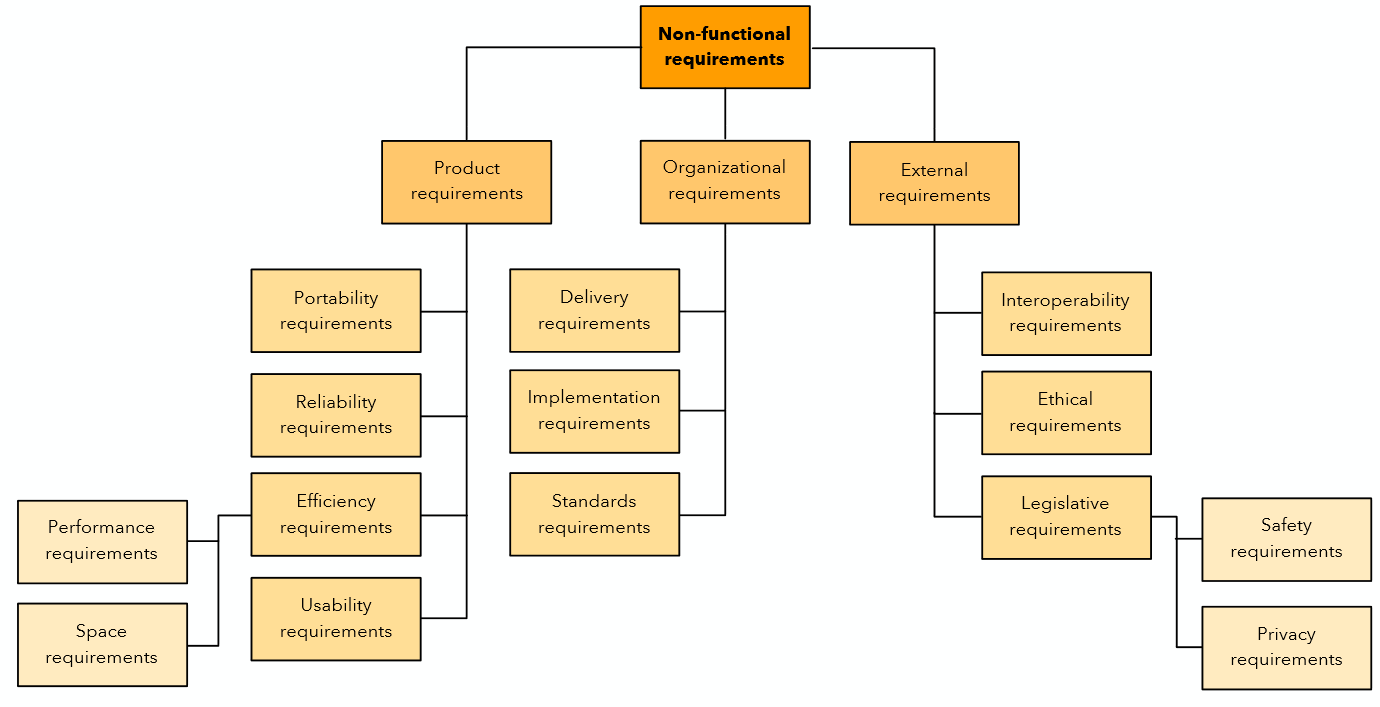
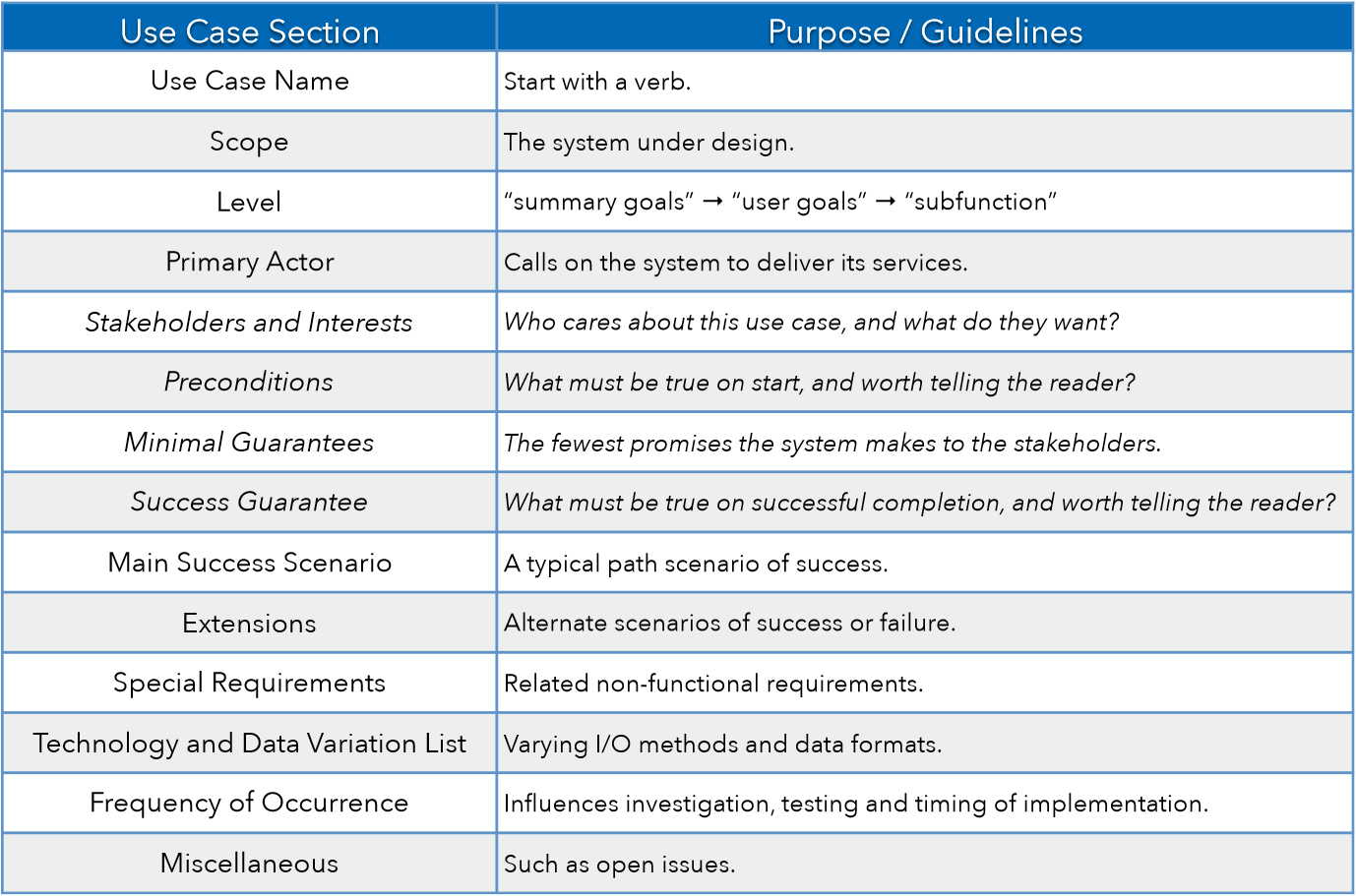
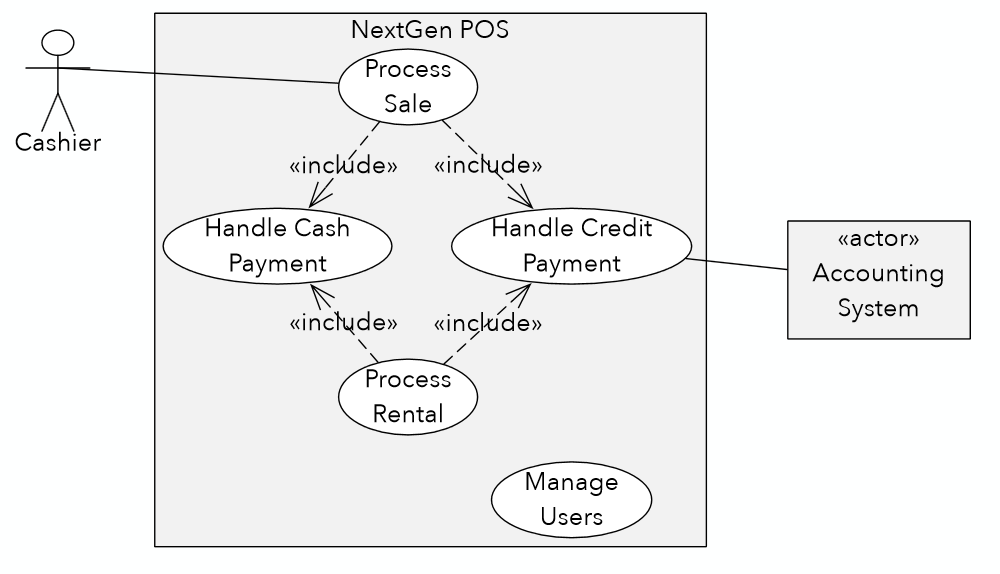
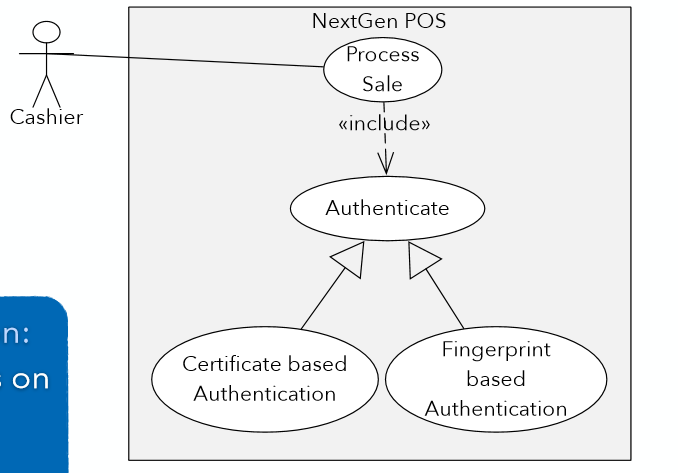
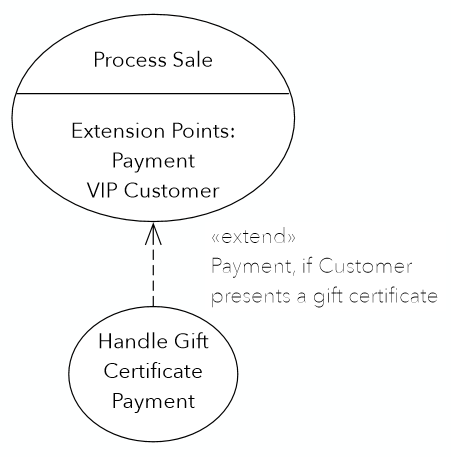
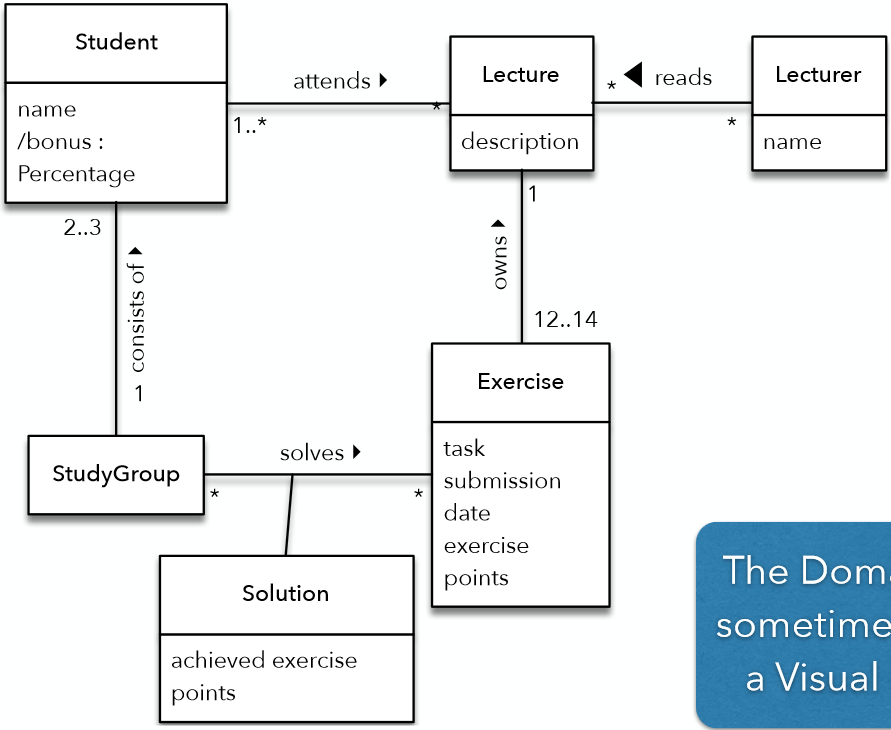
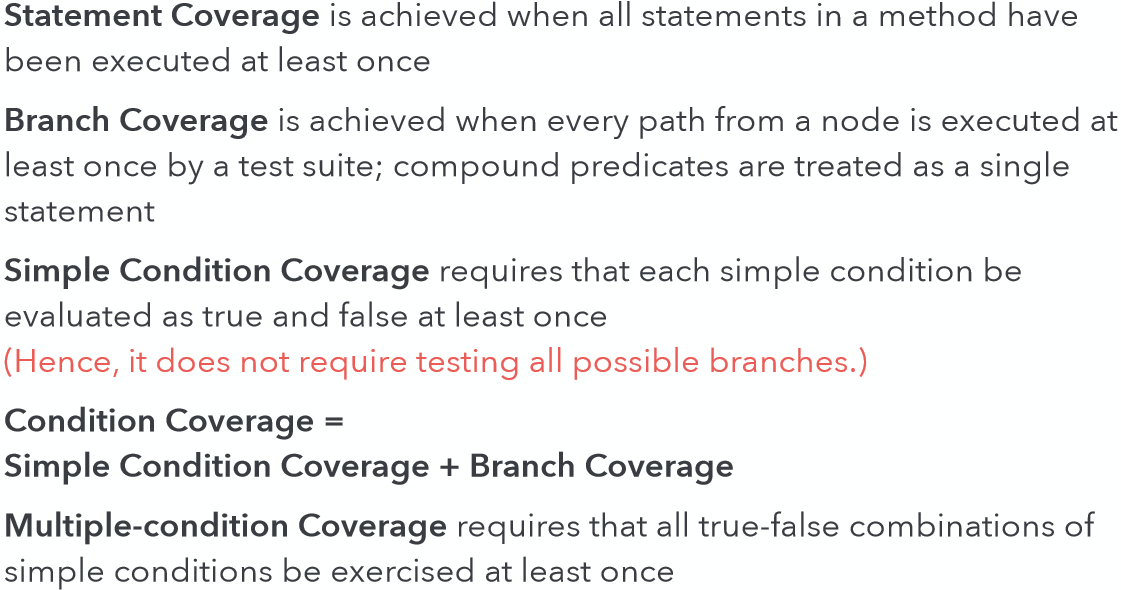
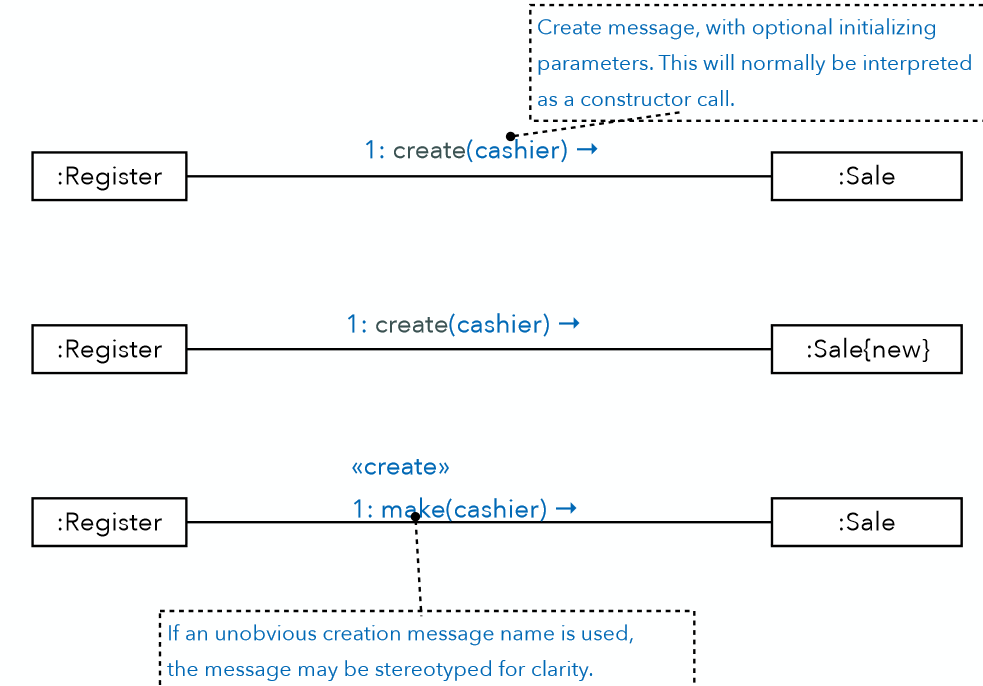
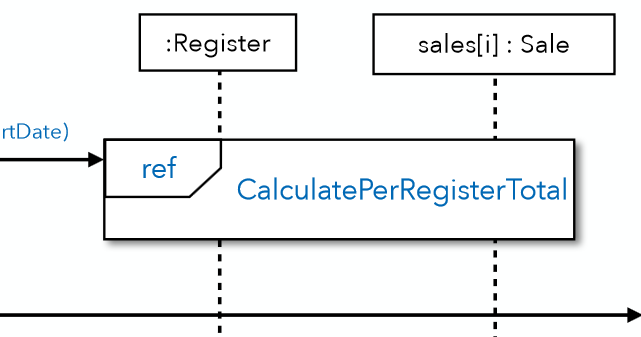
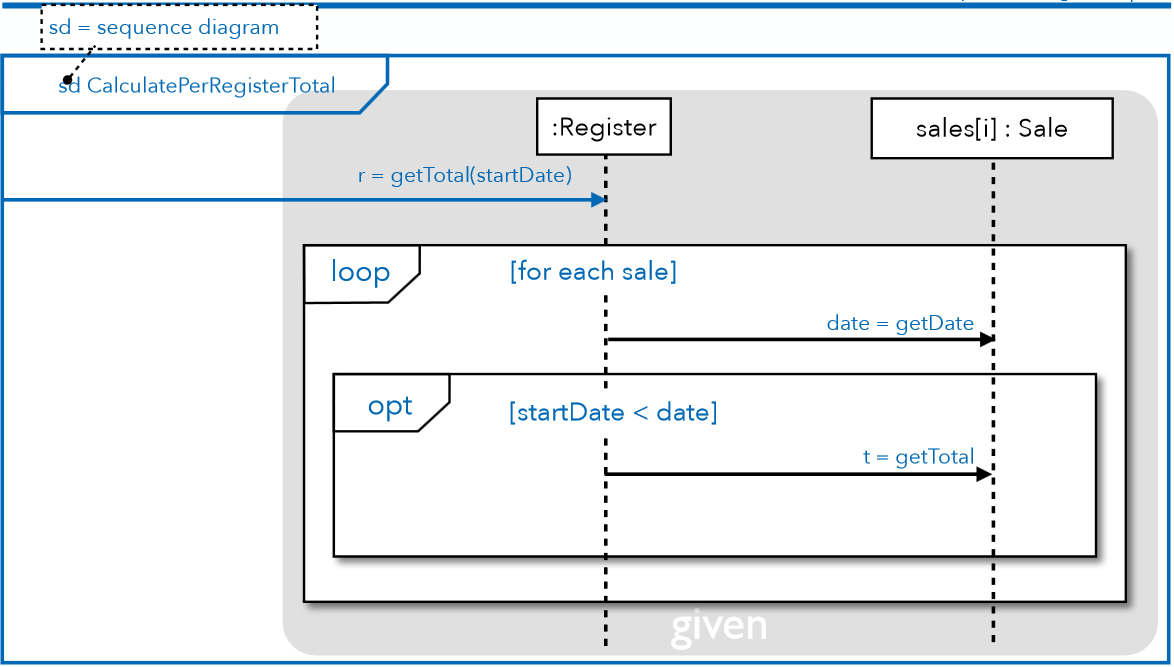
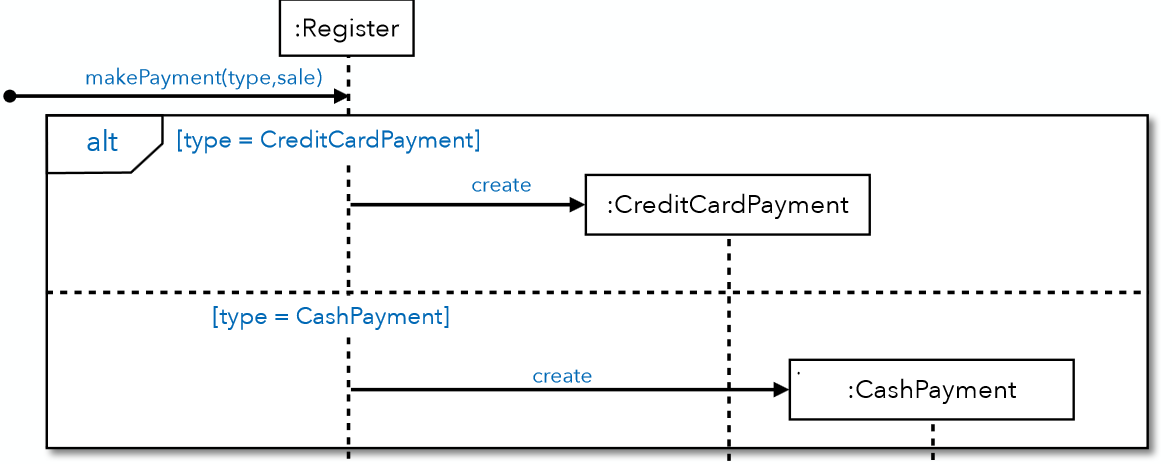
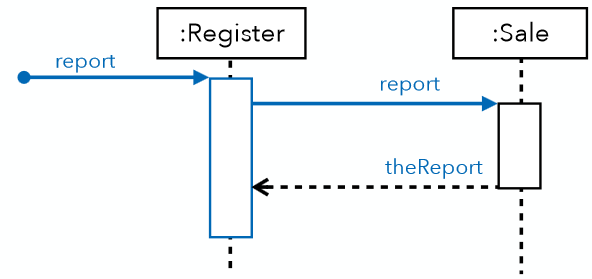
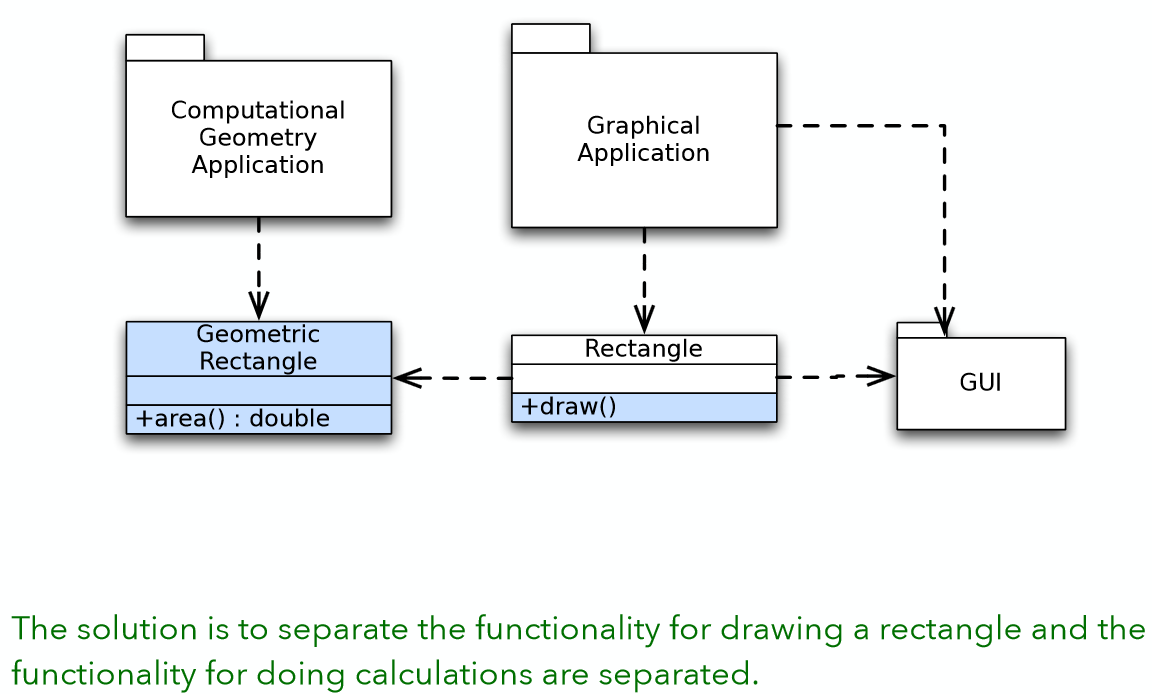
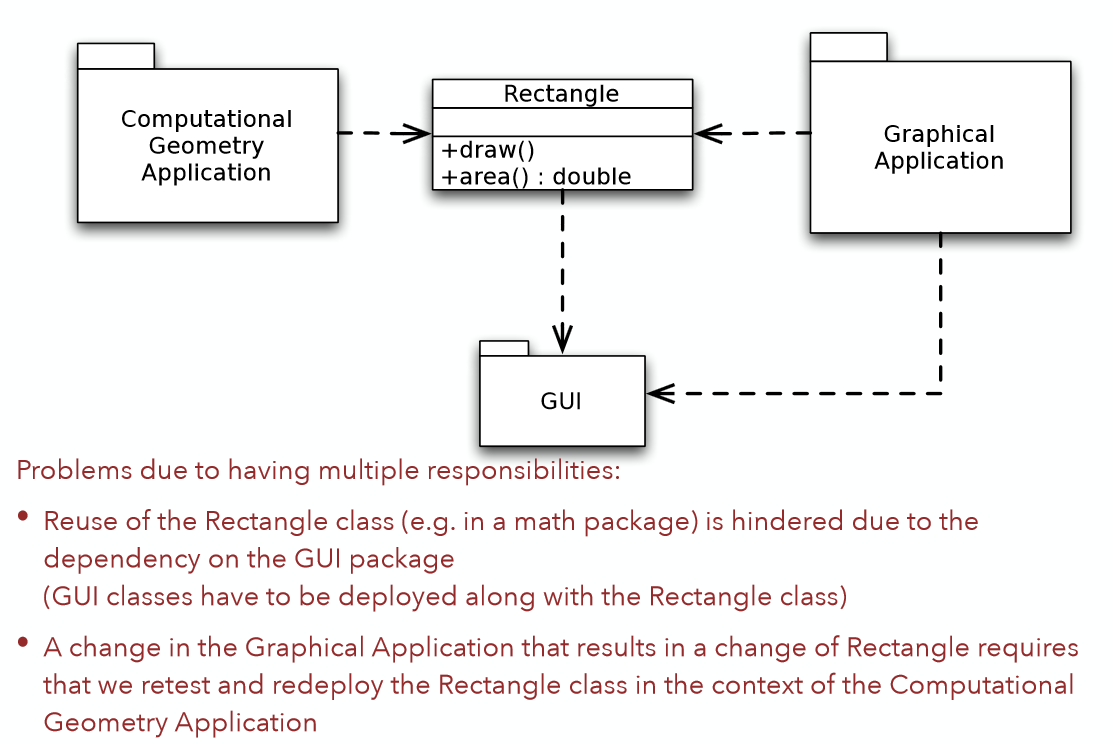
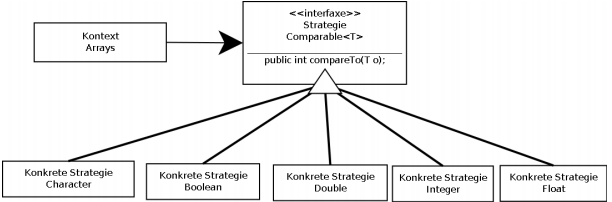
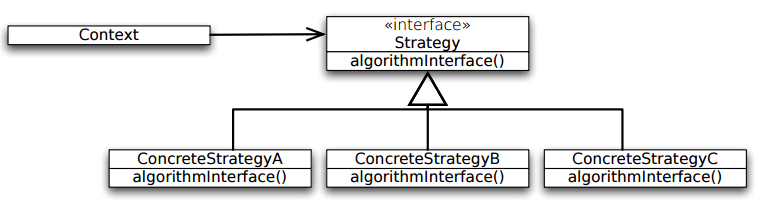
Anforderungsmanagement (Requirements Engineering)

**Benutzeranforderungen (User Requirements)(grndlg. für Lastenheft):** informell, Diagramme: Welche Dienste soll das System bieten, Einschränkungen  
**Systemanforderungen (System Requirements)(grndlg. für Pflichtenheft):** präzise, detailliert, evtl. formale Modellierung  
**Funktionale Anforderungen:** Dienste des Systems, wie es (nicht) auf bestimme Inputs/Situationen reagieren soll, durch Code ausdrückbar  
**Nichtfunktionale Anforderungen:** Einschränkungen der Dienste des Systems, u.a. timing, standards, Entwicklungsprozess Einschr., meistens das komplette System betreffend  
  
Anwendungsfall (Use Case)  
   
  
  
  
  
  
Domain Model  
Testing:  
**Validation:** Are we building the right product?  
**Verification:** Are we building the product right?  
  
Sequenzdiagramme:  
  
Systemsequenzdiagramme  
Zeigt für ein Szenario eines Use Cases die Events, die external actors auslösen, ihre Reihenfolge, und inter-system events. Das System wird als black-box behandelt.  
Kopplung  
Misst die Stärke der Abhängigkeit zwischen Klassen und Packages. C1 ist gekoppelt an C2, wenn C1 C2 direkt oder indirekt benötigt.  
Hohe kopplung ist schlecht, wegen Änderungen in den Abhängigkeiten lokale Änderungen verlangen können, sie schwerer in Isolation zu verstehen ist und schwerer zu reusen, weil alle anderen Klassen included werden müssen.  
Niedrige Kopplung ist wichtig, vor allem bei generischen, oft wiederverwendeten Klassen. Kann Kosten und Komplexität des Projekts erhöhen.  
Kohäsion  
Misst die Stärke der Beziehung zwischen Elementen der Klasse. Alle Operationen und Daten sollten natürlich zusammenhängen. Typen:  
Coincidental Cohesion: Keine bedeutungsvolle Beziheung zwischen den Elementen der Klasse  
Logical/Functional Cohesion: Die Elemente führen eine Art logischer Funktion aus  
Temporal Cohesion: Alle Elemente werden zusammen ausgeführt  
Template Method Design Pattern:  
Teile eines Algorithmus einer abstrakten Klasse werden in Subklassen implementiert.

Strategy Design Pattern:  
Strategie: FancyHandle (auch …Strategy)  
Konkrete Strategie: ChineseWallStrategy, FloodingStrategy,… (auch …Action)  
Kontext: Map  
Methode: action(Map,Player):int  
Variante: Benötigte Informationen werden als Parameter übergeben  
  
Composite Design Pattern:  
**Component**

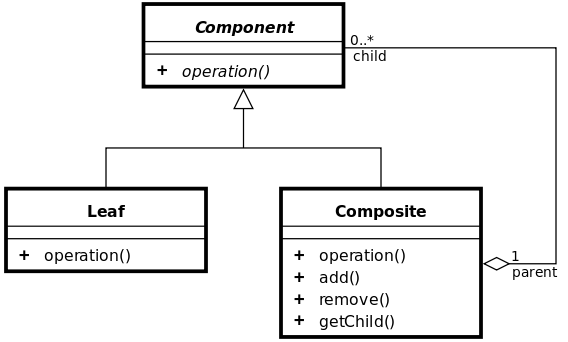
* is the abstraction for all components, including composite ones
* declares the interface for objects in the composition
* (optional) defines an interface for accessing a component's parent in the recursive structure, and implements it if that's appropriate

**Leaf**

* represents leaf objects in the composition
* implements all Component methods

**Composite**

* represents a composite Component (component having children)
* implements methods to manipulate children
* implements all Component methods, generally by delegating them to its children

  
Notation: Komponente, Kompositum, Blatt, Variante (Transparenz/Sicherheit), Methode  
Variante ist „Sicherheit“, wenn Composite die Operationen add, remove, getChild zum managen der Kinder deklariert.  
Variante ist „Transparenz“, wenn diese in Component deklariert werden. Ist bequemer, aber weniger sicher  
operation() in Composite führt für alle Kinder die Leaf.operation() aus.  
  
The Composite Design Pattern facilitates to compose objects into tree structures to represent part-whole hierarchies. Apply the composite pattern if clients can treat individual objects and compositions of objects uniformly