et Mäntylä 1996] [Agbodan, et al. 1998, Pierra, et al. 1996a, Pierra, et al. 1996b]. Les modèles mixtes proposent d'utiliser l'approche déclarative en 2D et l'approche fonctionnelle en 3D.

3.2.2 Structure des modèles paramétriques

L'une des spécificités des technologies paramétriques est que la structure de données des systèmes et modèles paramétriques est duale. D'une part, ils contiennent, telle une photographie, un modèle géométrique permettant de présenter la forme géométrique et/ou topologique du produit conçu. Nous appelons cette représentation l'**instance courante**. D'autre part, ils enregistrent le processus de conception duquel résulte l'instance courante. C'est la **spécification paramétrique** qui est constituée de contraintes, représentées sous forme d'équations, d'inéquations ou de fonctions, qui référencent l'instance courante et mettent en œuvre différents types d'opérateurs [Agbodan, et al. 1998, Pierra, et al. 1996a].

Cette propriété constitue une caractéristique particulière des systèmes paramétriques: à la différence des systèmes de programmation par démonstration où l'espionnage peut ou pas être activé, les systèmes paramétriques regroupent au même endroit (dans le modèle de l'objet) d'une part, son processus de conception qui peut être vu comme un programme et d'autre part, l'instance courante qui peut être vue comme un exemple d'exécution de ce programme. Donc, dans un système paramétrique, à chaque fois que l'utilisateur construit une nouvelle pièce en manipulant des entités géométriques et en définissant des relations entre elles, il définit en parallèle le programme contenant le processus de construction de l'objet. On voit ici une similitude avec la programmation par démonstration ou l'utilisateur définit un programme en montrant un exemple de son exécution. Ici, l'exemple est l'objet en cours de construction et le programme est le processus de construction de l'objet. La particularité des systèmes paramétriques, où exemple et programme existent toujours conjointement, fait que les représentations de ces deux éléments peuvent être profondément imbriquées.

Souvent, dans un système paramétrique, les objets géométriques connaissent (i.e. référence) leur processus de construction. L'exemple de la Figure I.31 montre la construction d'un cercle dont le centre est l'intersection de deux droites. L'objet cercle conserve son arbre de construction. Ainsi, lorsque l'une des droites est modifiée, le cercle est lui aussi modifié de sorte que son centre soit toujours à l'intersection des deux droites (les objets modifiés sont en gris sur la figure).

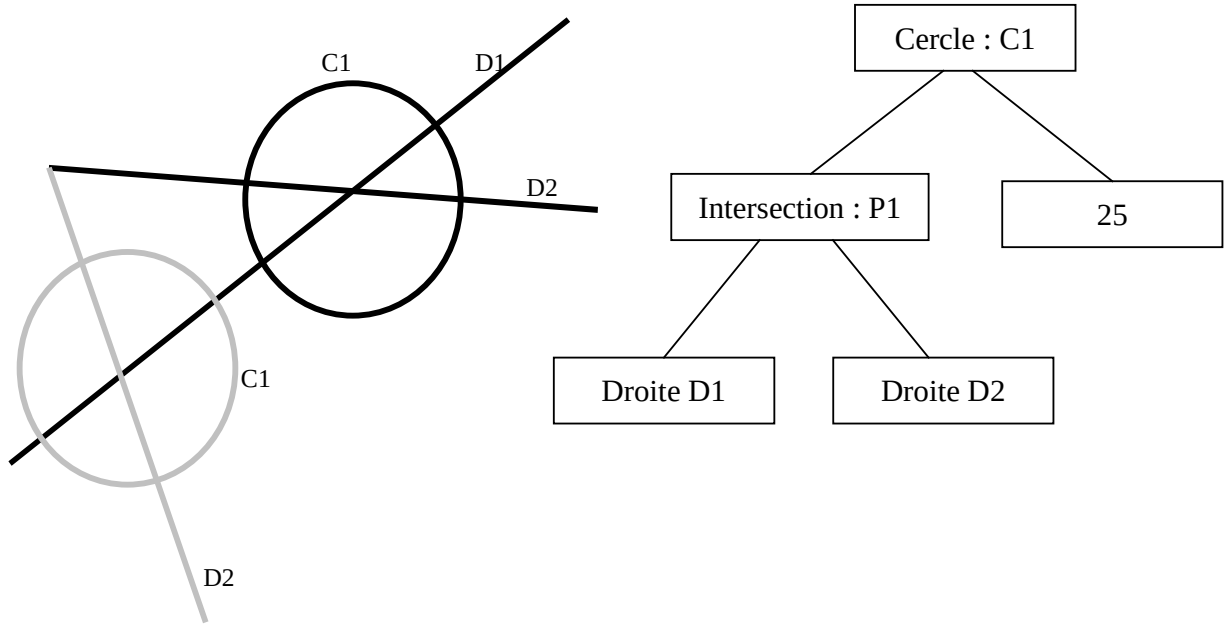


Figure I.31: Construction paramétrique

La spécification paramétrique contient toujours, de façon ordonnée (dans le cas fonctionnel) ou sans ordre particulier (dans le cas équationnel), le processus de conception de la pièce, c'est-à-dire l'ensemble des contraintes ayant modifié les entités du modèle. L'instance courante contient le résultat du processus de conception à un instant donné. Comme dans tout système de programmation par démonstration, un des problèmes majeurs des systèmes paramétriques est la conservation des liens entre les références (telles qu'elles apparaissent dans la spécification paramétrique) et les entités de l'instance courante. Ce problème peut être décomposé en deux sous-problèmes selon le modèle de données utilisé (CSG ou B-Rep). En effet, ces deux modèles se distinguent par le fait que le modèle CSG (Constructive Solid Geometry) définit les entités géométriques comme des éléments simples résultant d'opérations simples comme les opérations booléennes ou les extrusions, alors que le modèle B-Rep (Boundary Representation) les définit comme une composition d'entités (volume représenté, faces, arêtes et sommets). Dans le cas du modèle CSG, chaque fonction rend un seul et unique résultat et peut donc servir à l'identifier, alors que dans le cas du modèle B-Rep, chaque fonction rend un résultat composé de l'ensemble des entités constituant l'objet sur lequel porte la fonction. Nous étudions dans un premier temps le problème des relations entre l'instance courante et la spécification paramétrique dans le cas du CSG. Puis, nous exposons ce problème de nomination dans le cas des modèles B-Rep.

3.2.2.1 Notion de référence paramétrique

Pour illustrer le problème des relations entre les entités de l’instance courante et la spécification paramétrique, nous prenons l’exemple (Figure I.32) d’un cercle dont le rayon est égal à la distance entre deux points. Si la spécification paramétrique conserve directement la valeur de l’instance courante, la spécification paramétrique du cercle conservera la valeur de la distance et non la manière dont est calculée cette valeur. Donc, si la distance entre les deux points change, le rayon du cercle ne sera pas modifié. Il faut donc conserver dans la spécification paramétrique non seulement les valeurs qui caractérisent l’instance courante mais aussi le processus (expression) dont résultent les valeurs de l’instance courante.

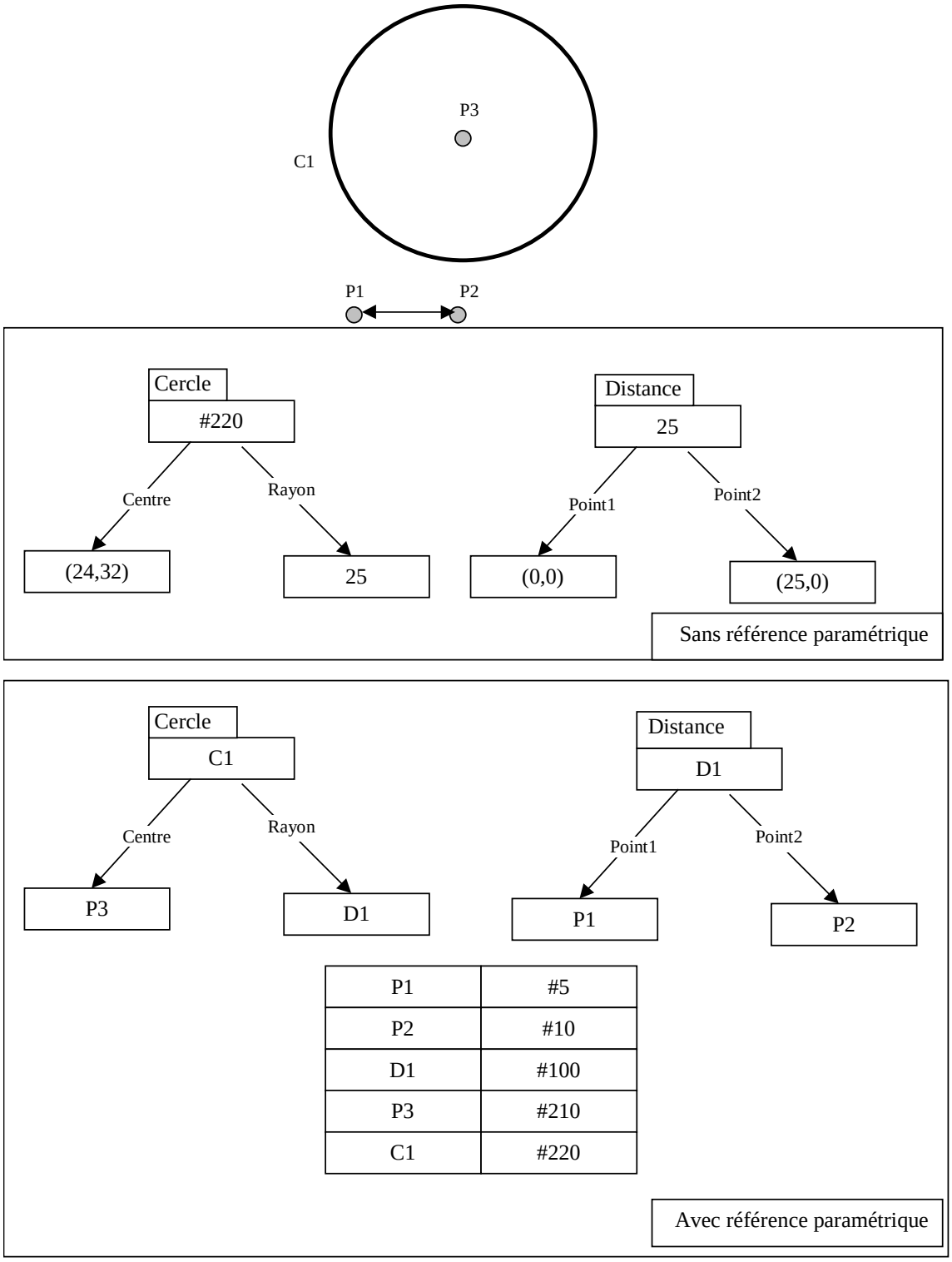


Figure I.32: Référence paramétrique

Le lien entre ce problème et le problème de contexte dans le cadre de la programmation par démonstration est évident. Un programme (la spécification paramétrique) est construit interactivement à partir d'un exemple de son exécution (l'instance courante). Le problème de

relation entre la spécification paramétrique et l'instance courante est le même que celui des relations entre nom et valeur de variables en programmation par démonstration.

Dans les langages de programmation, les programmes ne référencent pas directement les variables par leur valeur, mais par leur nom. Ceci permet de modifier la valeur en plusieurs endroits du programme. Le contexte du programme a pour rôle de conserver le lien entre les noms de variables et leurs valeurs. Il est construit lors de la compilation. Il assure un lien indirect entre les noms de variables et leur valeur au cours de l'exécution du programme. Cette association (nom/contexte/valeur) permet au programme de référencer, sans aucune ambiguïté, la valeur d'une variable où que puisse avoir été modifiée sa valeur. De même, la programmation par démonstration introduit la notion de contexte dynamique qui permet de faire l'association entre les noms des variables du programme et leur valeur dans l'exemple, tant en cours de construction (association valeur/nom) qu'en cours d'exécution (association nom/valeur).

Pour obtenir la même indépendance entre variable et valeur, [Pierra, et al. 1996a , Pierra, et al. 1996b] proposent également de régler le problème des relations entre la spécification paramétrique et l'instance courante, en utilisant la notion de contexte dynamique. Ils introduisent donc cette notion dans les modèles paramétriques sous le terme de référence paramétrique. Dans le modèle proposé, toutes les fonctions et les contraintes, qui composent la spécification paramétrique, sont basées sur des références paramétriques. Les références paramétriques elles-mêmes référencent des entités du modèle géométrique sauf si celles-ci ont, entre temps, été détruites. Ainsi, la spécification paramétrique n'est plus directement liée aux instances courantes. Ce lien se fait par l'intermédiaire d'un contexte dynamique appelé référence paramétrique (Figure I.32).

3.2.2.2 Problème de nomination

Dans le cadre de l'utilisation d'un modèle B-Rep, les fonctions de création d'entités ne rendent pas une valeur unique correspondant à l'objet créé, comme pour le modèle CSG, mais un ensemble de valeurs structurées représentant l'objet créé. Par exemple, un cube est constitué du cube lui-même, de ses 6 faces, de ses 12 arêtes et de ces 8 sommets. Donc, lorsque que ce cube est modifié, il faut savoir quels sont exactement les éléments le constituant qui ont été modifiés. Cela permet aux opérations réalisées sur ces éléments d'être réévaluées sur les éléments résultant de cette modification. Ce problème est connu sous le nom de problème de la **persistance des noms** [Hoffmann et Juan 1993, Laakko et Mäntylä 1996, Shah, et al. 1994]. Ce

problème est l'un des plus importants pour la réévaluation paramétrique. Il s'agit de caractériser les entités géométriques et topologiques d'un modèle paramétrique, c'est-à-dire leur donner un nom au moment de la conception et les « retrouver » au moment de la re-évaluation (i.e. faire la correspondance entre les entités du modèle initial et celles du modèle re-évalué) [Agbodan, et al. 2000] (Figure I.33).

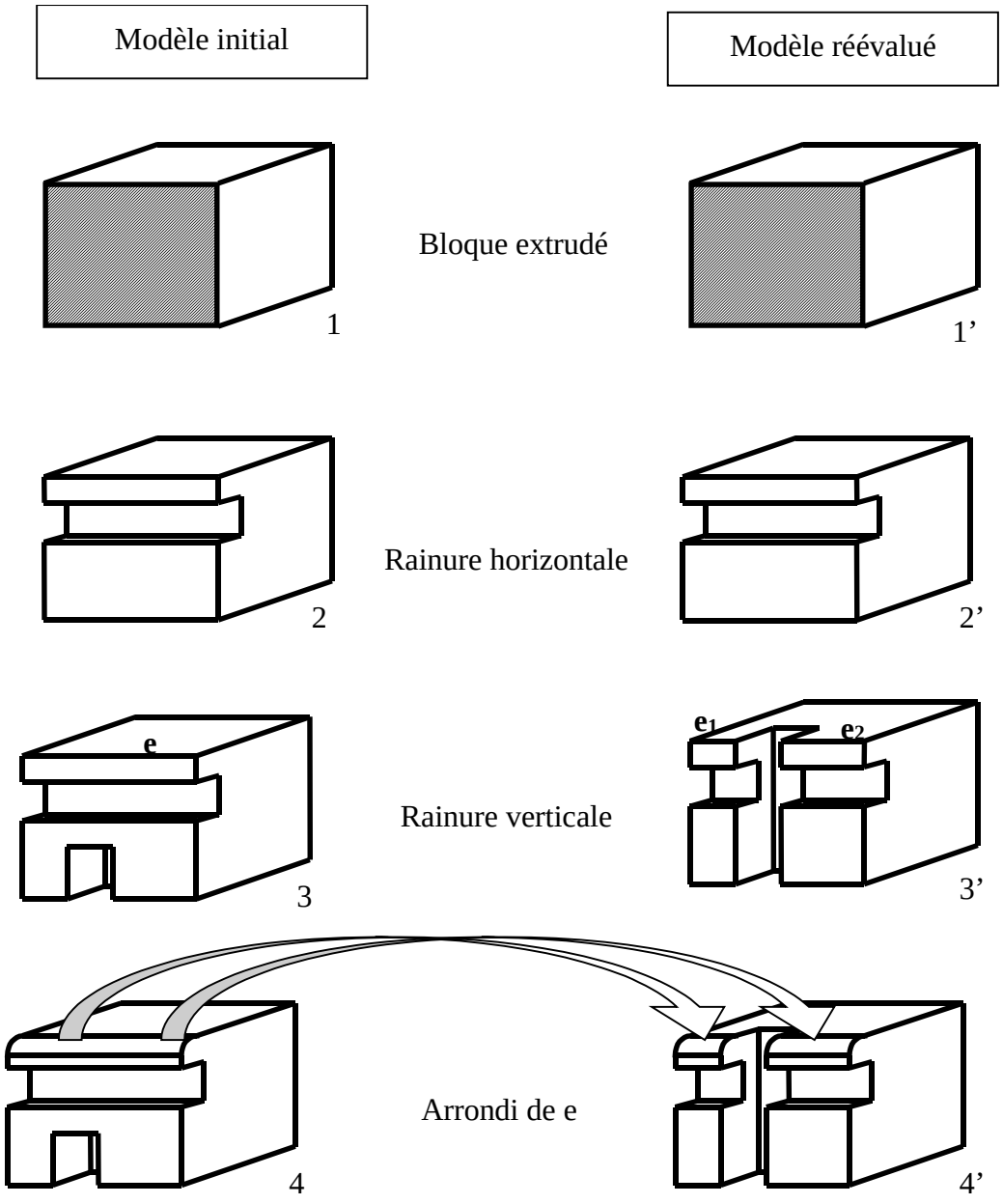


Figure I.33: Problème de nomination

Dans l'exemple ci-dessus, le modèle initial est conçu au moyen d'une spécification paramétrique constituée de quatre gestes successifs. Le quatrième consiste à arrondir l'arête *e*. Comme nous l'avons vu dans le paragraphe précédant, la spécification paramétrique ne

contient pas directement un lien vers l'arête *e*, mais elle contient la référence paramétrique de *e* (une variable de « nom » *e*). Sur l'exemple, lors de la réévaluation, le modèle ne contient plus d'arête « *e* », celle-ci a été divisée par la rainure verticale en deux arêtes e_1 et e_2 (la fonction de création de la rainure a notamment deux éléments en sortie). Ainsi, pour réaliser l'arrondi, la spécification paramétrique ne doit plus se baser sur *e* mais sur e_1 et e_2. La référence paramétrique de *e* doit donc conserver le fait que e a été transformé en e_1 et e_2.

Trois problèmes ont été identifiés [Agbodan, et al. 2000] pour atteindre un mécanisme permettant de retrouver toutes les entités résultant de modifications de valeurs lors de la réévaluation de la spécification paramétrique d'une pièce. Un mécanisme de nomination doit être défini (1er problème). Celui-ci doit être suffisamment puissant pour permettre d'effectuer une mise à jour des relations (*matching*) robustes par rapport aux réévaluations (2ème problème). Enfin, on doit pouvoir exprimer différentes sémantiques en utilisant des paramètres de haut niveau (agrégats d'entités géométriques ou topologiques) (3ème problème). La Figure I.34 montre un exemple nécessitant de pouvoir exprimer deux sémantiques différentes.

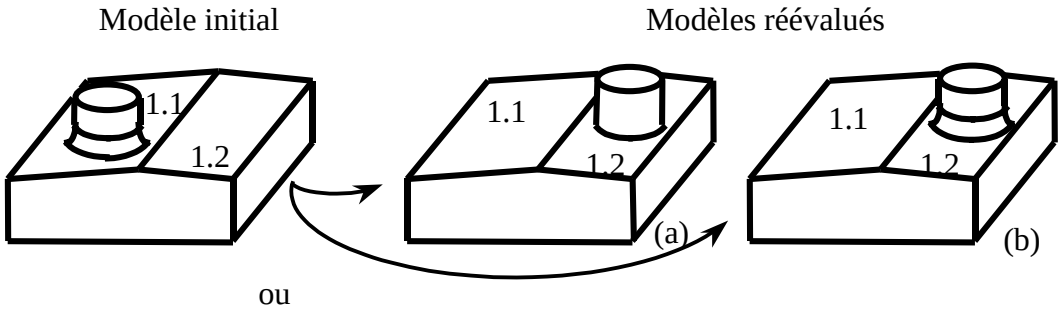


Figure I.34: Différence de sémantique

L'objet de la Figure I.34 a été créé à l'aide de trois gestes constructifs successifs : création d'un bloc par extrusion d'un contour polygonal, création du cylindre sur le bloc, puis création de l'arrondi entre le bloc et le cylindre. Selon que la fonction arrondi a été exprimée dans la spécification paramétrique entre le cylindre et la face 1.1 (1er cas) ou entre le cylindre et la coque composée des faces latérales de l'extrusion (2ème cas), le modèle réévalué doit être différent, car la sémantique exprimée dans la spécification paramétrique est différente. Dans le premier cas (a), l'arrondi a disparu puisqu'il ne peut plus être fait entre le cylindre et la face 1.1. Dans le deuxième cas (b), l'arrondi existe toujours puisqu'il a pu être fait entre le cylindre et la coque composée des faces latérales 1.1 et 1.2. Pour permettre l'expression de ces deux sémantiques différentes, le mécanisme de nomination doit gérer la nomination d'entités de plus haut niveau que les faces.

Pour résoudre le problème de la persistance des noms, plusieurs solutions ont été proposées. Certains auteurs se sont surtout attachés à analyser la structure interne d'un modèle paramétrique en explicitant les possibles représentations [Hoffmann et Juan 1993, Laakko et Mäntylä 1996, Pierra, et al. 1996a, Pierra, et al. 1996b, Solano et Brunet 1994]. D'autres auteurs ont analysé la structure mathématique sous-jacente [Pierra, et al. 1996a, Raghotama et Shapiro 1997]. La plupart des auteurs se sont surtout intéressés à la phase de construction (1er problème). Récemment, différents mécanismes de nomination plus ou moins robustes en cas de modification topologique lors de la réévaluation ont étés proposés. Chen [Chen 1995] a proposé un mécanisme de nomination, non-ambiguë en phase de construction, qui s'appuie sur deux représentations du modèle paramétrique, une non évaluée (sous forme textuelle) qui contient les noms persistants, et une évaluée basée sur un modèle d'arête (E-Rep) [Hoffmann et Juan 1993]. Le lien entre les deux modèles est effectué par une sorte de contexte qui associe à chaque entité du modèle évalué un nom (persistant) du modèle non évalué. [Kripac 1995] propose d'utiliser une table de correspondance entre chaque entité topologique du modèle initial et les faces résultantes dans le modèle final. Cette table conserve en fait l'historique des faces. L'approche la plus récente, faite par [Agbodan, et al. 2000], propose de conserver, dans la référence paramétrique, un historique des coques ainsi que leurs relations associé à un historique des faces et de leurs relations dans un graphe, lorsque la réévaluation met en jeu d'importantes modifications topologiques.

Bien que les solutions proposées récemment pour résoudre le problème de la persistance des noms soient de plus en plus puissantes, aucune, à notre connaissance, ne permet de résoudre la totalité des problèmes rencontrés.

3.2.3 Synthèse sur la géométrie paramétrée

La géométrie paramétrée a pour but de permettre la modification aisée des objets géométriques créés dans un système CAO. Pour cela, les systèmes paramétriques conservent dans chaque objet le processus qui a permis de le construire. Lorsqu'une des valeurs utilisées lors de la construction est modifiée, le système re-exécute ce processus de construction et modifie l'objet en conséquence.

Il existe deux grandes méthodes pour construire les objets dans les systèmes de CAO. La première consiste à créer un ensemble d'objets simples et à la contraindre ensuite pour obtenir la pièce voulue. Cette méthode est connue sous le nom d'approche variationnelle ou équationnelle. Dans ce cas, le système définit la pièce paramétrée comme étant un ensemble

d'objets liés entre eux par des équations ou inéquations de contraintes. Lorsqu'une des valeurs utilisées pour construire un des objets de base est modifiée, le système réévalue alors l'ensemble des contraintes. La seconde, connue sous le nom d'approche paramétrique ou fonctionnelle, consiste à enregistrer dans la pièce, la composition de fonctions qui a permis de la créer. On obtient alors l'arbre de construction de l'objet. A chaque fois que l'un des paramètres d'une fonction de l'arbre est modifié, cette fonction est réévaluée, et si son résultat était utilisé par d'autres fonctions, celles-ci sont aussi réévaluées, cela jusqu'à la dernière fonction utilisée pour créer l'objet.

Si l'approche variationnelle est plus facile à utiliser dans le cadre des applications en deux dimensions, elle n'est pas applicable pour la majeure partie des opérations 3D. En général, les modeleurs géométriques du commerce utilisent le mode variationnel en 2D et le mode fonctionnel en 3D.

La géométrie paramétrée permet à l'utilisateur de créer un programme (le processus de conception d'une pièce) sans qu'il le sache. Dans la suite de notre travail, notre objectif sera d'utiliser la notion de géométrie paramétrée non seulement pour conserver le processus de conception d'un objet géométrique (qui correspond à un constructeur de classe), mais aussi pour créer des fonctions permettant d'accéder au attributs d'un objet de cette classe. C'est ce que nous présenterons dans le chapitre III.

3.3 Conclusion sur les techniques de programmation interactive

La programmation interactive permet à un utilisateur non informaticien de créer des programmes sans écrire de code. Pour cela, les actions de l'utilisateur sont capturées par le système qui génère un programme à partir des informations collectées. Ce programme peut ensuite être ré-exécuté par l'application qui l'a créé avec d'autres valeurs pour ses paramètres. Nous avons identifié deux méthodes pour créer des programmes interactivement.

La première est connue sous le nom de programmation par démonstration. Son but est de créer des programmes qui contiennent l'ensemble des interactions nécessaires pour effectuer un certain type de tâche. Ainsi, lorsque l'utilisateur veut effectuer de nouveau une telle tâche, plutôt que de refaire l'ensemble des interactions nécessaires, il demande la ré-exécution du programme capturé.

La seconde est appelée géométrie paramétrée, elle a pour but de permettre à l'utilisateur d'un système de CAO de modifier facilement une pièce qu'il a créée. Pour cela, la pièce conserve

son processus de conception, et lorsque l'une des valeurs utilisées pour la construire est modifiée, le processus de conception est réévalué ce qui modifie la pièce.

Ces deux méthodes de programmation interactive ont comme principal point commun la possibilité de décrire un programme à partir d'un exemple de son exécution. En programmation par démonstration, le programme est créé au moment où la tâche est effectuée pour la première fois. Avec la géométrie paramétrée, le programme est construit lorsque la pièce paramétrée est créée avec le système CAO. Pour ces deux approches, on distingue deux méthodes pour définir le programme. La première, la méthode équationnelle, est qualifiée de déclarative pour la PBD, et variationnelle pour la géométrie paramétrée. Cette méthode consiste à enregistrer des contraintes entre les objets de l'application. Les systèmes utilisent ensuite des moteurs spécifiques qui assurent que toutes les contraintes sont satisfaites. L'utilisateur ne peut pas déterminer comment le système doit résoudre le système d'équations issu des contraintes. La seconde méthode est appelée impérative en PBD et fonctionnelle en géométrie paramétrée. Elle consiste à décrire un programme comme une succession de fonctions. Cette méthode offre la possibilité à l'utilisateur de définir exactement la structure de contrôle de l'enchaînement des fonctions du programme.

La programmation par démonstration et la géométrie paramétrée appartiennent au même univers mais sont différentes sur certains points :

- la structure de données qui enregistre le programme est fréquemment différente. En programmation par démonstration, le programme est séparé des entités sur lequel il porte comme dans les langages de programmation classiques. La géométrie paramétrée conserve au même endroit le programme et la valeur de l'exemple qui a permis de le construire. Le programme et les données sur lequel il porte, sont réunis comme dans une programmation orientée objet,
- la programmation par démonstration décrit trois niveaux de données dans les programmes : les paramètres qui sont fournis par l'utilisateur avant la ré-exécution, les variables et les constantes. La géométrie paramétrée ne connaît que les paramètres, c'est-à-dire que toutes les données utilisées dans le processus de conception d'un objet peuvent, en général, être modifiées,
- en programmation par démonstration, l'utilisateur dit explicitement au système quand ce dernier doit enregistrer un programme alors qu'avec la géométrie paramétrée, le système

enregistre en permanence les constructions d'objets sans que l'utilisateur sache qu'il crée un programme,

Pour créer de nouvelles classes d'objets géométriques, il est nécessaire de définir non seulement des constructeurs, mais aussi des attributs et leurs fonctions d'interrogation. La géométrie paramétrée nous semble une bonne approche pour créer des constructeurs de classes, ainsi que des fonctions d'interrogation d'attributs. Mais, il faut que l'utilisateur puisse définir explicitement les paramètres et variables internes des constructeurs. La programmation par démonstration nous offre des solutions à ce problème. Notre idée est donc d'utiliser un modèle paramétrique, décoré de concepts empruntés à la programmation par démonstration.

4 Conclusion générale

Notre but est d'étudier comment permettre à un utilisateur d'un système de CAO généraliste de spécialiser son système afin qu'il réponde à ses besoins particuliers. Il s'agit de lui permettre de définir des classes d'objets spécifiques de son domaine de manière interactive sans qu'il ait à les programmer de manière classique, puis de personnaliser l'interface de dialogue afin de pouvoir interagir avec ces nouvelles classes d'objets.

Pour créer une application graphique interactive, les modèles d'architecture préconisent tous de séparer l'interface, composée de la présentation, du contrôleur de dialogue et du noyau fonctionnel. Nous avons vu, dans ce chapitre, que le modèle architecture H^4 était le mieux adapté aux spécificités des systèmes de CAO (tâches multi-objets, tâches structurées, objets du modèle en relation les uns avec les autres, objets structurés). Il décrit non seulement l'organisation et les liens qui existent entre les macro-agents qui composent l'application, mais également leur structure interne en termes de micro-agents. Pour spécialiser une application, il apparaît nécessaire de modifier, d'une part, son interface, et d'autre part, son modèle en y ajoutant de nouvelles classes d'objets spécifiques du domaine visé.

Concernant l'interface, il existe deux grandes approches pour la définition de l'interface d'une application interactive.

La première, l'approche ascendante, utilise des boîtes à outils, les widgets, pour décrire la couche de présentation. Elle offre des mécanismes de contrôle de l'application simples (les fonctions de rappel) mais insuffisants pour décrire un dialogue structuré (basé sur des tâches

structurées). Cette approche est très générale et elle permet de créer différents types d'applications.

La seconde, l'approche descendante, propose de décrire certains aspects de l'application à l'aide de langages spécifiques, puis de générer l'interface de l'application à partir de ces descriptions. Cette méthode permet de décrire le dialogue structuré des systèmes de CAO, mais elle est très spécialisée (i.e. un générateur descendant permettant de décrire des éditeurs de texte ne peut pas créer de système de CAO et vice versa). Elle ne permet pas non plus de décrire finement la présentation comme le permettent les boîtes à outils.

Notre idée est de proposer une extension aux boîtes à outils permettant de décrire le dialogue structuré de la même manière qu'est décrite la présentation à l'aide des widgets. Nous obtiendrions ainsi un outil aussi généraliste que les boîtes à outils de présentation, et permettant une description du dialogue avec des éléments spécialisés. Nous développerons cette proposition et nous proposerons de l'appeler boîte à outils du dialogue au chapitre II.

En ce qui concerne la définition interactive de classes spécifiques d'un domaine d'activité, nous avons présenté deux approches permettant à un utilisateur non informaticien de créer des programmes. La première, connue sous le nom de programmation par démonstration, consiste à enregistrer les interactions de l'utilisateur lorsque celui-ci réalise une tâche, et à abstraire un programme à partir des données recueillies. Lorsque l'utilisateur veut, de nouveau, réaliser cette tâche, il n'a plus qu'a ré-exécuter le programme créée. La seconde approche, basée sur la géométrie paramétrée, consiste à conserver dans chaque pièce crée le processus qui a permis de la construire. Lorsque l'utilisateur ou le système modifient une des valeurs utilisées pour créer la pièce, le système ré-exécute son processus de conception et la modifie en conséquence. Ces deux approches permettent d'enregistrer un programme, mais elles ne nous disent rien quant à la création de classes d'objets qui nécessitent la définition de plusieurs programmes (les constructeurs et les sélecteurs) associés à un type d'objets. Notre proposition sera d'utiliser une approche de type paramétrique. Mais cette approche sera modifiée pour permettre, en particulier, la différenciation entre les variables internes et les paramètres. Ce modèle paramétrique, spécialisé dans la création de classes d'objets, sera décrit dans le chapitre III.

Le dernier point de notre étude (Chapitre IV) présente la maquette réalisée pour valider nos propositions. Elle montre comment les concepts et les outils décrits dans ces chapitres II et III

ont été utilisés. Nous montrons, enfin, un exemple d’utilisation de cette maquette pour créer et intégrer interactivement une nouvelle classe d’objets.

Chapitre II
La boite à outils du dialogue

1 Introduction

Les travaux sur les modèles d'architecture des applications graphiques interactives telles que Seeheim [Pfaff 1985], Arch [Bass, et al. 1991, UIMS 1992] ou Pac [Coutaz 1987] préconisent de séparer la réalisation de l'interface (présentation + contrôleur de dialogue) de celle de noyau fonctionnel. Lors de la spécialisation interactive d'une application de conception technique, il apparaît donc nécessaire de pouvoir modifier dynamiquement la partie interface de l'application, ceci afin de permettre à l'utilisateur d'accéder aux nouvelles classes d'objets qu'il a créées.

Deux grandes approches pour créer l'interface d'une application ont été proposées [Coutaz 1990]:

- **L'approche ascendante** consiste à créer la couche de présentation à l'aide d'une boîte à outils de présentation. Les boîtes à outils sont composées de widgets qui sont des réifications d'éléments de présentation. Elles offrent un mécanisme de contrôle simple, appelé fonctions de rappel, qui consiste à préciser le comportement de l'application en réponse à un événement sur un widget (par exemple le clic sur un bouton). La création d'une application avec la méthode ascendante se déroule en trois étapes. La première consiste à réaliser le noyau fonctionnel de l'application. Dans la seconde, le concepteur crée la couche de présentation en instanciant différents widgets. Enfin, dans la troisième, il relie la présentation au noyau fonctionnel en utilisant les fonctions de rappel.
- **L'approche descendante** permet de décrire l'interface de l'application en utilisant un langage spécifique. Cette description est ensuite utilisée pour générer le code du contrôleur de dialogue ainsi que celui de la présentation. Dans cette approche, pour créer une application, le concepteur crée le noyau fonctionnel de l'application. Puis, il décrit l'interface à l'aide d'un langage spécialisé. Enfin, il lie le code généré au noyau fonctionnel.

L'approche ascendante est plus générique que l'approche descendante. En effet, les générateurs descendants sont souvent spécialisés dans la description d'un type d'application : un générateur descendant capable de créer des applications de dessin ne pourra pas générer

d'application de traitement de texte et vice versa, alors qu'une même boîte à outils de présentation peut être utilisée pour décrire n'importe quel type d'application.

Par contre, l'approche ascendante n'est pas du tout adaptée à la description du contrôleur de dialogue des applications de CAO. Ces applications supportent en effet un dialogue structuré pour lequel la réponse à un évènement dépend de l'état du dialogue qui est fonction de la suite des évènements reçus. L'approche descendante offre des méthodes et des outils pour décrire explicitement le contrôleur de dialogue d'une application, ce qui est nécessaire lorsque le dialogue est structuré [Pierra 1995].

Dans ce chapitre, nous proposons une approche permettant de répondre à la question suivante:

Comment permettre à un concepteur de créer un contrôleur de dialogue supportant le dialogue structuré en gardant l'approche ascendante ?

La méthode que nous proposons consiste à ajouter aux boîtes à outils usuelles un certain nombre d'éléments qui sont non pas des réifications d'éléments de présentation mais des réifications d'éléments de dialogue. Pour notre étude, nous avons complété la boîte à outils par des éléments de dialogue issus du modèle H^4 [Guittet 1995]. Ces éléments sont instanciés et gérés d'une manière similaire à la gestion des widgets.

Nous commençons notre étude par un rappel des différents éléments utilisés pour spécifier le contrôleur de dialogue dans l'architecture H^4. Puis nous détaillons leur traduction dans la boîte à outils du dialogue. Ensuite, nous définissons les différents outils que nous proposons d'associer aux classes de cette bibliothèque pour permettre de contrôler finement le dialogue entre l'utilisateur et l'application. Enfin, nous montrons, sur un exemple simple, comment concevoir une application en utilisant notre concept de boîte à outils de dialogue.

2 Le contrôleur de dialogue de H^4

Ainsi que nous l'avons vu dans le chapitre I, le modèle d'architecture H^4 a été créé pour répondre aux besoins des concepteurs de systèmes de CAO en termes d'architecture logicielle. Il précise notamment la structure du contrôleur de dialogue qui doit être à même de supporter un dialogue structuré où le résultat d'une tâche peut être utilisé par une autre tâche et où les tâches portent sur plusieurs objets. Par exemple, lors de la réalisation d'une tâche de création d'un cercle centré sur l'intersection de deux droites, la tâche de création du cercle utilise le résultat de la tâche intersection qui se sert de deux objets (les deux droites) pour s'exécuter.

2.1 Les jetons

Les jetons représentent les unités d'information utilisées par les tâches dans H^4. On distingue deux types de jetons :

- une commande contient le noms de la tâche à réaliser, et est représentée dans l'interface par un bouton ou par une case de menu permettant à l'utilisateur de désigner quelle est la tâche que le système doit réaliser,
- un paramètre est utilisé par une tâche pour s'exécuter. Il est défini par le type de données qu'il représente, et contient une valeur de ce type.

2.2 Les questionnaires

Les questionnaires du modèle d'architecture H^4 représentent les abstractions des tâches du système au niveau du contrôleur de dialogue. Ce sont des signatures de fonctions invoquées par le contrôleur de dialogue pour réaliser des traitements sur le noyau fonctionnel. Ils possèdent, en entrée, une liste de jetons paramètres, et en sortie un éventuel jeton paramètre représentant le résultat de la tâche pour le contrôleur de dialogue. A chaque fois qu'un questionnaire est invoqué, il émet un compte rendu qui indique au contrôleur de dialogue si la tâche a été correctement effectuée.

2.3 Les interacteurs de contrôle

Les interacteurs de contrôle, ou plus simplement interacteurs, regroupent les différents questionnaires représentant des tâches d'un même niveau d'abstraction. Chaque interacteur contient un réseau de transitions augmenté (RTA, ou ATN en anglais) [Woods 1970] chargé de conserver les jetons qu'il reçoit et d'appeler les questionnaires en leur fournissant la liste des jetons nécessaires.

2.4 Le moniteur

L'organisation hiérarchique des tâches est à la charge du moniteur. Cette hiérarchie se présente sous la forme d'une hiérarchie d'interacteurs. Les interacteurs sont organisés de bas en haut en suivant un ordre croissant correspondant à un choix du concepteur sur les niveaux d'abstraction. Le moniteur est chargé de récupérer les jetons venant de la couche d'entrée-sortie et de les transmettre aux interacteurs en suivant la hiérarchie. L'indépendance entre les tâches vient du fait qu'une tâche (représentée par un questionnaire) ne sait pas d'où vient le

jeton que lui fourni le moniteur par l'intermédiaire de l'interacteur. Elle ignore aussi comment sera utilisé le jeton qu'elle produit.

Nous avons vu ici les différents éléments qui permettent de réaliser le contrôleur de dialogue d'une application supportant des tâches structurées tels qu'ils sont décrits dans le modèle d'architecture H^4 . Dans la suite de ce chapitre, nous proposons de définir un nouveau type de boîte à outils, qualifiée de boîte à outils du dialogue [Texier et Guittet 1998, Texier et Guittet 1999a], par réification de ces divers éléments.

3 Principe et cahier des charges

Le but de la boîte à outils du dialogue que nous proposons est de permettre aux concepteurs d'applications graphiques interactives de créer explicitement le contrôleur de dialogue de leur application de la même manière que pour la réalisation de la couche de présentation.

Avec une boîte à outils, la couche de présentation est créée en instanciant des éléments de présentation, la boîte à outils du dialogue va donc être composée de classes d'objets représentant des éléments de contrôle du dialogue décrits dans le modèle d'architecture H^4 .

Dans la première partie de cette section, nous décrivons le principe de base utilisé par la boîte à outils du dialogue pour décrire le contrôleur de dialogue de l'application. Dans la seconde partie, nous décrivons la spécification de la boîte à outils du dialogue en nous basant sur les modes de dialogue rencontrés dans les systèmes CAO.

3.1 Principe de base

Une des difficultés de la mise en œuvre des principes décrits dans H^4 est la définition des interacteurs. En effet, la structure sous-jacente d'un interacteur est un réseau de transitions augmenté. Or, la création de tels automates, si elle est assez simple, reste longue pour le concepteur. Pour un système de CAO standard, possédant une cinquantaine d'actions constructives (donc du même niveau d'abstraction) avec en moyenne deux paramètres chacune, l'automate résultant posséderait environ 2x50 états et 2x50x50 transitions (car l'on peut toujours cesser une action constructive)!

Nous sommes partis du principe que les interacteurs avaient la charge des appels des questionnaires. Un appel est réalisé lorsque l'interacteur a reçu les données nécessaires, c'est-à-dire lorsqu'il a reçu le nom de la commande correspondant au questionnaire (sous la forme d'un jeton commande) ainsi que la suite des paramètres utilisés par le questionnaire (sous la

forme de jetons paramètres). L'automate est utilisé afin de contrôler que les jetons reçus sont bien ceux qui manquaient pour l'appel d'un questionnaire. On s'aperçoit donc que si l'on spécifie un questionnaire en termes de commande d'activation et de natures de jetons paramètres, la suite de transitions permettant son appel est strictement déterminée par la suite de ses paramètres.

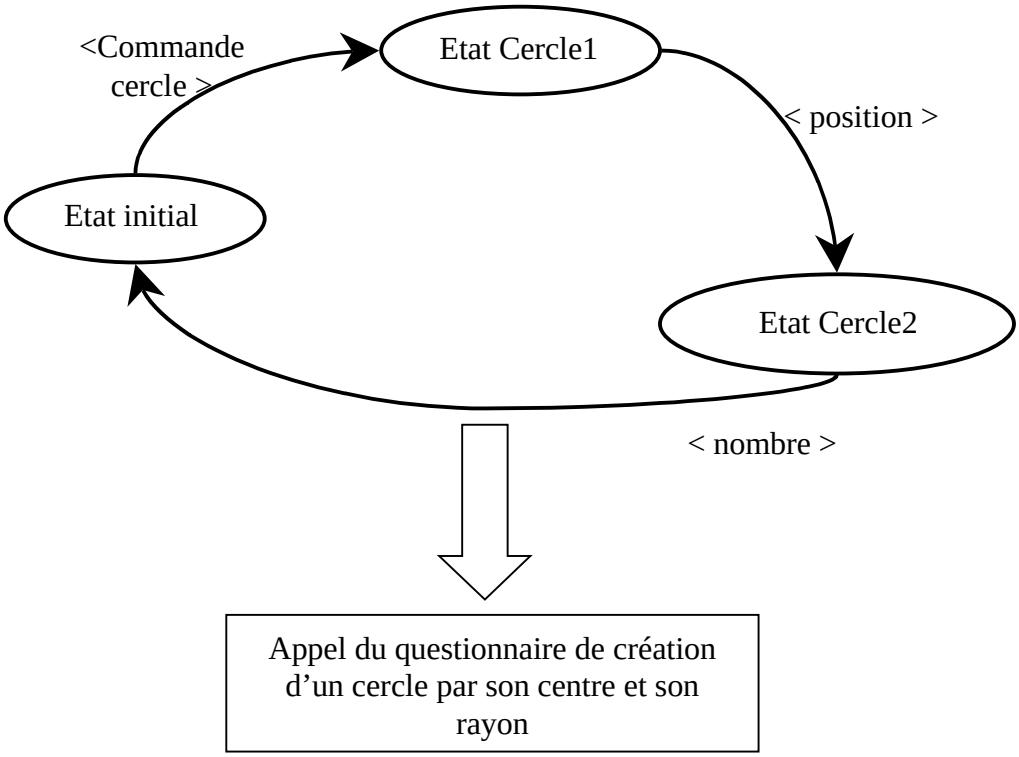


Figure II.1: Automate de création de cercle

La Figure II.1 présente un automate responsable de l'appel d'un questionnaire de création d'un cercle par son centre et son rayon. La valeur du centre sera contenue dans un jeton de type position et le rayon dans un jeton de type nombre. La première transition se fait sur un jeton commande représentant la commande cercle permettant à l'utilisateur de spécifier au système qu'il veut créer un cercle. L'enchaînement des transitions amenant à la création du cercle est <Commande cercle>, <position>, <nombre>. Cela correspond exactement à la spécification du questionnaire. Il est donc possible de générer automatiquement l'automate à partir de cette spécification.

Nous pouvons donc énoncer le principe de base de la boîte à outils du dialogue :

La construction du contrôleur de dialogue est réalisée à partir d'objets représentant la signature des fonctions qui constituent les tâches du système.

3.2 Analyse des besions

Pour identifier les besoins auxquels doit répondre la boîte à outils du dialogue, nous nous basons sur l'étude des différents types de tâches rencontrées dans les systèmes CAO. Nous en déduisons les différents types d'outils nécessaires dans la boîte à outils du dialogue, ainsi que certaines propriétés de ces outils, nécessaires pour décrire complètement le contrôleur de dialogue d'un système CAO.

On peut distinguer trois types de tâches.

Les tâches simples représentent les tâches du système décrites sous la forme de questionnaires dans H***4*** , c'est-à-dire par :

- le nom de la commande d'activation,
- une suite finie de paramètres d'entrée.

Les tâches répétitives permettent de traiter des listes de paramètres dont le nombre n'est pas connu au moment de l'initialisation de la tâche. Elles font appel à trois actions différentes comme les répétitions dans les langages de programmation : une action d'initialisation, une action d'itération et une action de fin. Par exemple, elles permettent de traiter des tâches comme la création d'une polyligne Figure II.2 où l'utilisateur fournit un premier point (action d'initialisation) puis, à chaque nouveau point qu'il donne un segment est créé entre le dernier point et le point précédent (action d'itération) jusqu'à ce qu'il indique au système qu'il a fini (action de fin).

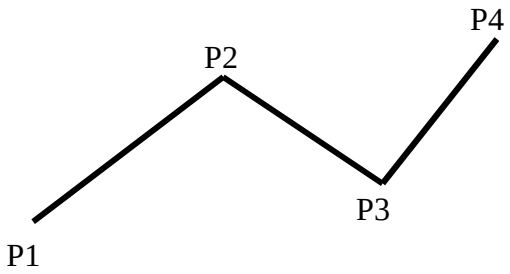


Figure II.2: création d'une polyligne

Les tâches récursives décrivent des tâches dont le résultat peut être utilisé par une autre tâche du même niveau d'abstraction. Elles sont utilisées dans le cadre des expressions grapho-numériques [Gardan 1991] comme la multiplication qui peut utiliser le résultat d'une autre multiplication. Du point de vue de H^4, cela signifie qu'un interacteur gérant des tâches

récursives peut avoir plusieurs instances d'une même tâche actives en même temps. On dit d'une tâche (représentée par un questionnaire) qu'elle est **active** lorsque la commande la représentant a été fournie à l'interacteur, et que celui-ci n'a pas encore reçu tous les paramètres permettant de l'exécuter.

En plus de ces différents types de tâches, le mode de dialogue des systèmes CAO possède deux grandes particularités dues au fait qu'il supporte des tâches structurées et multi-objets :

1. Le dialogue est préfixé, donc, pour réaliser une action, l'utilisateur indique d'abord la commande puis les opérandes. Si de nombreuses discussions ont eu lieu sur les mérites respectifs de ces deux approches du point de vue ergonomique [Meinadier 1991], il apparaît que le mode postfixé (opérande commande) s'adapte mieux aux tâches mono-objet alors que le dialogue préfixé est préférable pour les applications supportant des tâches multi-objets [Guittet 1995].
2. L'ordre des paramètres décrit dans la spécification des tâches ne doit pas être imposé : il est en effet souhaitable que l'utilisateur puisse fournir les paramètres d'une action dans n'importe quel ordre pour peu que la nature de ses paramètres soit différente. Par exemple, pour créer un cercle par centre et rayon, l'utilisateur doit pouvoir fournir, soit le rayon en premier, dans ce cas la tâche cercle par centre et rayon est équivalente à la tâche cercle par rayon et centre, soit le centre en premier. Par contre, si une tâche possède plusieurs paramètres du même type, leur ordre d'interprétation est prédéfini (en l'absence de mécanisme permettant à l'utilisateur de spécifier l'interprétation voulue). Par exemple, lorsque l'utilisateur crée un cercle par centre et point de passage, le premier point qu'il fournit est toujours considéré comme le centre du cercle. Il ne pourra jamais créer un cercle par point de passage et centre en utilisant la même commande.

En résumé, la boîte à outils du dialogue doit offrir la possibilité aux concepteurs de créer un contrôleur de dialogue capable de manipuler des tâches simples, des tâches répétitives et des tâches récursives. De plus, elle doit prendre en compte le fait que le dialogue est préfixé et que l'ordre des paramètres à fournir pour réaliser une tâche doit être libre.

4 Composition de la boîte à outils du dialogue

Le but de la boîte à outils du dialogue est de permettre aux concepteurs d'applications graphiques interactives de réaliser le contrôleur de dialogue de leur application de la même

manière qu'ils créent la couche de présentation. Comme nous l'avons vu, la couche de présentation est créée en instanciant des objets qui sont des réifications d'éléments de présentation. De la même façon, la boîte à outils du dialogue va être basée sur la réification des éléments du contrôleur de dialogue décrits dans H***4*** .

4.1 Les jetons

La boîte à outils du dialogue offre un certain nombre de classes de jetons, et permet au concepteur d'en définir de nouvelles. On distingue deux grandes classes de jetons : commandes et paramètres.

- La classe commande représente, comme décrit dans H^4, l'intention de l'utilisateur. La classe commande ne peut pas être dérivée. Un jeton commande est instancié en fournissant le nom de la commande qu'ils représentent.
- La classe paramètre est une classe abstraite. Elle permet au concepteur de créer une hiérarchie de classes de paramètres. Pour créer une sous-classe de paramètres, le concepteur doit fournir le type de données référencées et la classe dont elle hérite.

La hiérarchie des jetons paramètres est une notion nouvelle par rapport aux implantations habituelles du modèle H^4. Elle permet de factoriser l'écriture de la spécification des questionnaires. Par exemple, sur la Figure II.3, la classe *Objet* est dérivée en deux sous classes *Cercle* et *Droite*. Donc, s'il existe dans l'application une fonction qui agit indifféremment sur les cercles ou les droites (par exemple pour changer la couleur d'un objet), le concepteur créera un seul questionnaire avec, en entrée, un jeton *objet* plutôt que deux questionnaires, l'un utilisant des jetons *droite* et l'autre des jetons *cercle*.

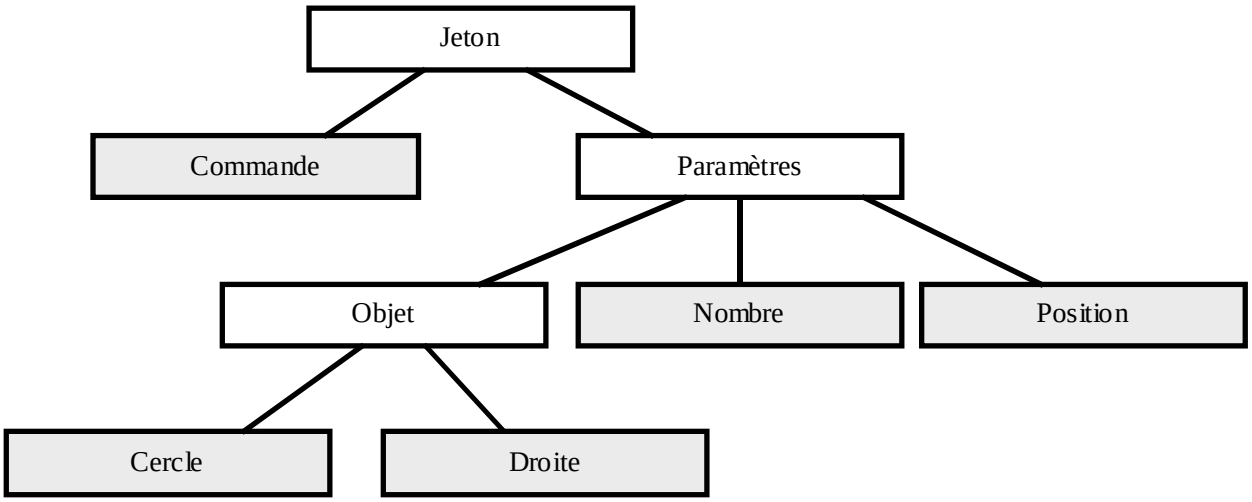


Figure II.3 : la hiérarchie des jetons

La Figure II.3 montre un exemple de hiérarchie possible pour les jetons d'une application. Plusieurs remarques sur cette hiérarchie apparaissent:

- Seule la classe de jetons paramètres peut être dérivée par le concepteur. La classe de jetons commandes correspond aux jetons commandes tels qu'ils sont décrits dans H[4] . Ces jetons servent à exprimer, au niveau du contrôle de dialogue, les intentions de l'utilisateur. Pour cela, ils contiennent le nom de la tâche à réaliser. Ils ne sont jamais utilisés comme opérandes d'une action du système mais toujours comme opérateur.
- La hiérarchie des jetons est utilisée au moment de la définition du contrôleur de dialogue, et plus particulièrement au moment de la définition des questionnaires. Il est important de noter que seules les classes terminales (en gris sur la figure) seront effectivement instanciées au moment de l'exécution.

4.2 Les questionnaires

Les questionnaires permettent de décrire la spécification des tâches de l'application. Les automates contenus dans les diagets (outils de dialogue) sont générés à partir de ces spécification.

La boîte à outils du dialogue propose deux classes de tâches différentes correspondant à deux types de tâches décrites au paragraphe 3.2:

- les tâches simples, représentées sous la forme d'une classe *questionnaire,*
- les tâches répétitives représentées sous la forme d'une classe *questionnaire_repetitif.*

4.2.1 La classe questionnaire

La classe questionnaire permet d'instancier des questionnaires représentant des tâches simples. Un questionnaire est défini par le nom de sa commande d'activation et par la liste des types de jetons d'entrée. La classe questionnaire possède deux attributs (le dernier est un attribut facultatif) :

- L'attribut « fonction » est utilisé pour spécifier le nom de la fonction appelée lors de l'exécution du questionnaire.
- L'attribut « sortie » permet de spécifier le type du jeton de sortie si le questionnaire représente une sous-tâche de production.

4.2.2 La classe questionnaire répétitif

Cette classe, sous-classe de la classe questionnaire, permet d'instancier des tâches répétitives. Comme la classe questionnaire, elle possède un nom de commande d'activation et une liste de types de jetons d'entrée. Le dernier de ces jetons est considéré comme celui qui sera répété. En plus, elle possède plusieurs attributs permettant de traiter la liste des jetons reçus par ses instances :

- L'attribut « fonction d'itération » permet de spécifier la fonction qui est appelée lors de chaque itération (c'est-à-dire à chaque fois que le dernier jeton défini dans la liste des jetons d'entrée sera fourni). Cette fonction a, en entrée, le jeton répétitif.
- L'attribut « commande de fin » est utilisé pour spécifier le nom de la commande utilisée pour terminer l'itération.
- L'attribut « fonction de fin » spécifie la fonction appelée à la fin de l'itération. Elle ne possède pas d'entrée, et peut éventuellement avoir un jeton en sortie si le questionnaire correspond à une sous-tâche de production. Dans ce cas, le type de jeton de sortie est spécifié par l'attribut « *Sortie* » hérité de la classe *Questionnaire*.

Il est important de noter que la « fonction d'initialisation » est spécifiée grâce à l'attribut « fonction » hérité de la classe *Questionnaire*.

4.2.3 Règles de surcharge

Dans les systèmes CAO, il existe souvent plusieurs façons de réaliser une opération. A titre d'exemple (Figure II.4), une droite peut être construite à partir de deux points de passage ou à

partir d'un point et d'un angle, etc. Dans ce cas, il nous apparaît souhaitable que le système ne possède qu'une seule commande *Droite* et que la méthode de construction soit déterminée par la suite de données qui arrive au système. Ce type de contrôle est connu sous le nom de pilotage par les données. La réaction du système dépend de la suite des types de valeurs qui arrivent en entrée.

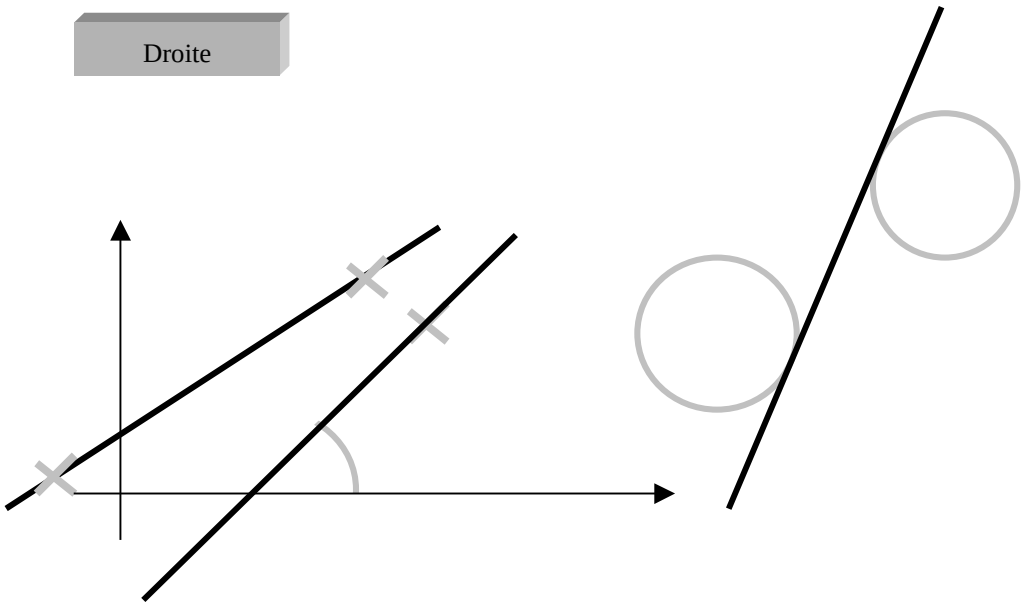


Figure II.4: Création de droites

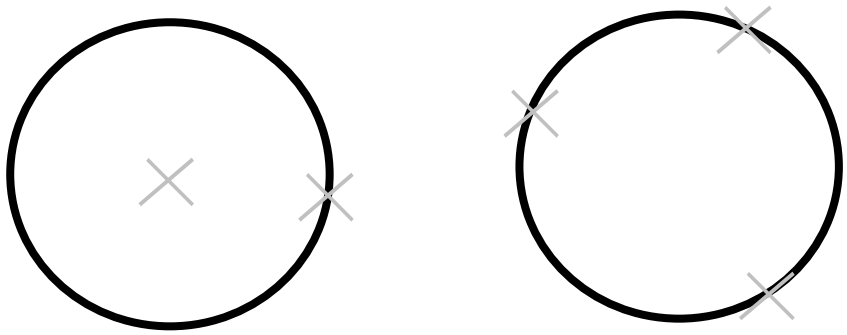
Au niveau de la définition du contrôleur de dialogue, le pilotage par les données se traduit par la surcharge des questionnaires. Cela signifie que deux (ou plus) questionnaires peuvent avoir le même nom, donc la même commande d'activation. Ainsi, toutes les opérations ayant la même sémantique peuvent être représentées par une seule commande au niveau de la présentation. Le système doit alors être capable de retrouver le questionnaire à appeler lors de l'exécution en connaissant la suite des jetons qu'il a reçue après la commande.

Cependant, surcharger les questionnaires peut conduire à des ambiguïtés qui font, qu'à certains moments, le système ne sait pas quel questionnaire il doit exécuter.

Le contrôleur de dialogue différencie les questionnaires par leurs noms et par la suite des types de jetons d'entrée. Il est impossible pour lui de différencier deux questionnaires en se basant sur le type de jetons de sortie. Comme pour tous les langages autorisant la surcharge, deux questionnaires ayant le même nom ne devront pas avoir la même suite de jetons d'entrée.

En fait, les contraintes de surcharge sont un peu plus fortes. Pour une plus grande souplesse du système, le modèle H^4 préconise de permettre à l'utilisateur de fournir les opérandes d'une action dans un ordre indifférent si elles sont de types différents. Par exemple, pour construire un cercle par centre et rayon, l'utilisateur peut fournir dans n'importe quel ordre la position du centre et la valeur du rayon (soit position nombre, soit nombre position).

De plus, le même objet peut parfois être créé avec un nombre de paramètres différents suivant le type de construction choisi. Par exemple (Figure II.5), un cercle peut être créé par son centre et un point de passage (2 paramètres de type position) ou par trois points de passages (3 paramètres de type position). Dans ce cas que se passe-t-il lorsque deux positions sont arrivées au système après la commande cercle ? Le système doit-il attendre un troisième point ou doit-il créer un cercle par son centre et par un point de passage ?



Cercle par deux points

Cercle par trois points

Figure II.5: création de cercle par deux ou trois points

Nous pouvons maintenant décrire la règle de surcharge des questionnaires permettant de lever toutes les ambiguïtés au moment de l'exécution :

Règle de surcharge des questionnaires : *Deux questionnaires d'un même interacteur peuvent avoir le même nom, si l'ensemble des types de jetons d'entrée de l'un n'est pas inclus dans l'ensemble des jetons d'entrée de l'autre, et ce, quel que soit l'ordre des types de jetons dans chacune des deux listes.*

4.2.4 Surcharge des questionnaires et hiérarchie de jetons

La surcharge des questionnaires et la hiérarchie des jetons n'ont pas été créées dans le même but et n'ont donc pas le même rôle.

- La surcharge des questionnaires permet d'appeler des questionnaires différents qui ont le même résultat mais dont les paramètres d'entrée sont différents.

- L'utilisation d'un type de jetons abstrait (comme le type *objet* de la Figure II.3) dans la définition d'un questionnaire, signifie que le même traitement sera effectué sur tous les types de jetons dérivés de ce type de jetons abstrait.

Il est possible de surcharger un questionnaire qui a un type abstrait en entrée par un questionnaire ayant un de ses sous-types en entrée. Au moment de l'exécution, le questionnaire le plus spécifique possible sera appelé conformément au principe de liaison tardive des langages à objets. Prenons l'exemple d'une classe de jetons *objet* décomposée en deux sous-classes *droite* et *cercle* (voir Figure II.3). On définit deux questionnaires Q1 et Q2 ayant le même nom, Q1 a, en entrée, un jeton de type *objet*, Q2 a, en entrée, un jeton *droite* (spécialisation de *objet*). En phase d'exécution, le questionnaire Q2 est appelé dans le cas où le système reçoit un jeton *droite* après avoir reçu la commande d'activation de Q1 et Q2. Le questionnaire Q1 est appelé lorsque le système reçoit n'importe quel autre *objet*.

4.3 Les diagets

Les diagets sont des réifications des interacteurs de contrôle de H[4] . Nous avons défini ce nom par similitude entre eux et les widgets de la présentation. Les diagets permettent l'organisation hiérarchique des tâches du système décrites sous la forme de questionnaires. Les diagets sont chargés de conserver et d'organiser les jetons qu'ils reçoivent entre deux appels de questionnaires afin de les transmettre aux questionnaires.

La boîte à outils du dialogue possède deux types de diagets : d'une part, les diagets simples (appelés diaget dans la suite) qui n'ont qu'un seul questionnaire actif à la fois lors de l'exécution, et d'autre part, les diagets récursifs qui permettent de modéliser les tâches récursives. Ces derniers ont donc plusieurs questionnaires actifs en même temps.

4.3.1 La classe diaget

La classe diaget permet de définir les diagets non récursifs. Ils possèdent un nom et la liste des questionnaires qu'il permettent d'appeler. Pour ajouter un questionnaire à cette liste, le concepteur se sert de la méthode *ajoute_Q*. Cette méthode est chargée, entre autre, de la vérification de la règle de surcharge des questionnaires. Si le nouveau questionnaire respecte cette règle, il est pris en compte par le diaget. Sinon le diaget refuse le questionnaire et ce dernier ne sera jamais appelé.

La structure sous-jacente des diagets est un réseau de transitions augmenté. Chaque diaget possède un nombre fini d'états reliés entre eux par des transitions. Une transition est définie

par un état de départ, un état d'arrivée, et est étiquetée par la nature du jeton qui la valide. Lorsqu'un jeton permet de valider une transition lors de l'exécution, on dit que ce jeton est consommé.

Le rôle des diagets est de contrôler les appels des questionnaires. Pour cela, le diaget conserve les jetons qu'il consomme dans son registre jusqu'à ce qu'un questionnaire puisse être appelé (lorsque tous les paramètres du questionnaire ont été reçus par le diaget). Alors, le diaget réalise cet appel, en fournissant au questionnaire les jetons conservés dans son registre après ré-ordonnancement (en effet, l'ordre des paramètres fournis par l'utilisateur n'est pas obligatoirement le même que celui des paramètres d'entrée du questionnaire).

4.3.2 Génération de l'automate

L'une des difficultés de la mise en œuvre du modèle d'architecture H^4 est la définition des automates contenus dans les interacteurs. Construire « manuellement » les automates contrôlant un système CAO demande beaucoup de temps et est souvent source d'erreur. La boîte à outils du dialogue propose de réduire le temps et les erreurs de conception en générant automatiquement les automates à partir des spécifications des questionnaires qu'ils permettent d'appeler. Nous distinguons deux cas de génération différents :

- Génération des automates contrôlant des questionnaires simples
- Génération des automates contrôlant des questionnaires répétitifs

4.3.2.1 Génération à partir de questionnaires simples

Pour créer l'automate à partir de la description d'un questionnaire, le générateur va procéder en trois étapes :

1. Il crée d'abord une transition partant de l'état initial vers un état appelé *état de commande*. Cette transition est validée par le jeton commande qui encapsule le nom du questionnaire.
2. Il génère les transitions et les états permettant de récupérer l'ensemble des paramètres nécessaires à l'appel des questionnaires. Lors de cette phase de génération, il prend en compte toutes les combinaisons possibles entre les types de jetons différents.
3. Il ajoute aux transitions sur le dernier paramètre l'appel du questionnaire. L'état d'arrivée du questionnaire est remplacé par l'état initial. Cela signifie qu'à chaque fois qu'un diaget appelle un questionnaire, il revient dans l'état initial.

A chaque fois qu'un questionnaire est ajouté à un diaget, l'automate sous-jacent est modifié. Dans le but de réduire le nombre d'états, le générateur réutilise autant que possible les états existants. Ainsi, lorsque qu'un questionnaire ayant le même nom qu'un autre est ajouté, le générateur ne crée pas de nouvel état de commande mais il réutilise l'ancien, et lui ajoute des transitions si nécessaire. Ainsi, nous utilisons un algorithme optimal par construction pour générer les automates. Cet algorithme est moins coûteux en temps qu'un algorithme d'optimisation [Autebert 1994].

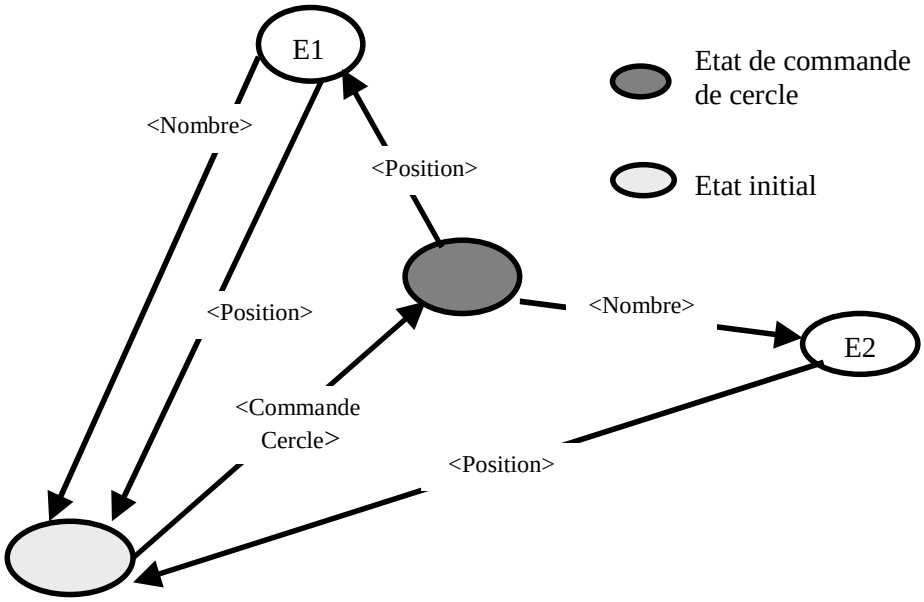


Figure II.6: Génération d'un automate

La Figure II.6 montre un automate généré à partir de deux questionnaires :

- questionnaire Q1 (« cercle »,{Position, Nombre})
- questionnaire Q2 (« cercle », {Position, Position})

Les questionnaires ci-dessus sont définis par leur nom (entre guillemets) et par la liste des types de jetons d'entrée (entre accolades). On remarque que le générateur réutilise le maximum d'états et de transitions déjà créés. Lors de l'ajout du questionnaire Q2, il ajoute une seule transition entre l'état E1 et l'état initial. On voit ici que la sémantique des deux positions de Q2 ne peut être inversée pour être compatible avec celle de Q1. La première position doit être le centre du cercle.

4.3.2.2 Génération à partir de questionnaires répétitifs

L'ajout d'un questionnaire répétitif entraîne la génération d'une partie d'automate différente de celle décrite ci dessus. Cette partie d'automate doit gérer l'appel de trois fonctions

différentes (la fonction d'initialisation, la fonction d'itération et la fonction de fin), et le fait que la transition sur le jeton répété conduise l'automate dans le même état.

Le générateur d'automates fonctionne de la même manière que précédemment pour créer la partie contrôlant l'appel de la fonction d'initialisation. Puis, il crée une transition répétitive dont l'état de départ est le même que l'état d'arrivée. Cet état est appelé *état de répétition*. Enfin, il va créer une transition allant de l'état de répétition jusqu'à l'état initial. Cette transition est validée par la commande de fin définie dans le questionnaire, et appelle la fonction de fin du questionnaire. La Figure II.7 montre l'automate généré à partir d'un questionnaire de création d'une polyligne.

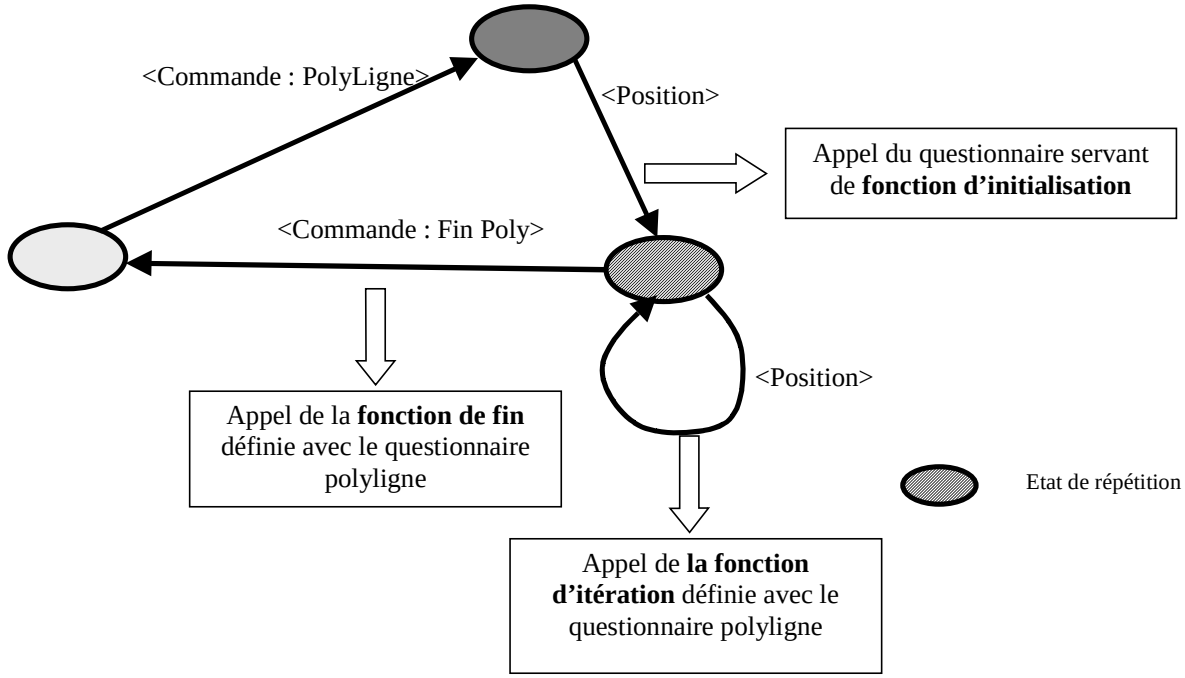


Figure II.7: Génération d'un automate à partir d'un questionnaire répétitif

4.3.3 Diaget récursif

La classe diaget récursif permet d'instancier des diagets qui gèrent l'appel de tâches récursives. Cette classe est issue de la classe diaget. Donc tous les diagets récursifs possèdent un nom et une liste de questionnaires. La différence entre les diagets et les diagets récursifs réside dans le fait que ces derniers permettent d'avoir plusieurs instances de questionnaires actives en même temps, le résultat d'une instance étant utilisé par une autre. Par exemple, le questionnaire multiplication, qui possède deux nombres en entrée, peut utiliser le résultat d'une autre multiplication comme l'une de ses entrées. Il est donc nécessaire que plusieurs

instances du questionnaire multiplication soient actives en même temps et que le résultat de l'une d'elles puisse être transmis à l'autre.

Le diaget récursif ne génère pas (statiquement) d'automate à chaque fois qu'un questionnaire est ajouté. Il génère dynamiquement un automate (en utilisant toutes les surcharges d'un questionnaire) à chaque fois que la commande d'activation du questionnaire est fournie par l'utilisateur. Le mécanisme de génération dynamique est le même que celui décrit ci-dessus dans les cas statiques et il dépend du type de questionnaire (simple ou répétitif). Dès qu'il a été généré, l'automate passe immédiatement dans l'état de commande. Il est alors ajouté à la fin d'une pile d'automates. Lorsqu'il a reçu tous les paramètres permettant d'appeler le questionnaire, il réalise cet appel et il fournit le résultat du questionnaire à l'automate précédent dans la pile. Il est alors désactivé, c'est-à-dire que le diaget le supprime de la pile puis le détruit. Si on reprend l'exemple de la multiplication, le diaget va créer deux automates permettant d'appeler deux fois le questionnaire multiplication. Le résultat du premier appel est utilisé pour réaliser le second appel du questionnaire.

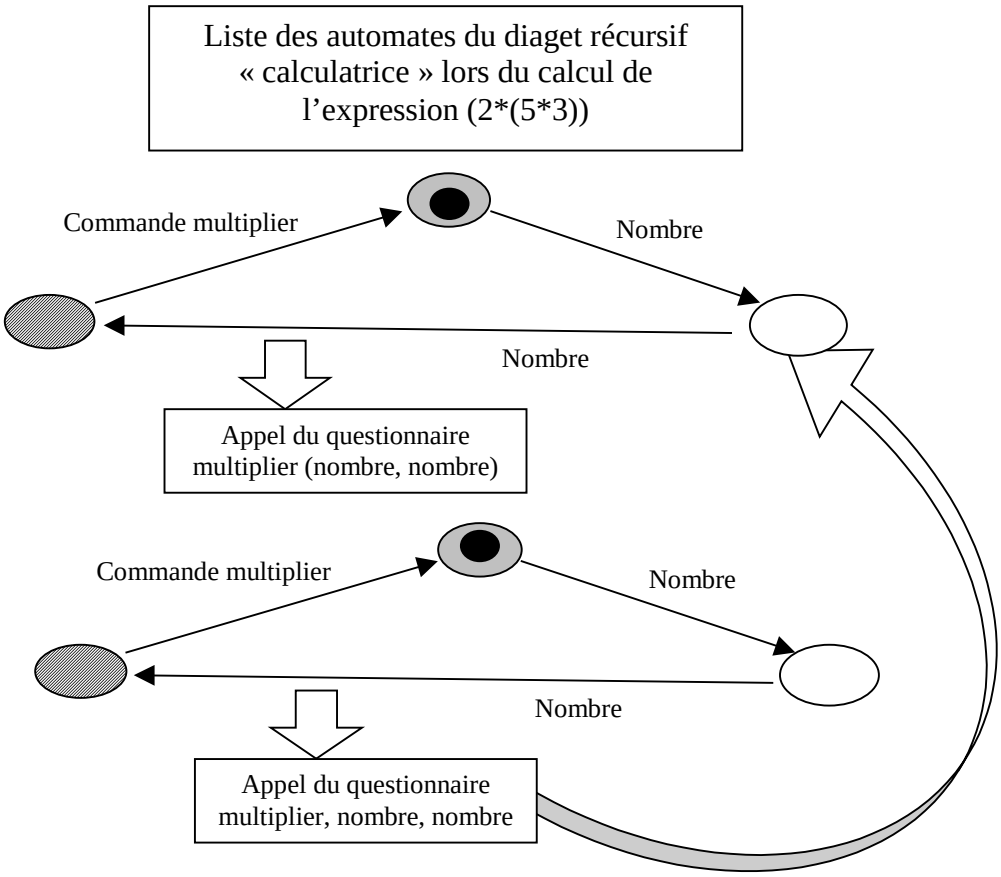


Figure II.8: les diagets récursifs

La Figure II.8 montre la pile des automates d'un diaget récursif lors du calcul d'une expression (2x(5x3)). Cette expression est en fait écrite en préfixé * 2 * 5 3. Le résultat de 5 * 3 est utilisé comme deuxième paramètre par le premier automate de la liste (le plus haut sur le dessin de la Figure II.8).

4.4 Le moniteur

La classe moniteur est chargée de l'organisation hiérarchique des diagets comme cela est décrit dans le modèle H^4 . Elle possède donc une liste de diagets, le premier diaget de la liste est le plus haut dans la hiérarchie. Le moniteur récupère les jetons venant de la couche d'entrée-sortie et les transmet de diaget en diaget jusqu'au dernier, ou jusqu'à ce qu'il n'y ait plus de production. Il est important de noter qu'il n'y a qu'un seul moniteur par application.

4.5 Synthèse

Le schéma EXPRESS[2] [Schenck et Wilson 1994] (Figure II.9) montre les relations entre les différentes classes de la boîte à outils du dialogue. Les traits fin représentent des relations d'agrégation (La classe *Diaget* possède un *nom*). Les traits épais représentent de relations d'héritage (La classe *Diaget récursif* hérite de la classe *Diaget*). On notera que la classe moniteur utilise une liste (mot clé List) de diagets. Ces derniers sont soit des diagets simples (diagets), soit des diagets récursifs utilisés pour modéliser des tâches récursives. Chaque diaget est chargé de contrôler un ensemble (mot clé Set) de questionnaires qui peuvent être simples ou répétitifs. Sur la Figure II.9, les rectangles représentent des classes, les rectangles barrés représentent les types de base, les traits fins modélisent des relations d'attributs et les traits épais spécifient des relations d'héritage. La classe *type de paramètres* représente les chaînes de caractères utilisée pour définir la nature des jetons paramètres utilisés par le contrôleur de dialogue de l'application.

[2] Une description du langage Express graphique est fournie en annexe de ce document

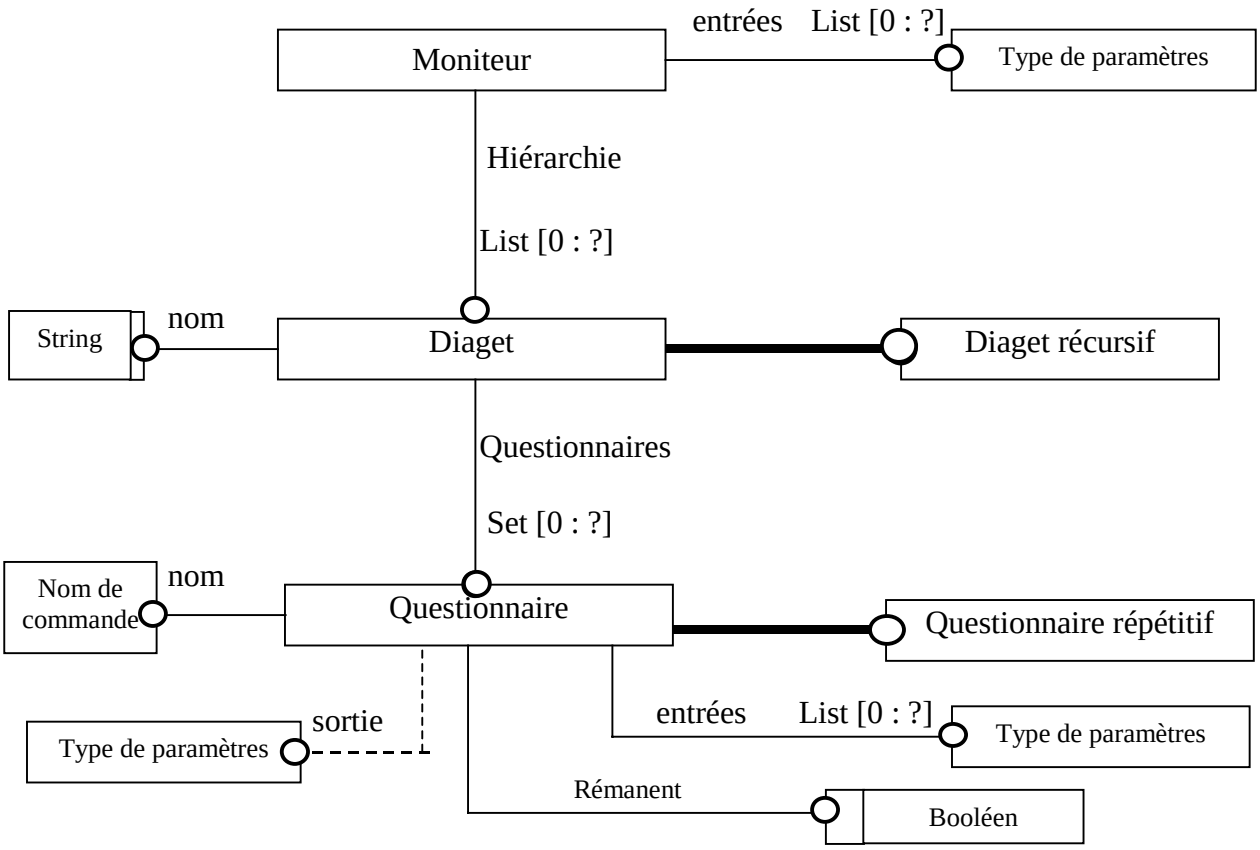


Figure II.9: Organisation de la boîte à outils du dialogue

De plus, ce schéma EXPRESS montre en outre les relations entre les jetons et les différents éléments de la boîte à outils du dialogue. On remarque que les questionnaires possèdent une liste de types de jetons paramètres représentant les paramètres d'entrée du questionnaire. L'attribut nom d'un questionnaire contient le nom d'un jeton commande. Le moniteur connaît la liste des types de jetons que la couche de bas niveau est susceptible de lui fournir. Cette liste est représentée par son attribut *entrées*.

Dans cette section, nous avons décrit explicitement la boîte à outils du dialogue. Elle est composée de plusieurs classes qui sont des réifications des éléments décrits dans l'architecture H^4. Cette boîte à outils permet de modéliser tous les types de tâches rencontrés dans les systèmes supportant des tâches structurées et multi-objets.

L'instanciation de ces classes permet au concepteur de définir explicitement le contrôleur de dialogue d'une application de la même façon qu'il décrit la couche de présentation à l'aide de widgets.

Dans la section suivante, nous décrivons différents outils réalisés dans le but d'aider les concepteurs d'applications graphiques interactives à définir plus finement le contrôle de l'application.

5 Contrôle de l'application

La boîte à outils du dialogue offre la possibilité au concepteur de créer le contrôleur de dialogue de l'application à partir de la simple description des tâches réalisées par le système. Le rôle principal du contrôleur de dialogue est de réaliser l'appel des actions du noyau fonctionnel en fonction des entrées de l'utilisateur. En fait, la boîte à outils du dialogue offre deux types de services complémentaires au concepteur :

- un mécanisme de contrôle des entrées de l'utilisateur permettant de gérer automatiquement les autorisations et les interdictions d'entrées particulières,
- différentes méthodes de correction d'erreurs.

5.1 Appels des actions

Le rôle du contrôleur de dialogue est de réaliser l'appel des actions de l'application en fonctions des données (commandes ou paramètres) fournies par l'utilisateur. Dans notre approche, cet appel est réalisé par l'intermédiaire des questionnaires. Dans cette section, nous commençons par expliquer comment fonctionne le contrôleur de dialogue, c'est-à-dire comment est réalisé l'appel des questionnaires. Puis, nous décrivons la notion de sous-questionnaire qui permettent de réaliser des actions systématiques selon les types de données fournies par l'utilisateur.

5.1.1 Fonctionnement du contrôleur de dialogue

Conformément au modèle d'architecture H^4 , le fonctionnement du contrôleur de dialogue consiste à véhiculer des jetons à travers la hiérarchie de diagets. Le moniteur récupère un jeton venant de la couche d'entrée/sortie (correspondant à une entrée autorisée de l'utilisateur), puis le transmet de diaget en diaget dans l'ordre de la hiérarchie jusqu'au dernier diaget. Lorsqu'un diaget reçoit un jeton, l'alternative suivante est offerte :

- soit le jeton lui permet de changer d'état, et il le conserve dans son registre (notion de consommation)
- soit le jeton ne lui permet pas de changer d'état, et il le rend au moniteur.

Un diaget appelle un questionnaire dès qu'il a reçu les jetons nécessaires pour réaliser cet appel (le nom du questionnaire et l'ensemble des paramètres). Si le questionnaire crée un nouveau jeton, ce dernier est fourni au moniteur pour qu'il le transmette au diaget au-dessus dans la hiérarchie (notion de production).

Il est important de noter que, si le compte-rendu d'un questionnaire est faux (i.e. la tâche modélisée par le questionnaire n'a pas pu être réalisée correctement), le diaget ne change pas d'état, et attend donc un nouveau jeton pour réaliser l'action qui n'a pu être réalisée.

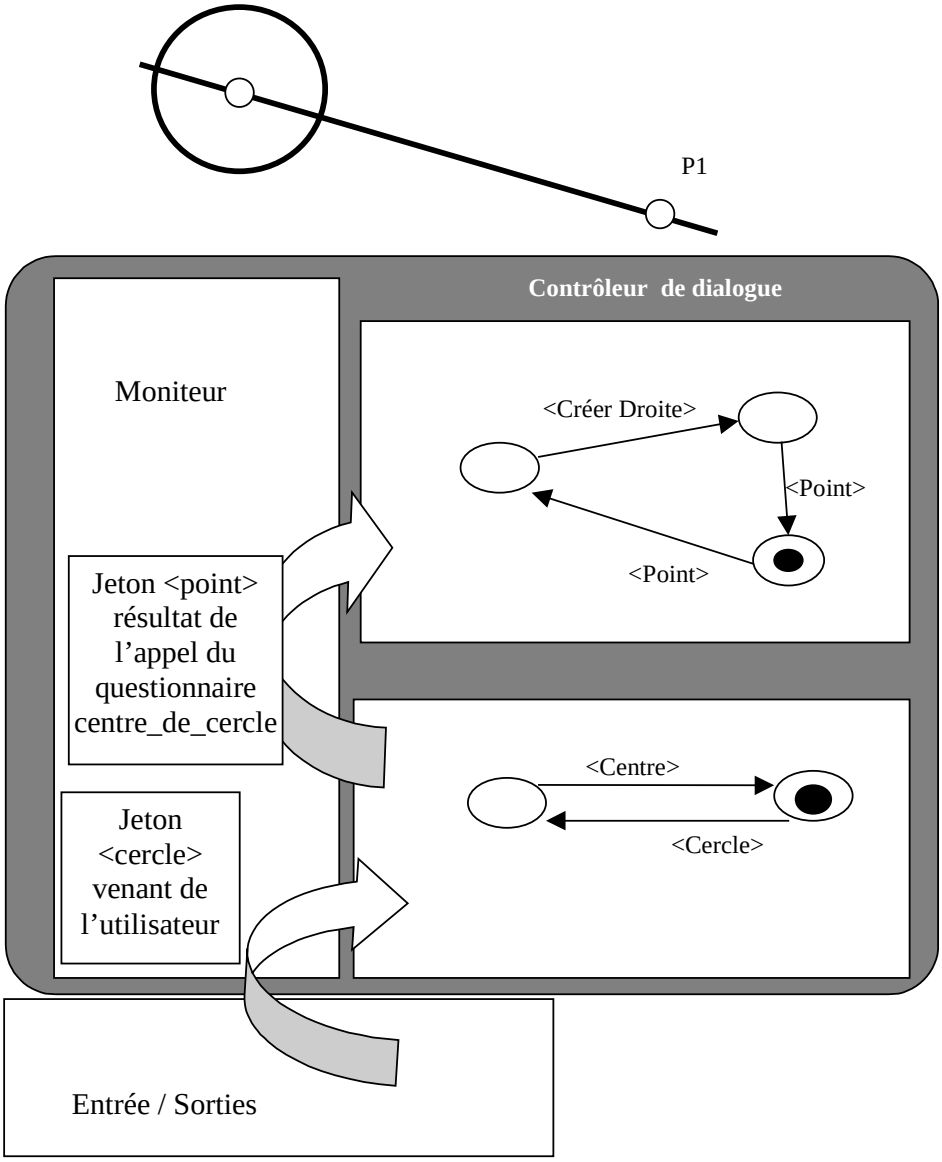


Figure II.10: Contrôle de l'appel des actions

La Figure II.10 montre le fonctionnement du contrôleur de dialogue lors de la création d'une droite passant par un point et par le centre d'un cercle. Le contrôleur de dialogue est composé de deux diagets :

- le premier (diaget de création) contient le questionnaire de création d'une droite par deux points,
- le second (diaget de sélection d'information) gère l'appel du questionnaire qui calcule le centre d'un cercle.

Sur l'exemple de la Figure II.10, l'utilisateur a fourni, dans l'ordre, la commande « créer droite », un jeton <point> P1 non consommé par le diaget de sélection d'information mais consommé par le diaget de création, puis la commande « centre ». A ce moment, le diaget de création attend une position pour créer la droite, et le diaget de sélection d'information attend un jeton cercle pour appeler le questionnaire de calcul du centre. Lorsque ce dernier reçoit un jeton cercle, il appelle le questionnaire centre de cercle qui retourne un jeton <point> représentant le centre du cercle. Ce jeton est alors envoyé au moniteur qui le transmet au diaget situé au-dessus. Le diaget de création utilise ce point pour appeler le questionnaire « créer droite » par deux points, la droite est alors créée et affichée.

5.1.2 Définition d'actions intermédiaires

L'automate sous-jacent des diagets est généré à partir de la spécification des questionnaires gérés. Cette méthode permet de créer très facilement le contrôleur de dialogue de l'application, sans que le concepteur n'ait à décrire d'automate. Un questionnaire est appelé une fois que le diaget a reçu tous les paramètres nécessaires. Cependant, il est parfois nécessaire que le système réalise une action avant que tous les paramètres ne soient arrivés. Cette action, appelée action intermédiaire, peut permettre :

- de réaliser un écho de l'état du système afin d'assister l'utilisateur dans la saisie des paramètres suivant, et/ou
- de vérifier la valeur de certains paramètres, et ainsi définir des pré-conditions partout sur les valeurs du registre de l'automate et garder certaines transitions. Par exemple, ce mécanisme permet de vérifier que le rayon défini pour un cercle est strictement positif lors de la création d'un cercle par son centre et rayon.

Pour permettre au concepteur de spécifier des actions intermédiaires, nous proposons d'introduire la notion de sous-questionnaire. Un sous-questionnaire est un questionnaire qui a le même nom que le questionnaire dont il est issu (le questionnaire père) et dont la liste de jetons d'entrée est contenue dans celle du questionnaire père.

Lorsqu'un sous-questionnaire est ajouté à un diaget, le système ne modifie pas l'automate du diaget, mais il calcule en fonction des paramètres d'entrées du sous-questionnaire sur quelles transitions il doit placer son appel. Cet appel est réalisé à chaque fois que l'interacteur obtient la liste des jetons d'entrée définie dans le sous-questionnaire. Il est donc impératif d'avoir associé le questionnaire père au diaget avant d'ajouter un de ses sous-questionnaires, afin que ce sous-questionnaire soit bien interprété comme une action intermédiaire.

Comme le questionnaire, le sous-questionnaire possède un compte-rendu en sortie. Si le compte rendu est faux, le diaget ne change pas d'état.

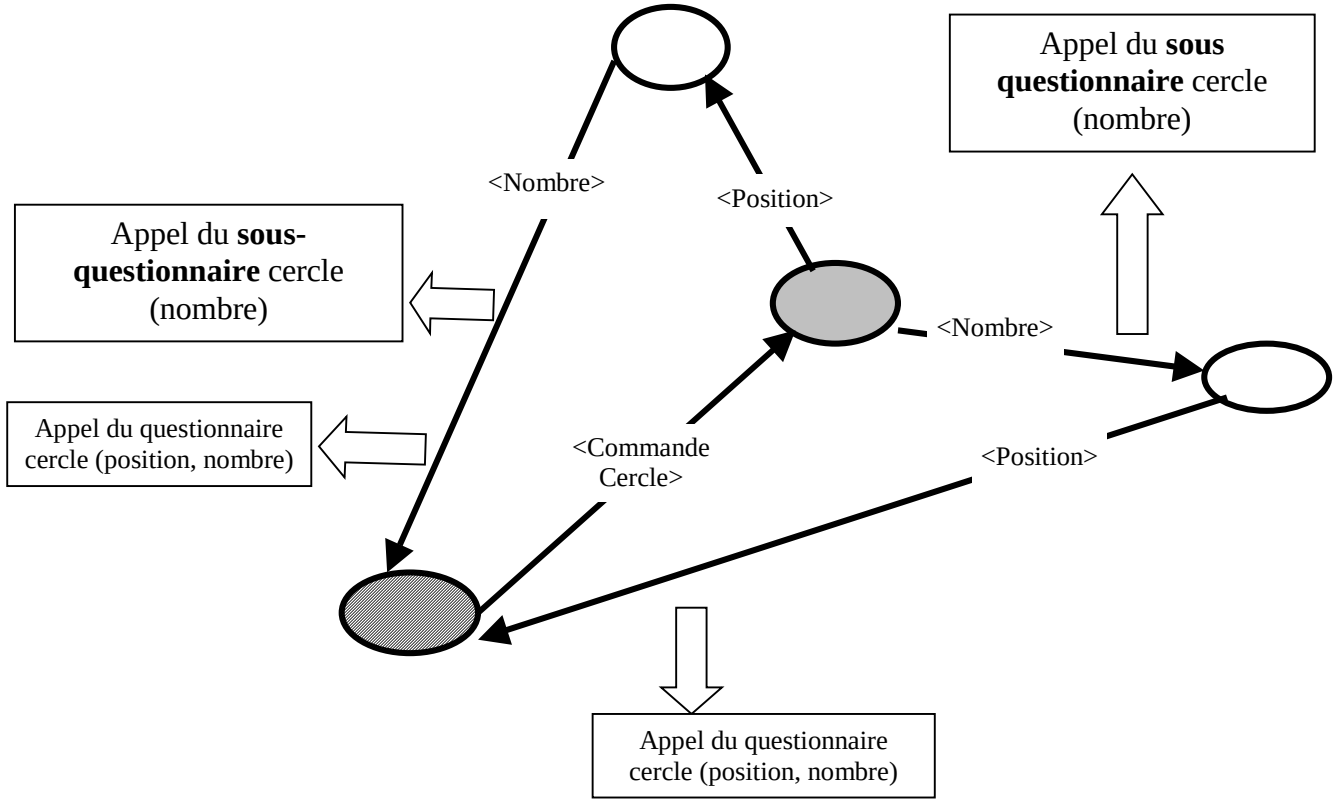


Figure II.11: Ajout de sous-questionnaire

L'exemple de la Figure II.11 montre l'ajout d'un sous-questionnaire au diaget de création de cercle. Le questionnaire père est défini par la commande « Cercle », et a pour paramètres un jeton « Position » et un jeton « Nombre ». Le sous-questionnaire qui a été ajouté permet de réaliser une action sur la saisie du rayon. Il est défini par la commande « Cercle » et le jeton « Nombre ». Il réalise une action lorsque l'utilisateur active la tâche de création d'un cercle et saisit le rayon. Par exemple, si le nombre fourni est négatif, le sous-questionnaire renvoie un compte-rendu faux. Le diaget ne change pas d'état et ne réalise aucune autre action tant que le

moniteur ne lui fournit pas un nombre positif. Ce nombre, s'il est positif, peut alors servir à réaliser un écho montrant la valeur mémorisée par le système.

5.2 Gestion des erreurs

Etant donné que *l'erreur est humaine*, l'utilisateur d'un système interactif fait des erreurs. Le concepteur doit prendre en compte la gestion de ces erreurs afin que l'application soit la plus robuste possible [Jambon 1997]. Cependant, il n'existe que peu d'outils permettant de réaliser la gestion des erreurs facilement. La boîte à outils du dialogue offre certains mécanismes automatiques pour gérer les erreurs de l'utilisateur.

La gestion des erreurs peut se faire de deux manières [Lewis et Norman 1986, Van der Schaaf 1997]: la prévention d'erreur et la correction d'erreur.

- La prévention d'erreur consiste à empêcher l'utilisateur de réaliser des actions qui mettraient le système dans un état d'erreur, comme par exemple empêcher l'utilisateur de saisir des positions lorsque le système n'attend que des nombres. C'est une gestion des erreurs de l'utilisateur en amont de l'interaction.
- La correction d'erreur consiste à fournir à l'utilisateur un ensemble de fonctionnalités lui permettant de changer l'état du système et de revenir dans un état stable après que l'utilisateur ait fait une erreur.

La boîte à outils du dialogue que nous proposons dispose de plusieurs mécanismes permettant de gérer les erreurs de l'utilisateur.

5.2.1 Prévention des erreurs

L'une des difficultés de création de l'interface d'une application graphique interactive est la gestion des entrées de l'utilisateur. Dans le cadre des applications pilotées par les données, où le système change d'état en fonction des types données fournies par l'utilisateur, il est nécessaire que ce dernier puisse fournir tous les types de données attendus par le système à un certain moment, et qu'il lui soit impossible de fournir des types de données non attendus. Pour cela, il faut que le système soit en mesure de contrôler la couche de présentation, afin d'activer les widgets permettant de fournir des données attendues, et de désactiver les widgets qui fournissent des données non désirées. Par exemple, si le système attend uniquement des positions (lorsque l'utilisateur crée une droite par deux points), l'utilisateur ne doit pas être en mesure de lui fournir de valeurs numériques. Cependant, toutes les commandes susceptibles

d’appeler une action qui produit des points doivent tout de même être accessibles à l’utilisateur. Dans l’exemple de la Figure II.10, la commande « centre » permettant d’appeler le questionnaire de calcul du centre d’un cercle, est accessible alors que l’interacteur de création attend des points.

La boîte à outils du dialogue que nous proposons offre un mécanisme permettant de réaliser le contrôle des entrées facilement. Le moniteur possède une méthode d’analyse qui permet à chaque instant de connaître, d’une part, l’ensemble des types de jetons paramètres attendus par le contrôleur de dialogue pour réaliser une action, et d’autre part, l’ensemble des commandes permettant de produire l’un des jetons attendus. Cela permet au moniteur d’autoriser les jetons (commandes ou paramètres) susceptibles d’être utilisés par le système.

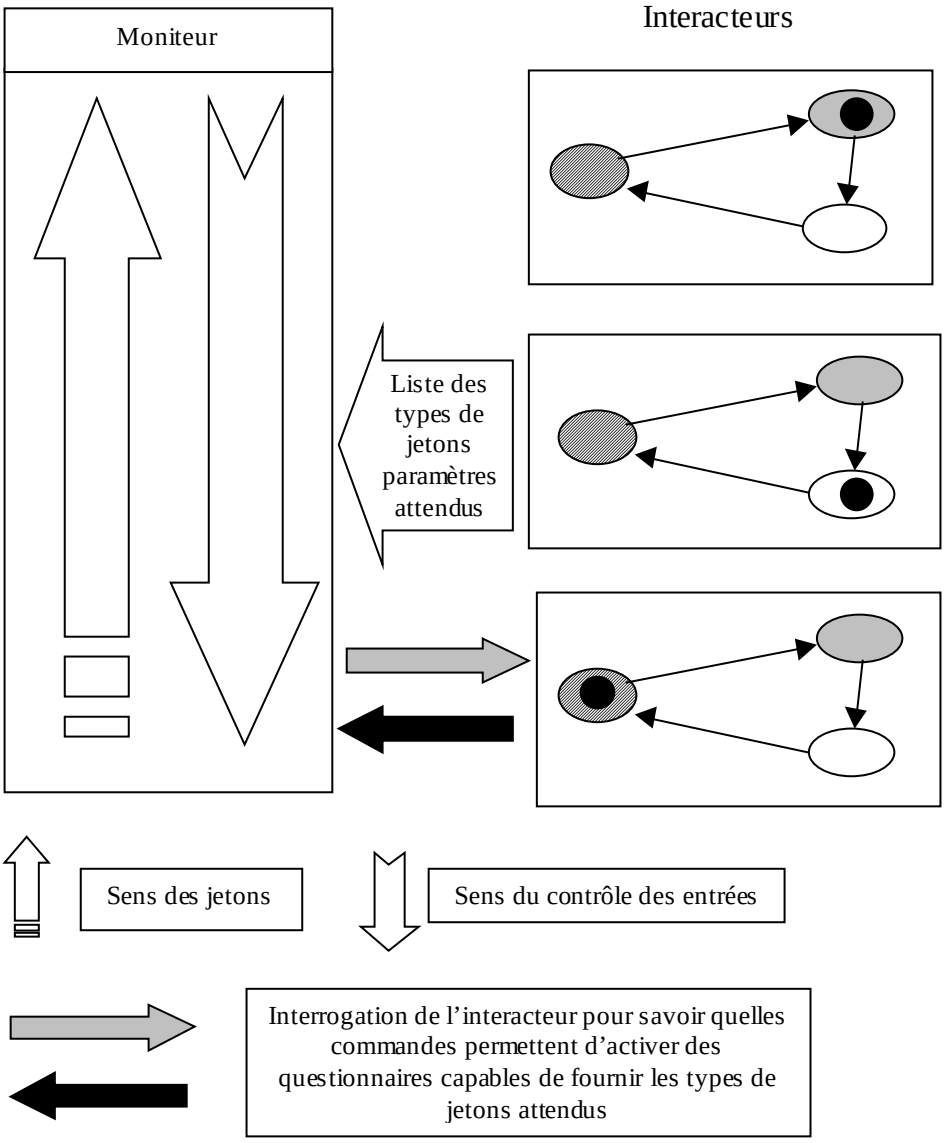


Figure II.12: Contrôle des entrées

Pour connaître l'ensemble des jetons admissibles à un moment donné, le moniteur scrute chaque diaget dans le sens inverse de la hiérarchie (du haut vers le bas) (Figure II.12). Il cherche le diaget le plus bas qui n'est pas dans l'état initial et donc qui attend des jetons paramètres. Il conserve les types des paramètres attendus par ce diaget, et pour chaque diaget en dessous dans la hiérarchie, il conserve les commandes susceptibles d'appeler un questionnaire capable de créer un des paramètres attendus. Ainsi, cette technique d'analyse permet de connaître à chaque instant l'ensemble des commandes et des paramètres attendus par le système. Cette liste est utilisée pour activer tous les widgets qui permettent de créer des jetons attendus par le système et d'interdire tous ceux qui permettent de créer des jetons non attendus.

5.2.2 Correction des erreurs

La gestion du dialogue d'une application graphique interactive doit également offrir des mécanismes permettant à l'utilisateur de corriger l'expression des tâches et des sous-tâches à réaliser avant que la tâche de plus haut niveau d'abstraction ne soit achevée (c'est-à-dire tant que la phase en cours n'a pas été complètement évaluée par le contrôleur de dialogue). En général, on considère qu'il existe deux grandes méthodes pour la correction d'erreurs [Dix, et al. 1993] :

- la correction en arrière, dans le cas où le système revient dans l'état où il se trouvait précédemment,
- la correction en avant, dans le cas où l'utilisateur réalise une tâche spécifique pour aller vers un état proche de celui désiré.

Les mécanismes de correction d'erreurs de la boîte à outils du dialogue que nous proposons concernent la première approche. La seconde approche est, en effet, fortement liée à la sémantique de l'application : il faut connaître la nature de l'erreur pour pouvoir déterminer quelle action sera à même de la corriger. Au contraire, la première est indépendante de cette sémantique, puisqu'elle ne consiste qu'à modifier l'état du contrôleur de dialogue pour permettre à l'utilisateur de ré-exprimer son but. Une méthode générale peut donc être élaborée.

Dans la boîte à outils du dialogue que nous proposons, il est possible de mettre en œuvre deux types de correction en arrière. La première correspond à la fonction « annuler » dans la

taxonomie des corrections d'erreur de [Yang 1992], la seconde correspond à la fonction « undo » [Dix, et al. 1997, Thimbleby 1990].

5.2.2.1 La fonction « Annuler »

L'annulation d'une phrase dans le cadre de l'architecture H^4 a déjà été abordée par L. Guittet [Guittet 1995]. Il propose que toute commande arrivant dans un interacteur soit interprétée comme l'arrêt de la commande en cours et le démarrage d'une nouvelle commande. Ceci permet à l'utilisateur qui souhaite interrompre un fil d'activité en cours de l'arrêter simplement en le démarrant à nouveau. Pour ce faire, le modèle H^4 préconise que lorsqu'un interacteur reçoit un jeton commande, il vide le registre de son automate. Si ce jeton lui permet de passer de l'état initial à un autre état, il effectue également cette transition, sinon il reste dans l'état initial.

Les diagets de la boîte à outils suivent exactement le même comportement. De plus, la boîte à outils du dialogue possède un jeton commande spécifique « Annuler » qui permet d'initialiser toutes les instances de diaget du contrôleur de dialogue. Cependant, cette approche ne suffit pas pour effacer toutes les conséquences déjà acquises de la phrase en cours.

Lorsque l'utilisateur annule une phrase en cours, il est probable que le contrôleur de dialogue ait appelé un ou plusieurs questionnaires et/ou réalisé un ou plusieurs échos. Le système doit alors annuler toutes les actions réalisées lors de ces appels et/ou effacer les échos. L'ordre d'annulation des questionnaires est important, il doit correspondre à l'ordre inverse de leur appel. En effet, l'annulation d'une action doit précéder l'annulation du calcul de ses paramètres : ceux-ci doivent, en effet, encore être disponibles pour permettre de réaliser l'action inverse.

Le modèle d'architecture H^4, et par conséquent la boîte à outils du dialogue que nous proposons, permettent à un diaget de réaliser plusieurs actions lors de l'expression d'un même but de plus haut niveau. Par exemple, sur la Figure II.13, le cercle est créé en utilisant comme paramètres le milieu de deux segments et l'extrémité d'un autre. Le diaget, gérant les questionnaires responsables du calcul du milieu d'un segment et de l'extrémité d'un segement, a appelé trois questionnaires. Il a effectué également trois actions. Si l'utilisateur décide d'annuler la création du cercle en cours, il est nécessaire d'annuler tous les calculs et/ou échos déjà réalisés et dont les résultats devaient être utilisés pour créer le cercle.

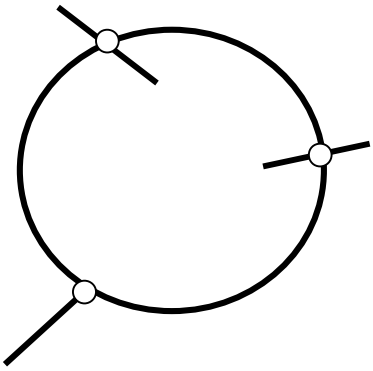


Figure II.13: Création d'un cercle par trois contraintes

Les diagets doivent conserver l'ensemble des questionnaires qu'ils ont appelés lors de la réalisation d'une tâche de haut niveau tant que la tâche en question n'a pas été réalisée. Si le diaget de plus haut niveau reçoit un ordre d'annulation, il envoie au moniteur l'ordre d'annuler l'appel des questionnaires des interacteurs de plus bas niveau. Les interacteurs annulent alors l'appel des questionnaires et des échos qu'ils ont mémorisés. Il est important de noter que lorsque le diaget de plus haut niveau passe de l'état initial à l'état de commande, le moniteur ordonne aux diagets de plus bas niveau d'abstraction d'annuler les questionnaires qu'ils ont éventuellement conservés lors de la réalisation de la tâche précédente.

Pour annuler une action qu'il a appelé, un diaget exécute un questionnaire particulier qui possède en entrée le nom du questionnaire dont on veut annuler l'exécution. Ce questionnaire est chargé d'appeler une méthode d'annulation spécifique du noyau fonctionnel. Cette méthode fait partie des trois services indispensables que doit fournir le noyau fonctionnel d'une application graphique interactive [Fekete 1996a].

Ce comportement que nous proposons induit deux remarques au niveau de la définition des questionnaires et du contrôle des entrées :

Remarque 1 : La commande d'annulation doit toujours être activable quels que soient les jetons attendus par le contrôleur de dialogue. Donc, lors de la phase de contrôle des entrées, la commande d'annulation est systématiquement ajoutée à la liste des commandes possibles (voir § 5.2.1).

Remarque 2 : Pour permettre à l'utilisateur de changer de tâche terminale de manière rapide, les commandes représentant ces tâches doivent être accessibles à chaque instant. Les tâches terminales sont représentées par des questionnaires qui ne produisent pas de jetons. Donc, lors de l'analyse des commandes et des paramètres attendus par le système, le contrôleur de dialogue doit ajouter à la liste des commandes possibles, l'ensemble des commandes activant

des questionnaires n'ayant aucun jeton de sortie. Ainsi, si l'utilisateur est en train de créer un cercle, il peut à chaque instant changer de tâche terminale et activer la création d'une droite. Les diagets se trouvant en dessous du diaget de création dans la hiérarchie sont alors re-initialisés (c'est-à-dire qu'ils reviennent dans l'état initial et vident leurs registres). Le changement de tâche, exprimé lors d'une tâche en cours, provoque une annulation suivie de la nouvelle commande.

5.2.2.2 La fonction « Undo »

La fonction « Undo » (littéralement défaire) est la fonction de correction d'erreur la plus connue et la plus répandue. Elle permet de faire revenir le système dans l'état précédent la dernière interaction de l'utilisateur. Elle doit gérer le retour en arrière du noyau fonctionnel et de l'interaction. Mais, la granularité du « undo » est plus faible que l'annulation étudiée précédemment, car l'utilisateur n'annule qu'une partie de l'expression en cours plutôt que toute l'expression. Lorsqu'on la répète, la fonction « undo » permet également de revenir plusieurs pas en arrière pour retrouver le système tel qu'il était avant que l'utilisateur ne réalise plusieurs interactions. Le nombre maximum de « undo » possible est la portée du « undo » (« range » en anglais). Plus la portée est grande plus, le « undo » pourra revenir en arrière et plus le système a d'informations à mémoriser.

Nous proposons une approche pour offrir un mécanisme de « undo » dans la boîte à outils du dialogue. Ce mécanisme permet de faire revenir le dernier diaget, qui a changé d'état, dans l'état précédant. La portée du undo de la boîte à outils du dialogue n'est pas fixe. En fait le « undo » mémorise tous les états successifs du dialogue jusqu'au démarrage d'une nouvelle tâche de haut niveau.

D'un point de vu technique, le undo peut être réalisé par le mécanisme suivant (que nous avons implanté dans notre prototype). Le moniteur conserve tous les jetons qui lui arrivent dans une liste. Lorsqu'il reçoit la commande undo, il envoie au diaget la commande d'annulation (c.f. § 5.2.2.2). Les diagets reviennent alors tous dans leur état initial, et toutes les actions, réalisées au cours de l'expression du but en cours, sont annulées. Ensuite, le moniteur retransmet un à un tous les jetons mémorisés sauf le dernier. Ainsi, l'application retrouve l'état qui était le sien juste avant d'avoir reçu le dernier jeton.

La liste des jetons est initialisée dès que le diaget de plus haut niveau passe à l'état de commande, ce qui correspond au début de l'expression d'un nouveau but de haut niveau pour l'utilisateur.

6 Exemple d'utilisation

Dans l'exemple qui suit, nous montrons comment réaliser le contrôleur de dialogue d'une application de dessin. Nous expliquons plus en détails les liens qui existent entre l'adaptateur de présentation et les éléments de la boîte à outils du dialogue.

La première partie de cette section est consacrée à la description de l'application à réaliser en termes de fonctionnalités. Dans la deuxième partie, nous montrons l'instanciation des éléments qui composent le contrôleur de dialogue. Enfin, la dernière partie concerne les liens nécessaires entre ces éléments et les widgets de la présentation.

6.1 Description de « Dessine »

L'application « Dessine » permet de dessiner des cercles et des segments. Les cercles peuvent être créés par centre et point de passage, par centre et rayon ou passant par trois points. La seule manière de créer un segment est de fournir ses deux extrémités. Le système « Dessine » permet de réaliser des calculs arithmétiques entre deux nombres. Il possède une fonction grapho-numérique [Gardan 1991] calculant la distance entre deux points, et les fonctions d'interrogation : milieu d'un segment, rayon d'un cercle ou centre de cercle.

La couche de présentation contient notamment une zone de saisie de nombres, des menus permettant de fournir des commandes, et une zone graphique utilisée pour l'affichage des objets géométriques et pour leur sélection ainsi que pour fournir des pointés graphiques. Nous faisons l'hypothèse que la présentation utilise un modèle vectoriel [Fekete 1996b] pour l'affichage et qu'elle est donc capable de gérer la sélection d'éléments graphiques.

6.2 Création du contrôleur de dialogue

Pour créer le contrôleur de dialogue d'une application à l'aide de la boîte à outils du dialogue, le concepteur commence par définir les types de jetons représentant les données et commandes de son application. Puis, il crée les questionnaires en utilisant ces types de jetons. La phase suivante consiste à regrouper les questionnaires en niveaux d'abstraction en instanciant les diagets. Puis, il crée le moniteur en lui fournissant tous les diagets qu'il doit contrôler dans l'ordre de la hiérarchie. Dans la suite de cette section, nous détaillons chacune de ces quatre étapes.

6.2.1 Création des jetons

La phase de création des types de jetons consiste en fait à créer de nouvelles classes de jetons paramètres à l'aide de la classe générique *Jeton_Paramètre*. Les commandes sont directement instanciées avec la classe *Jeton_Commande*. Pour créer de nouvelles classes de jetons paramètres, le concepteur fournit deux chaînes de caractères, la première représente le type du jeton, la seconde constitue le père dont le jeton hérite. Pour des raisons de simplification d'écriture, nous faisons abstraction, dans cette description de la définition, de la valeur contenue dans le jeton.

L'application « Dessine » manipule quatre types de données : les nombres, les positions, les cercles et les segments. Le concepteur devra donc créer quatre types de jetons paramètres :

```
Class J_Nombre is new Jeton_Paramètre( « Nombre », « Paramètre ») ;

Class J_Position is new Jeton_Paramètre(« Position »,  « Paramètre ») ;
```

Les classes *J_Nombre* et *J_Position* héritent directement de la classe *Jeton_Paramètre*, leur nature est, respectivement, Nombre et Position.

Les classes de jetons *J_Cercle* et *J_Segment* héritent du même ancêtre : le jeton *J_Objet* qui représente au niveau de la définition du dialogue l'ensemble des objets géométriques.

```
Class J_Objet is new Jeton_Paramètre( « Objet »,  « Paramètre ») ;

Class J_Cercle is new Jeton_Paramètre( « Cercle »,  « Objet ») ;

Class J_Segment is new Jeton_Paramètre(« Segment », « Objet ») ;
```

La Figure II.14 représente l'arbre des jetons paramètres de l'application « Dessine », cet arbre sera utilisé lors de la création des questionnaires afin de factoriser leur définition.

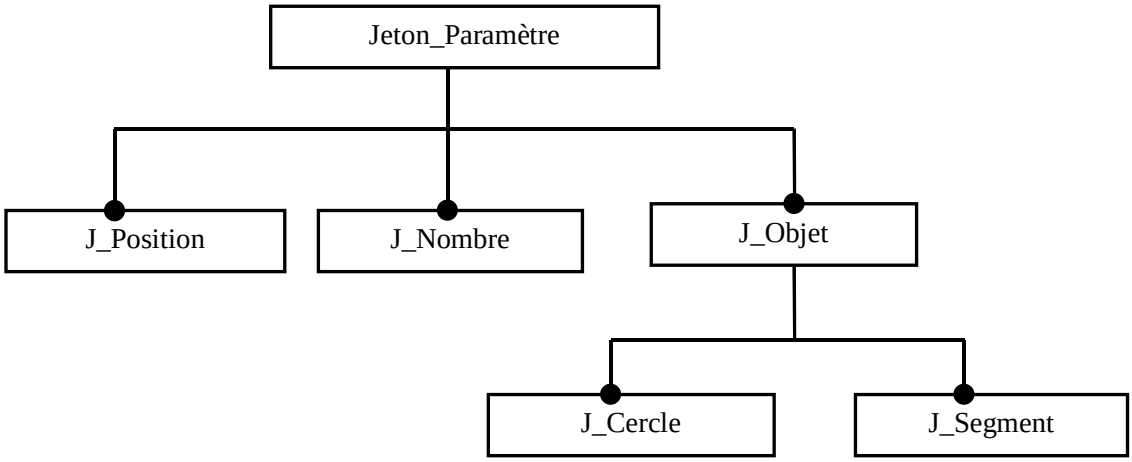


Figure II.14: Hiérarchie des paramètres de "Dessine"

6.2.2 Définition des questionnaires

La seconde phase de la création du contrôleur de dialogue est relative à la définition des différents questionnaires correspondant aux différentes tâches et sous-tâches du système. Les questionnaires sont des instances des classes *Questionnaire* ou *QuestionnaireItératif*. Il sont définis par :

- une chaîne de caractères représentant la commande d'activation du questionnaire,
- une liste de chaînes de caractères représentant le type des jetons d'entrée,
- une chaîne représentant le jeton de sortie (si cette chaîne n'est pas spécifiée, le questionnaire n'a pas de jeton de sortie).

Questionnaire Q_Cercle1 (« Créer Cercle », { « Position », « Nombre »})

Questionnaire Q_Cercle2 (« Créer Cercle », { « Position », « Position »})

Questionnaire Q_Cercle3 (« Créer Cercle 3 pts », { « Position », « Position », « Position »})

On remarque que la commande « Créer Cercle » permet d'activer les questionnaires *Q_Cercle1* et *Q_Cercle2*. Le questionnaire *Q_Cercle3* ne peut avoir la même commande d'activation car sa liste de paramètres d'entrée contient la liste de paramètres d'entrée du questionnaire *Q_Cercle2*. Ces trois questionnaires décrivent les trois méthodes de création de cercles nécessaires à l'application « Dessine ».

Questionnaire Q_Segment (« Créer Segment », {« Position », «Position »})

Comme pour les questionnaires de création de cercle, le questionnaire *Q_Segment* ne rend aucun jeton au système. Il ne représente donc pas une sous-tâche de production.

Questionnaire Q_Détruire (« Detruire objet », « Objet »)

Le questionnaire *Q_Détruire* permet de détruire un objet. Au moment de l'exécution, ce questionnaire est appelé soit avec un cercle, soit avec un segment. Cet exemple montre l'utilité de la hiérarchie des jetons. Sans cette hiérarchie, le concepteur aurait dû créer deux questionnaires (un pour les cercles et un pour les segments).

Questionnaire Q_Distance (« Distance »,{« Position », « Position »}, « Nombre »)

Le questionnaire Q_Distance permet de calculer la distance entre deux positions. Le résultat de ce calcul est contenu dans le jeton de sortie qui est de type « Nombre ».

Les questionnaires permettant les calculs arithmétiques prennent deux nombres en entrée et le résultat du calcul est encapsulé dans le jeton de sortie qui est de type Nombre.

Questionnaire Q_Addition (« + »,{« Nombre », « Nombre »})

Q_Addition . Sortie= « Nombre »

Questionnaire Q_Soustraction (« - »,{« Nombre », « Nombre »})

Q_Soustraction . Sortie= « Nombre »

*Questionnaire Q_Multiplication (« * »,{« Nombre », « Nombre »})*

Q_Multiplication . Sortie= « Nombre »

Questionnaire Q_Division (« / »,{« Nombre », « Nombre »})

Q_Division . Sortie= « Nombre »

Les trois derniers questionnaires composant le contrôleur de dialogue de « Dessine » sont les questionnaires d'interrogation qui permettent de calculer une valeur spécifique d'un objet graphique, en l'occurrence le centre d'un cercle, le rayon d'un cercle ou le milieu d'un segment.

Questionnaire Q_Centre (« Centre », {« Cercle »})

Q_Centre . Sortie= « Position »

Questionnaire Q_Rayon (« Rayon », {« Cercle »} , « Nombre »)

Q_Rayon . Sortie= « Nombre »

Questionnaire Q_Milieu (« Milieu », {« Segment »}, « Position »)

Q_Milieu . Sortie= « Position »

Pour avoir un meilleur contrôle de l'application, le concepteur souhaite ajouter un sous-questionnaire afin de vérifier que le rayon du cercle fourni par l'utilisateur est supérieur à zéro.

Fonction SQ_Cercle_Rayon (« Cercle », {« Nombre »})

6.2.3 Création des diagets

Les diagets sont chargés de regrouper les questionnaires représentant des tâches d'un même niveau d'abstraction. La définition d'un diaget nécessite l'attribution d'un nom sous la forme d'une chaîne de caractères. Le concepteur doit utiliser la méthode *ajoute_Q* pour désigner les questionnaires qu'un diaget contrôle. Les tâches de l'application « Dessine » se décomposent en trois niveaux d'abstraction :

- le niveau le plus haut est celui de la création et de la destruction des objets.
- le niveau intermédiaire est le niveau des expressions.
- le dernier (le plus bas dans la hiérarchie) est celui des interrogations.

Les questionnaires regroupés dans les deux derniers niveaux représentent des sous-tâches de production. Leur résultat sera utilisé par les questionnaires du niveau de la création.

Diaget D_Création (« Création »)

D_Création.ajoute_Q (Q_Cercle1)

D_Création.ajoute_Q (Q_Cercle2)

D_Création.ajoute_Q (Q_Cercle3)

D_Création.ajoute_Q (Q_Segment)

D_Création.ajoute_Q (Q_Détruit)

A chaque fois qu'un nouveau questionnaire est ajouté, le diaget contrôle si les règles de surcharge sont respectées. Si c'est le cas, le diaget génère la partie de l'automate contrôlant l'appel du questionnaire, sinon il ne tient pas compte du questionnaire.

Diaget_Recursif D_Calcul (« Calcul »)

D_Calcul.ajoute_Q(Q_Addition)

D_Calcul.ajoute_Q(Q_Soustraction)

D_Calcul.ajoute_Q(Q_Multiplication)

D_Calcul.ajoute_Q(Q_Division)

D_Calcul.ajoute_Q(Q_Distance)

Le diaget *D_Calcul* est un diaget récursif. Il peut donc appeler plusieurs instances des questionnaires qu'il contrôle dans la même expression, contrairement aux diagets simples qui ne peuvent appeler qu'un questionnaire à la fois. Le questionnaire *Q_Distance* est contrôlé par le diaget *D_Calcul* car le résultat de ce questionnaire peut être utilisé dans une expression numérique, et ses paramètres pourraient être le résultat d'une autre expression récursive comme le projeté d'un point sur un objet qui a pour résultat un point (si l'on voulait introduire ce type d'opérateur géométrique dans notre étude de cas).

Le dernier diaget de notre application gère les questionnaires qui interrogent les objets géométriques afin d'obtenir la valeur de certains de leurs attributs. Ce diaget est le plus bas de notre hiérarchie car les résultats obtenus peuvent être utilisés pour réaliser un calcul (distance entre deux centres de cercles) ou pour créer un nouvel objet (créer un cercle dont le centre est le milieu d'un segment).

```
Diaget D_Interrogation (« Interrogation »)
D_Interrogation.ajoute_Q(Q_Centre)
D_Interrogation.ajoute_Q(Q_Rayon)
D_Interrogation.ajoute_Q(Q_Milieu)
```

Le concepteur ajoute le sous-questionnaire permettant la vérification de la valeur du rayon à l'interacteur de création :

```
D_Creation. ajoute_Contrôle (SQ_Cercle_Rayon)
```

Le questionnaire *SQ_Cercle_Rayon* sera appelé lorsque le diaget aura reçu la commande « Cercle » et un jeton nombre.

6.2.4 Le moniteur

La hiérarchisation des diagets se fait au moment de la création du moniteur. Le constructeur du moniteur possède en entrée la liste des diagets de l'application classés dans l'ordre décroissant de la hiérarchie. Pour notre cas d'étude, le diaget *D_Creation* est le plus haut alors que le diaget *D_Interrogation* est le plus bas.

```
Moniteur M_Moniteur ({D_Creation, D_Expression, D_Interrogation})
```

6.3 Liens entre le contrôleur de dialogue et l'adaptateur de présentation

L'adaptateur de présentation, tel qu'il est décrit dans H[4] a pour rôle de gérer les transformations de l'espace. D'une part, il transforme les données venant de l'utilisateur par l'intermédiaire des fonctions de rappel en jetons qui seront ensuite exploités par le moniteur. Par exemple, il transforme les coordonnées de l'espace (en pixels) fournies par la fonction de rappel « clic dans la zone de dessin » en jeton position contenant les coordonnées correspondantes dans l'espace utilisateur (en centimètre par exemple). D'autre part, il fournit des primitives permettant aux objets du noyau fonctionnel de s'afficher. Dans ce cas, il transforme des coordonnées de l'espace utilisateur en coordonnées de l'espace écran.

En fait, le rôle de l'adaptateur de présentation est plus important que le seul rôle des transformations de l'espace. Ce composant réalise l'indépendance entre la couche de présentation et le contrôleur de dialogue. Il est donc chargé de transformé les données venant de la couche de présentation en jetons utilisables par le contrôleur de dialogue. De plus, il récupère les types de jetons attendus à chaque instant par le contrôleur de dialogue (ces types

lui sont fournis par l'intermédiaire de la fonction *analyse* du moniteur), et il réalise les autorisations et les interdictions des widgets de la présentation afin que seuls des types de données attendus par le contrôleur de dialogue puissent être fournis par l'utilisateur.

6.4 Synthèse

La création d'une application interactive, en utilisant la boîte à outils du dialogue comporte cinq étapes correspondant aux cinq composants du modèle Arch.

- La première étape consiste à créer le noyau fonctionnel de l'application. C'est dans ce composant qu'est définie toute la sémantique de l'application (i.e. ce que fait l'application).
- La seconde étape est la réalisation de la couche de présentation à l'aide de widgets. Ceci peut être réalisé en utilisant une boîte à outils de présentation quelconque associée à un outil interactif de construction d'interface pour définir finement l'aspect du système pour l'utilisateur.
- La troisième étape consiste à créer le contrôleur de dialogue de l'application à l'aide des éléments décrits dans la boîte à outils du dialogue que nous proposons. Ainsi que nous l'avons vu sur notre exemple, cette étape est extrêmement simple.
- La quatrième étape consiste à relier la couche de présentation et le contrôleur de dialogue. Cette liaison correspond à l'adaptateur de présentation. Le concepteur y décrit notamment les liens entre les widgets et les jetons.
- Le lien entre le contrôleur de dialogue et le noyau fonctionnel, qui constitue la cinquième étape, est réalisé dans le corps des questionnaires. Les questionnaires sont chargés de l'appel des actions du noyau fonctionnel. Pour cela, ils transforment les jetons qu'ils ont en entrée et utilisent les données ainsi créées pour réaliser cet appel. Le résultat de l'appel des actions du noyau fonctionnel est transformé en jeton utilisable par le contrôleur de dialogue.

Une remarque importante concernant la mise en œuvre de la boîte à outils du dialogue est que la définition des noms des questionnaires se fait en utilisant des chaînes de caractères plutôt qu'en utilisant des noms de classes ou des types de données spécifiques. Or, ce type de définition peut conduire à des erreurs (le concepteur fait une faute en écrivant le type d'un jeton paramètre) qui ne seront pas vues par le compilateur. Cependant, ce choix est nécessaire du fait que nous voulons utiliser cette boîte à outils pour modifier dynamiquement le dialogue

de l'application. Nous avons donc besoin d'utiliser des structures dynamiques pour décrire les différents éléments du contrôleur de dialogue. De la même manière, on s'aperçoit que les diagets ne sont pas créés directement avec la liste des questionnaires qu'ils contrôlent. Les questionnaires sont ajoutés un à un aux diagets par l'intermédiaire de la méthode *ajoute_Q*. Cela permet d'ajouter dynamiquement de nouveaux questionnaires au diagets de l'application, et donc de gérer de nouvelles tâches crées dynamiquement.

7 Boîte à outils du dialogue et outils de conceptions d'interface

Nous avons vu au chapitre précédent les deux approches et les deux types d'outils existants pour concevoir une interface. Le concept de boîte à outils de dialogue que nous proposons bénéficie des avantages respectifs des deux approches et contribue à en limiter les inconvénients.

D'abord, de nombreuses similitudes entre l'utilisation des widgets et l'utilisation des diagets, même si leurs rôles sont différents (les widgets servent à décrire la présentation alors que les diagets servent à décrire le contrôle), peuvent être notées.

Pour créer une couche de présentation, un concepteur instancie deux types de widgets : les conteneurs et les terminaux [Fekete 1996b]. Les widgets conteneurs n'ont qu'une tâche articulatoire. Ils servent à organiser les widgets terminaux afin d'améliorer l'aspect de la présentation. Les widgets terminaux sont utilisés soit pour appeler une fonctionnalité de l'application, soit pour réaliser un affichage, ou soit les deux à la fois. De la même manière, la boîte à outils de dialogue possède deux types d'objets de contrôle : le moniteur et les diagets. Le moniteur sert à organiser les diagets selon la sémantique du dialogue. Les diagets sont chargés de contrôler l'appel des fonctions de l'application.

En ce qui concerne l'appel des actions de l'application, il existe là aussi une similitude entre widgets et diagets : Les widgets appellent une action (la fonction de rappel) lorsqu'ils reçoivent certains événements. La fonction de rappel appelée dépend de l'événement et du widget. Les diagets appellent des actions (les questionnaires) lorsqu'ils ont reçu une certaine suite de jetons. La fonction appelée dépend du diaget et de la suite de jetons reçus. On peut donc faire une similitude entre le comportement des diagets et celui des widgets. En fait, dans la boîte à outils du dialogue, les diagets font office de widgets, les listes de jetons sont les évènements, et enfin, les questionnaires représentent les fonctions de rappel. Cette similitude montre que l'approche que nous proposons est le prolongement naturel, pour l'aspect conception du dialogue, de la démarche basée sur les boîtes à outils qui est presque

universellement utilisée pour concevoir des interfaces graphiques. La deuxième boîte à outils que nous proposons possède, cependant, un avantage essentiel par rapport aux boîtes à outils de présentation: basée sur des automates, elle permet très facilement de représenter l'aspect contextuel des conséquences des entrées de l'utilisateur.

Concernant les outils suivant l'approche descendante, en particuliers les systèmes basés sur modèle [Szekely, et al. 1993] (traduction de Model Based System ou MBS), ils permettent de décrire l'interface à l'aide de langages spécifiques. Pour cela, ils utilisent souvent plusieurs modèles tels qu'un modèle des tâches, un modèle de la présentation, un modèle de l'application, un modèle de contrôle, etc ...

Ces modèles permettent, dans certains cas de générer une partie du code de l'application, ou peuvent être exécutés par un interpréteur spécifique.

Les MBS et la boîte à outils du dialogue ont un point commun : ils permettent aux concepteurs de créer ou de décrire explicitement le contrôleur de dialogue de l'application en utilisant différentes représentations des éléments qui le compose. Ainsi, la boîte à outils du dialogue que nous proposons permet d'instancier différents modèles de l'application :

- les jetons paramètres représentent le modèle des objets manipulés par l'application,
- les questionnaires servent de modèles des tâches du système,
- les diagets représentent le modèle du contrôle de l'application.

Elle peut être considérée comme une partie de MBS. De plus, comme les MBS, la boîte à outils du dialogue utilise une description déclarative du composant qu'elle permet de créer (i.e. le contrôleur de dialogue).

Cependant, il existe deux différences majeures entre la boîte à outils du dialogue et les MBS. D'abord, la boîte à outils du dialogue est un outil « local » qui s'intéresse uniquement à la construction du contrôleur de dialogue, et permet de concevoir un tel contrôleur pour une très grande variétés d'applications, à la différence des MBS dont nous avons vu le caractère global, et donc très spécialisé. Ensuite, la boîte à outils du dialogue est composée d'un ensemble de classes qui sont instanciables en utilisant un langage de programmation et un environnement standard (à ce jour nous l'avons intégré dans deux boîtes à outils : Tcl/Tk et MFC). Cela permet de créer donc et de modifier dynamiquement le contrôleur de dialogue, par exemple en instanciant de nouveaux questionnaires et de nouveaux diagets, et de les intégrer dynamiquement dans le contrôleur de dialogue de l'application. Cette propriété est

primordiale dans le cadre de notre travail. En effet, nous voulons intégrer dynamiquement de nouvelles fonctionnalités à une application de CAO. L'ajout de ces fonctions entraîne une modification de l'interface afin que l'utilisateur puisse y accéder.

8 Conclusion

Les travaux sur les modèles d'architecture d'application graphique interactive montrent que le concepteur doit séparer l'application en trois modules principaux : la présentation, le contrôleur de dialogue et le noyau fonctionnel. Parallèlement à ces travaux, un grand nombre d'outils d'aide à la conception ont été créés afin d'aider le concepteur dans sa tâche. Ils appartiennent à deux grandes familles : les boîtes à outils (approche ascendante) et les générateurs d'interfaces (approche descendante).

Les boîtes à outils sont, de beaucoup, les environnements les plus utilisés. Elles permettent de réutiliser assez aisément des outils d'interaction conviviaux pour créer tous types d'applications. Le mécanisme de fonction de rappel s'avère suffisant pour contrôler l'application lorsque les tâches supportées par le système sont autonomes et peu structurées. Inversement, lorsqu'une tâche se décompose de façon récursive en sous-tâches, rendant le langage de dialogue fortement contextuel et structuré, le mécanisme de fonction de rappel s'avère tout à fait insuffisant. Une approche alors suivie consiste à disperser dans le code tout le contrôleur de dialogue, rendant l'application coûteuse à développer et difficile à maintenir.

Les générateurs d'interfaces, quant à eux, permettent de modéliser explicitement le contrôleur de dialogue de l'application en utilisant un ou plusieurs langages spécifiques. A partir de cette description, ils sont capables de générer entièrement l'interface de l'application (présentation + contrôleur de dialogue). Certains d'entre eux permettent de décrire un dialogue structuré nécessaire, en particulier, à la réalisation d'applications de conception technique. Cependant, ces outils sont très peu génériques, un générateur d'interface utilisable pour créer une application de traitement de texte ne sera certainement pas capable de générer l'interface d'un système CAO.

Dans ce chapitre, nous avons proposé une nouvelle approche pour la spécification et la conception du contrôleur de dialogue. Notre approche introduit, sous la forme d'une boîte à outils, une nouvelle catégorie d'unités qui vise à représenter les différents niveaux de tâches que l'utilisateur peut activer par l'intermédiaire du système.

Cette approche permet d'allier la généricité des boîtes à outils à la puissance de description des générateurs d'interfaces. Elle consiste :

- à créer des éléments du dialogue (les Diagets = dialogue + gadget) à partir de la description des tâches de l'application,
- à représenter les structures tâches/sous-tâches par une simple relation d'ordre entre diagets.

Ainsi, le concepteur conserve le mode de conception où il lie les actions du noyau fonctionnel à des éléments d'une bibliothèque de composants (« widget » + fonctions de rappel).

Notre approche s'adapte parfaitement à toutes les applications dont le dialogue est structuré, c'est-à-dire où le résultat d'une tâche peut-être utilisé comme paramètre d'une autre. Ce type de dialogue est fréquemment utilisé par les applications de conception technique, mais aussi par toutes les applications utilisant des expressions numériques où l'une des opérandes peut provenir d'un calcul antérieur.

La boîte à outils du dialogue offre plusieurs avantages. Elle :

- permet de spécifier explicitement le contrôleur de dialogue d'une application graphique interactive, ce qui est absolument nécessaire pour concevoir des applications supportant un dialogue structuré,
- conserve le mode de conception bien connu où les briques de base d'un composant sont associées à des actions du noyau fonctionnel. On peut faire une similitude entre widget + fonctions de rappel et diaget + questionnaires,
- permet la modification dynamique du contrôleur de dialogue grâce à l'instanciation de nouvelles briques de base qui le composent,
- offre plusieurs outils de contrôle automatique de l'application, tels que le contrôle des entrée ou les fonctions « Undo » et « Annulation », qui simplifient l'écriture du code et diminuent donc le temps de conception.

La boîte à outils du dialogue que nous avons proposée dans ce chapitre a fait l'objet de deux implantations différentes. Une première version a été réalisée en Tcl, elle est associée à la boite à outils Tk. La seconde version a vu le jour dans le cadre du développement de TexAO

(cf. Chapitre IV). Elle est implantée en C++ et elle est associée aux MFC (Microsoft Foundation Classes) pour la partie présentation.

Chapitre III
Définition de classes par l'exemple

1 Introduction

La conception du noyau fonctionnel représente une part importante de la création de toute application graphique interactive, et en particulier des applications de conception technique. Le noyau fonctionnel définit la sémantique de l'application et contient l'ensemble des classes qu'elle manipule. La spécialisation d'une application, c'est-à-dire son adaptation à un domaine ou à un problème particulier, nécessite la définition de nouvelles classes d'objets spécifiques du domaine visé. Or, actuellement, cette spécialisation ne peut se faire sans décrire de manière explicite le code des nouvelles classes, ce qui ne peut être réalisé que par un spécialiste informaticien, et non pas par un expert du domaine d'application. Ainsi, un des buts de notre travail serait de permettre à des experts du domaine d'application visé, d'adapter interactivement une application de manière à ce qu'elle réponde à leurs besoins spécifiques. Il faut donc leur fournir des outils capables de créer, sans programmation explicite, de nouvelles classes d'objets.

Une classe est la description d'une famille d'objets ayant la même structure et le même comportement. Elle regroupe un ensemble d'attributs (propriétés) et un ensemble de procédures ou de fonctions. Chaque classe possède donc une double composante [Masini, et al. 1989] :

- Une composante statique, les attributs (données), constitués de champs nommés, qui possèdent chacun une valeur. Les champs caractérisent l'état des objets pendant leur évolution.
- Une composante dynamique, les procédures, appelées méthodes, qui représentent le comportement commun des objets appartenant à la classe. Les méthodes manipulent les champs des objets et représentent les actions pouvant être effectuées par ou sur les objets. Elles permettent des transitions entre les états décrits par les attributs.

Ainsi, pour définir une classe de manière interactive, il est nécessaire de définir ses champs, appelés attributs dans la suite, et ses méthodes de manière interactive. Les méthodes d'une classe peuvent être divisées en trois catégories :

- les constructeurs qui utilisent des paramètres en entrée et permettent de créer un objet de la classe en donnant des valeurs à ses attributs,
- les méthodes d'interrogation, ou sélecteurs, utilisées pour obtenir les valeurs des attributs d'un objet,
- les méthodes de modification qui possèdent des paramètres d'entrée utilisées pour modifier la valeur de certains attributs de l'objet.

Les deux grandes techniques, voisines, permettant à un « non-informaticien » de créer des programmes de manière interactive, ont été passées en revue au chapitre I (section 3). Il s'agit de la programmation par démonstration et de la géométrie paramétrée. Bien que ces deux techniques de programmation interactive soient capables d'enregistrer des programmes, elles sont insuffisantes pour créer des classes d'objets. En effet, il est impossible, à l'aide de ces deux techniques, de définir et de regrouper différentes méthodes au sein d'une même classe. De plus, elles ne permettent ni de définir des attributs, ni de définir des relations d'héritage entre classes.

La solution que nous suggérons, pour permettre à un utilisateur de définir interactivement de nouvelles classes d'objets, se base sur un modèle paramétrique que nous augmenterons par l'ajout des principes d'abstraction provenant du domaine de la programmation par démonstration. L'approche que nous proposons introduit différentes structures de données capables non seulement de créer et de conserver le processus de construction d'objets paramétrés mais aussi de définir, pour ces objets, des attributs et des méthodes spécifiques. Elle est décrite dans ce chapitre.

Dans la section suivante, nous détaillons les apports et les manques de la programmation par démonstration et de la géométrie paramétrée dans le cadre de la définition interactive de classes d'objets. La troisième partie de ce chapitre est consacrée à la description de notre modèle pour la définition de classes. Nous détaillerons en particulier les moyens mis en œuvre pour décrire interactivement les constructeurs et les sélecteurs. La quatrième partie explique comment le modèle que nous proposons est exploité, c'est-à-dire comment une classe est créée et comment elle est exploitée. Enfin, la dernière section indique quels sont les domaines d'application visés par la définition interactive de classes d'objets.

2 Les méthodes de programmation interactive

Pour définir une nouvelle classe d'objets à partir de la description de l'une de ces instances, le système doit en premier lieu enregistrer un programme qui correspond au constructeur de la classe. Ce programme est appelé à chaque fois que la classe définie par l'utilisateur est instanciée. Comme nous l'avons vu au Chapitre I (section 3), deux approches ont été développées pour enregistrer des programmes sans programmation explicite : la première est la programmation par démonstration, la seconde est la géométrie paramétrée (ou paramétrique).

2.1 La programmation par démonstration

La technique de programmation par démonstration permet à des utilisateurs non-informaticiens de créer des programmes sans programmation explicite. Ces systèmes sont capables de générer des « macros » qui enregistrent, après mise en œuvre d'un mécanisme d'abstraction, puis ré-exécutent automatiquement des séquences d'actions de l'utilisateur. Certains systèmes sont même capables de générer des programmes dans un langage neutre qui peuvent être ensuite utilisés sur d'autres systèmes [Potier 1995].

Programmer par démonstration signifie créer un programme à partir d'un exemple de son exécution. Une notion importante, amenée par la programmation par démonstration, est la méthode d'abstraction. En programmation classique, les programmes manipulent les variables par leur nom. Le lien entre un nom et une valeur est réalisé à l'aide d'une table de correspondances appelée contexte du programme. Ainsi, toutes les variables du programme sont référencées dans cette table, sans ambiguïté, quelle que soit leur valeur. A l'inverse, la programmation par démonstration autorise le « programmeur » à manipuler directement les variables par leurs valeurs. Créer un programme qui manipule des variables, à partir d'un exemple de son exécution nécessite donc d'associer un nom de variable à chaque valeur. Cette tâche est réalisée par le contexte dynamique [Pierra, et al. 1996b]. Pour chaque valeur créée dans l'exemple, un nouveau nom de variable (dont le type est défini par la valeur) est ajouté dans le contexte dynamique. Ces noms de variable sont utilisés pour abstraire les valeurs dans le programme généré.

La possibilité de spécifier les paramètres du programme doit aussi être fournie par les systèmes de programmation par démonstration. Il existe deux méthodes pour la définition des paramètres d'un programme créé par démonstration. La méthode implicite, préconisée par [Bauer 1979], consiste à créer plusieurs exemples d'un même programme. Le système infère, alors,

automatiquement les paramètres après une analyse des différents exemples. Dans la méthode explicite [Girard 1992, Halbert 1984], l'utilisateur indique au système les valeurs qui représentent les paramètres du programme. Les autres valeurs sont considérées comme des variables internes ou comme des constantes.

2.2 La géométrie paramétrée

[Cugini, et al. 1988] donne deux définitions de la géométrie paramétrée :

1. *Un dessin paramétrique représente une famille d'objets qui partagent les mêmes contraintes topologiques, mais qui diffèrent les uns des autres par leurs caractéristiques géométriques.*
2. *Un dessin technique paramétrique, en particulier, est un dessin où les dimensions sont paramétrées. L'utilisateur doit avoir la possibilité de modifier les paramètres afin d'obtenir pour chaque ensemble de valeurs, la représentation d'un composant particulier d'une famille d'éléments standardisés. Ainsi, tous les composants appartenant à un groupe d'éléments partagent les mêmes caractéristiques fonctionnelles.*

Ces définitions mettent en évidence deux caractéristiques de la géométrie paramétrée :

- le fait d'être associé à des contraintes,
- le caractère génératif du modèle qui permet de recréer une forme à partir de ses paramètres.

Dans les systèmes CAO, les objets géométriques sont souvent spécifiés en utilisant des constructions par contraintes. Par exemple, une droite peut être explicitement parallèle à une autre [Shah et Mäntylä 1995, Zalik 1996]. Le but de la géométrie paramétrée est de permettre la modification dynamique d'entités géométriques par exploitation des contraintes de création. Par exemple, si un cercle est créé en utilisant l'intersection de deux droites, les contraintes liant le cercle et les droites sont conservées de telle sorte que lorsque l'une des droites est modifiée, le cercle est aussi modifié en conséquence.

Il existe deux approches pour conserver les contraintes entre les objets d'un modèle paramétrique :

- l'approche équationnelle ou variationnelle qui consiste à décrire une pièce paramétrée comme un ensemble de valeurs reliées entre elles par des contraintes,
- l'approche fonctionnelle ou paramétrique qui consiste à décrire une pièce paramétrée comme une composition de fonctions de construction.

Bien que l'approche équationnelle soit plus simple à utiliser dans le cadre de la géométrie 2D, elle est difficilement applicable pour les constructions en 3 dimensions [Shah, et al. 1994]. Les systèmes 3D sont donc basés (au moins pour les constructions 3D) sur une approche paramétrique.

La Figure III-1 montre un exemple de la construction paramétrique d'un cercle dont la valeur du rayon est le résultat du calcul d'une expression et dont le centre est l'intersection de deux droites.

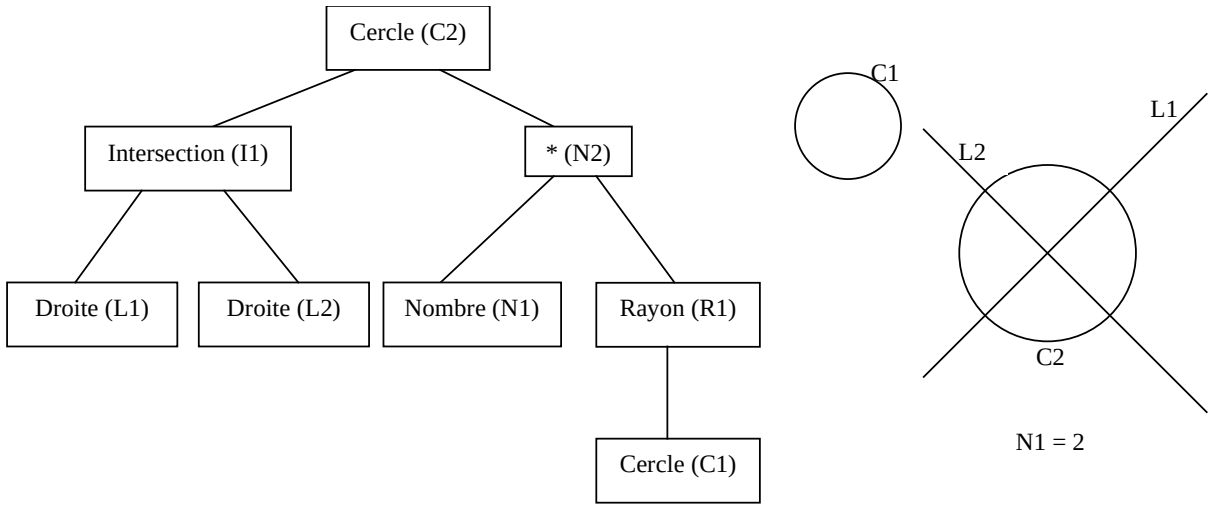


Figure III-1: Arbre de construction

Dans les modèles paramétriques, chaque entité connaît la suite des fonctions qui ont été utilisées pour sa construction. Cette séquence de fonctions peut être vue comme un arbre de construction (Figure III-1). Elle peut être utilisée pour créer un programme représentant le processus de construction d'un objet. Cependant, la plupart des systèmes de CAO qui utilisent la géométrie paramétrée n'offrent pas les outils nécessaires pour abstraire un programme comme en programmation par démonstration. Avec la géométrie paramétrée, il n'y a pas de notion de paramètres et de constantes, tous les objets représentés dans l'arbre de construction peuvent être modifiés, provoquant la réévaluation des objets qui en dépendent.

L'avantage principal de cette approche, en ce qui concerne la génération de programme, est qu'elle est transparente pour l'utilisateur qui, dans la plupart des cas, ne sait pas qu'il construit un programme lorsqu'il crée une entité géométrique. Par contre, cette méthode a deux inconvénients :

- le premier concerne l'abstraction du programme à partir de l'ensemble des relations représentées dans la base de données. La possibilité d'isoler un programme sous forme éditable et modifiable à partir de l'arbre de construction est rarement offerte à l'utilisateur,
- le second est que peu de systèmes permettent de définir les paramètres du programme : le programme n'a pas de signature.

3 Modèle pour la définition de classe

Le modèle que nous proposons se base sur un modèle paramétrique fonctionnel permettant non seulement d'enregistrer le processus de construction d'un objet sous forme d'un programme explicite, mais aussi les méthodes de calcul de ses attributs. Pour définir des attributs, nous proposons de générer, toujours par utilisation d'un modèle paramétrique fonctionnel, la fonction de calcul de chaque attribut à partir du résultat de la méthode de construction. Le nom et le type du résultat de cette fonction définissent l'attribut quand sa réalisation sous forme d'un programme paramétrique permet d'y accéder.

Nous proposons un modèle paramétrique fonctionnel plutôt qu'un modèle équationnel comme base de nos travaux pour deux raisons :

- l'utilisation de primitives de construction en 3 dimensions qui sont difficiles à mettre en œuvre dans les modèles équationnels,
- pour décrire les différentes méthodes, constructeurs et sélecteurs, nous proposons de nous baser sur l'enregistrement du processus mis en œuvre sur l'exemple. Or les modèles équationnels ne se basent pas sur un mécanisme d'enregistrement, ils se basent sur des contraintes définies à posteriori qu'ils ont la charge de satisfaire.

Dans cette section, nous décrivons le modèle que nous proposons pour la définition interactive de classes. En premier lieu, nous présentons la structure générale du modèle paramétrique que nous proposons d'utiliser pour représenter les classes et leurs méthodes. Ensuite, nous abordons la définition des classes proprement dites en commençant par décrire la création du constructeur, puis en montrant la définition des paramètres et des fonctions de calculs des attributs.

3.1 Le modèle paramétrique

La principale caractéristique des modèles paramétriques par rapport aux modèles géométriques est la possibilité de conserver le processus de construction de toutes les entités géométriques. Cette section décrit comment les objets de notre modèle sont à même de conserver leur processus de conception.

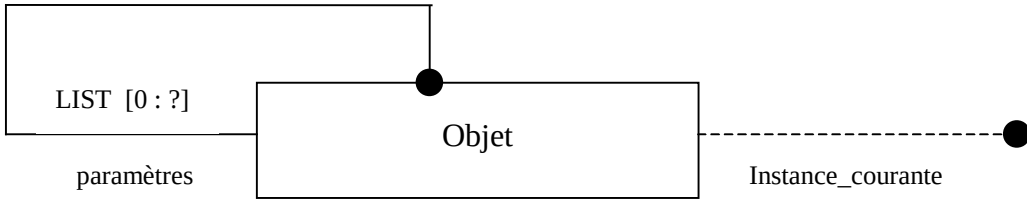


Figure III-2: Classe du modèle paramétrique

La Figure III-2 montre un schéma EXPRESS[3] [Schenck et Wilson 1994] de notre modèle paramétrique. La classe *Objet* représente la classe mère de notre modèle, toutes les autres classes en dérivent. Ainsi, chaque classe de notre modèle possède un lien (optionnel) vers une valeur du modèle géométrique (ou numérique) qui dépend de la classe (l'attribut *Instance_courante*). Par exemple, la classe *Cercle* possède un lien vers un cercle du modèle géométrique alors que la classe nombre possède un lien vers le type réel prédéfini.

Ce lien est optionnel car au cours de la construction, certains éléments géométriques peuvent disparaître. Par exemple, les deux solides impliqués dans une relation booléenne peuvent disparaître de la base de données lors de la réalisation de cette opération. Les références à ces objets géométriques, devenues inexistantes, peuvent être représentées grâce aux instances d'objets du modèle paramétrique qui leurs correspondaient lors de leurs créations.

L'attribut *paramètres* est la liste des instances du modèle paramétrique utilisées pour créer une instance. Ainsi, chaque instance du modèle paramétrique connaît ses paramètres, et chacun d'entre eux connaît ses propres paramètres etc…

Notre modèle utilise le modèle de représentation de la connaissance procédurale proposé dans [Aït-Ameur, et al. 1995, ISO13584.20 1998], et en particulier, le schéma de définition des expressions simples. Il est constitué d'un ensemble de classes représentant les différents types de données manipulées par l'application. Ces classes sont dérivées en sous-classes représentant

[3] Le langage de description EXPRESS est décrit en annexe

les constructions permettant de calculer une de leurs instances. Par exemple, la classe numérique est la mère des classes addition et multiplication. Ainsi, chaque classe non abstraite représente une fonction constructeur de l'application. Elle hérite de la classe décrivant le type de sortie de la fonction. Les classes représentant un type de données, indépendamment de leur processus de construction, sont appelées classes de base.

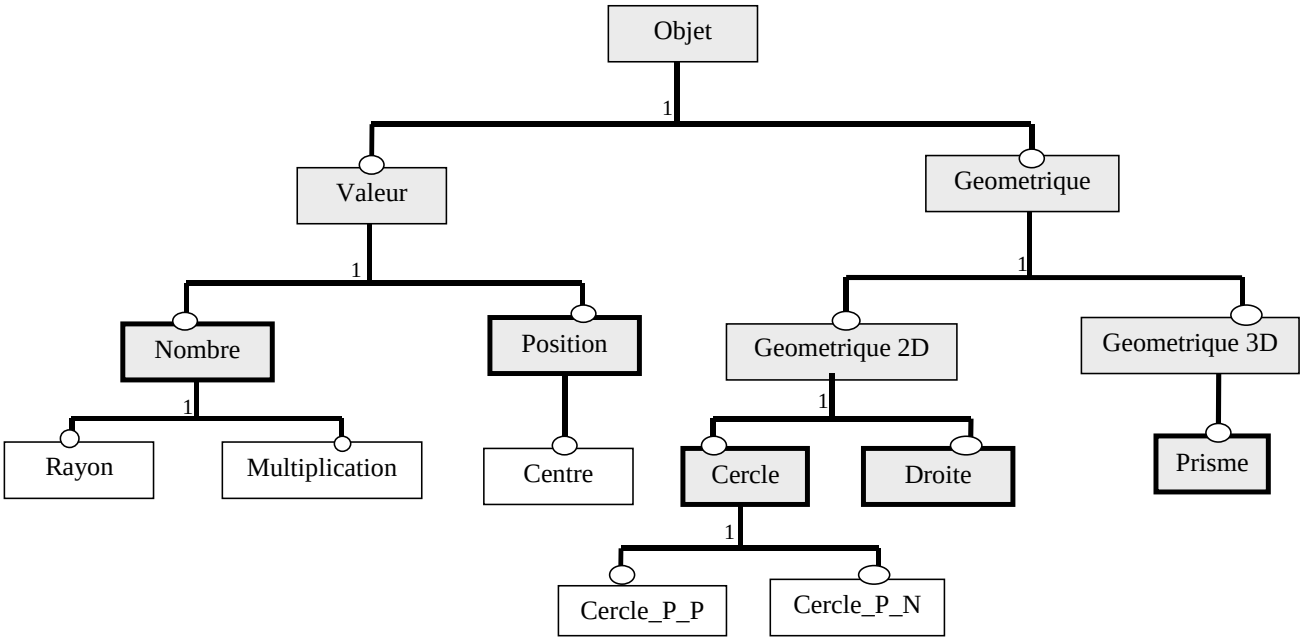


Figure III-3: Exemple d'arbre d'héritage

La Figure III-3 représente un exemple de modèle paramétrique. Les rectangles grisés représentent les classes abstraites de ce modèle. Les rectangles aux contours épais représentent les classes de bases. Ainsi, la classe abstraite de base *Cercle* est dérivée en deux sous-classes : *Cerle_P_N* qui caractérise les cercles construits par centre (position) et rayon (numérique), et *Cercle_P_P* qui permet de créer des cercles en spécifiant leur centre et un point de passage. Ainsi, chaque objet utilisé par l'application connaît, non seulement son type et ses paramètres de construction, mais aussi son processus de construction.

Il est important de noter que les objets venant directement de la couche de présentation tels que les pointés de l'utilisateur ou les nombres saisis directement sont instances de sous-classes de positions et nombres. Ils sont appelés littéraux : *Nombre_Littéral*, *Position_Littéral*.

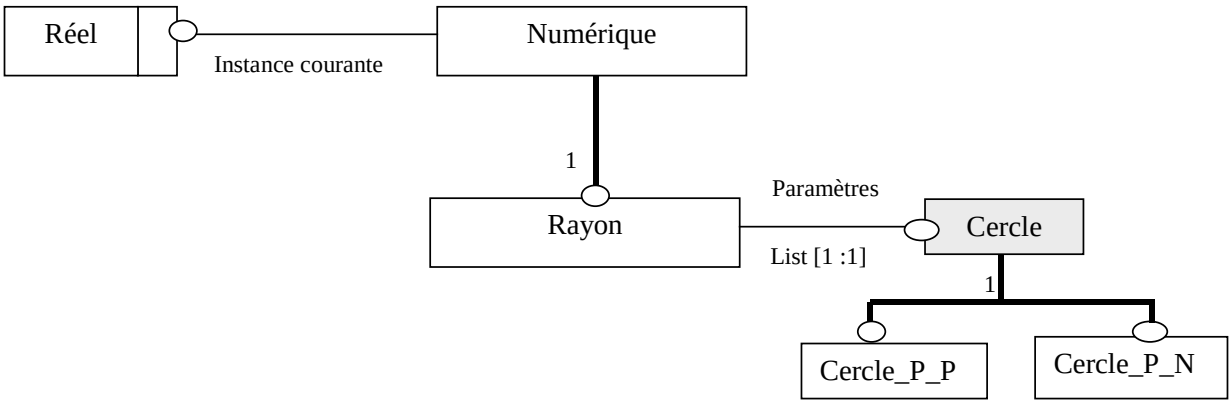


Figure III-4: Relations entre les classes

La Figure III-4 montre les relations entre les classes de notre modèle en se basant sur l'exemple de la classe *Rayon* qui permet de calculer le rayon d'un cercle. Cette classe hérite de la classe *Numérique* dont l'instance courante est de type Réel. Elle ne possède qu'un paramètre qui est de type Cercle. Lors de l'instanciation d'un objet rayon, son paramètre sera effectivement soit un *Cercle_P_P,* soit un *Cercle_P_N* (polymorphisme).

L'attribut *Paramètres* de chaque objet permet de définir complètement l'arbre de construction d'une pièce paramétrique. Chaque objet connaît son type (correspondant au type de sa classe de base), sa méthode de construction et les objets du modèle utilisé pour cette construction (attribut Paramètres de chaque objet).

[Pierra, et al. 1996b] propose de décomposer les modèles paramétriques en trois niveaux distincts :

- **l'instance courante** conserve les valeurs géométriques, topologiques et/ou numériques constituant la pièce en cours de construction,
- **la référence paramétrique** conserve le nom de chaque objet ayant participé au processus de construction et ayant abouti à l'instance courante,
- **la spécification paramétrique** contient l'ensemble des relations entre les différentes références paramétriques de la pièce. Elle représente le processus de construction de la pièce.

De la même manière, nous remarquons que le modèle que nous proposons est lui aussi composé de trois niveaux :

- l'instance courante contient la valeur des objets du modèle géométrique à un moment donné. Elle est représentée dans notre modèle par l'attribut *instance_courante*,

- la spécification paramétrique est constituée de l'ensemble des fonctions et contraintes qui forment le processus de construction d'une pièce. Elle est formé de la classe représentant la fonction constructeur et par son attribut Paramètres,
- la référence paramétrique, qui assure l'indépendance entre la spécification paramétrique et l'instance courante, est représentée par le fait que les différentes classes du modèle héritent de la classe *Objet* qui permet de représenter un objet géométrique même si celui-ci n'existe plus dans la base de données.

Cependant, nous remarquons que, si l'instance courante est clairement séparée de la référence paramétrique, la référence paramétrique et la spécification paramétrique sont étroitement liées. En fait, on peut considérer que les classes de base représentent la référence paramétrique alors que leurs sous-classes représentent la spécification paramétrique.

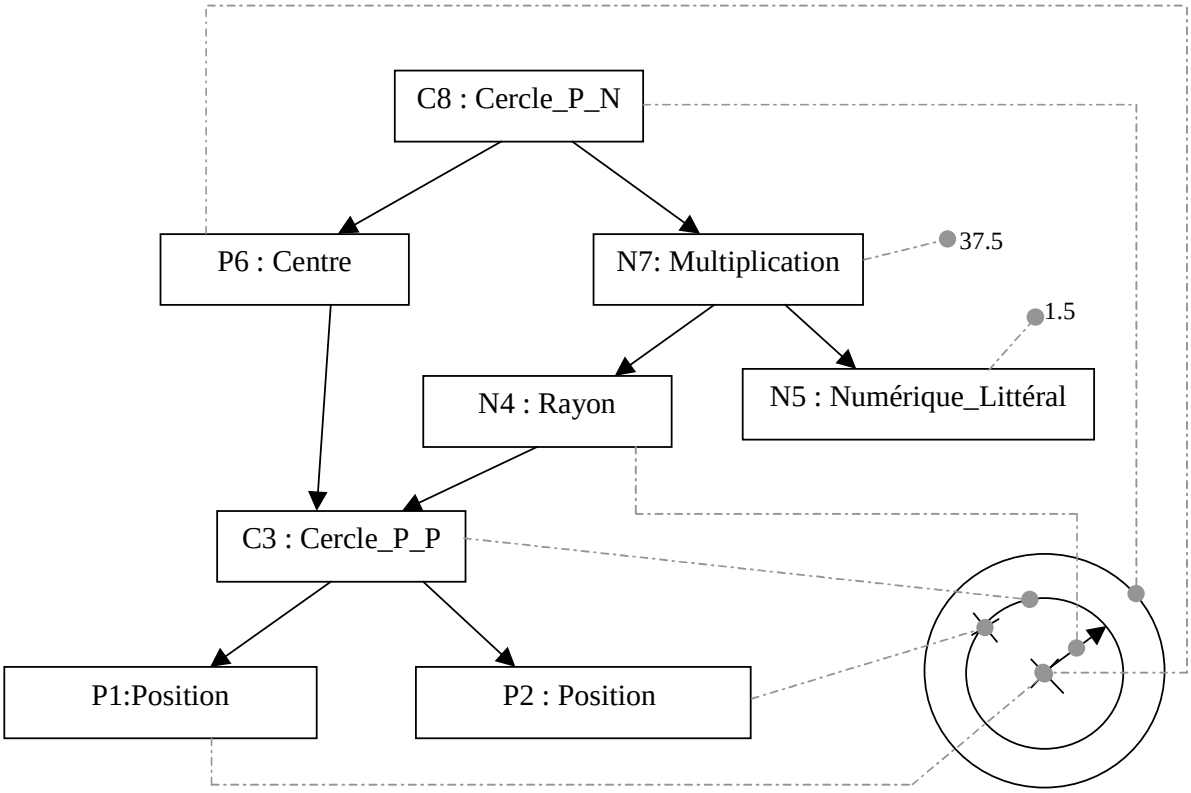


Figure III-5: Arbre de construction d'une pièce : La roue

La Figure III-5 montre l'arbre de construction d'une pièce constituée de deux cercles concentriques.

Remarque : contrairement à la terminologie usuelle qui parle d' « arbre », le graphe de construction d'une pièce paramétrique de notre modèle est en fait un graphe acyclique orienté (DAG Directed Acyclic Graph). L'objet *C3* est en effet utilisé plusieurs fois dans le processus de conception.

Le cercle *C3* est construit à l'aide de deux positions fournies par l'utilisateur. Le cercle *C8* a le même centre que *C3* et son rayon est une fois et demi plus grand. Les rectangles sur la figure représentent des instances de classes du modèle paramétrique. Les flèches représentent les relations entre les objets et leurs paramètres de construction. Par exemple, le cercle *C8* a pour paramètres de construction la position *P6* et le numérique *N7*. Les traits mixtes gris désignent l'instance courante de chaque objet. Les références paramétriques sont les noms *P1, P2, C3, N4, N5, P6, N7, C8*. La spécification paramétrique de *C8* est tout son arbre de construction (en fait son « DAG »). L'instance courante de la roue est l'ensemble des instances courantes du DAG de tête C8.

Les DAG de construction des objets du modèle paramétrique sont utilisés dans notre modèle de définition de classes pour décrire tous les processus de construction ou de calcul. Ainsi, ils permettent de décrire le constructeur des classes et les fonctions de calcul des attributs. Nous appelons DAG(objet) l'ensemble des sous-objets de racine « objet ».

3.2 Définition de classes

La définition interactive d'une nouvelle classe d'objets nécessite de surmonter deux difficultés :

1. la définition du constructeur de la classe permettant de créer une instance de la classe. Cette phase de la définition interactive d'une classe nécessite la définition de paramètres de construction,
2. la définition d'attributs et l'enregistrement de leur processus de calcul à partir des valeurs contenues dans l'instance.

Dans la suite de cette section, nous proposons une solution pour chacun de ces deux problèmes. Pour cela, nous proposons de nous baser sur le modèle paramétrique défini dans la section précédente.

3.2.1 Les constructeurs

Nous proposons d'abstraire le constructeur d'une classe à partir d'un objet du modèle paramétrique représentant une instance de cette classe, tel que cela est réalisé en programmation par démonstration où un programme est abstrait à partir d'un exemple de son utilisation.

Comme nous l'avons vu dans la section précédente, chaque objet du modèle paramétrique conserve son arbre de construction sous la forme d'un DAG dont les feuilles sont des objets dont la valeur a été directement fournie par l'utilisateur et n'est pas le résultat d'un calcul. Pour définir le constructeur d'une classe, nous proposons d'utiliser ce processus de construction en identifiant parmi tous les nœuds et les feuilles du DAG quels sont les objets qui représentent des paramètres effectifs de l'instance. Les autres objets du DAG, utilisés pour abstraire le constructeur de la classe, seront considérés comme des constantes s'il s'agit de feuilles du DAG ou sinon, comme des variables internes.

Contrairement à la programmation par démonstration, la géométrie paramètrée ne fait pas la distinction entre les paramètres, les variables internes et les constantes du processus de construction. Nous proposons donc d'introduire la notion de paramètre dans notre modèle paramétrique afin que l'utilisateur soit en mesure d'identifier parmi tous les objets qui composent le DAG de l'instance lesquels sont des paramètres de la classe. Pour cela, nous proposons d'introduire un attribut de type booléen (Class_Constructor_Param) (Figure III-6) qui indique si un objet du modèle paramétrique représente, ou non, un paramètre de construction pour une classe. Cet attribut possède la valeur faux par défaut. Ainsi, par défaut, toutes les feuilles du DAG servant à créer le constructeur sont considérées comme des constantes de ce constructeur, et les nœuds de l'arbre sont considérés comme des variables internes. C'est à l'utilisateur de spécifier quels sont les paramètres du constructeur de l'instance parmi les objets composant le DAG de construction de l'exemple.

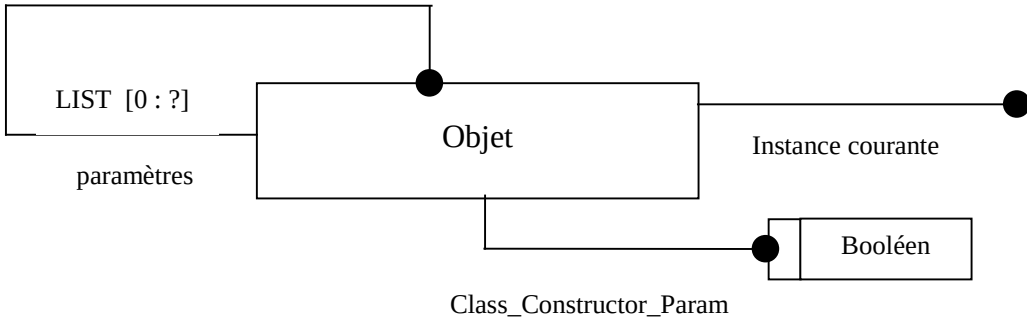


Figure III-6: Ajout de la notion de paramètre dans le modèle paramétrique

Reprenons l'exemple de la Figure III-5 appelé Roue : les paramètres de cette pièce sont le centre des cercles et un point de passage du plus petit cercle.

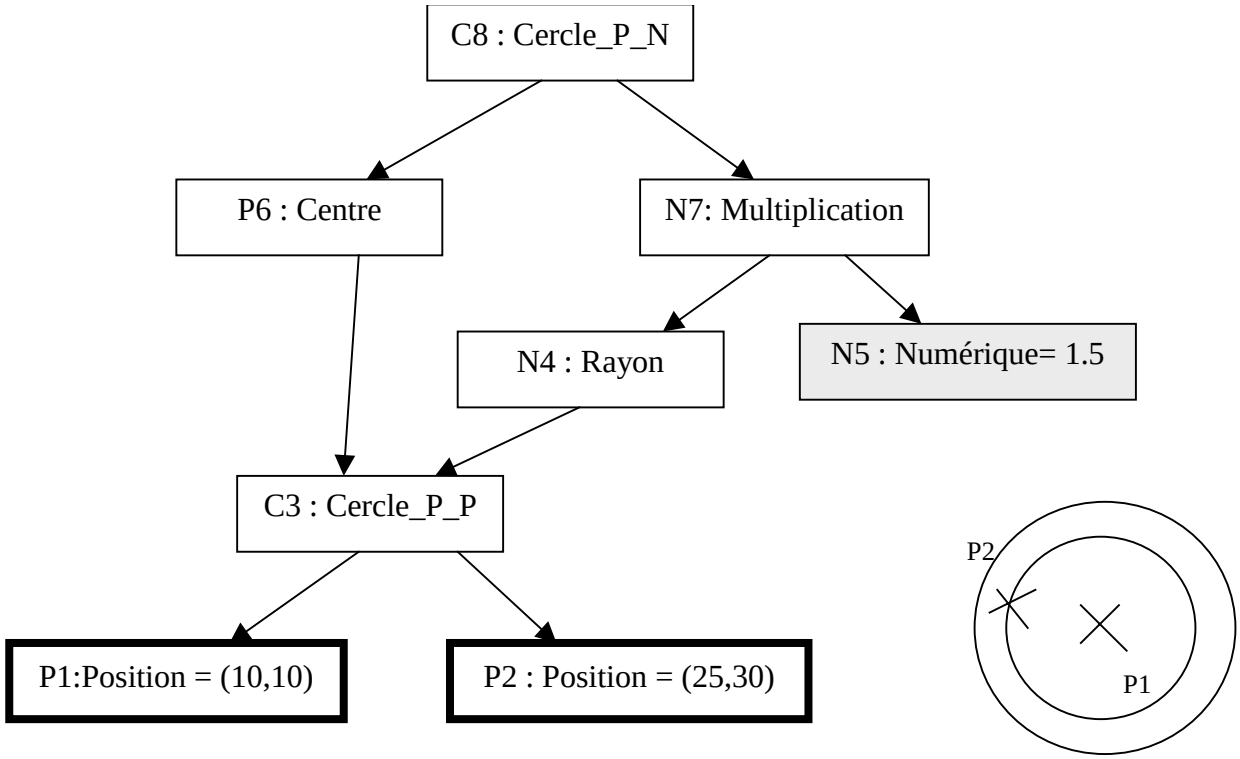


Figure III-7: Paramètres et constantes de la roue

Sur la Figure III-7, les rectangles aux contours épais représentent les paramètres du processus de construction de la pièce, le rectangle grisé représente une constante ; les autres rectangles sont des variables du processus de construction. Donc, pour cet exemple, le rayon du grand cercle conservera toujours le même rapport avec le rayon de l'autre et ce quelle que soit la valeur des deux positions paramètres.

Lors de la génération de la classe à partir du DAG de construction de l'exemple, les objets représentant des paramètres effectifs de l'exemple sont transformés en paramètres formels (i.e. le générateur ne tient pas compte de leur DAG de construction, l'expression ayant fourni une valeur à ce paramètre lors de la construction de l'exemple, est ignorée au moment de la génération du constructeur de la classe). Leur type est remplacé par le type de base de leur classe. Par exemple, si l'utilisateur indique que le centre d'un cercle est un paramètre de la classe, alors le système le remplace au moment de la génération par un objet de type position et ne tient pas compte de la manière dont sa valeur a été calculée. Par exemple, si la position P1 est construite par l'intersection de deux droites *D11* et *D12*, son type est intersection. Lors de la génération de la classe, il est remplacé par le type position, et les droites *D11* et *D12* n'apparaissent pas dans le constructeur de la classe.

3.2.2 Les attributs

Bien que ce ne soit, à notre connaissance, jamais présent ni dans les systèmes de programmation par démonstration, ni dans les systèmes paramétriques, la possibilité de définir des attributs et leurs fonctions de calcul est un point essentiel lors de la création d'une classe d'objets. Pour passer de la création interactive de programme à la création de classes, il est nécessaire d'offrir à l'utilisateur la possibilité de décrire non seulement le processus de construction des objets de la classe mais aussi un processus d'accès aux attributs de ces objets. Les attributs d'un objet, comme par exemple le centre ou le rayon d'un cercle, ont un rôle essentiel dans les modèles paramétriques. Ils permettent en effet d'exprimer des contraintes entre les objets (par exemple, droite passant par le centre d'un cercle). Donc, pour que des objets, issus de classes définies interactivement, soient utilisés dans le processus de construction d'un autre objet, il est nécessaire que certains de leurs attributs puissent être référencés.

La méthode que nous proposons pour définir un attribut, ainsi que sa fonction de calcul, consiste à associer, au sein de l'objet représentant l'instance de la classe, un objet du modèle paramétrique à une chaîne de caractères et à un arbre de construction paramétrique. La chaîne de caractères représente le nom de l'attribut. L'objet contient le type de l'attribut ainsi que sa méthode de calcul sous la forme de son arbre de construction.

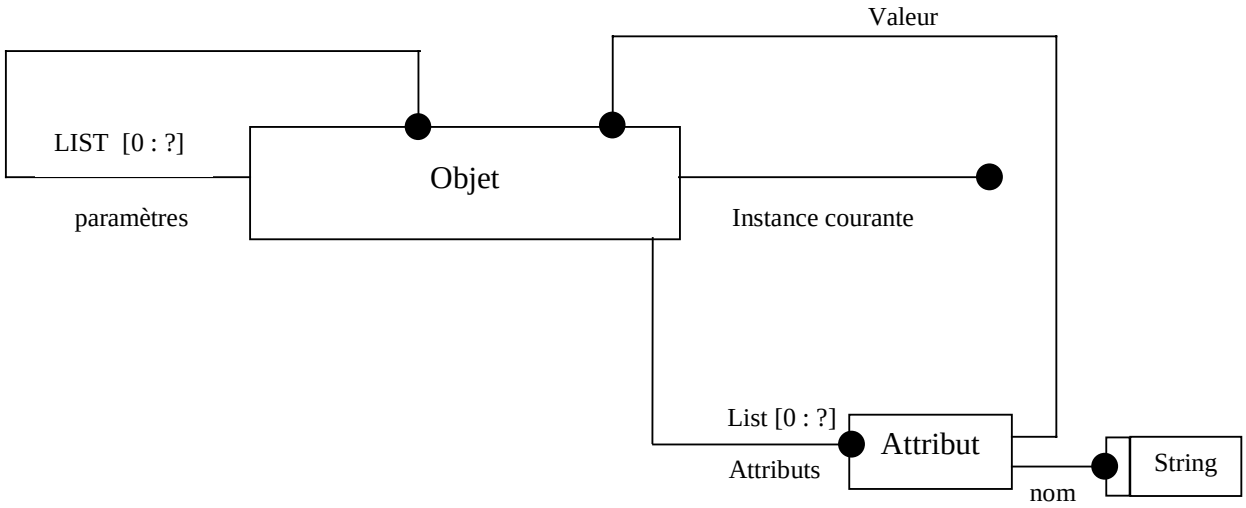


Figure III-8: Ajout de la notion d'attribut dans le modèle paramétrique

La Figure III-8 montre les ajouts apportés à notre modèle afin qu'il puisse supporter la définition d'attributs. L'attribut « Attributs » constitue la liste des attributs caractéristiques définis pour une classe. Dans l'instance courante, un « Attribut » est défini par un nom contenant une chaîne de caractères et par un objet du modèle paramétrique représentant son type. Cet objet possède son DAG de construction comme tous les objets du modèle paramétrique. Donc, lorsque l'un des paramètres de l'instance est modifié, la valeur de l'instance est réévaluée ainsi que la valeur de ces attributs.

A titre d'exemple, nous ajoutons, à la pièce décrite sur la Figure III-7, un attribut dérivé représentant la distance entre les contours des deux cercles. Cet attribut représente la taille du pneu de la roue.

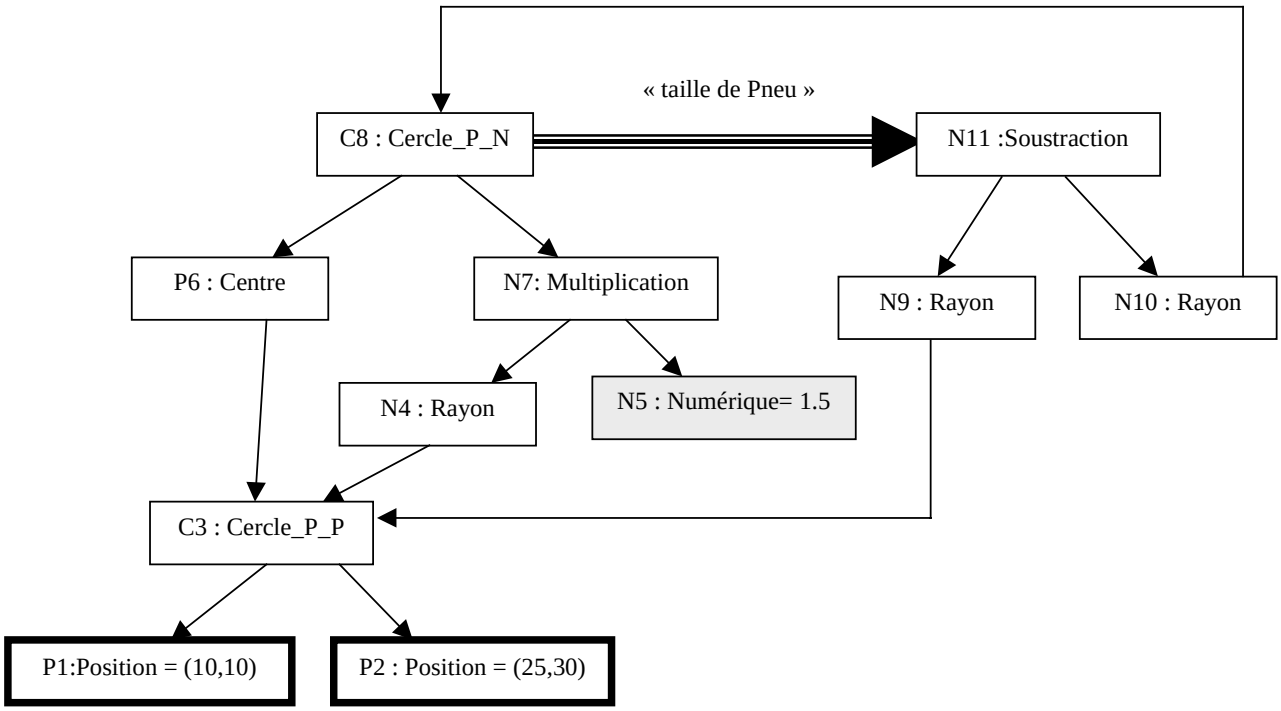


Figure III-9: Définition de l'attribut « taille de pneu » de la roue

La triple flèche sur la Figure III-9 représente le lien entre l'objet C8 et son attribut. Il est important de noter que les feuilles du DAG de construction de l'attribut sont, soit des objets du DAG de construction de la classe, soit des constantes. En effet, les attributs d'un objet représentent la structure interne de cet objet, et à ce titre, leurs valeurs doivent être calculées au moment de l'instanciation de l'objet. Donc, le calcul de ces valeurs ne peut dépendre que de constantes ou des valeurs des paramètres de construction de l'objet.

3.3 Synthèse

Dans cette section, nous avons proposé un modèle permettant de conserver les données nécessaires à la génération d'une classe d'objets à partir d'un exemple d'instance. Pour cela, nous nous sommes basés sur un modèle paramétrique fonctionnel que nous avons étendu par deux notions :

- La notion de paramètre et de signature issue du domaine de la programmation par démonstration,
- La notion d'attribut issue du paradigme objet et défini par la fonction paramétrique permettant de le calculer.

Afin de préciser les relations entre un système paramétrique usuel et le système que nous proposons pour la génération interactive de classes, nous présentons, dans cette section, un modèle de ces relations. Dans la Figure III-10 ci-dessous, la partie grisée représente un « modèle paramétrique », c'est à dire une structure permettant de représenter et de ré-exécuter des arbres de construction telle qu'elle peut exister dans les systèmes usuels supportant la géométrie paramétrique. La partie non grisée représente la super-structure que nous proposons pour permettre de transformer un système paramétrique en un système permettant l'adjonction interactive de nouvelles classes d'objets. Nous présentons ensuite, les règles qu'un générateur de classe (basé sur l'approche que nous proposons) doit respecter.

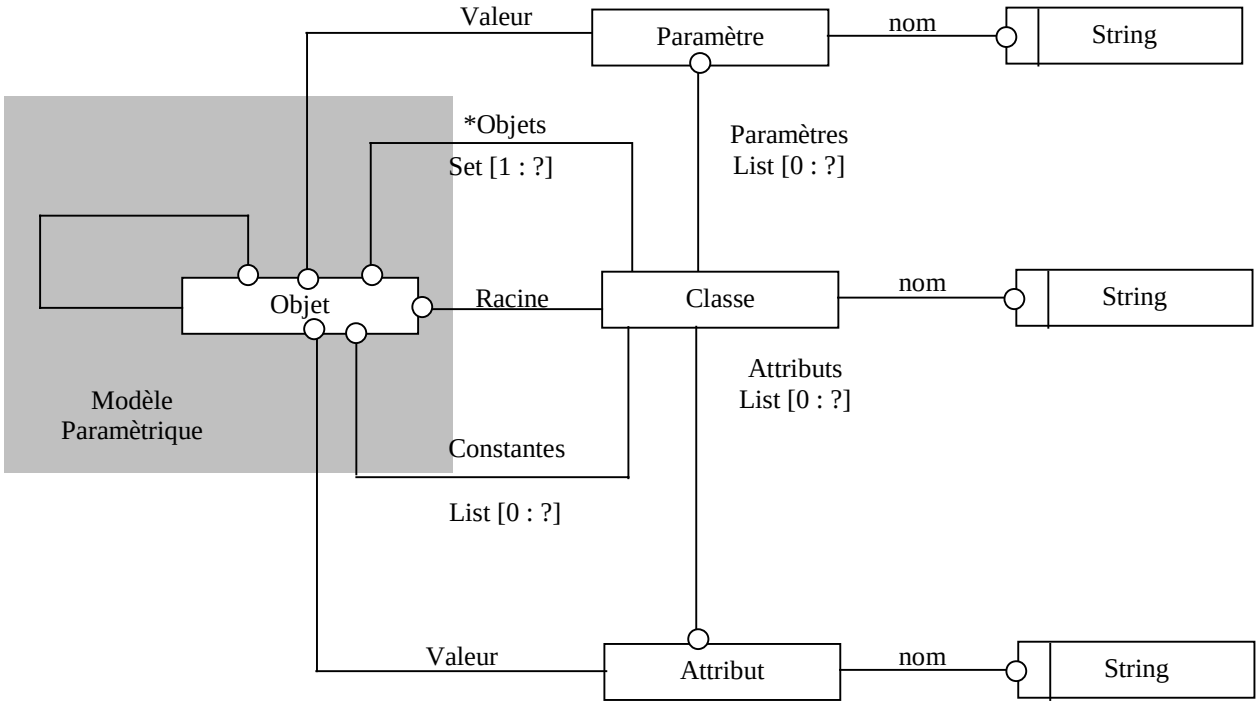


Figure III-10: Modèle des classes

Le modèle présenté sur la Figure III-10 correspond au modèle de définition des classes à partir d'un modèle paramétrique. Toutes les relations entre les attributs nécessaires à la définition d'une classe apparaissent:

- son nom, le processus de construction de ses instances, représenté par l'attribut *racine*,
- la liste de ses paramètres de construction, composés d'un nom et d'une valeur qui est un objet du modèle paramétrique. En phase de définition de l'instance, exemple de la classe, ces objets jouent le rôle de paramètres effectifs du constructeur de la classe. En phase de

génération de la classe, ces objets permettent au générateur de connaître le type des paramètres formels qu'ils représentent,

- la liste de ses attributs ; un attribut est composé d'un nom et de son processus de calcul contenu dans l'objet représenté par l'attribut *Valeur*,
- l'attribut dérivé *Objets* contient l'ensemble des objets intervenant dans le constructeur de la classe. Il est calculé à partir d'une méthode de parcours du DAG de l'instance que nous détaillons dans la suite.

Les principales caractéristiques du générateur de constructeurs de classe sont illustrées dans la suite. Ce générateur possède deux rôles importants dans le processus de définition d'une classe :

1. il est chargé d'abstraire le DAG représentant le processus de construction de la classe à partir de l'arbre de construction de l'instance exemple (représenté par l'attribut *Racine* sur la Figure III-10),
2. il doit définir le DAG représentant les fonctions de calcul des attributs associés à la classe à partir de l'objet contenant leur spécification paramétrique.

3.3.1 Génération du constructeur de la classe

Pour abstraire le constructeur d'une classe sous la forme d'un DAG à partir du DAG de construction de l'instance, le générateur doit faire la distinction entre les variables internes, les constantes et les paramètres contenus dans le DAG de construction de l'exemple. La définition des paramètres formels est effectuée grâce à la liste des objets présentant les paramètres de la classe. Pour distinguer les objets représentant des variables internes du constructeur de ceux représentant des constantes, nous proposons d'utiliser le principe selon lequel tout objet, contenu dans le DAG de construction de l'instance qui n'est ni un paramètre ni le résultat d'un calcul, représente une constante du constructeur. C'est-à-dire que les constantes du constructeur sont des objets dont la valeur a été fournie par l'utilisateur, et qui ne font pas partie de la liste des paramètres de la classe.

La transformation des paramètres effectifs de l'instance en paramètres formels de la classe entraîne des modifications du DAG représentant le constructeur de la classe par au DAG représentant la spécification paramétrique de l'instance. En effet, les processus de calcul des valeurs des paramètres effectifs ne sont pas pris en compte dans le DAG représentant le

constructeur. Cela nous conduit à énoncer une règle sur la génération du constructeur de la classe à partir du DAG de construction de la classe :

« Un objet o appartient au DAG de construction de la classe si et seulement si il existe un chemin entre l'objet Racine de la classe et o (non compris) ne comprenant aucun paramètre. »

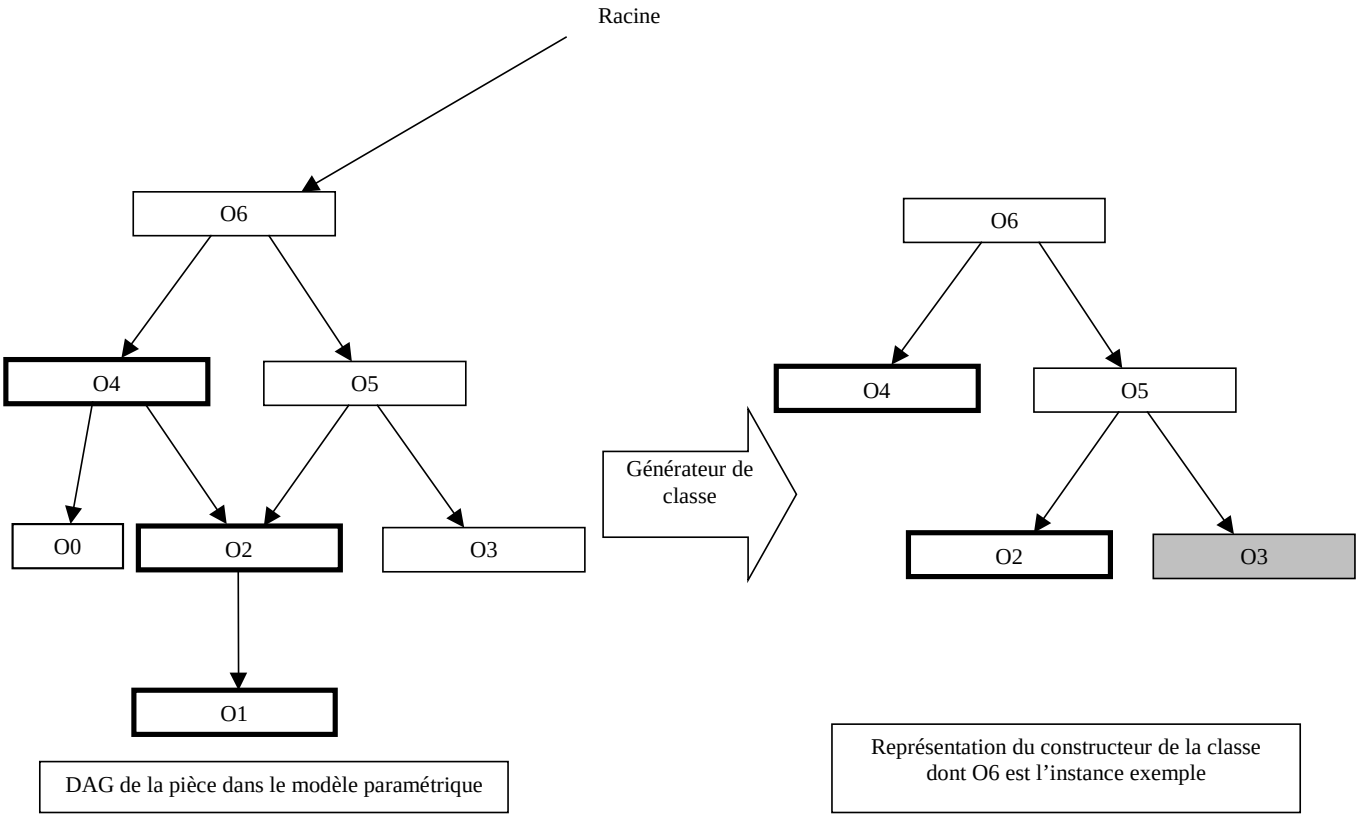


Figure III-11: Création de l'arbre de construction

La Figure III-11 montre la création du DAG représentant le processus de construction des instances de la classe. Les rectangles épais représentent les paramètres, les rectangles grisés sont des constantes, et les autres sont des variables. Les objets utilisés lors de la création de l'exemple pour donner une valeur à un paramètre ne sont pas pris en compte (les objets *O0* et *O1* n'apparaissent pas dans le constructeur de la classe et la liaison entre *O4* et *O2* n'existe plus).

3.3.2 Génération des fonctions de calcul des attributs

Contrairement aux constructeurs, les fonctions de calcul des attributs d'une classe ne possèdent pas de paramètre. Elles utilisent, soit des objets contenus dans le constructeur de la classe (attribut dérivé *Objet* Figure III-10), soit des constantes qui leurs sont propres.

Pour abstraire une fonction de calcul à partir de l'objet représentant la valeur d'un attribut dans l'exemple, le générateur de classes parcourt le DAG de définition de l'attribut de manière descendante à partir de la racine jusqu'à ce qu'il trouve, soit un objet contenu dans le constructeur de la classe, soit une constante (i.e. un objet dont la valeur n'est pas le résultat d'un calcul). Il construit ainsi un DAG, composé de fonctions paramétriques, constituant la fonction de calcul d'un attribut de la classe.

4 Utilisation du modèle

Dans la section précédente, nous avons décrit notre modèle pour la création interactive de classes d'objets. Dans cette section, nous abordons l'utilisation de ce modèle. Nous commençons par donner un exemple de construction de classes. Puis, nous montrons deux exploitations des classes créées interactivement. La première, que nous appelons ré-interprétation, est utilisée pour tester la nouvelle classe dès que l'utilisateur a fini de la créer. La seconde consiste à générer le code de la classe dans un langage de programmation afin que cette classe puisse être intégrée au système utilisé pour la créer ou à un autre système CAO.

Nous basons notre étude sur l'exemple de la création de la classe tube. Un tube est le résultat de la coupure de deux cylindres concentriques. Les paramètres de création d'un tube sont son centre (C), son rayon extérieur (R) et sa hauteur (H). Cette classe possède un attribut : l'épaisseur du tube (e) (Figure III-12).

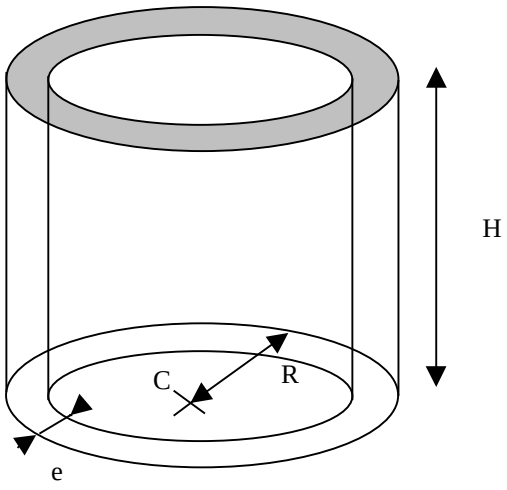


Figure III-12: L'exemple

4.1 Construction de classes

La définition interactive d'une classe se déroule en deux phases : la définition du constructeur et la création des attributs et de leur fonction de calcul.

4.1.1 Définition du constructeur

La définition du processus de construction des instances de la classe se fait automatiquement en créant un exemple d'instance de cette classe. Le constructeur de la classe correspond exactement à l'arbre de construction de l'objet décrivant l'instance de la classe.

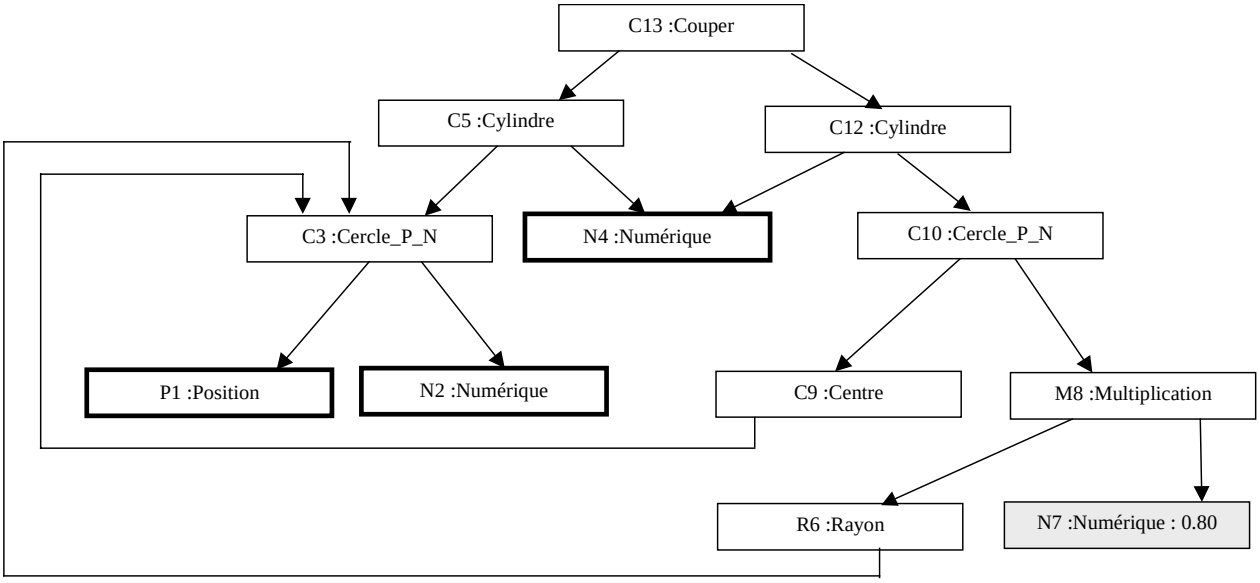


Figure III-13: Arbre de construction d'un tube

On remarque sur l'arbre de construction d'un tube (Figure III-13) que les feuilles de l'arbre sont soit des paramètres, soit des constantes, et que tous les nœuds de l'arbre représentent des variables du processus de construction. Lors de la création d'une fonction ou d'une procédure, le concepteur spécifie l'ordre des paramètres d'entrée. Lors de la création d'un objet servant d'exemple pour la création d'une classe, il ne spécifie pas dans quel ordre les paramètres sont utilisés. Le système utilise, par défaut, l'ordre de création des objets représentant les paramètres de la classe. Sur l'exemple, l'ordre des paramètres du constructeur de la classe tuyau est P1, N2, N4 correspondant aux paramètres C, R et H.

4.1.2 Définition des attributs

L'attribut épaisseur *e* (voir Figure III-12) de la classe *tube* est le résultat de la soustraction entre les rayons des deux cercles ayant permis de créer les deux cylindres. L'objet contenant la valeur de l'attribut épaisseur est enregistré dans l'objet représentant l'instance. Il est associé au nom de l'attribut qu'il représente (ici « épaissseur »). Cet objet est ensuite utilisé par le générateur de classes pour abstraire la fonction de calcul de l'attribut.

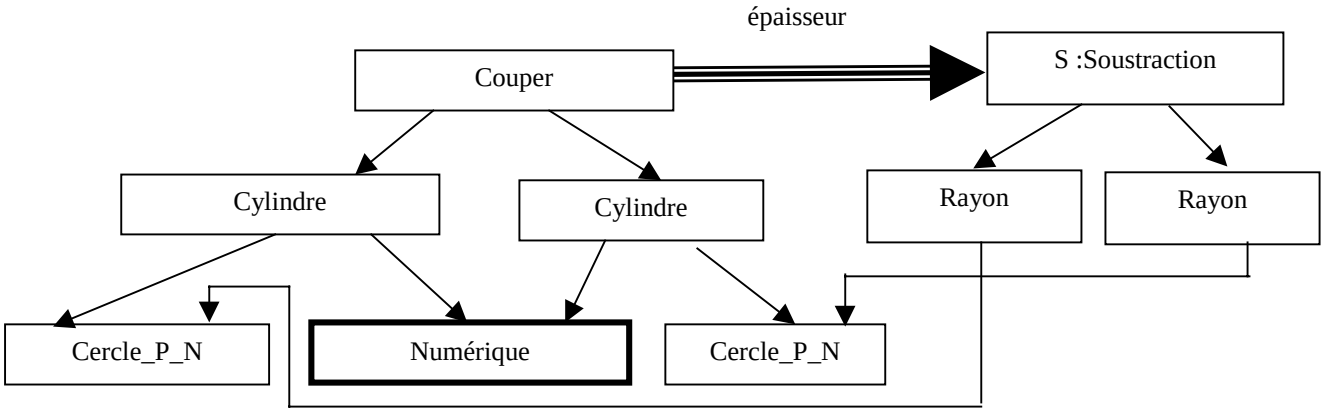


Figure III-14: Définition de l'attribut épaisseur

L'objet *S* définit la fonction de calcul de l'attribut épaisseur. Cette fonction n'a pas de paramètre d'entrée, elle utilise des objets de l'arbre de construction du tube.

4.2 Ré-interprétation

La ré-interprétation des classes créées interactivement permet à l'utilisateur de tester les nouvelles classes qu'il a intégrées dynamiquement au système sans qu'il ait besoin de recompiler l'application. Ainsi, l'utilisateur peut immédiatement corriger les erreurs de conception puisqu'il peut encore accéder à l'exemple ayant permis de créer la classe. On peut comparer la phase de ré-interprétation à une phase de débogage en programmation classique.

L'instanciation de la classe consiste à copier la structure de l'objet servant d'exemple. Tous les DAG de construction, que ce soit de l'objet lui même ou de ses attributs, sont dupliqués sans conserver les liens vers le modèle géométrique. On obtient alors un arbre non évalué (sans lien vers le modèle géométrique) qui représente la classe. A chaque fois que l'utilisateur décide de créer une nouvelle instance de la classe, le système duplique l'arbre non évalué et remplace les objets utilisés comme paramètres formels par des objets (paramètres effectifs) fournis par

l’utilisateur qui possèdent une valeur (i.e. un lien vers le modèle géométrique). Puis, pour créer l’instance, cet arbre est réévalué, en post-ordre, en fonction des valeurs des objets paramètres(voir Figure III-15).

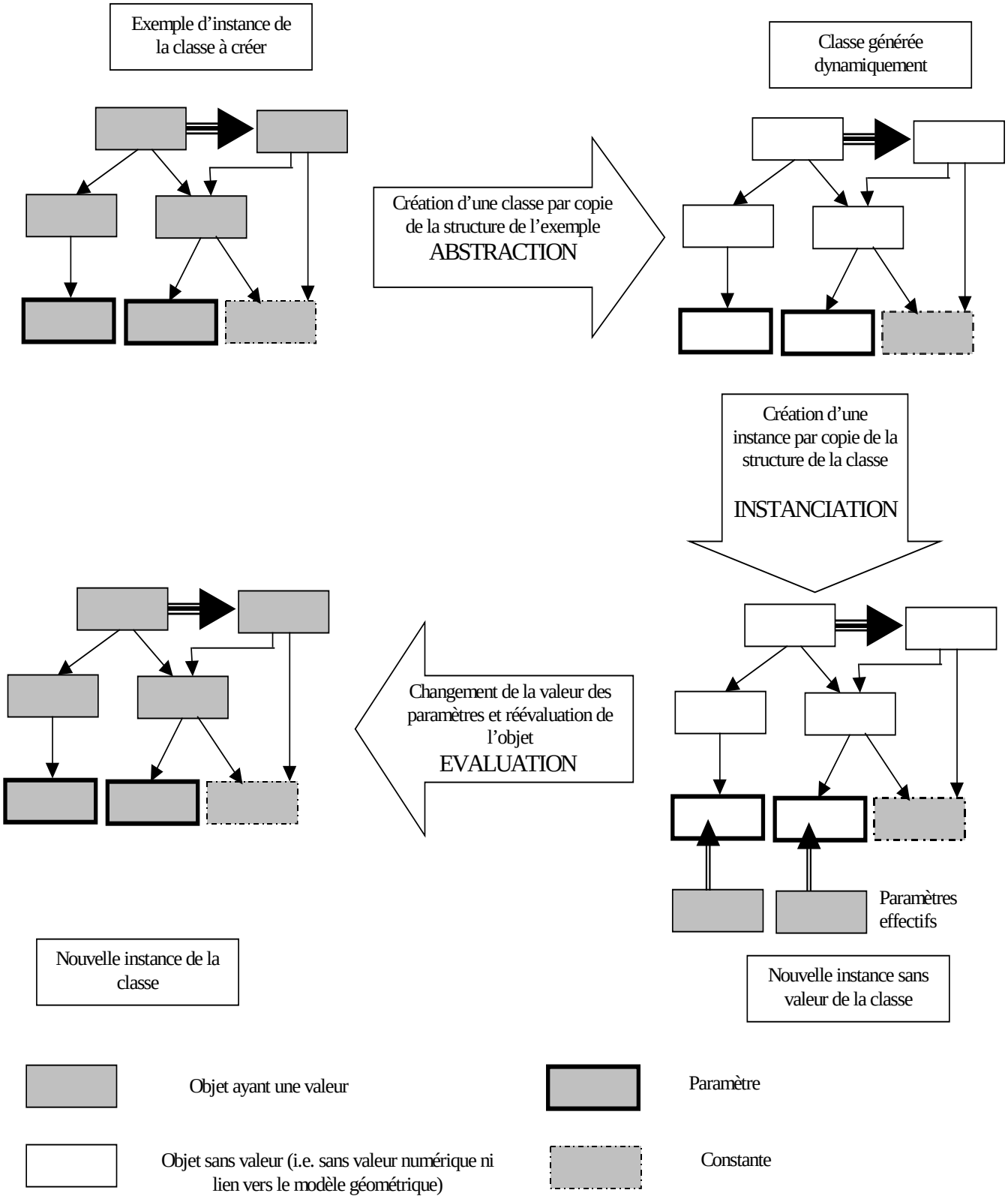


Figure III-15: Utilisation dynamique des classes

La Figure III-15 illustre la création dynamique d'une classe à partir d'un exemple d'une de ces instances. Cette création correspond à l'abstraction décrite en programmation par

démonstration qui consiste à créer un programme à partir d'un exemple de son exécution. Les attributs sont aussi recopiés. Leurs valeurs sont ensuite calculées lors de la phase d'évaluation. En terminologie orientée objet, ce type d'instanciation est appelée prototypage [Cohen et Murphy 1984].

4.3 Génération de code

Une fois qu'une classe créée interactivement a été corrigée, si nécessaire, grâce au mode dynamique permettant le débogage, elle doit être enregistrée pour permettre de la réutiliser comme une classe native du système lors de futures exécutions de l'application.

Cet enregistrement peut se faire de deux manières :

- en générant un fichier dans un langage spécifique qui pourra être réinterprété par l'application qui l'a généré,
- en générant le code de la classe dans un langage de programmation neutre.

La deuxième solution nous semble être la meilleure pour deux raisons :

- elle permet de réaliser l'indépendance entre le système de génération et le code généré. Ce code pourra être utilisé dans le cadre d'autres applications.
- elle permet de créer des classes similaires aux classes natives du système. Cela permet ensuite leur intégration statique dans l'application qui les a générées, améliorant ainsi leur rapidité d'exécution.

4.3.1 Principe

Pour générer le code d'une classe créée dynamiquement, nous reprenons les approches classiques en compilation des langages. En règle générale, la traduction en langage de programmation nécessite l'existence de deux structures de données [AHO, et al. 1991] :

- l'arbre abstrait, définissant la structure et les composants du programme. Le parcours post-fixé de cet arbre reflète l'ordre dans lequel les opérateurs d'une expression peuvent être évalués. La signature d'une action représente le type des paramètres qui doivent lui être fournis et le type du résultat généré,
- la table des symboles, représente le domaine sémantique, où sont collectées toutes les informations concernant chaque identificateur du programme. Par exemple, ce peut être l'emplacement mémoire assigné à un identificateur, son type, sa portée (c'est-à-dire la

partie du programme dans lequel il est valide), et dans le cas des noms de procédures, des renseignements comme la signature, le mode de passage de chaque paramètre (par exemple, si cela s'effectue par valeur ou par référence) etc...

4.3.2 Mise en œuvre sur notre modèle paramétrique

Dans notre cas, les différents DAG de construction présents dans la définition de la classe représentent les arbres abstraits. La table des symboles, quant à elle, est éclatée et représentée par l'ensemble des nœuds de l'arbre. Chacun d'eux, comme nous l'avions vu, porte à la fois son type résultat (par exemple : Cercle) et, par ses liens, sa signature : *Cercle_P_N : position x Numérique* $\rightarrow$ *cercle*.

Cette particularité résulte du fait que le programme n'étant pas textuel, les « noms de variables» usuels sont directement représentés par des pointeurs sur les définitions de variables. La dualité nom/définition, usuellement représentée dans la table des symboles, n'a donc pas lieu d'être.

L'arbre abstrait, représenté ici par les DAG de construction, supporte les différentes méthodes de réécriture usuelles en compilation, notamment les traductions descendantes et ascendantes. Cette distinction fait référence à l'ordre dans lequel les nœuds de l'arbre abstrait sont traduits. Dans les premières, la traduction débute à la racine du DAG pour descendre vers les feuilles; il s'agit en fait d'un parcours d'arbre préfixé. Dans les secondes, la traduction part des feuilles de l'arbre pour remonter vers sa racine ; il s'agit là d'un parcours d'arbre post-fixé.

Il est important de noter que dans le cas de notre étude, pour les raisons évoquées au-dessus, les structures sont des DAG. Le générateur de code doit donc être en mesure de réaliser un parcours postfixé sans réévaluer deux fois le même sous DAG.

Dans ces deux méthodes, la traduction de programmes consiste à parcourir les arbres abstraits, et à générer, pour chacun des nœuds, les instructions correspondantes en langage cible. Cette correspondance peut être réalisée de deux façons [Potier 1995]:

- en créant une bibliothèque de sous-programmes, où chacune des actions susceptibles d'apparaître dans l'arbre abstrait possède une version en langage neutre avec la même interface d'appel, la réécriture d'un nœud se réduit alors à un appel de fonction,
- en créant une bibliothèque de blocs d'instructions, où chacune des actions susceptibles d'apparaître dans l'arbre abstrait possède son propre modèle d'instructions, caractérisé par des identifiants formels à substituer lors de l'intégration dans le code source de la classe.

Les blocs d'instructions ont l'avantage d'être adaptables et réutilisables à volonté, mais présentent l'inconvénient de répéter un grand nombre de fois, dans la classe générée, des séquences d'instructions identiques. A l'inverse, l'utilisation des bibliothèques de sous-programmes a l'avantage d'assurer un certain niveau d'abstraction et une certaine réduction du code, mais a l'inconvénient de manquer d'ouverture lorsque l'on souhaite ajouter de nouvelles primitives au langage abstrait (i.e. de nouvelles actions à l'application), puisque la nouvelle bibliothèque de sous-programmes doit alors être fournie à tous les systèmes devant exécuter le programme.

Le code à générer pour créer une classe d'objets dépend bien sûr du langage cible, mais aussi du système et de l'application destinataire de ce code. Si la classe est uniquement utilisée dans le cadre de l'application qui la créée, le générateur peut utiliser les bibliothèques de sous-programmes disponibles, comme par exemple, l'interface d'accès programmée avec les classes qui représentent les objets natifs du système. Dans le cas où la classe est destinée à un autre système, il faut impérativement utiliser des bibliothèques de blocs d'instruction où ne figurent que des appels à des interfaces d'accès programmées, neutres et standards, par exemple normalisées [ISO13584.31 1998].

4.3.3 Exemple

```
Tube : :Tube (Position *P1, Numérique *N2 , Numérique *N4) {
        Cercle_P_N *C3 = new Cercle_P_N (P1,N2) ;
        Cylindre *C5 = new Cylindre (C3,N4) ;
        Rayon *R6 = new Rayon (C3) ;
        Numérique *N7 = new Numérique ;
        N7->Value = 0.8 ;
        Multiplication *M8 = new Multiplication ( R6, N7) ;
        Centre *C9 =new Centre(C3) ;
        Cercle_P_N *C10 =new Cercle_P_N (C9,M8) ;
        Cylindre *C12 = new Cylindre (C10, N4) ;
        Couper *C13 = Couper (C5, C12) ;
}
```

Figure III-16: Exemple de code généré

Le code de la Figure III-16 est une partie simplifiée du code généré pour le constructeur de la classe « Tube » dont le DAG de construction est décrit sur la Figure III-13. Nous utilisons ici la bibliothèque de sous-programmes constituée des classes C++ natives de l'application qui a servi pour définir l'exemple d'instance.

5 Exploitation des classes générées

Les classes générées à partir de notre modèle peuvent être utilisées dans deux cadres distincts : la spécialisation d'applications (qui constitue le cadre principal de notre étude) et la création de bibliothèques de composants standards.

En général, le terme composant standard désigne un objet dont l'usage est multiple ou répétitif. Typiquement, les vis, rondelles, butées, etc., appartiennent à cette catégorie [Potier 1995]. Parmi les composants standards, plusieurs catégories peuvent être différenciées. Différents travaux [Hochberg 1989, Moranne 1988] mettent en évidence une différence entre les standards externes aux entreprises (composants normalisés ou du commerce dont la description est normalement effectuée par le fournisseur, et donc échangée) et les standards internes à chaque entreprise, souvent décrits sur le système où ils sont utilisés. La spécialisation d'application revient à définir des composants internes à une entreprise. La création de bibliothèques de composants revient à la définition de composants externes.

Selon le cadre d'utilisation du modèle, le mode de génération du code des classes est sensiblement différent.

5.1 Spécialisation d'application

La spécialisation interactive d'applications consiste à créer de nouvelles classes d'objets à partir de la description de l'une de leur instance. Le code généré est alors compilé et intégré à l'application de telle manière que la nouvelle classe ait le même comportement que les classes natives du système.

Dans le cas de la spécialisation d'application, la correspondance entre les nœuds de l'arbre abstrait et les instructions du programme peut être réalisée à l'aide d'une bibliothèque de sous-programmes. Cette bibliothèque est en fait l'ensemble des classes natives du système. A chaque fois qu'une nouvelle classe est créée de manière interactive, elle est ajoutée à la bibliothèque des classes du système et peut alors être utilisée pour définir d'autres classes d'objets.

5.2 Bibliothèques de composants

Pour permettre les échanges entre systèmes CAO hétérogènes, des projets normatifs internationaux visent à définir des bibliothèques portables de composants dont les géométries sont définies par des programmes de paramétrage de type procédural appuyés sur une interface de création de géométrie, elle-même normalisée. Par exemple, la norme [ISO13584.31 1998] définit une interface normalisée de création de géométrie en Fortran. Cette norme est en cours d'extension pour supporter aussi une interface utilisant le langage JAVA.

Le modèle paramétrique que nous proposons est capable d'enregistrer et de générer des classes d'objet. Ces classes peuvent être traduites dans un langage neutre en suivant les spécifications fournies par une interface de création de géométrie normalisée. Dans ce cas, la traduction de l'arbre abstrait représentant la classe ne passe plus par l'utilisation d'une bibliothèque de sous-programmes mais par l'utilisation de blocs d'instructions. Ainsi, le programme généré peut être ré-exécuté par n'importe quel système supportant l'interface de programmation normalisée.

6 Conclusion

La conception interactive d'applications spécialisées de conception technique nécessite d'offrir, la possibilité à l'utilisateur de créer de nouvelles classes d'objets interactivement, c'est-à-dire sans programmation explicite.

Une classe d'objets est composée d'une partie statique regroupant des attributs décrivant l'objet et d'une partie dynamique constituée de la méthode de création de l'objet ainsi que des méthodes de construction utilisé pour fournir des valeurs aux attributs.

Il existe deux grandes méthodes de programmation interactive :

- la programmation par démonstration qui se base sur un exemple d'exécution pour générer un programme, associée à une signature, et
- la géométrie paramétrée qui conserve sous forme d'un graphe acyclique le processus de construction de chaque objet afin de pouvoir le réévaluer lors des modifications de ses constituants.

Ces deux méthodes, si elles permettent de créer des « programmes », ne permettent

- ni de les regrouper au sein d'une même structure de données pour créer une classe,
- ni de les associer à des attributs,

- ni même, pour la géométrie paramètrée, d'identifier leur signature.

Afin de permettre la génération interactive de classes d'objets, nous avons proposé ici une approche capable de regrouper, au sein d'une même classe, plusieurs types de programmes différents. Cette approche permet de générer non seulement le processus de conception des instances de la classe (constructeur), mais aussi des fonctions de calcul permettant d'accéder aux attributs de l'objet (selecteurs).

Cette approche se base sur un modèle paramétrique (conservation des processus de construction des objets) que nous avons décoré avec des attributs permettant d'enregistrer le processus de construction d'autres objets (définition d'attributs). De plus :

1. nous avons emprunté à la programmation par démonstration la notion de différenciation entre paramètres, constantes et variables, chaque objet contenu dans l'arbre de construction connaissant son statut (paramètre, constantes ou variable). Ceci nous a permis, comme cela est nécessaire en programmation, d'associer à tout programme une signature,
2. nous avons défini un mécanisme permettant d'associer à la classe des attributs interrogeables. Chaque attribut est en fait défini par la fonction (paramétrique) qui permet d'en générer la valeur, par accès à l'ensemble de l'arbre paramétrique correspondant au constructeur de la classe. C'est donc la fonction sélecteur qui, à la fois, définit et rend accessible l'attribut.

Ces deux mécanismes permettent alors de créer complètement interactivement de nouvelles classes d'objet à partir d'un exemple de leurs instances. Ces classes sont alors directement utilisables dans le cadre du système ayant permis leur définition.

Nous avons proposé de distinguer deux modes d'exploitation de ces classes :

- le mode dynamique permet, sans quitter l'application de tester les nouvelles classes créées. Pour cela nous avons proposé un mécanisme de copie des arbres de construction (en fait des graphes acycliques) définissant la classe,
- le mode statique se base sur les techniques de génération de code développées dans le domaine de la compilation. Il permet de générer le code des classes dans un langage de programmation standard. Nous pouvons ainsi les intégrer dans le système qui a permis de les créer et en faire des classes natives de ce système. Nous pouvons également les utiliser dans le cadre de la définition de bibliothèques destinées à l'échange entre systèmes CAO hétérogènes.

L’approche que nous avons proposée permet donc, comme nous le souhaitions, de définir de nouvelles classes d’objets, possédant des attributs et des méthodes, à partir d’un exemple de leurs instances, de la même manière que la programmation par démonstration permet d’abstraire un programme à partir d’un exemple de son exécution.

Dans le chapitre suivant, nous décrivons l’application que nous avons réalisée dans le cadre de nos travaux. Nous montrerons notamment comment intégrer de nouvelles classes d’objets, définies avec la méthode proposée dans ce chapitre, dans l’application interactive, en étendant l’interface à l’aide des méthodes et outils définis au chapitre précédent.

Chapitre IV
TexAO

1 Introduction

Le but de notre travail est d'étudier la réalisation d'une application de conception technique permettant à un utilisateur de définir interactivement de nouvelles classes d'objets et de les intégrer à l'application de telle sorte qu'elles puissent être utilisées comme les classes natives du système. Pour cela, nous avons conçu deux outils : la boîte à outils du dialogue et le modèle de définition des classes.

- La boîte à outils du dialogue, étudiée au chapitre II, permet au concepteur d'application graphique interactive de créer le contrôleur de dialogue de la même manière qu'il crée la couche de présentation. Pour cela, nous avons proposé d'introduire de nouveaux constituants pour le contrôle de dialogue. Ceux-ci peuvent être créés dynamiquement, puis liés aux actions du noyau fonctionnel par l'intermédiaire de fonctions spécifiques.
- Un modèle pour la génération interactive de classes d'objets, à partir de la description d'une de leurs instances, a été proposé et décrit au chapitre III. L'approche proposée utilise un modèle paramétrique auquel nous avons associé à la fois une interface explicite et un ensemble de méthodes.

Dans ce chapitre, nous présentons l'application de conception technique que nous avons développée pour valider et illustrer les concepts proposés et étudiés aux chapitres précédents. Cette application permet à l'utilisateur de définir de nouvelles classes interactivement. Ces classes peuvent être intégrées dynamiquement au système, et leur comportement vis-à-vis de l'interface est le même que pour les classes natives du système.

La section suivante présente l'architecture générale du système TexAO que nous avons réalisé. Nous montrons notamment les liens qui existent entre les éléments du contrôleur de dialogue et le modèle (paramétrique) de l'application.

La troisième section est consacrée à la description de l'étude de cas, c'est-à-dire la description de la pièce que nous souhaitons représenter par une classe.

Enfin, la dernière section montre l'utilisation de notre système pour réaliser la pièce décrite dans l'étude de cas et pour la transformer en classe.

2 Architecture générale de l'application

Le système TexAO [Texier et Guittet 1999b] est basé sur le modèle d'architecture H^4. Il est composé de cinq macro-éléments : la présentation, l'adaptateur de présentation, le contrôleur de dialogue, l'adaptateur de noyau fonctionnel et le noyau fonctionnel. Dans cette section nous présentons ces macro-éléments et nous étudions les liens qui existent entre eux, tant au niveau de la définition de l'application qu'au niveau de son exécution.

2.1 Le noyau fonctionnel

Le noyau fonctionnel décrit le modèle de données relatif à l'application ainsi que les primitives agissant sur ce modèle. Dans le cadre de l'utilisation des concepts de géométrie paramétrée, il est nécessaire de définir deux modèles distincts : le modèle géométrique et le modèle paramétrique.

2.1.1 Le modèle géométrique

Pour la conception de TexAO, nous avons utilisé le modèle géométrique fourni dans l'environnement de développement CAS.CADE de Matra Datavision [CAS.CADE 2000]. CAS.CADE comprend un ensemble de composants logiciels réutilisables (gestion de données, outils de visualisation, modélisation géométrique, solveur de contraintes, etc…). Ces composants sont représentés sous la forme de classes C++.

Dans le cadre de la conception du système TexAO, nous avons principalement utilisé deux des outils de CAS.CADE, à savoir l'outil de modélisation géométrique et l'outil de visualisation de ces objets.

L'outil de modélisation géométrique est composé de classes permettant de créer des objets 2D et 3D. Les objets 2D peuvent être créés directement (cercle par centre et rayon) ou par contraintes (par exemple, cercle tangent à trois droites). CAS.CADE permet de construire des objets géométriques 3D par révolution, extrusion ou en effectuant des opérations booléennes. La bibliothèque géométrique sous-jacente est basée sur un modèle B-Rep, et offre des mécanismes pour parcourir la structure interne des objets.

2.1.2 Le modèle paramétrique

Le modèle paramétrique de TexAO est identique au modèle décrit au chapitre III (section 3.1.1). Les instances courantes sont des instances des classes d'objets géométriques de

CAS.CADE pour tout ce qui concerne la géométrie, et sont des éléments de type de base de C++ pour ce qui concerne les valeurs. Par exemple, l'instance courante de la classe *Cercle* (voir Figure IV-1) est de type *Geom2D_Circle* défini dans CAS.CADE alors que celle de la classe *Nombre* est de type *float* de C++.

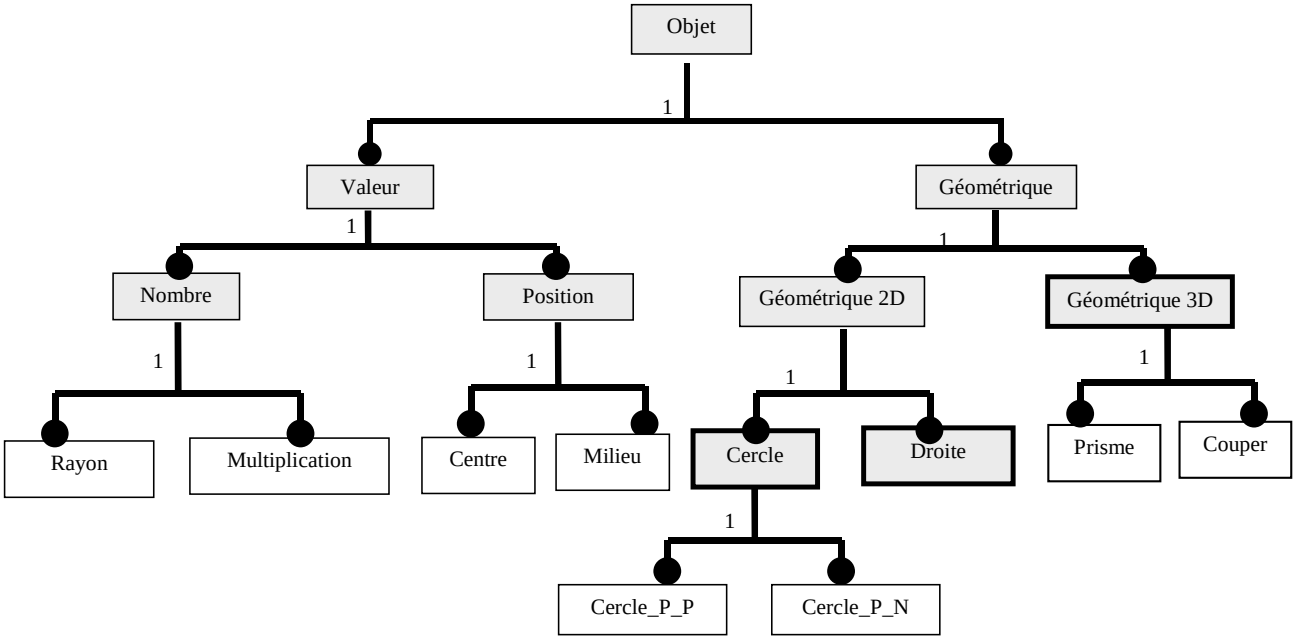


Figure IV-1: Arbre d'héritage des classes du modèle paramétrique

La Figure IV-1 représente une partie de l'arbre d'héritage des classes du modèle paramétrique de TexAO. Les rectangles grisés représentent les classes abstraites, les rectangles épais désignent les classes de base (voir chapitre III) et enfin, les rectangles fins sont les classes de construction des classes de base.

Remarque : Notre modèle est basé sur les concepts de décomposition des classes décrits dans [Aït-Ameur, et al. 1995 , ISO13584.20 1998]. Les attributs des classes sont réifiés, c'est-à-dire que leurs fonctions de calcul sont transformées en classes. Le paramètre du constructeur d'un attribut est un objet de la classe contenant l'attribut. Par exemple, la classe *Cercle* possède deux attributs, son centre et son rayon. Il existe donc une classe *Rayon* et une classe *Centre* possédant un objet cercle comme paramètre de construction. Les classes attributs héritent du type de données de l'attribut : la classe *Centre* hérite de *Position*, la classe Rayon hérite de *Nombre*.

Ce modèle permet à chacune de ses instances de conserver une trace de son processus de construction sous la forme d'une composition d'opérateurs. Ce processus de construction est ensuite utilisé par le générateur de classe.

2.2 Contrôleur de dialogue

Le contrôleur de dialogue de notre application a été conçu en utilisant la boîte à outils du dialogue que nous avons proposée au chapitre II. Sa définition correspond donc à l'instanciation des composants de la boîte à outils du dialogue. Dans cette section, nous allons mettre en avant les liens existants entre le noyau fonctionnel et le contrôleur de dialogue de l'application, en terme de conception de système interactif.

2.2.1 Définition des jetons

Comme nous l'avons vu au chapitre I, la spécification des jetons paramètres représente le modèle des objets manipulés par l'application au niveau du contrôleur de dialogue. Ainsi, pour définir la hiérarchie des jetons, il est nécessaire de connaître la hiérarchie des objets de l'application. Cette dernière correspond, dans le cas du système TexAO, à la hiérarchie des classes abstraites décrites dans le modèle paramétrique (voir Figure IV-1).

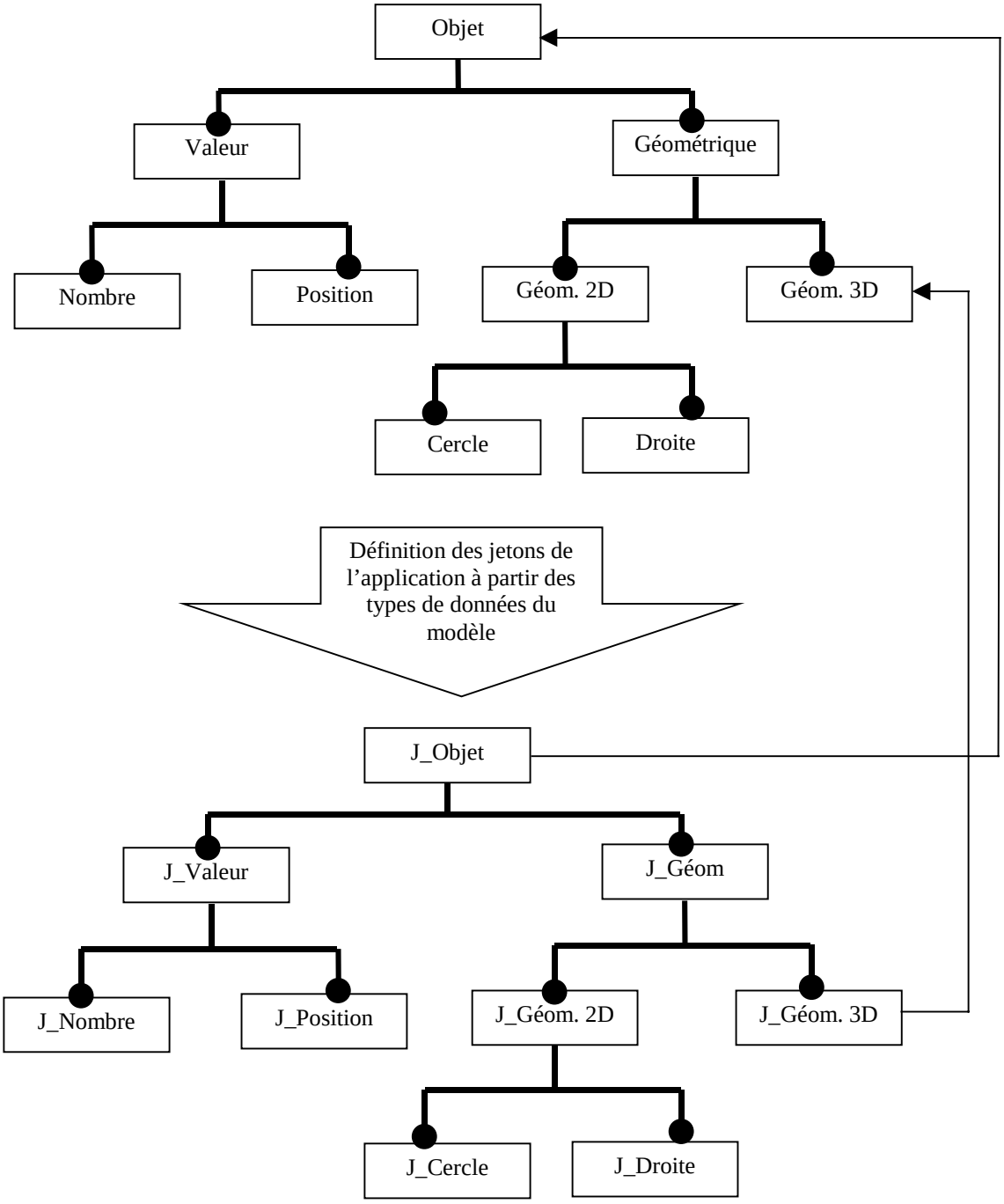


Figure IV-2: Définition des jetons du contrôleur de dialogue

La Figure IV-2 montre la définition des jetons du contrôleur de dialogue en fonction des types d'objets manipulés par l'application. Les flèches représentent le lien entre un jeton et la valeur qu'il contient (seule une partie de ces relations est exprimée sur la figure). Ainsi, un jeton de type *J_Cercle* contient la référence à un objet de type cercle, un jeton de type *J_Position* contient la référence à un objet de type position.

2.2.2 Définition des questionnaires

Les questionnaires de la boîte à outils du dialogue représentent des abstractions des services du noyau fonctionnel au niveau du contrôleur de dialogue. Ce sont des mécanismes typés de question réponse. Ils possèdent en entrée une liste de jetons et éventuellement un jeton en sortie, et sont chargés de l'appel des actions du noyau fonctionnel.

Les questionnaires représentent les actions du noyau fonctionnel, aussi appelées tâches système. Celles-ci peuvent être décomposées en tâches terminales ou en sous-tâches de production. Les tâches terminales ne produisent pas de jeton et ne peuvent pas être utilisées comme sous-tâche d'une autre tâche. Les sous-tâches de production sont utilisées pour représenter des tâches dont le résultat est nécessaire pendant l'exécution d'autres tâches. Pour définir les questionnaires du contrôleur de dialogue, le concepteur doit, en premier lieu, décider quelles actions correspondront à des tâches terminales et quelles sont celles qui correspondent à des sous-tâches de production. Ainsi, il est en mesure de spécifier l'éventuel jeton de sortie pour chaque questionnaire.

Les questionnaires peuvent être décrits en utilisant la spécification des actions du noyau fonctionnel. Dans le cadre de l'application TexAO, ces actions sont les constructeurs des classes du modèle (comme *Cercle_P_N* ou *Centre*). Chaque primitive est identifiée par un nom d'action. Elle possède une liste de paramètres d'entrée qui sont des objets du modèle paramétrique. Il est important de noter qu'elle ne possède rien en sortie ; en effet, les constructeurs ne possèdent pas de paramètre de sortie.

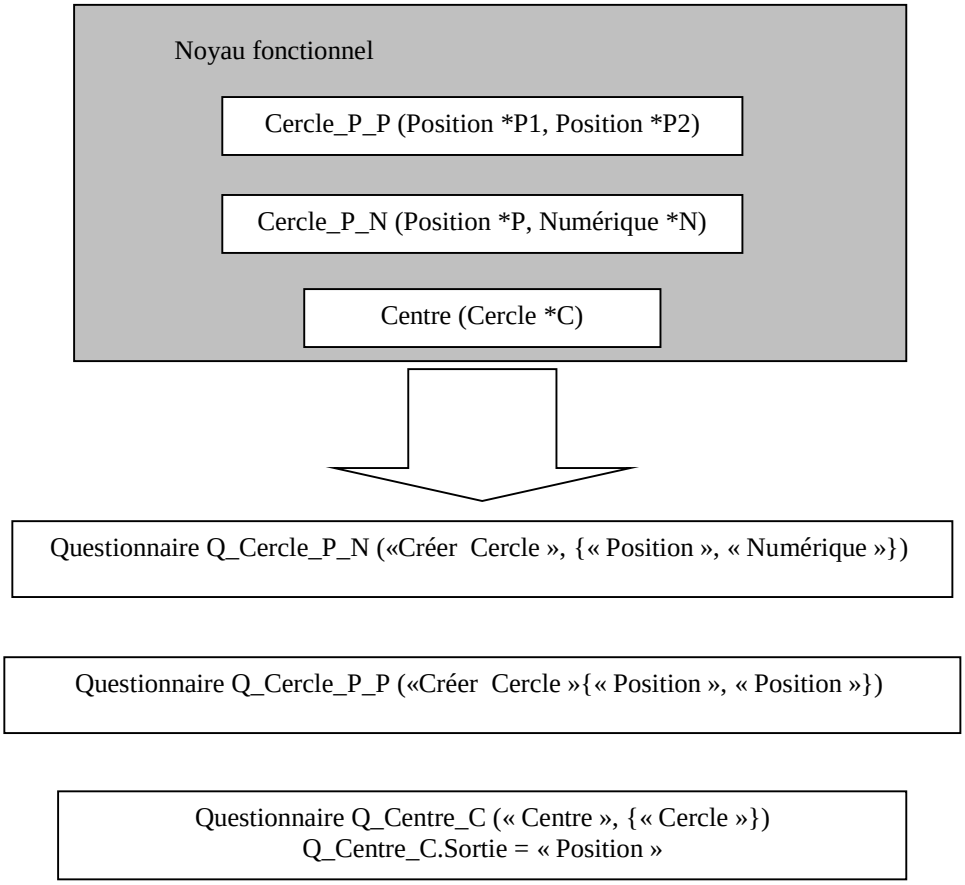


Figure IV-3: Conception des questionnaires

La Figure IV-3 montre la création de questionnaires à partir de la spécification des actions du noyau fonctionnel. Les constructeurs *Cercle_P_N* et *Cercle_P_P* sont considérés comme des tâches terminales du système; les questionnaires qui les représentent ne possèdent pas de jeton de sortie. L'action qui crée le centre d'un cercle (Centre) est représentée par un questionnaire dont le jeton de sortie est de type *J_Position*. Ce jeton contient un objet de type *centre* créé lors de la création de ce cercle.

2.2.2.1 Les actions répétitives

Les actions répétitives du noyau fonctionnel sont représentées par des questionnaires répétitifs au niveau du contrôleur de dialogue tels qu'ils sont décrits au chapitre II (section 4.2.2). Par exemple, le noyau fonctionnel de TexAO possède une classe *Contour* permettant de créer des contours fermés à partir d'un ensemble d'objets 2D.

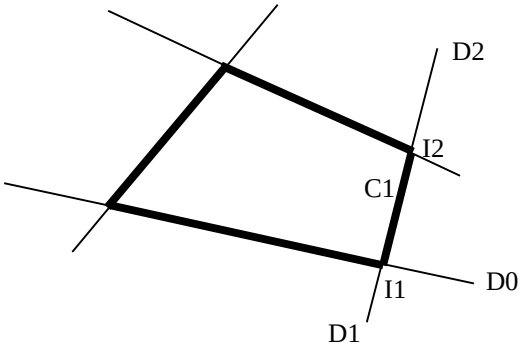


Figure IV-4: Création d'un contour fermé

La Figure IV-4 illustre la création d'un contour fermé à partir de quatre droites. Dans notre modèle paramétrique, le contour est défini comme un ensemble d'éléments de contours. L' élément de contour (C_i) est construit à partir d'un objet 2D support (O_i) et de deux positions (I_i et I_{i+1}) sur ce support représentant les extrémités de C_i. Par exemple, un segment est un élément de contour dont le support est une droite. Un élément de contour C_i peut aussi être créé avec trois objets 2D (O_{i-1}, O_i, O_{i+1}) s'intersectant. Le support est alors le deuxième objet (O_i) ; les extrémités (I_i et I_{i+1}) sont calculées à partir de l'intersection des objets O_{i-1} et O_i et des objets O_i et O_{i+1}. Sur la Figure IV-4, l'élément de contour *C1* est construit à partir des droites *D0*, *D1* et *D2*. Son support est la droite *D1*, ses extrémités *I1* et *I2*, sont issues de l'intersection de *D0* et *D1* et de *D1* et *D2*. Le contour est constitué de quatre éléments de contours construits de la même manière que *C1*.

D'un point de vue interactif, pour créer un tel contour, l'utilisateur sélectionne successivement les droites qui le composent. Du point de vue du dialogue, cette interaction correspond à l'utilisation d'un seul questionnaire répétitif. Sa fonction d'initialisation est chargée de récupérer les deux premiers objets 2D (O_0 et O_1) nécessaires à la création du premier élément de contour. La fonction d'itération utilise un objet 2D (O_{i+1}) pour créer un nouvel élément de contour (C_i) à l'aide deux derniers objets (O_{i-1} et O_i) mémorisés par le système. La fonction de fin construit le contour à partir des éléments de contour créés lors de la phase répétitive.

Le questionnaire répétitif est défini à partir de la description de la fonction du noyau fonctionnel qui est appelée plusieurs fois lors de l'interaction (ici, il s'agit du constructeur des éléments de contour). La nature répétitive du questionnaire vient du fait qu'il représente, au niveau du contrôleur de dialogue, une tâche d'arité non définie. Par exemple, un contour est créé à partir d'un nombre d'éléments de contour compris entre 3 et une infinité. Le questionnaire de création des contours s'écrit:

Questionnaire_répétitif Q_Contour (« Contour », {« J_Geom2D », « J_Geom2D », « J_Geom2D »})

Ce questionnaire est chargé d'appelé plusieurs fois, l'action du noyau fonctionnel qui crée un élément de contour à partir de trois entités géométriques 2D. Le questionnaire possède en entrée trois jetons de type J_Geom2D contenant des références sur des objets géométriques 2D du modèle paramétrique.

2.2.3 Définition des diagets et du moniteur

Le rôle des diagets est d'organiser les questionnaires en niveaux d'abstraction. Le système TexAO possède quatre niveaux de diagets :

- Le diaget d'information est chargé de contrôler l'appel des actions fournissant des informations sur les objets géométriques tel que le rayon d'un cercle,
- Le diaget d'expression permet la construction et l'évaluation de toutes les expressions grapho-numériques telles que la multiplication de deux nombres ou le calcul de la distance entre deux positions. Ce diaget est un diaget récursif, il est capable d'appeler plusieurs instances de la même tâche et d'utiliser un résultat produit par un autre diaget de même type.
- Le diaget de création regroupe l'ensemble des questionnaires représentant des tâches constructives telle que la création d'un cercle. Les questionnaires qu'il contient sont des tâches terminales, ils ne fournissent donc aucun jeton pour le diaget situé plus haut dans la hiérarchie.
- Le diaget de génération se charge de tous les questionnaires ayant attrait à la définition interactive de classes tels que le questionnaire de définition des paramètres ou celui de création d'attributs.

Le moniteur gère tous les diagets. Il récupère les jetons venant de la couche de présentation et les transmet aux diagets, dans l'ordre correspondant à l'énumération ci-dessus.

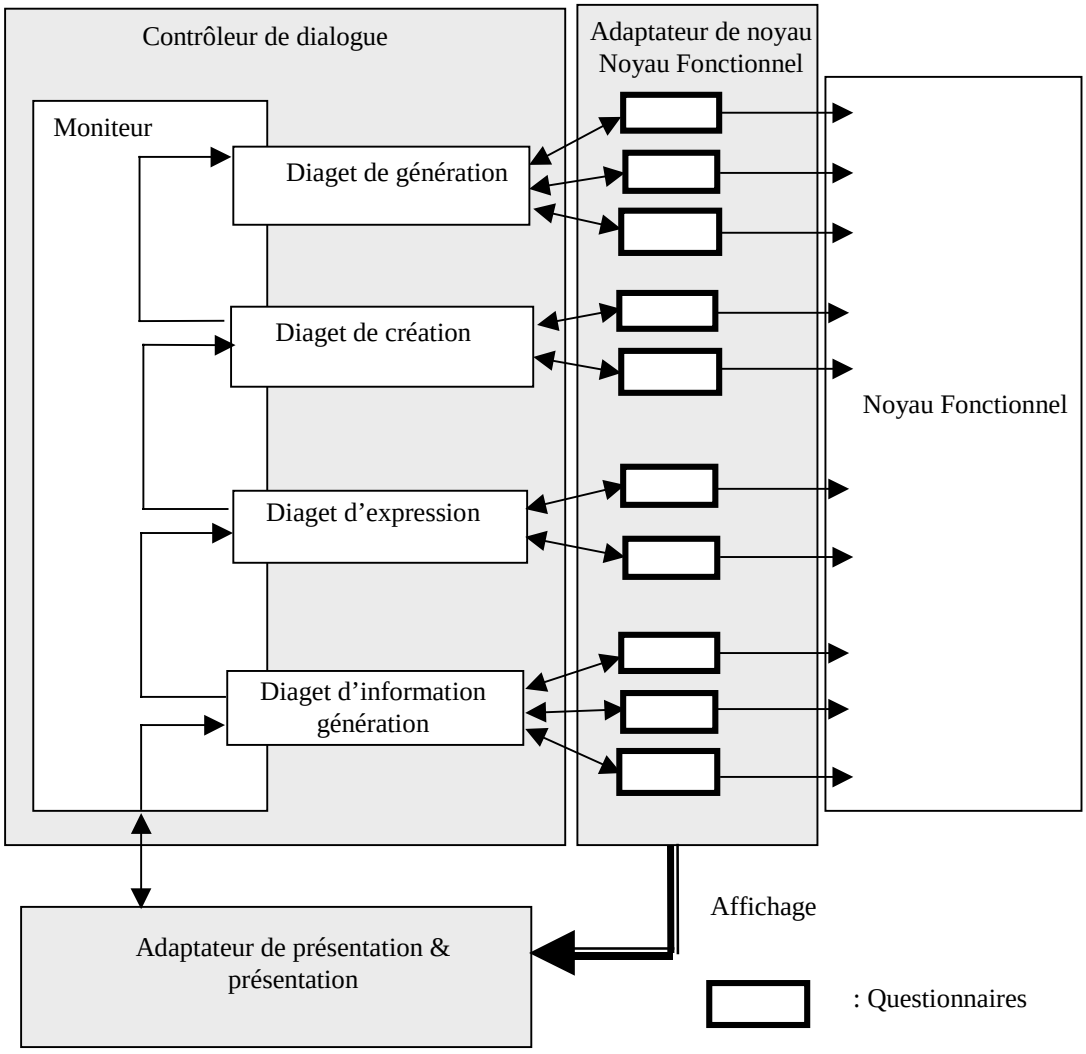


Figure IV-5: Organisation des éléments du contrôleur de dialogue

La Figure IV-5 montre les liens existants entre les éléments qui composent le contrôleur de dialogue et les autres composants de l'application tels qu'ils sont décrits dans le modèle d'architecture H^4.

2.3 Définition de la présentation

La présentation de TexAO utilise deux boîtes à outils distinctes :

- Les widgets des MFC (Microsoft Foundation Classes) [Procise 1999] sont utilisés pour définir les objets d'interaction (tels que les boutons, les menus, les cases de menu etc…) ainsi que pour le squelette général de l'application, c'est-à-dire l'ensemble des conteneurs de l'application.
- Le CAS.CADE AIS (Application Interactive Service) est utilisé pour réaliser l'affichage des objets géométriques. Cette bibliothèque fournit non seulement des mécanismes

d'affichage ayant en entrée des objets géométriques de CAS.CADE, mais aussi des mécanismes de sélection permettant de désigner les objets géométriques affichés. Il est important de noter que le CAS.CADE AIS autorise différents modes de sélections permettant d'accéder à la structure interne (faces, arêtes ou sommets) des objets affichés.

2.3.1 Générateur de présentation

L'adaptateur de présentation possède des primitives permettant de créer dynamiquement de nouveaux conteneurs dans des zones spécifiques du squelette de présentation, ainsi que des boutons dans ces conteneurs. La seule fonction de rappel définie pour ces boutons est appelée lorsque l'utilisateur appuie dessus. Il crée un jeton commande contenant le nom du bouton. Ce générateur de présentation est nécessaire pour la génération dynamique de nouvelles classes d'objets. Il permet au système de créer automatiquement des boutons correspondants à l'appel du constructeur de la nouvelle classe. Dans le cadre du développement de TexAO, ce générateur a aussi été utilisé pour générer l'ensemble des boutons permettant de créer des jetons « commande ». Pour organiser la présentation de manière plus lisible, nous avons regroupé tous les boutons créant les jetons « commande » d'un même diaget dans un même conteneur.

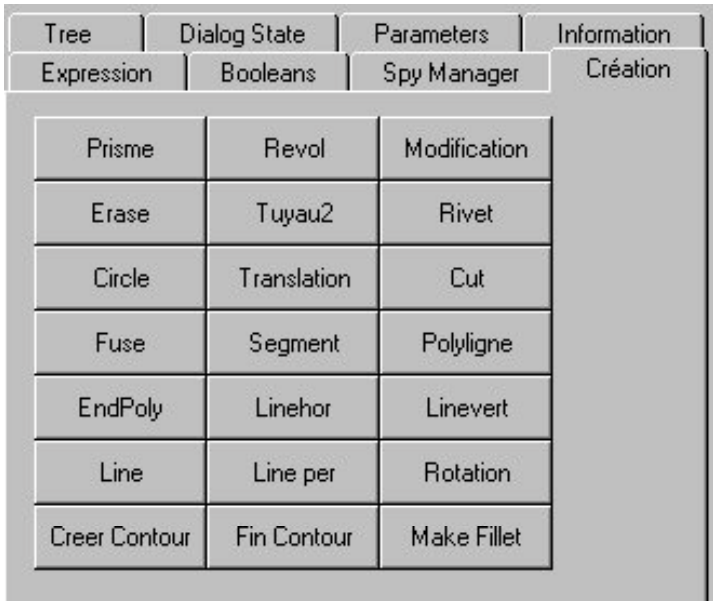


Figure IV-6: Présentation générée

La Figure IV-6 montre la partie générée de la présentation de TexAO. Les conteneurs créés par le générateur sont des onglets venant s'intégrer à une zone qui en contient déjà. Ainsi, les onglets « Spy Manager », « Creation », « Expression » et « Information », correspondants

respectivement aux diagets *Génération*, *Création*, *Expression* et *Information*, sont créés automatiquement. Les boutons visibles sur la Figure IV-6 correspondent aux jetons « commande » utilisés par le diaget de création.

2.3.2 Désignation et jetons

La désignation est une action primordiale pour les applications graphiques interactives, en particulier pour les actions de dessin. CAS.CADE offre des mécanismes de désignation simples permettant de récupérer l'objet graphique (*AIS_Shape*) le plus proche d'un pointé de l'utilisateur. Cet objet connaît l'objet géométrique qu'il représente ; il est donc possible de retrouver l'objet du modèle paramétrique désigné. Pour que la désignation puisse être contrôlée par le contrôleur de dialogue, il est nécessaire que l'adaptateur de présentation offre la possibilité au moniteur de définir quels sont, à chaque instant, les objets pouvant être désignés (i.e. correspondant à des jetons attendus par le contrôleur de dialogue) et ceux qui ne le sont pas.

Pour atteindre cet objectif, nous avons réalisé une sur-couche d'affichage de CAS.CADE qui permet d'ajouter des enregistrements (tags) aux objets graphiques, à la manière de Tk [Ousterhout 1994]. Lorsqu'un objet graphique est créé, un enregistrement lui est associé. Cet enregistrement contient le type du jeton représentant l'objet du modèle.

Par exemple, lorsqu'un cercle est créé et dessiné, l'objet graphique le représentant contient l'information qu'il s'agit d'un « *J_Cercle* ». Ainsi, lorsque cet objet graphique est désigné, la fonction de rappel du pointé graphique crée un jeton de type *J_Cercle* contenant l'adresse en base de données de l'objet *Cercle* désigné et le transmet au moniteur.

De plus, cette sur-couche d'affichage offre des primitives permettant de spécifier, à chaque instant, quels types d'objets peuvent être désigné. Ainsi, si le système est en attente d'un cercle pour réaliser une opération, l'utilisateur ne pourra désigner que des objets de type *Cercle*. Les autres objets affichés ne seront pas désignables, et ne passeront pas en sur-brillance lorsque le curseur passera dessus.

2.4 Synthèse

En conclusion de cette section sur l'architecture de TexAO, nous voyons que cette application utilise trois boîtes à outils distinctes :

- les MFC sont utilisées pour la partie présentation de l'application (autre que la visualisation du modèle),
- des instances d'éléments de notre boîte à outils du dialogue définissent le contrôleur de dialogue de l'application,
- la boîte à outils CAS.CADE offre ses services pour la définition du modèle géométrique et pour sa représentation.

La combinaison de plusieurs outils hétérogènes lors du développement d'une application peut amener plusieurs difficultés notamment :

- des problèmes de conversion de types peuvent apparaître lors du passage de valeurs entre des éléments provenant de bibliothèques différentes,
- des difficultés de choix d'implantation lorsque des services similaires sont disponibles dans plusieurs bibliothèques.

Dans le cadre du développement de TexAO, nous n'avons pas été confronté au problème de conversion de type puisque les trois bibliothèques utilisées ont été développées en C++. Par contre, les MFC et CAS.CADE offrent toutes les deux des structures de données génériques permettant d'instancier des listes, des tableaux et des tables de hachage. Or, comme nous l'avons vu au chapitre III (section 3), notre modèle pour la définition de classes utilise largement ce type de structure de données. Nous avons choisi d'utiliser celles fournies par les MFC parce qu'à notre sens, la documentation des MFC était plus claire que celle de CAS.CADE.

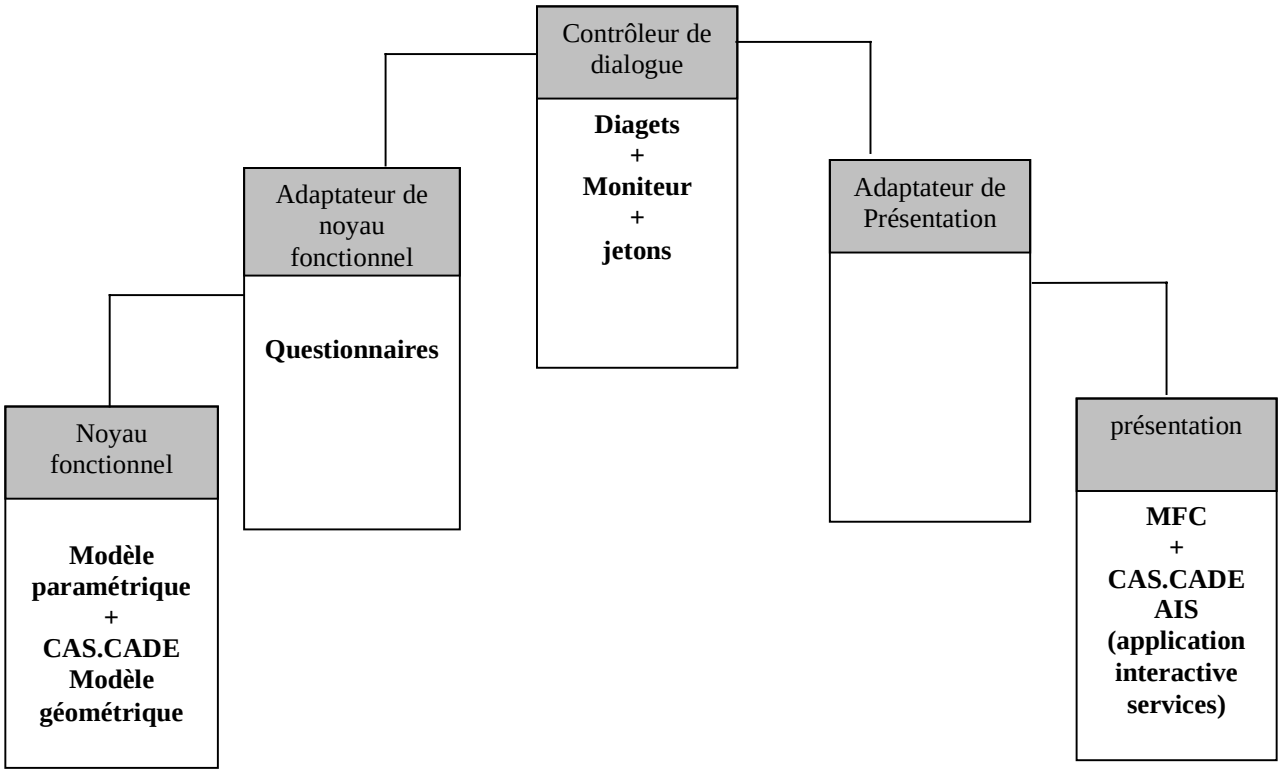


Figure IV-7: Architecture de TexAO

La Figure IV-7 montre une représentation des composants de TexAO. Cette figure précise notamment comment sont intégrées les différentes boîtes à outils utilisées pour la conception de ce système.

3 Exemple utilisé pour générer une classe

Nous présentons un exemple typique d'entité qu'un utilisateur pourrait avoir envie de considérer comme une classe native du système si l'activité de cet utilisateur consistait principalement à concevoir et représenter des moteurs à explosion.

La classe bielle dont l'utilisateur voudrait disposer serait constituée de :

- une méthode de construction permettant de créer une bielle, à partir de deux positions représentant les points de passage des axes de la bielle et d'un numérique représentant l'épaisseur de la bielle,
- une fonction (numérique) de calcul de l'attribut « longueur » qui représente la longueur totale de la bielle,
- une fonction (géométrique) permettant de calculer le centre de la bielle.

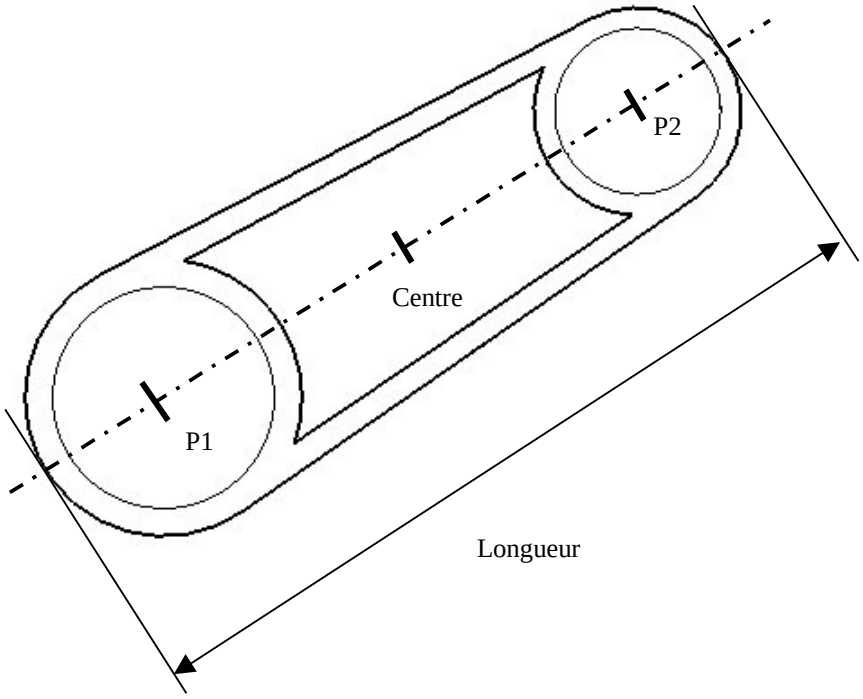


Figure IV-8: Dessin de définition de la bielle

La bielle (Figure IV-8) se présente sous la forme d'un prisme de hauteur H (paramètre du constructeur de type numérique). Les positions P1 et P2 sont les deux autres paramètres de construction. Les rayons des cercles servant à la construction sont définis par deux fonctions de calcul dont le paramètre est la distance entre les positions P1 et P2. La longueur de la bielle est le résultat de la somme de la distance entre P1 et P2, et les rayons des cercles extérieurs. Le centre de la bielle correspond au milieu du segment dont les extrémités sont P1 et P2.

4 Description du système TexAO

Cette section est consacrée à la description du système TexAO. Pour cela, nous nous basons sur l'étude de la réalisation interactive d'une classe permettant de créer des bielles telles que celle décrite dans la section précédente. Nous commençons par présenter brièvement l'interface du système. Puis, nous abordons la définition du constructeur, et nous décrivons les attributs de la nouvelle classe et la façon d'y accéder. Ensuite, nous illustrons son intégration dynamique au système. Enfin, nous montrons la génération du code de la classe et son intégration statique dans le système.

4.1 L’interface de TexAO

L’interface de TexAO possède des zones d’affichage permettant la visualisation des entités géométriques. Les commandes du système sont représentées par des boutons et des cases de menus. Cette interface a une zone de saisie de nombres. Elle possède également une zone permettant de visualiser l’arbre de construction des objets géométriques.

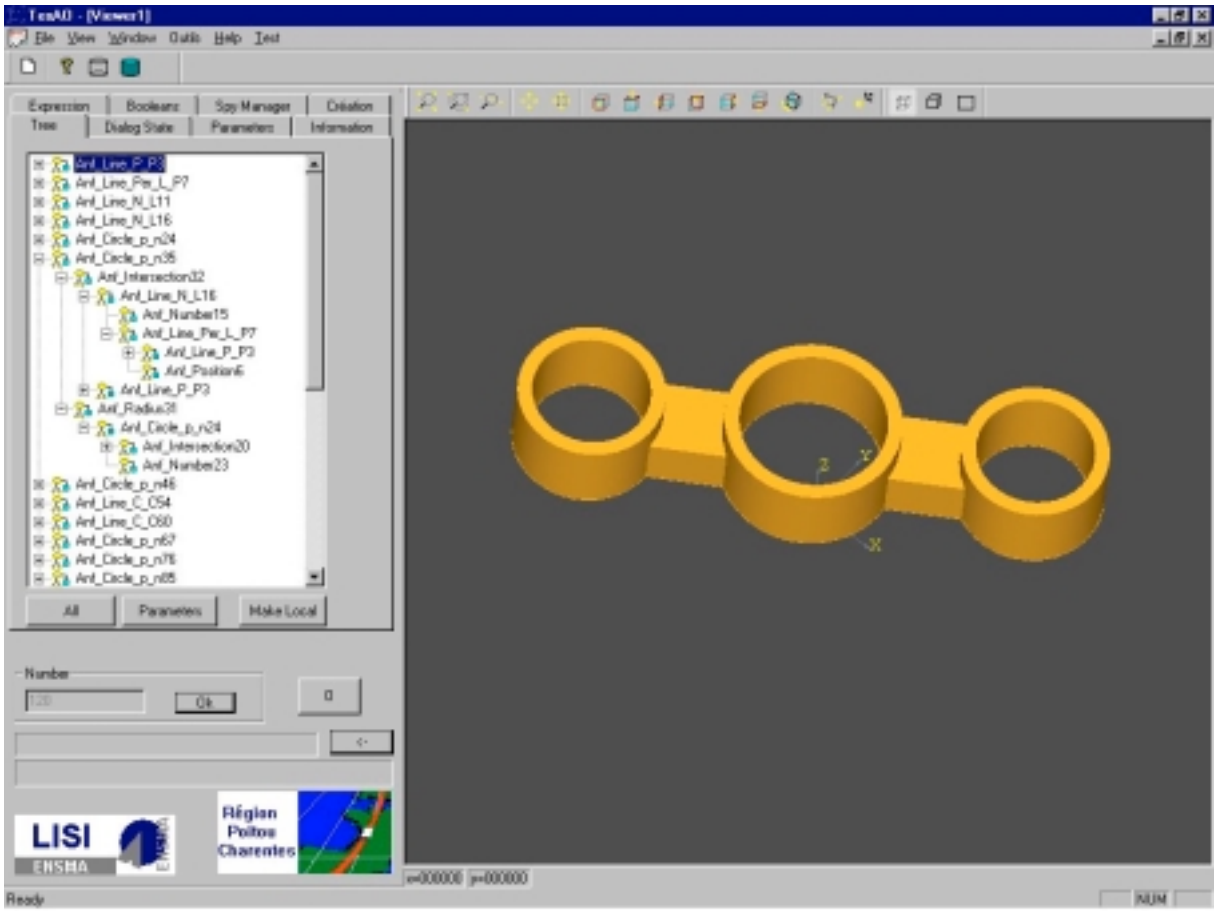


Figure IV-9: Zones de visualisation de TexAO

La Figure IV-9 montre l’interface de TexAO. L’accent est mis sur les zones de visualisation. Nous remarquons deux zones de visualisation du modèle. La première, à droite, montre les objets géométriques ainsi que leurs cotes en trois dimensions. La seconde, à gauche, est une représentation des arbres de construction de chaque objet géométrique créé par l’utilisateur. Il est intéressant de noter que ces deux zones peuvent être utilisées pour la sélection des objets. Cependant, étant donné que le système utilise les pointés de désignation pour lever les ambiguïtés de construction, la désignation des objets dans l’arbre ne peut être utilisée que pour réaliser des actions non constructives. Ainsi, nous avons dû restreindre l’utilisation des

désignations dans l'arbre à la définition de classes, de paramètres ou d'attributs, et à la désignation des objets dont l'utilisateur veut modifier la valeur.

Grâce aux méthodes d'analyse de l'état courant du dialogue disponible dans la boîte à outils de dialogue que nous avons proposée (cf. Chapitre II), TexAO est en mesure d'indiquer à l'utilisateur les types de jetons attendus ainsi que les types de données reçus à chaque instant. De plus, TexAO propose une fenêtre de visualisation de l'état du dialogue. Ainsi, l'utilisateur sait à quel endroit, de l'expression de l'arbre de ces buts/sous-buts, il se trouve.

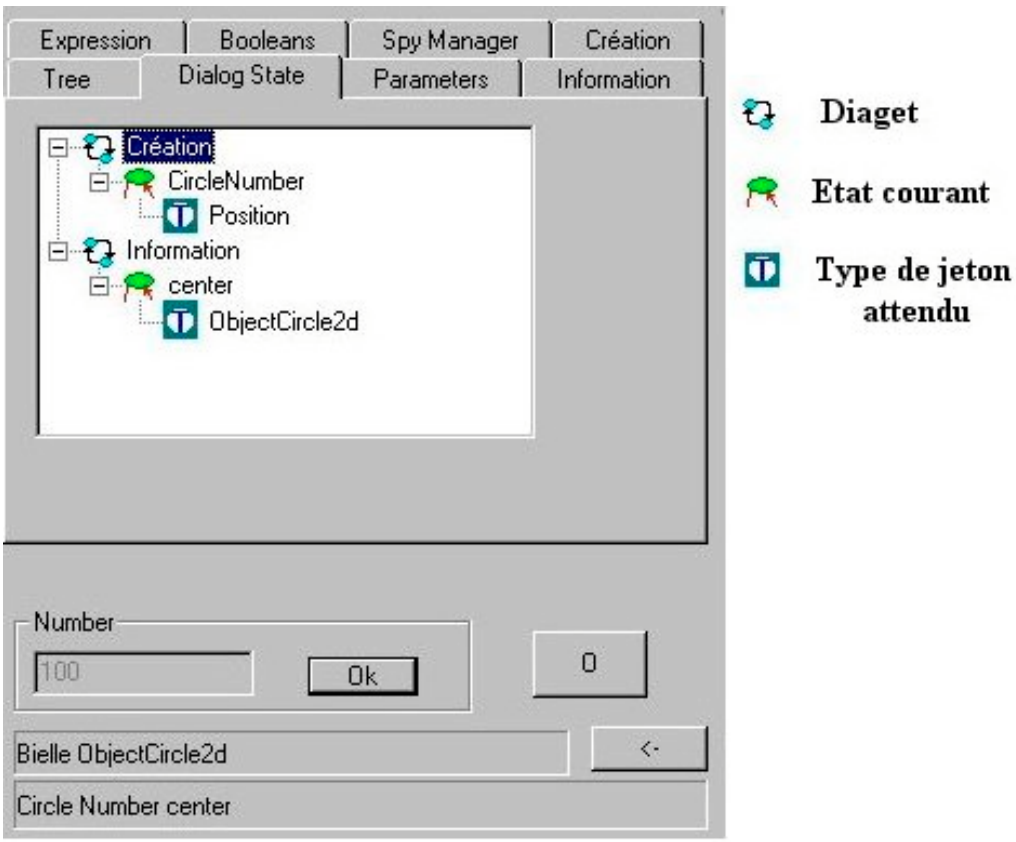


Figure IV-10: état du dialogue

La Figure IV-10 montre les zones de visualisation de l'état du dialogue. La zone « Dialog State » représente la hiérarchie des diagets. On remarque que le diaget de création est dans l'état « CircleNumber » ce qui signifie qu'il a consommé une commande « Circle » et un jeton de type « Number ». Ce diaget attend un jeton de type « Position » pour activer un questionnaire. Le diaget d'information est dans l'état « Center », il attend un jeton de Type « ObjectCircle2D » qui représente un objet cercle. Les deux zones d'affichage en bas de l'image montrent une représentation simplifiée de l'état du dialogue. La première ligne (la plus haute sur la figure) indique les types de données attendues par le système. La seconde indique les types de jetons reçus. Sur l'exemple, l'utilisateur a indiqué qu'il désirait créer un cercle, il a

fourni son rayon et il souhaite à présent calculer le centre d'un autre cercle afin de créer deux cercles concentriques.

4.2 Construction de l'exemple

La base de la définition interactive de classes d'objets est, comme en programmation par démonstration, la définition de l'instance qui permettra par la suite d'abstraire la classe. L'arbre de construction de l'exemple d'instance est utilisé par le générateur pour créer le constructeur de la classe.

Du point de vue de la définition du processus de construction d'une classe, TexAO se comporte essentiellement comme un système de géométrie paramétrique traditionnel, sauf qu'il permet d'associer des paramètres, et donc une signature à un tel processus.

La définition des paramètres du constructeur est un point primordial de la définition d'une classe puisqu'il transforme un simple historique de construction en un programme. Lors de l'instanciation de cette classe, l'utilisateur devra fournir des valeurs (paramètres effectifs). La définition des paramètres peut être accomplie à n'importe quel moment du processus de construction de la pièce. TexAO offre deux solutions pour définir les paramètres :

- soit à priori, en créant un nouvel objet défini comme paramètre; celui-ci sera par la suite considéré comme un paramètre tout au long de la construction,
- soit à posteriori en désignant dans l'arbre de construction des objets devant être considérés comme des paramètres lors de la génération de la classe.

La première solution nous semble devoir être utilisée plus fréquemment parce qu'en général l'utilisateur sait quels sont les paramètres de la pièce avant de la créer. En particulier, dans le cadre de la définition de classes de composants normalisés, ces paramètres sont décrits dans la norme.

Pour créer un nouveau paramètre, l'utilisateur active la commande « créer paramètre ». Il donne un nom et une valeur en créant un nouvel objet (définition à priori) ou en sélectionnant un objet existant (définition à posteriori). Cette valeur représente la valeur courante du paramètre. Un nouveau bouton représentant le paramètre est alors créé dans la présentation. Ce bouton est utilisé pour désigner le paramètre et l'utiliser dans une construction.

Note : Si le paramètre possède une valeur désignable graphiquement (comme une position), on peut également désigner cette valeur.

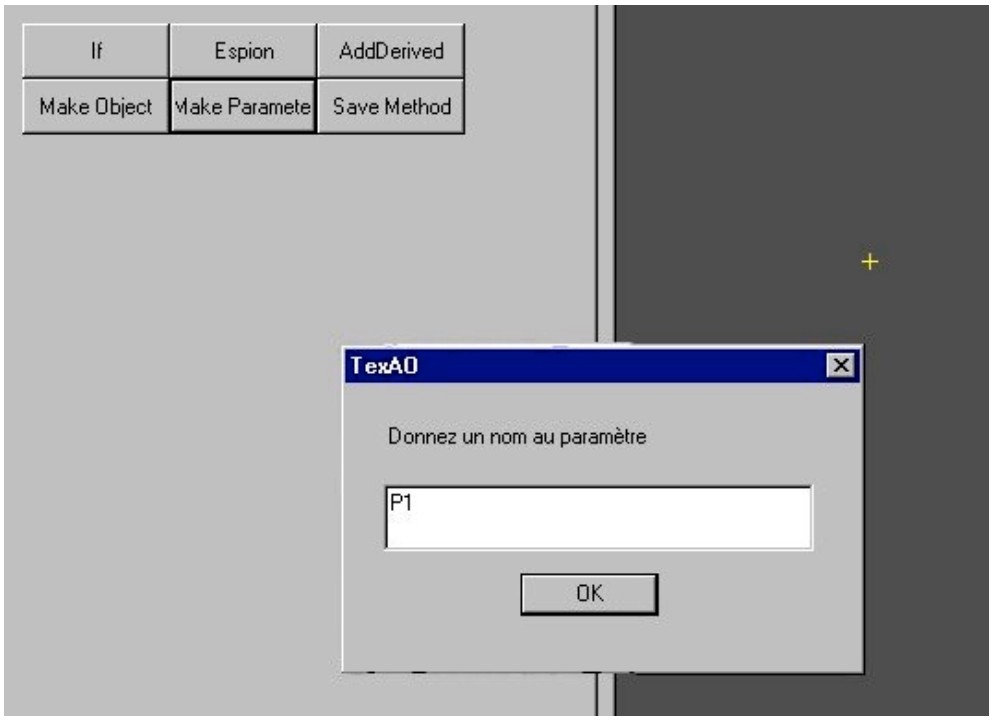


Figure IV-11: Création d'un paramètre

La Figure IV-11 montre la création d'un paramètre, l'utilisateur sélectionne la commande « Make Parameter », puis il donne une valeur (ici une position représentée par la croix). Le système ouvre une boîte de dialogue pour qu'il puisse saisir le nom. Le nom de chaque paramètre devant être unique (il correspond au nom du paramètre formel du constructeur), le système vérifie si un paramètre ne possède pas un nom identique avant de le créer.

Pour la définition de la classe bielle, l'utilisateur commence par définir les paramètres P1 et P2, qui sont les centres des axes de la bielle. Puis, à partir de ces deux paramètres, il réalise la construction de la bielle. Il définit le paramètre de hauteur (H) de la bielle juste avant de réaliser les extrusions (i.e. avant de s'en servir).

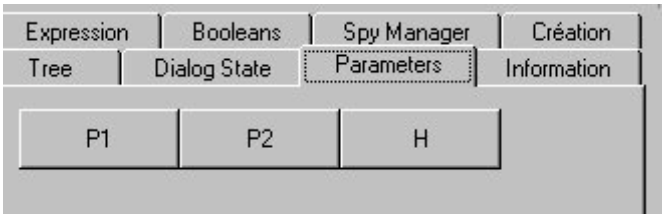


Figure IV-12: Boutons de sélection des paramètres

La Figure IV-12 montre les boutons générés après que l'utilisateur ait défini les paramètres de la bielle.

La bielle est construite à partir de l'extrusion de son contour extérieur. Cette extrusion est ensuite coupée par les deux cylindres intérieurs (résultant de l'extrusion des cercles intérieurs) et par l'extrusion du contour intérieur (voir Figure IV-13).

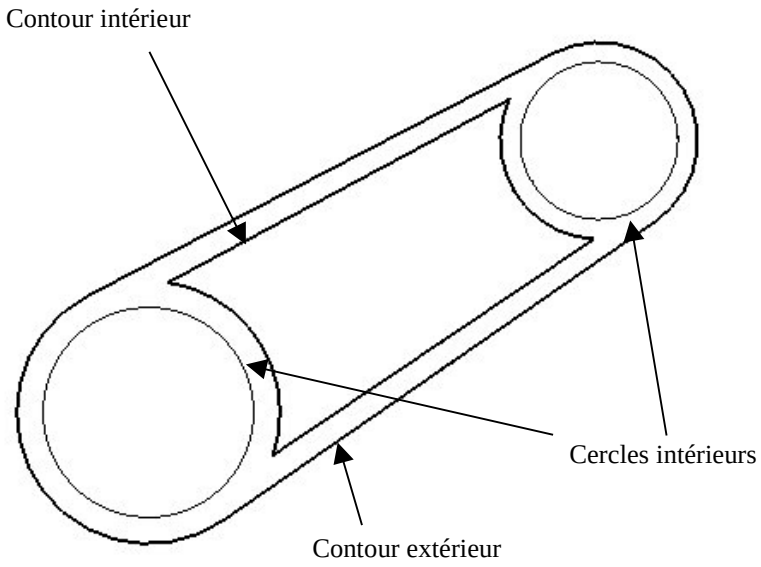


Figure IV-13: Contour à extruder lors de la création de la bielle

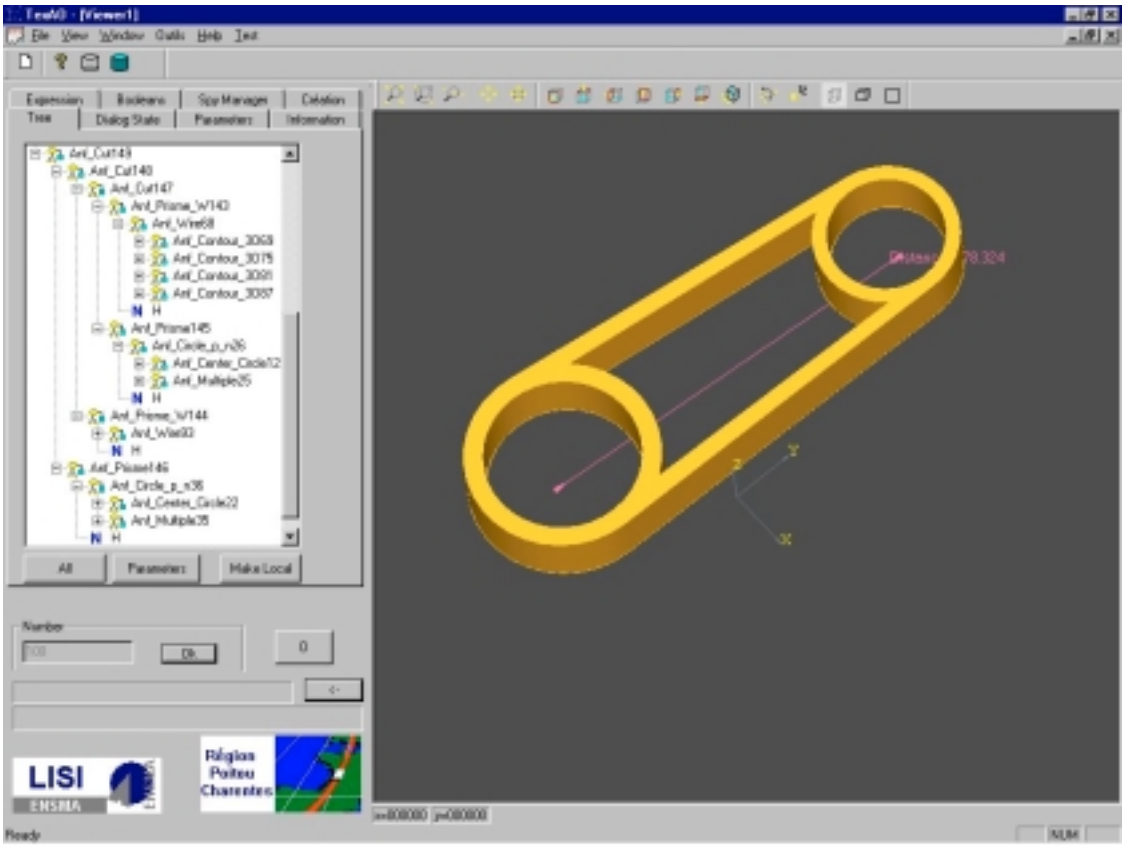


Figure IV-14: La bielle servant d'exemple à la création de la classe

La Figure IV-14 montre la bielle construite pour servir d'exemple à la création de la classe. La fenêtre de visualisation d'arbre permet à l'utilisateur de vérifier le processus de construction de la pièce qui servira pour définir le constructeur de la classe.

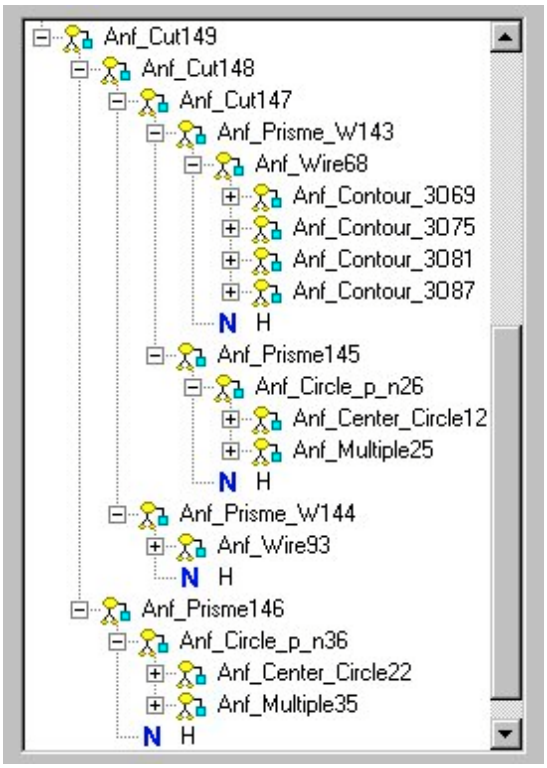


Figure IV-15: Arbre de construction de la bielle

Une partie de l'arbre de construction de la bielle est représenté sur la Figure IV-15. Les objets du modèle paramétrique sont représentés par les noms de leurs constructeurs (par exemple, *Anf_Circle_p_n* représente un cercle construit par son centre et son rayon) suivis du numéro de l'objet. Nous remarquons que les objets paramètres sont des feuilles de l'arbre. De plus, afin que l'utilisateur puisse facilement les repérer dans l'arbre de construction, leur icône est différente de celle des autres objets (icône *N* sur la figure).

4.3 Définition des attributs

Une originalité importante de TexAO par rapport à un système CAO de type paramétrique est que l'interface du système offre à l'utilisateur la possibilité d'associer des attributs à un objet. Ceci permet ensuite au générateur d'abstraire de nouvelles fonctions de calcul pour la classe. Par cette technique, comme dans les langages à objets, plusieurs méthodes pourront être définies dans le contexte d'une seule classe.

Pour ajouter un attribut à un objet, l'utilisateur sélectionne la commande « créer attribut » (« Add derived » sur les figures). Ensuite, il désigne l'entité représentant la valeur courante de l'attribut. Le système demande alors de saisir le nom de l'attribut. L'arbre de construction de l'entité représentant la valeur de l'attribut permet alors au générateur d'abstraire la fonction de

calcul de cette valeur. Lorsque l'utilisateur définit un nouvel attribut, l'objet représentant l'attribut et son nom sont enregistrés dans la liste des attributs de l'exemple et constituent donc autant de méthodes qui seront donc à la disposition de l'utilisateur lorsque la classe sera intégrée dans le système.

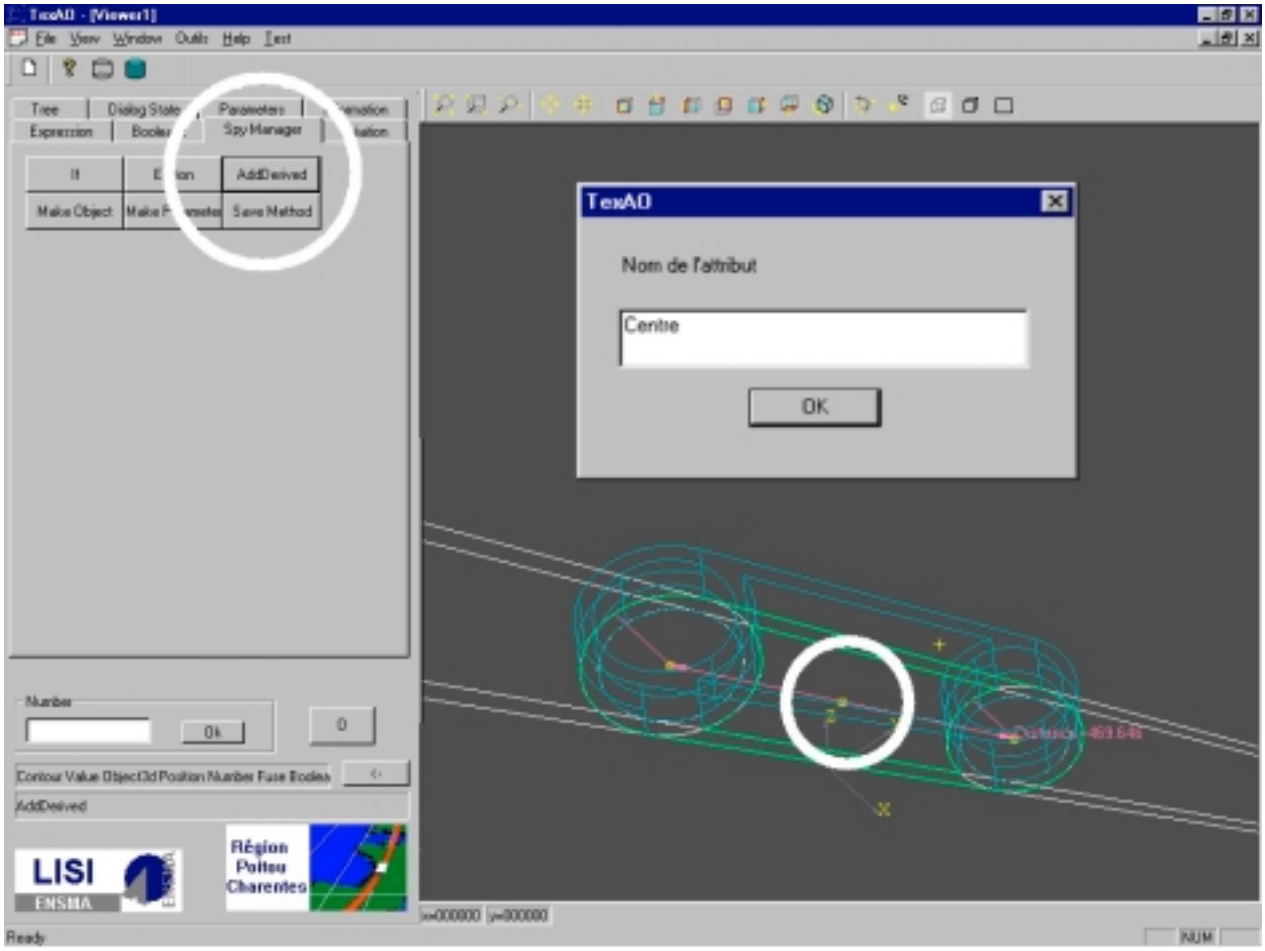


Figure IV-16: Définition de l'attribut Centre

Ainsi, dans notre exemple, la classe bielle possède deux attributs qui sont le centre de la bielle et sa longueur.

Le centre de la bielle est déterminé par le calcul du milieu des points paramètres *P1* et *P2*. Pour ajouter cet attribut, l'utilisateur sélectionne la commande « créer attributs », puis la commande « milieu » et enfin, il fournit les deux points nécessaires, soit en les désignant sur le dessin ou dans l'arbre, soit en utilisant les commandes « P1 » et « P2 ». A ce moment, le système lui demande de saisir le nom de l'attribut. Il vérifie alors que ce nom n'existe pas déjà et ajoute l'attribut à la liste des attributs de l'objet représentant la bielle.

Nous remarquons, sur la Figure IV-16, que la position déterminant le centre de la bielle est affichée immédiatement après son calcul. Une boîte de dialogue demande alors à l'utilisateur de fournir le nom de l'attribut.

La longueur de la bielle est calculée par la somme des rayons des deux cercles extérieurs et de la distance entre leurs centres. Cet attribut est ajouté à la bielle de la même manière que l'attribut centre, en utilisant la calculette grapho-numérique pour introduire l'expression de calcul: *Somme (Rayon (C1) , Somme (Rayon (C2), Distance (P1,P2)))*

4.4 Intégration dynamique

La classe à introduire dans le système est maintenant complètement définie. L'intégration dynamique consiste à insérer, dans l'application, cette nouvelle classe, dont l'utilisateur vient de décrire une instance, sans avoir à relancer l'application. La classe peut alors être instanciée de la même manière que les classes natives du système. Cela permet à l'utilisateur de la tester et éventuellement de la modifier en changeant l'exemple afin d'apporter des corrections.

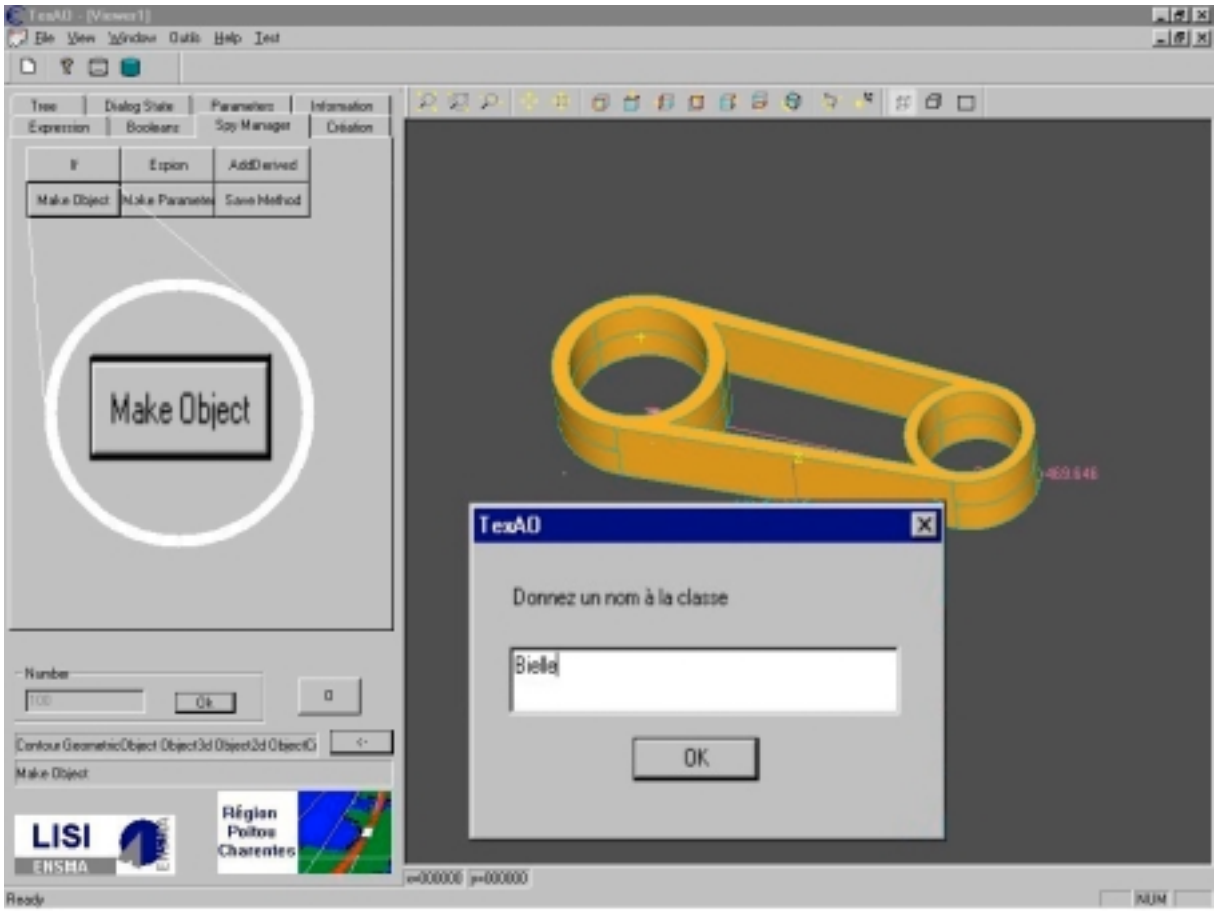


Figure IV-17: Création de la classe "Bielle"

La classe « Bielle » est créée à partir de l'exemple de bielle construit interactivement. L'utilisateur sélectionne la commande « Make Object » puis désigne l'objet représentant la bielle. Le système demande alors de saisir le nom de la classe.

4.4.1 Principe d'abstraction

L'intégration dynamique d'une classe consiste, comme nous l'avons vu au chapitre III (section 4.2), à créer une copie non évalués (sans lien vers le modèle géométrique) des arbres de construction (en fait des graphes orientés acycliques « DAG ») qui composent l'exemple. Ces DAG « non évalués » sont conservés dans une base de données spécifique qui associe à chaque DAG le nom de la classe à laquelle il correspond. Ainsi, le constructeur de la bielle est référencé par le nom « bielle », et le DAG de calcul de la longueur de la bielle est référencé par « longueur bielle ». Ces noms sont utilisés par le système pour modifier l'interface afin que la nouvelle classe soit accessible à l'utilisateur.

La classe bielle est ajoutée de manière implicite (voir chapitre III) à l'arbre d'héritage du modèle paramétrique. Elle hérite donc de la classe *Géométrique 3D*.

Pour conserver l'homogénéité avec le modèle paramétrique de TexAO, chaque attribut entraîne la définition d'une nouvelle classe. Ainsi, la classe « Longueur bielle » est générée et ajoutée au système par le générateur. Elle hérite, de manière implicite, de la classe de base *nombre*, et son constructeur a pour paramètre une instance de la classe bielle (Figure IV-18).

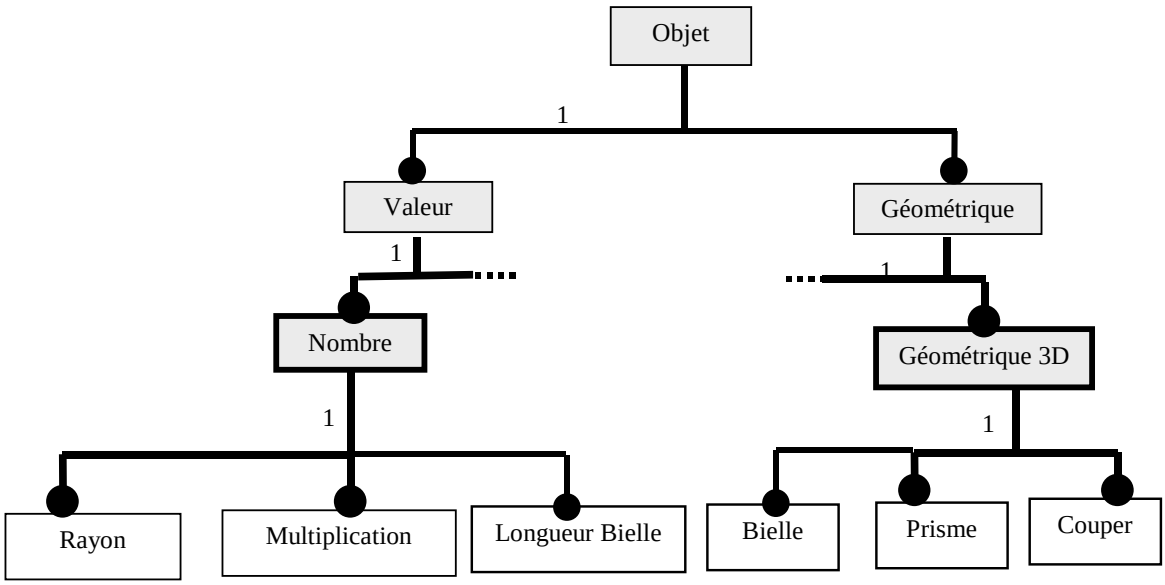


Figure IV-18: Insertion de la bielle dans la hiérarchie des classes

4.4.2 Intégration au système

Pour que l'utilisateur puisse instancier une classe intégrée dynamiquement, il est nécessaire de modifier l'interface (dialogue + présentation). Le système génère alors les différents éléments de dialogue pour manipuler la nouvelle classe. Ceux-ci comportent :

- un nouveau type de jetons afin de représenter les instances de la classe au niveau du contrôleur de dialogue. Il sera utilisé lors de toutes les désignations d'objets qui sont des instances de la classe, et il est référencé par les questionnaires chargés de calculer les attributs de l'objet,
- le questionnaire capable d'appeler le constructeur de la classe. Son nom est le même que celui de la classe,
- les questionnaires responsables de l'appel des fonctions de calcul des attributs dérivés.

La figure ci-dessous (Figure IV-19) montre, dans un pseudo-langage, le code généré.

Class J_Bielle is new Jeton_Paramètre(« Bielle », « Paramètre ») ;

Questionnaire Q_Bielle («Bielle »{« Position », « Position », «Numérique »})

Questionnaire Q_Bielle_Longueur («Longueur », {« Bielle »}, « Numérique »)

Questionnaire Q_Bielle_Centre («Centre», {« Bielle »}, « Position »)

Figure IV-19 : Génération des éléments du contrôleur de dialogue

Pour contrôler l'appel des constructeurs des trois classes générées lors de la définition de la bielle (un constructeur de bielle et les deux constructeurs correspondants aux fonctions de calcul des attributs dérivés), le générateur crée donc trois questionnaires et définit un nouveau type de jetons (Figure IV-19).

Le questionnaire chargé de la construction de l'objet est ajouté au diaget de « création » qui regroupe les tâches terminales du système. Les questionnaires responsables du calcul des attributs sont ajoutés au diaget d' « information ». L'addition de ces nouveaux questionnaires entraîne donc, en général, la création de nouveaux boutons dans la couche de présentation, sauf si leur nom existait déjà (exemple : « *centre* » déjà disponible pour les objets de type *Cercle*). Ces commandes permettent ensuite à l'utilisateur d'appeler les nouvelles fonctions (constructeurs et calcul d'attributs dérivés) créés à partir de la description de la classe.

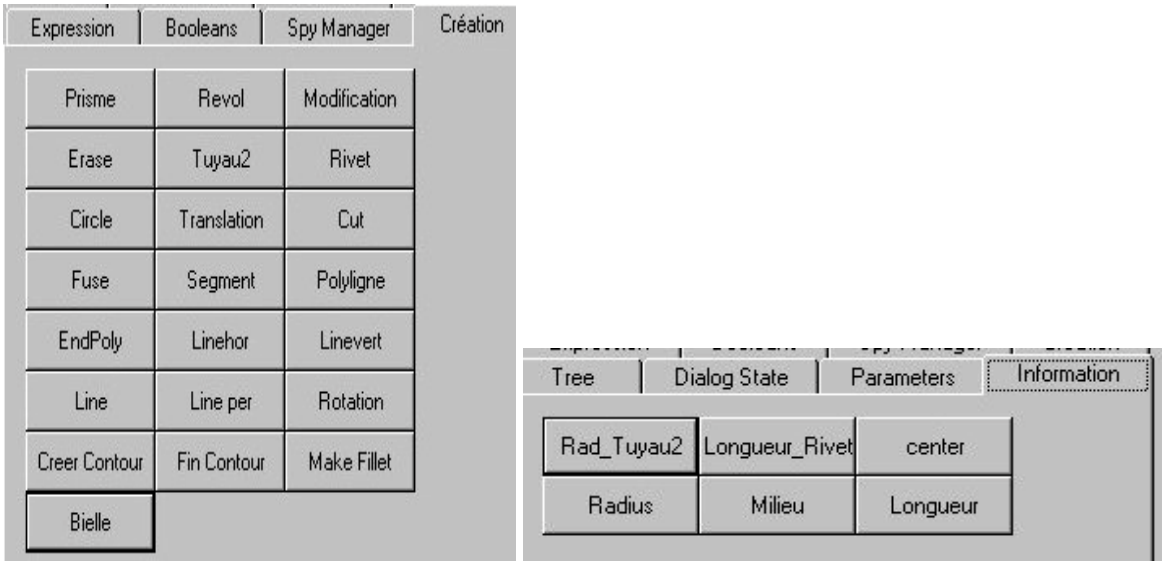


Figure IV-20: Modification des menus

La Figure IV-20 montre les boutons créés par le générateur de présentation à partir des questionnaires générés au moment de la création de la classe bielle. Le bouton « Bielle » permet de créer une instance de bielle ; les boutons « Centre » et « Longueur » sont utilisés pour interroger les attributs d'une bielle.

4.4.3 Instanciation des classes

Lorsque l'utilisateur veut créer une instance d'une nouvelle classe, il indique la classe en utilisant le bouton représentant la classe dans la présentation. Il fournit au constructeur les valeurs des paramètres effectifs.

Pour créer une instance, le système retrouve l'arbre de construction dans la base de données des arbres avec le nom de la commande. Il copie cet arbre et substitue les objets représentant les paramètres formels par les nouveaux objets contenus dans les jetons et représentants les paramètres effectifs. Le nouvel arbre créé est alors réévalué pour créer une nouvelle instance. Il est important de noter que tous les attributs de l'objet sont alors copiés dans l'instance et que leur valeur est recalculée.

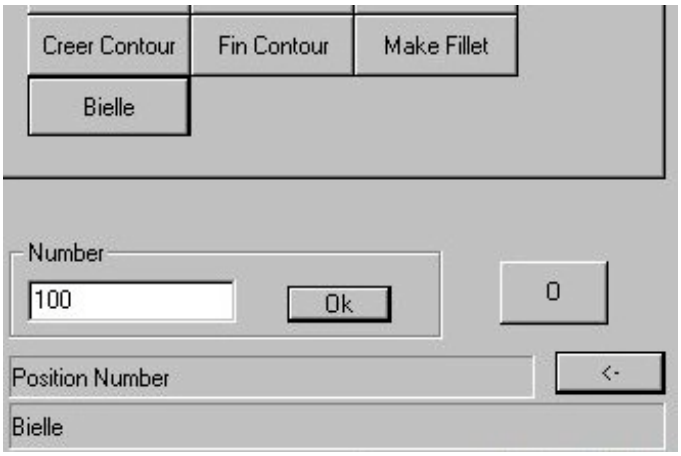


Figure IV-21: Instanciation d'une bielle

Lorsque l'utilisateur sélectionne la commande « Bielle », le système attend qu'il fournisse des valeurs pour les paramètres, il attend donc deux jetons de type position et un jeton de type nombre (Figure IV-21). Dès que l'utilisateur lui a fourni ces données, soit directement, soit par un calcul, le système appelle le constructeur de la classe bielle et une bielle est affichée (Figure IV-22).

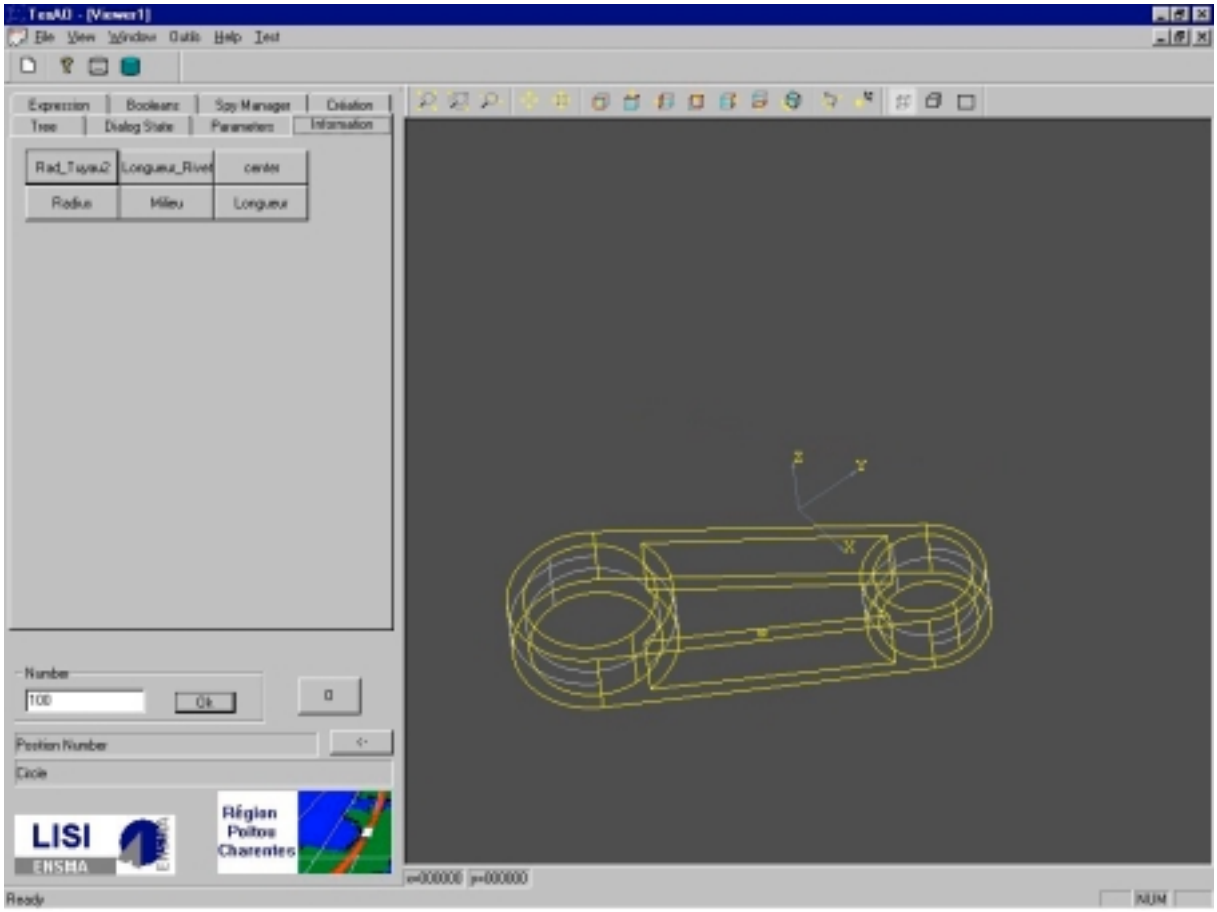


Figure IV-22: Une instance de la classe Bielle

Pour calculer la valeur d'un attribut, l'utilisateur sélectionne la commande qui correspond au nom de l'attribut et l'objet pour lequel il veut que la valeur de l'attribut soit calculé. Un questionnaire spécifique est alors utilisé pour extraire la valeur de l'attribut de la liste des attributs contenus dans l'objet à partir du nom de l'attribut. Ce questionnaire crée un jeton dont la nature correspond au type de l'attribut et qui contient l'entité représentant la valeur de l'attribut. Celui-ci peut, à son tour, être utilisé pour établir une relation entre une bielle et un autre élément du modèle en cours, comme si la bielle était un objet natif du système, et disposait d'opérateurs permettant de la mettre en relation avec d'autres objets.

4.5 Intégration statique

L'intégration dynamique permet d'obtenir toutes les fonctionnalités voulues par l'utilisateur final sauf que l'exécution en est relativement lente (code interprété). Cet inconvénient est supprimé par l'intégration statique.

L'intégration statique consiste à générer le code des classes à partir de leurs arbres de construction et à intégrer ces nouvelles classes au code de l'application de la même manière que les classes natives. Pour cela, la description statique du contrôleur de dialogue est modifiée. Le code des questionnaires nécessaires à la manipulation des nouvelles classes ainsi que les modifications nécessaires à leur intégration dans le contrôleur de dialogue sont générés. Enfin, le générateur de code produit la spécification statique de jetons représentant les objets de la nouvelle classe.

L'application doit ensuite être recompilée. Cette phase qui serait difficile à réaliser par un utilisateur non informaticien peut-être automatisée avec la génération de fichiers de compilation et d'édition de liens (fichier « make file» de C++).

Les dualité statique/dynamique des classes générées par TexAO peut être comparée à la dualité langages de classes et langages de prototypes des langages orientés objets. La création d'un objet à partir d'une classe statique se fait par instanciation de la manière classique des langages de classes. Par contre, la création d'un objet provenant d'une classe dynamique se fait par copie de l'objet représentant la classe en changeant la valeur des paramètres de création (cf. Chapitre III section 4.2) ce qui correspond à la notion de prototypage dans les langages orientés objets interprétés.

5 Conclusion

Au cours de ce chapitre, nous avons présenté l'application TexAO. Cette application de conception technique offre la possibilité, tout à fait originale à notre connaissance, de définir interactivement de nouvelles classes d'objets et ainsi, d'enrichir ou de spécialiser l'application initiale.

Cette application utilise du point de vue géométrique, un modèle paramétrique fonctionnel, mais nous avons étendu celui-ci à l'aide de concepts issus de la programmation par démonstration, et tout particulièrement la distinction entre les paramètres, les variables et les constantes. De plus, et à la différence des systèmes paramétriques habituels, TexAO ne permet pas seulement de construire une famille (paramétrique) d'objets, il permet également d'associer à cette famille différents attributs, ainsi que des méthodes permettant de les évaluer. Ces attributs, à leur tour, permettent alors d'établir des relations entre les objets appartenant à des classes normalement définies et d'autres objets pré-existants. Enfin, pour permettre d'intégrer dynamiquement à l'interface les nouvelles classes définies par l'utilisateur, le système TexAO fait appel à la nouvelle boîte à outils du dialogue que nous avons proposée au chapitre II. Elle permet non seulement de définir la partie statique du contrôleur de dialogue mais aussi de l'enrichir dynamiquement afin que le comportement des classes définies interactivement soit semblable à celui des classes natives du système.

Enfin, ce chapitre a permis de mettre en évidence, grâce à un exemple, la simplicité d'utilisation de ce système pour définir de nouvelles classes d'objets. L'utilisateur crée des pièces paramétrées de la même manière qu'avec les autres systèmes de CAO paramétriques. Il définit les éléments caractéristiques de la classe (paramètres, attributs,…) au moment qui lui semble le plus opportun et dispose alors de nouvelles classes d'objets qui se comportent comme les objets natifs du système.

Conclusion

L'utilisation des systèmes CAO est souvent complexe, ils possèdent en effet de nombreuses fonctionnalités permettant de créer toutes sortes de formes géométriques et de leur appliquer toutes sortes de méthodes. La difficulté de leur utilisation réside dans le fait que la granularité des actions qu'ils proposent est souvent très faible. En général, ils ne s'adaptent pas directement aux objets manipulés par l'entreprise. Pour faciliter leur utilisation dans un cadre particulier, et donc augmenter leur productivité, il est nécessaire de les adapter au besoin spécifique de leurs utilisateurs, c'est-à-dire leur ajouter de nouvelles fonctionnalités permettant de créer et manipuler les objets spécifiques de l'activité d'une entreprise ou d'un processus. Cette adaptation correspond à une spécialisation du système CAO. Une telle spécialisation peut consister, par exemple, à transformer un système CAO orienté « bâtiment » en un système de conception de cuisine, ou à transformer un système à vocation mécanique en système de conception de mécanismes articulés.

Avec les méthodes de programmation traditionnelles, la spécialisation d'un système CAO nécessite un gros effort de développement. Chaque élément spécifique du domaine visé est ajouté en écrivant le code nécessaire à sa description et à son intégration. L'interface du système est ensuite modifiée pour permettre la manipulation des nouveaux éléments. Ce travail ne peut pas être réalisé par un spécialiste du domaine visé, mais par un informaticien qui n'est pas spécialiste du domaine d'application.

Le but de notre travail est de montrer qu'il est possible à un non informaticien de spécialiser lui-même son outil de conception technique sans programmation explicite. Ainsi, les spécialistes d'un domaine particulier auront la possibilité d'intégrer facilement les éléments spécifiques de leur domaine d'activité dans leur système CAO.

Les applications graphiques interactives, tels que les systèmes CAO, sont constituées de deux grands composants. L'interface prend en charge la présentation, responsable de la gestion et de l'organisation des entrées sorties de l'application, et le contrôle du dialogue chargé de l'appel des fonctionnalités de l'application en fonction des interactions de l'utilisateur. Le noyau fonctionnel regroupe l'ensemble des primitives spécifiques de l'application. Par conséquent, la spécialisation d'une application nécessite, d'une part, l'ajout au noyau fonctionnel des actions de création et de manipulation des objets spécifiques du domaine visé, et, d'autre part, la

modification de l'interface pour permettre à l'utilisateur d'accéder à ces fonctionnalités. Dans l'approche que nous proposons, ces modifications de l'application doivent pouvoir être réalisées sans programmation explicite par un spécialiste du domaine d'application visé.

L'attente de notre objectif nécessitait donc de mener à bien deux types de travaux :

1. l'étude et la réalisation d'outils de définition de l'interface permettant d'intégrer de nouvelles fonctionnalités définies de manière interactive (i.e. sans programmation explicite),
2. la conception et l'utilisation d'un modèle pour la définition interactive (i.e. sans programmation explicite) de nouvelles classes d'objets spécifiques d'un domaine d'application.

Dans la première partie de ce travail (chapitre I), nous avons analysé les différentes méthodes et les différents outils existants dans nos deux axes d'étude.

Dans le domaine de la définition d'interface des applications interactives, deux types d'approches ont été proposées :

- la méthode ascendante utilise une description de la couche de présentation à partir de briques de base de la présentation (« boîte à outils »). Ces briques peuvent être instanciées dynamiquement. Elles sont assez simple d'utilisation, mais elles n'offrent qu'un mécanisme trop simple de contrôle de l'application qui ne peut être utilisé seul dans le cadre de la création d'un système CAO.
- la méthode descendante utilise une description du contrôleur de dialogue pour générer l'application. Elle peut être utilisée pour créer l'interface d'un système CAO mais supporte mal les modifications dynamiques de l'interface.

Cette dualité nous amènera à proposer une « boîte à outils du dialogue » basée sur la réification d'éléments de contrôle. Ainsi, cette boîte à outils du dialogue bénéficiera du caractère dynamique et de la simplicité d'utilisation des boîtes à outils pour décrire explicitement le contrôleur de dialogue des systèmes CAO.

Pour permettre la définition interactive et sans programmation explicite de nouvelles classes d'objets au sein d'un système interactif, deux formes de programmation interactive ont été proposées :

- La programmation par démonstration permet de générer des programmes à partir de l'espionnage des interactions entre l'utilisateur et le système. Le programme est enregistré dans l'interface du système sur lequel il s'exécute et possède en général une représentation différente de celle des fonctions natives du noyau fonctionnel. Par contre, la programmation par démonstration fabrique de réels programmes dont les paramètres sont identifiés.
- La géométrie paramétrée quant à elle, utilise la représentation interne du noyau fonctionnel pour conserver les relations entre les différents objets géométriques composant une pièce paramétrée. Ainsi, lorsque la valeur de l'un de ces objets change, le système est en mesure de recalculer la valeur de chacun des autres objets de façon à ce que les relations soient toujours respectées. Ce mode de programmation interactive est totalement transparent pour l'utilisateur qui ne sait pas qu'il construit une sorte de programme lorsqu'il crée une pièce, et que cette sorte de programme est exécutée lors d'une modification. Par contre, les paramètres d'un tel « programme » ne sont pas identifiés. La seule manière d'utiliser le programme pour créer un nouvel objet consiste à dupliquer le programme (et donc sa structure relationnelle sous-jacente), puis à éditer cette structure pour en changer un quelconque élément (par exemple une cote), et enfin à demander la réévaluation.

La géométrie paramétrique correspond au niveau de transparence recherché. Par contre, à la différence de la programmation par démonstration, elle ne permet pas d'associer au programme une signature, et donc une possibilité de ré-exécuter plusieurs fois le programme. Elle ne permet pas non plus d'associer plusieurs programmes à une même classe d'objets. Les propositions que nous avons développées utilisent les avantages d'ergonomie et de transparence de la géométrie paramétrique mais en l'étendant de façon à combler les deux lacunes identifiées.

La seconde partie de notre travail (chapitre 2) traite du problème de la modification dynamique de l'interface d'un système interactif graphique. On y propose une boîte à outils dite « de dialogue » qui possède à la fois les avantages des approches ascendantes et descendantes. Cette boîte à outils est composée de réifications d'éléments de description du dialogue. Elle permet de décrire le contrôle de dialogue d'une application de manière descendante en instanciant des briques de base de description du dialogue. Comme les briques de base de la présentation permettant de créer l'interface de manière ascendante, elles peuvent être instanciées

dynamiquement et ainsi permettre l'intégration dynamique de nouvelles classes d'objets décrites interactivement par l'utilisateur.

Le troisième chapitre décrit l'approche que nous proposons pour la génération interactive de classes additionnelles au sein d'un système interactif. Cette approche est basée sur un modèle paramétrique. Nous avons utilisé cette représentation pour enregistrer au sein du système les types d'opérations qu'il est nécessaire de faire apparaître dans une classe :

- les constructeurs, qui permettent de re-générer un objet de la classe en définissant ses paramètres effectifs,
- les sélecteurs qui permettent d'accéder à tout élément caractéristique des instances de la classe, qu'il s'agisse d'une entité interne ou d'une simple valeur, à condition qu'une fonction d'accès ait été définie lors de la création.

Cette notion de sélecteur, inspirée de la notion d'attribut dérivé du langage EXPRESS, permet donc de nommer et de rendre accessible des attributs spécifiques des objets issus de classes définies interactivement, ce qu'aucun système CAO paramétrique ne permet à notre connaissance. Elle permet également d'exploiter ces attributs pour établir des relations entre objets comme pour les objets relatifs où l'on peut exprimer, par exemple, que le « centre » d'un cercle se confond avec le milieu d'un segment.

Ainsi, les entités que permet de construire l'approche que nous proposons ne sont plus enregistrées sous la forme d'un unique programme, mais sous la forme d'une véritable classe possédant ses propres données, une fonction de construction et des fonctions d'interrogation.

La dernière partie (Chapitre IV) présente sommairement l'application CAO que nous avons réalisée pour valider les propositions faites aux chapitres II et III. Nous présentons d'abord son architecture générale. Puis, nous décrivons comment lier les éléments de description du dialogue de la boîte à outils du dialogue avec un modèle paramétrique pour la génération de classes. Un exemple de définition de classe nous permet de montrer les différentes fonctionnalités du système.

L'approche que nous proposons apparaît très naturelle pour un utilisateur non informaticien qui n'a, en aucun cas, à se préoccuper de programmation explicite. Toute la programmation est effectuée de façon transparente, de même que l'intégration au sein du système, effectuée

automatiquement. Nous avons également montré que cette intégration pouvait s'effectuer de deux manières :

- de façon dynamique, pour une mise au point immédiate, en utilisant l'approche interprétée,
- de façon statique en générant automatiquement du code C++ puis en compilant ce code, ce qui rend les nouvelles classes complètement identiques aux classes initiales du système.

En résumé, notre travail se situait à l'intersection de deux domaines : celui des interfaces homme-machine et celui de la programmation interactive sur l'exemple (ou programmation par démonstration). Afin de résoudre le problème qui nous était posé, nous avons développé des propositions originales dans chacun de ces deux domaines.

- Dans le domaine des interfaces homme-machine, nous avons proposé une nouvelle méthode et de nouveaux outils pour la conception du contrôleur de dialogue des applications graphiques interactives. Cette approche permet d'étendre au contrôleur de dialogue la notion, très efficace et bien acceptée, de « boite à outils » jusque là utilisée seulement au niveau de la présentation. Les apports essentiels de cette approche sont :
  - une homogénéité dans la manière de concevoir le contrôleur de dialogue et la couche de présentation,
  - la possibilité de spécifier et de générer très simplement le contrôleur de dialogue d'une application,
  - la possibilité de modifier dynamiquement le dialogue de l'application,
- la possibilité de manipuler des objets définis dynamiquement par l'utilisateur.

  Cependant, nous avons identifié deux limites à cette approche :
  - le fait de générer les automates de gestion du dialogue ne permet pas au concepteur de définir des modes d'interactions différents de ceux proposés par notre approche. A l'heure actuelle, il est très difficile de définir un dialogue supportant la manipulation directe, ce qui paraît préjudiciable pour une large diffusion de cette approche. Une étude est en cours pour palier à ce besoin particulier.
  - Aucune notion de temps n'a été introduite dans notre boîte à outils du dialogue, elle ne peut pas prendre en compte la fusion de modalité nécessaire à la gestion d'interface multimodales.

- Dans le domaine de la programmation par démonstration, nous avons proposé une approche consistant à séparer la spécification (au sens de la signature) des programmes et leur implémentation. Plusieurs spécifications et donc plusieurs programmes peuvent être associés à une même classe. Leurs implémentations sont enregistrées sous forme d'un arbre de construction de géométrie paramétrique. Cette approche permet en particulier, et sans aucune programmation explicite :
  - de définir de nouveaux types d'objets et la manière de les instancier,
  - de définir des fonctions, liées aux nouveaux types d'objet, permettant d'accéder aux entités ou aux propriétés caractéristiques des objets pour lesquelles elles sont définies,
  - d'intégrer ces types d'objets, définis interactivement, de manière dynamique ou statique dans l'application qui a permis de les définir.

Nous sommes donc très proche d'un véritable environnement permettant à un utilisateur de système CAO de spécifier son système sans aucune programmation.

Concernant les perspectives, dans le domaine des interfaces homme-machine, notre approche de conception du contrôleur de dialogue peut évoluer afin d'atteindre deux objectifs qui nous paraissent intéressant :

- le premier consiste à permettre la définition d'éléments permettant de gérer un dialogue basé sur la manipulation directe intégrée à un dialogue structuré. Une étude est d'ailleurs en cours afin d'avancer sur ce problème.
- la seconde consiste à étendre notre approche à d'autre types de dialogue en offrant au concepteur la possibilité de générer automatiquement l'automate contrôlant les appels des actions en fonction des interactions de l'utilisateur.

En ce qui concerne la génération interactive de classe, l'évolution qui nous paraît être la plus pertinente est l'adaptation de la méthode présentée aux modèles B-Rep. Cela permettrait à l'utilisateur de définir comme attributs d'un objet des éléments topologiques et ainsi il pourrait y accéder directement par leur nom. Cette extension nous paraît réalisable facilement en enregistrant le parcours dans l'arbre topologique permettant d'accéder à l'entité représentant l'attribut dans l'objet. Et vu les récents progrès dans la résolution des problèmes de persistance des noms, on peut imaginer que cette méthode sera fiable à topologie fixe et à topologie non fixe.

Bibliographie

[Agbodan, et al. 2000] Agbodan, D., Marcheix, D. et Pierra, G. Persistent Naming For Parametric Models. In *Proceedings of WSCG'2000* Prague), 2000.

[Agbodan, et al. 1998] Agbodan, D., Pierra, G. et Marcheix, D. A data model architecture for parametrics. In *Proceedings of ICECGDG* Anstin, Tx, USA), 1998, pp. 468-479.

[AHO, et al. 1991] AHO, A., SETHI, R. et ULLMAN, J. *COMPILATEURS : Principes et Techniques et Outils*. Paris, 1991.

[Aït-Ameur, et al. 1995] Aït-Ameur, Y., Besnard, F., Girard, P., Pierra, G. et Potier, J.-C. Specification and Metaprogramming in the EXPRESS Language. In *Proceedings of Intern. Conference on Software Engineering and Knowledge Engineering* (June, Rockville, USA), IEEE, ACM, 1995, pp. 181-189.

[Autebert 1994] Autebert, J.-M. *Théorie des langages et des automates*. Masson, 1994.

[Autodesk 1992] Autodesk. *3D Studio Reference Manual Version 3*. 1992.

[Balzert, et al. 1996] Balzert, H., Hofmann, F., Kruschinski, V. et Niemann, C. The JANUS Application Development Environment-Generating more than the User Interface. In *Proceedings of Computer-Aided Design of User iterface (CADUI'96)* (5-7 June, Namur, Belgium), Presse Universitaire de Namur, 1996, pp. 183-206.

[Bargen et Donnelly 1998] Bargen, B. et Donnelly, P. *Inside DirectX*. Microsoft Press, 1998.

[Bass, et al. 1991] Bass, l., Pellegrino, R., Reed, S., Sheppard, S. et Szezur, M. The Arch Model : Seeheim revisited. In *Proceedings of User Interface Developper's Workshop* 1991.

[Bauer 1979] Bauer, M.A. Programming by Examples. *Articial Intelligence*. 1979, pp. 1-21.

[Bellemain 1992] Bellemain, F. *Conception, Realisation et Utilisation d'un Logiciel d'aide a l'Enseignement de la Géométrie : Cabri Géomètre*. Doctorat d'Université (PhD Thesis) : Université Joseph Fourier, 1992.

[Boehm et Jacopini 1966] Boehm, C. et Jacopini, G. Flow Diagrams, Turing Machines and Languages with only two formation rules. *ACM communication*. 1966,

[Bouma, et al. 1995] Bouma, W., Fudos, I., Hoffmann, C., Cai, J. et Paige, R. Geometric Constraint Solver. *Computer Aided Design*. 1995, pp. 487-501

[CAS.CADE 2000] CAS.CADE. CAS.CADE. Matra Datavision. 2000. http://www.opencascade.com/.

[Chen 1995] Chen, X. *Representation, Evaluation and Editing of Feature-Based and Constraint-Based Design.* : Purdue University, 1995.

[Cohen et Murphy 1984] Cohen, B. et Murphy, G.L. Models of Concept. *Cognitive Science*. 1984, pp. 27-58

[Coutaz 1987] Coutaz, J. PAC, an Implementation Model for the User Interface. In *Proceedings of IFIP TC13 Human-Computer Interaction (INTERACT'87)* (September, Stuttgart), North-Holland, 1987, pp. 431-436.

[Coutaz 1990] Coutaz, J. *Interfaces Homme-Ordinateur, Conception et Réalisation*. Dunod Informatique, Paris, 1990.

[Cugini, et al. 1988] Cugini, U., Folini, F. et Vicini, I. A Procedural System for Definition and Storage of Technical drawings in Parametric Form. In *Proceedings of EUROGRAPHICS'88* Eurographics, 1988, pp. 183-196.

[Cypher 1993] Cypher, A. *Watch What I Do: Programming by Demonstration*. The MIT Press, Cambridge, Massachusetts, 1993.

[Cypher, et al. 1993] Cypher, A., Kosbie, D.S. et Maulsby, D. Characterizing PBD Systems. Cypher, A. (Ed.). In *Watch What I Do: Programming by Demonstration*, The MIT Press, Cambridge, Massachusetts, 1993, pp. 467-484.

[Dix, et al. 1993] Dix, A., Finlay, J., Abowd, G. et Beale, R. *Human-Computer Interaction*. Prentice Hall, 1993.

[Dix, et al. 1997] Dix, A., Mancini, R. et Levialdi, S. The cube - extending systems for undo. In *Proceedings of Eurographics Workshop on Design,*

Specification and Verification of Interactive Systems (DSV-IS'97) (4-6 June, Granada, Spain), Springer-Verlag, 1997, pp. 473-495.

[Duke et Harrison 1993] Duke, D.J. et Harrison, M.D. Abstract Interaction Objects. *Computer Graphics Forum*. 1993, pp. 25-36

[EXPRESS 1994] EXPRESS. *The EXPRESS language reference manual*. ISO, 1994 ISO 10303-11.

[Fekete 1996a] Fekete, J.-D. Trois besoins pour les systèmes interactifs (à vérifier). In *Proceedings of Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'96)* (16-18 Septembre, Grenoble), Cépaduès, 1996a, pp. 45-50.

[Fekete 1996b] Fekete, J.-D. *Un modèle multicouche pour la construction d'applications graphiques interactives*. Doctorat d'Université (PhD Thesis) : Université Paris-Sud, 1996b.

[Gardan 1991] Gardan, Y. *La CFAO introduction, techniques et mise en oeuvre*. Hermès, Paris, 1991.

[Girard 1992] Girard, P. *Environnement de Programmation pour Non-Programmeur et Paramétrage en Conception Assistée par Ordinateur : le système LIKE*. Doctorat d'Université (PhD Thesis) : Université de Poitiers, 1992.

[Girard 2000] Girard, P. *Ingénierie des système interactifs: vers des méthodes formelles intégrant l'utilisateur*. HDR : Ecole Nationnale de Mécanique et d'Aérotechnique, 2000.

[Girard et Pierra 1990] Girard, P. et Pierra, G. End User Programming Environments : Interactive Programming-On-Example in CAD Parametric Design. In *Proceedings of EUROGRAPHICS'90* (3-7 Sept, Montreux), Eurographics, 1990, pp. 261-274.

[GKS 1985] GKS. *Graphical Kernel System - Functional Description*. ISO, 1985 IS 7942.

[Goldberg 1984] Goldberg, A. *Smalltalk-80: The Interactive Programming Environment*. Addison-Wesley, 1984.

[Green 1986] Green, M.W. A Survey of three Dialogue Models. *ACM Transactions on Graphics*. 1986, pp. 244-275

[Guittet 1995] Guittet, L. *Contribution à l'Ingénierie des Interfaces Homme-Machine - Théorie des Interacteurs et Architecture H4 dans le système NODAOO*. Doctorat d'Université (PhD Thesis) : Université de Poitiers, 1995.

[Guittet et Pierra 1993a] Guittet, L. et Pierra, G. Conception modulaire d'une application graphique interactive de conception technique : la notion d'interacteur. In *Proceedings of Journées Francophones sur l'Ingénierie de l'Interaction Homme-Machine (IHM'93)* (Octobre, Lyon), École Centrale Lyon, 1993a, pp. 151-156.

[Guittet et Pierra 1993b] Guittet, L. et Pierra, G. *La notion d'interacteur : un concept unificateur pour la conception modulaire d'une application graphique interactive de conception technique*. Laboratoire d'Informatique Scientifique et Industrielle de l'Ecole Nationale Supérieure de Mécanique et d'Aérotechnique, 1993b 93008.

[Halbert 1984] Halbert, D. *Programming by Example*. PhD Thesis : University of California, 1984.

[Harrison et Duce 1994] Harrison, M.D. et Duce, D.A. *A review of formalisms for describing interactive behaviour*. University of York, January 7 1994.

[Heller et Ferguson 94] Heller, D. et Ferguson, P.M. *Motif Programming Manual*. 94.

[Hochberg 1989] Hochberg, S. De l'apport d'une bibliothèque au service des méthodes. In *Proceedings of MICAD* Paris), 1989.

[Hoffmann et Juan 1993] Hoffmann, C.M. et Juan, R. *EREP: an editable hight-level representation for geometric design and analysis*. Departement of Computer Sciences, Purdue University, 1993 1993, Tchenical Report CER-92-24.

[ISO13584.20 1998] ISO13584.20. ISO 13584-20, *Industrial Automation Systems and Intregration - Parts Library - Logical Model of Expressions*, ISO. In *Proceedings of* Geneve), 1998.

[ISO13584.31 1998] ISO13584.31. *ISO 13584-31, Industrial Automation Systems and Intregration - Parts Library - Part 31: Implementation resource: Geometric programming interface, ISO*. 1998.

[Jacob 1986] Jacob, R.J.K. A Specification Language for Direct Manipulation User Interfaces. *ACM Transaction on Graphics*. 1986, pp. 1203-1221

[Jambon 1997] Jambon, F. Error Recovery Representations in Interactive System Development. In *Proceedings of Third Annual ERCIM Workshop on "User Interfaces for All"* (3-4 november, Obernai, France), 1997, pp. 177-182.

[Kramer 1992] Kramer, G. *Solving Geometric Constraint Systems*. MIT Press, 1992.

[Kripac 1995] Kripac, J. A mechanism for persistently naming topological entities in history-based parametric solid models (Topological ID System). In *Proceedings of Solid Modelling'95* Salt Lake City, Utha, USA), 1995, pp. 21-30.

[Laakko et Mäntylä 1996] Laakko, T. et Mäntylä, M. Incremental constraint modelling in a feature modelling system. In *Proceedings of EUROGRAPHIC'96* Poitiers), Blackwell Plublishers, 1996, pp. 366-376.

[Laborde et Laborde 1991] Laborde, C. et Laborde, J.-M. MicroMonde et environnements d'Apprentissage. In *Proceedings of XIII Journées francophones sur l'informatique* Grenoble), 1991, pp. 157-177.

[Landay et Myers 1995] Landay, J.A. et Myers, B.A. Interactive Sketching for the Early Stages of User Interface Design. In *Proceedings of Human Factors in Computing Systems (CHI'95)* (7-11 May, Denver, Colorado), ACM/SIGCHI, 1995, pp. 43-50.

[Lecolinet 1999] Lecolinet, È. XXL: A Visual+Textual environment for Building Graphical User Interfaces. In *Proceedings of Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)* (21-23 October, Louvain-la-Neuve, Belgique), Kluwer Academic Publishers, 1999, pp. 115-126.

[Lewis et Norman 1986] Lewis, C. et Norman, D.A. Designing for Error. Norman, D. A. et Draper, S. W. (Ed.). In *User Centered System Design / New*

Perspectives on Human-Computer Interaction, Lawrence Erlbaum Associates, Hillsdale NJ, USA, 1986, pp. 411-432.

[Lieberman, et al. 1998] Lieberman, H., Nardi, B.A. et Wright, D. Grammex : Defining Grammar by Example. In *Proceedings of Human Factors in Computing Systems (CHI'98)* (18-23 April, Los Angeles, California), ACM/SIGCHI, 1998, pp. 11-12.

[Lonczewski et Schreiber 1996] Lonczewski, F. et Schreiber, S. The FUSE-System: an integrated User Interface Design Environment,. In *Proceedings of Computer-Aided Design of User iterface (CADUI'96)* (5-7 June, Namur, Belgium), 1996, pp. pp. 37-56.

[Loukipoudis 1996] Loukipoudis, E.N. Object management in a programming-by-example, parametric, computer-aided-design system. *The Visual Computer*. 1996, pp. 296-306

[Mäntylä 1990] Mäntylä, M. A modelling system for top-down design of assembled products. *IBM Journal of Research and Developpement*. 1990, pp. 636-659

[Masini, et al. 1989] Masini, G., Napoli, A., Colnet, D., Léonard, D. et Tombre, K. *Les Langages à Objets*. InterEditions, Paris, 1989.

[McDaniel et Myers 1998] McDaniel, R.G. et Myers, B.A. Building Applications Using Only Demonstration. In *Proceedings of IUI'98* (January 6-9, San Francisco), 1998, pp. 119-128.

[McDaniel et Myers 1999] McDaniel, R.G. et Myers, B.A. Gamut : Creating Complete Applications Using Only Programming by Demonstration. In *Proceedings of CHI'99* 1999.

[Meinadier 1991] Meinadier, J.P. *L'interface Utilisateur : pour une informatique plus conviviale*. Dunod Informatique, Paris, 1991.

[Microsoft 1996] Microsoft. *Microsoft Office 97 / Visual Basic Programmer Guide*. Microsoft Press, 1996.

[Microsoft 1999] Microsoft. *Formation à Microsoft Visual C++ 6*. Microsoft, 1999.

[Mills 1975] Mills, H. The new Math of Computer Programming. *Communivation of the ACM*. 1975, pp. 74-82

[Moranne 1988] Moranne, J.M. Les bibliothèques SGAO de composants mécaniques. In *Proceedings of MICAD* Paris), 1988.

[Myers 1998] Myers, A.B. Scripting Graphical Applications by Demonstration. In *Proceedings of Human Factors in Computing Systems (CHI'98)* (18-23 April, Los Angeles, Californie), ACM/SIGCHI, 1998, pp. 534-541.

[Myers 1990] Myers, B.A. A New Model For Handling Input. *ACM Transactions on Information Systems*. 1990, pp. 289-320

[Myers 1995] Myers, B.A. User Interface Software Tools. *ACM Transactions on Computer Human Interaction*. 1995, pp. 64-103

[Myers, et al. 1997] Myers, B.A., McDaniel, R.G., Miller, R.C., Ferrency, A.S., Faulring, A., Kyle, B.D., Mickish, A., Klimovitski, A. et Doane, P. The Amulet Environment: New Models for Effective User Interface Software Development. *IEEE Transactions on Software Engineering,*. 1997, pp. 347-365

[Myers et Rosson 1992] Myers, B.A. et Rosson, M.B. Survey on User Interface Programming. In *Proceedings of SIGCHI'92* (May, Monteray, CA), 1992, pp. 192-202.

[Neider, et al. 1993] Neider, J., Davis, T. et Woo, M. *OpenGL Programming Guide*. Reading, MA, USA, 1993.

[Newell, et al. 1983] Newell, R., Parden, G. et Parden, P. Parametric Design in MEDUSA System. In *Proceedings of CAPE'83* (Apr. 25-28, Amsterdam), 1983.

[Nigay 1994] Nigay, L. *Conception et Modélisation Logicielle des Systèmes Interactifs : Application aux Interfaces Multimodales*. Doctorat d'Université (PhD Thesis) : Université Joseph Fourier, 1994.

[Norman 1986] Norman, D. *User Centered System Design*. Lawrence Erlbaum Associates, 1986.

[Olsen 1983] Olsen, D.R. SYNGRAPH : a Graphical User Interface Generator. *ACM Transaction on Graphics*. 1983, pp. 43-50

[Olsen 1992] Olsen, D.R. *User Interface Management Systems: Models and Algorithms*. Morgan Kaufmann Publisher, San Mateo (CA), USA, 1992.

[Olsen et Dance 1988] Olsen, D.R. et Dance, J.R. Macros by Example in a Graphical UIMS. *IEEE Computer Graphics and Applications*. 1988, pp. 68-78

[Olsen 1986] Olsen, R. Mike : the Menu Interaction Kontrol Environment. *ACM Transaction on Graphics*. 1986, pp. 318-344

[Ousterhout 1994] Ousterhout, J. *Tcl and the Tk Toolkit*. Addison-Wesley, 1994.

[Owen 1991] Owen, J. Algebraic Solution for Geometry from Dimensional Constraints. In *Proceedings of ACM Symposium on Foundations of Solid Modelling* (8-10 May, Austin, Texas), ACM/SIGGRAPH, 1991, pp. 397-407.

[Paternò 1994] Paternò, F. A Theory of User-Interaction Objects. *Journal of Visual Languages and Computing*. 1994, pp. 227-249

[Paternò et Faconti 1994] Paternò, F. et Faconti, G.P. A semantics-based approach for the design and implementation of interaction objects. *Computer Graphics Forum*. 1994, pp. 195-204

[Patry 1999] Patry, G. *Contribution à la conception du dialogue Homme Machine dans les applications graphiques interactives de conception technique : le système GIPSE*. Doctorat d'Université (PhD Thesis) : Université de Poitiers, 1999.

[Patry et Girard 1999] Patry, G. et Girard, P. GIPSE: a Model-Based System for CAD. In *Proceedings of Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)* (21-23 October, Louvain-la-Neuve, Belgique), Kluwer Academic Publishers, 1999, pp. 61-72.

[Pfaff 1985] Pfaff, G.E. *User Interface Management Systems, Proceedings of the Workshop on User Interface Management Systems held in Seeheim*. Springer-Verlag, Berlin, 1985.

[PHIGS 1989] PHIGS. *Programmers Hierarchical Interactive Graphics System - Functional Description*. ISO, 1989 IS 9592.

[Pierra 1991] Pierra. *Les bases de la programmation et du Génie Logiciel*. Dunod informatique, Paris, 1991.

[Pierra 1995] Pierra, G. Towards a taxonomy for interactive graphics systems. In *Proceedings of Eurographics Workshop on Design, Specification, Verification of Interactive Systems* (June 7-9, Bonas), Springer-Verlag, 1995, pp. 362-370.

[Pierra, et al. 1996a] Pierra, G., Aït Ameur, Y., Besnard, F., Girard, P. et Potier, J.-C. A General Framework for Parametric Product Model within STEP and Parts Library. In *Proceedings of European PDT Days* (18-19 April, London, UK), PDTAG-AM, 1996a, pp. 69-104.

[Pierra, et al. 1996b] Pierra, G., Potier, J.-C. et Girard, P. The EBP system : Example Based Programming for Parametric Design. Teixeira, J. et Rix, J. (Ed.). In *Modelling and Graphics in Science and Technology*, Springer-Verlag, 1996b, pp. 124-140.

[Potier 1995] Potier, J.-C. *Conception sur exemple, mise au point et génération de programmes portables de géométrie paramétrée dans le système EBP*. Doctorat d'Université (PhD Thesis) : Université de Poitiers, 1995.

[Potier, et al. 1995] Potier, J.-C., Girard, P., Pierra, G. et Besnard, F. Génération graphique interactive de programmes de géométrie paramétrée. *Revue d'Automatique et de Productique Appliquée (RAPA)*. 1995, pp. 229-234

[Procise 1999] Procise, J. *Programming Windows with MFC, Second Edition*. Microsoft Press, 1999.

[Puerta 1996] Puerta, A. The MECANO project : comprehensive and integrated support for Model-Based Interface development. In *Proceedings of Computer-Aided Design of User iterface (CADUI'96)* (5-7 June, Namur, Belgium), Presse Universitaire de Namur, 1996, pp. 19-35.

[Puerta et Eisenstein 1998] Puerta, A. et Eisenstein, J. Interactively Mapping Task Model to Interfaces in Mobi-D. In *Proceedings of Eurographics Workshop on Design, Specification and Validation of Interactive Systems (DSV-IS'98)* (3-5 June, Abingdon, UK), 1998, pp. 261-274.

[Puerta 1997] Puerta, A.R. A Model-Based Interface Development Environment. *IEEE Software*. 1997, pp. 40-47

[Puerta, et al. 1999] Puerta, A.R., Cheng, E., Ou, T. et Min, J. MOBILE: User-Centered Interface Building. In *Proceedings of CHI'99* Pittsburgh, PA, USA), ACM/SIGCHI, 1999, pp. 426-433.

[Raghotama et Shapiro 1997] Raghotama, S. et Shapiro, V. *Boundary Representation Variance in Parametric Solid Modelling*. University of Wisconsin-Madison, 1997 1997 SAL 1997-1.

[Roller 1990] Roller, D. Dimension-Driven Geometry in CAD: a Survey. In *Theory and Practice of Geometric Modelling*, Springer-Verlag, 1990, pp. 509-523.

[Scapin et Pierret-Golbreich 1990] Scapin, D.L. et Pierret-Golbreich, C. Towards a method for task description : MAD. Berliguet, L. et Berthelette, D. (Ed.). In *Working with display units*, Elsevier Science Publishers, North-Holland, 1990, pp. 371-380.

[Schenck et Wilson 1994] Schenck, D. et Wilson, P. *Information Modelling The EXPRESS Way*. Oxford University Press, 1994.

[Senella 1993] Senella, M. *The Skyblue Constraint Solver*. Departement of Computer Sciences and Engineering, University of Washington, Februrary 1993 1993, Tchenical Report 92-07-02.

[Shah, et al. 1994] Shah, J., Balakrishnan, G., Rogers, M. et Urban, S. Comparative Study of Procedural And Declarative Feature Based Geometric Modeling. In *Proceedings of IFIP Internationnal Conference* (May, 1994, Valenciennes), 1994.

[Shah et Mäntylä 1995] Shah, J.J. et Mäntylä, M. *Parametric and Feature-based CAD/CAM: Concepts, Techniques and Applications*. John Wiley & Sons, New York, 1995.

[Shumerck 1986] Shumerck, K. MacApp : An application Framework. *Byte*. 1986, pp. 189-193

[Singh et Green 1991] Singh, G. et Green, M. Automating the Lexical and Syntaxic Design of Graphical User Interfaces : the UoA*UIMS. *ACM Transaction on Graphics*. 1991, pp. 213-254

[Smith et Cypher 1995] Smith, D. et Cypher, A. KidSim : Child Constructible simulation. In *Proceedings of Imagina'95* Monte-Carlo, Février), 1995, pp. 87-99.

[Solano et Brunet 1994] Solano, L. et Brunet, P. Constructive Constraint-Based Model for Parametric CAD Systems. *Computer Aided Design*. 1994, pp. 614-621

[Sutherland 1963] Sutherland, I.E. Sketchpad : A man machine graphical communication system. In *Proceedings of Proceedings-Spring Joint Computer Conference* IFIP/AFIPS, 1963, pp. 329-346.

[Szekely 1996] Szekely, P. Retrospective and challenge for Model Based Interface Development. In *Proceedings of Eurographics Workshop on Design, Specification, and Verification of Interactive Systems (DSV-IS'96)* (5-7 June, Namur, Belgium), Springer-Verlag, 1996, pp. 1-27.

[Szekely, et al. 1993] Szekely, P., Luo, P. et Neches, R. Beyond Interface Builders: Model-Based Interface Tools. In *Proceedings of InterCHI93* 1993, pp. 383-390.

[Szekely, et al. 1995] Szekely, P., Sukaviriya, P., Castells, P., Muthukumarasamy, J. et E. Salcher. Declarative interface models for user interface construction tools : the MASTERMIND approach. In *Proceedings of IFIP TC2/WG2.7 Working Conference on Engineering for Human-Computer Interaction (EHCI'95)* (14-18 August, Grand Targhee Resort (Yellowstone Park), USA), Chapman & Hall, 1995, pp. 120-150.

[Tarby 1993] Tarby, J.-C. *Gestion Automatique de Dialogue Homme-Machine à partir de spécifications conceptuelles*. Doctorat d'Université (PhD Thesis) : Université de Toulouse I, 1993.

[Texier et Guittet 1998] Texier, G. et Guittet, L. A toolset for supporting dialog. In *Proceedings of Nîmes 98* (26-28 May, Nîmes), 1998.

[Texier et Guittet 1999a] Texier, G. et Guittet, L. Dialog + Gadget = Diaget. In *Proceedings of IHM'99* (22-26 Novembre, Monpellier), Cepadues, 1999a, pp. 70-77.

[Texier et Guittet 1999b] Texier, G. et Guittet, L. User Defined Objects are First Class Citizen. In *Proceedings of Third Conference on Computer-Aided Design of User Interfaces (CADUI'99)* (21-23 October, Louvain-la-Neuve, Belgique), Kluwer Academic Publishers, 1999b, pp. 231-244.

[Thimbleby 1990] Thimbleby, H. *User Interface Design*. ACM Press, 1990.

[UIMS 1992] UIMS. The UIMS Workshop Tool Developers: A Metamodel for the Runtime Architecture of an Interactive System. *SIGCHI Bulletin*. 24, 1 (1992), pp. 32-37.

[Van der Schaaf 1997] Van der Schaaf, T.W. Prevention and Recovery of Errors in System Software. In *Proceedings of Workshop on Human Error and System Development* (19-22 March, Glasgow University, Scotland), Glasgow Accident Analysis Group, 1997, pp. 49-57.

[Van Emmerick 1991] Van Emmerick, M. *Interactive Design of Parametrized 3D models by Direct Manipulation*. PhD Thesis : Université de Delft, 1991.

[Wall, et al. 1996] Wall, L., Christiansen, T. et Schwartz, R. *Programming Perl, Second Edition*. O'Reilly and Associates, Inc., 1996.

[Wiecha, et al. 1989] Wiecha, C., Bennet, W. et al, e. Generating Higly Interactive User Interfaces. In *Proceedings of Human Factors in Computing Systems (CHI'89)* (30 April-4 May 1989, Austin, USA), ACM/SIGCHI, 1989, pp. 277-282.

[Wiecha, et al. 1990] Wiecha, C., Bennet, W., Boies, S., Gould, J. et Greene, S. ITS: a tool for rapidly developing interactive applications. *ACM Transactions on Information Systems*. 1990, pp. 204-236

[Wolber 1996] Wolber, D. Pavlov : Programming by Stimulus-Response Demonstration. In *Proceedings of Human Factors in Computing Systems (CHI'96)* (13-18 April, Vancouver, Canada), ACM/SIGCHI, 1996, pp. 252-269.

[Woods 1970] Woods, W. Transition Network Grammars for Natural Language Analysis. *Communications of the ACM*. 1970, pp. 591-606

[Yang 1992] Yang, Y. Anatomy of the design of an undo support facility. *International Journal of Man-Machine Studies*. 36, 1 (1992), pp. 81-95.

[Zalik 1996] Zalik, B. An Interactive Constraint-Based Graphics System with Partially Constrained Form-Features. In *Proceedings of Computer-Aided Design of User iterface (CADUI'96)* (5-7 Juin, Namur, Belgium), Presse Universitaire de Namur, 1996, pp. 129-139.

Annexe

Le langage EXPRESS-G

La représentation textuelle des schémas EXPRESS les rend difficilement lisibles. C'est pour pallier ce problème de lisibilité qu'un formalisme graphique de représentation a été élaboré : EXPRESS-G [EXPRESS 1994, Schenck et Wilson 1994]. Ce formalisme permet de donner une vue synthétique des modèles de données. De plus, une telle représentation est très adaptée dans le cadre de la phase d'analyse d'un problème de modélisation. Ce symbolisme de représentation ne permet d'exprimer que le caractère structurel et descriptif du modèle de données. Les contraintes d'intégrité ne sont pas exprimables, mais il est cependant possible de spécifier un attribut ou bien une classe comme possédant une contrainte dans la phase avale d'écriture du modèle de données textuel, ceci par l'apposition du caractère '*' devant l'attribut ou la classe contraints. Enfin, notons que le rôle assigné aux attributs (optionnel, dérivé ou inverse), ainsi que leur type associé, est représenté. La représentation en EXPRESS-G d'un modèle représentant des enseignants et des étudiants est donnée sur la figure ci-dessous.