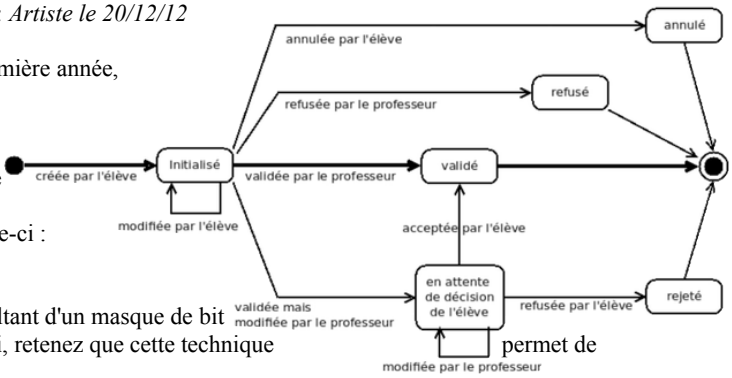
Rappel du contexte :

Au cours de l'année 2012, une application, appelée GeTAP, fut créée. Celle-ci permet de gérer le temps d'accompagnement personnalisé par élèves. Ce temps est compté grâce aux demandes à l'initiative des élèves .

Exemple : 2h de soutien en Art moderne avec le Professeur Mr. Artiste le 20/12/12

Je vous renvois au contexte de ce projet : « lien vers portfolio ».
Pendant la réalisation de ce projet, application développée en première année, la classe (voir le paradigme de la programmation objet) DemandeValidationConsoTempsAccPers.java fut créée.

Celle ci donne la possibilité de créer/gérer le cycle de vie d'une demande .
Voici un diagramme exprimant les différents états possible de celle-ci :



Lors de la première adaptation de ce diagramme dans GeTAP,
Ces états se distinguent par une valeur numérique en base 10 résultant d'un masque de bit (valeur en base 2), l'utilité du masque de bit n'est pas expliquée ici, retenez que cette technique permet de connaître tous les états « vécu » par une demande.

Exemple : 2 → Initialisé ; 4→ Validé ; 6 → Refusé ; 8 → ...

Problème :

Cette représentation des états dans le code, vous le conviendrez, n'a rien d'explicite ! Et ne respecte donc pas la convention de nommage des variables qui impose des noms parlants. De plus, cette application maintenant en phase de test/production, sera sûrement reprise par d'autres développeurs impatients d'apporter leur pierre à cet édifice.
La non-compréhension du fonctionnement de l'existant pourrait en décourager plus d'un. Il fallait donc proposer un refactoring (voir le rapport complet portfolio) :

« Nous souhaitons que le code source exprime de façon plus claire, plus communicante, les différentes transitions responsables des changements d'états des demandes »

Plan d'actions :

Ce refactoring, de par sa taille et son importante (impacte la classe centrale de l'application), se décomposa en plusieurs étapes.
La première consista à créer une batterie de tests unitaires validant le passage dans les différentes méthodes de l'existant. Ces tests ont permit de détecter des problèmes de sémantiques importants comme le non respect du cycle de vie de la demande dans certaines situations. (*exemple : la modification d'une demande par l'élève après la validation du professeur*), ces tests serviront également dans l'une des dernières étapes de ce refactoring.

La seconde étape à été d'extraire cette classe de son projet et de procéder au renommage des états, ainsi 32 est devenu une constante définie comme tel : *private static final int VALIDEE_PROF = 32;* Ce passage par une constante permet d'exprimer plus explicitement les états d'une demande.
Une méthode par état fut également générée afin de centraliser les conditions de passage a celui-ci.

Exemple :

```
/**
 * Lève le bit de validation par le professeur en fonction de l'état en cour
 *
 * @throws DVCTAPException
 */
public void valideeParLeProfesseur() throws DVCTAPException {
	if (!this.isAnnuleeEleve() && !this.isRefuseeProf()
			&& !this.isAccepteeEleve() && !this.isRejeteeEleve()
			&& !this.isValideeProf()) {
		this.etat = this.etat | VALIDEE_PROF;
	} else {
		throw new DVCTAPException(errorReporting());
	}
}
```

Les is... appelé « iser » sont des méthodes de cette même classe, retournant un booléen, qui permettent de vérifier chaque bit de la valeur de l'état de la Demande en cours afin de connaître son chemin.
Traduction littérale de la méthode valideeParLeProfesseur() :
Si la demande n'est ni annulée par l'élève ni refusée par le professeur ni acceptée par l'élève ni rejetée par celui-ci et enfin ni validée par le professeur, on lève le bit du VALIDEE_PROF soit «32» à 1. Sinon on génère une exception dédiée.

La troisième étape plus légère consista à modifier toutes les affectations directes des états d'une demande par la méthode qui lui est propre. Exemple remplacement de laDvctap.setEtat(32) ; par laDvctap.valideeParLeProfesseur() ; les conditions délocalisées dans les nouvelles méthodes, cela permit d'alléger les autres processus métiers et donc améliorer la lisibilité générale du code source.

La quatrième et dernière étape a été de ré-implémenter la classe modifiée dans l'existant et s'assurer du bon fonctionnement de celle-ci notamment par l'intermédiaire des tests unitaires créés dans l'étape 1.

Exemple d'un test unitaire :

```
@Test
public void testValiderProfesseur() {
	try {
		dvctap.setEtat(0);
		dvctap.valideeParLeProfesseur();
		dvctap.modifieeParEleve();
		fail("Etat modifié élève");
	} catch (DVCTAPException e) {
		assertTrue(1 == 1);
	}
}
```

Ce test implique qu'une exception de type DVCTAPException doit être levée pour passer ce test, en effet si la demande est modifiée après avoir été validée par le professeur le test fail avec « État modifié élève » en message.
Si tous les tests passent, comme le montre l'image ci-dessous, nous avons réussis !