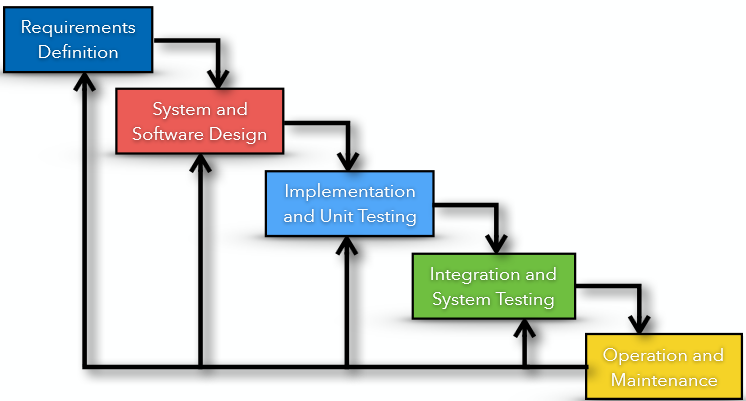
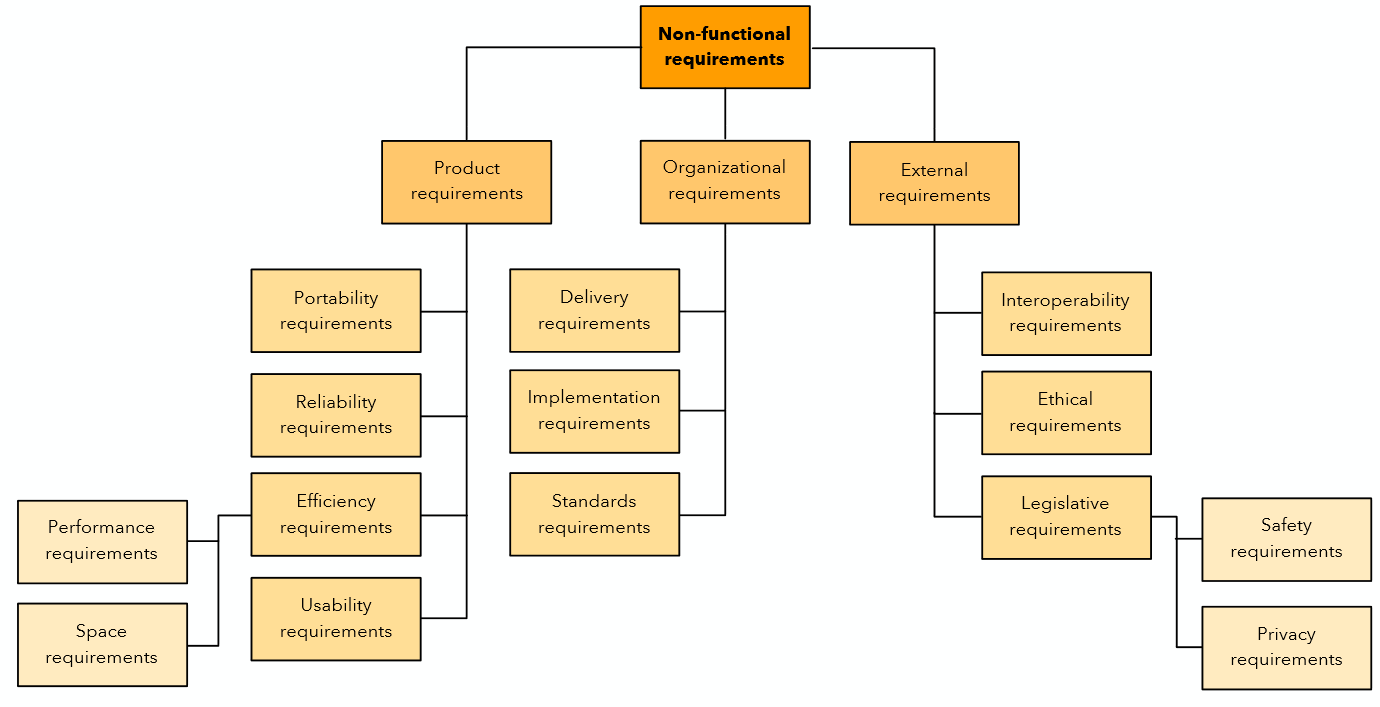
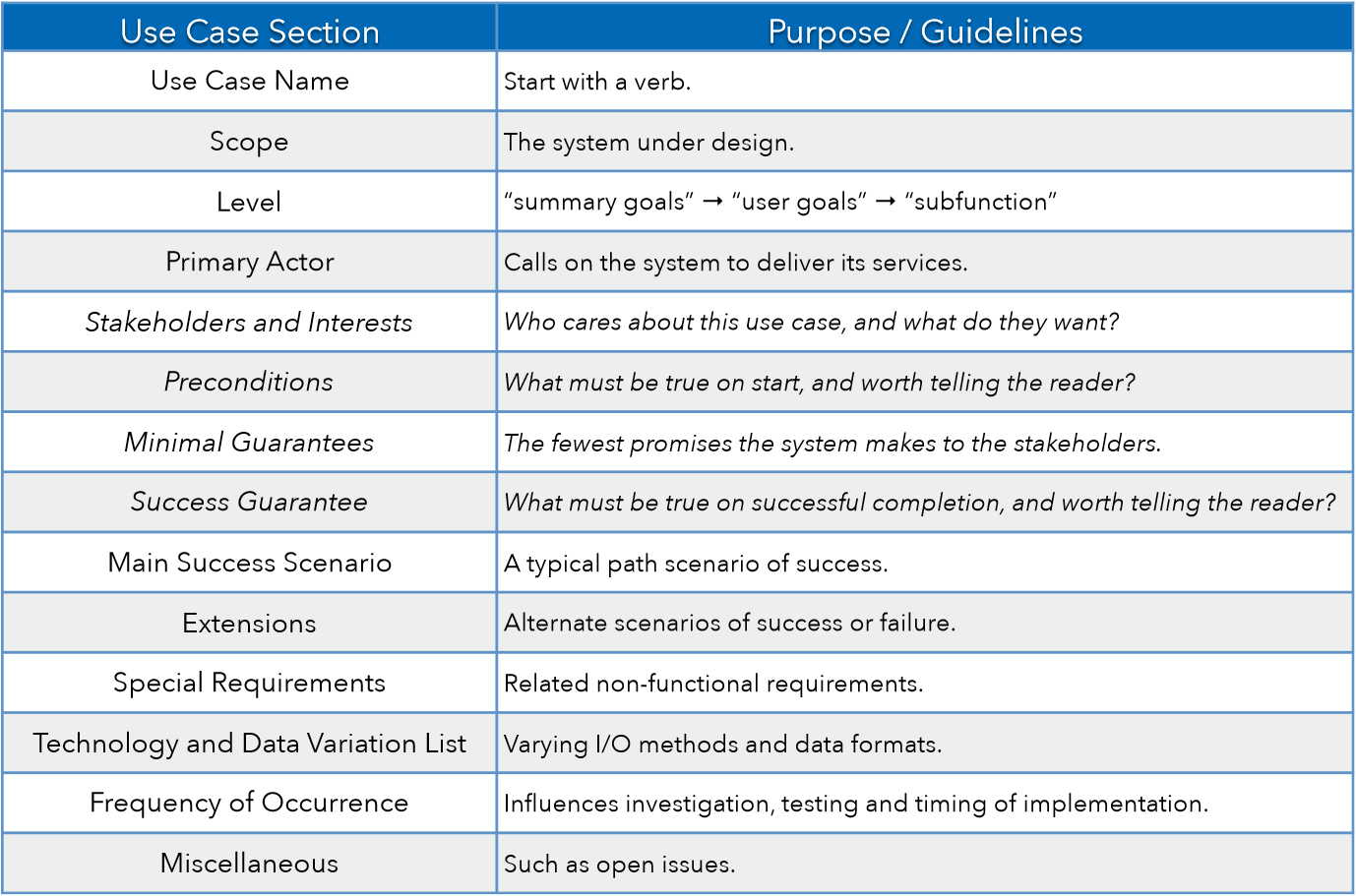
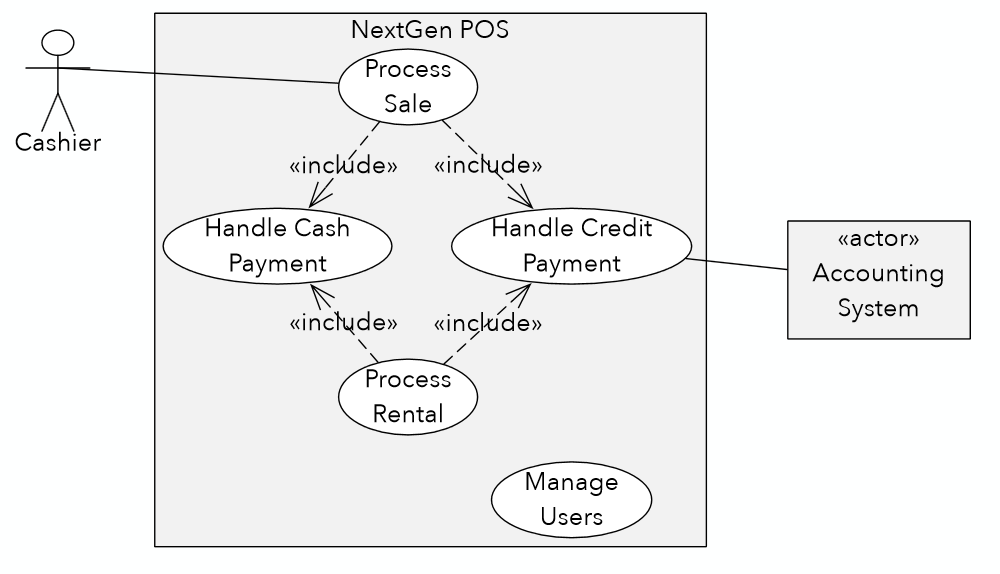
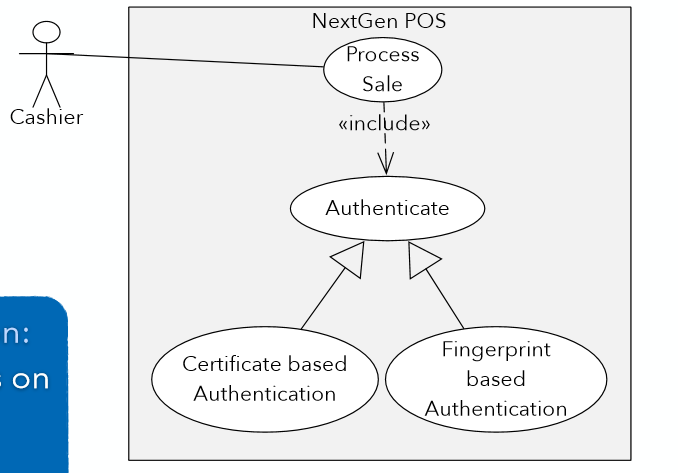
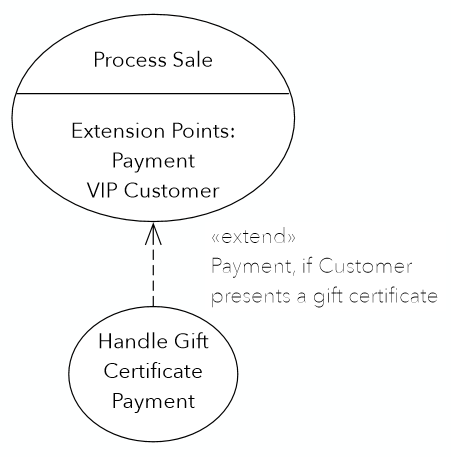
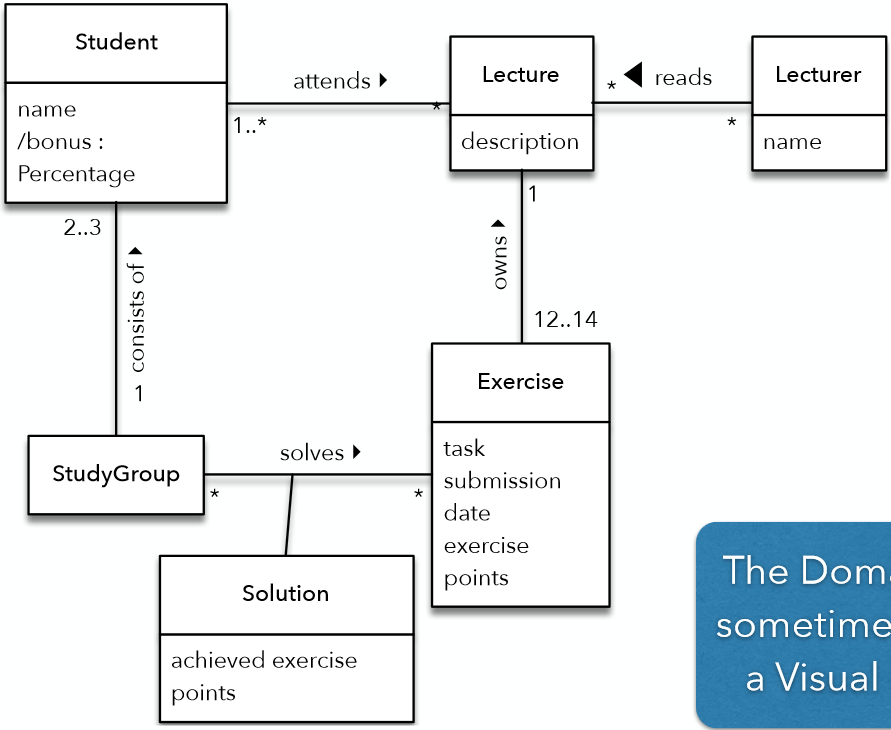
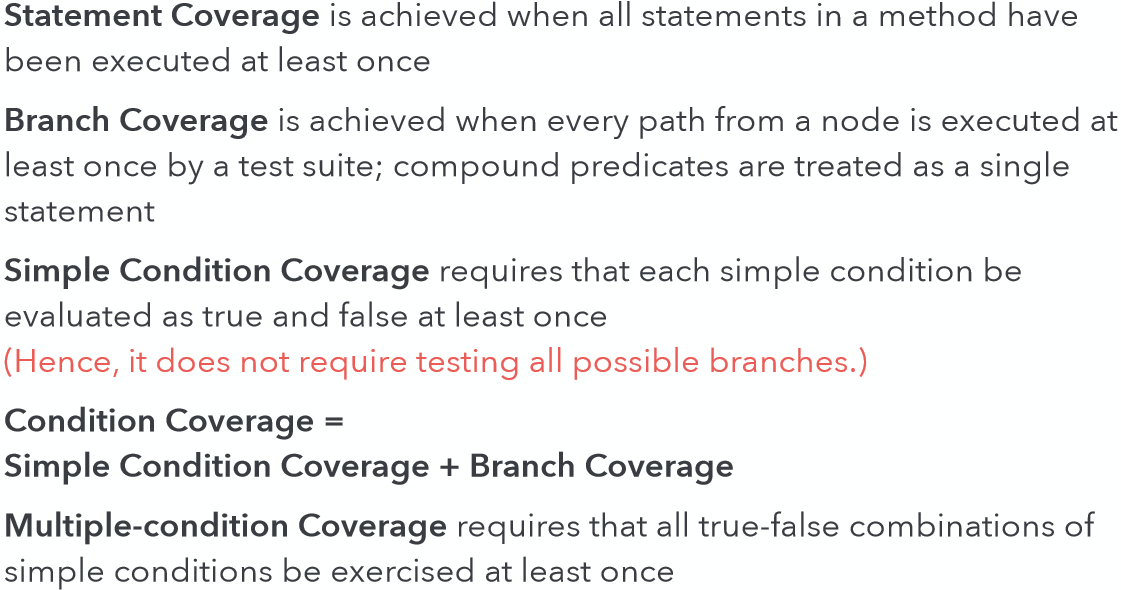
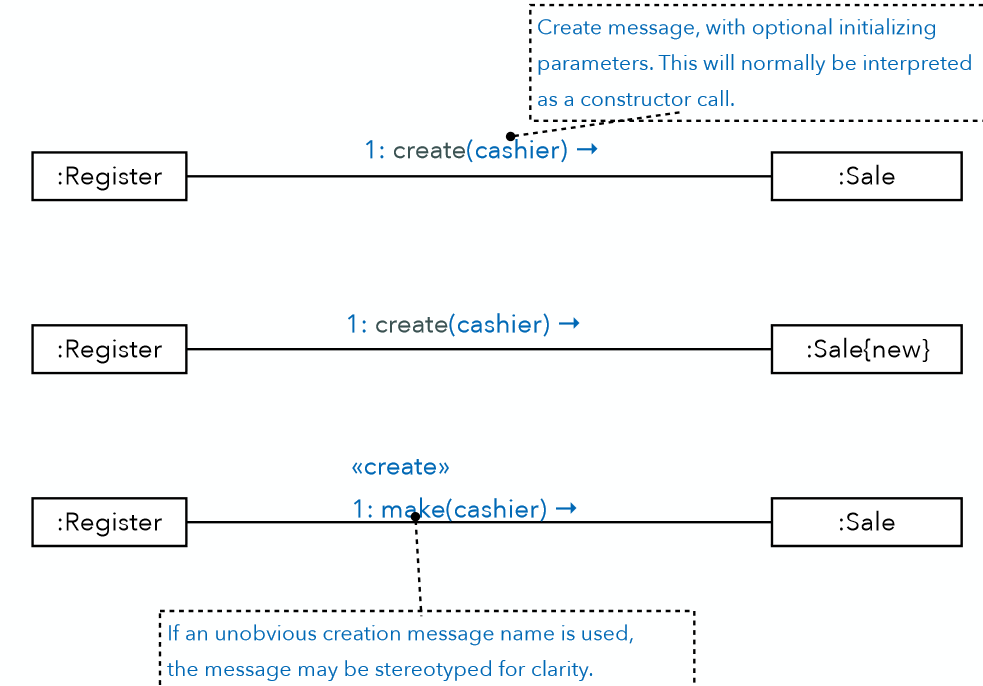
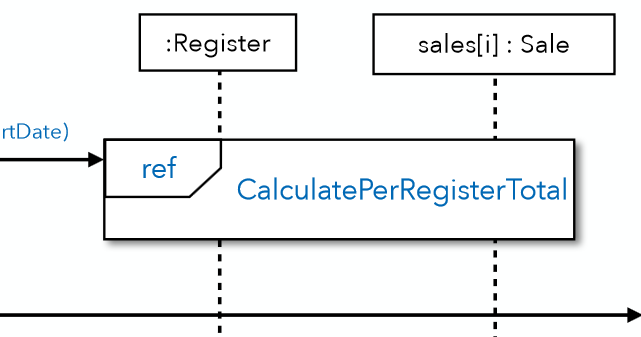
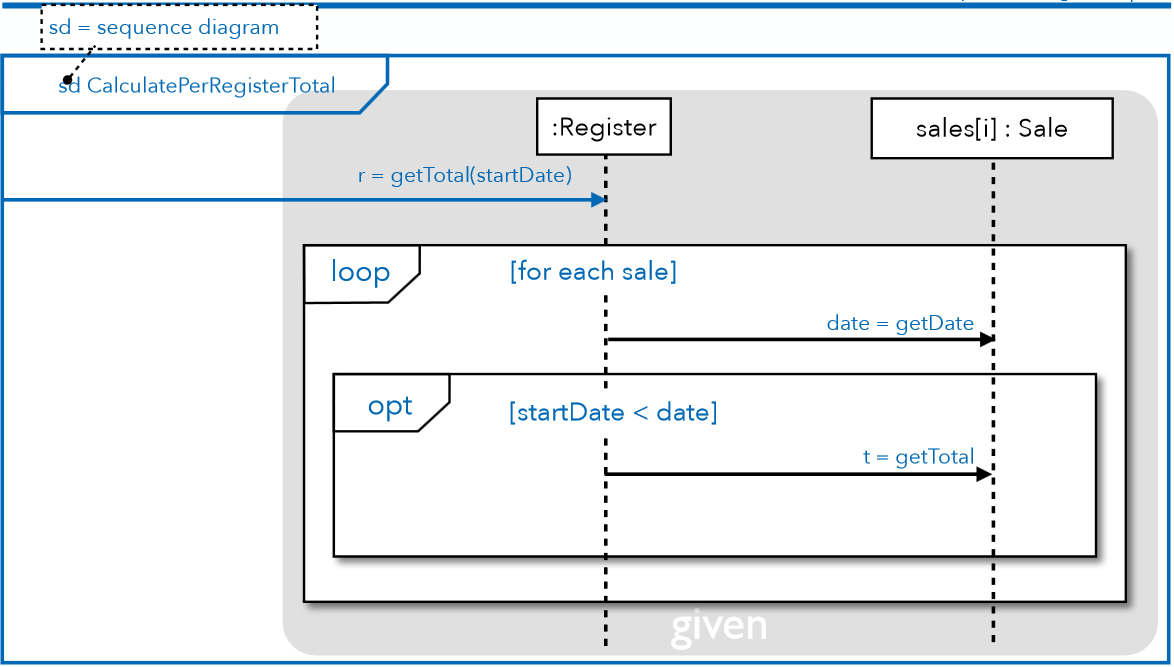
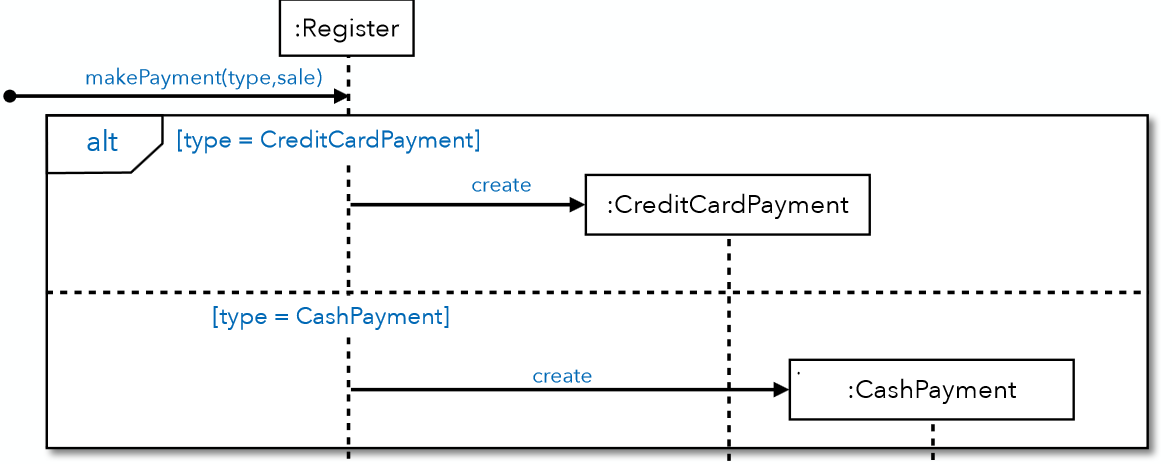
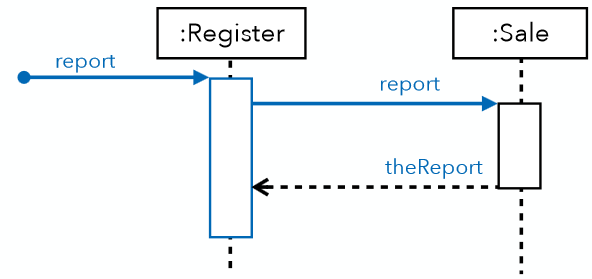
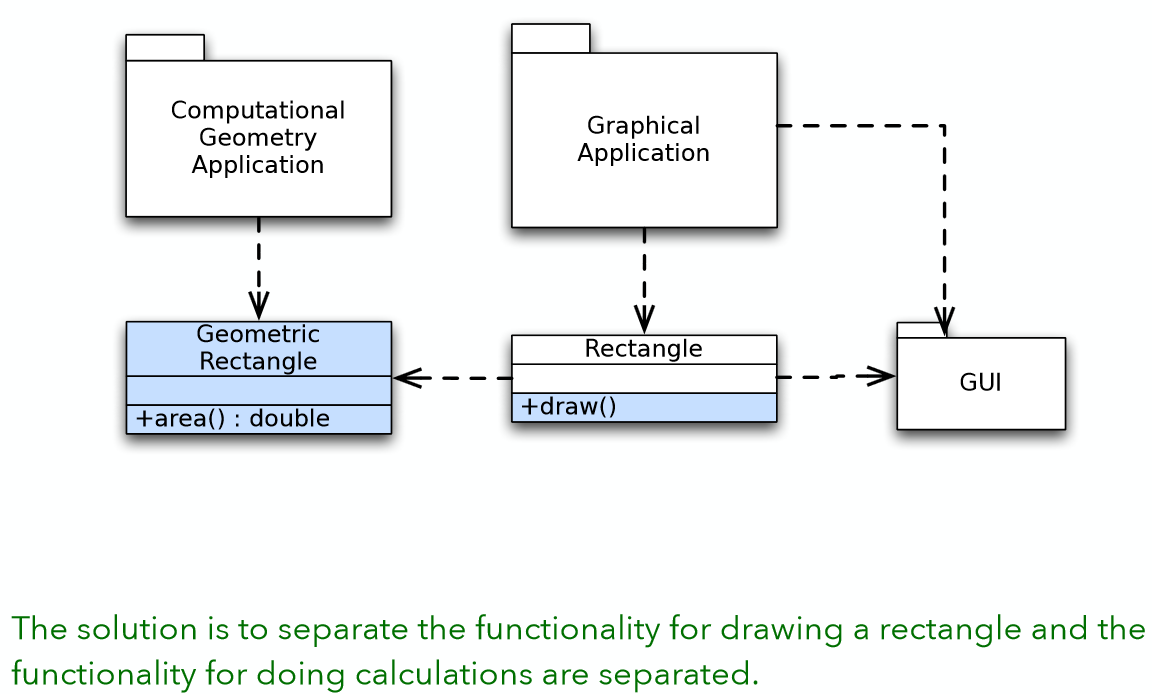
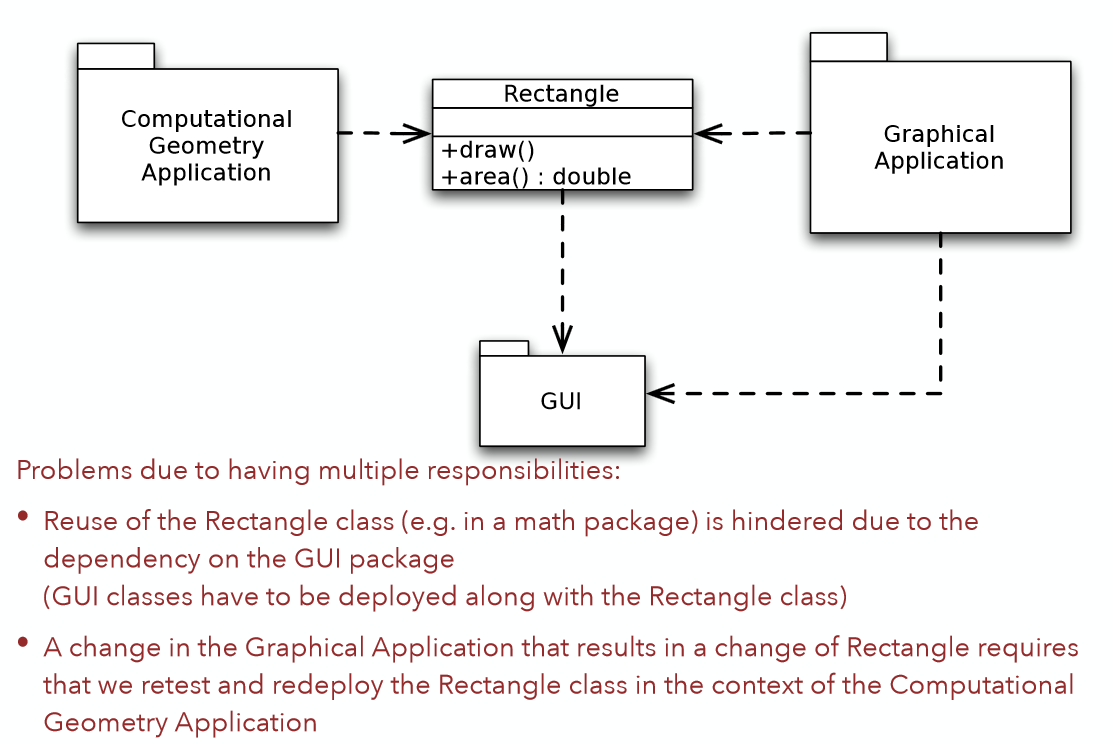
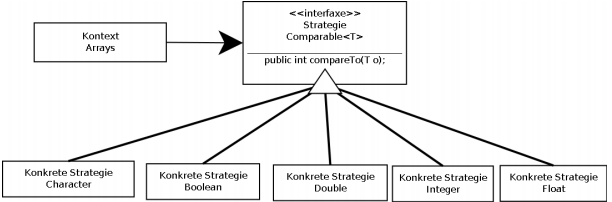
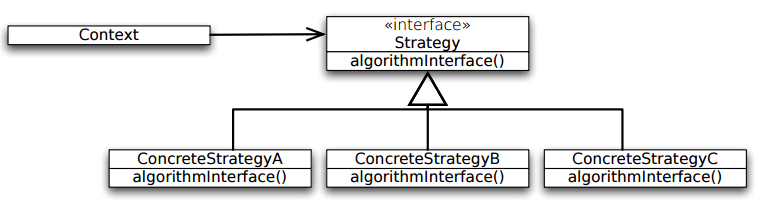
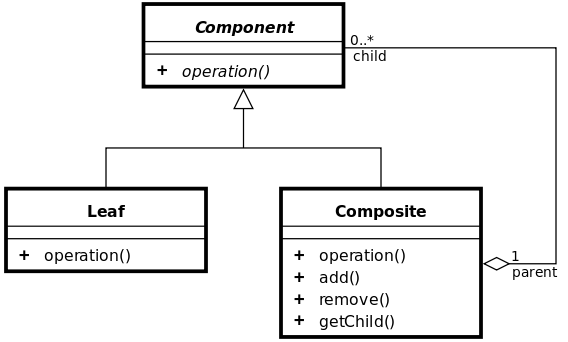
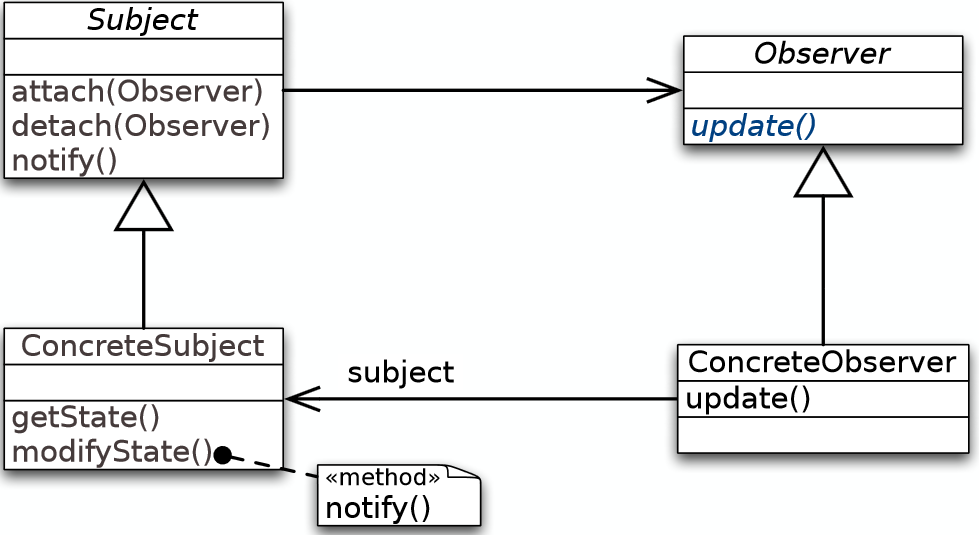
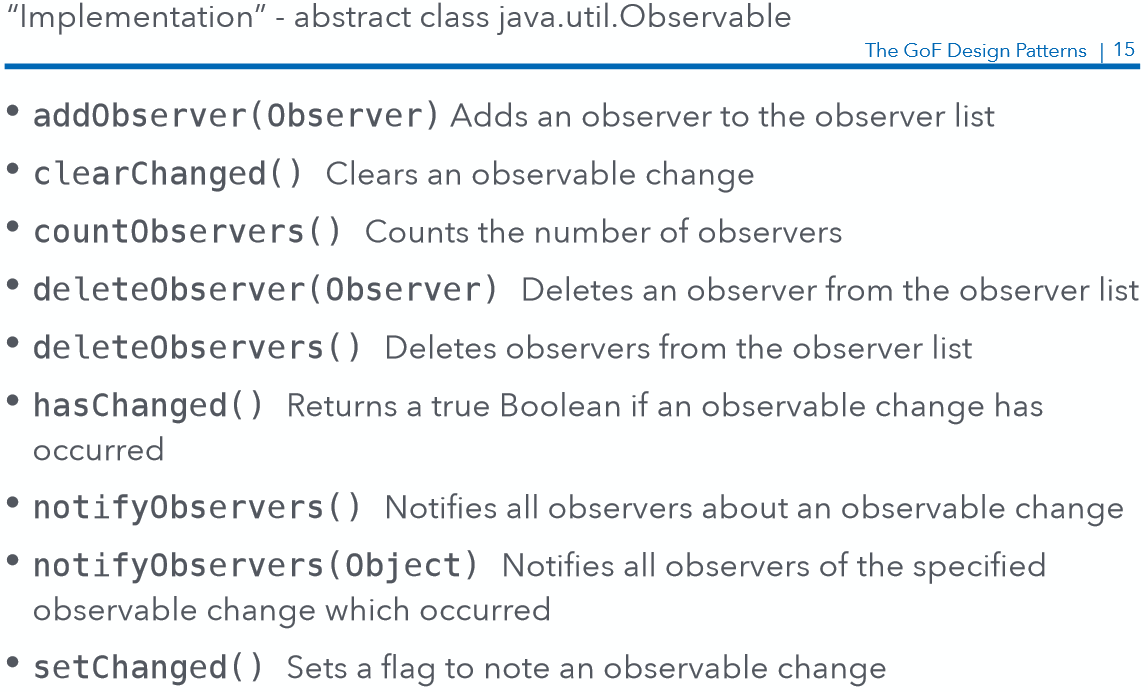
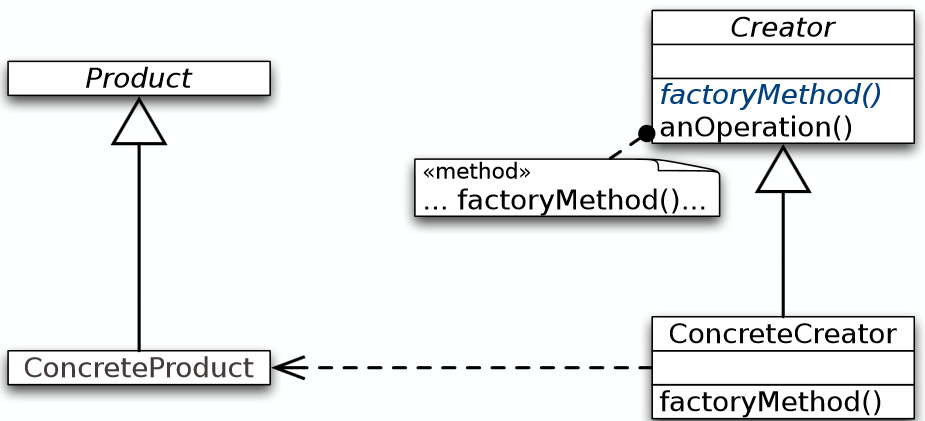
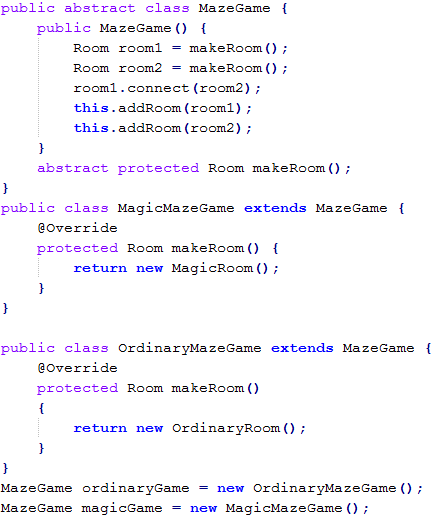
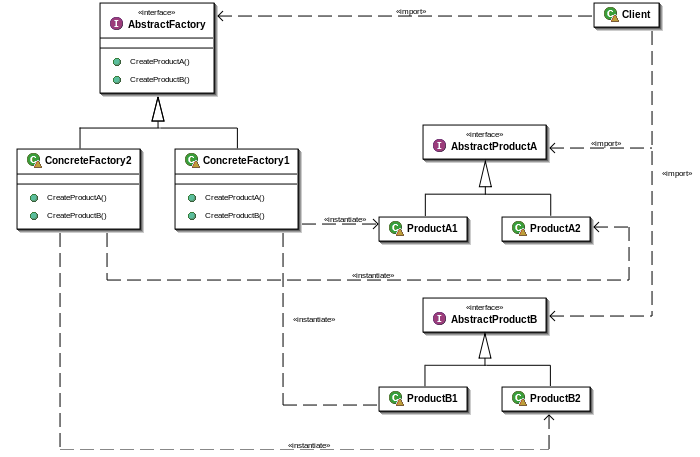
  
Software Prozess Modelle  
Wasserfallmodell:  
  
Agile Development:  
Kunden zufriedenstellen durch häufiges Veröffentlichen funktionierender Software  
Das liefert ein Maß für den bereits vorhandenen Fortschritt  
Verlangt und führt zu gutem Design  
Es kann flexibel auf veränderte Anforderungen eingegangen werden  
Regelmäßige Reflektion im Team  
Geschäftsleute und Entwickler müssen täglich zusammenarbeiten  
Keine Überforderung der Entwickler gegen Ende, eher konstante Entwicklungsgeschwindigkeit  
Beispiele für agile Prozesse: SCRUM (als project management method), Unified Process, Extreme Programming, …  
Software Quality  
External software qualitfy factors: Korrektheit (software erfüllt Aufgaben der Spezifikation entsprechend), Robusteit (software reagiert angemessen auf ungewöhnliche Konditionen), Erweiterbarkeit (wie leicht die software auf Änderungen der Spezifikation anzupassen ist), Wiederverwendbarkeit (software elemente haben verschiedene Anwendungen), Kompatibilität (wie leicht sich Elemente der Software miteinander kombinieren lassen), Effizienz (software benötigt wenig hardware), Portabilität (Wie leicht sich die software auf hardware oder software umgebungen transferieren lässt), einfache Verwendbarkeit, Funktionalität (die Möglichkeiten, die die software zur Verfügung stellt, Featurism vermeiden)  
Internal software quality factors: Wartbarkeit (software lässt sich leicht an geänderte Anforderungen anpassen), Effizienz (keine system ressourcen verschwenden), Verwendbarkeit (software muss vom vorgesehenen Benutzer verwendbar sein), Dependability/Verlässlichkeit (Fehlertoleranz, reparierbares System, Robustheit, avalilability/verfügbarkeit(service wird ausgeführt wenn angefordert), reliability/zuverlässigkeit(services werden wie spezifiziert geliefert), safety/Betriebssicherheit(ohne katastrophale Fehler funktionierendes System), security/Systemsicherheit(Schutz gegen Eindringlinge))  
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:  
Definiere eine familie von Algorithmen, verkapsle sie (abstraktion) und mach sie Runtime interchangeable. Der Algorithmus ist unabhängig von den Clients.  
Strategie: FancyHandle (auch …Strategy)  
Konkrete Strategie: ChineseWallStrategy, FloodingStrategy,… (auch …Action)  
Kontext: Map  
Methode: action(Map,Player):int  
Variante: Benötigte Informationen werden als Parameter übergeben. Alternative: Der Kontext übergibt sich selber.  
  
Pros: Algorithmus hängt nicht vom Kontext ab, kann leichter getauscht, verstanden und erweitert werden  
Zu beachten: Clients müssen über die verschiedenen Strategys wissen, wie sie sich unterscheiden und die richtige wählen können.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  
Composite Design Pattern:  
Component: Die Abstraktion aller Komponenten (auch composite)  
Composite: ein component mit children, implementiert alle component methoden mit Verwendung der Kinder

  
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  
Vorteile: Primitive Objekte können leicht komponiert werden. Clients können composites und primitives gleich behandeln. Neue Komponenten können hinzugefügt werden  
Nachteil: Das Design kann zu abstrakt werden. Typsicherheit kann zu Problemen führen, beispielsweise schwer sich einzuschränken, dass ein composite nur bestimmte components hat

Observer Pattern  
  


Notation: Subject, Observer, ConcreteSubject, ConcreteObserver, Methode (update())  
Varianten: Pull Mode: Observer besorgen sich die Informationen (kann schwer sein), Push mode: Informationen werden übergeben  
Probleme: Unerwartete/unkontrollierte updates; update wird an alle observer weitergegeben, auch wenn diese eventuell nicht daran interessiert sind; kein detail was sich geändert hat, informationen können für die observer schwer zu finden sein; keine optionalen parameter die das subject benutzen könnte  
  
  
  
  
  
Factory Method Pattern  
Definiert eine Schnittstelle um ein Objekt zu erstellen, aber lässt Subklassen entscheiden, von welchem Typ.  
Produkt: definiert das Interface der Objekte, die die factory method kreiert.  
Konkretes Produkt: implementiert das Produkt interface  
Createor: deklariert die factory method, welche ein Objekt vom typ Product zurückgibt. Kann eine Standartimplementation der factory method definieren, welche ein standart concrete product zurückgibt.  
ConcreteCreator: Überschreibt die factory method um eine Instanz eines ConcreteProducts zurückzugeben.  
Beispielimplementation:  
  
  
  
Abstract Factory Pattern:  
Wie Factory Method Pattern, nur stellt es verschiedene Factories zu Verfügung.  
Abstract Factory: interface um produkte einer familie zu erstellen  
Abstract Product: deklariert ein interface für concrete products  
Concrete Factory: implementiert die Operationen um concrete products zu kreieren  
Concrete Product: implementiert das produkt, welches von der zugehörigen ConcreteFactory erstellt wurde  
Methoden:CreateProductA(), CreateProductB()  
Client: Erstellt Produkte, indem die ConcreteFactory aufgerufen wird  
  
Beispielimplementation:  
